COMPARING TECHNIQUES OF MAPPING PYRAMID ALGORITHMS ONTO THE HYPERCUBE: A CASE STUDY FOR THE CONNECTION MACHINE

by
Muhammad Ali Siddiqui

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering
May 1992

APPROVAL PAGE

Comparing Techniques of Mapping Pyramid Algorithms onto the Hypercube: A Case Study for the Connection Machine

by

Muhammad Ali Siddiqui

Dr. Sotirios G. Ziavras, Adviser
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. John Carpinelli, Committee Member
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member
Assistant Professor of Electrical and Computer Engineering, NJIT

ABSTRACT

Comparing Techniques of Mapping Pyramid Algorithms onto the Hypercube: A Case Study for the Connection Machine

by

Muhammad Ali Siddiqui

The pyramid structure is most widely used for low-level and intermediate-level image processing and computer vision because of its efficient support of both local and global operations. However, the cost of pyramid computers (PC) may be very high. They also do not support the efficient implementation of the majority of the scientific algorithms. In contrast, the hypercube network has widely been used in the field of parallel processing because it offers a high degree of fault tolerance, a small diameter and rich interconnection structure that permits fast communication at a reasonable cost. Thus, several algorithms have been developed for the efficient simulation of pyramids on hypercubes. Stout [2], Lai and White [3], and Patel and Ziavras [14] have proposed four different algorithms that map pyramids onto the hypercube. This thesis carries out a comparative analysis that involves all these algorithms. The comparison is based on results derived with the application of analytical techniques and actual program runs. A Connection Machine CM-2 system containing 16K processors was used to derive the latter type of results. Stout's algorithm is cost effective, as it requires a hypercube with a number of PEs which is equal to the total number of nodes in the base of the pyramid. Thus, it needs a $2n$-dimensional hypercube to map a pyramid with $n+1$ levels. Lai and White have proposed two mapping algorithms. They require double the number of PEs used by Stout's algorithm. Finally, the algorithm proposed by Patel and Ziavras requires the same number of PEs as Stout's algorithm but allows the simultaneous simulation of multiple levels, as long as the

leaf level is not included in the set of the levels required to be active at the same time. A comparative analysis is carried out for all four mapping algorithms through the incorporation of analytical techniques and results obtained on the Connection Machine system CM-2 for some important image processing algorithms.

BIOGRAPHICAL SKETCH

Author: Muhammad Ali Siddiqui

Degree: Master of Science in Electrical Engineering

Date: May, 1992

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering, New Jersey Institute of Technology, Newark, NJ, 1992
- Bachelor of Science in Electronics Engineering, NED University of Engineering and technology, Karachi, Pakistan, 1988

Major: Electrical Engineering

Positions Held:

5/91-Research Assistant, Electrical and Computer Engineering, New Jersey Institute of Technology, Newark NJ.
9/91-Graduate Assistant, Computer Maintenance Facility, New Jersey Institute of Technology, Newark, NJ.
1/92-CO-OP Engineer, Glatt Air Techniques, Ramsey, NJ.

This thesis is dedicated
to my parents
whose love and inspiration
instilled in me the belief
that what I set out to accomplish
I can.

ACKNOWLEDGEMENT

I wish to express my sincere appreciation and deep gratitude to my academic advisor Dr. Sotirios G. Ziavras, Chairman of the graduate committee, for his helpful suggestions and constructive criticism.

Special thanks to Dr. John Carpinelli and Dr. Edwin Hou for serving in my thesis committee and for their valuable suggestions.

I would also like to thank Dr. Larry S. Davis, Director of the University of Maryland Institute for Advanced Computer Studies (UMIACS) for his permission to access the Connection Machine CM-2 System in UMIACS.

I wish to thank Nayyar for her continuous help, encouragement and sacrifices she made for me, without which this work would never have been possible.

And finally, a thank you to Ashraf Siddiqui for his support, inspiration and valuable suggestions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Requirements of Image Processing

One of the most important, difficult, and computationally intensive problems in scientific computing is image processing and computer vision. Computer systems used for image processing range from microprocessor devices to computer systems capable of performing computationally intensive functions on large image arrays. Image processing and computer vision algorithms employ a very broad spectrum of techniques from several areas such as signal processing, advanced mathematics, graph theory, and artificial intelligence. The computational requirements to perform algorithms from these fields are tremendous when executed individually, and when they need to be integrated in a meaningful way to perform a broader function in a reasonable amount of time, the computation becomes almost intractable [23].

The principal parameters influencing the structure of a computer are the intended application and the required data throughput. Therefore, the question arising here as what kind of architecture can provide the tremendous amount of processing power required by image processing and computer vision. Parallel processing, which has progressed tremendously ın the past decade, seems to be the consensus approach to providing the necessary computational power. Parallel processing holds the potential for computational speeds that surpass by far those achievable by technological advances in sequential computers. This potential is predicated on two assumptions, namely, that many computations can take place concurrently and the time spent in data exchanges between these computations is small. In order to meet these assumptions, algorithms must be partitioned into computational blocks that can execute in parallel and have communication requirements efficiently supported by the target parallel computer. Fortunately, most image processing algorithms are characterized by massive parallelism, so spatial decomposition of an image provides a natural way of generating low-level parallel tasks. For higher level analysis operations, parallelization may be based on other image characteristics and may be data dependent.

Another important requirement of image processing is real-time processing of image data. It is useful to consider why parallel architectures are so important for image processing. Clearly any algorithm can be implemented on a sequential computer. So, why is a powerful minicomputer or mainframe not adequate? The answer is that the general-purpose computers can not easily exploit the parallelism in an arbitrary algorithm and can not process the algorithm in real-time. The whole essence of using parallel architecture for image processing is to exploit the special forms of parallelism found in the image data and to process them in real-time. Parallel architectures can not only out-perform powerful minicomputers and mainframes but they can do so at a much lower cost.

1.2 Pyramid Structure

The *pyramid* structure is composed of successive layers of mesh-connected two-dimensional arrays, where the size of the arrays decreases with the increase of the level number (assuming that the base corresponds to level 0). In addition, each node at any level, except for nodes at the lowest level, is directly connected to four children located at the immediately lower level (i.e., the reduction between pairs of neighboring levels is 2×2, and the size of each array is 1/4 the size of the array at the immediately lower level [15]). The pyramid structure is appropriate for low-level and intermediate-level computer vision algorithms because of its efficient support of both local and global operations [2, 9, 10, 15]. It is well known that low-level and intermediate-level image processing and computer vision are characterized by local and global operations, with the majority of them being local. In addition, this structure is capable of supporting the efficient implementation of multilevel solvers which involve local processing on different scales with various inter-scale interactions [19]; such solvers are used in the solution of partial differential equations, constrained optimization, image reconstruction [18], multivariate interpolation, etc.

In the rest of the discussion P_n denotes a standard pyramid with $2^n \times 2^n$ nodes at its base. Such a pyramid has $n + 1$ levels. Fig. 1.1 shows the P_2 pyramid.

1.3 Pyramid Algorithms

The pyramid architecture provides straight forward implementation of divide-and-conquer techniques and efficiently carries out both local and global operations. However, the effectiveness and performance of the pyramid architecture is limited to applications that use such techniques and/or operations. Low-level and intermediate-level image processing and computer vision are candidate application domains [23].

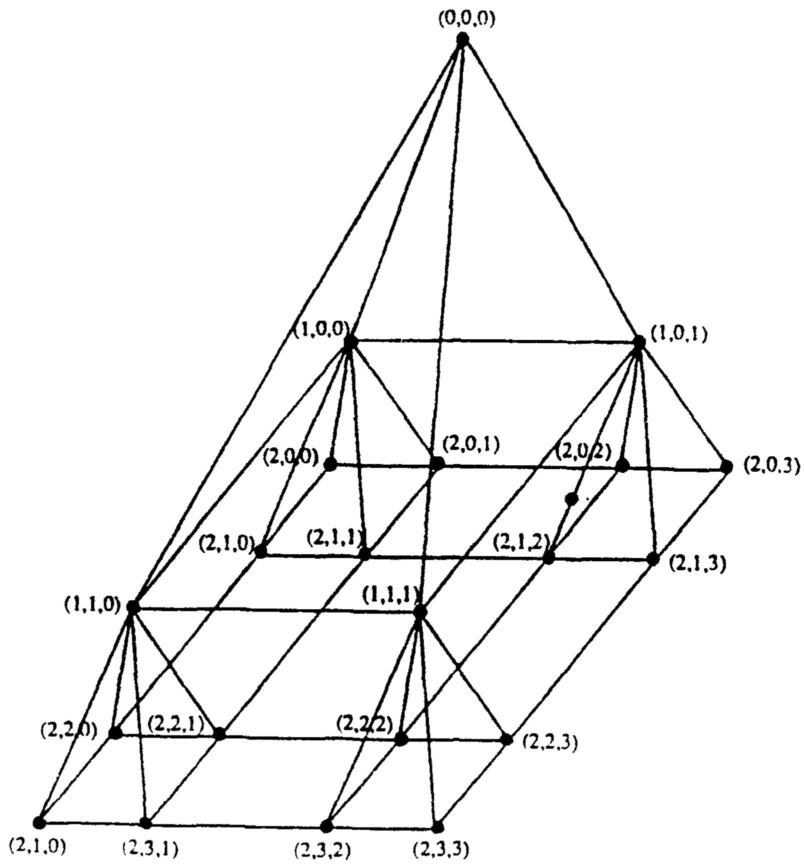

Figure 1.1. The two-level (P_2) pyramid.

Several image processing algorithms have been proposed for implementation on the pyramid structure. A brief discussion of various pyramid algorithms follows.
The Counting algorithm: The counting of connected regions is needed for object recognition tasks. Euler, the mathematician, developed an algorithm that can characterize any polygon. This algorithm is used to identify the connected regions within an image. The image is loaded into the base of the pyramid and some logical functions are performed on the vertices and edges of the object to recognize the connected regions [24].
Image Smoothing algorithm: One of the most common operations performed on images is to blur or smooth the image brightness values. Smoothing enhances an image by reducing the effect of noise so that subsequent processing is simplified and regularized. In addition, the amount of smoothing can be adjusted so as to optimally set the resolution at which to locate image features (e.g., edges and textures) which naturally occur at a variety of spatial scales [24]. The Gaussian pyramid may be used for image smoothing; each level of the Gaussian pyramid represents a smoothed version of the original image.
Object Segmentation: Segmentation as used here means to separate connected regions of a binary image into separate memory planes, so that each region can be analyzed individually. One frequently used algorithm for segmentation is region-filling; that is, an expansion that starts at a randomly chosen object pixel and continues until the whole region has been filled. When this idea is applied to a pyramid structure, the timing is logarithmic for images with large blobs.

1.4 Motivations and Objectives

The hypercube is a general purpose topology which can very efficiently simulate other frequently used structures, like the mesh, tree, pyramid, etc. As a consequence, hypercube-based machines have become commercially available, such as the Intel iSPC, NCUBE, Connection Machine, etc. In contrast, powerful pyramid machines are not cost-effective, are difficult to build with the current technology, and have very special and limited applications. Therefore, several algorithms to map the pyramid onto the hypercube have been developed. Such algorithms have been presented by Stout [2], Patel-Ziavras [14], and Lai-White [3]. These algorithms are characterized by different costs and performances. The algorithms proposed by Stout and Patel-Ziavras require an H_{2n} hypercube to simulate a P_n pyramid with $2^n \times 2^n$ PEs at its base. In contrast, two algorithms proposed by Lai and White require a H_{2n+1} hypercube to simulate the same pyramid. However, Stout's algorithm does not allow more than one level of the pyramid to be active at the same time. On the other hand, Patel-Ziavras algorithm allows all of the levels, except the leaf level, to be active at the same time. Lai-White's both algorithms allow all of the pyramid levels to be active at the same time but they require twice as many PEs as required by Stout's and Patel-Ziavras' algorithms.

The implementation of these mapping techniques on a real hypercube system under various conditions becomes absolutely necessary for a comparative analysis.

The main objective of this research is to implement these mapping algorithms on a real system and then run some representative image processing applications to measure their performance. A Connection Machine CM-2 system containing 16K processors will be used to derive results. Results are obtained for three important image processing algorithms: finding the perimeter of an object, convolution and segmentation.

1.5 Thesis Outline

This thesis is organized as follows. Chapter 2 presents the hypercube topology and relevant applications. Detailed description of the Connection Machine system, which is used to derive the results for these algorithms, is also included. Chapter 3 discusses the four mapping algorithms in detail. It also introduces several performance measures which are important for performance analysis techniques. Comparative analysis for all these algorithms is also carried out in Chapter 4 using Connection Machine results. Finally, Chapter 5 presents conclusions.

CHAPTER 2

THE HYPERCUBE STRUCTURE

Various parallel processor structures have been used in parallel systems. In recent years, hypercube computers have become popular parallel computers for a variety of applications due to their powerful network which is characterized by a small diameter, regularity and high degree of fault tolerance. Most of the other important topologies like the linear array, mesh, ring and pyramid can efficiently be mapped onto the hypercube [22]. Therefore, most of the applications for these structures can be implemented on the hypercube very efficiently. Formally, an *n-dimensional* hypercube contains 2^n nodes. Nodes are connected directly with each other if and only if their binary addresses differ by a single bit. Hypercubes of zero, one, two and three dimensions are shown in Figure 2.1.

Hypercube computers are loosely coupled parallel processor systems based on the binary n-cube network, also known as cosmic cube, n-cube, binary n-cube, Boolean n-cube, etc. Various parallel computers have been developed using this structure. The Connection Machine system, which is used to derive the results in this thesis, is one of the most well known systems and is manufactured by Thinking Machine Corporation. It operates in the SIMD mode and may contain up to 65.536 PEs. The topological properties of the hypercube and the Connection Machine architecture are presented in the following sections of this chapter.

2.1 Topology

In the *d-dimensional* hypercube or d-cube H_d, each processor is directly connected with d neighboring processors. Each processor has a unique $d-bit$ binary address in the interval 0 to 2^d-1. In a hypercube computer, PEs are placed at each vertex of the hypercube and the edges of the hypercube represent communication links between PEs. Each PE has its local memory, which makes every PE an independent unit. In the SIMD mode, this memory only contains data whereas in the MIMD mode this memory also contains instructions. Hypercube PEs are homogeneous because all the nodes can be treated equally; any hypercube can be mapped onto itself by mapping a node to any other. When a node i is mapped onto another node j, the addresses of all nodes are changed and the new address of a node is found by taking the XOR of its previous address and the address of node i.

The communication time between two PEs of the hypercube depends on the number of links between them. The maximum communication time between any two PEs in the $d-dimensional$ hypercube is $O(d)$ because the maximum number of intermediate links is d. The total number of 1s in the XOR between the binary addresses of two PEs gives the maximum number of communication links between these PEs. If PE Y is connected with PE X in its ith dimension, then the addresses of X and Y will differ only in the ith bit position.

The hypercube can be partitioned into smaller dimensional hypercubes and a d-dimensional hypercube can be constructed recursively from lower dimensional hypercubes: for example if two $(d-1)-dimensional$ hypercubes are combined.

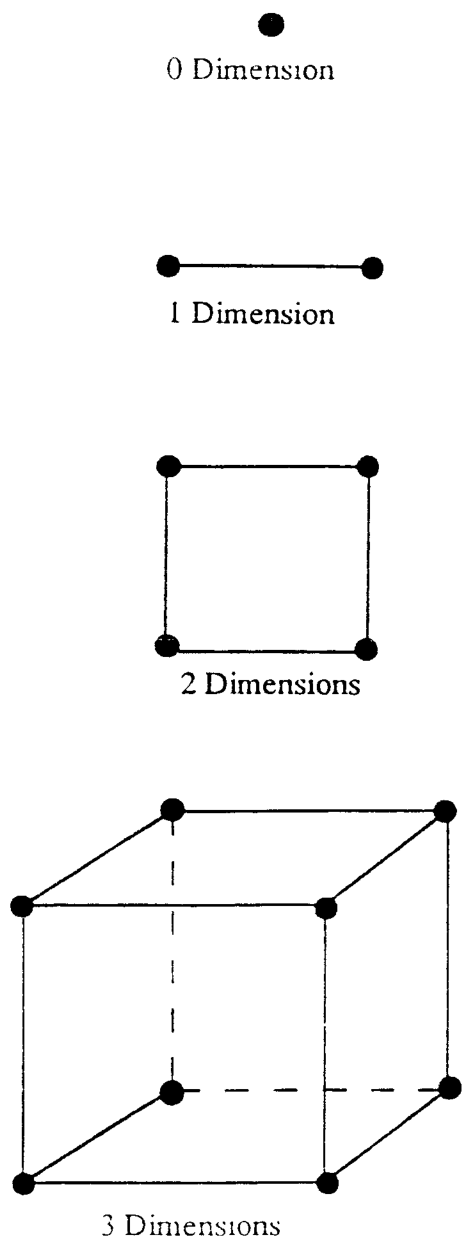

Figure 2.1. Hypercubes of different dimensions.

they produce a $d-dimensional$ hypercube. Consider two identical $(d-1)-dimensional$ hypercubes with labels from 0 to $2^{d-1}-1$; by joining vertices with the same addresses, a $d-dimensional$ hypercube is formed. Figure 2.2 shows how two 3-cubes are combined to produce a 4-cube.

To summarize:

1. Any $d-cube$ can be tiered in d possible ways into two $(d-1)-subcubes$.

2. There are $d! \times 2^d$ ways of numbering the 2^d nodes of the $d-cube$.

3. The maximum distance between any two nodes in the $d-cube$ is equal to d, which is also called the diameter of the hypercube.

4. Any two processors in the d-cube can communicate with each other. In order to communicate, data has to travel at least a distance which is equal to the number of 1s in the XOR between the addresses of these PEs (this is known as the Hamming distance H(X,Y) between PEs X and Y).

2.2 Applications

Various topologies can be mapped efficiently onto the hypercube. There are basically two reasons for the importance of such a mapping.

1. Some algorithms may be developed for some other topology for which they fit perfectly. Then, one might wish to implement the same algorithm on the hypercube with little programming effort. If the original architecture can efficiently be mapped onto the hypercube then this will be achieved easily.

2. A given problem may have a well defined structure, which requires a particular pattern of communication. Mapping that pattern onto a hypercube may result in short communication time.

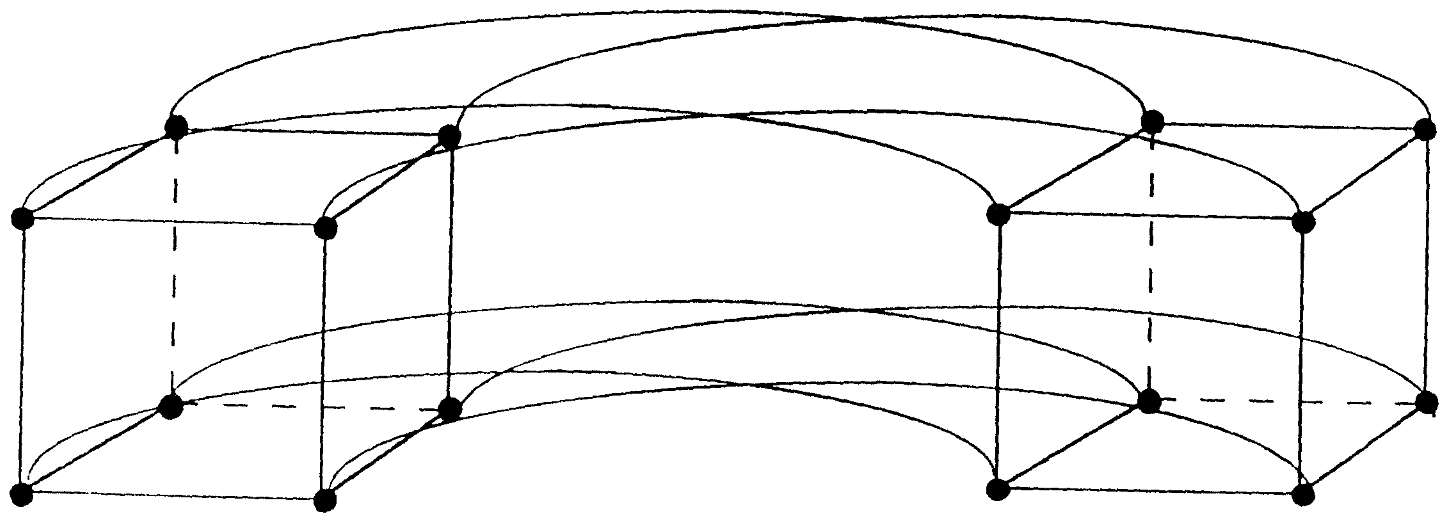

Figure 2.2. A 4-cube formed from two 3-cubes.

Some important mappings are discussed in the following section.

2.2.1 Mapping Rings onto the Hypercube

Consider a ring structure containing 2^d PEs. Also consider a target d-dimensional hypercube. The ring can be mapped onto the hypercube in such a way that the proximity property is preserved (i.e., any two adjacent vertices of the ring map onto two neighboring nodes of the hypercube). Another way of visualizing this problem is that we are seeking a string of length $N = 2^d$ that crosses each node of the hypercube once and only once.

According to the definition of the hypercube network, any two adjacent nodes have binary addresses that differ only by one bit. This means that the hypercube addresses should be represented by a sequence of $d-bit$ binary numbers such that any two successive numbers have only one different bit. A binary sequence with such a property is the reflected Gray code.

The mapping of the 8-node ring onto the 3-dimensional hypercube is shown in Fig. 2.3. This figure shows the linear array with the extra connections which are present in the hypercube.

2.2.2 Mapping the Mesh onto the Hypercube

One of the most important reasons that the hypercube is popular is that meshes can easily be mapped onto hypercubes. Consider a n-dimensional mesh that has size m_i in each dimension which is a power of 2 (i.e., $m_i = 2^{p_i}$).

Now consider the $d - dimensional$ hypercube on which this mesh is to be mapped. Let $d = p_1 + p_2 + .. + p_n$, where 2^d is the total number of processors in the n-dimensional grid, which is also the total number of nodes in the hypercube.

