VISION ALGORITHMS FOR HYPERCUBE MACHINES[1]

T.N. Mudge

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-1109

ABSTRACT

Several commercial hypercube parallel processors have recently been announced with the potential to deliver massive parallelism cost-effectively. They open the door to a wide variety of applications 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 thick film inspection problem are used as a concrete example. The time needed to complete a typical inspection is used to determine the performance necessary by a hypercube machine to be suitable for such inspections.

1. Introduction

Simple computer vision (CV) problems such as inspecting a printed circuit can easily require the processing of 10M bytes of data in under a second. 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 only possible with a high degree of parallelism. This has not been available cost-effectively. However, several commercial systems offering 100-1000 processors in a hypercube configuration have recently been announced. Intel's "personal supercomputer," the iPSC, is an example. It is comprised of 32, 64, or 128 processing nodes connected in a regular hypercube topology. The processing nodes are constructed from standard 80286 16 bit microprocessors and 512K bytes of memory. Connections between adjacent node processors is by point-to-point 10M bit per second ethernet connections. I/O is achieved also by a 10M bit per second ethernet connection that links a system manager and all of the node processors. A similar machine is available from Ametek Corporation. The node processor is also a 80286; however, in addition each node includes a separate 80186 processor to handle internode messages and can have as much as 1M bytes of memory. Up to 256 nodes can be incorporated into a system. And, finally, a hypercube machine is being offered by the NCUBE Corporation. 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 array is not new, going back to proposals as early as 1962 [1]. 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 has remained unexploited until the construction and demonstration of the Cosmic Cube at Caltech in 1983 [2]. 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, at the expense of having the nodes far apart ($0(\sqrt{N})$ steps on average), and a completely connected array in which the internode connection costs are high, since there are $0(N^2)$ of them, in order to have nodes only one step apart. This paper develops a general model for hypercube machines, and uses it to show how vision algorithms can be executed on a hypercubes. In particular, the steps in the thick film inspection problem 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 thick film inspection problem and a set of algorithms to perform it. Section 4 maps those algorithms onto the hypercube model. The performance necessary by a cube machine capable of doing the inspection within the desired time limit is estimated.

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

[1]This work was sponsored in part by Army Research Office Contract No. DAAG29-84-K-0070.

Proceedings of the IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management, November 1985, pages 225-230.

processing. The shared memory model assumes the processors have identical access mechanisms 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 into a hypercube. This hypercube array is in turn connected to the outside world through an I/O structure. The order of an N processor hypercube is given by n, and it is also referred to as an n-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 2 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 $0xx\cdots xx$. Second, prefix the node labels in the other $(n-1)$-cube with a 1 so they are of the form $1xx\cdots xx$. Finally, connect the two $(n-1)$-cubes with communication links between nodes that have labels differing only in their most significant bit (Fig. 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 it consists of a CPU with a cache or large register file, main memory and $(n+1)$ bidirectional DMA channels. The first n of the DMA channels are connected to the communication links that joins a node processor to its 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

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

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,

$$T(N) = T_p + (1-\alpha)T_c. \qquad (2)$$

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,

$$E_p(N) = \frac{T_p(1)}{NT_p(N)} \leq 1. \qquad (3)$$

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

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

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

$$E(N) = \frac{E_p(N)}{1 + (1-\alpha)\dfrac{T_c(N)}{T_p(N)}} \qquad (4)$$

From (4) we can define a *perfectly scalable* algorithm as one where $E(N) = 1$. In other words, the node processing is 100% efficient,

$$E_p(N) = 1 \qquad (5)$$

and the communication overhead is zero,

$$(1-\alpha)T_c(N) = 0. \qquad (6)$$

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. 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; circuit in a conductor.

Each TF circuit is imaged as a composite of about 40 frames each composed of 512×512 1-byte pixels to obtain a resolution of 0.1 of a design rule. This results in about 10M bytes of data to be processed per substrate. Currently, the printers that create the TF pattern can produce approximately one per second. Thus if a single automatic inspection device is to handle the printer's output it must process 10M bytes of input data per second. At present, it is not possible to build a system that can do this cost-effectively, so the inspection usually relies on inspecting only a sample subregion of the substrate.

After the data from the imaging device has been input, 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. We will assume that Δx and Δy are less than 3% of the linear dimensions of the substrate. Further, we will assume that $\Delta\theta$ is less than 3°. In current 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. Non-maximal 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.

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 intended operation. It is important to report them but it is not necessary to reject the circuit. Often non-fatal 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.

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 *iconic* 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 non-iconic data structures, and a final interpretive phase where simple elements of machine intelligence are required.

4. Mapping the Algorithms onto the Hypercube

The starting point for mapping the steps of Sec. 3 onto the hypercube is to subdivide the image among the node processors. A natural assignment is to partition the $M \times M$ image into a Gray coded tesselation of $m \times m$ subimages similar to an n-dimensional Karnaugh map and then to place each subimage with its like numbered processor (i.e., $M^2 = m^2 N$). Figure 3 shows how this can be done for the hypercube of Fig. 1. This is the data input step. The remaining steps in the TF inspection task can then be performed as follows.

