A Reprint from the

PROCEEDINGS

Of SPIE-The International Society for Optical Engineering

Volume 755

Image Pattern Recognition: Algorithm Implementations, Techniques, and Technology

12–13 January 1987
Los Angeles, California

An analysis of hypercube architectures for image pattern recognition algorithms

T. N. Mudge
Advanced Computer Architecture Laboratory
and
Robot Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122

An Analysis of Hypercube Architectures for Image Pattern Recognition Algorithms

by

T.N. Mudge
Advanced Computer Architecture Laboratory
and
Robot Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

Abstract

Hypercube architectures are introduced. The reasons behind their becoming the first widespread commercial massively parallel processors are outlined. A classification for image pattern recognition is proposed and characteristic algorithms are presented. Some simple ideas for organizing these algorithms to execute efficiently on hypercubes are also presented. The current programming model for hypercube machines is explained and illustrated with one of the characteristic algorithms. Preliminary performance figures are discussed for the NCUBE/six, a 64 processor commercial hypercube, followed by some concluding remarks.

1. Introduction

Parallel processing seeks to improve the speed with which a computational task can be done by breaking it into subtasks and executing as many as possible of these subtasks simultaneously. This idea has a long history in computer science and it has been written about extensively. However, due to the almost total lack of parallel machines, very little experimentation has been done to test the idea. Nevertheless, there is a consensus of opinion that parallel processing will be essential if computing speeds are to continue to increase. One of the strongest arguments for this point of view is the "speed-of-light" argument. This states that the speed of light limits the rate at which information can be transmitted and, by implication, the rate at which a single processor can perform computations. For example, in [1] it is estimated that the speed of light limitation will prevent a single sequential processor from exceeding 1 GFLOPS (giga floating point operations per second). Such arguments coupled with the improvements that have occurred in hardware—dramatic increases in levels of integration and equally dramatic reductions in costs—have finally made parallel processing appear to be a practical method of achieving improved rates of computation, and in the past few years relatively inexpensive parallel commercial systems have begun to appear.

These recent parallel systems fall into three classes. The first class are conventional multiprocessors with at most a few dozen processors connected to a shared memory over a high speed bus. Examples are the Sequent Balance 8000 [2] and the Encore Multimax [3]. They have been used, until now, in a multiprogramming mode where complete programs execute sequentially on a single processor—there is no parallelism within each program. They are intended for a timeshared central computing facility where there is a job stream of logically independent processes being created by the users of the facility. The second class of systems represent a more radical departure from the traditional sequential processor. They are intended for very large scale programs in which it is desirable to execute subprograms in parallel and they have hundreds of processors with their own local memory that are connected in some regular array. Examples are the Intel iPSC (128 processors) [4], the Ametek 14/n (256 processors) [5], the NCUBE/ten (1024 processors) [6], Floating Point System's T series (16 thousand processors) [7], and the Connection Machine (65 thousand processors) [8]. The numbers in parentheses are the largest configurations possible. All of these machines are interconnected as a hypercube array and, with the exception of the Connection Machine, their processors are general purpose computers that can operate independently (in MIMD mode). The processors of the Connection Machine operate in lockstep (SIMD mode) with each processor obeying the same instruction. The size of the processors is much smaller than those in the other machines—they are designed to operate on data one bit at a time rather than 16 or 32 bits. The third class of systems are massively parallel but, unlike the hypercube processors, provide a connection from each processor to a large multiport shared memory. Examples are the BBN Butterfly [9] and the RP3, an experimental

This work was supported in part by Army Research Office grant DAAG29-84-K-0070.

machine being built by IBM [10]. The key feature of these machines is the interconnection network (ICN) that connects the processors to the shared memory. Both machines use variations on the Omega network first proposed by Lawrie [11].

An important point to note for this discussion is that the overwhelming majority of commercial massively parallel architectures are hypercube machines. In the next section we will explain the reasons that may have led to this situation. In Section 3 we will move onto a discussion of image pattern recognition algorithms and how they can be executed by hypercube machines. This is followed in Section 4 by a detailed example that illustrates how these machines are currently programmed. Section 5 discusses some performance figures followed by a few concluding remarks.

2. Why Hypercubes?

There have been numerous proposals for interconnecting large numbers of processors together that pre-date the commercial machines mentioned above. Examples include 2-dimensional meshes, pyramids, and numerous ICN-based machines. Indeed, the ICN has been the subject of a considerable amount of research in its own right (see [12]). A number of experimental machines have been or are being built to test out these ideas. Some notable examples are the Illiac IV [13] (mesh connected), the Goodyear MPP [14] (14 thousand processors, mesh connected), the pyramid machines of Tanimoto [15], the Purdue PASM [16] (ICN-based), and the NYU Ultracomputer [17] (this served as the prototype of the RP3).

Given the large variety of proposals it is interesting to note that the overwhelming majority of first generation commercial massively parallel machines have been hypercube connected. There are a number of good reasons for this, but before discussing them we will clarify what is meant by a hypercube multiprocessor.

A hypercube is a generalization of the (3-dimensional) cube to spaces with higher dimension (hyperspaces). Just as a 3-dimensional cube has 2^3 corners (vertices), so an n-dimensional cube has 2^n corners. Similarly, each corner of a (3-)cube has 3 edges connected to it, and each corner of an n-cube has n edges connected to it. Hypercube multiprocessors take this simple geometry and use it to define the interconnection pattern among the processors: processors are placed at the vertices of the cube and are connected by links along the edges. For the sake of consistency, lower dimensional cubes (squares, lines, and points) are also regarded as hypercubes (strictly speaking they are hypocubes). Thus the conventional uniprocessor is a 0-cube. In general, an $(n+1)$-dimensional cube can be constructed by duplicating an n-cube and then connecting each vertex in the original cube with its duplicate in the duplicated cube (see Figure 1).