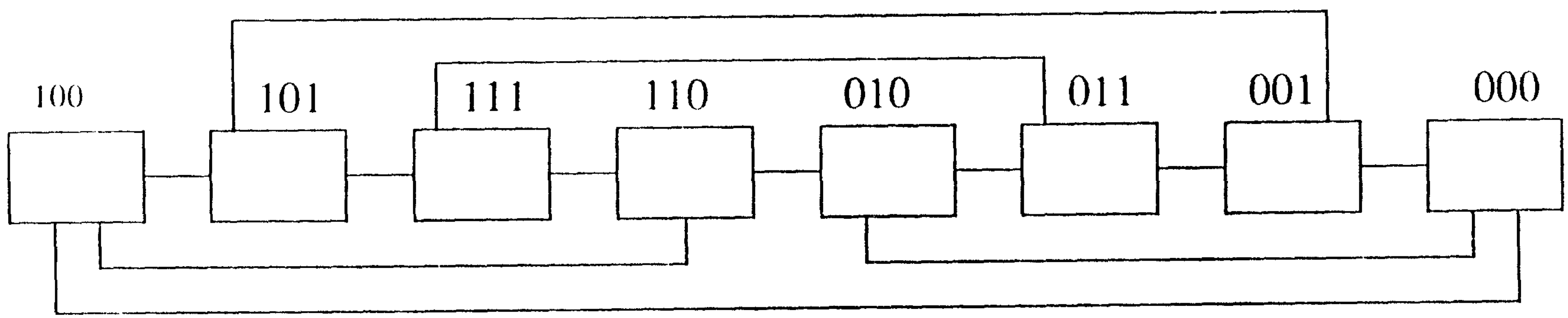

Figure 2.3. A linear array mapped onto the 3-cube.

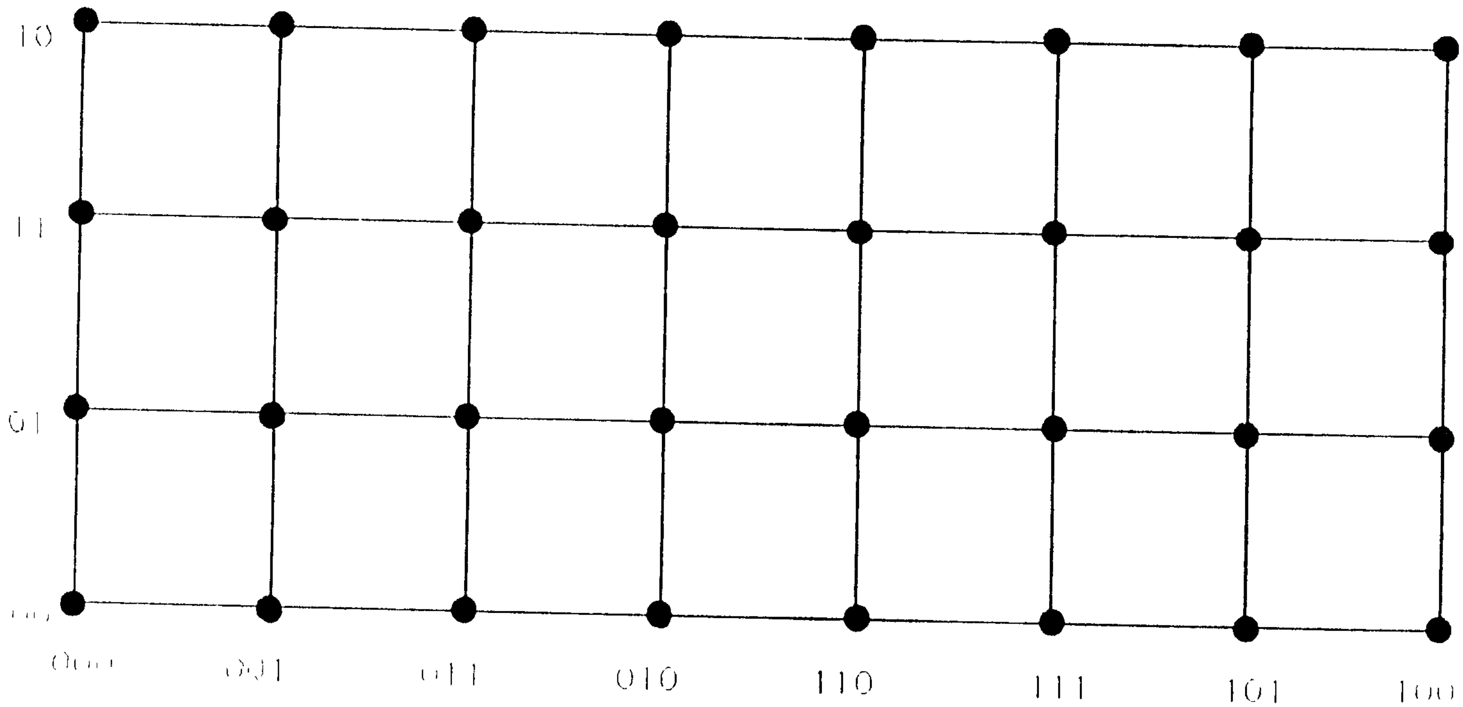

Figure 2.4. Mapping of an 8×4 mesh.

In order to perfectly map the mesh onto the hypercube, neighboring nodes in the mesh must be assigned to neighboring nodes in the hypercube. In the previous section, the mapping of the one-dimensional mesh (i.e., the linear array or ring) was discussed. The mapping of higher dimension meshes is done as follows. The nodes in each dimension are numbered sequentially using the respective reflected Gray code. A node of the mesh is mapped onto the node in the hypercube whose address is obtained by concatenating the numbers of the particular node for all the dimensions. For example, Fig. 2.4 shows a two-dimensional 8×4 mesh and the appropriate Gray codes

2.3 The Connection Machine

The Connection Machine is a data parallel computing system. Data parallel computing associates one processor with each data element. This computing style exploits the natural computational parallelism inherent in many data-intensive problems.

The Connection Machine is an integrated system of hardware and software. The hardware elements of the system include front-end computers that provide development and execution environment for the user's software, a parallel processing unit of up to 64K processors (PEs) that execute data parallel operations in the SIMD mode, and a high performance data parallel I/O system. Each PE has its own local memory of 8-kilobytes which is bit-addressable and its word length is one bit. The hypercube is the dominant topology in the system. More specifically, a 10-dimensional hypercube is the backbone (router) of the communication network. Each vertex of this hypercube contains a router node (communication processor) to which sixteen PEs are attached [16]. The largest Connection Machine CM-2 system contains 64K PEs and router nodes are located at the vertices of a 12-dimensional hypercube. In addition, the CM-2 hardware includes specific communications hardware, the router and the NEWS grid. These communication techniques are discussed in detail in the following sections.

Message passing is implemented in parallel: algorithmically selected subsets of PEs are allowed to simultaneously send data into the local memories of other PEs or fetch data from the local memories of other PEs into their own. The communications hardware is also capable of combining multiple messages going to the same destination PE by applying some arithmetic or logical combining (i.e., reduction) operation. The destination PE then receives the result. The router nodes forward messages and also perform some dynamic load balancing. Processing of messages by the router is divided into stages which are called petit cycles. A petit cycle is just enough to process all the bits of a destination address and a message. This is also true for a message that traverses all twelve or ten dimensions of a 64K or a 16K machine respectively. Therefore, a petit cycle consists of multiple ALU/route cycles. A single communication pattern may consume a single petit cycle if only a small number of PEs are involved. In contrast, if almost all of the PEs are active, then many petit cycles may be consumed.

The following sections describe these two communication techniques in more detail.

2.3.1 The Router

The most general communication mechanism of CM-2 is the router, which allows any processor to communicate with any other processor. One may think of the router as allowing every processor to send a message to any other processor, with all messages being sent and delivered at the same time. Alternatively, one may think of the router as allowing every processor to access any memory location within the parallel 0 unit with all processors making memory accesses at the same time.

Each CM-2 processor chip contains one router node, which serves 16 data processors on the chip. The router nodes on all the processors are wired together to form the complete router network. Each message travels from one router node to another until it reaches the chip containing the destination processor. The router nodes automatically forward messages and perform some dynamic load balancing. It is possible for a message to traverse many dimensions, possibly all twelve, in a single petit cycle, provided that contention does not cause it to be blocked. The message data is forwarded through multiple router nodes in a pipelined fashion. A message that cannot be delivered by the end of a petit cycle is buffered in whatever router node it happens to have reached, and continues its journey during the next petit cycle.

2.3.2 The NEWS Grid

Communication operations between processors that are nearest neighbors within a Cartesian grid are much more efficient than the general router mechanism because they exploit three different transfer methods, two of which have special hardware support[16].

The fully configured CM-2 system (with 64k PEs) has 2^{12} processor chips with connecting wires forming a boolean 12-cube: these are the same physical wires that serve the general router mechanism. A subset of these wires can be chosen so that they connect the 2^{12} chips as a two-dimensional grid of shape. The hardware is flexible enough to accommodate any shape. For example the per-chip permutation circuit can organize its 16 physical processors as 8×2, or 1×16, or $4 \times 2 \times 2$, or $2 \times 2 \times 2 \times 2$, and so on. Due to this specialized hardware support, the NEWS grid of any shape or number of dimensions can be handled with great speed and efficiency.

CHAPTER 3

MAPPING PYRAMIDS ONTO HYPERCUBES

A first level comparison of various embedding is enabled by the introduction of three measures of the cost of graph embeddings; namely expansion, dilation, and congestion. Before we discuss the mapping algorithms, these performance measures are presented.

3.1 Performance Measures

Let the function $h : G \rightarrow G'$ represent the mapping of the source graph G onto the target graph G'. It is a mapping of the vertices of G to the vertices of G' in a one-to-one or many-to-one fashion. The three measures are then defined as follows [3].

Expansion: The expansion of h is the ratio $\frac{|V(G')|}{|V(G)|}$, where $V(G)$ and $V(G')$ are the vertex sets of G and G' respectively, and $|V(G)|$ and $|V(G')|$ are the numbers of elements in those sets. When $|V(G')| \geq |V(G)|$, the expansion measures how much of the target graph G' is not assigned nodes from the source graph G. The closer the value of this measure to one, the smaller is the portion of unused resources in G'.

Dilation: When two neighboring nodes from G are mapped onto two distinct nodes in G', the dilation of the edge connecting the two nodes in G is the length of the corresponding path in G'. The maximum dilation is the maximum length of such a path in G'. The dilation measures the increase of the communication overhead when compared to one-hop transfers in the source graph. Of course, the smaller the value of the dilation is, the lower the communication overhead associated with the mapping h.

Congestion: The congestion is the number of edges in G with the same image in G'. The maximum number of edges in G with the same image in G' is the maximum value of the congestion for the chosen mapping h. The smaller the value of the congestion, the less amount of time that messages will have to wait in the queues of intermediate target PEs for communication channels to become available.

3.2 Stout's Algorithm

The mapping algorithm which was presented by Stout [2] embeds the P_n pyramid into the H_{2n} hypercube. Therefore, the total number of nodes in the hypercube is equal to the number of nodes in the base of the pyramid. Since a pyramid with a base of size $2^n \times 2^n$ contains a total of $\lfloor \frac{2^{2(n+1)}}{3} \rfloor$ nodes, the expansion is less than 1. A one-to-one mapping of nodes from the base of the pyramid onto PEs of the hypercube is accomplished as follows. The n-bit reflected Gray code is used to encode separately the rows and columns of the base. The binary addresses of the corresponding PEs in the hypercube are found by either interleaving or concatenating the bits of the encoded row and column numbers. This process produces a perfect mapping for the base of the pyramid: that is. all three measures associated with the cost of the base's mapping are optimal (i.e.. they are equal to 1). Every node at the immediately higher level of the pyramid (i.e.. level 1) has four children at the leaf level (i.e.. level 0). and as a consequence one PE from each square of four PEs is chosen to simulate the parent node. PEs having the least significant bit of their encoded row and column numbers equal to 0 are chosen to represent level 1 nodes of the pyramid. In general. PEs having the lower k bits of their encoded row and column numbers equal to 0 will simulate nodes from level k of the pyramid. Thus. one of the children will use two communication links when sending data to its parent (i.e.. the dilation of such a data transfer is equal to two). Fig. 3.1 shows the mapping of the P_3 pyramid onto the H_6 hypercube: the numbers within the squares represent level numbers. This way. the dilation of all lateral edges in the pyramid is equal to one for all of the levels. However. the maximum dilation of this mapping is equal to two and corresponds to edges connecting pairs of parents and children as discussed above.

The two significant advantages of this mapping are the smallest possible resultant dilation and the relatively small number of PEs in the hypercube (more specifically. the total number of PEs in the target hypercube is smaller than the total number of nodes in the source pyramid). The maximum congestion of this mapping is equal to three.

Since a single hypercube PE may be used to simulate a number of pyramid nodes from different levels (for example. the PE with row number 0 and column number 0 is used to simulate nodes from all levels of the pyramid). the hypercube is not capable of simulating multiple levels of the pyramid at the same time. In fact. if many levels of the pyramid need to be active simultaneously. a hypercube PE will not only be incapable of simulating nodes from several levels of the pyramid simultaneously but may spend some extra time in switching from one simulation to

RGC	000	001	011	010	110	111	101	100
000	0.1.2.3	0	0	0.1	0.1	0	0	0.1.2
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0.1	0	0	0.1	0.1	0	0	0.1
110	0.1	0	0	0.1	0.1	0	0	0.1
111	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
100	0.1.2	0	0	0.1	0.1	0	0	0.1.2

Figure 3.1: Mapping the P_3 pyramid onto the H_6 hypercube with Stout's Algorithm . (RGC: 3-bit Reflected Gray Code.)

the next one; in addition, the storage space needed to store data for the simulated nodes may become prohibitively large.

Algorithms that keep active all, or a large subset, of the pyramid's levels most of the time are common; for example, algorithms that implement pipelining fall into this category [13]. However, this mapping of the pyramid does not consume prohibitively long time if the pyramid algorithm proceeds level by level: as discussed earlier, the only delay occurs during the communication of values between parents and one of their children.

3.3 Patel-Ziavras' Algorithm

Similar to Stout's algorithm, the mapping algorithm proposed by Patel and Ziavras [14] maps the P_n pyramid onto the H_{2n} hypercube. However, in contrast to Stout's Algorithm, this algorithm allows multiple levels of the pyramid to be active simultaneously. More specifically, it allows any subset of levels, excluding the leaf level, to be active at a time. The simulation of the leaf level excludes the simultaneous simulation of any other level in the pyramid because the total number of leaf nodes is the same as the number of PEs in the hypercube. The mapping algorithm operates as follows. Similarly to Stout's algorithm, the reflected Gray code is used to independently encode the row and column numbers of the leaf level. A perfect mapping is then produced for this level by either concatenating or interleaving the bits of the encoded row and column numbers of the nodes in order to find the addresses of the corresponding target PEs in the hypercube. The mapping of level 1 nodes is also similar to the mapping produced by Stout. More specifically, the PEs of the hypercube chosen to simulate parents of leaf nodes correspond to encoded row and column numbers that have their least significant bit equal to 0. For each set of four PEs representing sibling nodes at level 1 of the pyramid which have a common parent at level 2, a PE is again chosen to represent their parent. The PE chosen to serve as the parent is neighbor to one of the PEs representing the children and all parent PEs for level 2 form mirror images in squares outlined by their children. This procedure is repeated until the apex of the pyramid is reached. For example, as shown in Fig. 3.2, the leaf level nodes of the P_3 pyramid are simulated by all 2^6 PEs of the H_6 hypercube (using a one-to-one assignment). There are sixteen groups (squares) of 2×2 PEs at the leaf level that have a common parent at level 1. The parent at the next higher level (i.e., level 1) of the children in such a square is simulated by the PE marked with 1 in the square. These PEs marked with 1 are again grouped into groups of four PEs that have a common parent. Parents at the next higher level are simulated by the PEs marked with 2. Finally, the parent at the next higher level (i.e., level 3) of the children marked with 2 is simulated by the PE marked with 3. Thus, PEs marked with 0, 1, 2 and 3 simulate nodes from levels 0, 1, 2 and 3 respectively of the P_3 pyramid. Since PEs that simulate different levels of the pyramid, except for the leaf level, are distinct, any subset of pyramid levels that does not include the leaf level can be simulated simultaneously.

RGC	000	001	011	010	110	111	101	100
000	0.1	0.2	0.3	0.1	0.1	0	0.2	0.1
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0.1	0	0	0.1	0.1	0	0	0.1
110	0.1	0	0	0.1	0.1	0	0	0.1
111	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
100	0.1	0.2	0	0.1	0.1	0	0.2	0.1

Figure 3.2: Mapping the P_3 pyramid onto the H_6 hypercube with Patel-Ziavras algorithm. (RGC: 3-bit Reflected Gray Code.)

The maximum dilation of the mapping for an edge connecting a parent at level 1 and one of its children at level 0 is two (as in Stout's algorithm). However, the maximum dilation for higher levels is equal to three. In general, both the maximum dilation and the maximum congestion associated with this mapping algorithm are equal to three

3.4 Lai-White's Algorithm I

Two algorithms suggested by Lai and White [3] for mapping a pyramid onto a hypercube map distinct nodes of the pyramid onto distinct PEs of the hypercube while maintaining minimal expansion. More specifically, they require an H_{2n+1} hypercube for the mapping of the P_n pyramid (this is the smallest allowable hypercube size if distinct nodes of the pyramid need to be mapped onto distinct PEs of the hypercube). This subsection and the next one describe these two mapping algorithms. Their first mapping algorithm is recursive and yields maximum congestion two and maximum dilation three. It starts by defining an embedding function for the two-level 0 with apex $(n, 0, 0)$: the elements of the triplet represent the level number, the row position, and the column position respectively of the node. It then applies a recursive function in order to derive the mapping of lower level 0 until the base is reached. This technique results in the mapping of the leaf nodes. The mapping of higher level nodes is then achieved with the application of a bottom-up approach. Before we briefly present the algorithm, some definitions become pertinent. In addition, we need to emphasize that contrary to our up to this point notation, for the sake of simplicity the following description assumes that the apex is level 0 while the base of the pyramid is level n (i.e., the numbering of levels starts from the top). The embedding $f_1 : P_1 \mapsto H_3$ of the two-level subpyramid, as illustrated in Fig 3.3, is first defined as:

- $f_1(0,0,0) = 000$,
- $f_1(1,0,0) = 010$,
- $f_1(1,0,1) = 110$,
- $f_1(1,1,0) = 011$, and
- $f_1(1,1,1) = 111$.

where the triplets represent the addresses of nodes in the source pyramid and the binary numbers on the 1 side of the equations are the binary addresses of PEs in the target cube. This process maps all four children of the apex onto a side of the cube which is opposite from the side containing the apex. Then, three additional embeddings are defined through vertical, horizontal, and diagonal exchanges of children on the cube side suggested by f_1. The new embeddings are called the reflections of f_1 and are denoted by f_1^V, f_1^H, and f_1^{VH} respectively.

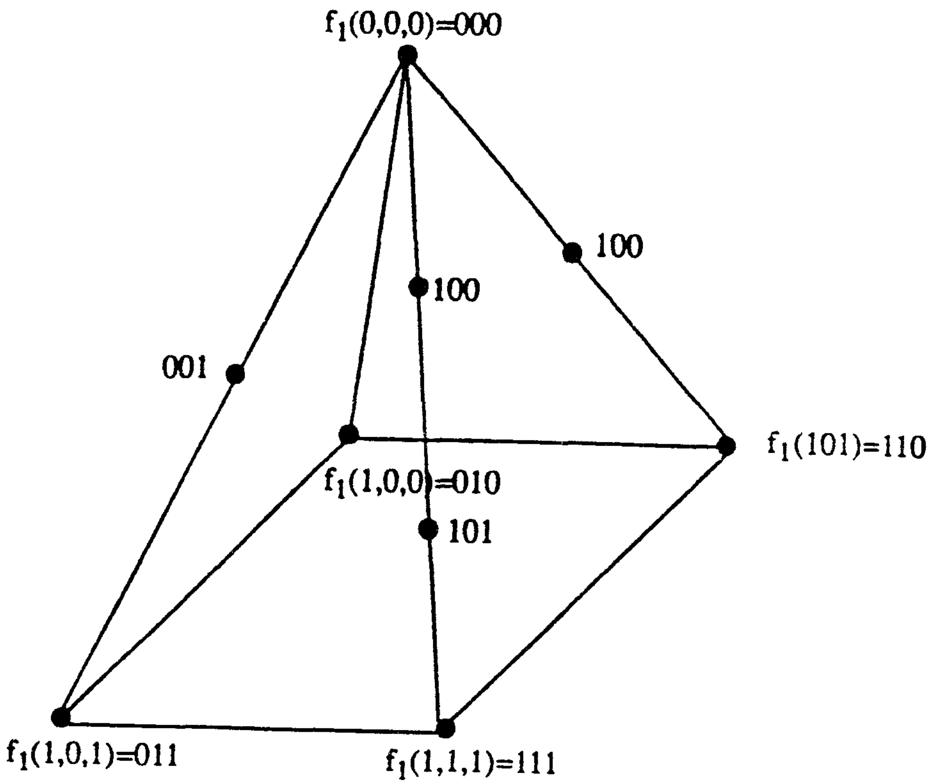

Figure 3.3. Embedding of two level sub-pyramid.

These embeddings are generalized as follows. Let $P_{k-1}(1, x_1, x_2)$ denote the subpyramid of P_k containing k levels and having as apex the node $(1, x_1, x_2)$. Define $\Phi^V_{k-1} : P_{k-1} \mapsto P_{k-1}$ so that node $(i, x_1, x_2) \in V(P_{k-1})$ is mapped onto $(i, x_1, 2^i - x_2 - 1)$. In addition, define $\Phi^H_{k-1} : P_{k-1} \mapsto P_{k-1}$ so that (i, x_1, x_2) is mapped onto $(i, 2^i - x_1 - 1, x_2)$. The three additional embeddings of P_{k-1} into H_{2k-1} are:

- $f^V_{k-1} = f_{k-1}\Phi^V_{k-1}$.
- $f^H_{k-1} = f_{k-1}\Phi^H_{k-1}$.
- $f^{VH}_{k-1} = f_{k-1}\Phi^V_{k-1}\Phi^H_{k-1}$.

For any pair of binary numbers b_1 and b_2, define the prefix function $h_{b_2b_1} : H_{2k-1} \mapsto H_{2k+1}$ so that $h_{b_2b_1}(x) = b_2b_1x$ and $h_{b_2b_1}(x, x') = (b_2b_1x, b_2b_1x')$ for any vertex x and edge (x, x') in H_{2k-1}. Then, define $t_{x_1x_2} : P_{k-1}(1, x_1, x_2) \mapsto P_{k-1}$ so that $t_{x_1x_2}(i, x'_1, x'_2) = (i - 1, x'_1 - x_1 \times 2^{i-1}, x'_2 - x_2 \times 2^{i-1})$ for any node (i, x'_1, x'_2) in $P_{k-1}(1, x_1, x_2)$ and $t_{x_1x_2}(u, v) = (t_{x_1x_2}(u), t_{x_1x_2}(v))$ for any edge (u, v) in $P_{k-1}(1, x_1, x_2)$.

