HYPERCUBES AND PYRAMIDS

Quentin F. Stout
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109 USA

1. INTRODUCTION

Hypercube computers have recently become popular parallel computers for a variety of engineering and scientific computations. However, despite the fact that the characteristics which make them useful scientific processors also makes them efficient image processors, they have not yet been extensively used as image processing machines. This is partially due to the hardware characteristics of current hypercube computers, partially to the particular history of the groups which first built hypercubes, and partially to the fact that the image processing community did not initially realize some of the advantages of hypercubes. In this paper, their suitability for image processing will be put forth, showing that they can be viewed as competitors to, and collaborators with, mesh and pyramid computers, architectures which are often promoted as being ideal for image processing.

First some of the general graph-theoretic properties of hypercubes will be given. Second, the primary reasons for the initial interest of the engineering and scientific community, namely hypercubes' suitability for mesh calculations and as general purpose parallel computers, will be shown. Third, the ability of hypercubes to efficiently execute pyramid algorithms is shown, an ability which has not yet been taken advantage of. Fourth, currently available commercial hypercubes are examined to show some of the trends in the increasing sophistication of their implementation. Finally, a new architecture is suggested which combines hypercubes and pyramids to make machines capable of rapidly processing large images and performing image analyses from lower level image processing to higher level image understanding.

Pyramid computer means the standard model used in most of the other papers in this volume, namely, a generic processing element (PE) is connected to four neighboring PEs on the same level, a parent PE on the level above, and four children PEs on the level below. **Mesh computer** will usually mean a square 2-dimensional mesh in which a generic PE is connected to four neighbors, but in some clearly understood cases it will mean a mesh of any dimension. Hypercubes will be defined below. For positive func-

NATO ASI Series, Vol. F 25
Pyramidal Systems for Computer Vision
Edited by V. Cantoni and S. Levialdi
© Springer-Verlag Berlin Heidelberg 1986

tions f and g defined on the positive integers, the notation $f = \Omega(g)$ ("f is of order at least g") means that there are constants $C, N > 0$ such that $f(n) \geq C*g(n)$ for all $n \geq N$, and $f = \Theta(g)$ ("f is of order exactly g") means that there are constants $C, D, N > 0$ such that $D*g(n) \geq f(n) \geq C*g(n)$ for all $n \geq N$. For example, $n*\sin(n)^2 + \log_2(n) = \Omega(\log_2(n))$, and $3n + n^2 = \Theta(n^2)$.

2. HYPERCUBES AS GRAPHS

Hypercubes have long been studied by graph theorists. For integer dimension $d \geq 0$, the (binary) **hypercube of dimension *d***, denoted H_d, is as a graph of 2^d vertices with unique d-bit binary strings used as labels, where there is an edge connecting two vertices if and only their labels differ by a single bit. Often the labels will be interpreted as binary numbers. Figure 1 shows hypercubes of small dimensions. A hypercube computer is formed by placing a PE at each node, and a communication link along each edge. This makes a local, or distributed, memory machine where information is passed as messages between PEs, as opposed to a global, or shared, memory machine where information is exchanged by placing it in the global memory.

Several properties of hypercube graphs are immediately apparent. First, they are **homogeneous**, meaning that for any dimension d, given any two vertices p,q in H_d, there is a graph isomorphism of H_d onto itself which maps p onto q. To see this, let r = label(p) xor label(q) (all logical operations are performed bitwise). The mapping which maps a vertex s to the vertex labeled r xor label(s) is one such isomorphism. Homogeneity implies that all nodes can be treated equally, and in particular means that in a computer implementation it is natural to allow input/output to all nodes. It also means that if an algorithm treats a node specially (for example, if node 0 is used as the root of a tree), then by using xor the algorithm can be "translated" so that any other desired node is the special one. Pyramids and meshes are not homogenous, since the apex is unique and corners can only be mapped to other corners, but tori are homogeneous. Binary hypercubes are special types of tori.

Routing messages between nodes is particularly simple in a hypercube. A message from node p to node q must travel along at least as many edges as there are 1s in the xor of p and q's labels, and there are paths which attain this lower bound. Further, such paths are quite easy to compute dynamically: if a message destined for q is currently at p, then let i be the location of any 1 the xor of p and q's labels, and have p forward the message to the neighbor with a label differing in the i^{th} bit. Notice that there are many such paths of minimal length, which can allow for routing variations that do not increase the path length. The **diameter** of H_d, i.e., the largest distance between nodes, is d, or $\log_2$(number of nodes). In a 2-dimensional mesh the diameter is the square root of the number of nodes, while in a pyramid it is $2*\log_2$(number of nodes in the base).

