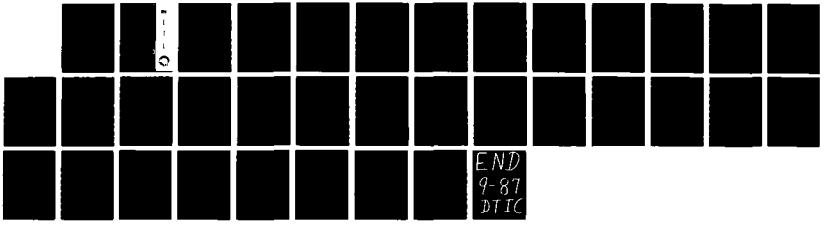
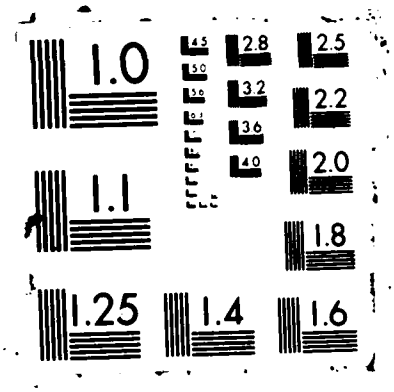
AD-A183 755 PARALLEL ALGORITHMS FOR COMPUTER VISION(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE T POGGIO ET AL.
JAN 87 ETL-0456 DACA76-85-C-0010

UNCLASSIFIED F/G 17/7 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0
11.2
12.5
14.0
16.0
18.0
20.0
22.4
25.0
28.0
32.0
36.0
40.0

ETL-0456

AD-A183 755

Parallel algorithms for computer vision

Tomaso Poggio
James Little

Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

January 1987

Prepared for
U.S. ARMY CORPS OF ENGINEERS
ENGINEER TOPOGRAPHIC LABORATORIES
FORT BELVOIR, VIRGINIA 22060-5546

E T L

U.S. ARMY ENGINEER TOPOGRAPHIC LABORATORIES
ETL

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp Date Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	Approved for public release; distribution is unlimited.
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
	ETL-0456
6a NAME OF PERFORMING ORGANIZATION: Massachusetts Institute of Technology	6b OFFICE SYMBOL (If applicable) / 7a NAME OF MONITORING ORGANIZATION: U.S. Army Engineer Topographic Laboratories
6c ADDRESS (City, State, and ZIP Code): 545 Technology Square, Cambridge, MA 02139	7b ADDRESS (City, State, and ZIP Code): Fort Belvoir, Virginia 22060-5546
8a NAME OF FUNDING / SPONSORING ORGANIZATION / 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DACA76-85-C-0010
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
62301E			

11 TITLE (Include Security Classification)

PARALLEL ALGORITHMS FOR COMPUTER VISION

12 PERSONAL AUTHOR(S)

Tomaso Poggio and James Little

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Contract Report	FROM 8/85 TO 8/86	January 1987	32

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Vision Connection Machine Parallel Algorithms
17	07		
17	08		

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

The general goals of this research effort is to explore the potential applications and performance of fine grained computer architectures for vision. The body of this report gives a brief overview of the results of the research during the first twelve months of effort. A taxonomy of parallel vision algorithms has been refined. Parallel versions of several Class 1 algorithms (for early vision, directly derived from regularization methods) have been designed. This preparatory work allowed the implementation of several of these algorithms on the Connection Machine within a few days when it became available to the researchers. Class 2 and 3 algorithms have also been designed, while at the same time studying some of the associated basic problems in recognition and representation.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL: E. JAMES BOOKS	22b TELEPHONE (Include Area Code): 202-355-3039	22c OFFICE SYMBOL: CEETL-IM-T

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

PARALLEL ALGORITHMS FOR COMPUTER VISION

U.S. Army Topographic Laboratories
Tomaso Poggio and James Little
August 31, 1985—August 31, 1986

First year report

1. TABLE OF CONTENTS

2. SUMMARY

2.1. Vision algorithms for fine grained parallelism

The general goals of our research effort supported by the current contract is to explore the potential applications and performance of fine grained computer architectures for vision. To do this, we will develop efficient parallel algorithms for the Connection Machine system (CM) that will facilitate real-time understanding tasks for autonomous vehicle navigation. Our work is critical to allow an effective comparison of fine-grained models of parallel computation with the more common coarse-grained machines, such as the Warp and the Butterfly.

2.2. The first twelve months

From August 31th 1985 to July 31th our research has concentrated in classifying and designing three different classes of parallel algorithms for vision. In that period of time we also implemented some of the algorithms using a CM system simulator, running on the Symbolics 3600. On July 31th Thinking Machines Corporation delivered to the Artificial Intelligence Laboratory the first Connection Machine system, consisting of 16384 processors. During August several vision algorithms were successfully implemented on the CM, achieving speed-ups of several hundred times relative to the Symbolics 3600.

The body of the report gives a brief overview of the results of our research during the first twelve months of funding. Details can be found in the enclosed publications. Some of the work is reported in publications now in preparation. We enclose a more formal description (by J. Little) of our evaluation of specific vision algorithms on the CM.