These reflection are illustrated in Fig 3.4 .

The algorithm is as follows:

Algorithm

1 For $k = 1$, embed P_1 into H_3 using f_1.

2. For $k > 1$, use f_{k-1} to define f_k: i. Embed the subpyramid $P_1(0, 0, 0)$ into a three-dimensional subcube of H_{2k+1}, as follows:

- $f_k(0, 0, 0) = 00^3$.

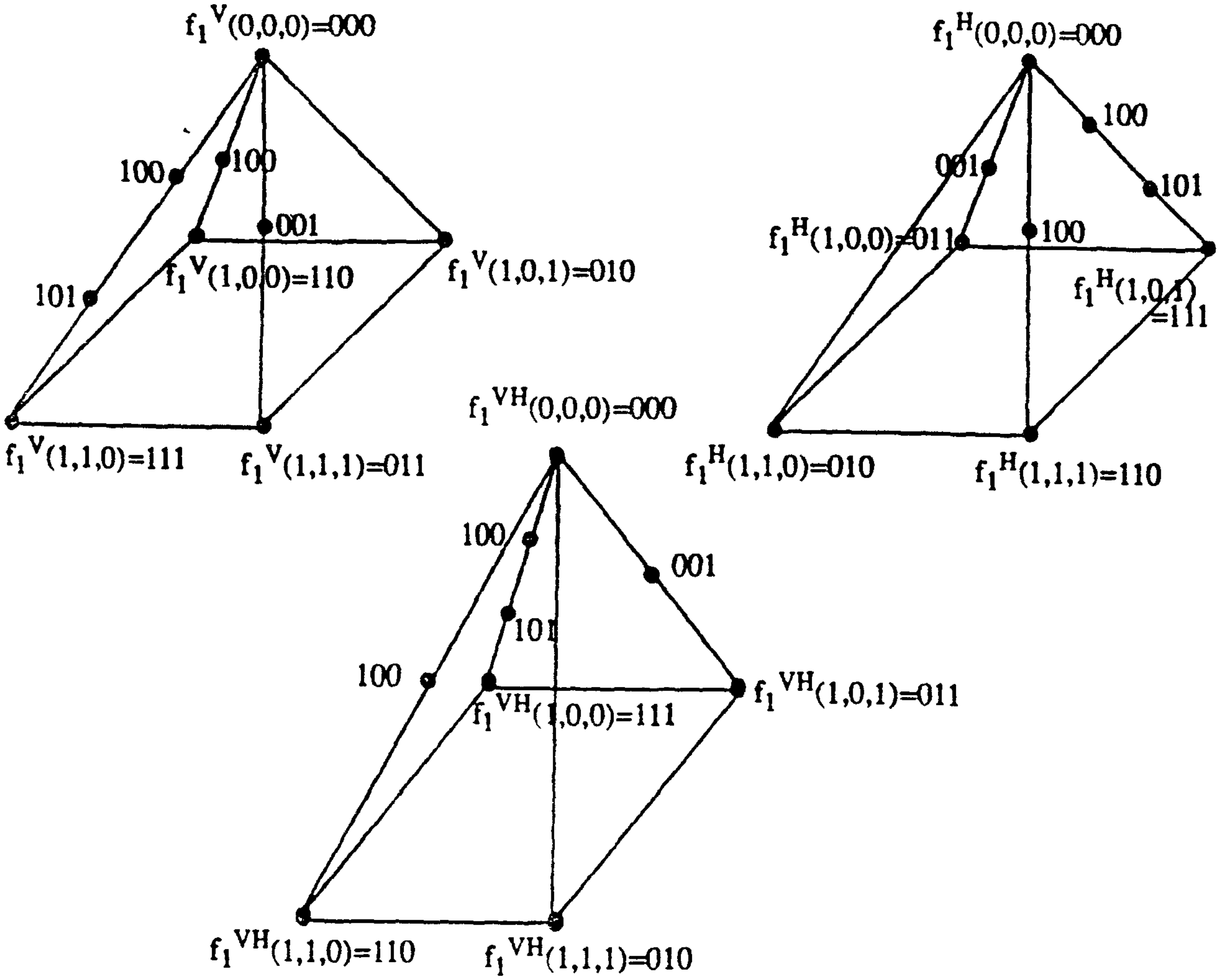

Figure 3.4 The image of P_1 under f_1^V, f_1^H, f_1^{VH}.

- $f_k(1,0,0) = 00\alpha$, $f_k(1,0,1) = 10\alpha$,
- $f_k(1,1,0) = 01\alpha$, $f_k(1,1,1) = 11\alpha$
- $f_k((0,0,0),(1,0,0)) = (00\beta, 00\alpha)$,
- $f_k((0,0,0),(1,0,1)) = (00\beta, 10\beta, 10\alpha)$,
- $f_k((0,0,0),(1,1,0)) = (00\beta, 01\beta, 01\alpha)$,
- $f_k((0,0,0),(1,1,1)) = (00\beta, 10\beta, 11\beta, 11\alpha)$.

ii. For each node $(1, x_1, x_2)$, embed $P_{k-1}(1, x_1, x_2)$ into the $(2k-1)$-dimensional subcube $x_2 x_1 H_{2k-1}$ of H_{2k+1} using the reflections of f_{k-1}:

- $h_{00} f_{k-1} t_{x_1 x_2}$, if $x_1 = x_2 = 0$,
- $h_{10} f_{k-1}^{H} t_{x_1 x_2}$, if $x_1 = 1$ and $x_2 = 0$,
- $h_{01} f_{k-1}^{V} t_{x_1 x_2}$, if $x_1 = 0$ and $x_2 = 1$, and
- $h_{11} f_{k-1}^{VH} t_{x_1 x_2}$, if $x_1 = 1$ and $x_2 = 1$.

The mapping algorithm is illustrated in Fig. 3.5 .

3.5 Lai-White's Algorithm II

The embedding algorithm of the previous section has optimal expansion, but its maximum dilation, although small, is not optimal. In this section, a substantially more complex algorithm that embeds the pyramid into the hypercube with optimal expansion, maximum dilation two, and maximum congestion three is presented.

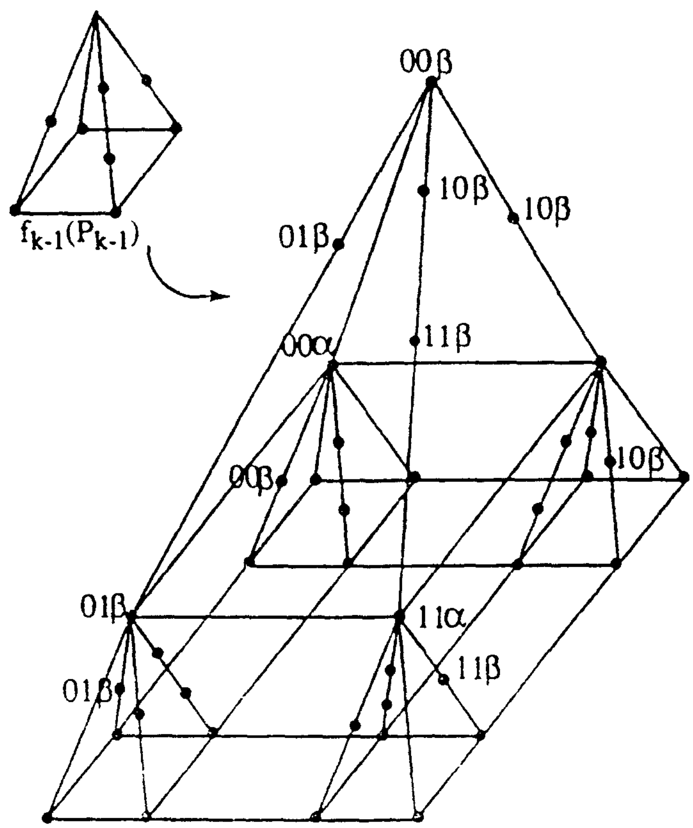

Figure 3.5. The recursive definition of f_k.

The second mapping algorithm proposed by Lai and White has maximum dilation two and maximum congestion three. Like their first algorithm, this algorithm also requires an H_{2n+1} hypercube for the mapping of the P_n pyramid and maps distinct nodes of the pyramid onto distinct PEs of the hypercube. This algorithm is also recursive, but in contrast to previous algorithm, it applies a top-down approach; i.e., pyramid nodes are mapped onto the target hypercube starting with the apex and the mapping process proceeds with the mapping of lower level nodes. This recursive process is much more complex than that of their first algorithm.

The algorithm is as follows.

Let $a = (k-1, x_1, x_2)$ be a node at level $k-1$, and $b = (k, 2x_1, 2x_2)$, $c = (k, 2x_1+1, 2x_2)$, $d = (k, 2x_1, 2x_2+1)$, and $e = (k, 2x_1+1, 2x_2+1)$ be its children at level k. Let also $P_1(k-1, x_1, x_2)$ denote this subpyramid of P_k with height one and apex $(k-1, x_1, x_2)$. For $v \in V(H_{2k+1})$ and $1 \leq p, q, r \leq 2k+1$, let $H_3(v; p, q, r)$ be the 3-dimensional 1 of H_{2k+1} containing the set of nodes $\{v, v^p, v^q, v^r, v^{pq}, v^{pr}, v^{qr}, v^{pqr}\}$, where the one, two, or three terms in the exponent show the position of the bits that must be complemented in v (one, two, and three bits respectively). Four embeddings of $P_1(k-1, x_1, x_2)$ are proposed, as shown below:

- $g_1(a) = v$, $g_1(b) = v^{qr}$, $g_1(c) = v^{pq}$, $g_1(d) = v^{r}$, and $g_1(e) = v^{pr}$.
- $g_1^V(a) = v$, $g_1^V(b) = v^{r}$, $g_1^V(c) = v^{pr}$, $g_1^V(d) = v^{qr}$, and $g_1^V(e) = v^{pq}$.
- $g_1^H(a) = v$, $g_1^H(b) = v^{pq}$, $g_1^H(c) = v^{qr}$, $g_1^H(d) = v^{pr}$, and $g_1^H(e) = v$
- $g_1^{VH}(a) = v$, $g_1^{VH}(b) = v^{pr}$, $g_1^{VH}(c) = v^{r}$, $g_1^{VH}(d) = v^{pq}$, and $g_1^{VH}(e) = v^{qr}$.

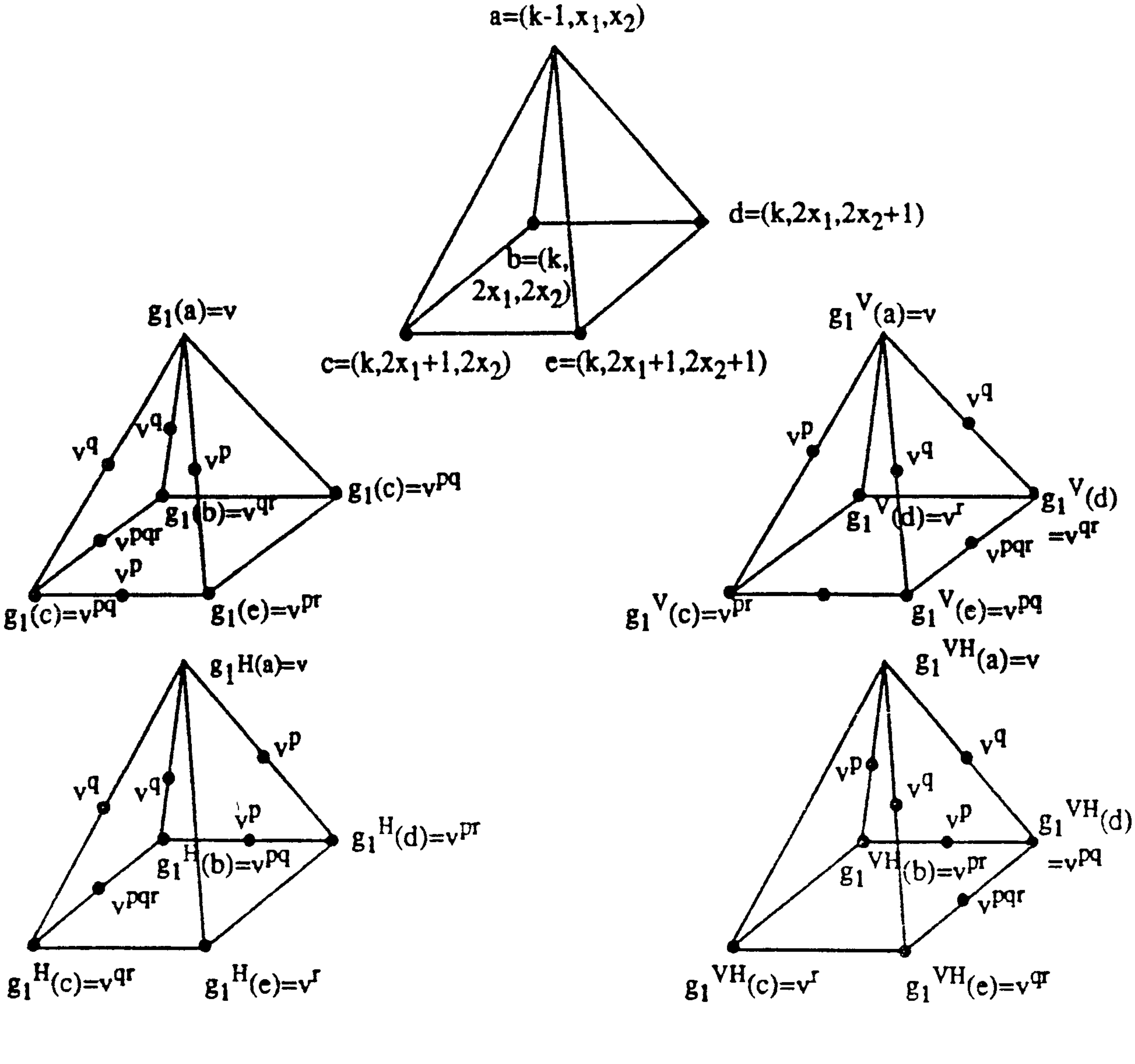

Figure 3.6. $P_1(k-1, x_1, x_2)$ and its images under $g_1, g_1^H, g_1^V, g_1^{VH}$.

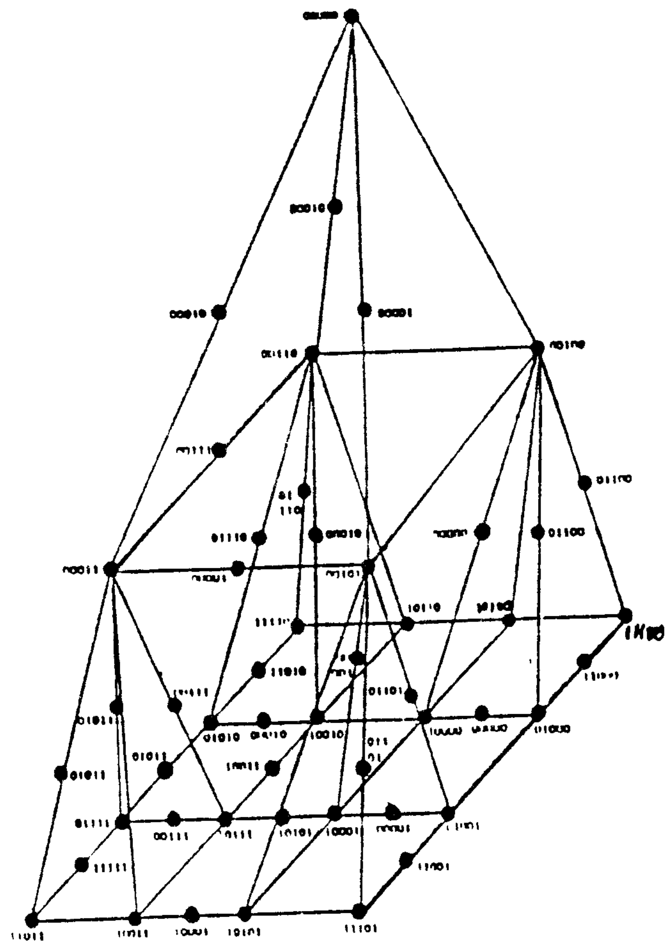

Figure 3.7. $g_2(P_2)$.

For $0 \leq a, b \leq m-1$, we define $\{a\}_m = \{x : x \mod m = a\}$, and $\{a, b\}_m = \{a\}_m \cup \{b\}_m$.
These embeddings and reflections are shown in figure 3.6. A pyramid with two levels (P_2), mapped on a 5-dimensional hypercube (H_5) is shown in Fig 3.7. The recursive algorithm for the embedding $g_k : P_k \mapsto H_{2k+1}$ is as follows:

Algorithm

1. For $k = 1$, use g_1 to embed P_k into H_{2k+1}.

2. For $k > 1$, use g_{k-1} to define g_k: i. Embed the top P_{k-1} subpyramid of P_k into H_{2k+1} using hg_{k-1}, where $h : H_{2k-1} \mapsto H_{2k+1}$, such that $h(x) = 00x$. ii. For each node $u = (k-1, x_1, x_2)$, embed the subpyramid $P_1(u)$ into $H_3(00g_{k-1}(u); 2k-1, 2k, 2k+1)$ using the mapping:

- g_1, if $x_1, x_2 \in \{0\}_2$,
- g_1^V, if $x_1 \in \{0\}_2$, $x_2 \in \{1\}_2$,
- g_1^H, if $x_1 \in \{1\}_2$, $x_2 \in \{0\}_2$, and
- g_1^{VH}, if $x_1, x_2 \in \{1\}_2$

CHAPTER 4

COMPARATIVE ANALYSIS

This chapter carries out a comparative analysis that involves all four mapping algorithms of the previous chapter. This analysis is based on analytical techniques and actual runs on a Connection Machine CM-2 system consisting of 16K PEs.

4.1 Analytical Techniques

Patel-Ziavras' algorithm has maximum dilation three and maximum congestion two, while these measures for Stout's algorithm are equal to two. Both algorithms need an H_{2n} hypercube to embed the P_n pyramid. Thus, Patel-Ziavras' algorithm will be inferior to Stout's algorithm with respect to communication overheads since the maximum dilation is increased by one. Nevertheless, if several levels of the pyramid are required to be active simultaneously, then Patel-Ziavras' algorithm will be superior to Stout's algorithm with respect to reduced execution times and high utilization of the target hypercube's resources. This is because Stout's algorithm can not support simultaneous simulation of multiple pyramid levels. In contrast, the only type of concurrency not allowed by Patel-Ziavras' algorithm is the simultaneous simulation of the leaf level along with other higher levels.

Lai-White's algorithms I and II have maximum dilation three and two respectively, and maximum congestion two and three respectively. In addition, they require double the number of PEs required by Patel-Ziavras' and Stout's algorithms, so their cost is much higher. This is because distinct nodes of the pyramid are mapped onto distinct PEs of the hypercube in order to allow the simultaneous simulation of any subset of pyramid levels. Therefore, Patel-Ziavras' algorithm is a compromise between Stout's algorithm and the pair of Lai-White's algorithms with respect to cost and performance for applications that require simultaneous simulation of multiple levels of the pyramid.

Patel-Ziavras' algorithm should be expected to yield lower performance than Lai-White's both algorithms if there is a need for simultaneous simulation of the leaf level along with other higher levels of the pyramid. However, when compared to Stout's algorithm, the communication delay in Lai-White's algorithms due to the increased dilation may prohibitively increase the communications overhead for application algorithms that do not require the simultaneous simulation of multiple pyramid levels. In addition, Stout's algorithm implements in the latter case smaller amounts of vertical data transfers.

4.2 Connection Machine Results

This subsection presents and analyzes results obtained from actual runs of some image processing algorithms on a Connection Machine CM-2 system consisting of 16K PEs.

We must emphasize that the results presented here are not always indicative of the performance of pure hypercube systems because of two reasons. Firstly, the routers become the bottlenecks for communication intensive operations because any single router node is shared by sixteen PEs. To alleviate this problem, the majority of the results presented here use one PE per router node. Secondly, for algorithms where many-to-one communication operations are followed by the application of associative (reduction) operations on the received data, the Connection Machine routers implement the reduction "on the fly," thus reducing the amount of traffic going to distant PEs. In the presentation of the results below, the influence of both issues on the Connection Machine's performance is discussed.

Results are presented for three image processing algorithms. The first algorithm finds the perimeter of objects in images, the second algorithm performs 2-dimensional convolution and the third algorithm performs image segmentation.

4.2.1 Finding the Perimeter of Objects

This application algorithm assumes the assignment of a single pixel to each node at the base of the pyramid and, for the sake of simplicity, the existence of a single object in the image. Assuming that the boundary pixels are known, a bottom-up process is applied to count the total number of boundary pixels. More detail follows. Nodes at the base of the pyramid that contain a boundary pixel send 1 to their parent at level 1, while base level nodes that do not contain a boundary pixel send 0 to their parent. Nodes at level 1 add the four values sent by their children and send the result to their parent at level 2. To reduce the overall communication overhead, the latter addition is performed as a reduction operation, where router nodes add the values on the fly before they reach their destination. This process continues with higher levels until the apex is reached. The addition of the values received by the apex is the perimeter of the object.

Results were obtained for two cases. In the first case, all sixteen PEs attached to any single router node are used, for a total of 16K "active" PEs. Therefore, the base of the pyramid assumed by Stout's and Patel-Ziavras' algorithms is $2^7 \times 2^7$ (i.e., eight levels), while the base of the pyramid assumed by Lai-White's algorithms is $2^6 \times 2^6$ (i.e., seven levels). In the second case, in order to reduce the communication overhead for a "pure" hypercube network, only one PE per router node is used, for a total of 1K "active" PEs. Therefore, the base of the pyramid assumed by Stout's and Patel-Ziavras' algorithms is now $2^5 \times 2^5$ (i.e., six levels), while the base of the pyramid assumed by Lai-White's algorithms is $2^4 \times 2^4$ (i.e., five levels). Average times calculated over several runs are presented here.