Each node in H_d has **degree** d, meaning that it has d edges. In a physical implementation the degree of some nodes must be d+1 to allow communication to the outside world, so if communication is homogeneously implemented then all nodes will have degree d+1. To build hypercubes of increasing size, the degree must also increase, which makes the individual nodes slightly more complex. Meshes and pyramids of fixed dimension but increasing size do not have this problem, which at some point can become a limiting factor on the size of hypercubes. As will be shown below, current technology makes it possible to build hypercubes with thousands of nodes, so the node degree is not as serious a problem as it was in the 1970's, during which time alternatives such as cube-connected cycles [6] were suggested to alleviate this problem.

Hypercubes are eminently partitionable into smaller hypercubes. For example, H_{d+1} can be partitioned into two disjoint copies of H_d by taking one of the d+1 coordinates (bit positions in the label) and using all nodes with a 0 in that coordinate as one copy of H_d, and all nodes with a 1 in that coordinate as the other copy. More generally, it is easy to show that if one is given a collection of hypercubes and wants to embed them in non-overlapping fashion in a given hypercube H, then this can be done if and only if the total number of nodes in the collection is no more than the number of nodes in H. (Embedding means that no two nodes are mapped to the same node, and that neighbors are mapped to neighbors.) In a multiuser environment, this means that it is quite easy to allocate subcubes to different users, and there is a great deal of flexibility possible for dynamic allocation.

3. MESH AND GENERAL PURPOSE CALCULATIONS

Many scientific and engineering applications deal with data organized as a matrix where the operations performed on an entry involve nearby elements of the matrix. For example, relaxation methods for solving partial differential equations update a point based on its current value and the value of its nearest neighbors. One quite useful property of hypercubes is that meshes (matrices) of all dimensions can be embedded in them, as shown below, so a matrix can be distributed in a hypercube such that each PE can update the values of the entries it contains by using only its own entries and the entries of neighboring PEs. This miminizes communication time, and is as good as any mesh computer with the same number and type of PEs. In other problems the communication between PEs is quite irregular, in which case the hypercube is useful because it is a fairly fast general purpose message delivery system. Thus the hypercube supports problems with both regular and irregular communication patterns.

3.1 MESHES

Meshes are embedded into hypercubes by using Gray codes. Given positive integers i and k, a k-bit **Gray code** G for $0 \ldots i$ is a 1-1 mapping from $0 \ldots i$ into k-bit binary strings, where $G(j)$ and $G(j+1)$ differ by exactly 1 bit for all $0 \leq j < i$. Since the mapping is 1-1, it must be that $k \geq \log_2(i+1)$. A particularly useful collection of Gray codes are the reflexive ones G_d. For all positive integers d, G_d is an d-bit Gray code for $0 \ldots 2^d\text{-}1$, recursively given by $G_1(0) = 0$, $G_1(1) = 1$, and $G_{d+1}(j) = 0G_d(j)$ (that is, the concatenation of 0 and $G_d(j)$) for $j<2^d$, and $G_{d+1}(j) = 1G_d(2^{d+1}\text{-}1\text{-}j)$ for $2^d \leq j < 2^{d+1}$. Figure 2 shows these reflexive Gray codes for small sets. From now on only these Gray codes will be used, and their special properties, beyond those of generic Gray codes, will be used in the pyramid mappings.

A matrix $a[\,0..2^{i1}\text{-}1, 0..2^{i2}\text{-}1, \ldots, 0..2^{id}\text{-}1\,]$ can be mapped in a 1-1 fashion onto a hypercube of dimension $i1+i2+\ldots+id$ by mapping $a[j_1, j_2, \ldots, j_d]$ to the node labeled $G_{i1}(j_1)\, G_{i2}(j_2) \ldots G_{id}(j_d)$. Adjacent matrix entries differ by exactly 1 in one coordinate, so they are mapped to nodes with labels that differ by exactly 1 bit, that is, they are mapped to adjacent nodes. (Besides being easy to compute, the reflexive Gray codes have the additional property that they also preserve toroidal neighbors, where neighbors are computed in a modular fashion.) Therefore matrix operations which update an entry based on entries in a local neighborhood can be implemented so that a node need only obtain entries for nearby nodes. This has been exploited in hypercube implementations of scientific applications, and of course it applies equally well to local operations on images. Figure 3 shows a mapping of a 2-dimensional mesh onto a hypercube. Such a mapping also maps nonadjacent matrix positions to adjacent nodes, much like crumpling a piece of paper maps points far apart on the 2-dimensional paper surface to nearby points in 3-dimensional space.