3. Achievements in the first year

As planned in our original proposal we have refined our taxonomy of parallel visual algorithms. More importantly, we have designed parallel versions of several Class 1 algorithms (for early vision, directly derived from regularization methods). When in August the Connection Machine system became available, the preparatory work allowed us to implement within a few days several of these algorithms. During the year we have also designed Class 2 and 3 algorithms, while at the same time studying some of the associated basic problems in recognition and representation. We list here briefly the main achievements: details can be found in the enclosed technical reports.

3.1. Simulator specifications

Specifications have been formulated for a Connection Machine simulator that has been later implemented at GE (and TMC). We have been using the TMC simulator running on the Symbolics 3600 for the last several months. The experience with it has been very positive: programs debugged on the simulator ran without problems on the machine. The simulator is a very useful tool for leveraging the use of a high-performance, single-user machine such as the Connection machine system. Program development can be done almost completely without wasting precious resources.

3.2. Image manipulation and display tool

We have developed a powerful image display and manipulation tool as a Lisp Machine system. It has been extensively used for evaluating vision algorithms on both the Symbolics machines the Connection Machine system. It has become a critical component for our vision work.

3.3. Vision utilities

We have extended a collection of Lisp utilities for doing interactive vision work. These utilities also serve as powerful primitives for constructing larger vision programs. We have begun transporting the system from the Symbolics machines to the Connection Machine system.

3.4. Taxonomy of vision algorithms

3.4.1. Three classes of vision algorithms

Class I algorithms correspond to early vision problems that can be formulated in terms of standard regularization principles, such as edge detection, stereo, motion, surface interpolation and shape from shading. The basic members of class I are convolution and multigrid algorithms that map into parallel, fine-grained architectures with local connections. *Class II* algorithms correspond to general, intermediate vision problems such as the fusion of information from different sources and the detection and use of discontinuities. Algorithms in this class are based on non-standard regularization principles and are more tightly interfaced with high-level, symbolic information. They require fine-grained architectures with local and non-local communication capabilities. *Class III* algorithms correspond to higher level vision problems such as shape representation, object recognition and analysis of spatial relations. This heterogeneous class of algorithms cannot presently be described in terms of regularization analysis. They map into parallel architectures with the capability of processing pointers and symbolic, non-retinotopic data structures.

3.4.2. Classification for early vision

We have provided a classification of vision algorithms derived from standard regularization and including most early vision problems. Early vision is the set of visual modules that perform the first steps of processing by extracting from the images a map of the physical surfaces around the viewer. High-level vision can be identified with the "later" problems of object recognition and shape representation. From this point of view early vision is the *inverse problem* of optics and computer graphics. The natural way to approach this problem is to exploit *a priori* knowledge about our 3-D world to remove the ambiguities of the inverse mapping. One of the major achievements of computer vision work in the last decade is the demonstration that *generic* natural constraints, that is, general assumptions about the physical world that are correct in almost all situations, are sufficient to solve the problems of early vision. Very specific, high-level, domain-dependent knowledge is not needed.

Two main themes are therefore intertwined at the heart of the main achievement of early vision research. They are: a) the identification and characterization of generic constraints for each problem and b) their use in

an algorithm to solve the problem. A general method for translating a certain class of common constraints into parallel algorithms is provided by regularization theory. This new unifying theoretical framework is based on the recognition that most early vision problems (see Table 1) are mathematically ill-posed problems (in the sense of Hadamard). A problem is well-posed when its solution exists, is unique, and depends continuously on the initial data. Ill-posed problems fail to satisfy one or more of these criteria. In vision, edge detection – the detection and localization of sharp intensity changes – is ill-posed, when considered as a problem of numerical differentiation, because the result does not depend continuously on the data. Another example is the reconstruction of 3-D surfaces from sparse data points which is ill-posed for a different reason: the data alone, without further constraints, allow an infinite number of solutions, so that uniqueness is not guaranteed without further assumptions.

The main idea in mathematics for "solving" ill-posed problems – that is, for restoring well-posedness – is to restrict the space of admissible solutions by introducing suitable *a priori* knowledge. In vision, this is identical to exploiting the natural constraints described earlier. Mathematicians have developed several formal techniques for achieving this goal that go under the name of regularization theory. It is therefore not too surprising to find that some of the algorithms proposed in the past for solving specific early vision problems are precisely regularization algorithms of one type or another. In particular, Table 1 shows the solutions of several vision problems provided by standard regularization (Tikhonov, 1977; Poggio et al., 1985). Some of the solutions were known already (see especially the work by Horn, 1986, Grimson, 1981 and Hildreth, 1984), others were generated by the recognition of this common framework. Table 1 represents our present classification of regularization algorithms.

In standard regularization the solution is found as the function that minimizes a certain functional. This functional can be regarded as an "energy" or a "cost" that measures how close the solution is to the data and how well it respects the a priori knowledge about its properties. From the point of view of implementation the following two classes of algorithms are suggested by regularization:

- Steepest descent methods that can always be applied.
- Convolution schemes can be used when the data are given on a regular lattice and the direct operator is space invariant.

