JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 4, 79–94 (1987)

Vision Algorithms for Hypercube Machines*

T. N. MUDGE AND T. S. ABDEL-RAHMAN

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109

Received April 21, 1986

Several commercial hypercube parallel processors with the potential to deliver massive parallelism cost-effectively have been announced recently. They open the door to a wide variety of application areas that could benefit from parallelism. Computer vision is one of these application areas. This paper develops a general model for hypercube machines, and uses it to show how vision algorithms can be executed on hypercubes. In particular, the steps in the problem of thick-film inspection are used as a concrete example. The time needed to complete a typical inspection is used to demonstrate the performance of hypercube machines. Experimental results from a hypercube machine illustrate the potential use of such machines. © 1987 Academic Press, Inc.

1. INTRODUCTION

Simple computer vision (CV) problems such as inspecting a printed circuit can easily require the processing of 10 Mbyte of data in a few seconds. If the inspection task is at all complex the processing power required runs into billions of operations per second. Therefore, in practice only a reduced version of such problems is implemented. The required level of processing power is possible only with a high degree of parallelism.

In the past there have been numerous proposals for massively parallel machines. Examples of such machines include the Illiac IV [2], which consists of 256 processors organized as four 8×8 arrays; Cytocomputers [17], which employ a number of identical processing elements that are cascaded in series to form a pipeline of processing stages; the Massively Parallel Processor (MPP) [3], which employs a 128×128 array of processors interconnected in a nearest-neighbor topology; the Pyramid Machine [22], in which the

* This work was supported in part by the Army Research Office under Contract DAAG29-84-K-0070.

0743-7315/87 $3.00

processors are interconnected in a pyramid topology; and the PASM architecture [19], in which a multipath routing network is used to connect a set of 1024 processing elements.

Recently several commercial systems offering 100–1000 processors in a hypercube configuration have been announced in the super-mini price range. Intel's "personal supercomputer," the iPSC, is an example. It comprises 32, 64, or 128 processing nodes connected in a regular hypercube topology [9]. The processing node is constructed from a standard 80286 16-bit microprocessor and 512 kbyte of memory. Connections between adjacent node processors are by point-to-point 10 Mbit/s ethernet connections. I/O is achieved also by a 10 Mbit/s ethernet connection that links a system manager and all of the node processors. A similar machine is available from Ametek Corporation [1]. The node processor is also an 80286 processor; however, in addition each node includes a separate 80186 processor to handle internode messages and can have as much as 1 Mbyte of memory. Up to 256 nodes can be incorporated into a system. Finally, a hypercube machine is offered by the NCUBE Corporation [8]. It has a custom 32-bit processor for the node processor that is capable of executing floating point operations. The connection between adjacent node processors is by point-to-point bit–serial links, and its I/O structure allows data transfers to/from the cube array over separate bit–serial links to each node. Its high level of integration allows systems with up to 1024 nodes to be assembled.

The idea of interconnecting processors in a hypercube topology is not new, going back to proposals as early as 1962 [20]. In 1975, IMS Associates Inc. announced a 256-node hypercube made up of 8080's, but it was never produced. For the most part however, the idea remained unexploited until the construction and demonstration of the Cosmic Cube at Caltech in 1983 [18, 23, 15, 7]. The hypercube topology yields a regular array in which nodes are quite close together: no more than $\log_2 N$ steps apart, where N is the number of nodes. At the same time the number of connections from each node to its neighbors is quite low (also $\log_2 N$). It thus strikes a balance between a two-dimensional array in which internode connection costs are low, but the nodes are far apart ($O(\sqrt{N})$ steps on average), and a completely connected array in which the internode connection costs are high, but the nodes are only one step apart. The hypercube topology is homogeneous, in the sense that all the nodes are identical. There are no special nodes such as those on the boundaries of mesh connected arrays, for example. It is also possible to divide a large hypercube array into smaller hypercube arrays, allowing fault tolerance and making it easy to support multiprogramming [14, 8]. The hypercube topology can also efficiently embed other regular topologies such as grids, trees, and pyramids [4, 21].

This paper develops a general model for hypercube machines, and uses it to show how vision algorithms can be executed on hypercubes. In particu-

lar, the steps in the problem of thick-film inspection are used as a concrete example.