If the matrix has more entries than there are nodes in the hypercube, then the above mapping must be modified. The matrix should be partitioned into equal sized rectangular blocks, where the number of blocks equals the number of nodes. The blocks themselves form a d-dimensional mesh, so they are mapped to the nodes as before. (This assumes that a node can hold an entire block.) If the blocks are of equal size then each PE will have roughly the same amount of work to do, so PEs will not sit idle waiting for others to finish. (It is only roughly the same amount of work because the calculations may not be identical at all entries, and in particular edge entries may need to be computed differently.)

Another important consideration is the ability to overlap calculations with PE/PE communication. While this would not be very important if such communication were as fast as individual calculations, in practice communication is significantly slower. This

can become the dominate factor in determining the running time, and can significantly decrease the efficiency of the parallelism. The **efficiency** of an n PE parallel machine running a program is defined to be $T_1/(n{*}T_n)$, where T_n is the time of the program and T_1 is the time of the fastest program solving the same problem on a machine with a single PE. The time for PE/PE communication does not appear in T_1, but does in T_n, which is why it decreases efficiency.

For near-neighbor operations on a matrix, overlapping computation and PE/PE communication is fairly straightforward and is commonly used whenever the hardware can support it. For example, in a 2-dimensional matrix, suppose each entry gets a new value which is a function of its current value and the values of its four nearest neighbors. (We will ignore what happens along edges of the matrix.) In each block of entries assigned to a single PE, entries along the edges need to be updated using the values of entries from neighboring PEs, and conversely, neigbhboring PEs need the values of these entries. However, entries in the center can be updated using only information currently in the PE. To overlap operations, each PE sends the values of entries on the right edge of its block to the neighboring PE holding the block to the right, sends the top edge to the PE holding the block to the top, the left edge to the left PE, and the bottom edge to the bottom PE. It is assumed that these transmissions can be performed by initiating them and then proceeding to other calculations while the transmissions are completed. While they are being completed, entries in the center are updated. If the transmissions are completed while the center entries are updated, then the edge entries can be updated without having to wait for neighboring values to arrive. The time to initiate the communications will still appear in T_n and not in T_1, but this should be much less than the time to complete communication.

3.2 GENERAL MESSAGE TRANSMISSION

One common global operation, **broadcasting**, sends data from one PE to all other PEs, and another common operation, **reporting**, sends and condenses information from all PEs to a single one. In reporting the condensation may be to sum values, taking an OR of logical values, taking the maximum, etc. Even with matrix input, these global operations may be needed to perform tasks such as determing average gray level and sending it to all PEs so they can use it to set a threshhold.

Broadcasting can be quickly performed on a hypercube using **recursive doubling**. If PE 0 has the data to be broadcasted, then at time 1 it sends the data to PE 1. (Time units are taken to be the time required to have a message sent and received, and to determine the next PE to send to.) At time 2 PE 0 sends the data to PE 2 and PE 1 sends the data to PE 3. In general, at the end of time i PEs $0 \ldots 2^i\text{-}1$ have the data, and during time unit $i+1$ each one of them sends a copy to the neighbor 2^i greater. This continues until all

PEs are informed, taking d time units for H_d. If the condensation occuring in reporting is a semigroup operation such as summing or finding a maximum, then recursive doubling can be performed in reverse (recursive halving), where each PE receiving a value combines it with its own, taking d time units to complete a report. Here the time unit includes the time to do the condensation. For both operations, since the diameter is d this is the best one could hope for, and it can be achieved even if each PE can communicate with only one other at a time. Further, by using the homogeneity properties noted above, any PE can play the role of the distinguished PE, without any loss of time (except for the cost of performing xor). In a pyramid, broadcasting or reporting is best if the apex is used as the distinguished node. If each PE can communicate with all four of its children in one time unit, then broadcasting and reporting can be done in $\log_2(n)/2$ time units, where n is the number of PEs at the base. With one-at-a-time communication and the simplest approach, the time is $2*\log_2(n)$. (Less natural approaches can reduce this.)