There are several attractive features of the hypercube geometry. First, the geometry is "isotropic" in the sense that it appears the same from each processor. There are no edges or borders where processors may need to be treated as special cases. This isotropic property can even be extended to include I/O if, as is the case with the NCUBE machines, each processor has a separate I/O channel. Second, the geometry provides a manageable trade-off between two extremes. On the one hand one would like a completely connected geometry to reduce communications time. Unfortunately, this requires that a multiprocessor with N processors would need $N^2/2$ communication links to interconnect the processors and that each processor would be connected to N-1 links. For a system with 1024 processors over a million links would be needed and each processor would have to manage 1024. Even if the links were simple bit-serial channels the system would be dominated by the interconnect and by the power required to run it. On the other hand one would like a small number of links between processor to keep the system cost within reason. The simplest is a ring—each processor has only two links to deal with. Unfortunately, the communication time grows linearly with N and in the case of a system with 1024 processors some messages must travel over 511 links before reaching their destination. The hypercube strikes a balance between the high-cost/high-connectivity of a completely connected geometry and the low-cost/low-connectivity of a ring geometry. It guarantees that any two processors are no more than n $(=\log_2 N)$ links apart, and that each processor is connected to only n links. For a system with 1024 processors this means no more than 10 links separate processors and that only 10 links need to be managed by each processor. It is interesting to compare this to the other two interconnection geometries that have been widely studied—2-dimensional meshes and pyramids. Figure 2 shows mesh and pyramid geometries. For a small class of problems the 2-dimensional mesh is ideal but, in general, interprocessor communications can be a limitation—in the worst case, communications between processors must traverse $2\sqrt{N}-1$ links and the small fixed number of links at each processor (≤ 4) is a source of congestion. The pyramid is better in many respects—the communication delay between processors is logarithmic as it is with the cubes, but the fixed number of links connected to each processor (≤ 9) is also a source of congestion.

In addition to the balance between connectivity and cost offered by the hypercube geometry, routing messages between processors can also be accomplished in a straightforward fashion provided some thought is given to establishing the unique ids or addresses of the processors within the hypercube array. This was done in Figure 1 during the construction procedure (the ids are shown as binary numbers). The id labeling proceeds as follows. 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,

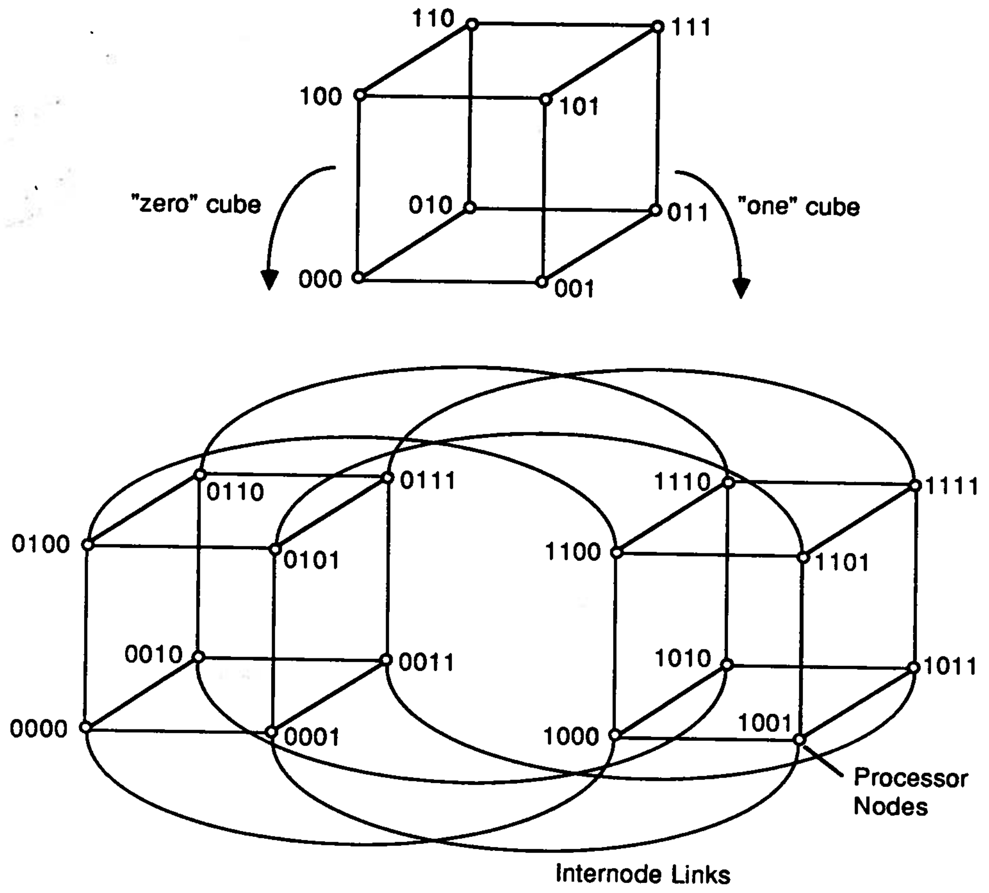

Figure 1: Constructing a 4-dimensional cube from two 3-dimensional cubes.