3.5. Design and implementation of early vision algorithms (Class 1 and 2)

In addition to the planned design and implementation of several early vision algorithms, we have been able to implement already, ahead of schedule, the following ones on the Connection Machine system (some of them in collaboration with Thinking Machines Corporation):

- Parallel convolution
- Zero-crossing detection
- Stereo-matching
- Surface reconstruction
- Canny's edge detector
- Measurement of optical flow

In all these cases, we obtained speed improvements from hundred- to thousand-fold with respect to the Symbolics 3600.

3.6. Design of other Class 2 algorithms

We have also analyzed and designed but not yet implemented (as of August 31th) a number of other parallel vision algorithms:

- Concurrent multigrid methods
- Surface reconstruction combining depth and slope measurements in parallel
- Parallel detection of texture boundaries
- Stereo matching of image intensities
- Stereo matching and resolution of contours

3.6.1. Algorithms based on stochastic regularization

Other 'regularization" methods exist. Geman and Geman (1984), for instance, have introduced coupled Markov Random Fields models which can be regarded as a probabilistic regularization method. The attraction of this formulation is that it can be used to exploit a much larger class of natural constraints than standard regularization. It can take into account, for instance, constraints on discontinuities such as their continuity. Depth boundaries in a

depth map, for instance, are usually continuous, connected, nonintersecting lines. Furthermore, coupled Markov Random Field models seem a powerful, albeit expensive, method for approaching a most critical problem in early vision: how to integrate several modules of early vision, such as stereo and motion and shading, to obtain a consistent and robust map of 3-D surfaces around the viewer. We have implemented on the Connection Machine simulator a MRF based system to integrate information from several visual sources. The system has been used to perform surface reconstruction in the presence of discontinuities. A preliminary Connection Machine implementation exists since August: it allows, for the first time, extensive simulations of this class of algorithms which is computationally very expensive.

3.7. Design of high-level (Class 3) algorithms

Computation of spatial relations and visual recognition are two "high-level" problems in vision for which no well defined class of algorithms exists. We have done basic research in these problems and designed related parallel algorithms. In particular:

- We have developed parallel processes, to extract extended spatial elements from an image.
- We have developed a parallel version of the object recognition algorithm of Grimson and Lozano-Perez.
- We have extended the previous algorithm by fully exploiting the connection Machine router.
- We have developed two additional CM implementations of the Grimson-Lozano-Perez recognition algorithm.
- We have developed a Connection Machine-based scheme for matching silhouettes of natural objects to models of the objects

4. RELEVANT TECHNICAL REPORTS AND ABSTRACTS

Drumheller, M. and Poggio. T "On Parallel Stereo", Proc. of IEEE Conf. on Robotics and Automation, 1986.

Harris, J. "The Coupled Depth/Slope Approach to Surface Reconstruction", A.I.T.R. 908, May, 1986.

Harris, J. and Flynn. A "Object Recognition Using the Connection Machine's Router", IEEE ICVPR, Miami, 1986.

Koch, C., Marroquin, J. and Yuille, A. "Analog "neural" networks in early vision", P.N.A.S. 1986

Lim, W. "Using the Connection Machine for Matching and Indexing" , to be presented at IU Workshop, LA, 1987

Little, J.J. "GROK Doc: An Image Display Tool", A.I. Working Paper 287, April 1986.

Marroquin, J.L. "Probabilistic Solution of Inverse Problems", T.R. 860, September, 1985.

Negahdaripour, S., "Direct Passive Navigation: Analytical Solution for Planes:, A.I. Memo 863, 1985

Poggio, T. "Integrating vision modules with coupled MRF's", A.I. Working Paper 285, 1985

Terzopoulos, D. "Concurrent Multilevel Relaxation", Proc. DARPA Image Understanding Workshop, December, 1985

Voorhees, Harry "Vision Utilities" A.I. Working paper 281, 1985

Voorhees, Harry and Poggio, T. "Detecting Texture Boundaries in Natural Images", to be presented at IU Workshop, LA, 1987

Table 1: Classification of regularization principles in early vision

Consider the direct problem of finding y, given z and the mapping A:

$$Az = y$$

The inverse and usually ill-posed problem is to find z from y. Standard regularization suggests transferring Equation (1) into a variational problem by writing a cost functional consisting of two terms. The first term measures the distance between the data and the desired solution z; the second terms measure the cost associated with a functional of the solution $\|Pz\|$ that embeds the *a priori* information on z. In summary, the problem is reduced to finding z that minimizes

$$\|Az - y\|^2 + \lambda\|Pz\|$$

where λ, the regularization parameter, controls the compromise between the degree of regularization of the solution and its closeness to the data. Mathematical results characterize various properties of this method such as uniqueness and behavior of the solution. The table shows some of the early vision problems that have been "solved" in terms of standard regularization.

The first five are standard, quadratic regularization principles (see Poggio et al., 1985). In edge detection the data on image intensity ($i = i(x)$) (for simplicity in one dimension) are given on a discrete lattice: the operator S is the "sampling" operator on the continuous intensity distribution f to be recovered. Regularization in this case is equivalent to the following *convolution* algorithms: convolve the image with the appropriate derivative of a 2-D spline filter, which is very close to a gaussian function. Gaussian-like filters at various resolutions (corresponding to different λ values) have been extensively used in computer vision (Marr, 1982). A similar functional may be used to approximate time-varying imagery. The spatio-temporal intensity to be recovered from the data $i(x, y, t)$ is $f(x, y, t)$; the stabilizer imposes the constraint of constant velocity V in the image plane.