The remainder of this paper is organized as follows. The next section presents a general model for hypercube machines. Section 3 outlines the problem of thick-film inspection and a set of algorithms to perform it. Section 4 maps those algorithms onto the hypercube model and shows the performance of hypercube machines for the algorithms. Section 5 presents experimental results obtained from an existing hypercube machine, a prototype NCUBE. Finally, Section 6 presents general comments and conclusions.

2. A Model for Hypercube Machines

There are two broad classifications for models of parallel processing: the *shared memory* model that characterizes tightly coupled processing, and the *distributed memory* model that characterizes loosely coupled processing. The shared memory model assumes that the processors have identical mechanisms for access to a common memory. The distributed memory model assumes that each processor has its own memory and communicates with others by an I/O access. Hypercube machines are distributed memory machines.

Hypercube machines are constructed from $N = 2^n$ identical processors connected through point-to-point bidirectional links in an n-dimensional hypercube array, or an n-cube. This hypercube array is in turn connected to the outside world through an I/O structure. The dimension n is also referred to as the order of the cube. Hypercubes can be constructed and their node processors labeled with a unique binary number according to the following recursive rule. Form a 1-cube as a system of two processors connected by a single communication link. Label one processor with a 0 and the other with a 1. This is the basis step. The general step constructs an n-cube from two $(n-1)$-cubes as follows. First, prefix the node labels in one of the $(n-1)$-cubes with a 0 so they are of the form $0xx \ldots xx$. Second, prefix the node labels in the other $(n-1)$-cube with a 1 so they are of the form $1xx \ldots xx$. Finally, connect the two $(n-1)$-cubes with communication links between nodes that have labels differing only in their most significant bit. Figure 1 shows a 4-cube.

Several points are worth noting. Nodes connected by a link have labels differing only in one bit. Each processor connects to the cube through n (or $\log_2 N$) links. At any point in time up to N links can be in use.

Figure 2 sketches a node processor. For the purposes of our model we will assume that it consists of a CPU with a cache or register file, main memory, and $(n+1)$ bidirectional DMA channels. The first n of the DMA channels are connected to communication links that join the node processor to its

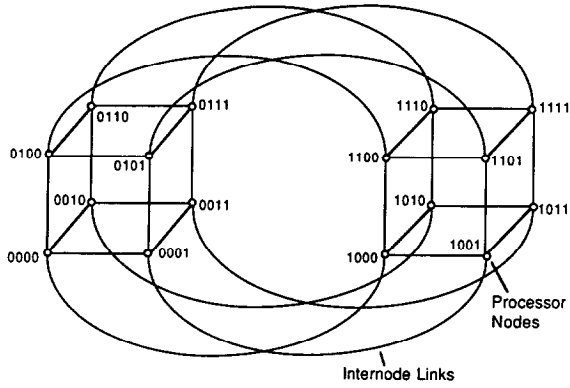

FIG. 1. A 4-cube.

nearest neighbors in the cube. The $(n + 1)$st DMA channel provides a link for communicating with the cube I/O subsystem. The channels are bidirectional and can support broadcasting from the processor on from 1 up to $(n + 1)$ of the links. DMA actions are modeled as buffer transfers that cycle-steal the bus from the CPU. It is assumed that caching allows the DMA to proceed so that a fraction α of the internode communication time can be overlapped with the node processing; α is termed the *degree of transparency*. The I/O structure is modeled as a channel into and out of the cube array with a bandwidth of B_{io} bytes per second.

The time for an algorithm to run on a hypercube is given by

$$T(N) = T_i + T_p + (1 - \alpha)T_c + T_o, \tag{1}$$

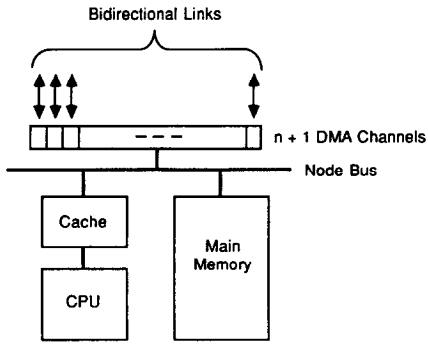

FIG. 2. Model of a node processor.