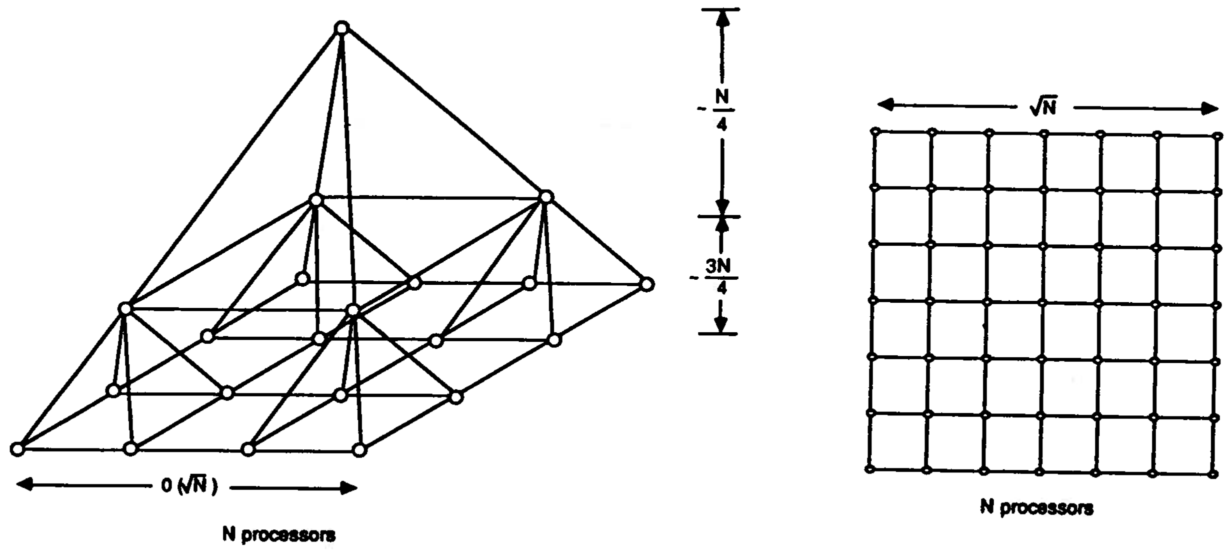

Figure 2: Mesh and pyramid geometries.

prefix the node labels in one of the $(n-1)$-cubes with a 0 so they are of the form $0zz \cdots zz$. Second, prefix the node labels in the other $(n-1)$-cube with a 1 so they are of the form $1zz \cdots zz$. Finally, connect the two $(n-1)$-cubes with communication links between nodes that have labels differing only in their most significant bit. The resulting labeling connects each processor to those n processors whose ids differ in just one bit position. Given this property it is easy to determine the route from processor $s_{n-1} \cdots s_0$ to processor $d_{n-1} \cdots d_0$. First, take the bit-wise exclusive-OR of these two ids, $z_{n-1} \cdots z_0$. (The positions where the z's are 1 will indicate the route.) Scan $z_{n-1} \cdots z_0$ from left to right to find the first integer i such that $z_i = 1$. The first step of the route then goes from $s_{n-1} \cdots s_i \cdots s_0$ to $s_{n-1} \cdots \overline{s}_i \cdots s_0$. The scan is continued to find the next integer j such that $z_j = 1$. The next step of the route then goes from $s_{n-1} \cdots \overline{s}_i \cdots s_j \cdots s_0$ to $s_{n-1} \cdots \overline{s}_i \cdots \overline{s}_j \cdots s_0$. This procedure is repeated until the scan reaches z_0. It is easy to show that at this point processor $d_{n-1} \cdots d_0$ has been reached.

The above points undoubtedly made hypercubes attractive to commercial interests wishing to produce massively parallel processors, but there was another ingredient important to their popularity: researchers at CalTech demonstrated that a hypercube could easily be built with off-the-shelf microprocessor components [18,19]. Many of the other proposals for massively parallel machines require complex custom chips if they are not to be constructed from large numbers of simple ICs. This is a more serious restriction than it might first appear, if one considers that a component count of more than a few tens of thousands of ICs puts air-cooled systems at the outer limits of reliability (regardless of the complexity of the subsystem within the IC).

3. Algorithms for Image Pattern Recognition

Image pattern recognition proceeds in three major stages [20]. In the first stage, referred to as the *low level processing* stage, the image is enhanced and features in the image are detected. Algorithms that perform these functions are referred to as *low level algorithms*. In the second stage, referred to as the *intermediate level processing* stage, the features detected in the first stage are extracted from the image. Algorithms that perform such function are referred to *intermediate level algorithms*. In the third and final stage, referred to as the high level processing stage, the extracted features from the image in the second stage are classified and analyzed. Algorithms that perform these operations are referred to as *high level algorithms*.

For the remainder of this section we will discuss each of the three stages of image pattern recognition in more detail, and show how they can be executed on a hypercube. The discussion focuses on the low and intermediate level stage. for which efficient algorithms are usually known. Executing high level algorithms on hypercubes, or massively parallel machines in general, is an area that is only just beginning to receive attention.

3.1. Low Level Processing

Low level processing generally involves enhancement, restoration, noise removal and feature detection operations. This stage of processing is characterized by:

1. The image is represented as a two dimensional array of pixels. Each pixel represents the gray level value at it coordinates. This representation of the image, which is referred to as a *low level representation,* can be characterized by the following:
   - (a) The spatial information regarding pixel location is implicit in the representation of the array. That is, no explicit address information is stored.
   - (b) The classification of pixels is explicit in the representation through the values of the pixels in the array.
   - (c) The features in the image are implicit in the representation through the relationships among the pixels.

2. A set of deterministic operations is applied to each and every pixel in the image. As a consequence, equal size areas of the image take equal amounts of processing times. Furthermore, because of the deterministic nature of the operations, no data dependent decisions are made during run-time. These operations basically are of four types:
   - (a) *Input/Output Operations:* for human interaction and image storage/retrieval.
   - (b) *Context-Free Operations:* point-wise operations on single or multiple images. Examples include: histogram generation and general tonal mapping.
   - (c) *Context-Dependent Operations:* the value of a pixel is modified based on the values of the pixels in its context or neighborhood. Hence, these operations are also referred to as neighborhood operations. These operations form the majority of low level operations. Examples include: edge and line detection noise removal and filtering.
   - (d) *Global Transformation Operations:* in which the image is transformed by Fourier, Cosine, Hadamard and similar transforms that combine data from the entire image.