In area-based optical flow (Horn, 1986), i is the image intensity; u and v are the two components of the optical flow field. In contour-based optical flow v is the "velocity" vector to be retrieved, v^n is its known normal component along the contour (Hildreth, 1984). In surface reconstruction (Grimson, 1981) the surface $f(x, y)$ is computed from sparse depth data $d(x, y)$. In the case of color the brightness is measured on each of three appropriate color coordinates

I^ν ($\nu = 1, 2, 3$). The solution vector z contains the illumination and the albedo components separately. Minimization of an appropriate stabilizer enforces the constraint of spatially smooth illumination and either constant or sharply varying albedo. For shape from shading (Ikeuchi and Horn, 1981) and stereo, we show two nonquadratic regularization functionals. R is the reflectance map, f and g are related to the components of the surface gradient. The regularization of the disparity field d involves convolution with the Laplacian of a Gaussian of the left (L) and the right (R) images and a Tikhonov stabilizer corresponding to the disparity gradient. Steepest descent algorithms can be used in all the cases (with the exception of stereo).

Table 1

Classification of Regularization Algorithms in Early Vision

Problem	Regularization Principle
Edge detection	$\int \left[(Sf - i)^2 + \lambda (f_{xx})^2 \right] dx$
Optical flow (area based)	$\int \left[(i_x u + i_y v + i_t)^2 + \lambda (u_x^2 + u_y^2 + v_x^2 + v_y^2) \right] dx\, dy$
Optical Flow (contour based)	$\int \left[(\mathbf{V} \cdot \mathbf{N} - v^N)^2 + \lambda (\frac{\partial}{\partial s} \mathbf{V})^2 \right] ds$
Surface reconstruction	$\int \left[(S \cdot f - d)^2 + \lambda (f_{xx}^2 + 2 f_{xy}^2 + f_{yy}^2)^2 \right] dx\, dy$
Spatiotemporal approximation	$\int \left[(\mathbf{S} f - i)^2 + \lambda (\nabla f \cdot \mathbf{V} + f_t)^2 \right] dx\, dy\, dt$
Color	$\left\| I^\nu - Az \right\|^2 + \lambda \left\| Pz \right\|^2$
Shape from shading	$\int \left[(E - R(f,g))^2 + \lambda (f_x{}^2 + f_y{}^2 + g_x{}^2 + g_y{}^2) \right] dx\, dy$
Stereo	$\int \left\{ \left[\nabla^2 G * \Big(L(x,y) - R(x + d(x,y), y) \Big) \right]^2 + \lambda (\nabla d)^2 \right\} dx\, dy$

PARALLEL ALGORITHMS FOR COMPUTER VISION

James J. Little
Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

We present solutions to a set of benchmark problems for Image Understanding 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 utilized in Image Understanding. The descriptions of solutions on the Connection Machine embody several general methods of using the machine to filter images, to determine connectivity among image elements, to determine geometry of image elements and, finally, to compute graph properties, such as matchings and shortest paths. 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 algorithms and good ideas about using the Connection Machine.

Vision System

The parallel computing environment at the MIT AI Lab consists of a Connection Machine [4] with 16K processsors with a Symbolics 3640 Lisp Machine as host.

The Connection Machine

The Connection Machine (CM) [4] is a parallel computing machine having 64K processors, operating under a single instruction stream broadcast to all processors (figure 1). Each of the processors is a simple 1-bit processor 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 256×256 grid network (the NEWS network, so-called because of 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. Figure 2 shows a 4-dimensional hypercube; each processor is connected by 4 wires to other processors. Each processor in the CM is identified by a unique integer in the range $0 \ldots 65536$, its hypercube address, imposing a linear order on the processors. This address denotes the destination of messages handled by the router. Messages pass along the edges of the hypercube from source processors to destination processors. To allow the machine to handle data with more than 64K elements, the Connection Machine supports the concept of *virtual processors* where a single physical processor can operate as multiple virtual processors by serializing operations in time. The number of virtual processors assigned to a physical processor is denoted

Figure 1: Block Diagram of the Connection Machine (from [4])