Table 4.1 shows results for the algorithm that finds the perimeter of an object when using all 16K PEs in the system. *Base* represents the amount of time it takes to send data from the base level to the parents at level 1. This process takes a relatively large amount of time because all PEs are active and share router nodes in groups of sixteen. We also need to emphasize that all data transfers in our implementation involve integer variables; this was chosen for uniformity reasons, because several algorithms in image processing have a lot of similarities with the perimeter counting algorithm but they deal with integer variables. *Top* represents the amount of time it takes the level located immediately below the highest level to send data to the topmost level and for the topmost level to process the received data. *Total* represents the total amount of time taken by the algorithm. Table 4.1 shows that Lai-White's algorithm II is associated with the worst performance. This can be explained as follows. While all 16K PEs of the system are initially used, PEs involved in the simulation of higher levels of the pyramid do not share router nodes for Stout's, Patel-Ziavras' and Lai-White's algorithm I. In contrast, Lai-White's algorithm II is such that for the simulation of higher levels of the pyramid, multiple PEs attached to the same router node become "active." Therefore, the communication overhead is tremendously increased for Lai-White's algorithm II. In addition, we may observe that first three algorithms are characterized by almost similar performance for this image processing problem.

As it can be observed from earlier paragraphs, the relatively small value of the dilation in these algorithms does not have a very critical influence here due to CM-

Algorithms	Base	Top	Total	Levels
Stout	1.89	0.56	5.53	8
Patel-Ziavras	1.53	0.46	4.99	8
Lai-White I	1.11	0.54	3.66	7
Lai-White II	2.52	0.31	10.66	7

Table 4.1: Finding the perimeter of an object: one level active at a time. 16 PEs per router node. (Times in msec for CM-2.)

Algorithms	Base	Top	Total	Levels
Stout	0.59	0.55	2.84	6
Patel-Ziavras	0.64	0.55	2.86	6
Lai-White I	0.64	0.55	2.35	5
Lai-White II	0.59	0.51	2.13	5

Table 4.2: Finding the perimeter of an object: one level active at a time. 1 PE per router node. (Times in msec for CM-2.)

2's reduction operations and the implementation of petit cycles for transfers of data. In addition, the congestion is not a critical factor here due to the SIMD mode of computation and the simulation of one level at a time. Finally, Lai-White's algorithm I and, more importantly, Lai-White's algorithm II produce "irregular" embeddings that may increase the overhead resulting from the control structure of application algorithms. We have dramatically reduced this overhead by generating pointers to parents, children and lateral neighbors during initialization.

Table 4.2 shows results when only one PE per router node is initially active. As expected, all four mapping algorithms are characterized by almost similar performance because of the special architecture of the Connection Machine, as discussed before, and the fact that at no point during the execution of the algorithm does any router node serve multiple PEs.

Algorithms	Base	Other Levels	Total	Levels
Patel-Ziavras	0.69	0.68	1.37	6
Lai-White I			0.67	5
Lai-White II			0.87	5

Table 4.3: Finding the perimeter of multiple objects; pipelining 1 PE per router node. (Times in msec for CM-2.)

Table 4.3 shows steady state results for the same image processing algorithm, assuming that pipelining is applied and that the number of active PEs attached to a single router node is initially one. More specifically, we assume that either multiple images are processed, where each image contains a single object, or a single image that contains multiple objects is processed. Results are not shown for the Stout algorithm as it can not be pipelined for the reasons discussed earlier. In spite of the limitations imposed by the SIMD mode of computation implemented on the Connection Machine, pipelining for this image processing algorithm is feasible due to the fact that all of the operations carried out at different levels, or a subset of the operations for the base and the apex, are identical. *Total* in Table 4.3 represents the amount of time it takes to produce a single output in steady state (i.e., all stages of the pipeline are full). Patel-Ziavras' algorithm yields the lowest performance of the three algorithms due to the fact that the base level can not be simulated along with other levels. In fact, Table 4.3 shows that the time taken by the base level for Patel-Ziavras' algorithm is approximately half of the total time. In contrast, all of the pyramid's levels may be active simultaneously when Lai-White's algorithms are used.

4.2.2 2D Convolution

Two-dimensional convolution using the pyramid structure was the second image processing algorithm that we implemented on the Connection Machine. The convolution algorithm convolves a $k \times k$ window of weighting coefficients with a $2^n \times 2^n$ image matrix. Let $X = \{x_{i,j}\}$ and $W = \{w_{i,j}\}$ be the image matrix and the window respectively. The goal is to compute $Y = \{y_{r,s}\}$ where

$$y_{r,s} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} w_{i,j} \times x_{r+i,s+j}$$

with $0 \leq r, s \leq 2^n - k$. This algorithm is very frequently applied in image processing.

The convolution algorithm for the source pyramid structure is as follows. We assume the assignment of a single pixel per node in the base of the pyramid. The smallest integer γ is then found for which $2^\gamma \geq k$. Then the base of the pyramid is partitioned into square blocks of size $2^\gamma \times 2^\gamma$. Each such partition contains the leaves of a subpyramid whose apex is at level γ. The weighting coefficients are then loaded into the upper leftmost part of each partition. This can be implemented on a pyramid machine using a top down process, assuming that the coefficients are contained in the apex [17]. On the Connection Machine, the coefficients are loaded using k^2 broadcasting operations with appropriate sets of PEs selected each time. This part was not included in the total execution time of the presented results. The rest of the PEs in each partition receive a zero as the weighting coefficient if the window size is smaller than the partition size. The PEs then multiply the weighting coefficient with the pixel value they contain and send the result to their parent. Parents at level 1 add the values they receive from their children and send the result to their parent. This process continues until the apexes of the subpyramids are reached. Each apex at level γ adds the values it receives from its children and sends the result, through the necessary intermediate PEs at lower levels, to the leaf PE in the upper leftmost corner of its partition. Each window at the base that contains the weighting coefficients is then shifted to the right once, multiplications are performed as above, the results are shifted to the left once, and the values are sent to the parents at level 1. The bottom-up and top-down processes described earlier are then applied, with the result now stored in the PE with offset (0,1) in the partition. To conclude, the convolution algorithm involves lateral shifts and multiplications at the base, bottom-up addition of numbers, and finally top-down transmission of final results. These steps are repeated $2^{2\gamma}$ times, which is equal to the total number of PEs in each partition.

Results are presented in Table 4.4 for windows with k from 2 to 8. Note that the number of levels in the pyramids is not shown. This is because only levels 0 through γ are involved in the algorithm. The results in Table 4 indicate that there are not any significant differences among the timings of the four mapping algorithms. Therefore, this observation suggests that Stout's and Patel-Ziavras' algorithms are more appropriate than Lai-White's algorithms due to the lower cost of the system that the former two algorithms require. The same table also shows the total time for lateral data transfers at level 0.

Then an attempt was made to pipeline the convolution process. The only part of the entire process which can be pipelined is the bottom-up and top-down communication phases, while the weighting coefficients are multiplied with pixel values concurrently within each partition.

The sequence of operations for this process is entirely different from the previous one. All the weighting coefficients are loaded in each partition from the front end as previously defined. Next all these coefficients are multiplied with the pixel values and the results are stored in an array within each PE. The window is laterally shifted to the right and the whole precess is repeated for PE(0,1) within each partition. This entire shifting and multiplication process is completed and no

Algorithms	Lateral	Total	$k \times k$
Stout	3.36	17.61	
Patel-Ziavras	3.36	17.82	2×2
Lai-White I	5.21	20.24	
Lai-White II	4.82	21.90	
Stout	12.78	109.81	
Patel-Ziavras	12.53	105.89	3×3
Lai-White I	18.55	116.51	
Lai-White II	18.01	107.86	
Stout	12.66	104.38	
Patel-Ziavras	12.66	103.56	4×4
Lai-White I	18.58	114.49	
Lai-White II	18.01	106.17	
Stout	46.69	577.16	
Patel-Ziavras	45.96	547.88	5×5
Lai-White I	69.38	593.91	
Lai-White II	67.64	547.67	
Stout	47.06	558.76	
Patel-Ziavras	45.81	518.26	6×6
Lai-White I	69.27	594.09	
Lai-White II	67.48	547.67	
Stout	46.28	554.14	
Patel-Ziavras	46.82	550.76	7×7
Lai-White I	70.01	599.93	
Lai-White II	67.66	551.07	
Stout	49.21	543.16	
Patel-Ziavras	45.83	544.38	8×8
Lai-White I	69.26	584.81	
Lai-White II	67.33	541.15	

Table 4.4: 2D convolution. 1 PE per router node. (Times in msec for CM-2.)

bottom-up or top-down communication takes place during this operation. Then all these results are sent to the coordinator PEs at level γ in a pipelined fashion. These values are loaded into an array at level γ. Finally, all the coordinator PEs at level γ send all these values back to the leaf level PEs and these values are loaded within each partition in correct PE. This top-down communication phase is also pipelined.

Execution times are improved for Lai-White's algorithms about 40 to 45 percent. Results are much improved for larger window sizes (greater than 4). For window size 2×2, execution time is slightly increased. This is due to a different control structure in the main program.

For Patel-Ziavras' algorithm, results are not much improved as compared to non-pipelined results. Pipelining is not implemented for window size less than 5 because the benefit of the pipelining can not be obtained due to the fact that the base can not be active simultaneously with the other levels of the pyramid.

Implementation of pipelining for this algorithm is similar to that of the Lai-White's except that the bottom-up and top-down communication phases take two cycles. All the results are sent from the base to level 1 and then from level 1 to level γ. The same procedure is repeated for top-down communication. Control structure for this two-step pipelining procedure does not allow to take much advantage of pipelining. However, the results of the pipelining will be much improved for large window sizes, when level γ will be at much higher level as compared to window sizes 5 to 8. Results are shown in Table 4.5 .

4.2.3 Segmentation

Segmentation is the process which partitions the image into regions with more or less homogeneous properties. A cooperative, iterative approach to segmentation

Algorithms	Total	$k \times k$
Patel-Ziavras	-	2×2
Lai-White I	24.41	
Lai-White II	24.35	
Patel-Ziavras	-	3×3
Lai-White I	83.68	
Lai-White II	79.56	
Patel-Ziavras	-	4×4
Lai-White I	83.57	
Lai-White II	79.03	
Patel-Ziavras	482.10	5×5
Lai-White I	322.20	
Lai-White II	307.69	
Patel-Ziavras	493.404	6×6
Lai-White I	321.36	
Lai-White II	309.17	
Patel-Ziavras	500.56	7×7
Lai-White I	329.36	
Lai-White II	308.11	
Patel-Ziavras	481.65	8×8
Lai-White I	320.83	
Lai-White II	307.59	

Table 4.5: Pipelining 2D convolution. 1 PE per router node. (Times in msec for CM-2.)

in which each process at a given iteration is used to adjust the other process at the next iteration is used here [21]. This approach uses an overlapped pyramid that implements 50 percent overlapping in each direction. Thus each node has four parents and 16 children. A son-father relationship is defined between nodes in adjacent layers, but unlike other pyramids, this relationship is not fixed and may be redefined at each iteration.

There are four time dependent variables associated with each node (PE):

- $c[i, j, l][t]$: the value of the local image property;
- $a[i, j, l][t]$: the area over which the property was computed;
- $p[i, j, l][t]$: a pointer to the node's father at the next higher level;
- $s[i, j, l][t]$: the segment property, the average value for the entire segment containing the node.

t is the iteration number. The value of c at each leaf level node is set equal to the corresponding image sample value, while the c value for each lower level node is the average of all 16 of the node's candidate sons. Iterations following the initialization ($t > 0$) are divided into three phases.

Phase 1: Son-father links are established for all nodes below the top of the pyramid according to the following condition:

If $d[m] < d[n]$ for all $n \neq m$, then $p[i, j, l][t] = m$, where $d[n]$ is the absolute difference between the c value of node $[i, j, l]$ and its n^{th} candidate father. If two or more of the candidate fathers are equally likely, then decision is made at random.

Phase 2: The c and a values are computed bottom up on the basis of the new son-father links.

For $l = L$, $a[i, j, l][t] = 1$, $c[i, j, l][t] =$ image sample value $I(x, y)$, where L is the

Object	Stout	Patel-Ziavras	Lai-White I	Lai-White II	Iterations
I	246.68	255.78	212.76	212.04	5
II	247.27	253.81	222.45	212.17	5
III	352.78	351.65	310.76	322.55	8
IV	353.57	353.56	309.29	321.74	8
V	349.92	351.02	311.18	322.65	8

Table 4.6: Image segmentation for five different objects. 1 PE per router node. (Times in msec for CM-2.)

leaf level.

For $l > L$, $a[i,j,l][t]$ = sum of the areas of the linked children.

$c[i,j,l][t]$ = sum of the c values of linked children, $a[i,j,l][t]$.

Phase 3: Segment values are assigned in a top down fashion. At the topmost level, the segment value of each node is set equal to its local property value $s[i,j,l][t] = c[i,j,l][t]$.

For higher levels, each node's value is just that of its father.

At the end of phase 3, the highest level segment values represent the current state for the smoothing-segmentation process. Any change in pointers in a given iteration will result in changes in the values of local image properties associated with pyramid nodes. These changes may alter the nearest father relationship and necessitate a further adjustment to pointers in the next iteration. Changes always shift the boundaries of segments in a direction which makes their contents more homogeneous, so convergence is guaranteed. The iterative process is continued until no changes occur from one iteration to the next. The results for the segmentation problem are shown in Table 4.6.

Results for all four mapping algorithms are almost the same The execution time for Stout's and Patel-Ziavras' algorithms are higher than that of Lai-White's algorithms. This is due to an extra pyramid level for Stout's and Patel-Ziavras' algorithms. Five different objects are used for the segmentation and estimation of image region properties. All links become stable after 5 or 8 iterations. These results suggest the use of Stout's and Patel-Ziavras' algorithms for such applications because they require a smaller dimension hypercube (i.e., H_{2n}) as compared to the hypercube required for Lai-White's (i.e., H_{2n+1}).

CHAPTER 5

CONCLUSIONS

This thesis has carried out a comparative analysis of algorithms that map pyramids onto hypercubes. The comparative analysis incorporates both analytical techniques and actual runs on a Connection Machine CM-2 system composed of 16K processors. The results show that while Stout's and Patel-Ziavras' algorithms require target systems with approximately half the cost of those required by Lai-White's algorithms, Stout's algorithm is not capable of simulating multiple levels of the pyramid simultaneously. Since a wide variety of pyramid algorithms can take advantage of concurrent multilevel computations, this restriction is a major drawback of Stout's algorithm. However, Stout's algorithm has the lowest dilation and congestion which result in the lowest communication times between adjacent levels.

In contrast, Patel-Ziavras' algorithm does not impose this restriction. For concurrent multilevel computations, the results on the Connection Machine indicate that this algorithm achieves very good performance when compared to Lai-White's algorithms, at half the cost.

When one level of the pyramid is considered to be active at a time, Patel-Ziavras' and Lai-White's algorithms perform almost the same because of very similar communication times between adjacent levels. Therefore, Patel-Ziavras' algorithm is a compromise between the pair of Lai-White's algorithms and Stout's algorithm with respect to cost and performance.

The performance of Patel-Ziavras' and Lai-White's algorithms is also investigated for pipelined processing. Stout's algorithm can not implement pipelining because a subset of PEs simulate all the levels of the pyramid. Lai-White's algorithms perform better than Patel-Ziavras' algorithm when it is required to simulate a pyramid with all of its levels active at the same time. Patel-Ziavras' algorithm imposes a restriction on the base of the pyramid which can not be active simultaneously with the other levels. Therefore, two pipelined stages are required for Patel-Ziavras' algorithm. One stage works between the base and the next higher level, while the other stage works for all of the remaining levels, except the base. Good performance of pipelined processing can be obtained for Patel-Ziavras' algorithm if a large pyramid structure is used, i.e., the apex is at a very high level.