3.1.1. Convolving with an FIR function

Convolving an the input image with a Finite Impulse Response (FIR) function to give the output image is a common lower level operation. The FIR function is a $m \times m$ matrix $K(\alpha,\beta)$ of constant coefficients (m is usually odd). This FIR matrix is also referred to as the *kernel*. The kernel is moved across the image in one pixel steps to implement the convolution function. In each step, the pixel $P(i,j)$ coinciding with the center of the kernel is replaced by $Q(i,j)$, such that,

$$Q(i,j) = \sum_{\alpha=u}^{\alpha=v} \sum_{\beta=u}^{\beta=v} P(i+\alpha,j+\beta)K(\alpha,\beta)$$

where $u = -\lfloor \frac{m}{2} \rfloor$ and $v = \lfloor \frac{m}{2} \rfloor$.

Convolving the image with a FIR function falls into the type of context dependent operations. The kernel $K(\alpha,\beta)$ can be viewed as coinciding with the $m \times m$ neighborhood for the pixel $P(i,j)$. The value of $P(i,j)$ is then replaced by a new value, $Q(i,j)$, which is a function, linear in this case, of the pixels in the neighborhood (see Figure 3).

A typical example of an algorithm involving the convolution of the image with FIR functions is the Sobel edge detection algorithm. The image is convolved with the each of the two FIR kernels shown in Figure 4 (an integer approximation to the exact kernels is shown). The result of the convolution are the two images e_x and e_y where $e_x(i,j)$ is an $M \times M$ array of x-direction edge (gradient) strengths, and $e_y(i,j)$ is an $M \times M$ array of y-direction edge (gradient) strengths. These two arrays are then combined to form a combined edge strength array, E, and an edge direction array, θ where

$$E(i,j) = \sqrt{e_x^2 + e_y^2}$$

and

$$\theta(i,j) = \tan^{-1}\left(\frac{e_x}{e_y}\right) - \frac{\pi}{2} \quad .$$

As noted earlier, for low level algorithms, equal size areas of the image take equal amounts of processing. This is certainly the case when convolving with an FIR function. Therefore, the natural approach to executing these algo-

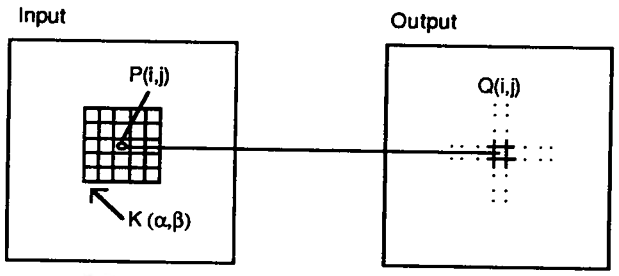

Figure 3: Convolving with an FIR function.

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

Δ_x

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

Δ_y

Figure 4: The Sobel edge detector kernels.

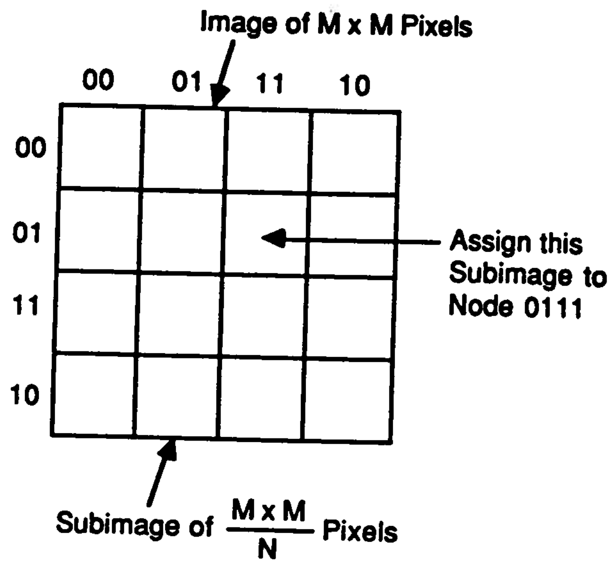
Image of M x M Pixels
00
01
11
10
00
01
11
10
Assign this Subimage to Node 0111
Subimage of $\frac{M \times M}{N}$ Pixels

Figure 5: Partitioning the image.

rithms on hypercubes is to partition the image into subimages of equal sizes, and assign each subimage to a node separate processor. A natural assignment for the hypercube is to partition the $M \times M$ image into a Gray code of $m \times m$ subimages similar to an n-dimensional Karnaugh map, and then to place each subimage with its like numbered processor (see Section 4 for further details).

An example of this assignment is shown in Figure 5 for a 4-cube. This method for partitioning the image and assigning the partitions to processors guarantees that adjacent subimages are in adjacent processors.

In general, low level algorithms can be implemented as identical programs running in parallel, each in a separate processor. The only potential contributor to inefficiency is the communication overhead that results from the need to exchange data around the edges of the subimage to implement neighborhood operations. This is depicted in Figure 6. It shows a subimage in processor A and the data that has to be moved from adjacent processors. The number of pixels that has to be transferred is roughly $\frac{2Mm}{\sqrt{N}}$ if n is even, and $\frac{3Mm}{\sqrt{2N}}$ if n is odd. As before $m \times m$ is the size of the kernel, $M \times M$ is the size of the subimage, and $N = 2^n$ is the number of processors. The communication time necessary to move the pixels is proportional to their number. In the case of some hypercube architectures the interprocessor data communications are performed as DMA operations and can be completely overlapped with processing. Of course, for large kernels and small subimages a point can be reached where overlap is impossible and communication times start to dominate.

Convolve or Transform?

When the convolution kernel K becomes large enough, it is computationally more efficient to perform the convolution by multiplying in the frequency domain. To do this we make use of the convolution theorem:

$$K * P = F^{-1}(F(K) \times F(P)) \ ,$$

where K and P are the kernel and the image to be convolved respectively, $*$ denotes the convolution operation, F is the Discrete Fourier Transform (DFT), and F^{-1} is the inverse DFT. It can be shown, under certain conditions, that it is more efficient to work in the frequency domain for a 512×512 image if m is greater than about 10 [20].