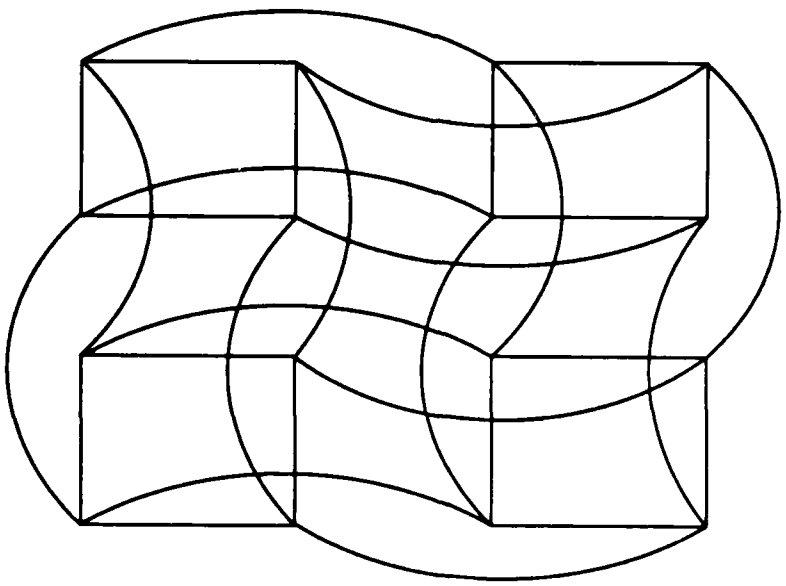


Figure 2: 4-dimensional Hypercube

by the *virtual processor ratio* (VP ratio), which is always ≥ 1. When the VP ratio is greater than 1, the CM is necessarily slowed down by that factor.

Many of the IU benchmark problems must be solved by a combination of both communication modes on the CM. The CM implementation of algorithms can take advantage of the underlying architecture of the machine in novel ways. There are several common, elementary operations which recur throughout this discussion of parallel algorithms. Sorting, for example, of all 8-bit pixel values 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.

Another primitive, global operation is the *scan* operation, which uses the hypercube connections underlying the router to distribute values among the processors of the CM. *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 $[0, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$. Binary associative operations include min, max, and plus. A *max-scan* operation acts on a field in the CM memory, storing 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. Recently, the *scan* operations have be augmented to use the NEWS addressing scheme, so that summing, taking maxima, and copying can operate in the grid coordinate system. In addition, *segment bits* permit the scan operations to divide the processors into segments, whose limits are marked by processors whose segments bits are set. *copy-scan* copies values to processors, in the chosen address mode. The scan operations implement the abstract operation known as *parallel prefix* [10]. 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 an example of *plus-scan* taken from [10]; on the up sweep, each node in the tree executes $\oplus$ on the sum values of its two children nodes, and stores the value of the sum in the left subtree. These are indicated by the two values at each node. On the down sweep, each node passes to its left child the value from its parent, and passes to its right child $\oplus$ of its parent and the value of the left child kept from the up sweep.

An important primitive operation is *doubling* [5], which facilitates finding the extremum of a number contained in each processor in a ring. Using message-passing on the router, *doubling* can propagate the extreme value to 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 number is chosen to be the cube-address (a unique integer identifier) of the processor. At termination, each processor connected 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

Figure 3: Parallel Scan on a Tree (from [10])

Step	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	0
	4	1	5	2	11	12	19	3
1	2	3	4	5	6	7	0	1
	4	5	5	11	12	19	19	19
2	4	5	6	7	0	1	2	3
	19	19	12	19	19	19	19	19
3	0	1	2	3	4	5	6	7
	19	19	19	19	19	19	19	19

Figure 4: Distance Doubling, upper entry is the address, the lower is the 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 all values within 2^{i-1} processors away. In the example, he maximum value has been propagated to all 8 processors in $\log 8 = 3$ steps.

Rules of the Game

For the purposes of these benchmarks, 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 CM 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 benchmarks specify 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 benchmarks 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.

Connection Machine programs utilize Lisp syntax, in a language called *Lisp [11]. 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 CM software environment has been that programming the CM is a relatively easy progression from using Lisp, and that uses can, within a week, begin programming complex programs on the CM. The improvements in execution time from implementation to estimated times reflect expected improvements in micro-code for certain operations on the CM, as well as re-coding of the algorithms in a low-level language for the CM (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.

IU BENCHMARKS

1. Edge detection

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

1. **Convolve the image with an 11 x 11 sampled "Laplacian" operator [1]. (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 [2], Section 11.2.2.)**

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

Convolution with Laplacian

The 11x11 sample "Laplacian" actually corresponds to filtering with a Gaussian where σ is 1.4,([1], but see [8], 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.

Detecting Zero-Crossings

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

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 CM 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 CM, the points on the curves should be numbered in order to create a stream of connected points. The curve-labelling 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 ($200\mu s$). We use sorting to get the points on the curves into a stream order for output. For the i^{th} point in the I^{th} curve, we construct the number $MaxI + i$ to encode the point's position on the curve and its membership in the I^{th} curve. Each point is *ranked* by this value. Points of the I^{th} end up ranked in order of their position on the curve. This takes $O(\log Max + \log N + 1)$ ms.
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	$2.5(\log Max + \log N + 1)ms$
Send	1 ms

For typical values 512 x 512 image, Size = 512, Max = 512, N = 256, so:
$\log Max = 9$
$\log N = 8$
$\log Size = 9$

Propagate label and enumerate points	$40ms$
Enumerate curves	$350\mu s$
Rank pixels	$45ms$
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 $40ms$. The remaining time, to prepare for output, is $46ms$. In total, approximately $100ms$ is need to perform Border Following.

The first two steps, Convolution and Detecting Zero Crossings, add negligible time to this process, so approximately 100*ms* should suffice.