APPENDIX

```
/*****************************************************************
**                                                             **
**      **************   stout Algorithm. ********************   **
**                       one PE / Router                       **
**                    C O N V O L U T I O N                    **
*****************************************************************/

#include        <stdio.h>
#include        <cstimer.h>

     shape[512][32] base;
/* 512 / 16 =32; hence a regular shape of 32x32 will be obtained     */

/*  Integer x and y contain  decimal values of x and y coordinates.  */
/*  While gx and gy contain Reflexive Gray codes for each x and y    */
/*  coordinates.                                                     */
/*  index0 and index1 contain x and y coordinates for each master Pe */
/*  'level' contains the level number for each PE . level=0 means    */
/*  PE is active only for base. level=7 means this PE is active for  */
/*  all levels. 'dest' stores the reslut of each level operation.    */
/*  b,c and d are used as temporary storage variables.               */


        int:base px,py, x,y,xx,yy,gx,gy, level=9, p=0, q=0 ,r=0;
        int:base test=1,index0,index1,dest1,dest2,dest3,dest4,dest5,con_val=0,co
n_result=0, fcon_result=0, mul=0;
        static int e=0,f=0;
        static int count=0;
        int row,  col, k=0, clk_tck=0, l,m,n, coord_pe=0 ;
        int:base a=0,b=0,c=0,d=0, addr=0, mark=0  ;
        int:base x0,y0,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7;
        int:base ch1x1,ch1y1,ch2x1,ch2y1,ch3x1,ch3y1,ch4x1,ch4y1;
        int:base ch1x2,ch1y2,ch2x2,ch2y2,ch3x2,ch3y2,ch4x2,ch4y2;
        int:base ch1x3,ch1y3,ch2x3,ch2y3,ch3x3,ch3y3,ch4x3,ch4y3;
        int:base ch1x4,ch1y4,ch2x4,ch2y4,ch3x4,ch3y4,ch4x4,ch4y4;
        int:base ch1x5,ch1y5,ch2x5,ch2y5,ch3x5,ch3y5,ch4x5,ch4y5;
        int:base southx,southy,northx,northy,eastx,easty,westx,westy;

        int win_size=0, win_odd=0, part_size=0, no_of_part=0;
        int:base win_val[64];   /* max. window value is 8 x 8 */
        int window_val[64]; /* scalor array */

/* south , north, east, west variables are used for lateral communication. */
/* ch1x1 means children 1, x-axis value, for level 1 */
/* x0 to x7 and y0 to y7 variables stores address of master PE for  */
/* each level of Pyramid.                                           */
        double time_val, val, val1;

main()
{
        with(base){
                xx=pcoord(1);
                yy=pcoord(0);
        for(col=0; col<32;col++){
           for(row=0;row<512;row++){
                if(row==0 && col==0) k=0;
                  else k+=1;
                [row][col]addr=k; }  }

        a=0x0F;  b=a&addr;

        where(b==0) { /*   b=0, means those PEs whose 4LSBs =0 */
                mark=1; }

                k=0;
for(row=0;row<512;row+=16){ if(row==0) k=0; else k+=1;
```

```
  for(col=0; col<32; col++){
        [row][col]x=col ;  [row][col]y=k ;
                                  }  }
where(mark==1)  {
        dec_to_gray();
        level=0;
        /* set the level number */
        q=1;
        for(k=0;k<5;k++){
          p=gx|gy;
          where(level==k){
          p=q&p;
          where(p==0) level++;
          r=q;
          r<<=1;
          q|=r;
                          }
                          }
        alg();

        printf("\n");
                }
/*
for(row=0;row<128;row++){
  for(col=0; col<32; col++){
        k=[row][col]b;
 printf("\n [%d][%d]level = %d\t y=%d \t gx=%d \t gy=%d ",row,col,[row][col]leve
l,[row][col]y,[row][col]gx,  [row][col]gy ),}}
*/

        } /* with(base) loop ends here */
}



dec_to_gray()
{
        gx=x>>1; gx^=x ; gy=y>>1; gy^=y  ;
}

alg()
{       /* first calculate the addresses of master PEs and store in
        variables x[n] and y[n] for each Pe. */

        index0=gx;  index1=gy;
        index0>>=1; index0<<=1;  index1>>=1 ; index1<<=1;
        gray_to_dec();   x0=index0;  y0=index1; y0<<=4;

        index0=gx;  index1=gy;
        index0>>=2, index0<<=2;  index1>>=2 ; index1<<=2;
        gray_to_dec();   x1=index0;  y1=index1; y1<<=4,

        index0=gx;  index1=gy;
        index0>>=3; index0<<=3;  index1>>=3 ; index1<<=3;
        gray_to_dec();   x2=index0;  y2=index1; y2<<=4;

        index0=gx;  index1=gy;
        index0>>=4; index0<<=4;  index1>>=4 ; index1<<=4,
        gray_to_dec();   x3=index0;  y3=index1; y3<<=4;

        index0=gx;  index1=gy;
        index0>>=5; index0<<=5;  index1>>=5 ; index1<<=5,
        gray_to_dec();   x4=index0;  y4=index1; y4<<=4;
/*
        for(row=0;row<512;row+=16){ printf("\n row = %d \n",row);
          for(col=0;col<32;col++){
```

```
printf("\n[%3d][%3d]x0=%d,  y0=%d  ", row, col,[row][col]x0, [row][col]y0  );
                                        }  }
*/
              /* Now calculate the addresses of each children ( total 4)
                for each master Pe and store in variable  ch[n]x[k] */

       where(level>=1)
              {index0=x; index1=y;
               ch1x1=index0; ch1y1=index1;     /* same child 1. */
               ch1y1<<=4;
              index0=gx;index1=gy;index0|=1;gray_to_dec();
              ch2x1=index0; ch2y1=index1;  /* child 2 at 1 x-axis distance.*/
              ch2y1<<=4;
              index0=gx; index1=gy; index1|=1; gray_to_dec();
                              /* child 3 at 1 y-axis distance. */
              ch3x1=index0; ch3y1=index1;
              ch3y1<<=4;
              index0=gx; index1=gy; index0|=1; index1|=1; gray_to_dec();
                      /* child 4 at 1 x-axis distance & 1 y-axis distance.*/
              ch4x1=index0; ch4y1=index1;
              ch4y1<<=4;  }


       where(level>=2)
              {index0=x; index1=y;
               ch1x2=index0; ch1y2=index1;     /* same child 1. */
               ch1y2<<=4;
              index0=gx;index1=gy;index0|=2;gray_to_dec();
              ch2x2=index0; ch2y2=index1;  /* child 2 at 1 x-axis distance.*/
              ch2y2<<=4;
              index0=gx; index1=gy; index1|=2; gray_to_dec();
                              /* child 3 at 1 y-axis distance. */
              ch3x2=index0; ch3y2=index1;
              ch3y2<<=4;
              index0=gx; index1=gy; index0|=2; index1|=2; gray_to_dec();
                      /* child 4 at 1 x-axis distance & 1 y-axis distance.*/
              ch4x2=index0; ch4y2=index1;
              ch4y2<<=4;  }

       where(level>=3)
              {index0=x; index1=y;
               ch1x3=index0; ch1y3=index1;     /* same child 1. */
               ch1y3<<=4;
              index0=gx;index1=gy;index0|=4;gray_to_dec();
              ch2x3=index0; ch2y3=index1;  /* child 2 at 1 x-axis distance.*/
              ch2y3<<=4;
              index0=gx; index1=gy; index1|=4; gray_to_dec();
                              /* child 3 at 1 y-axis distance. */
              ch3x3=index0; ch3y3=index1;
              ch3y3<<=4;
              index0=gx; index1=gy; index0|=4; index1|=4; gray_to_dec();
                      /* child 4 at 1 x-axis distance & 1 y-axis distance.*/
              ch4x3=index0; ch4y3=index1;
              ch4y3<<=4;  }


       where(level>=4)
              {index0=x; index1=y;
               ch1x4=index0; ch1y4=index1;     /* same child 1. */
               ch1y4<<=4;
              index0=gx;index1=gy;index0|=8;gray_to_dec();
              ch2x4=index0; ch2y4=index1;  /* child 2 at 1 x-axis distance.*/
              ch2y4<<=4;
              index0=gx; index1=gy; index1|=8; gray_to_dec();
                              /* child 3 at 1 y-axis distance. */
              ch3x4=index0; ch3y4=index1;
              ch3y4<<=4;
              index0=gx; index1=gy; index0|=8; index1|=8; gray_to_dec();
                      /* child 4 at 1 x-axis distance & 1 y-axis distance.*/
              ch4x4=index0; ch4y4=index1;
              ch4y4<<=4;  }


       where(level>=5)
              {index0=x; index1=y;
               ch1x5=index0; ch1y5=index1;     /* same child 1  */
               ch1y5<<=4;
              index0=gx;index1=gy;index0|=16;gray_to_dec();
              ch2x5=index0; ch2y5=index1;  /* child 2 at 1 x-axis distance */
              ch2y5<<=4;
              index0=gx; index1=gy; index1|=16; gray_to_dec();
                              /* child 3 at 1 y-axis distance  */
              ch3x5=index0; ch3y5=index1;
              ch3y5<<=4;
              index0=gx; index1=gy; index0|=16; index1|=16, gray_to_dec();
                      /* child 4 at 1 x-axis distance & 1 y-axis distance.*/
              ch4x5=index0; ch4y5=index1;
              ch4y5<<=4;  }

/* Children addresses calculation ends here. */

       /* Now calculate parameters for lateral communication  */
       where(level>=0) /* for base */
       {        northx=x; northy=(y-1);
                 southx=x; southy=(y+1);
                 eastx=(x+1); easty=y;
                 westx=(x-1); westy=y;
       where(northy==-1) northy=0;   where(westx==-1)  westx=0;
       where(eastx==32)  eastx=31;  where(southy==32)  southy=31;
       southy<<=4;       northy<<=4; easty<<=4; westy<<=4;
       }

/*
        for(row=0;row<64;row+=16){ printf("\n row = %d \n",row);
         for(col=0;col<32;col++){
  printf("\n[%3d][%3d]northx  = %d , northy  = %d  ", row, col,[row][col]northx
 ,[row][col]northy  );
  printf("\n[%3d][%3d]southx  = %d , southy  = %d  ", row, col,[row][col]southx
,[row][col]southy  );
  printf("\n[%3d][%3d]eastx  = %d , easty  = %d  ", row, col,[row][col]eastx  ,[
row][col]easty  );
  printf("\n[%3d][%3d]westx = %d , westy = %d  \n", row, col,[row][col]westx  ,[
row][col]westy  );
                                        }  }
*/

       printf("\n Your PYRAMID COMPUTER base is 32 x 32 \n"),
wind:   printf("\nFor k x k size window, Enter value of k\n");
       scanf("%d",&win_size);
               if(win_size <=1 || win_size>8)  {
  printf("\n wrong value : Valid window sizes are 2,3,4,5,6,7,8 \n"); goto wind;
  }

               printf("\n You've to enter total %d elemets; starting from 0 to
%d ", win_size*win_size,((win_size*win_size)-1));
               printf("\nEnter window elements by rows: ");
       for(k=0;k<(win_size*win_size); k++){
               printf("\n Enter element #%d  :",k);
               scanf("%d",&n  );
               win_val[k]=n   ;
                                       }
       for(part_size=1; part_size<win_size ; part_size*=2);
```

```
                m=32/part_size;
        printf("\n You entered Window size %d x %d ",win_size,win_size);
        printf("\n partition  size = %d x %d ", part_size, part_size ),
        printf("\n PC base is divided into  %d partitions of size %d x %d ", m,p
art_size, part_size   );
        if(part_size>win_size) win_odd=1;
        if(m==16){ where(level==1) coord_pe=1;}
                else if(m==8){ where(level==2) coord_pe=2;}
                        else if(m==4){ where(level==3) coord_pe=3;}
/* coord_pe shows the level number of coordinator PEs. */
                printf("\n Coordinator  PEs are at level  %d ", coord_pe );

/* Now initialize each partition with its own x y coords */
        where(x<part_size) px=x;
        where(y<part_size)  py=y;
        where(x>=part_size ) px=x%part_size;
        where(y>=part_size ) py=y%part_size;

/*********************************************************/
        where((x%3)==0) test=0;

printf("\n\n STARTING CONVOLUTION PROCESS....\n");
/* assign corresponding value of window elements to parallel variable a */
                a=(win_size * py)+px;
                con_val=win_val[a];  /* con_val gets the right value of its own
                                        index for first convolution process */

        if(win_odd==1){where(px>=win_size) con_val=0;
                        where(py>=win_size)con_val=0 ; }


/*      repeat: printf("\n\n con process row#=%d  col=%d ",e,f);  */
CMC_timer_start(1);
repeat: con_result=con_val*test;
        adjust_for_com();
CMC_timer_start(2);
        send();
CMC_timer_stop(2);
CMC_timer_start(3);
        adjust_window();
CMC_timer_stop(3);
        if(count!=10) goto repeat;
CMC_timer_stop(1);
        [0][1]fcon_result=[0][4]fcon_result ;
/*
        for(row=0;row<32;row+=16){
                for(col=0;col<32;col++){
printf("\n [%d][%d]test = %d     fcon_result= %d ",row,col,[row][col]test, [row]
[col]fcon_result);

} }  */

        time_val=CMC_timer_read_cm_busy(1);
        val=CMC_timer_read_cm_busy(2);
        val1=CMC_timer_read_cm_busy(3);

        printf("\n*********************************************************");
        printf("\n\t Total time CM busy for Convolution  is %f ",time_val);
        printf("\n\t Total time to send values to coordinator\n          PEs an
d get the result back is  %f ", val);
        printf("\n\t Total time CM takes to move the window is  %f ", val1);
        printf("\n\n");
}


adjust_for_com()
{       /* send con_val to top left corner of each partition */
        if(e==0 && f==0) return; /*e=0 and f=0 means start of convolution */
        else {
        a=e-1;
        if(e>0){ [westy][westx-a]con_result=con_result,
/*
printf("\n [%d][%d]con_result = %d ",f,e, [0][0]con_result ),
printf("\n [%d][%d]con_result = %d ",f,e, [0][1]con_result );
printf("\n [%d][%d]con_result = %d ",f,e, [0][2]con_result );
printf("\n [%d][%d]con_result = %d ",f,e, [0][3]con_result ); */
                }

        m=f-1;
        a=f-1; if(m>0)  a<<=4;
        if(f>0){   [northy-a][northx]con_result=con_result;
/*
printf("\n [%d][%d]con_result = %d ",f,e, [0][0]con_result ),
printf("\n [%d][%d]con_result = %d ",f,e, [0][1]con_result );
printf("\n [%d][%d]con_result = %d ",f,e, [0][2]con_result );
printf("\n [%d][%d]con_result = %d ",f,e, [0][3]con_result ), */
                }
        if(win_odd==1){where(px>=win_size) con_result=0,
                        where(py>=win_size)con_result=0 ; }

        }
/*      for(row=0;row<32;row+=16){
                for(col=0;col<32;col++){
        printf("\n[%d][%d]con_val =%d   con_result =%d ",row,col,[row][col]con_v
al, [row][col]con_result  );
                }       }
*/
}

adjust_window()
            /* When the final value reaches within each partition , then the
               following routine puts the correct value within correct PE. Then
               it shifts the window (to calculate next convolution) */
{
        where(py==f  && px==e )fcon_result=con_result;
        e++;
        con_result=0 ;  /* erase all convolved values now.*/
        dest1=0; dest2=0;dest3=0;dest4=0;dest5=0;

/* Now shift the window for correct position */

if(e<(part_size)) [easty][eastx]con_val=con_val;
        else {
                a=x-(part_size-1);
        where(a==-1) a=0; /* shift window to original position  */
         [westy][a]con_val=con_val; f++; e=0;
/*
printf("\n after to orig. position ");
printf("\n con_val = %d ", [0][0]con_val);
printf("\n con_val = %d ", [0][1]con_val);
printf("\n con_val = %d ", [0][2]con_val);
printf("\n con_val = %d ", [0][3]con_val);
*/
if(f>0 && f<part_size) [southy][southx]con_val=con_val;

                }

  if(f==part_size) {printf("\n returning f= %d ",f); count=10, return;}
/*
        for(row=0;row<48;row+=16){ printf("\n\nwindow values after shifting :");
                for(col=0;col<32;col++){
        printf("\n[%d][%d]con_val =%d   con_result =%d ",row,col,[row][col]con_v
```

```
al, [row][col]con_result  );
                  }        }
*/
}



send() /* This routine sends all values from base to the coordinators and then
          send them back to  the right partition. */

{       l=coord_pe;
        where(level>=0) [y0][x0]dest1+=con_result; /* printf("\n dest 1= %d ", [
0][0]dest1);  */
        l-=1; if(l>0) { where (level>=1) [y1][x1]dest2+=dest1 ;/* printf("\n des
t 2=%d ", [0][0]dest2);*/ }
        l-=1; if(l>0) { where (level>=2) [y2][x2]dest3+=dest2 ;/*  printf("\n de
st 3=%d ", [0][0]dest3); */}
        l-=1; if(l>0) { where (level>=3) [y3][x3]dest4+=dest3 ; /*  printf("\n d
est 4=%d ", [0][0]dest4); */  }
        l-=1; if(l>0) { where (level>=4) [y4][x4]dest5+=dest4 ;/*  printf("\n de
st 5=%d , l = %d ", [0][0]dest5,l) ; */ }



switch(coord_pe){
        case 5: where(level>=5){
                [ch1y5][ch1x5]dest4=dest5;
                [ch2y5][ch2x5]dest4=dest5;
                [ch3y5][ch3x5]dest4=dest5;
                [ch4y5][ch4x5]dest4=dest5;  }

        case 4: where(level>=4){
                [ch1y4][ch1x4]dest3=dest4;
                [ch2y4][ch2x4]dest3=dest4;
                [ch3y4][ch3x4]dest3=dest4;
                [ch4y4][ch4x4]dest3=dest4;  }

        case 3: where(level>=3){
                [ch1y3][ch1x3]dest2=dest3;
                [ch2y3][ch2x3]dest2=dest3;
                [ch3y3][ch3x3]dest2=dest3;
                [ch4y3][ch4x3]dest2=dest3;  }

        case 2: where(level>=2){
                [ch1y2][ch1x2]dest1=dest2;
                [ch2y2][ch2x2]dest1=dest2;
                [ch3y2][ch3x2]dest1=dest2;
                [ch4y2][ch4x2]dest1=dest2;  }

        case 1: where(level>=1){
                [ch1y1][ch1x1]con_result=dest1;
                [ch2y1][ch2x1]con_result=dest1;
                [ch3y1][ch3x1]con_result=dest1;
                [ch4y1][ch4x1]con_result=dest1;  break; }

        default: printf("\n Sorry! Can not perform top down communication\n");

                }
}



gray_to_dec()
{
        int k=0;
```

```
          where(index0>1)  { b=1;
            for(k=0;k<7;k++){
              where(b<=index0)  b<<=1;
                                }
                              }
         where(b!=0)  { b>>=1;  d=b;  b>>=1;  c=b;      }
       for(k=0;k<7;k++){
         where(c!=0){ b^=index0;  b&=c;  d|=b;  c>>=1;  b>>=1;
                    }
                        }
       where(index0>1)  index0=d;
           b=0;c=0;d=0;
where(index1>1)  { b=1;
            for(k=0;k<7;k++){
              where(b<=index1)  b<<=1;
                      }
                              }
         where(b!=0)  { b>>=1;  d=b;  b>>=1;  c=b,      }
       for(k=0;k<7;k++){
         where(c!=0){ b^=index1 ;  b&=c;  d|=b;  c>>=1;  b>>=1;
                    }
                        }
       where(index1>1)  index1=d;
       b=0;  c=0;d=0;
}
```

```
/*****************************************************************
**                                                             **
**    *************   Ziavras  Algorithm. *****************    **
**                     one PE / Router                         **
**                   C O N V O L U T I O N                     **
*****************************************************************/

#include        <stdio.h>
#include        <cstimer.h>

     shape[512][32] base;
/* 512 / 16 =32; hence a regular shape of 32x32 will be obtained     */

/*  Integer x and y contain  decimal values of x and y coordinates.  */
/*  While gx and gy contain Reflexive Gray codes for each x and y    */
/*  coordinates.                                                     */
/*  index0 and index1 contain x and y coordinates for each master Pe */
/*  'level' contains the level number for each PE . level=0 means    */
/*  PE is active only for base. level=7 means this PE is active for  */
/*  all levels. 'dest' stores the reslut of each level operation.    */
/*  b,c and d are used as temporary storage variables.               */


        int:base px,py,x,y,xx,yy,gx,gy, level=9, p=0, q=0 ,r=0, s=0;
        int:base test=1,index0,index1,dest1=0,dest2=0,dest3=0,dest4=0,dest5=0,co
n_val=0,con_result=0, fcon_result=0, mul=0;
        static int e=0, f=0;
        static int count=0;
        int row,  col, k=0, clk_tck=0,l,m,n, coord_pe=0 ;
        int:base a=0,b=0,c=0,d=0,  addr=0, mark=0;
        int:base x0,y0,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7;
/* x0 to x7 and y0 to y7 variables stores address of master PE for
   each level of Pyramid. For base level variables 'index0' and index1'
contain the address of master PE */
        int·base ch1x1,ch1y1,ch2x1,ch2y1,ch3x1,ch3y1,ch4x1,ch4y1;
        int:base ch1x2,ch1y2,ch2x2,ch2y2,ch3x2,ch3y2,ch4x2,ch4y2;
        int:base ch1x3,ch1y3,ch2x3,ch2y3,ch3x3,ch3y3,ch4x3,ch4y3;
        int:base ch1x4,ch1y4,ch2x4,ch2y4,ch3x4,ch3y4,ch4x4,ch4y4;
        int:base ch1x5,ch1y5,ch2x5,ch2y5,ch3x5,ch3y5,ch4x5,ch4y5;
        int:base southx,southy,northx,northy,eastx,easty,westx,westy;

        int win_size=0, win_odd=0, part_size=0, no_of_part=0;
        int:base win_val[64];   /* max. window value is 8 x 8 */
        int window_val[64]; /* scalor array */

/* south , north, east, west variables are used for lateral communication. */
/* ch1x1 means children 1, x-axis value, for level 1 */
/* x0 to x7 and y0 to y7 variables stores address of master PE for  */
/* each level of Pyramid.                                           */
        double time_val, val, val1;

main()
{
        with(base){
                xx=pcoord(1);
                yy=pcoord(0);
        for(col=0; col<32;col++){
           for(row=0;row<512;row++){
                if(row==0 && col==0) k=0;
                  else k+=1;
                [row][col]addr=k; }  }

        a=0x0F;  b=a&addr;

        where(b==0) { /*   b=0, means those PEs whose 4LSBs =0 */
                mark=1; }
                        k=0;
for(row=0;row<512;row+=16){ if(row==0) k=0; else k+=1;
  for(col=0; col<32; col++){
        [row][col]x=col ; [row][col]y=k ;
                                }  }

where(mark==1)
{              /* mark=1 means only those PEs whose 4 LSBs are zero */

        dec_to_gray();  /* gives right values of gx and gy */

        level=0;
        a=gx|gy;
        b=1;
        a&=b;  /* a=0 means LSB is zero */
        where(a==0) level=1;

        a=3;    and_a_b();
        where(a==1 && b==0) level=2;
        a=7;    and_a_b();
        where(a==3 && b==0) level=3;
        a=15;   and_a_b();
        where(a==3 && b==1) level=4;

        [16][1]level=5;  /* apex on 5th level ..  0 to 5 ) */

        address();

/*
for(row=0;row<128;row++){
  for(col=0; col<32; col++){
        k=[row][col]level ;
if(k==0 || k<5)printf("\n [%3d][%3d]gx = %d\t gy=%d \t   level=%d   x1=%d   y1=%
d",row,col,[row][col]gx,[row][col]gy,[row][col]level,[row][col]x1,[row][col]y1),
}       }  */


                }       /* where(mark=1) loop  ends here   */

          } /* with(base) loop ends here */

}         /* program main() loop ends here */

and_a_b()
{
        b=a&gy; a=a&gx;
}


address()
{
        where(level==1)
        {       index1=gx;  index1>>=2; index1<<=2;
                index1|=1;
                index0=gy;  index0>>=2;  index0<<=2;
                        gray_to_dec();
                x1=index0;   y1=index1; x1<<=4;
        }
        where(level==2)
        {       index1=gx;  index1>>=3; index1<<=3,
                index1|=3;
                index0=gy;  index0>>=3;  index0<<=3;
                        gray_to_dec();
                x1=index0;   y1=index1; x1<<=4;
        }
```

```
        where(level==3)
        {       index1=gx;  index1>>=4; index1<<=4;
                index1|=3;
                index0=gy;  index0>>=4;  index0<<=4;
                index0|=1;
                        gray_to_dec();
                x1=index0;   y1=index1; x1<<=4;
        }
        where(level==4)
        {       index1=gx;  index1>>=5; index1<<=5;
                index1|=1;
                index0=gy;  index0>>=5;  index0<<=5;
                index0|=1;
                        gray_to_dec();
                x1=index0;   y1=index1; x1<<=4;
        }

        a=gx;    a>>=1;   a<<=1;  /*  index0 and index1 contain address */
        index0=a;                 /* of master PE for base only. */
        a=gy;  a>>=1;  a<<=1;
        index1=a;
        gray_to_dec();
        index1<<=4;    x0=index1;  y0=index0;

/*****  Now calculate the addresses of children's x and y values for each
        PE of each level. These values will be stored in each PE's variables
        defined above. Every PE will have 8 values ( both x and y coordinates)
        for all 4 children
******/

        where(level==1)   /*For level 1 to level 0 communication valuse are same
 as stout alg. */
                {p=index0; q=index1  ;  /* save index 0 & 1 */
                 index0=x; index1=y;
                 ch1x1=index0; ch1y1=index1;    /* same child 1. */
                 ch1y1<<=4;
                index0=gx;index1=gy;index0|=1;gray_to_dec();
                ch2x1=index0; ch2y1=index1;  /* child 2 at 1 x-axis distance.*/
                ch2y1<<=4;
                index0=gx; index1=gy; index1|=1; gray_to_dec();
                                  /* child 3 at 1 y-axis distance. */
                ch3x1=index0; ch3y1=index1;
                ch3y1<<=4;
                index0=gx; index1=gy; index0|=1; index1|=1; gray_to_dec();
                         /* child 4 at 1 x-axis distance & 1 y-axis distance.*/
                ch4x1=index0; ch4y1=index1;
                ch4y1<<=4;
                index0=p;   index1=q; }

        where(level==2)
                {p=index0;  q=index1; /* save index0 & 1 values */
                 r=gx ;s=gy;
                 r>>=1;   r<<=1;
                 index0=r;  index1=s;  gray_to_dec();
                 ch1x2=index0  ; ch1y2=index1 ; ch1y2<<=4;

                 r=gx ;s=gy;
                 r>>=1;   r<<=1;  s|=2;
                 index0=r;  index1=s;  gray_to_dec();
                 ch2x2=index0  ; ch2y2=index1 ; ch2y2<<=4;

                 r=gx ;s=gy;
                 r>>=1;   r<<=1;  r|=2;
                 index0=r;  index1=s;  gray_to_dec();
                 ch3x2=index0  ; ch3y2=index1 ; ch3y2<<=4;
```

```
                 r=gx ;s=gy;
                 r>>=1;   r<<=1;  r|=2;
                 s>>=1;   s<<=1;  s|=2;
                 index0=r;  index1=s;  gray_to_dec();
                 ch4x2=index0  ; ch4y2=index1 ; ch4y2<<=4;
                 index0=p;   index1=p;   /* restore oringinal values */
                }

        where(level==3)
                {p=index0;  q=index1; /* save index0 & 1 values */
                 r=gx ;s=gy;
                 r>>=2;   r<<=2;  r|=1;
                 index0=r;  index1=s;  gray_to_dec();
                 ch1x3=index0  ; ch1y3=index1 ; ch1y3<<=4;

                 r=gx ;s=gy;
                 r>>=3;   r<<=3;  r|=5;
                 index0=r;  index1=s;  gray_to_dec();
                 ch2x3=index0  ; ch2y3=index1 ; ch2y3<<=4,

                 r=gx ;s=gy;
                 r>>=2;   r<<=2; r|=1;  s|=4;
                 index0=r;  index1=s;  gray_to_dec();
                 ch3x3=index0  ; ch3y3=index1 ; ch3y3<<=4;

                 r=gx ;s=gy;
                 r>>=3;   r<<=3;  r|=5;  s|=4;
                 index0=r;  index1=s;  gray_to_dec();
                 ch4x3=index0  ; ch4y3=index1 ; ch4y3<<=4;
                 index0=p;   index1=p;   /* restore oringinal values */
                }

        where(level==4)
                {p=index0;  q=index1; /* save index0 & 1 values */
                 r=gx ;s=gy;
                 s>>=1;   s<<=1;
                 index0=r;  index1=s;  gray_to_dec();
                 ch1x4=index0  ; ch1y4=index1 ; ch1y4<<=4;

                 r=gx ;s=gy;
                 s>>=1;   s<<=1;  r|=8;
                 index0=r;  index1=s;  gray_to_dec();
                 ch2x4=index0  ; ch2y4=index1 ; ch2y4<<=4;

                 r=gx ;s=gy;
                 s>>=1;   s<<=1;  s|=8;
                 index0=r;  index1=s;  gray_to_dec();
                 ch3x4=index0  ; ch3y4=index1 ; ch3y4<<=4;