A far more complicated activity is to rapidly transmit messages from many PEs to "random" or "bad" destinations. For example, by using a wire-cutting argument it is easy to show that if a pyramid has a base of n PEs, and if each PE sends a message to a destination given by a uniformly generated random permutation, then the expected time to deliver all messages is $\Omega(n^{1/2})$. This occurs despite the fact that the diameter of a pyramid is $\Theta(\log_2(n))$, and in fact for this type of computation the pyramid is no better than a 2-dimensional mesh. For H_d, however, by using Bitonic sort where the message address is used as the key, any possible permutation of messages can be delivered in $\Theta(d^2)$ time, i.e., in $\Theta(\log^2(n))$ time, where n is the number of PEs. By using randomization, Valiant showed that any message permutation could be delivered in $\Theta(d)$ expected time [12], although it is not clear whether the greater complexity of this algorithm can ever beat bitonic sort on the sizes of hypercubes to be available in the foreseeable future. Further, empirically it seems that the simple strategy used on all current MIMD hypercubes works exceedingly well. This strategy is to have each PE keep a small queue of messages to be forwarded, and at each time step take the next message and forward it by using the routing algorithm mentioned in Section 2. Either a high-to-low or low-to-high scan is used to find the next 1 in the xor'ed pattern showing which coordinates need to be traversed.

4. EMBEDDING PYRAMIDS IN HYPERCUBES

The material in the previous sections has been widely used in solving various scientific and engineering problems on hypercubes. In this section we show the less well known fact that hypercubes can also efficiently execute pyramid algorithms by simulating pyramids. Unlike meshes, pyramids cannot be embedded in hypercubes so that neighbors in the pyramid are mapped to neighbors in the hypercube. This is because a

cycle proceeding from a parent to one child, then to an adjacent child, and back to the parent, has an odd number of nodes, while all cycles in a hypercube have an even number of nodes. Instead, one can ask that the pyramid be mapped into a hypercube so that neighbors in the pyramid are mapped to PEs close to each other in the hypercube.

As with meshes, the most important case is where the number of PEs in the hypercube is less than the number of pixels, i.e., less than the number of PEs in the base of the pyramid. For example, the hypercube with the largest number of PEs presently available, the Connection Machine [2], has 64K PEs, while a 512×512 image has 256K pixels, and a 1024×1024 image has 1024K pixels. As before, the best way to map the base onto the hypercube is to subdivide the base into squares, where the number of squares equals the number of PEs. Each PE will simulate its base level pyramid nodes, as well as all nodes in higher levels which sit only above the nodes in the square. For example, using H_{10} on a 512×512 image, each PE will receive a 16×16 square, so each PE will simulate the work of a subpyramid with a 16×16 base, a total of 5 levels. Just as for the base, for each of these levels, each pyramid node will either have all of its neighboring pyramid nodes at the same level in the same PE or in neighboring PEs. For pyramid nodes in these levels, the only parent/child pairs perhaps not in the same PE are the apex of the subpyramid and its parent.

For the next level up, notice that these nodes are parents of the apexes of subpyramids in 4 PEs, where the 4 PEs form a square. One of these PEs will also do the work of simulating the node at the next level. Therefore one of the PEs must send its data over two communication links to reach the parent. To pick which PE to use, notice that the reflexive Gray codes have the property that in each square exactly one of the rows and one of the columns has a least significant bit of 0 in its Gray code. If the PE in this row and column is used to simulate the parent, then it is easy to see that pyramid neighbors in this higher level are again simulated by neighboring PEs. Figure 4 shows the chosen nodes in each square for a small example. The PEs simulating this level form a subcube of dimension 2 less than the original one, so to simulate pyramid nodes at still higher levels the process is repeated within this subcube. When all nodes are simulated in this manner, no pyramid neighbors are separated by more than two communication links.

When H_{10} simulates a pyramid with a 512×512 base (a pyramid with 9 levels), PE 0 simulates nodes on all 9 levels. It simulates a total of 346 nodes, while some other PEs, such as PE 1, simulate only 341 nodes. If the pyramid algorithm is such that the amount of work is the same at each node, then PE 0 has to do only about 1% more work than any other node, so the load is well balanced.