Edge Detection		
Sub-task	Implemented	Estimated
Convolution	3*ms*	2*ms*
Find Zero-Crossings	0.5*ms*	0.5*ms*
Propagate label	40*ms*	40*ms*
Enumerate curves	350μs	350μs
Rank and send pixels	91*ms*	46*ms*
Total - without Output	44*ms*	43*ms*
Total - with Output	135*ms*	99*ms*

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

2. Connected component labeling

1. Here the input is a 1-bit digital image of size 512 x 512 pixels. The output is a 512 x 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 [2], Section 11.3.1.)

A fast practical alogrithm for labeling connected components in 2-D image arrays using the Connection Machine has been developed by Willie Lim [5]. 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 [5] 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 is either on the array boundary or has 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 x 512 image.

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

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

3. Hough transform

The input is a 1-bit digital image of size 512 x 512. Assume that the origin (0,0) image is at the lower left-hand corner of the image, with the x-axis along the bottom row. The output is a 180 x 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 [2], 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. A ***boundary processor*** is one which occurs at the beginning of one of the normal lines. Then a special ***plus-scan*** operation can accumulate the numbers for the histogram in the boundary processors. 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 should require about 4 ms (VP 4:1), and only $3ms$ for VP 1:1. Then entire Hough Transform should then be computed in approximately $720ms$. This estimate is, of course, based on a 512x512 image. Then, the CM is using a 4:1 VP ratio, resulting in a reduction in processing speed by a factor of 4 for most operations. For a 256x256 image, the time for the histogram would be 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 is feasible in which 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 point in the machine can then generate an integer identifying its Hough Transform value, using no more than 17 bits (512 x 180). These values can be sorted in $25ms$, ***plus-scanned***, and then ***sent*** to the table. The total should be no more than $30ms$.

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

4. Geometrical constructions

The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in

Hough Transform		
Method	Implemented	Estimated
Full 180 steps (512x512)	—	720*ms*
Full 180 steps (256x256)	—	540*ms*
From edge elements (512x512)	—	30*ms*

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 [3], 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.

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 should take $O(H)$ steps, where H is the cardinality of the set of rays. Each step involves following a pointer in the CM, less than $1ms$.

An alternative method for the convex hull calculation begins from Graham's sequential algorithm [3,p.103], and does not rely on the underlying grid. Each point is assigned an angle by constructing a vector from a point interior to all points. This point can be determined in 4 extremum operations on the CM, finding the x and y extrema of the points. Then the points are sorted in $20ms$, by this angle. Graham's algorithm then recursively constructs convex wedges from pairs of neighboring points and the center point. Initially, these are triangles. The outer curves of these wedges can be merged into new convex wedges in $O(\log N)$ steps [7]. 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. Also, the analysis here considers worst case.

A simple *Lisp implementation of the Jarvis march algorithm [3] was constructed. In each iteration, each point computes its slope from a reference point, which is on the hull or outside (at first). Computing the slope means two subtractions and one division. This, plus finding the

global minimum slope, and finding the point with that slope, constitutes each step. The 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 would be $90ms$ in the PARIS version. This method would require 3 seconds if all 1000 points were on the hull, but it is marginally faster in the expected case.

Voronoi Diagrams

Aggarwal et al. [6] 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 CM has the NEWS network, a set of mesh connections among the processors, a brush-fire method can be easily implemented on the CM. The points have coordinates in the range [0,1000], so the CM 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 512x512 is a reasonable range. A VP ratio of 4:1 results from a 512x512 grid. Using the full Euclidean-distance metric, and propagating the index of the processor containing the point, the Voronoi region around a point can be labelled in D steps, where D is the diameter of the largest Voronoi region. Constructing the Delaunay triangulation, the dual of the graph of the Voronoi diagram, can be done 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 can be accomplished in 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 should be improved to no more than $10ms$ per step. With a VP ratio of 16:1, a propagation step should take $160ms$. 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 should take less than 2 seconds (16:1), which reduces to $500ms$ for 512x512. The additional work to identify Voronoi vertices and send the information about connections will take less than $10ms$.

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) should then 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)	—	50*ms*
Convex Hull (Graham scan)	—	200*ms*
Convex Hull (Jarvis march)	150*ms*	100*ms*
Voronoi Diagram (1024x1024)	4 s	2 s
Voronoi Diagram (512x512)	1 s	500*ms*
Minimum Spanning Tree (from VD)	—	150*ms*

Note: The times quoted here are based on a configuration of a 64K CM. For the two Voronoi Diagram methods, the Virtual Processor ratios are 16:1 and 4:1, and the data points are quantized to 1024 x 1024 or 512 x 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.

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).

Each triangle can, in a preprocessing step, generate its plane equation in the form $Ax+By+C = 0$. A point can then be tested for visibility by evaluating that form for its (x, y) coordinates. When the point is behind the plane containing the triangle, each triangle can test whether it encloses the projection of the point onto the plane. All points can 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.