                 r=gx ;s=gy;
                 s>>=1;   s<<=1;  s|=8;  r|=8;
                 index0=r;  index1=s;  gray_to_dec();
                 ch4x4=index0  ; ch4y4=index1 ;  ch4y4<<=4;
                 index0=p;   index1=p;   /* restore oringinal values */
                }


        where(level==5)
        {
                ch1x5=x+1;    ch1y5=y;
                ch2x5=29;     ch2y5=y;
                ch3x5=x+1;    ch3y5=464;
                ch4x5=29;     ch4y5=464;
        }
/* Children addresses calculation ends here. */
```

```
        /* Now calculate parameters for lateral communication. */

        where(level>=0) /* for base */
        {         northx=x; northy=(y-1);
                   southx=x; southy=(y+1);
                   eastx=(x+1); easty=y;
                   westx=(x-1); westy=y;
        where(northy==-1) northy=0;   where(westx==-1)  westx=0;
        where(eastx==32)  eastx=31;  where(southy==32)  southy=31;
        southy<<=4;        northy<<=4; easty<<=4; westy<<=4;
        }

/*
         for(row=0;row<64;row+=16){ printf("\n row = %d \n",row);
          for(col=0;col<32;col++){
  printf("\n[%3d][%3d]northx  = %d , northy  = %d  ", row, col,[row][col]northx
 ,[row][col]northy  );
  printf("\n[%3d][%3d]southx  = %d , southy  = %d  ", row, col,[row][col]southx
,[row][col]southy  );
  printf("\n[%3d][%3d]eastx  = %d , easty  = %d  ", row, col,[row][col]eastx  ,[
row][col]easty  );
  printf("\n[%3d][%3d]westx = %d , westy = %d  \n", row, col,[row][col]westx  ,[
row][col]westy  );
                                        }  }
*/

        printf("\n Your PYRAMID COMPUTER base is 32 x 32 \n");
wind:   printf("\nFor k x k size window, Enter value of k\n");
        scanf("%d",&win_size);
                if(win_size <=1 || win_size>8)  {  printf("\n wrong value: Valid
 sizes are 2,3,4,5,6,7,8  \n"); goto wind;  }
                printf("\n You've to enter total %d elemets; starting from 0 to
%d ", win_size*win_size,((win_size*win_size)-1));
                printf("\nEnter window elements by rows: ");
        for(k=0;k<(win_size*win_size); k++){
                printf("\n Enter element #%d  :",k);
                scanf("%d",&n  );
                win_val[k]=n   ;
                                                }
        for(part_size=1; part_size<win_size ; part_size*=2);
                m=32/part_size;
        printf("\n You entered Window size %d x %d ",win_size,win_size);
        printf("\n partition  size = %d x %d ", part_size, part_size );
        printf("\n PC base is divided into  %d partitions of size %d x %d ", m,p
art_size, part_size   );
        if(part_size>win_size) win_odd=1;
        if(m==16){ where(level==1) coord_pe=1;}
                else if(m==8){ where(level==2) coord_pe=2;}
                        else if(m==4){ where(level==3) coord_pe=3;}

/* coord_pe shows the level number of coordinator PEs. */
                printf("\n Coordinator  PEs are at level  %d ", coord_pe );

/* Now initialize each partition with its own x y coords */
        where(x<part_size) px=x;
        where(y<part_size)  py=y;
        where(x>=part_size ) px=x%part_size;
        where(y>=part_size ) py=y%part_size;

/*********************************************************
        where((x%3)==0) test=0;
*/
printf("\n\n STARTING CONVOLUTION PROCESS....\n");
/* assign corresponding value of window elements to parallel variable a */
                a=(win_size * py)+px;
                con_val=win_val[a];  /* con_val gets the right value of its own
                                        index for first convolution process */
        if(win_odd==1){where(px>=win_size) con_val=0;
                        where(py>=win_size)con_val=0 ; }
 /*   repeat: printf("\n\n con process row#=%d  col=%d ",e,f),
*/
CMC_timer_start(1);
repeat: con_result=con_val*test;
        adjust_for_com();
CMC_timer_start(2);
        send();
CMC_timer_stop(2);
CMC_timer_start(3);
        adjust_window();
CMC_timer_stop(3);

        if(count!=10) goto repeat;
CMC_timer_stop(1);

        [0][0]fcon_result=[0][3]fcon_result;
        [0][1]fcon_result=[0][4]fcon_result;
        time_val=CMC_timer_read_cm_busy(1);
        val=CMC_timer_read_cm_busy(2);
        val1=CMC_timer_read_cm_busy(3);
/*
        for(row=0;row<16;row+=16){
                for(col=0;col<32;col++){
printf("\n [%d][%d]test = %d     fcon_result= %d ",row,col,[row][col]test, [row]
[col]fcon_result);
} }  */

        printf("\n*****************************************************");
        printf("\n\t Total time CM busy for Convolution  is %f ",time_val);
        printf("\n\t Total time to send values to coordinator\n          PEs an
d get the result back is  %f ", val);
        printf("\n\t Total time CM takes to move the window is  %f ", val1),
        printf("\n\n");
}

adjust_for_com()
{       /* send con_val to top left corner of each partition */
        if(e==0 && f==0) return; /*e=0 and f=0 means start of convolution */
        else {
        a=e-1;
        if(e>0){  [westy][westx-a]con_result=con_result,
/*
printf("\n [%d][%d]con_result = %d ",f,e, [0][0]con_result ),
printf("\n [%d][%d]con_result = %d ",f,e, [0][1]con_result );
printf("\n [%d][%d]con_result = %d ",f,e, [0][2]con_result ),
printf("\n [%d][%d]con_result = %d ",f,e, [0][3]con_result );
*/                }
        m=f-1;
        a=f-1; if(m>0)  a<<=4;
        if(f>0){  [northy-a][northx]con_result=con_result;
/*
printf("\n [%d][%d]con_result = %d ",f,e, [0][0]con_result ),
printf("\n [%d][%d]con_result = %d ",f,e, [0][1]con_result );
printf("\n [%d][%d]con_result = %d ",f,e, [0][2]con_result );
printf("\n [%d][%d]con_result = %d ",f,e, [0][3]con_result ),
*/                }
        if(win_odd==1){where(px>=win_size) con_result=0;
                        where(py>=win_size)con_result=0 , }
        }
/*      for(row=0,row<32;row+=16){
                for(col=0;col<32;col++){
        printf("\n[%d][%d]con_val =%d   con_result =%d ",row,col,[row][col]con_v
```

```
al, [row][col]con_result  );
                    }         }
*/
}


adjust_window()
            /* When the final value reaches within each partition , then the
               following routine puts the correct value within correct PE. Then
               it shifts the window (to calculate next convolution) */
{
        where(py==f  && px==e )fcon_result=con_result;
/*
 f<<=4;  printf("For testing [%d][%d]con_result = %d \n",f,e,[f][e]con_result );
 f>>=4;
*/
        e++;
        con_result=0 ;  /* erase all convolved values now.*/
        dest1=0; dest2=0;dest3=0;dest4=0;dest5=0;
/* Now shift the window for correct position */
if(e<(part_size)) [easty][eastx]con_val=con_val;
        else {
                a=x-(part_size-1);
        where(a==-1) a=0; /* shift window to original position. */
         [westy][a]con_val=con_val; f++; e=0;
/*
printf("\n after to orig. position ");
printf("\n con_val = %d ", [0][0]con_val);
printf("\n con_val = %d ", [0][1]con_val);
printf("\n con_val = %d ", [0][2]con_val);
printf("\n con_val = %d ", [0][3]con_val);
*/
if(f>0 && f<part_size) [southy][southx]con_val=con_val;
                }
  if(f==part_size) {printf("\nNow f is equal to partition size so  returning f=
%d ",f); count=10; return; }   /* return is remove from here inside bracket */
/*
        for(row=0;row<48;row+=16){ printf("\n\nwindow values after shifting :");
                for(col=0;col<32;col++){
        printf("\n[%d][%d]con_val =%d   con_result =%d ",row,col,[row][col]con_v
al, [row][col]con_result  );
                    }         }
*/
}



send() /* This routine sends all values from base to the coordinators and then
          send them back to  the right partition. */
{       l=coord_pe;/*   printf("\n  l in start of send routine is %d ",l);  */


        where(level>=0) {/* printf("\n con results before sending are  %d %d %d
%d ", [0][0]con_result, [0][1]con_result, [16][0]con_result, [16][1]con_result )
; */}

        where(level>=0){ [x0][y0]dest1+=con_result; } /*  printf("\n dest1 = %d.
......... ", [0][0]dest1);*/
        l-=1; if(l>0) { where (level==1) [x1][y1]dest2+=dest1 ;/* printf("\n des
t 2=%d ", [0][1]dest2);*/ }
        l-=1; if(l>0) { where (level==2) [x1][y1]dest3+=dest2 ; /* printf("\n de
st 3=%d ", [0][2]dest3);*/ }
        l-=1; if(l>0) { where (level==3) [x1][y1]dest4+=dest3 ; /* printf("\n de
st 4=%d ", [16][2]dest4);*/ }
        l-=1; if(l>0) { where (level==4) [x1][y1]dest5+=dest4 ; /* printf("\n de
st 5=%d , l = %d ", [16][1]dest5,l) ; */ }
```

```
switch(coord_pe){

        case 5: where(level==5){
                [ch1y5][ch1x5]dest4=dest5;
                [ch2y5][ch2x5]dest4=dest5;
                [ch3y5][ch3x5]dest4=dest5;
                [ch4y5][ch4x5]dest4=dest5;   }

        case 4: where(level==4){
                [ch1y4][ch1x4]dest3=dest4;
                [ch2y4][ch2x4]dest3=dest4;
                [ch3y4][ch3x4]dest3=dest4;
                [ch4y4][ch4x4]dest3=dest4;   }

        case 3: where(level==3){
                [ch1y3][ch1x3]dest2=dest3;
                [ch2y3][ch2x3]dest2=dest3;
                [ch3y3][ch3x3]dest2=dest3;
                [ch4y3][ch4x3]dest2=dest3;   }

        case 2: where(level==2){
                [ch1y2][ch1x2]dest1=dest2;
                [ch2y2][ch2x2]dest1=dest2;
                [ch3y2][ch3x2]dest1=dest2;
                [ch4y2][ch4x2]dest1=dest2;   }

        case 1: where(level==1){
                [ch1y1][ch1x1]con_result=dest1;
                [ch2y1][ch2x1]con_result=dest1;
                [ch3y1][ch3x1]con_result=dest1;
                [ch4y1][ch4x1]con_result=dest1;   break; }

        default: printf("\n Sorry! Can not perform top down communication\n");
                }
}


dec_to_gray()
{
        gx=x>>1; gx^=x ; gy=y>>1; gy^=y  ;
}


gray_to_dec()
{
        int k=0;
           where(index0>1) { b=1;
             for(k=0;k<9;k++){
               where(b<=index0) b<<=1;
                              }
                         }
          where(b!=0) { b>>=1; d=b; b>>=1; c=b;     }
        for(k=0;k<7;k++){
          where(c!=0){ b^=index0; b&=c; d|=b; c>>=1; b>>=1;
                     }
                }
        where(index0>1) index0=d;
            b=0;c=0;d=0;
where(index1>1) { b=1;
             for(k=0;k<9;k++){
               where(b<=index1) b<<=1;
                   }
                         }
          where(b!=0) { b>>=1; d=b; b>>=1; c=b;     }
        for(k=0;k<7;k++){
```

```
          where(c!=0){ b^=index1 ; b&=c; d|=b; c>>=1; b>>=1;
                     }
                        }
        where(index1>1) index1=d;
        b=0; c=0;d=0;
}
```

```
/***********************************************************************
*                                                                      *
*   ****************   Lai & White Algorithm  I  ******************** *
*                        one PE / Router                               *
*                      C O N V O L U T I O N                           *
***********************************************************************/


#include        <stdio.h>
#include        <cstimer.h>

      shape[256][32] base;

/*  integer x and y contain  decimal values of x and y coordinates.  */
/*  While gx and gy contain Reflexive Gray codes for each x and y    */
/*  coordinates.                                                     */
/*  index0 holds address for row of master PE and index1 holds       */
/*  address for col of master PE for base only...                    */
/*  x1 contains row address for master PE of all other levels        */
/*  except for base; similarly y1 contains col address for master    */
/*  PE of all other levels except for base.                          */


        int:base bx,by,px,py, x,y,xx,yy,gx,gy, level=9, p=0, q=0 ,r=0;
        int:base test=1,index0,index1,dest1,dest2,dest3,dest4,dest5,con_val=0,co
n_result=0, fcon_result=0, mul=0;
        static int e=0,f=0;
        static int count=0;
        int row,  col, k=0, clk_tck=0, l,m,n, coord_pe=0 ;
        int:base a=0,b=0,c=0,d=0, addr=0, mark=0  ;
        int:base x0,y0,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7;
        int:base ch1x1,ch1y1,ch2x1,ch2y1,ch3x1,ch3y1,ch4x1,ch4y1;
        int:base ch1x2,ch1y2,ch2x2,ch2y2,ch3x2,ch3y2,ch4x2,ch4y2;
        int:base ch1x3,ch1y3,ch2x3,ch2y3,ch3x3,ch3y3,ch4x3,ch4y3;
        int:base ch1x4,ch1y4,ch2x4,ch2y4,ch3x4,ch3y4,ch4x4,ch4y4;
        int:base ch1x5,ch1y5,ch2x5,ch2y5,ch3x5,ch3y5,ch4x5,ch4y5;
        int:base southx,southy,northx,northy,eastx,easty,westx,westy;
        int win_size=0, win_odd=0, part_size=0, no_of_part=0;
        int:base win_val[64];   /* max. window value is 8 x 8 */
        int window_val[64]; /* scalor array */


/* south , north, east, west variables are used for lateral communication. */
/* ch1x1 means children 1, x-axis value, for level 1 */
/* x0 to x7 and y0 to y7 variables stores address of master PE for  */
/* each level of Pyramid.                                           */
        double time_val, val, vall;



main()
{
        with(base){
                xx=pcoord(1);
                yy=pcoord(0);
        for(col=0;col<32;col++){
            for(row=0; row<256; row++){
                if(row==0 && col==0) k=0;
                  else k+=1;
                [row][col]addr=k; }       }

        a=0x0F;     b=a&addr;
        where(b==0) mark=1;    /*  b=0, means those PEs whose 4 LSBs =0 */

        k=0;
for(row=0; row<256; row+=16)      {  if (row==0) k=0; else k+=1;
     for(col=0; col<32; col++) {
          [row][col]x=col;  [row][col]y=k;
                              }      }

     where(mark==1)
          {
          dec_to_gray();

/*  Set level # for each PE */

          level=10; /* initially level=10 for all unused base PE */
          a=gx&2; b=gx&3;
          where(a==2 || b==3) level=0;
          a=gx&7;
          where(a==0) level=1;
          a=gy&3;  b=gx&7;
          where(a==0 && b==1) level=2;
          a=gy&7;  b=gx&15;
          where(a==1 && b==1) level=3;
          a=gx&31;  b=gy&15;
          [96][1]level=4;


          /* index0 holds address for row of master PE and index1 holds */
          /* address for col of master PE                               */
               index0=gx; index1=gy;
          where(level==0)  {  index0>>=3; index0<<=3;  }
          where(level==1)  {index1>>=2;index1<<=2;index0|=1; }
          where(level==2)  {index0>>=4;index0<<=4;index0|=1;
                            index1>>=3;index1<<=3;index1|=1; }
          where(level==3){ index0=1; index1=80;  }/* gray 80= decimal 96 */

          gray_to_dec();  /* convert all final values of index0 & 1 to dec. */
                 index1<<=4;


/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
          Now calculate the addresses of children for all PE and for
          all levels.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

          where(level==0)
          {       a=7;     b=a&gx;
          where(b==3)  { [index1][index0]ch1x1=x;  [index1][index0]ch1y1=y; }
          where(b==2)  { [index1][index0]ch2x1=x;  [index1][index0]ch2y1=y; }
          where(b==6)  { [index1][index0]ch3x1=x;  [index1][index0]ch3y1=y; }
          where(b==7)  { [index1][index0]ch4x1=x;  [index1][index0]ch4y1=y; }
            }


          where(level==1)
          {       a=3;     b=a&gy;
          where(b==0)  { [index1][index0]ch1x2=x;  [index1][index0]ch1y2=y; }
          where(b==1)  { [index1][index0]ch2x2=x;  [index1][index0]ch2y2=y; }
          where(b==3)  { [index1][index0]ch3x2=x;  [index1][index0]ch3y2=y; }
          where(b==2)  { [index1][index0]ch4x2=x;  [index1][index0]ch4y2=y, }
          ch1y1<<=4; ch2y1<<=4; ch3y1<<=4; ch4y1<<=4 ;  }

          where(level==2)
          {       a=7;     b=a&gy; a=15;  c=a&gx;
          where(b==0 && c==1)  {[index1][index0]ch1x3=x,  [index1][index0]ch1y3=y; }
          where(b==0 && c==9)  {[index1][index0]ch2x3=x;  [index1][index0]ch2y3=y; }
          where(b==4 && c==1)  {[index1][index0]ch3x3=x;  [index1][index0]ch3y3=y; }
          where(b==4 && c==9)  {[index1][index0]ch4x3=x;  [index1][index0]ch4y3=y; }
          ch1y2<<=4; ch2y2<<=4; ch3y2<<=4; ch4y2<<=4 ;  }
```

```
        where(level==3)
        {
        ch1y3<<=4; ch2y3<<=4; ch3y3<<=4; ch4y3<<=4 ;  }

        where(level==4)
        {
        [6][1]ch1x4=1;  [6][1]ch1y4=1;
        [6][1]ch2x4=1;  [6][1]ch2y4=30;
        [6][1]ch3x4=14; [6][1]ch3y4=1;
        [6][1]ch4x4=14; [6][1]ch4y4=30;
        ch1y4<<=4; ch2y4<<=4; ch3y4<<=4; ch4y4<<=4 ;  }

/***** children calculation ends here ******/

/*&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
Now calculate paramters for lateral communication.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*/
        where(level==0) /* for base */
        {         northx=x; northy=(y-1);
                  southx=x; southy=(y+1);
                  eastx=(x+1); easty=y;
                  westx=(x-1); westy=y;
        where(northy==-1) northy=0;   where(westx==-1)  westx=0;
        where(eastx==32)  eastx=31;  where(southy==16)  southy=15;
        where(x==5 || x==13 || x==21 ) eastx+=4;
        where(x==10 || x==18 || x==26) westx-=4;
        southy<<=4;       northy<<=4; easty<<=4; westy<<=4;
        }


/*(((((((((((((((((((((((((((((((((((((((((((((((((((((((((
Initialize base of PC with logical x,y coords. namely bx and by.
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))*/

        k=0; n=-1;
for(row=0; row<256; row+=16)      {  if (row==0) k=0; else k+=1;
   for(col=0; col<32; col++) {
        m=[row][col]level;
        if(m==0){ n++;
        [row][col]bx=n%16;  [row][col]by=k;
                }            }       }
/****** logical coords. end here *******/

        printf("\n Your PYRAMID COMPUTER base is 16 x 16 \n");
wind:   printf("\nFor k x k size window, Enter value of k\n");
        scanf("%d",&win_size);
                if(win_size <=1 || win_size>8)  {
  printf("\n wrong value : Valid window sizes are 2,3,4,5,6,7,8 \n"); goto wind;
  }
                printf("\n You've to enter total %d elemets; starting from 0 to
%d ", win_size*win_size,((win_size*win_size)-1));
                printf("\nEnter window elements by rows: ");
        for(k=0;k<(win_size*win_size); k++){
                printf("\n Enter element #%d  :",k);
                scanf("%d",&n  );
                win_val[k]=n   ;
                                                }
        for(part_size=1; part_size<win_size ; part_size*=2);
                m=16/part_size;
        printf("\n You entered Window size %d x %d ",win_size,win_size);
        printf("\n partition  size = %d x %d ", part_size, part_size );
        printf("\n PC base is divided into  %d partitions of size %d x %d ", m,p
art_size, part_size   );
        if(part_size>win_size) win_odd=1;
        if(m==8){ where(level==1) coord_pe=1;}
                else if(m==4){ where(level==2) coord_pe=2,}
                        else if(m==2){ where(level==3) coord_pe=3,}

/* coord_pe shows the level number of coordinator PEs. */
                printf("\n Coordinator  PEs are at level  %d ", coord_pe );

/* Now initialize each partition with its own x, y coords. namely px & py  */
        where(bx<part_size) px=bx;
        where(by<part_size)  py=by;
        where(bx>=part_size ) px=bx%part_size;
        where(by>=part_size ) py=by%part_size;

/*******************************************************/
        where((x%3)==0) test=0;

printf("\n\n STARTING CONVOLUTION PROCESS....\n");

/* assign corresponding value of window elements to parallel variable a */
        if(part_size==2){ a=bx%4; con_val=win_val[a];}
                else{
                a=(win_size * py)+px;
                con_val=win_val[a]; } /* con_val gets the right value of its own
                                index for first convolution process */
        if(win_odd==1)(where(px>=win_size) con_val=0;
                        where(py>=win_size)con_val=0 ; }

/*      repeat: printf("\n\n con process row#=%d  col=%d ",e,f),  */
CMC_timer_start(1);
repeat: con_result=con_val*test;
        adjust_for_com();
CMC_timer_start(2);
        send();
CMC_timer_stop(2);
CMC_timer_start(3);
        adjust_window();
CMC_timer_stop(3);
        if(count!=10) goto repeat;
CMC_timer_stop(1);

/*

         [by][bx]fcon_result=win_val[1]+win_val[3] ;
         [by][bx]fcon_result=win_val[0]+win_val[3]+win_val[1]+win_val[2] ;
         [by][bx]fcon_result=win_val[0]+win_val[2] ;
*/
        time_val=CMC_timer_read_cm_busy(1);
        val=CMC_timer_read_cm_busy(2);
        val1=CMC_timer_read_cm_busy(3);
        printf("\n************************************************");
        printf("\n\t Total time CM busy for Convolution  is %f ",time_val);
        printf("\n\t Total time to send values to coordinator\n          PEs an
d get the result back is  %f ", val);
        printf("\n\t Total time CM takes to move the window is  %f ", val1);
        printf("\n\n");
        }

        }

}

dec_to_gray()
{
        gx=x>>1; gx^=x ; gy=y>>1; gy^=y  ;
}

gray_to_dec()
{   int k=0;
```

```
      where(index0>1)  { b=1;  for(k=0;k<9;k++){
            where(b<=index0) b<<=1; }    }
        where(b!=0)  { b>>=1; d=b; b>>=1; c=b;       }
        for(k=0;k<7;k++){
          where(c!=0){ b^=index0; b&=c; d|=b; c>>=1; b>>=1;}    }
        where(index0>1)index0=d;
            b=0;c=0;d=0;

        where(index1>1)  { b=1;  for(k=0;k<9;k++){
              where(b<=index1)  b<<=1; }   }
          where(b!=0)  { b>>=1; d=b; b>>=1; c=b;       }
        for(k=0;k<7;k++){
          where(c!=0){ b^=index1 ; b&=c; d|=b; c>>=1; b>>=1;   }  }
        where(index1>1)index1=d;
        b=0;  c=0;d=0;
}


send() /* This routine sends all values from base to the coordinators and then
          send them back to  the right partition. */
{       l=coord_pe;
        where(level==0) [index1][index0]dest1+=con_result; /*  printf("\n dest 1
= %d ", [0][0]dest1); */
        l-=1; if(l>0) { where (level==1) [index1][index0]dest2+=dest1 ;/* printf
("\n dest 2=%d ", [0][1]dest2); */ }
        l-=1; if(l>0) { where (level==2) [index1][index0]dest3+=dest2 ;/*  print
f("\n dest 3=%d ", [16][1]dest3);*/  }
        l-=1; if(l>0) { where (level==3) [index1][index0]dest4+=dest3 ;/*  print
f("\n dest 4=%d ", [96][1]dest4); */  }

switch(coord_pe){
        case 5: where(level==5){
                [ch1y5][ch1x5]dest4=dest5;
                [ch2y5][ch2x5]dest4=dest5;
                [ch3y5][ch3x5]dest4=dest5;
                [ch4y5][ch4x5]dest4=dest5;   }
        case 4: where(level==4){
                [ch1y4][ch1x4]dest3=dest4;
                [ch2y4][ch2x4]dest3=dest4;
                [ch3y4][ch3x4]dest3=dest4;
                [ch4y4][ch4x4]dest3=dest4;   }
        case 3: where(level==3){
                [ch1y3][ch1x3]dest2=dest3;
                [ch2y3][ch2x3]dest2=dest3;
                [ch3y3][ch3x3]dest2=dest3;
                [ch4y3][ch4x3]dest2=dest3;   }
        case 2: where(level==2){
                [ch1y2][ch1x2]dest1=dest2;
                [ch2y2][ch2x2]dest1=dest2;
                [ch3y2][ch3x2]dest1=dest2;
                [ch4y2][ch4x2]dest1=dest2;   }
        case 1: where(level==1){
                [ch1y1][ch1x1]con_result=dest1;
                [ch2y1][ch2x1]con_result=dest1;
                [ch3y1][ch3x1]con_result=dest1;
                [ch4y1][ch4x1]con_result=dest1;  break; }
        default: printf("\n Sorry! Can not perform top down communication\n");
                }
}


adjust_for_com()
{       /* send con_val to top left corner of each partition */
        if(e==0 && f==0) return; /*e=0 and f=0 means start of convolution */
        else {
        a=e-1;
        if(e>0){  [westy][westx-a]con_result=con_result;

                }

        m=f-1;
        a=f-1; if(m>0)   a<<=4;
        if(f>0){   [northy-a][northx]con_result=con_result;

                }
        if(win_odd==1){where(px>=win_size) con_result=0;
                        where(py>=win_size)con_result=0 ; }

        }
}

adjust_window()
            /* When the final value reaches within each partition , then the
               following routine puts the correct value within correct PE. Then
               it shifts the window (to calculate next convolution) */
{
        where(py==f  && px==e )fcon_result=con_result;
        e++;
        con_result=0 ;   /* erase all convolved values now.*/
        dest1=0; dest2=0;dest3=0;dest4=0;dest5=0;

/* Now shift the window for correct position */

if(e<(part_size)) [easty][eastx]con_val=con_val;
        else {
                a=bx-(part_size-1);
        where(a==-1) a=0; /* shift window to original position  */
         [westy][a]con_val=con_val; f++; e=0;

if(f>0 && f<part_size) [southy][southx]con_val=con_val;

            }

  if(f==part_size) {printf("\n returning f= %d ",f); count=10; return;}
}
```

```
/**********************************************************************
*                                                                     *
*   ****************   Lai & White Algorithm  II ********************  *
*                          one PE / Router                            *
*                          C O N V O L U T I O N                      *
***********************************************************************/


#include        <stdio.h>
#include        <cstimer.h>

      shape[8192] base;

/*  Integer x and y contain  decimal values of x and y coordinates.  */
/*  While gx and gy contain Reflexive Gray codes for each x and y    */
/*  coordinates.                                                     */
/*  index0 holds address for row of master PE and index1 holds       */
/*  address for col of master PE.                                    */

        int:base bx,by,px,py, x,y,xx,yy,gx,gy, level=9, p=0, q=0 ,r=0 ,s=0;
        int:base test=1,index0,index1,dest1,dest2,dest3,dest4,dest5,con_val=0,co
n_result=0, fcon_result=0, mul=0;
        static int e=0,f=0;
        static int count=0;
        int row,  col, k=0, clk_tck=0, l,m,n, coord_pe=0 ;
        int:base x0,y0,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7;
        int:base south,north,east,west, m_add;
        int win_size=0, win_odd=0, part_size=0, no_of_part=0;
        int:base win_val[64];   /* max. window value is 8 x 8 */
        int window_val[64]; /* scalor array */

        int:base a=0,b=0,c=0,d=0,mark=0,addr=0;
        int:base tempx1,tempx2,tempx3,tempx4,tempy1,tempy2,tempy3,tempy4;

/* These variables store temporary coordinate variables for communiacation
by each master PE.  */
        int or1,or2,or3,or4,xor,sh;
/* south , north, east, west variables are used for lateral communication. */
        double time_val, val, val1;


main()
{
        with(base){
                xx=pcoord(0),
        a=0x0F;     b=a&xx;

        where(b==0)  mark = 1; /* b=0 means  those PEs whose 4 LSB =0 */
        k=0;
        for(row=0, row<8192; row+=16 ){
                if(row==0) k=0;  else k+=1;
                   [row]x=k;
                                                }
where(mark==1){
                dec_to_gray();

/*  Set level # for each PE
    This routine starts from apex and goes top down. It calculates
    children for each master PE and then update children's level
    number and sends the address of master PE to its children for
    communicatioon purpose. Address of master Pe is saved  in memory
    variable m_add                                                   */

        level=9;  /* set level to an arbitrary value initially. */