Even if the number of hypercube PEs exceeds the number of pyramid nodes, the time used by the above scheme is approximately as good as possible if the pyramid algorithm proceeds level by level , for then no PE need simulate more than one node at a time. However, a PE simulating nodes on several levels may spend some extra time in switching from one simulation to the next, and may need extra space to store local variables of the simulated nodes. If the algorithm uses all levels simultaneously then the above scheme will have one PE simulating a number of pyramid nodes equal to the pyramid's height plus 1, which means that it can take many time units to simulate a single step of the pyramid. In this situation one can do slightly better. If the dimension of the hypercube is just large enough to exceed the number of pyramid nodes, then the hypercube has exactly twice as many PEs as there are nodes in the pyramid's base. The base should be simulated using the subcube which is the lower numbered half of the hypercube. Using the same representatives of squares as shown in Figure 4, now a parent is simulated by the PE in the upper half subcube which is connected to the representative used previously, i.e., the simulation has been moved along a coordinate. This parent is now three communication links from one of its children. If the hypercube had additional unused dimensions the process would just be repeated, moving along a new dimension for each level. Instead, note that this first level of parents does not using any PE in its half which has a 1 in its least significant bit. Therefore the least significant bit can be treated as a "new" dimension. This process of "reusing" dimensions can be continued to represent all levels of the pyramid, with each PE simulating at most one pyramid node, and with pyramid neighbors mapped to PEs at most three communication links apart.

For completeness, we note that when the number of hypercube nodes is at least twice the number of base nodes in the pyramid, it is possible to map the pyramid in a 1-1 fashion into the hypercube so that no two pyramid neighbors are more than two communication links apart. To do this, some neighbors on the same level, as well as some parent/child pairs, must be mapped to PEs two links apart. Suppose the pyramid base (level 0) is $2^b \times 2^b$, so the hypercube is H_{2b+1} . Define $E(i,j)$ to be 0 if $j = 0$, to be 00 if $j > 0$ and $i \equiv 0$ or 3 mod 4 , and 11 if $j > 0$ and $i \equiv 1$ or 2 mod 4 . Define G_d to be the empty string if $d \leq 0$. Then one such mapping is to map the node at level h , row i, column j to PE $G_{b-h}(i)\ (01)^{h/2}\ G_{b-h-1}(j \text{ div } 2)\ E(j, b-h)\ (01)^{h/2}$ if h is even, and to PE $G_{b-h-1}(i \text{ div } 2)\ E(i, b-h)\ (01)^{(h-1)/2}\ G_{b-h}(j)\ (01)^{(h+1)/2}$ if h is odd. Because of space limitations, no explanation of this formula will be given, but it is relatively straightforward to verify that it has the desired properties.

It should be noted that not all pyramid algorithms should be simulated on a hypercube. For problems in which pyramid algorithms do about the same amount of work at each level, usually the pyramid is about the best possible architecture, and other papers in this volume give examples of such algorithms. However, for some problems the best pyramid algorithms must work longer at higher levels of the pyramid [3]. For such problems, sometimes there are hypercube algorithms which utilize all of the hypercube connections and are faster than simulating a pyramid [4].

5. HYPERCUBE MACHINES

Hypercube computers were first proposed by Squire and Palais, at the University of Michigan, in 1962 [8,9]. Their study showed that, at that time, a 12-dimensional hypercube would need about 20 times as many parts as IBM's Stretch, a rather unreasonable component count. Another large paper study was performed by Sullivan et al in 1977 [10,11]. Finally, in 1983, the "Cosmic Cube", a hypercube computer with 64 nodes, was built at the California Institute of Technology [7].

In 1985 Intel Corporation delivered the first production hypercubes, the iPSC, based on the CalTech hypercubes. It has a maximum of 128 nodes, where each node uses a 80286/80287 processor and has 512 Kbytes of memory. Only a single user can use the cube at one time (thought several can use the host at once), and there is only a single I/O channel between the host processor and the hypercube. The time for PE/PE communication is quite high, requiring several milliseconds. A few months later Ametek introduced a very similar hypercube, with the primary changes being a maximum of 256 nodes, and an additional 80186 coprocessor on each board to handle communication. The extra coprocessor allows additional overlapping of communication and processing, and for messages traversing several communication links the intermediate 80286 processors need not be interupted. Because of very slow I/O between the cube and host, neither of these machines is suitable for real-time vision.