where N indicates the number of processors in the cube, T_i the time to input data to the cube, T_p the time to perform the processing at a node, T_c the internode communication time, and T_o the time to output data. These last four parameters are also, strictly speaking, functions of N. Frequently, we are interested in ignoring the effects of the I/O subsystem. Then

$$T(N) = T_p + (1 - \alpha)T_c. \tag{2}$$

The communication time is a consequence of having more than one processor since $T_c(1) = 0$, but $T_c(k) > 0$ for $k > 1$. Of course, if α can be kept close to 1.0 the effects of T_c can be hidden and the overall communication overhead, $(1 - \alpha)T_c$, kept to a minimum. The communication overhead is one of the two principal contributors to the intrinsic inefficiency of parallel algorithms. The other is the dependencies within the algorithm that do not permit all the N processors to be used all the time. A node processing efficiency of less than 1.0 is an indication of this. This efficiency measure is given by

$$E_p(N) = \frac{T_p(1)}{NT_p(N)} \leqslant 1. \tag{3}$$

If the efficiency of the system, excluding I/O considerations, is given by

$$E(N) = \frac{T(1)}{NT(N)}, \tag{4}$$

then from (2) and (3) and the fact that $T_c(1) = 0$, we can write

$$E(N) = \frac{E_p(N)}{1 + (1 - \alpha)(T_c(N)/T_p(N))}. \tag{5}$$

From (5) we can define a *perfectly scalable* algorithm as one where $E(N) = 1$. In other words, the node processing is 100% efficient,

$$E_p(N) = 1, \tag{6}$$

and the communication overhead is zero,

$$(1 - \alpha)T_c(N) = 0. \tag{7}$$

Loosely speaking a perfectly scalable algorithm can make use of large numbers of processors without diminishing returns.

3. The Thick-Film-Inspection Problem

To illustrate how a number of typical CV algorithms can be executed on a hypercube we will consider the steps in the automatic inspection of thick film (TF) circuits [10]. These circuits are a network of conductors and dielectrics printed onto a ceramic substrate. The circuits are populated with electronic components, but, prior to this, and as a quality control step, each TF circuit is inspected to see if it satisfies a set of geometric specifications. The geometric check, if passed, increases confidence in the likelihood of the correct electrical operation of the circuit. The geometric specifications are phrased in terms of a basic unit of length referred to as a *design rule.* At present, typical inspection systems acquire images of TF circuits that have 10 pixels per design rule. This allows defects as small as 0.1 of a design rule to be detected. A TF circuit is imaged as a composite of about 40 frames each composed of 512×512 1-byte pixels to obtain the 0.1 of a design rule resolution. This results in about 10 Mbyte of data to be processed per substrate.

After the data from the imaging device have been input into the cube array, the TF inspection problem breaks down into the following steps.

1. *Tonal mapping.* This step is needed if it is required to adjust for unevenness or imperfections in the imaging device. It can also be used as part of an automatic periodic recalibration of the imaging device.

2. *Alignment.* This step involves translating and rotating the image to a reference position and orientation. The amount of translation (Δx, Δy) and rotation ($\Delta\theta$) is determined by inspecting fiducial marks on the substrate. Current substrate handlers guarantee that Δx and Δy are less than 3% of the linear dimensions of the substrate, and that $\Delta\theta$ is less than 3°. In present systems, for performance reasons, this step is performed by mechanically aligning the substrate.

3. *Edge detection.* This step applies a simple edge operator, such as the Sobel operator. Nonmaximal suppression is performed on the resulting edge strength values to yield pixel-wide edges. An edge following operation with hysteresis is then carried out to yield a set of closed contours [16, 5].

4. *Reference check.* This step compares the segmented image output from the previous stage with prestored templates to determine if there are any geometric violations.

5. *Error reporting.* This final step interprets the results of the reference checking. Many types of geometrical errors found by the reference check do not cause failure in the circuit operation; for instance, a small spur of conducting material. However, the interpretation of the geometrical errors requires considerable knowledge about the properties of the TF and its in-

tended operation. It is important to report them but it is not necessary to reject the circuit. Often nonfatal geometric errors indicate trends in the manufacturing process that are harbingers of fatal errors later. Error reporting can most naturally be implemented as an expert system. However, in present systems this is not the case, and the level of reporting is fairly crude. In this paper we consider only simple reporting.