[0]level=4;  [32]level=3; [96]level=3; [112]level=3; [64]level=3;
```

```
        or1=8;or2=16;or3=24;xor=4;
for(k=3;k>0;k--){
where(level==k)
 {  a=gx;p=or2|a;q=or3|a;a^=xor;r=a|or1;s=a|or2;
index0=p; gray_to_dec();index0<<=4;  tempx1=index0;
index0=q; gray_to_dec();index0<<=4;  tempx2=index0;
index0=r; gray_to_dec();index0<<=4;  tempx3=index0;
index0=s; gray_to_dec();index0<<=4;  tempx4=index0;
        [tempx1]level=(k-1); [tempx1]m_add=(x<<4);
        [tempx2]level=(k-1); [tempx2]m_add=(x<<4);
        [tempx3]level=(k-1); [tempx3]m_add=(x<<4);
        [tempx4]level=(k-1); [tempx4]m_add=(x<<4);
 }
/*printf("\n or1=%d , 2=%d, 3=%d, xor=%d",or1,or2,or3,xor);  */
        or1<<=2;or2<<=2;or3<<=2;xor<<=2;
                    }
/***** Above code has calculated both the parent PE addresses and 4
        Children addresses. Since this routine goes top down  So, now
        every body within the shape has both parent and children addresses
        and hence no calculation for children is required like other
        algorithms; which goes bottom up and required extra code for
        children calculation.
        tempx1,tempx2,tempx3,tempx4 contain children addresses
        m_add contains parrent addresses  ***/

/*** Now calculate parameters for lateral communication for base  **/

data_0();
data_1();
data_2();
data_3();


/**** Lateral communication parametrs end here ******/

        printf("\n Your PYRAMID COMPUTER base is 16 x 16 \n"),
wind:   printf("\nFor k x k size window, Enter value of k\n");
        scanf("%d",&win_size);
                if(win_size <=1 || win_size>8)  {
  printf("\n wrong value : Valid window sizes are 2,3,4,5,6,7,8 \n"), goto wind;
  }
                printf("\n You've to enter total %d elemets; starting from 0 to
%d ", win_size*win_size,((win_size*win_size)-1));
                printf("\nEnter window elements by rows. ");
        for(k=0;k<(win_size*win_size); k++){
                printf("\n Enter element #%d  :",k);
                scanf("%d",&n  );
                win_val[k]=n   ;
                                                }
        for(part_size=1; part_size<win_size ; part_size*=2);
                m=16/part_size;
        printf("\n You entered Window size %d x %d ",win_size,win_size);
        printf("\n partition  size = %d x %d ", part_size, part_size );
        printf("\n PC base is divided into  %d partitions of size %d x %d ", m,p
art_size, part_size   );
        if(part_size>win_size) win_odd=1;
        if(m==8){ where(level==1) coord_pe=1;}
                else if(m==4){ where(level==2) coord_pe=2,}
                        else if(m==2){ where(level==3) coord_pe=3;}
/* coord_pe shows the level number of coordinator PEs. */
                printf("\n Coordinator  PEs are at level  %d ", coord_pe );
/* Now initialize each partition with its own x, y coords. namely px & py  */
        where(bx<part_size) px=bx;
        where(by<part_size)  py=by;
        where(bx>=part_size ) px=bx%part_size;
```

```
        where(by>=part_size ) py=by%part_size;

/*************************************************************
        where((x%3)==0) test=0;
/*/
printf("\n\n STARTING CONVOLUTION PROCESS....\n");
/* assign corresponding value of window elements to parallel variable a */
        if(part_size==2){ a=bx%4; con_val=win_val[a];}
                else{
                a=(win_size * py)+px;
                con_val=win_val[a]; } /* con_val gets the right value of its own
                                        index for first convolution process */
        if(win_odd==1){where(px>=win_size) con_val=0;
                        where(py>=win_size)con_val=0 ; }
/*      repeat: printf("\n\n con process row#=%d  col=%d ",e,f);   */
CMC_timer_start(1);
repeat: con_result=con_val*test;
        adjust_for_com();
CMC_timer_start(2);
        send();
CMC_timer_stop(2);
CMC_timer_start(3);
        adjust_window();
CMC_timer_stop(3);
        if(count!=10) goto repeat;
CMC_timer_stop(1);
/*
if(part_size==2){ where(level==0){
         [by][bx]fcon_result=win_val[1]+win_val[3] ;
         [by][bx]fcon_result=win_val[0]+win_val[3]+win_val[1]+win_val[2] ;
         [by][bx]fcon_result=win_val[0]+win_val[2] ;
                        }               }
*/
        time_val=CMC_timer_read_cm_busy(1);
        val=CMC_timer_read_cm_busy(2);
        val1=CMC_timer_read_cm_busy(3);

        printf("\n********************************************************");
        printf("\n\t Total time CM busy for Convolution  is %f ",time_val);
        printf("\n\t Total time to send values to coordinator\n          PEs an
d get the result back is  %f ", val);
        printf("\n\t Total time CM takes to move the window is  %f ", val1);
        printf("\n\n");
        }
        }
}
send() /* This routine sends all values from base to the coordinators and then
          send them back to  the right partition. */
{       l=coord_pe;
        where(level==0) [m_add]dest1+=con_result; /*  printf("\n dest 1= %d ", [
0][0]dest1); */
        l-=1; if(l>0) { where (level==1) [m_add]dest2+=dest1 ;/* printf("\n dest
 2=%d ", [0][1]dest2); */ }
        l-=1; if(l>0) { where (level==2) [m_add]dest3+=dest2 ;/*  printf("\n des
t 3=%d ", [16][1]dest3);*/  }
        l-=1; if(l>0) { where (level==3) [m_add]dest4+=dest3 ;/*  printf("\n des
t 4=%d ", [96][1]dest4); */  }
switch(coord_pe){
        case 4: where(level==4){
                [tempx1]dest3=dest4;
                [tempx2]dest3=dest4;
                [tempx3]dest3=dest4;
                [tempx4]dest3=dest4;   }

        case 3: where(level==3){
                [tempx1]dest2=dest3;
```

```
                [tempx2]dest2=dest3;
                [tempx3]dest2=dest3;
                [tempx4]dest2=dest3;   }

        case 2: where(level==2){
                [tempx1]dest1=dest2;
                [tempx2]dest1=dest2;
                [tempx3]dest1=dest2;
                [tempx4]dest1=dest2;   }

        case 1: where(level==1){
                [tempx1]con_result=dest1;
                [tempx2]con_result=dest1;
                [tempx3]con_result=dest1;
                [tempx4]con_result=dest1;  break; }

        default: printf("\n Sorry! Can not perform top down communication\n");
                }
}


adjust_for_com()
{       /* send con_val to top left corner of each partition */
        if(e==0 && f==0) return; /*e=0 and f=0 means start of convolution */
        else {
        a=e-1;
        if(e>0){  [west]con_result=con_result;
                }
        m=f-1;
        a=f-1; if(m>0)  a<<=4;
        if(f>0){   [north]con_result=con_result; }

        if(win_odd==1){where(px>=win_size)  con_result=0;
                        where(py>=win_size)con_result=0 ; }
                }

}

adjust_window()
           /* When the final value reaches within each partition , then the
              following routine puts the correct value within correct PE. Then
              it shifts the window (to calculate next convolution) */
{
        where(py==f  && px==e )fcon_result=con_result;
        e++;
        con_result=0 ;  /* erase all convolved values now.*/
        dest1=0; dest2=0;dest3=0;dest4=0;dest5=0;
/* Now shift the window for correct position */
if(e<(part_size)) [east]con_val=con_val;
        else {
                a=bx-(part_size-1);
        where(a==-1) a=0; /* shift window to original position  */
         [west]con_val=con_val; f++; e=0;


if(f>0 && f<part_size) [south]con_val=con_val;
                }
  if(f==part_size) {printf("\n returning f= %d ",f); count=10, return;}

}


dec_to_gray()
{
        gx=x>>1; gx^=x ; gy=y>>1; gy^=y  ;
}
```

```
gray_to_dec()
{    int k=0;
            where(index0>1)  { b=1;  for(k=0;k<14;k++){
                 where(b<=index0)  b<<=1;  }    }
          where(b!=0)  { b>>=1;  d=b;  b>>=1;  c=b;       }
        for(k=0;k<14;k++){
          where(c!=0){ b^=index0;  b&=c;  d|=b;  c>>=1;  b>>=1;}    }
        where(index0>1)index0=d;
            b=0;c=0;d=0;

}
```

```
data_0()
{
        int:current north,south,east,west;
/**** row 1 of base ****/
[2880]north=2880;  [2880]south=4928;  [2880]east=6976;  [2880]west=2880;
[6976]north=6976;  [6976]south=7344;  [6976]east=3904;  [6976]west=2880;
[3904]north=3904;  [3904]south=5952;  [3904]east=8000;  [3904]west=6976;
[8000]north=8000;  [8000]south=6320;  [8000]east=2992;  [8000]west=3904;
[2992]north=2992;  [2992]south=5040;  [2992]east=7088;  [2992]west=8000;
[7088]north=7088;  [7088]south=7232;  [7088]east=4016;  [7088]west=2992;
[4016]north=4016;  [4016]south=6064;  [4016]east=8112;  [4016]west=7088;
[8112]north=8112;  [8112]south=6208;  [8112]east=2944;  [8112]west=4016;
[2944]north=2944;  [2944]south=4992;  [2944]east=7040;  [2944]west=8112;
[7040]north=7040;  [7040]south=7280;  [7040]east=3968;  [7040]west=2944;
[3968]north=3968;  [3968]south=6016;  [3968]east=8064;  [3968]west=7040;
[8064]north=8064;  [8064]south=6256;  [8064]east=2928;  [8064]west=3968;
[2928]north=2928;  [2928]south=4976;  [2928]east=7024;  [2928]west=8064;
[7024]north=7024;  [7024]south=7440;  [7024]east=3952;  [7024]west=2928;
[3952]north=3952;  [3952]south=6000;  [3952]east=8048;  [3952]west=7024;
[8048]north=8048;  [8048]south=6272;  [8048]east=8048;  [8048]west=3952;

/**** row 2 of base ****/
[4928]north=2880;  [4928]south=3392;  [4928]east=7344;  [4928]west=4928;
[7344]north=6976;  [7344]south=7488;  [7344]east=5952;  [7344]west=4928;
[5952]north=3904;  [5952]south=3760;  [5952]east=6320;  [5952]west=7344;
[6320]north=8000;  [6320]south=7856;  [6320]east=5040;  [6320]west=5952;
[5040]north=2992;  [5040]south=3504;  [5040]east=7232;  [5040]west=6320;
[7232]north=7088;  [7232]south=7600;  [7232]east=6064;  [7232]west=5040;
[6064]north=4016;  [6064]south=3648;  [6064]east=6208;  [6064]west=7232;
[6208]north=8112;  [6208]south=6848;  [6208]east=4992;  [6208]west=6064;
[4992]north=2944;  [4992]south=3456;  [4992]east=7280;  [4992]west=6208;
[7280]north=7040;  [7280]south=7552;  [7280]east=6016;  [7280]west=4992;
[6016]north=3968;  [6016]south=3696;  [6016]east=6256;  [6016]west=7280;
[6256]north=8064;  [6256]south=7792;  [6256]east=4976;  [6256]west=6016;
[4976]north=2928;  [4976]south=3440;  [4976]east=7440;  [4976]west=6256;
[7440]north=7024;  [7440]south=7536;  [7440]east=6000;  [7440]west=4976;
[6000]north=3952;  [6000]south=3712;  [6000]east=6272;  [6000]west=7440;
[6272]north=8048;  [6272]south=7808;  [6272]east=6272;  [6272]west=6000;

/**** row 3 of base ****/
[3392]north=4928;  [3392]south=5440;  [3392]east=7488;  [3392]west=3392;
[7488]north=7344;  [7488]south=6832;  [7488]east=3760;  [7488]west=3392;
[3760]north=5952;  [3760]south=5808;  [3760]east=7856;  [3760]west=7488;
[7856]north=6320;  [7856]south=6464;  [7856]east=3504;  [7856]west=3760;
[3504]north=5040;  [3504]south=5552;  [3504]east=7600;  [3504]west=7856;
[7600]north=7232;  [7600]south=6720;  [7600]east=3648;  [7600]west=3504;
[3648]north=6064;  [3648]south=5696;  [3648]east=6848;  [3648]west=7600;
[6848]north=6208;  [6848]south=6576;  [6848]east=3456;  [6848]west=3648;
[3456]north=4992;  [3456]south=5504;  [3456]east=7552;  [3456]west=6848;
[7552]north=7280;  [7552]south=6768;  [7552]east=3696;  [7552]west=3456;
[3696]north=6016;  [3696]south=5744;  [3696]east=7792;  [3696]west=7552;
[7792]north=6256;  [7792]south=6528;  [7792]east=3440;  [7792]west=3696;
[3440]north=4976;  [3440]south=5488;  [3440]east=7536;  [3440]west=7792;
[7536]north=7440;  [7536]south=6784;  [7536]east=3712;  [7536]west=3440;
[3712]north=6000;  [3712]south=5760;  [3712]east=7808;  [3712]west=7536;
[7808]north=6272;  [7808]south=6512;  [7808]east=7808;  [7808]west=3712;

/**** row 4 of base ****/
[5440]north=3392;  [5440]south=2752;  [5440]east=6832;  [5440]west=5440;
[6832]north=7488;  [6832]south=6848;  [6832]east=5808;  [6832]west=5440;
[5808]north=3760;  [5808]south=3776;  [5808]east=6464;  [5808]west=6832;
[6464]north=7856;  [6464]south=7872;  [6464]east=5552;  [6464]west=5808;
[5552]north=3504;  [5552]south=3008;  [5552]east=6720;  [5552]west=6464;
[6720]north=7600;  [6720]south=7104;  [6720]east=5696;  [6720]west=5552;
[5696]north=3648;  [5696]south=4032;  [5696]east=6576;  [5696]west=6720;
[6576]north=6848;  [6576]south=8128;  [6576]east=5504;  [6576]west=5696;
[5504]north=3456;  [5504]south=3056;  [5504]east=6768;  [5504]west=6576;
[6768]north=7552;  [6768]south=7152;  [6768]east=5744;  [6768]west=5504;
[5744]north=3696;  [5744]south=4080;  [5744]east=6528;  [5744]west=6768;
[6528]north=7792;  [6528]south=8176;  [6528]east=5488;  [6528]west=5744;
[5488]north=3440;  [5488]south=2800;  [5488]east=6784;  [5488]west=6528;
[6784]north=7536;  [6784]south=6896;  [6784]east=5760;  [6784]west=5488;
[5760]north=3712;  [5760]south=3824;  [5760]east=6512;  [5760]west=6784;
[6512]north=7808;  [6512]south=7920;  [6512]east=6512;  [6512]west=5760;

}
```

```
data_1()
{

        int.current north,south, east, west;
/**** row 5 of base ****/
[2752]north=5440;   [2752]south=4800;   [2752]east=6848;   [2752]west=2752;
[6848]north=6832;   [6848]south=7472;   [6848]east=3776;   [6848]west=2752;
[3776]north=5808;   [3776]south=5824;   [3776]east=7872;   [3776]west=6848;
[7872]north=6464;   [7872]south=6448;   [7872]east=3008;   [7872]west=3776;
[3008]north=5552;   [3008]south=5056;   [3008]east=7104;   [3008]west=7872;
[7104]north=6720;   [7104]south=7216;   [7104]east=4032;   [7104]west=3008;
[4032]north=5696;   [4032]south=6080;   [4032]east=8128;   [4032]west=7104;
[8128]north=6576;   [8128]south=5424;   [8128]east=3056;   [8128]west=4032;
[3056]north=5504;   [3056]south=5104;   [3056]east=7152;   [3056]west=8128;
[7152]north=6768;   [7152]south=7168;   [7152]east=4080;   [7152]west=3056;
[4080]north=5744;   [4080]south=6128;   [4080]east=8176;   [4080]west=7152;
[8176]north=6528;   [8176]south=6144;   [8176]east=2800;   [8176]west=4080;
[2800]north=5488;   [2800]south=4848;   [2800]east=6896;   [2800]west=8176;
[6896]north=6784;   [6896]south=7424;   [6896]east=3824;   [6896]west=2800;
[3824]north=5760;   [3824]south=5872;   [3824]east=7920;   [3824]west=6896;
[7920]north=6512;   [7920]south=6400;   [7920]east=7920;   [7920]west=3824;

/**** row 6 of base ****/
[4800]north=2752;   [4800]south=3264;   [4800]east=7472;   [4800]west=4800;
[7472]north=6848;   [7472]south=7360;   [7472]east=5824;   [7472]west=4800;
[5824]north=3776;   [5824]south=3888;   [5824]east=6448;   [5824]west=7472;
[6448]north=7872;   [6448]south=7984;   [6448]east=5056;   [6448]west=5824;
[5056]north=3008;   [5056]south=3520;   [5056]east=7216;   [5056]west=6448;
[7216]north=7104;   [7216]south=7616;   [7216]east=6080;   [7216]west=5056;
[6080]north=4032;   [6080]south=3632;   [6080]east=5424;   [6080]west=7216;
[5424]north=8128;   [5424]south=7728;   [5424]east=5104;   [5424]west=6080;
[5104]north=3056;   [5104]south=3568;   [5104]east=7168;   [5104]west=5424;
[7168]north=7152;   [7168]south=7664;   [7168]east=6128;   [7168]west=5104;
[6128]north=4080;   [6128]south=3584;   [6128]east=6144;   [6128]west=7168;
[6144]north=8176;   [6144]south=7680;   [6144]east=4848;   [6144]west=6128;
[4848]north=2800;   [4848]south=3312;   [4848]east=7424;   [4848]west=6144;
[7424]north=6896;   [7424]south=7408;   [7424]east=5872;   [7424]west=4848;
[5872]north=3824;   [5872]south=3840;   [5872]east=6400;   [5872]west=7424;
[6400]north=7920;   [6400]south=7936;   [6400]east=6400;   [6400]west=5872;

/**** row 7 of base ****/
[3264]north=4800;   [3264]south=5312;   [3264]east=7360;   [3264]west=3264;
[7360]north=7472;   [7360]south=6960;   [7360]east=3888;   [7360]west=3264;
[3888]north=5824;   [3888]south=5936;   [3888]east=7984;   [3888]west=7360;
[7984]north=6448;   [7984]south=6336;   [7984]east=3520;   [7984]west=3888;
[3520]north=5056;   [3520]south=5568;   [3520]east=7616;   [3520]west=7984;
[7616]north=7216;   [7616]south=6704;   [7616]east=3632;   [7616]west=3520;
[3632]north=6080;   [3632]south=5680;   [3632]east=7728;   [3632]west=7616;
[7728]north=5424;   [7728]south=6592;   [7728]east=3568;   [7728]west=3632;
[3568]north=5104;   [3568]south=5616;   [3568]east=7664;   [3568]west=7728;
[7664]north=7168;   [7664]south=6656;   [7664]east=3584;   [7664]west=3568;
[3584]north=6128;   [3584]south=5632;   [3584]east=7680;   [3584]west=7664;
[7680]north=6144;   [7680]south=6640;   [7680]east=3312;   [7680]west=3584;
[3312]north=4848;   [3312]south=5360;   [3312]east=7408;   [3312]west=7680;
[7408]north=7424;   [7408]south=6912;   [7408]east=3840;   [7408]west=3312;
[3840]north=5872;   [3840]south=5888;   [3840]east=7936;   [3840]west=7408;
[7936]north=6400;   [7936]south=6384;   [7936]east=7936;   [7936]west=3840;

/**** row 8 of base ****/
[5312]north=3264;   [5312]south=2720;   [5312]east=6960;   [5312]west=5312;
[6960]north=7360;   [6960]south=6816;   [6960]east=5936;   [6960]west=5312;
[5936]north=3888;   [5936]south=3744;   [5936]east=6336;   [5936]west=6960;
[6336]north=7984;   [6336]south=7840;   [6336]east=5568;   [6336]west=5936;
[5568]north=3520;   [5568]south=2976;   [5568]east=6704;   [5568]west=6336;
[6704]north=7616;   [6704]south=7072;   [6704]east=5680;   [6704]west=5568;
[5680]north=3632;   [5680]south=4000;   [5680]east=6592;   [5680]west=6704;
[6592]north=7728;   [6592]south=8096;   [6592]east=5616;   [6592]west=5680;
[5616]north=3568;   [5616]south=3040;   [5616]east=6656;   [5616]west=6592,
[6656]north=7664;   [6656]south=7136;   [6656]east=5632;   [6656]west=5616;
[5632]north=3584;   [5632]south=4064;   [5632]east=6640;   [5632]west=6656;
[6640]north=7680;   [6640]south=8160;   [6640]east=5360;   [6640]west=5632;
[5360]north=3312;   [5360]south=2784;   [5360]east=6912;   [5360]west=6640;
[6912]north=7408;   [6912]south=6880;   [6912]east=5888;   [6912]west=5360;
[5888]north=3840;   [5888]south=3808;   [5888]east=6384;   [5888]west=6912;
[6384]north=7936;   [6384]south=7904;   [6384]east=6384;   [6384]west=5888;


}
```

```
data_2()
{
        int:current north, south, east, west;
/**** row 9 of base ****/
[2720]north=5312;  [2720]south=4768;  [2720]east=6816;  [2720]west=2720;
[6816]north=6960;  [6816]south=7504;  [6816]east=3744;  [6816]west=2720;
[3744]north=5936;  [3744]south=5792;  [3744]east=7840;  [3744]west=6816;
[7840]north=6336;  [7840]south=6480;  [7840]east=2976;  [7840]west=3744;
[2976]north=5568;  [2976]south=5024;  [2976]east=7072;  [2976]west=7840;
[7072]north=6704;  [7072]south=7248;  [7072]east=4000;  [7072]west=2976;
[4000]north=5680;  [4000]south=6048;  [4000]east=8096;  [4000]west=7072;
[8096]north=6592;  [8096]south=6224;  [8096]east=3040;  [8096]west=4000;
[3040]north=5616;  [3040]south=5088;  [3040]east=7136;  [3040]west=8096;
[7136]north=6656;  [7136]south=7184;  [7136]east=4064;  [7136]west=3040;
[4064]north=5632;  [4064]south=6112;  [4064]east=8160;  [4064]west=7136;
[8160]north=6640;  [8160]south=6160;  [8160]east=2784;  [8160]west=4064;
[2784]north=5360;  [2784]south=4832;  [2784]east=6880;  [2784]west=8160;
[6880]north=6912;  [6880]south=7440;  [6880]east=3808;  [6880]west=2784;
[3808]north=5888;  [3808]south=5856;  [3808]east=7904;  [3808]west=6880;
[7904]north=6384;  [7904]south=6416;  [7904]east=7904;  [7904]west=3808;

/**** row 10 of base ****/
[4768]north=2720;  [4768]south=3232;  [4768]east=7504;  [4768]west=4768;
[7504]north=6816;  [7504]south=7328;  [7504]east=5792;  [7504]west=4768;
[5792]north=3744;  [5792]south=3920;  [5792]east=6480;  [5792]west=7504;
[6480]north=7840;  [6480]south=8016;  [6480]east=5024;  [6480]west=5792;
[5024]north=2976;  [5024]south=3488;  [5024]east=7248;  [5024]west=6480;
[7248]north=7072;  [7248]south=7584;  [7248]east=6048;  [7248]west=5024;
[6048]north=4000;  [6048]south=3664;  [6048]east=6224;  [6048]west=7248;
[6224]north=8096;  [6224]south=7760;  [6224]east=5088;  [6224]west=6048;
[5088]north=3040;  [5088]south=3552;  [5088]east=7184;  [5088]west=6224;
[7184]north=7136;  [7184]south=7648;  [7184]east=6112;  [7184]west=5088;
[6112]north=4064;  [6112]south=3600;  [6112]east=6160;  [6112]west=7184;
[6160]north=8160;  [6160]south=7696;  [6160]east=4832;  [6160]west=6112;
[4832]north=2784;  [4832]south=3296;  [4832]east=7440;  [4832]west=6160;
[7440]north=6880;  [7440]south=7392;  [7440]east=5856;  [7440]west=4832;
[5856]north=3808;  [5856]south=3856;  [5856]east=6416;  [5856]west=7440;
[6416]north=7904;  [6416]south=7952;  [6416]east=6416;  [6416]west=5856;

/**** row 11 of base ****/
[3232]north=4768;  [3232]south=5280;  [3232]east=7328;  [3232]west=3232;
[7328]north=7504;  [7328]south=6992;  [7328]east=3920;  [7328]west=3232;
[3920]north=5792;  [3920]south=5968;  [3920]east=8016;  [3920]west=7328;
[8016]north=6480;  [8016]south=6304;  [8016]east=3488;  [8016]west=3920;
[3488]north=5024;  [3488]south=5536;  [3488]east=7584;  [3488]west=8016;
[7584]north=7248;  [7584]south=7328;  [7584]east=3664;  [7584]west=3488;
[3664]north=6048;  [3664]south=5712;  [3664]east=7760;  [3664]west=7584;
[7760]north=6224;  [7760]south=6560;  [7760]east=3552;  [7760]west=3664;
[3552]north=5088;  [3552]south=5600;  [3552]east=7648;  [3552]west=7760;
[7648]north=7184;  [7648]south=6672;  [7648]east=3600;  [7648]west=3552;
[3600]north=6112;  [3600]south=5648;  [3600]east=7696;  [3600]west=7648;
[7696]north=6160;  [7696]south=6624;  [7696]east=3296;  [7696]west=3600;
[3296]north=4832;  [3296]south=5344;  [3296]east=7392;  [3296]west=7696;
[7392]north=7440;  [7392]south=6928;  [7392]east=3856;  [7392]west=3296;
[3856]north=5856;  [3856]south=5904;  [3856]east=7952;  [3856]west=7392;
[7952]north=6416;  [7952]south=6368;  [7952]east=7952;  [7952]west=3856;

/**** row 12 of base ****/
[5280]north=3232;  [5280]south=2848;  [5280]east=6992;  [5280]west=5280;
[6992]north=7328;  [6992]south=6944;  [6992]east=5968;  [6992]west=5280;
[5968]north=3920;  [5968]south=3872;  [5968]east=6304;  [5968]west=6992;
[6304]north=8016;  [6304]south=7968;  [6304]east=5536;  [6304]west=5968;
[5536]north=3488;  [5536]south=3024;  [5536]east=7328;  [5536]west=6304;
[7328]north=7584;  [7328]south=7120;  [7328]east=5712;  [7328]west=5536;
[5712]north=3664;  [5712]south=4048;  [5712]east=6560;  [5712]west=7328;
[6560]north=7760;  [6560]south=8144;  [6560]east=5600;  [6560]west=5712;
[5600]north=3552;  [5600]south=2960;  [5600]east=6672;  [5600]west=6560;
[6672]north=7648;  [6672]south=7056;  [6672]east=5648;  [6672]west=5600;
[5648]north=3600;  [5648]south=3984;  [5648]east=6624;  [5648]west=6672;
[6624]north=7696;  [6624]south=8080;  [6624]east=5344;  [6624]west=5648;
[5344]north=3296;  [5344]south=2912;  [5344]east=6928;  [5344]west=6624;
[6928]north=7392;  [6928]south=7008;  [6928]east=5904;  [6928]west=5344;
[5904]north=3856;  [5904]south=3936;  [5904]east=6368;  [5904]west=6928;
[6368]north=7952;  [6368]south=8032;  [6368]east=6368;  [6368]west=5904;

}
```

```
data_3()
{

        int:current north, south, east, west;
/**** row 13 of base ****/
[2848]north=5280;  [2848]south=4896;  [2848]east=6944;  [2848]west=2848;
[6944]north=6992;  [6944]south=7376;  [6944]east=3872;  [6944]west=2848;
[3872]north=5968;  [3872]south=5920;  [3872]east=7968;  [3872]west=6944;
[7968]north=6304;  [7968]south=6352;  [7968]east=3024;  [7968]west=3872;
[3024]north=5536;  [3024]south=5072;  [3024]east=7120;  [3024]west=7968;
[7120]north=7328;  [7120]south=7200;  [7120]east=4048;  [7120]west=3024;
[4048]north=5712;  [4048]south=6096;  [4048]east=8144;  [4048]west=7120;
[8144]north=6560;  [8144]south=6176;  [8144]east=2960;  [8144]west=4048;
[2960]north=5600;  [2960]south=5008;  [2960]east=7056;  [2960]west=8144;
[7056]north=6672;  [7056]south=7264;  [7056]east=3984;  [7056]west=2960;
[3984]north=5648;  [3984]south=6032;  [3984]east=8080;  [3984]west=7056;
[8080]north=6624;  [8080]south=6240;  [8080]east=2912;  [8080]west=3984;
[2912]north=5344;  [2912]south=4690;  [2912]east=7008;  [2912]west=8080;
[7008]north=6928;  [7008]south=7312;  [7008]east=3936;  [7008]west=2912;
[3936]north=5904;  [3936]south=5984;  [3936]east=8032;  [3936]west=7008;
[8032]north=6368;  [8032]south=6288;  [8032]east=8032;  [8032]west=3936;

/**** row 14 of base ****/
[4896]north=2848;  [4896]south=3360;  [4896]east=7376;  [4896]west=4896;
[7376]north=6944;  [7376]south=7456;  [7376]east=5920;  [7376]west=4896;
[5920]north=3872;  [5920]south=3792;  [5920]east=6352;  [5920]west=7376;
[6352]north=7968;  [6352]south=7888;  [6352]east=5072;  [6352]west=5920;
[5072]north=3024;  [5072]south=3536;  [5072]east=7200;  [5072]west=6352;
[7200]north=7120;  [7200]south=7632;  [7200]east=6096;  [7200]west=5072;
[6096]north=4048;  [6096]south=3616;  [6096]east=6176;  [6096]west=7200;
[6176]north=8144;  [6176]south=7712;  [6176]east=5008;  [6176]west=6096;
[5008]north=2960;  [5008]south=3472;  [5008]east=7264;  [5008]west=6176;
[7264]north=7056;  [7264]south=7568;  [7264]east=6032;  [7264]west=5008;
[6032]north=3984;  [6032]south=3680;  [6032]east=6240;  [6032]west=7264;
[6240]north=8080;  [6240]south=7776;  [6240]east=4690;  [6240]west=6032;
[4690]north=2912;  [4690]south=3424;  [4690]east=7312;  [4690]west=6240;
[7312]north=7008;  [7312]south=7520;  [7312]east=5984;  [7312]west=4690;
[5984]north=3936;  [5984]south=3728;  [5984]east=6288;  [5984]west=7312;
[6288]north=8032;  [6288]south=7824;  [6288]east=6288;  [6288]west=5984;

/**** row 15 of base ****/
[3360]north=4896;  [3360]south=5408;  [3360]east=7456;  [3360]west=3360;
[7456]north=7376;  [7456]south=6864;  [7456]east=3792;  [7456]west=3360;
[3792]north=5920;  [3792]south=5840;  [3792]east=7888;  [3792]west=7456;
[7888]north=6352;  [7888]south=6432;  [7888]east=3536;  [7888]west=3792;
[3536]north=5072;  [3536]south=5584;  [3536]east=7632;  [3536]west=7888;
[7632]north=7200;  [7632]south=6688;  [7632]east=3616;  [7632]west=3536;
[3616]north=6096;  [3616]south=5664;  [3616]east=7712;  [3616]west=7632;
[7712]north=6176;  [7712]south=6608;  [7712]east=3472;  [7712]west=3616;
[3472]north=5008;  [3472]south=5520;  [3472]east=7568;  [3472]west=7712;
[7568]north=7264;  [7568]south=6752;  [7568]east=3680;  [7568]west=3472;
[3680]north=6032;  [3680]south=5728;  [3680]east=7776;  [3680]west=7568;
[7776]north=6240;  [7776]south=6544;  [7776]east=3424;  [7776]west=3680;
[3424]north=4690;  [3424]south=5472;  [3424]east=7520;  [3424]west=7776;
[7520]north=7312;  [7520]south=6800;  [7520]east=3728;  [7520]west=3424;
[3728]north=5984;  [3728]south=5776;  [3728]east=7824;  [3728]west=7520;
[7824]north=6288;  [7824]south=6496;  [7824]east=7824;  [7824]west=3728;

/**** row 16 of base ****/
[5408]north=3360;  [5408]south=5408;  [5408]east=6864;  [5408]west=5408;
[6864]north=7456;  [6864]south=6864;  [6864]east=5840;  [6864]west=5408;
[5840]north=3792;  [5840]south=5840;  [5840]east=6432;  [5840]west=6864;
[6432]north=7888;  [6432]south=6432;  [6432]east=5584;  [6432]west=5840;
[5584]north=3536;  [5584]south=5584;  [5584]east=6688;  [5584]west=6432;
[6688]north=7632;  [6688]south=6688;  [6688]east=5664;  [6688]west=5584;
[5664]north=3616;  [5664]south=5664;  [5664]east=6608;  [5664]west=6688,
[6608]north=7712;  [6608]south=6608;  [6608]east=5520;  [6608]west=5664;
[5520]north=3472;  [5520]south=5520;  [5520]east=6752;  [5520]west=6608;
[6752]north=7568;  [6752]south=6752;  [6752]east=5728;  [6752]west=5520;
[5728]north=3680;  [5728]south=5728;  [5728]east=6544;  [5728]west=6752;
[6544]north=7776;  [6544]south=6544;  [6544]east=5472;  [6544]west=5728;
[5472]north=3424;  [5472]south=5472;  [5472]east=6800;  [5472]west=6544;
[6800]north=7520;  [6800]south=6800;  [6800]east=5776;  [6800]west=5472;
[5776]north=3728;  [5776]south=5776;  [5776]east=6496;  [5776]west=6800;
[6496]north=7824;  [6496]south=6496;  [6496]east=6496;  [6496]west=5776;

}
```