In late 1985 NCUBE introduced a hypercube with a maximum of 1024 nodes, each of which uses a proprietary processor [1]. A single chip handles all node I/O via DMA, all standard processing on 32 bit words at about 1 million instructions per second, and floating point arithmetic at about 0.5 MFLOP on single precision reals. Each PE contains only 128 Kbytes of memory. Each PE has its own I/O channel to a host board, providing a maximum system I/O rate of about 720 Mbytes/sec. A multiuser operating system is provided for the hypercube. PE/PE communication proceeds at about 1 Mbyte/sec., with the startup time heavily dependent upon the protocol used. The University of Michigan is one of the beta test sites for this machine, and we hope to implement several vision applications, making use of its high I/O rates as well as high processing rates. The proposed implementation of one application is discussed in [5].

All of the machines mentioned above are MIMD, where each PE has its own program and executes it separately (except, of course, for communication) from other PEs. Most applications use the SCMD, "Single Code, Multiple Data" mode where all PEs are executing the same program, although at any point in time they may be following different branches. A SIMD bit-serial hypercube, the Connection Machine, has been built in 1986 by Thinking Machines Corporation [2]. This is a 12-dimensional hypercube, where each node contains 16 PEs. To process image data there is a special I/O mechanism which connects all the PEs as a 256×256 mesh. Each PE contains only 4K bits of memory. While this machine will be quite useful for some vision and artificial intelligence applications, its SIMD nature forces drastic revisions in algorithms, and the small amount of memory per PE can prove debilitating.

Also in 1986, two commercial hypercubes of vector processors were announced. The Intel iPSC-VX machine is based on the iPSC, with a maximum of 64 nodes. Each node consists of a iPSC node on one board, and a vector processor on a second board, with a total of 1.5 Mbytes of memory per node. All communication and control functions are performed by the iPSC node. The FPS "T Series" machine has a maximum of 16,384 nodes. Each node uses an Inmos Transputer for communication and control, with an additional vector processor, and has 1 Mbyte of memory. Initially the machine must be programmed in Occam, using subroutine calls to utilize the vector processor. Both machines currently allow only one user on the hypercube at a time.Both of these machines have been designed for engineering and scientific computations, with the vector processing being of miniscule utility for image processing.

6. HYPER-PYRAMIDS

One can borrow some of the ideas mentioned in preceeding sections to design a machine capable of rapidly processing large images and performing a range of image processing from lower level operations to higher level ones. The hypercubes of vector processors are an attempt to provide relatively inexpensive supercomputer performance by combining the best vector processors that can be made from inexpensive, commercially available parts, with a hypercube interconnection to provide inexpensive parallelism. The Connection Machine similarly combines the simplicity of mesh computing with hypercube interconnections for general purpose message delivery. The NCUBE machine and Connection Machine emphasize the importance of high I/O rates to balance high computation rates. Further, the hypercube can rapidly simulate meshes of any dimension, slightly less rapidly simulate pyramids (since neighbors are mapped to PEs two links apart), and act as a fairly fast general message delivery facility. Finally, as other papers in this volume show, pyramid algorithms seem to be quite useful for lower levels of image processing. For higher levels they are currently less useful, and a more flexible interconnection structure may be needed. Since the higher levels are poorly understood, it seems premature to optimize a machine to perform any given algorithm, but rather one should build a machine capable of running a variety of algorithms without imposing serious bottlenecks.

This suggests a hypercube of pyramid processors, a **hyper-pyramid**. Each node will have a master processor which is responsible for communication, control, and general purpose computing, and a pyramid processor. The pyramid processor need not be a physical pyramid, but must be capable of rapidly executing pyramid algorithms. Other papers in this volume suggest such processors. The master processor, like the NCUBE processor, should be capable of supporting very fast PE/PE communication. Each node should also have a high speed I/O link to the outside world. It may be that I/O is through the master processor, or, as in the Connection Machine, through the pyramid processor. The master processors will form an MIMD hypercube. The pyramid processor may be an SIMD machine, as would likely occur if it was a physical pyramid or mesh, where each master processor controls its own pyramid processor. Or, the pyramid processor may be a raster scan processor, in which case the machine would be very similar to the Intel iPSC-VX and FPS T Series machines.