From the above set of steps, it can be seen that the TF inspection problem forms a simple paradigm of the CV process in general. There are preliminary phases of *low-level* processing where it is required to work on the two-dimensional representation of the image, *intermediate-level* operations such as edge following that work on noniconic data structures, and a final *high-level* interpretive phase where simple elements of machine intelligence are required [12].

4. Mapping the Algorithms onto the Hypercube

The starting point for mapping the steps of Section 3 onto the hypercube is to subdivide the image among the processing nodes. A natural assignment is to partition the $M \times M$ image into a gray coded tessellation of $m_1 \times m_2$ subimages similar to an n-dimensional Karnaugh map and then to place each subimage with its like-numbered processor. Figure 3 shows how this can be done for the hypercube of Fig. 1. This is the data input step. In the case in which n is even, the subimages are square and $m_1 = m_2 = m = M \cdot 2^{-n/2}$. In the case in which n is odd, the subimages are rectangular, and $m_1 = M \cdot 2^{-(n-1)/2}$ and $m_2 = M \cdot 2^{-(n+1)/2}$. In either case, the subimages are equal in size. For simplicity, we carry out the analysis for the case in which n is even. A similar analysis can be performed for the other case. The remaining steps in the TF inspection task can then be performed as follows.

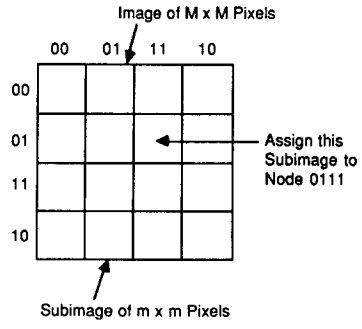

FIG. 3. Assignment of data to nodes.

4.1. *Tonal Mapping*

This is a simple byte-wise translation of the input image and can be done by table lookup. It is a scalable algorithm since (6) holds for any table lookup function and (7) holds because there is no internode data movement. The time for tonal mapping is proportional to

$$m^2 t_+, \tag{8}$$

where t_+ is the time for a node processor to perform an additive operation. We assume that this approximates the time taken to look up an item in a table.

4.2. *Alignment*

As a result of translation followed by rotation a pixel at location (x, y) will move to location (u, v), where

$$\begin{aligned} u &= (x + \Delta x)\cos(\Delta\theta) - (y + \Delta y)\sin(\Delta\theta) \\ v &= (x + \Delta x)\sin(\Delta\theta) + (y + \Delta y)\cos(\Delta\theta). \end{aligned} \tag{9}$$

Since $\Delta\theta$ is small enough to ignore second-order terms, these equations can be simplified to

$$\begin{aligned} u &= x + \Delta x - (x + y)\Delta\theta \\ v &= y + \Delta y + (x - y)\Delta\theta. \end{aligned} \tag{10}$$

In other words each pixel must undergo six additive and two multiplicative operations for the coordinate transformation. The values u and v should be rounded to nearest integers. Strictly speaking a resampling phase should be conducted to resample the aligned image onto an integer grid. However, rounding is an approximatism to resampling with nearest-neighbor interpolation. For an image with an initial granularity of 0.1 of a design rule, inaccuracies introduced by rounding can be ignored.

We assume that in the worst case the translation requires every subimage to be transferred to an adjacent node and that the rotation may also require subimages at the edge of the image to be transferred to an adjacent node (see Fig. 4). From our earlier observation concerning Δx, Δy, and $\Delta\theta$, this sets bounds on the ratio m/M by implying that $m/M > 0.03$ and $180m/M > 3°$, respectively. The data movements between adjacent pairs of nodes can proceed simultaneously; therefore, the upper bound on the time to align the image is proportional to

$$2(3t_+ + t_*)m^2 + 9m^2 t_L(1 - \alpha), \tag{11}$$

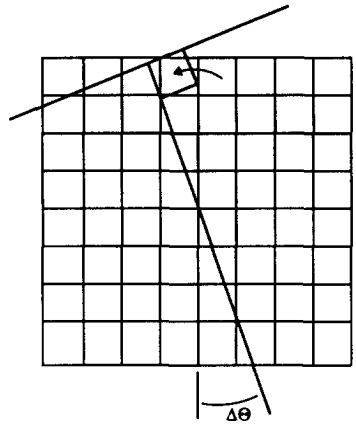

FIG. 4. Effects of rotation on edge subimages.