The same data layout as that shown in Figure 5 is also appropriate for calculating the DFT, for working in the frequency domain, and for calculating the inverse DFT to get back to the spatial domain. The DFT and its inverse require data to be communicated between subcubes, i.e., between adjacent regions in Figure 5. These, of course, are in adjacent processors.

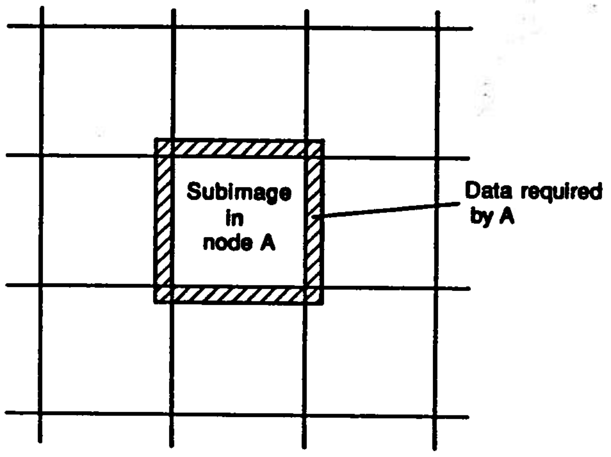

Figure 6: Data needed by a subimage in processor A.

3.2. Intermediate Level Processing

The stage of intermediate level processing involves the extraction of features from the enhanced image produced by low level processing. The extracted features are then represented in a more convenient data structure to facilitate their high level processing. Hence, this stage is basically a transducer stage between the low level and the high level processing stages. This stage of processing has the following general characteristics:

1. The representation of the image in the input to the processing is a low level representation.
2. The representation of the image in the output is a set of attributes that explicitly describe the features in the image. That is, the low level representation of the image at the input, in which the features in the image are not explicit, is transformed into another representation in which features are explicitly and, hence, more conveniently represented. In general, it is difficult to characterize the data structure representation at the output of intermediate level processing any further. This is because efficient and convenient representations of different features require different data structures. Furthermore, different algorithms may elect to represent the same feature in different ways depending on the use of the feature in the algorithm. However, the following can be said about that representation:
   (a) The spatial information regarding the location of the pixels is explicit in the representation.
   (b) The representation of a pixel value is generally restricted to a small set of values indicating the membership of the pixel in a feature set.
3. The set of operations applied to the pixels of the image generally depends on the value of the pixel and/or its context. As a consequence, equal size areas of the image take different amounts of processing times.

3.2.1. Edge Following Algorithms

Edge detection algorithms similar to the Sobel operation are usually followed by an edge thinning algorithm such as non-maximal suppression that yields one pixel wide edges [20]. Most thinning algorithms fall into the category of intermediate level algorithms. Typically the output of a thinning operation is not organized in any way that facilitates further processing. Furthermore, there are usually a large number of pixels that do not represent any edge points and, hence, are of no further interest. The edge following algorithm is therefore used to extract the edge points and organize them in the form of connected boundary segments. It also discards pixels that are not parts of these boundary segments.

The edge following algorithm works in several steps. It starts at any edge pixel in the image and then follows the boundary by moving from one edge pixel to its neighboring one. If at any pixel the edge strength is less than $\frac{T}{2}$, where T is a predefined threshold, then that pixel is discarded. If the edge strength is greater than T then the pixel is added to the boundary segment. Finally, if the edge strength at the pixel falls between $\frac{T}{2}$, and T, then the pixel is added to the boundary only if its predecessor pixel is already on the boundary. This is depicted in Figure 7. The

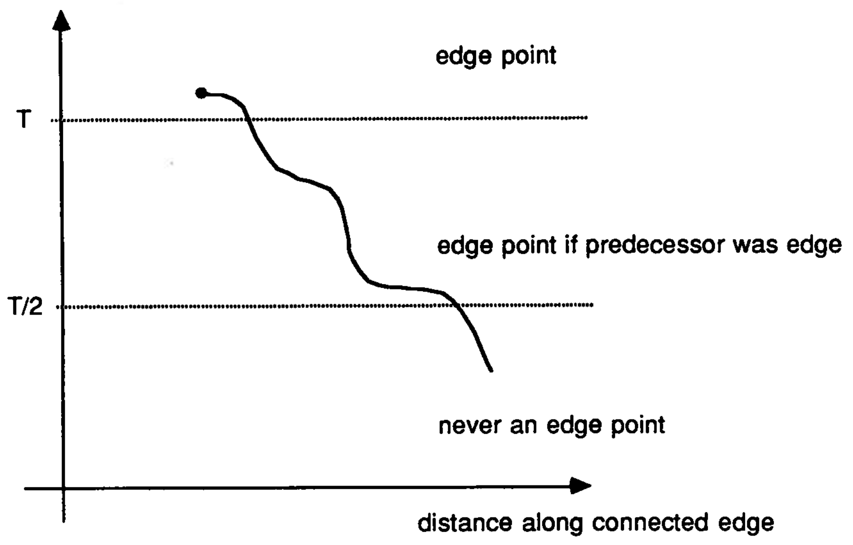
edge point
T
edge point if predecessor was edge
T/2
never an edge point
distance along connected edge

Figure 7: The edge following algorithm.

above process is repeated till there are no edge pixels that can be added to the boundary segment. Then the edge points that form the boundary segment are removed from the image and represented as a linked list of edge pixels. Each link in the list has information regarding the edge strength and direction at the pixel as well as the coordinates of the pixel. The above procedure is then repeated starting at another edge point in the image. The edge following algorithm terminates when there are no more edge points in the image.

Hence, the output of the algorithm is a set of linked lists each describing a boundary segment in the image. This illustrates the characteristic of intermediate level algorithms. The features in the image, in this case edge pixels, are extracted from it and re-represented along with explicit address information in another data structure.

Although equal size areas of the image take different amounts of processing time for intermediate level algorithms, the data assignment of Figure 5—equal subimages assigned to each processor—is still a good idea. There are two reasons. First, the low level algorithms leave the data assigned this way. Second, the difference in the amount of processing required by different subimages is "on average" quite small if the subimages are larger than a few hundred pixels [21]. Typically, the worst case difference in the work that needs to be done on the subimages is a lot less than that required to reorganize the data to achieve load balancing.