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. Segments bits are inserted at the termination of each set of triangles. *Scanning* a 96 bit (3 x 32) field takes $3ms$. All triangles test the active points in parallel, in $12ms$. 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 CM, by mapping a projection plane, anywhere in the visible region, orthogonal to a line of sight from the origin, onto the 256x256 grid of the CM. 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 method. This step should require no more than $25ms$. Finally, the projected vertices of triangles are distributed across the rows of the grid by a *grid-scan* operation, stopping at the

pixels containing projected edges of the triangles. The *scan* operations allow stop information to be included in the process. 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, should require 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 should be the worst case for this method; most practical examples should have maximum coverings approximately 10 or 20.

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

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

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. There is no known method by which any method for solving this problem, even with a number of processors polynomial in N, the size of the problem, can improve upon worst-case behavior depending on the exponential number of possible matchings. The graphs in this particular problem are uniform in degree, so that all nodes in H can match with all nodes in G, before any expansion of solution nodes occurs. Most heuristics for this problem rely on non-uniformity of the degrees of nodes 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. Statistical arguments from the theory of random graphs, concerning threshold functions, indicate that, when matching graphs of the specified sizes and degrees, there is a matching with probability one, or, to put it another way, there are a large number of candidate matches.

We will outline a method to distribute the matching process among the processors of the CM. A similar solution for objection recognition is described in [9]. The method will be specialized to this particular size of graph, but should be generic enough to be generalized for any sizes. We utilize dynamic allocation of processors to matchings. A partial matching is contained in a processor. At each step in the graph matching algorithm a matching (processor) will acquire the information necessary to determine all legal successors. It will then find processors to continue with the new matchings; it is then returned to the pool of free processors. The information concerning the graphs can be stored in several ways in the memory of the Connection Machine. Since $|G|$ is 100, 7 bits is needed to reference an entry in G. We store the adjacency list for a vertex in G as a 100-bit vector in a processor. The graph G is stored, with many copies, throughout the CM. Each of the 100 nodes is represented in a processor containing the adjacency list for a vertex in G. This means, with 64K processors, that there will be approximately 655 copies of the graph, one for every 100 matchings. Each processor can be made to access these copies randomly, so that contention among the processors is minimized. The address of the vertex neighbor list for vertex G_j needed by a matching can be calculated from the address of the matching processor, j and a random variable. $|H|$ is 30, so only 5 bits is needed 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 will contain a description of H as well as the partial matching it is expanding.

Initially no processors are allocated. We use *rendezvous allocation*[4] to assign processors to matchings. Consider a tableau (figure 5, in which the nodes of G are arranged left-to-right across the top, and the nodes of H are arranged top-to-bottom on the left. We represent matching node H_i with G_j by 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. Care must be taken to leave enough free processors so that all expanding search nodes can complete, that is, either fail or expand the full search sub-tree to leaf nodes. 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.

In each phase of matching generation, a matching at level k, initially 1, must expand itself to all legal successor matchings at the next level. Matching processors may be expanding at

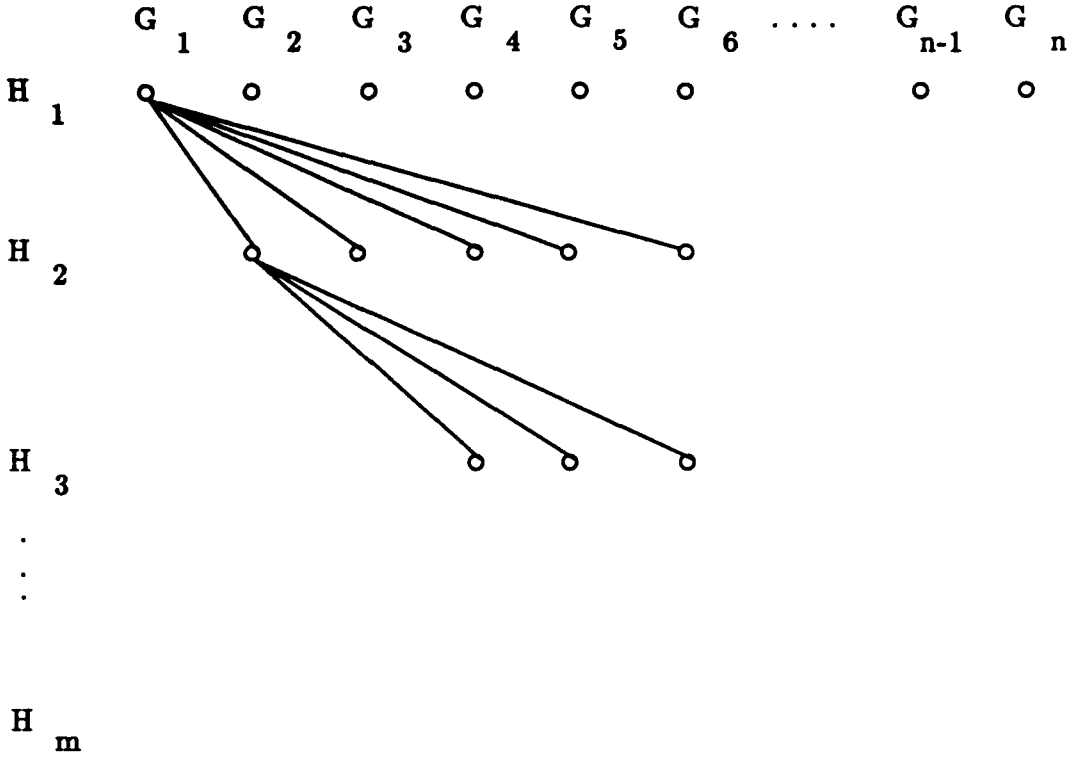


Figure 5: Match Expansion in Graph Matching

many different levels in the matchings, 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 the 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}, all processors step through the description of H, until they encounter the $k+1^{th}$ entry, and then record the contents of this entry. This should take no longer than $3ms$. This step finds the neighbors of the new vertex in H.