where t_* is the time for a node processor to perform a multiplicative operation and t_L is the time to move a byte across a link connecting adjacent nodes. The first term is the time to create u and v. The second term is the time for three subimage transfers (two for translation and one for rotation) between adjacent nodes. The factor of 9 arises because we assume 3 bytes need transferring—the pixel and u and v.

4.3. *Edge Detection*

A Sobel operator requires that the image be convolved with the two familiar kernels shown in Fig. 5.

Convolution with the left-hand kernel yields the x-direction edge strength (gradient) pixels, e_x. Convolution with the right-hand kernel yields the y-direction edge strength (gradient) pixels, e_y. The strengths e_x and e_y are combined to give the combined edge strength $\sqrt{e_x^2 + e_y^2}$ and the direction of the edge, $\arctan(e_y/e_x) - \pi/2$. Representing these two values requires 4 bytes per pixel. Since these values and the aligned image must exist together, the

-1	0	1
-2	0	2
-1	0	1

1	2	1
0	0	0
-1	-2	-1

FIG. 5. The Sobel kernels.

combined memory of the nodes must be greater than 5 times the image size to avoid having to use secondary memory. We will assume the node memory is sufficient. If multiplication by 2 is implemented as repeated addition, the convolutions require eight additive operations per pixel. The time for a node to complete these additions is proportional to $16m^2t_+$. In order to perform the convolution on the pixels around the edges of a subimage, a pixel-wide strip of pixels must be copied from the four adjacent subimages. In addition, four corner pixels must be copied in from two nodes away. Since movements between pairs of adjacent nodes can proceed simultaneously, the time for these data transfers is proportional to $(4m + 8)t_L$.

When calculating the edge strength, the square root can be performed by table lookup for the first 8 bits of the result followed by one iteration of the Newton–Raphson method to obtain a 16-bit result. This requires one additive operation, one multiplicative operation (a divide), and a right shift (divide by 2). Assuming the time for a shift is the same as that for an additive operation, square root extraction takes time units proportional to $m^2(2t_+ + t_*)$. With respect to edge direction, we only need to know if the edge direction is stronger in the y-direction or the x-direction. The stronger determines the direction of primary edge strength. This can be accomplished by testing to see if $|e_x| > |e_y|$ and then checking the signs of e_x and e_y. If comparisons are counted as additive operations the time for this step is proportional to $3m^2t_+$. The factor of 3 accounts for the need to check signs after comparing magnitudes.

Each pixel with greater than zero edge strength is a potential edge point. Nonmaximal suppression is used to thin the potential edge pixels to pixel-wide edges. This can be done by comparing the strength of each edge pixel to that of the two neighboring pixels that are orthogonal to its primary edge direction. If its strength is less than either of the neighbors it is discarded. Since every potential edge pixel must be examined, nonmaximal suppression time is proportional to $2m^2t_+$.

Finally, the pixel-wide edges that do not form closed contours of sufficient strength must be removed. A strong edge pixel is selected that is above a predefined threshold τ. The pixel-wide edge containing it is followed as long as no edge pixels occur with a strength of $<\tau/2$. Those lines with no edge pixels above $\tau/2$ are removed. The remaining edges form closed contours that separate the different regions of the substrate and form the pattern that must be examined for geometrical violation by the reference check. The time to do this contour generation depends on the number of contour pixels in a subimage. This can vary from subimage to subimage, with the result that some nodes have more work to do than others. This class of problems has been discussed in [11], where they are termed *feature dependent* algorithms. If p is the probability that a pixel is part of a contour we can estimate the time to generate the contours to be proportional to pm^2t_+. Again, we have

assumed that comparisons take as long as additive operations. Collecting all the terms together for the edge detection step yields

$$(23 + p)m^2t_+ + m^2t_* + 4(m + 2)t_L(1 - \alpha). \tag{12}$$

4.4. *Reference Check*

Given the previous steps the reference check can be accomplished by simple image comparison with a foreground and a background template (see Fig. 6). The templates, which define the reference position and orientation with which the image was originally aligned, are prestored across the hypercube nodes using the same mapping shown in Fig. 3 for the image. Each node works independently, comparing the subimage it contains to the two prestored subtemplates. If the contour is not contained between the templates the substrate fails the inspection.