3.3. High Level Processing

The objective of high level processing is generally the recognition of objects in the image. This is done through analysis, classification and identification of features extracted from the image during earlier low and intermediate level processing. Hence, high level processing may be characterized by symbolic manipulation of feature lists that aim at the identification of objects in the image.

High level processing and high level algorithms are, in general, less well understood than low level and intermediate level processing and algorithms. This is basically attributed to:

1. High level algorithms generally involve symbolic processing of feature lists and/or employ techniques from several areas such as calculus, graph theory, differential geometry, category theory, logic and artificial intelligence. These techniques possess very diverse characteristics.
2. There is no uniform structure for the representation of the image in high level processing. In fact, high level algorithms vary considerably in this aspect. This is contrary to low level algorithms, for example, in which the representation of the image data is very uniform (a 2-dimensional array of pixels).

As we noted earlier, research is only just beginning on high level algorithm design for massively parallel machines, and for hypercubes in particular. This is due in large part to the diversity of algorithm styles that are found in this stage of image pattern recognition. Therefore, few conclusions can be drawn about the suitability of hypercubes for high level algorithms.

4. Detailed Example

In this section we will illustrate the single code model of programming commonly used in hypercube machines. In the single code model each processor in the hypercube array executes its own copy of the same program. This is not the same as the SIMD model where each processor executes the same instruction in lockstep, because the single code model allows different processors to follow different execution paths through the program. Processor dependent execution can be achieved by operating system calls that return the processor id. This can be used to influence the execution path through the code.

Our example is a program to perform the Sobel edge detection of Section 3.1.1. It assumes the kernels of Figure 4. The program runs on an NCUBE/six. The processors used in the hypercube array of the NCUBE/six are similar in performance to a VAX 11/780 with a floating point accelerator. Its instruction set is similar too. The "six" refers to the fact that it is a 6-cube (64 processors). Each processor has 128 K-bytes of memory and 11 bi-directional bit-serial channels for inter-processor communications. Ten of the channels can be used to build systems as large as a 10-cube (1024 processors—NCUBE/ten). The eleventh channel can be used for I/O.

The first step is to map the input image data onto the processors of the n-dimensional hypercube. The host program reads an $M \times M$ image file. The image is then partitioned into a Gray coded tessellation of $k_1 \times k_2$ subimages similar to an n-dimensional Karnaugh map (as shown in Figure 5). Each subimage is then sent to its like numbered processor in the n-dimensional hypercube array. In the case in where n is even, the subimages are square and $k_1 = k_2 = k = M \times 2^{-\frac{n}{2}}$. In the case in where n is odd, the subimages are rectangular, and $k_1 = M \times 2^{-\frac{(n-1)}{2}}$ and $k_2 = M \times 2^{-\frac{(n+1)}{2}}$. In either case, the subimages are of equal size, and hence, the load on the processors is balanced.

The pixels that form the interior of the subimage do not require pixels from subimages in other processors to implement the convolution with the Sobel kernels (Figure 6). Those pixels can be processed immediately. However, pixels that form the exterior (or borders) of the subimage do require pixels from subimages in other processors, and hence, can not be processed until those pixels are obtained. Consequently, the processors must exchange border pixels in order to complete the algorithm. The method of Figure 5 for mapping the data onto the hypercube processors assures the minimum communication distance between each processors and its 8 neighbors: each processor has to communicate with four processors that are one link away (four sides of pixels) and four processors that are two links away (four corner pixels).

In order to minimize the communication overhead, the necessary communications among the processors is overlapped with computations. First, each processors initiates the communication by sending its border data to its 8 neighbors. Each processors then processes the interior of the subimage while the communication of the 8 messages is in progress. After processing the interior pixels is over, each processor reads 8 messages coming from its 8 neighbors which contained data necessary to process the exterior pixels. After the processing of the exterior, the final step is to send the output data back to the host.

A part of the program that implements the above algorithm is shown in Figure 8. It is written in Fortran 77. It shows the code for implementing the communication between a processor and its 8 neighbors and the code for processing the interior of the subimage. A copy of the program runs on each processor (single code model). The eight segments of the border data are loaded earlier in the program into the arrays *cb1* through *cb8*, which act as communication buffers. The function *nwrite* is then used to send each buffer to its destination node. The array *np* contains the processor ids of each processor's 8 neighbors. The variable *xbcnt* is k_1 as defined above, *ybcnt* is k_2, and *ebcnt* is 1. The type of the messages is defined by the variable *msg$*data. The *nwrite* function initiates the communication and returns when the data is loaded into local buffers in the operating system. This is a DMA activity that proceeds in the background while the processing of the interior of the image is performed. The code shows the implementation of the convolution with the kernels of Figure 4. The variables *ex* and *ey* are the edge strengths in the x and y directions respectively. The edge strength and the edge direction are computed. The direction of the edge is a binary variable indicating whether the edge is more in the x direction or more in the y direction. Finally, the border data is read using the *read* function into the communication buffers *cb1* through *cb8*. The processing of the exterior of the subimage proceeds in a similar way to that of the interior.

```
C
C Send the border data to neighboring nodes. Avoid sending data
C from a node to itself.
C
        icflag=.false.
        if (npy .eq. 1) go to 860
        istatus=nwrite(cb1,xbcnt,np(1),msg$data,icflag)
        istatus=nwrite(cb3,xbcnt,np(3),msg$data,icflag)
860     if (npx .eq. 1) go to 960
        istatus=nwrite(cb2,ybcnt,np(2),msg$data,icflag)
        istatus=nwrite(cb4,ybcnt,np(4),msg$data,icflag)
        if (npy .eq. 1) go to 960
        istatus=nwrite(cb5,ebcnt,np(5),msg$data,icflag)
        istatus=nwrite(cb6,ebcnt,np(6),msg$data,icflag)
        istatus=nwrite(cb7,ebcnt,np(7),msg$data,icflag)
        istatus=nwrite(cb8,ebcnt,np(8),msg$data,icflag)
960     continue
C
C Process the interior of the image.
C
        do 500 i=2,xl-1
        do 500 j=2,yl-1
        temp=(i-1)*yl+j
        temp1=temp-yl
        temp2=temp+yl
        ex=databuffer(temp1-1)+2*databuffer(temp1)+databuffer(temp1+1)-
        1  databuffer(temp2-1)-2*databuffer(temp2)-databuffer(temp2+1)
        ey=databuffer(temp1+1)+2*databuffer(temp+1)+databuffer(temp2+1)-
        2  databuffer(temp1-1)-2*databuffer(temp-1)-databuffer(temp2-1)
         rtemp=sqrt(ex*ex+ey*ey)
        if(rtemp.gt.255.0) rtemp=255.0
        str(temp)=rtemp
        if(abs(ex).gt.abs(ey)) then
                                  dir(temp)=0
                                else
                                  dir(temp)=1
        endif
500     continue
C
C Read the borders
C
750     icflag=.false.
```