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 (5) holds for any table lookup function and (6) holds because there is no internode data movement. The time for tonal mapping is given by,

$$m^2 t_+$$

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

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

$$u = (x + \Delta x)\cos(\Delta\theta) - (y + \Delta y)\sin(\Delta\theta)$$
$$v = (x + \Delta x)\sin(\Delta\theta) + (y + \Delta y)\cos(\Delta\theta)$$

Since, we assume $\Delta\theta$ to be small and ignore second order terms these equations can be simplified to

$$u = x + \Delta x - (x + y)\Delta\theta$$
$$v = y + \Delta y + (x - y)\Delta\theta.$$

In other words each pixel must undergo 6 additive and 2 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 resam-

pling 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 assumptions concerning Δx, Δy and $\Delta\theta$, this sets bounds on the ration m/M by implying that $\Delta x = \Delta y = \frac{m}{M} < 0.03$ and $\Delta\theta \approx \frac{90m}{M} < 3°$. The data movements between adjacent pairs of nodes can proceed simultaneously, therefore, the time to align the image is bounded above by,

$$2(3t_+ + t_*)m^2 + 6m^2t_L(1-\alpha)$$

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 2 subimage transfers (one for translation and one for rotation) between adjacent nodes. The factor of 6 arises because we assume 3 bytes need transferring — the pixel and. u and v.

3. **Edge detection.** A Sobel operator requires that the image be convolved with the two familiar kernels shown below. Convolution with the lefthand kernel yields the x-direction edge strength pixels, e_x. Convolution with the right-hand one yields the y-direction edge strength 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, $arctan\,(e_y/e_x)$. Representing these two values requires 4 bytes per pixel. Since these values and the aligned image must exist together, the 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 two is implemented as repeated addition, the convolutions require 8 additive operations per pixel. The time for a node to complete these additions is given by, $16m^2t_+$. 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, 4 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 given by, $(4m + 8)t_L$.

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

e_y

-1	0	1
2	0	2
-1	0	1

e_x

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 two). Assuming the time for a shift is the same as that for an additive operation, square root extraction takes $m^2(2t_+ + t_*)$ time units. 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 this step takes $3m^2t_+$. 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. Non-maximal suppression is used to thin the number of 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 strength. If its strength is less than either of the neighbors it is discarded. Since every potential edge pixel must be examined, non-maximal suppression may take as much as $2m^2t_+$ time units.

Finally, the pixel-wide edges that do not form close 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 resulting in some nodes having more work to do than others. This class of problems has been discussed in [3] 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 as pm^2t_+. Again, we have assumed comparisons take as long as additive operations. Collecting all the terms together for the edge detection step yields the following time,

$$(23 + p)m^2t_+ + m^2t_* + 4(m+2)t_L(1-\alpha)$$

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. 5). If the contour is not contained between the templates the substrate fails the inspection. This operation is perfectly scalable and requires only that each pixel be compared with two templates prestored across the nodes templates. The time for reference checking is

thus given by $2m^2t_+$.

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 can assemble the times from steps 1 through 5 into the table below.

STEP	TIME
0. Input	M^2B_{io}
1. Tonal mapping	m^2t_+
2. Alignment	$2(3t_+ + t_*)m^2 + 6m^2t_L(1-\alpha)$
3. Edge detection	$(23+p)m^2t_+ + m^2t_* + 4(m+2)t_L(1-\alpha)$
4. Reference check	$2m^2t_+$
Total	$M^2B_{io} + (32+p)m^2t_+ + 3m^2t_* + 2(3m^2+2m+4)t_L$

If we assume 1024 processors, no internode communication transparency, and a substrate image of 10M bytes, then $m^2 = 10$K bytes and the total inspection time can be approximated by,

$$10\times10^6B_{io} + 3\times10^5t_+ + 3\times10^4t_* + 6\times10^4t_L.$$

If we further assume an I/O subsystem capable of inputting data at 100M bytes/S and node processors capable of performing additive operations in $1\mu S$, multiplicative operations in $10\mu S$, and internode byte transfers in $1\mu S$, the inspection time works out to be about 0.9 seconds. This meets the deadline imposed by the circuit printer of 1 second. The node memory needs to be at least 64K bytes.

5. Conclusions

A model for hypercubes 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. The performance figures for node processors were very conservative; on the other hand, no account was taken of various housekeeping steps that are needed. Nevertheless, the brief analysis indicates that hypercube machines should be ideal for low to intermediate level computer vision algorithms. Future work will determine their suitability for higher level functions, such as those that will eventually be found in the error reporting step

References

[1] J. S. Squire and S. M. Palais, "Programming and design considerations for a highly parallel computer," *Proc. Spring Joint Computer Conf.*, pp. 395-400, 1963.