The reference check operation is perfectly scalable and requires only that each pixel be compared with the two templates prestored across the nodes. The time for reference checking is thus proportional to $2m^2t_+$.

4.5. *Error Reporting*

Those areas around the contour sections that fall outside of the templates in the reference check represent geometric errors in the substrate. This must be interpreted and appropriate reporting done. The time taken for this step depends heavily on the sophistication of the interpretation required. Currently, many systems simply report the error and its location. The time to do this can be ignored compared to the previous steps. We will assume for the purposes of this discussion that this is how errors are reported.

We have assembled the times from steps 1 through 5 in Table I. If we assume 1024 processors, no internode communication transparency, and a

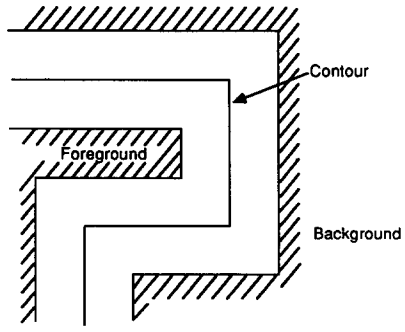

FIG. 6. Reference check templates.

TABLE I
EXECUTION TIMES

Step	Time
1. Tonal mapping	$m^2 t_+$
2. Alignment	$2(3t_+ + t_*)m^2 + 9m^2 t_L(1 - \alpha)$
3. Edge detection	$(23 + p)m^2 t_+ + m^2 t_* + 4(m + 2)t_L(1 - \alpha)$
4. Reference check	$2m^2 t_+$
Total	$(32 + p)m^2 t_+ + 3m^2 t_* + (9m^2 + 4m + 8)(1 - \alpha)t_L$

substrate image of 10 Mbyte, then $m^2 = 10$ kbyte and the inspection time is approximately proportional to

$$3 \times 10^5 t_+ + 3 \times 10^4 t_* + 9 \times 10^4 t_L. \tag{13}$$

In the following section, results obtained from a prototype NCUBE system are presented.

5. EXPERIMENTAL RESULTS

The algorithms for the TF inspection problem were implemented on a prototype NCUBE hypercube system. This section briefly overviews the NCUBE system and presents the results obtained from that system.

The NCUBE system [14, 8] has up to 1024 processing nodes that are interconnected in a regular hypercube array topology. The cube array is connected via a set of I/O channels to as many as eight I/O and/or host processors. The I/O structure allows data transfers with the cube array over separate bit-serial links to and from each node. These links can support a total data rate as high as 90 Mbyte/s into or out of the cube.

The processing node consists of a custom 32-bit processor chip and 128 kbyte of high-speed memory. The processor chip is a general-purpose processor which is capable of executing non-floating-point instructions at about 2 MIPS, single-precision floating point instructions at about 0.5 MFLOPS, and double-precision floating point instructions at about 0.3 MFLOPS. Benchmarks (Whetstone [6] and Dhrystone [24]) performed at the University of Michigan show that the performance of the NCUBE processor chip (10 MHz) is significantly better than that of a VAX 11/780 processor with a floating point accelerator. Figures are shown below.

	VAX 11/780	NCUBE node
Whetstones instr/s	426,000	476,000
Dhrystones/s	741	1,249

Communication with other nodes is done asynchronously through 22 bit-serial links. The links are paired into 11 bidirectional channels. Ten of these bidirectional channels are used to form the 10-dimensional hypercube array. The eleventh channel provides the I/O mentioned above. The channels operate at 10 MHz with parity check, which results in a data transfer rate of about 1 Mbyte/s in each direction. Each channel has two 32-bit write-only registers associated with it. One is the address register, which contains the location, in the node memory, of the first byte in the message. The other is the count register, which indicates the number of bytes left to send or receive in the message. Send or receive operations are initiated by the processor by writing the address of the first byte in the message to the address register and the size, in bytes, of the message to the count register (a nonzero count actually triggers the DMA action). The processor then continues with its operations while the DMA channels complete the communication operation. Interrupts are used to signal the processor when the channel is ready for a new transfer.