BIBLIOGRAPHY

[1] Seitz, C.L., "The Cosmic Cube." *Communications of the ACM*, Vol. 28, No. 1 (Jan. 1985), pp. 22-33.

[2] Stout, Q.F., "Hypercubes and Pyramids." *in Pyramidal Systems for Computer Vision*, Cantoni and S. Levialdi (Eds.), Spring-Verlag, Berlin, Heidelberg, Germany (1986), pp. 74-89.

[3] Lai, T.-H. and White, W.,"Mapping Pyramid Algorithms into Hypercubes," *Journal of Parallel and Distributed Computing*, Vol. 9 (1990), pp. 42-54.

[4] Wu, A.Y.,"Embedding of Tree Networks into Hypercubes," *Journal of Parallel and Distributed Computing*, Vol. 2, No. 3 (Aug. 1985), pp. 238-249.

[5] Johnsson, S.L., "Dilation d Embedding of a Hyper-pyramid into a Hypercube," *Communications of the ACM* (1989), pp. 294-303.

[6] Ho, C.-T. and Johnsson, S.L., "On the Embedding of Arbitrary Meshes in Boolean Cubes with Expansion Two Dilation Two," *Proceedings of International Conference on Parallel Processing*, Vol. I, Chicago, IL (Aug. 1987), pp. 188-191.

[7] Bhatt, S.N. and Ipsen, I.F., "How to Embed Trees in Hypercubes," *Tech. Rep. YALEU/CSD/RR-443*, Department of Computer Science., Yale University, New Haven, CT (Dec. 1985).

[8] Ho, C.-T. and Johnsson, S.L., "Spanning Balanced Trees in Boolean Cubes," *SIAM Journal on Scientific and Statistical Computing.*, Vol. 10 , No. 4 (July 1989), pp. 607-630.

[9] Ziavras, S.G., "Techniques for Mapping Deterministic Algorithms onto Multi-Level Systems." *Proceedings of International Conference on Parallel Processing*, Vol. I, Chicago, IL (Aug. 1990), pp. 226-233.

[10] Ziavras, S.G., "On the Mapping Problem for Multi-Level Systems," *Proceedings of Supercomputing '89 Conference.*, IEEE Computer Society and ACM SIGARCH, Reno, Nevada (Nov. 13-17), pp. 399-408.

[11] Chan, T.F. and Saad, Y., "Multigrid Algorithms on the Hypercube Multiprocessor," *IEEE Transactions on Computers.*, Vol. C-35, No. 11 (Aug. 1988), pp. 969-977.

[12] Hillis, W.D., *The Connection Machine*, MIT Press, Cambridge, MA (1985).

[13] Clermont, P. and Merigot, A., "Real Time Synchronization in a Multi-SIMD Massively Parallel Machine," *Proceedings of Architectures for Pattern Analysis and Machine Intelligence Conference.* (1987), pp. 131-136.

[14] Patel, S.C. and Ziavras, S.G., "Comparative Analysis of Techniques that Map Hierarchical Structures into Hypercubes," *Proceedings of Parallel and Distributed Computing and Systems Conference.*, Washington, D.C. (Oct. 1991), pp. 295-299.

[15] Cantoni, V. and Levialdi, S., "Multiprocessor Computing for Images," *IEEE Special Issue on Computer Vision*, Vol. 76, No. 8 (Aug. 1988), pp. 959-969.

[16] *Connection Machine Model CM-2 Technical Summary*, Version 6.0, Thinking Machines Corporation, Cambridge, MA, Nov. 1990.

[17] Chang, J.H., Ibarra, O.H., Pong, T.-C., and Sohn, S.M., "Two-Dimensional Convolution on a Pyramid Computer," *IEEE Transactions on Pattern Analysis and Machine Intelligence.*, Vol. 10, No. 4 (July 1988), pp. 590-593.

[18] Enkelmann, W., "Investigations of Multigrid Algorithms for the Estimation of Optical Flow Fields in Image Sequences," *Computer Vision, Graphics, and Image Processing*, Vol. 43, Aug. 1988, pp. 150-177.

[19] Brandt, A., "Multilevel Computations: Review and Recent Developments," in *Multigrid Methods - Theory, Applications, and Supercomputing*, S.F. McCormick

(Ed.), Lecture Notes in Pure and Applied Mathematics, Marcel Dekker Inc., New York, NY, 1988, pp. 35-62.

[20] Ziavras, S.G., "On the Problem of Expanding Hypercube-Based Systems," *Journal of Parallel and Distributed Computing*, May 1992.

[21] Burt, P.J. and Hong, T.H., "Segmentation and Estimation of Image region properties through cooperative hierarchical computation," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC 11, no.12, Dec. '81.

[22] Hwang, K. and Briggs, F.A., "*Computer Architecture and Parallel Processing*," McGraw Hill publication, 1984

[23] Choudhary, A.N. and Patel, H.J., "*Parallel Architecture and Parallel Algorithms for Integrated Vision System*," Kluwer Academic Publishers, 1990

[24] Uhr, L.,"*Parallel Computer Vision*", Academic Press Inc. 1987