The size of pyramid that the pyramid processor operates on is primarily dictated by cost and technology tradeoffs. It may be, for example, that the machine is designed to process 512×512 images, using 64 nodes each processing a 64×64 square. To run a 9 level pyramid algorithm over the entire image, the lowest 6 levels would be performed by the pyramid processors, and the top 3 levels would be performed by the master processors. If the work at each node is the same, then more than 99.9% of the work will be done by pyramid processors. On large images, often the top few levels are not used, so it may be that the master processors would rarely be needed in this fashion. However, for higher levels of image processing they will be crucial. For example, they may work together to make some initial global decisions concerning approximate locations of objects, and then each may direct its pyramid to look for specific features to confirm or refute such decisions. Since the master processors would be general purpose processors, they will be easier to program and they permit easier alterations. The pyramid processors may be harder to program, and probably would be used primarily by calling packaged routines. Since lower level image processing is somewhat better understood, these routines need not change as frequently.

One interesting question is whether master processors specially designed for such machines will soon be commercially available. The Transputer is an attempt at such a processor, but it provides only 4 bidirectional communication links which must be multiplexed in order to provide sufficient near-neighbor connections for a hypercube. The NCUBE node processor comes closer to what is desired, but it should be possible to build even better master processors. FPS and Intel have already produced "hypercube of X" machines for X="vector processor", and this paper proposes such machines with X="pyramid processor". There are presumably other interesting choices for X, so there may eventually be enough demand to provide high-quality commercial master processors for such machines.

7. CONCLUSION

We have shown that the hypercube can perform mesh calculations as fast as a mesh computer, in that mesh neighbors are mapped to hypercube neighbors, and it can perform pyramid calculations almost as fast as a pyramid computer. Besides supporting such regular communication requirements, it provides good support for irregular communications between processors, the sort of communication which seems prevalent in artificial intelligence and higher level image processing. Thus the hypercube seems a nearly ideal architecture for image processing and understanding, particularly when the emphasis is on using one machine to do all levels of the image processing and understanding process.

However, very large hypercubes need many wires per PE, which makes them difficult and expensive to build. For performing image processing and understanding on large images, rather than build a hypercube with a PE per pixel (which would almost certainly necessitate a SIMD machine with bit-serial processors), it may be better to retain a simpler efficient structure for lower level image processing, and reserve a MIMD hypercube primarily for control and higher levels of image processing. The hyper-pyramid attempts to do this, trying to balance flexibility and capacity against cost and design complexity constraints, utilizing architectures which have been demonstrated to be feasible with current technology.

REFERENCES

1 S. Colley, J.P. Hayes, T.N. Mudge, J. Palmer, and Q.F. Stout, "Architecture of a hypercube supercomputer", *Proc. 1986 Int'l. Conf. on Parallel Processing*, to appear.

2 W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass, 1985.

3 R. Miller and Q.F. Stout, "Data movement techniques for the pyramid computer", *SIAM J. Computing*, 1986, to appear.

4 R. Miller and Q.F. Stout, "Efficient graph and picture algorithms using general data movement operations for the mesh-of-trees and hypercube networks", submitted.

5 T.N. Mudge, "Vision algorithms for hypercube machines", *Proc. 1985 Comp. Arch. for Pattern Anal. and Image Database Manag.*, pp. 225-231.

6 F.P. Preparata and J. Vuillemin, "The cube-connected cycles: a versatile network for parallel computation", *Comm. ACM* 24 (1981), pp. 300-309.

7 C.L. Seitz, "The Cosmic Cube", *Comm. ACM* (28), 1985, pp. 22-33.

8 J.S. Squire and S.M. Palais, "Physical and logical design of a highly parallel computer", Tech. note, Dept. of Elec. Eng., Univ. Michigan, Oct. 1962.

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

10 H. Sullivan and T.R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine, I", *Proc. Computer Arch. Symp.*, 1977, pp. 105-117.

11 H. Sullivan, T. R. Bashkow and D. Klappholz, "A large scale, homogeneous, fully distributed parallel machine, II", *Proc. Computer Arch. Symp*, 1977, pp. 118-124.