The node operating system, called VERTEX, is a small nucleus that resides in each node. It basically provides communication between the nodes via a set of send and receive functions that facilitates message transfer between any two nodes in the hypercube array. The send function has the general form

nwrite(*length, message, dest, type, status, error*),

where *length* is the length of the outgoing message (the length of a message can be up to 64 kbyte), *message* is the name of the buffer that contains the message; *dest* is the logical number of the node that is to receive the message; *type* is the type of the message, an attribute that is used to distinguish messages; *status* indicates when the message has left the buffer *message* to the VERTEX buffer area; and *error* is an error code. The receive function has a similar general form:

nread(*length, message, source, type, status, error*).

nread looks for the first message from *source* of type *type* and copies it into the buffer *message*. It is possible to specify "don't care" conditions for *source* and *type* to receive a message regardless of its source or type.

Programs on the NCUBE can be developed in host and node assembly language or in one of two high-level languages, FORTRAN 77 and C.

The algorithms comprising the TF inspection task were implemented on a prototype NCUBE system using FORTRAN 77. The programs were run on a 6-dimensional cube with no internode communication transparency. The communication time between two nearest neighbors in the cube was found to be independent of the size of the cube. Therefore, appropriate scal-

ing of the size of the subimage made it possible to scale the results obtained from the 6-dimensional cube to those that could be obtained from a 10-dimensional cube. Also, since our prototype was running at 7 MHz, the results were scaled by a factor of 0.7 to give the 10 MHz equivalent. Table II shows the scaled execution times for the various steps in the TF inspection for 40 frames of 512×512 1-byte images on 1024 processors with no internode communication transparency (excluding the I/O time). The figures in this table reflect not only the basic operations involved in the TF inspection (shown in Table I), but also all the overhead incurred by the various housekeeping functions involved in the implementation of the algorithms.

There are a number of factors that contribute to the overhead. One factor is the additional instructions needed to actually implement the algorithms (e.g., DO loop instructions and index arithmetic). Another factor is the overhead incurred by using FORTRAN as the language of implementation (or any high-level language for that matter). This overhead can be eliminated by coding the algorithms in assembly language. The last factor contributing to the overhead is the node operating system. This results in a considerable overhead in communication time. This overhead can be minimized, in our application, by designing special-purpose communication routines for the 8 nearest-neighbor topology [13]. Preliminary studies suggest that eliminating the last two sources of overhead can improve the figures in Table II by a factor of 2–3.

Clock cycle counts suggest that the node processors are capable of performing additive operations in about 1 μs, multiplicative operations in about 10 μs, and internode byte transfers in about 1 μs. Then using (13), the TF inspection time works out to be about 0.7 s. It is important, however, to realize that this figure is obtained ignoring all the sources of overhead discussed above. Thus, it is an extreme lower bound on the performance of the hypercube, which can be approached only by careful hand coding of the algorithms.

TABLE II
EXECUTION TIMES FROM THE NCUBE SYSTEM

Step	Time (s)
1. Tonal mapping	0.09
2. Alignment	1.04
3. Edge detection	1.53
4. Reference check	0.16
Total	2.82

6. Conclusions

A model for hypercube machines has been developed and its performance estimated for a characteristic CV task, TF inspection. There was an element of arbitrariness in selecting the steps needed for TF inspection; however, alternative approaches will probably employ very similar algorithms. Based on simple operation counts a lower bound of 0.7 s was obtained for the inspection problem; however, no account was taken of various housekeeping steps that are needed. The effect of these steps was shown using the NCUBE hypercube system. The results indicate that hypercube machines have great potential for low- to intermediate-level computer vision algorithms, particularly if the node processor were optimized for vector processing on subimages (which is not the case with the NCUBE). Future work will determine their suitability for higher-level functions, such as those that will eventually be found in the error reporting step. The fact that hypercubes are MIMD machines makes them good candidates for this kind of processing.

Acknowledgment

The authors thank the referees for their comments and suggestions.