Figure 8: A single code program for Sobel edge detection.
(Continued on next page)

```
        if (npy .eq. 1) go to 751
        istatus=nread(cb1,xbcnt,np(1),msg$data,icflag)
        istatus=nread(cb3,xbcnt,np(3),msg$data,icflag)
751     if (npx .eq. 1) go to 760
        istatus=nread(cb2,ybcnt,np(2),msg$data,icflag)
        istatus=nread(cb4,ybcnt,np(4),msg$data,icflag)
        if (npy .eq. 1) go to 760
        istatus=nread(cb5,ebcnt,np(5),msg$data,icflag)
        istatus=nread(cb6,ebcnt,np(6),msg$data,icflag)
        istatus=nread(cb7,ebcnt,np(7),msg$data,icflag)
        istatus=nread(cb8,ebcnt,np(8),msg$data,icflag)
760     continue
C
C Process the exterior of the image.
C
        do 600 i=2,xl-1
C
C Border 1.
C
        temp=(i-1)*yl+1
        temp1=temp-yl
        temp2=temp+yl
        ex=cb1(i-1)+2*databuffer(temp1)+databuffer(temp1+1)-
        1  cb1(i+1)-2*databuffer(temp2)-databuffer(temp2+1)
        ey=databuffer(temp1+1)+2*databuffer(temp+1)+databuffer(temp2+1)-
        2  cb1(i-1)-2*cb1(i)-cb1(i+1)
        rtemp=sqrt(ex*ex+ey*ey)
        if(rtemp.gt.255.0) rtemp=255.0
        str(temp)=rtemp
        if(abs(ex).gt.abs(ey)) then
                                  dir(temp)=0
                                else
                                  dir(temp)=1
        endif
```

Figure 8: Continued

5. Performance Figures and Conclusion

Most hypercubes have been in use for less than a year, therefore there is limited data on their performance. However, some performance figures have been obtained by the author on an NCUBE/six hypercube multiprocessor. The results, so far, are typical of what one might expect from hypercube machines in general, if allowances are made for differences in the implementation of the cpu and inter-processor connections. We will summarize the results next.

We noted earlier that the performance of each processor used in the hypercube array was similar to a VAX 11/780 with a floating point accelerator. Our experiments show that a single 8 MHz NCUBE processor can execute 999 Fortran Dhrystones per second compared to 741 for the VAX, and 381,000 Fortran Whetstone instructions per second compared to 426,000 for the VAX [6]. The Dhrystone and Whetstone are popular synthetic benchmarks. The Dhrystone we used was a Fortran version transliterated from the original Ada program (100 Ada source statements); it provides a measure of performance for typical systems programming applications and contains no floating point operations. In contrast, the Whetstone provides a measure of performance for scientific applications that rely heavily on double precision floating point arithmetic (64-bit). The Linpack program was also run on the NCUBE/six. It solves systems of dense linear equations. We copied the parallel version of Moler [22]. The performance for a variety of cube sizes up to a 6-cube is shown in Table I (double precision arithmetic was used). The right hand column shows the best figures; they result from coding the time critical parts of the code in assembler. It can be seen that for a 6-cube solving a system with 477 unknowns the machine averages 3.077 MFLOPS (million floating point operations per second). This is quite a bit better than the similarly priced Alliant FX/1, and is an order of magnitude better in MFLOPS/$1000 than a Cray X-MP2 (30 MFLOPS for $10 M) [23]. Work is underway to measure the performance of a full 1024 processor system.

Perhaps the type of applications that the NCUBE is best suited to are those that have a high degree of parallelism ("embarrassingly" parallel) and that require a large amount of floating point arithmetic. Monte Carlo simulations of physical events are a typical example. We have run a Monte Carlo simulation of photon transport in a fusion plasma on a 7 MHz NCUBE/six and also on a single processor Cray X-MP [24]. The NCUBE/six took only about 30% longer. This represents a dramatic two orders of magnitude advantage in MFLOP/$1000 over the Cray. Additional experiments suggest that a 10-cube should show a factor of ten improvement in speed over the 6-cube, making the hypercube difficult to beat for this class of embarrassingly parallel problems.