[2] C. L. Seitz, "The Cosmic Cube," *Comm. ACM*, vol. 28, pp. 22-33, Jan. 1985.

[3] T. N. Mudge and T. S. Abdel-Rahman, "Efficiency of Feature Dependent Algorithms for the Parallel Processing of Images," *Proc. Int'l. Conf. on Parallel Processing*, pp. 369-373, Aug. 1983.

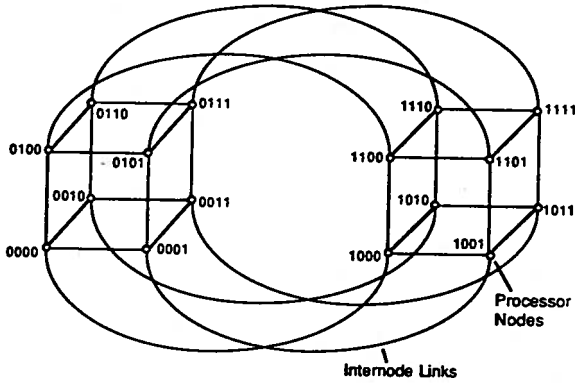

Figure 1. A 4-cube.

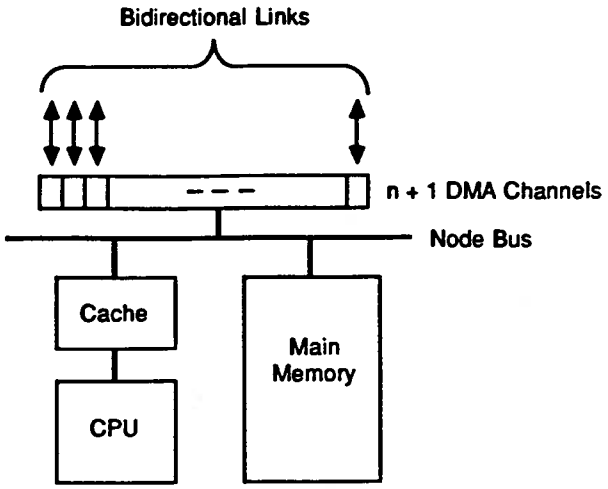

Figure 2. Model of a node processor.

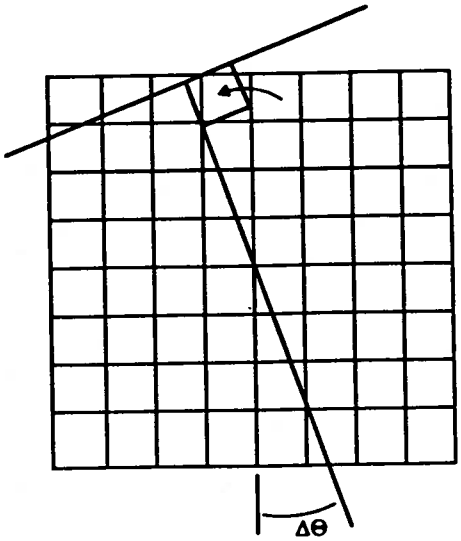

Figure 4. Effect of rotation on edge subimages.

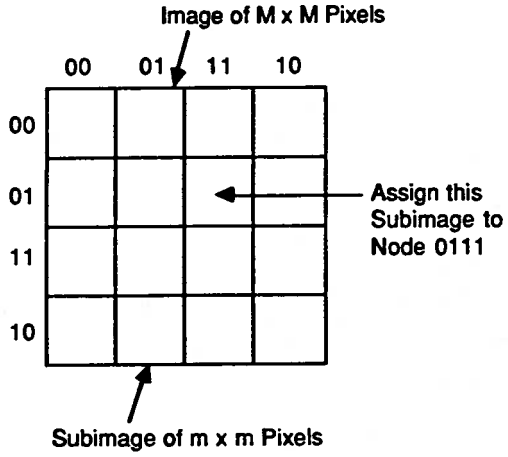

Figure 3. Assignment of data to nodes.

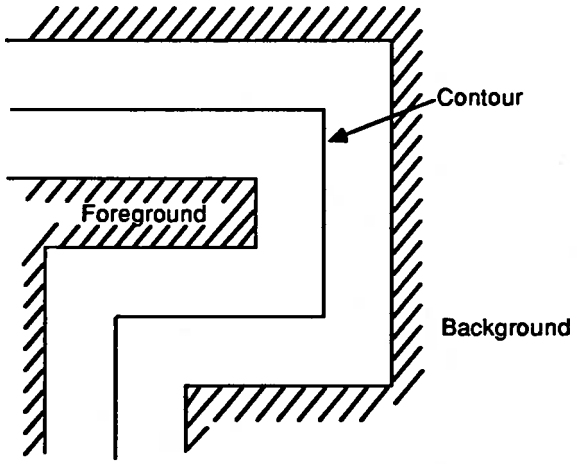

Figure 5. Reference check templates.