References

1. Ametek Corporation. Hypernet System 14/n, 1985.
2. Barnes, G. H., *et al.* The Illiac IV computer. *IEEE Trans. Comput.* **C-17,** 8 (Aug. 1968), 746–757.
3. Batcher, K. E. Architecture of a massively parallel processor. *Proc. 7th Annual Symp. on Computer Architecture,* May 1980, pp. 168–174.
4. Bhatt, S., Chung, F., Leighton, T., and Rosenberg, A. Optimal simulation of tree machines. *Proc. Foundations of Comp. Sci. Conference,* Toronto, 1986.
5. Canny, J. F., Finding edges and lines in images. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1983.
6. Curnow, H. J., and Wichmann, B. A. A synthetic benchmark. *Comput. J.* **19,** 1 (1976).
7. Fox, G. The performance of the CALTECH Hypercube in scientific calculations. Report CALT-68-1298, California Institute of Technology, Apr. 1985.
8. Hayes, J. P., *et al.* Architecture of a hypercube supercomputer. *Proc. Int'l. Conf. on Parallel Processing,* Aug. 1986.
9. Intel Corporation. The iPSC data sheet. Beaverton, OR, 1985.
10. Leonard, P. F., Svetkoff, D., Kelley, R., and Rohr, D. Machine vision applications in electronic manufacturing. *Proc. Vision 85,* Detroit, MI, Mar. 25–28, 1985, pp. 5-99 to 6-116.
11. Mudge, T. N., and Abdel-Rahman, T. S. Efficiency of feature dependent algorithms for the parallel processing of images. *Proc. Int'l. Conf. on Parallel Processing,* Aug. 1983, pp. 369–373.

12. Mudge, T. N., and Abdel-Rahman, T. S. Architectures for robot vision. In Graham, J. (Ed.). *Specialized Computer Architectures for Robotics and Automation.* Gordon & Breach, New York, 1986.
13. Mudge, T. N., Buzzard, G. D., and Abdel-Rahman, T. S. A high performance operating system for the NCUBE. *Proc. Second Conf. on Hypercube Multiprocessors,* Knoxville, TN, Sept. 29–Oct. 1, 1986.
14. NCUBE Corp., NCUBE Handbook, version 0.6. Beaverton, OR, Dec. 1985.
15. Peterson, J. C., Tuazon, J. O., Liberman, D., and Pniel, M. The Mark III Hypercube-Ensemble Concurrent Computer. *Proc. Int'l. Conf. on Parallel Processing,* Aug. 1985, pp. 71–73.
16. Rosenfeld, A., and Kak, A. *Digital Picture Processing.* Academic Press, New York, 1976.
17. Rutenbar, R. A., Mudge, T. N., and Atkins, D. E. A class of cellular architectures to support physical design automation. *IEEE Trans. CAD of IC's and Systems* **CAD-3,** 4 (Oct. 1984), 264–278.
18. Seitz, C. L. The Cosmic Cube. *Comm. ACM* **28** (Jan. 1985), 22–33.
19. Siegel, H. J., *et al.* PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comput.* **C-30,** 12 (Dec. 1981), 934–947.
20. Squire, J. S., and Palais, S. M. Programming and design considerations for a highly parallel computer. *Proc. Spring Joint Computer Conf.,* 1963, pp. 395–400.
21. Stout, Q. Hypercubes and pyramids. In Cantoni, V., and Levialdi, S. (Eds.). *Pyramidal Systems for Image Processing and Computer Vision.* NATO ASI Series ARW. Springer-Verlag, New York, 1986, in press.
22. Tanimoto, S. L. A pyramidal approach to parallel processing. *Proc. 10th Annual Int'l. Symp. on Computer Architecture,* Stockholm, June 1983, pp. 372–378.
23. Tuazon, J. O., Peterson, J. C., Pniel, M., and Liberman, D. CALTECH/JPL Mark II Hypercube Concurrent Processor. *Proc. Int'l. Conf. on Parallel Processing,* Aug. 1985, pp. 666–671.
24. Weicker, R. P. Dhrystone: A synthetic system programming benchmark. *Comm. ACM* **27,** 10 (Oct. 1984), 1013–1030.