Low level image pattern recognition algorithms are also embarrassingly parallel, as the example of the Sobel edge detector illustrates. So are many intermediate level algorithms. We have programmed a complete thick film inspection (TFI) problem to run on an NCUBE [25]. Thick film circuits are a network of conductors and dielectrics printed on a ceramic substrate. Image pattern recognition techniques are used to check geometric conformity to an ideal

Number of processors	Number of unknowns	MFLOPS Fortran 77	MFLOPS Assembler
1	69	0.039	0.099
8	188	0.264	0.661
16	259	0.482	1.156
32	354	0.850	1.938
64	477	1.432	3.077

Table I. Linpack benchmark on an NCUBE/six.

template. The algorithms involved are mainly low and intermediate level algorithms. One phase of the TFI is edge detection. In our implementation this involved a Sobel operation followed by a non-maximal suppression step to yield pixel wide edges. An edge following step with hysteresis concluded the edge detection phase. The Sobel operation was implemented by the program of Figure 8. The non-maximal suppression was implemented as a non-linear FIR following the implementation ideas of Section 3 (details can be found in [20]). And the edge following was also implemented as described in Section 3. The data layout followed that of Figure 5. For a frame of 512$\times$512 bytes the NCUBE/six completed the edge detection phase in about 800 milliseconds. This is faster than most special purpose video array processors, although they are usually more cost effective. This is principally due to the fact that the NCUBE is geared to floating point arithmetic.

It can be seen from the preliminary results presented above that hypercubes show some promise as parallel processors. For the special case of image pattern recognition algorithms, hypercubes appear well suited to the low and intermediate level algorithms. Due to the absence of floating point arithmetic in these algorithms, a more cost effective hypercube than the one we conducted our experiments on would be a machine with a larger number of simpler cpus each with instruction sets oriented towards byte handling, rather than floating point arithmetic. However, it is not clear that such a machine is worth building solely for image pattern recognition algorithms because the trend over the next decade is likely to be away from algorithms that contain a lot of simple integer arithmetic to ones that have significantly more floating point. The main reason for this is that the sensors (cameras) are becoming much larger, allowing more precision. The need for floating point is already apparent in stereo matching when the images are acquired by metric cameras. The basic algorithm fits the mold of a low level image pattern recognition algorithm of Section 3, but the images are typically 8000×8000 pixels and the stereo pairs subtend angles of less than a degree. To control the calculation so that intermediate results do not go out of range while at the same time precision is retained can only be done with a floating point representation.

References

[1]. P.J. Denning, "Parallel computing and the evolution," *Comm. ACM,* vol. 29, no. 12, Dec. 86, pp. 1163-1167.

[2]. G. Fielland and D. Rodgers, "32-bit computer system shares load equally among up to 12 processors," *Electronic Design,* Sep. 1984, pp. 152-168.

[3]. *Multimax Technical Summary,* Encore Computer Corporation, Marlboro, Mass., 01752, rev. a edition, May 1985.

[4]. *iPSC System Overview,* Order No. 175278-002, Intel Scientific Computers, Beaverton, Oregon, 1986.

[5]. *Ametek System 14/n,* Ametek Computer Research Div., Pasadena, Calif., 1985.

[6]. J.P. Hayes et al., "A microprocessor-based hypercube supercomputer," *IEEE Micro,* Oct. 1986, pp. 6-17.

[7]. J. Gustavson et al., "The architecture of a homogeneous multiprocessor," *Proc. of the 1986 Intl. Conf. on Parallel Proc.,* Aug. 1986, pp. 649-652.

[8]. W.D. Hillis, *The Connection Machine,* MIT Press, Cambridge, Mass., 1985.

[9]. W. Crowther et al., "Performance measurements on a 128-node butterfly parallel processor," *1985 Intl Conf. on Parallel Proc.,* Aug. 1985, pp. 531-540.

[10]. G.F. Phister et al., "The IBM research parallel processor prototype (RP3): introduction and architecture," *Proc. of the 1985 Intl Conf. on Parallel Proc.,* Aug. 1985, pp. 764-771.

[11]. D. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. on Computers,* vol. C-24, Dec. 1975, pp. 1145-1155.

[12]. H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexin Books, Lexington, Mass., 1985.

[13]. G.H. Barnes et al., "The Illiac IV computer," *IEEE Trans. on Computers,* vol. C-17, Aug. 1968, pp. 746-757.

[14]. J.P. Potter (ed), *The Massively Parallel Processor,* Cambridge, Mass., MIT Press, 1985.

[15]. S.L. Tanimoto, "A pyramidal approach to parallel processing," *Proc. 10th Annual Intl. Symp. on Computer Architecture,* Stockholm, June 1983, pp. 372-378.

[16]. H.J. Siegel et al., "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. on Comp.,* vol. C-30, Dec. 1981, pp. 934-947.

[17]. A. Gottlieb et al., "The NYU Ultracomputer—designing a MIMD, shared memory parallel machine," *IEEE Trans. on Comp.,* vol. C-32, Feb. 1983, pp. 175-189.

[18]. C.L. Seitz, "The cosmic cube," *Comm. ACM,* vol. 28, no. 1, Jan. 1985, pp. 22-33.

[19]. G. Fox, "The performance of the caltech hypercube in scientific calculations," Report CALT-68-1298, California Institute of Technology, Pasadena, Calif., Apr. 1985.

[20]. T.N. Mudge and T.S. Abdel-Rahman, "Architectures for robot vision", in *Specialized Computer Architectures for Robotics and Automation,* J. Graham (ed), Publ: Gordon and Breach, Inc., (to appear).

[21]. T.N. Mudge and T.S. Abdel-Rahman, "Efficiency of feature dependent algorithms for the parallel processing of images," *Proc. of the Intl Conf. on Parallel Proc.,* Aug. 1983, pp. 369-373.

[22]. C. Moler, "Matrix computation on distributed memory multiprocessors," technical report in preparation, Intel Scientific Computers, Beaverton, Oregon.

[23]. J.J. Dongarra, "Performance of various computers using standard linear equations software in a Fortran environment," Tech. Memo. No. 23, Math. and Computer Sci. Division, Argonne Natl. Lab., March 1985.

[24]. W.R. Martin et al., "Monte Carlo photon transport on the NCUBE," *Proc. of the 1986 Conf. on Hypercube Multiprocessors,* (to appear).

[25]. T.N. Mudge and T.S. Abdel-Rahman, "Vision algorithms for hypercube machines," *Jour. of Parallel and Distributed Computing,* (to appear).