The order of expansion of H nodes can be arranged so that every node is connected to at least one previously matched vertex in H. Each expanding matching must then examine the neighbors in H of this new vertex to determine the nodes in G to which they have been matched. There will be at least one and no more than 3. For each such vertex in G, the neighbors of the node in G must be retrieved from the distributed representation of G. Each matching processor *sends* to the appropriate processor representing the neighbors of the appropriate vertex in G, and receives a return message describing the neighbors of that vertex in G. Each such message is 100 bits long, representing the set of vertices in G adjacent to the matched vertex; this takes $5ms$ per vertex, so $15ms$ total may be required. Now, we must compute the intersection of these bit vectors, describing all possible nodes in G which are adjacent to the matches in G of neighbors of H_{k+1}. This can be done in less than $3ms$, at the same time recording all vertices in G which lie in the intersection. Then a step of $3ms$ can exclude from the list all nodes already matched in the

current matching. These 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. Each matching will then have $m \leq 10$ possible expansion matches, and can request m processors from the free pool, in a processor allocation step. The description partial matching at the k^{th} level can be distributed to the successor nodes during allocation. The matching processor then returns itself to the free pool of processors. Some matchings may be prevented from expanding to allow matchings which are more advanced to reach completion. A priority mechanism can be implemented to favor matchings which are nearer completion. The overhead of allocation and distribution should be no more costly than the entire previous computation, which requires approximately $25ms$, bringing the total to $50ms$. Clearly, it is only slightly more difficult to maintain the cost of a matching and return the matching with the minimum cost than it is to generate all possible matchings. In fact, the constraint will reduce search when standard alpha-beta pruning methods are applied.

A very conservative estimate of the time to expand one level in the search tree is $50ms$. From the initial expansion, 30 steps are required to finish at least the very first full matching, so $1.5s$ in total are used to finish the first full expansion. The total throughput of this problem can be measured in terms of the number of partial matchings in each step. The critical factor in this problem is to control the number of active matchings. The process can monitor itself to record the average number of successors at each level, allowing good control of allocation. The rate of expansion, the number of legal successors at each level, should, at first, be high, then should taper off as more constraint occurs. If, say, 20 per cent of the processors are actively expanding, then this method can explore approximately 10K partial matching expansions for every $50ms$.

Graph Matching		
Method	Implemented	Estimated
Per expansion step	—	$50ms$

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 CM. The algorithm, a CM implementation of Dijkstra's algorithm, is given in [4]. 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.

Consider sending messages 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 should 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 CM, using a Virtual Processor ratio of 1:1.

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 histogramming, 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	3*ms*	2*ms*
Find Zero-Crossings	0.5*ms*	0.5*ms*
Propagate label	40*ms*	40*ms*
Enumerate curves	350μs	350μs
Rank and send pixels	91*ms*	41*ms*
Total - without Output	44*ms*	43*ms*
Total - with Output	135*ms*	99*ms*
Connected Component Labeling		
Doubling method (length = 512 x 512)	—	300*ms*
Doubling method (length = 512)	—	150*ms*
Scan method (12 phases)	450*ms*	150*ms*
Hough Transform		
Full 180 steps (512x512)	—	720*ms*
Full 180 steps (256x256)	—	540*ms*
From edge elements (512x512)	—	30*ms*
Geometric Constructions		
Convex Hull (from VD)	—	50*ms*
Convex Hull (Graham scan)	—	200*ms*
Convex Hull (Jarvis march)	150*ms*	100*ms*
Voronoi Diagram (1024x1024)	4*s*	2*s*
Voronoi Diagram (512x512)	1*s*	500*ms*
Minimum Spanning Tree (from VD)	—	150*ms*
Triangle Visibility		
Multiple copies	—	850*ms*
Scanning	—	1.0*s*
Graph Matching		
Per expansion step	—	50*ms*
Minimum Cost Path		
	90*ms*	50*ms*

Figure 6: Summary Table

References

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

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

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

[4] D. Hillis, *The Connection Machine*, MIT Press, Cambridge, 1985.

[5] W. Lim, "Fast algorithms for labeling connected components in 2-D arrays", Technical report (in preparation). Thinking Machines Corporation, Cambridge, Massachusetts, 1986.

[6] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing and C. Yap, "Parallel Computational Geometry", *Proc. 25th IEEE Symp. Found. of Comp. Sci.*, 1985, 468-477.

[7] M.H. Overmars and J. Van Leeuwen, "Maintenance of Configurations in the Plane", *Journalof Computer and System Sciences*, **23**, 166-204.

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

[9] 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.

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

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

END

9-87

DTIC