12 L.G. Valiant, "A scheme for parallel communication", *SIAM J. Computing* 11 (1982), pp. 350-361.

This research was partially supported by the National Science Foundation, grant DCR 8507851.

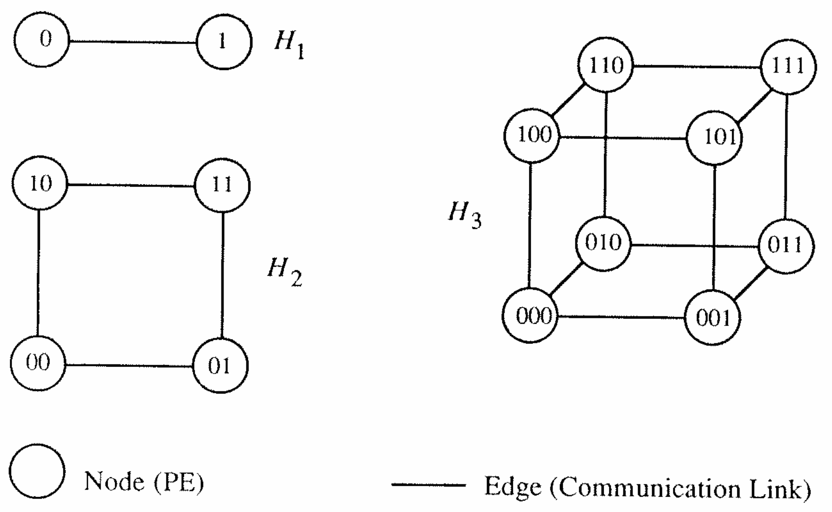

FIGURE 1. Small Hypercubes.

$$G_1:\quad \begin{array}{cc} 0 & 0 \\ 1 & 1 \end{array}$$

$$G_2:\quad \begin{array}{cc} 0 & 00 \\ 1 & 01 \\ 2 & 11 \\ 3 & 10 \end{array}$$

$$G_3:\quad \begin{array}{cc} 0 & 000 \\ 1 & 001 \\ 2 & 011 \\ 3 & 010 \\ 4 & 110 \\ 5 & 111 \\ 6 & 101 \\ 7 & 100 \end{array} \begin{array}{l} \left.\vphantom{\begin{array}{c}0\\0\\0\\0\end{array}}\right\} 0G_2 \\ \left.\vphantom{\begin{array}{c}0\\0\\0\\0\end{array}}\right\} 1G_2 \text{ reversed} \end{array}$$

FIGURE 2. Reflexive Gray Codes

G_3

G_2	000	001	011	010	110	111	101	100
00	00000	00001	00011	00010	00110	00111	00101	00100
01	01000	01001	01011	01010	01110	01111	01101	01100
11	11000	11001	11011	11010	11110	11111	11101	11100
10	10000	10001	10011	10010	10110	10111	10101	10100

FIGURE 3. Embedding a 4x8 mesh onto H_5

	000	001	011	010	110	111	101	100
000	*			*	*			*
001								
011								
010	*			*	*			*
110	*			*	*			*
111								
101								
100	*			*	*			*

FIGURE 4. Pyramid Parent PEs in Each Square

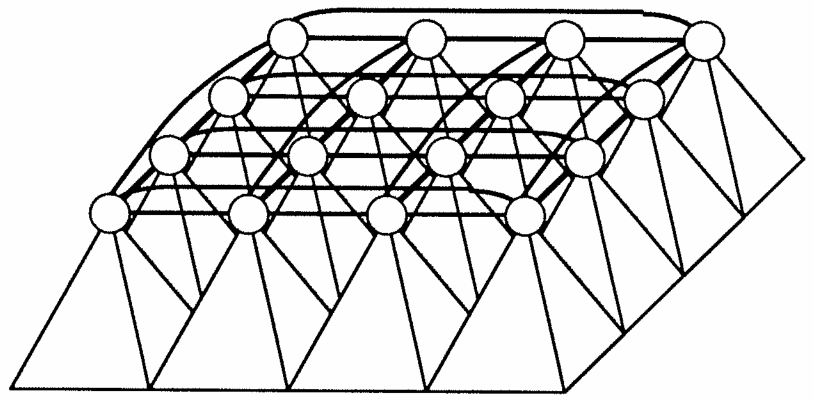

FIGURE 5. A Hyper-Pyramid