ABSTRACT

Performance Analysis of Pyramid Mapping Algorithms for the Hypercube

by
Jing-Chiou Liou

Comparative performance analysis of algorithms that map pyramids and multilevel structures onto the hypercube are presented. The pyramid structure is appropriate for low-level and intermediate-level computer vision algorithms. It is not only efficient for the support of both local and global operations but also capable of supporting the implementation of multilevel solvers. Nevertheless, pyramids lack the capability of efficient implementation of the majority of scientific algorithms and their cost may become unacceptably high. On a different horizon, hypercube machines have widely been used in the field of parallel computing due to their small diameter, high degree of fault tolerance, and rich interconnection that permits fast communication at a reasonable cost. As a result, hypercube machines can efficiently emulate pyramids. Therefore, the characteristics which make hypercube machines useful scientific processors also make them efficient image processors.

Two algorithms which have been developed for the efficient mapping of the pyramid onto the hypercube are discussed in this thesis. The algorithm proposed by Stout [4] requires a hypercube with a number of processing elements (PEs) which is equal to the number of nodes in the base of the pyramid. This algorithm can activate only one level of the pyramid at a time. In contrast, the algorithm proposed by Patel and Ziavras [7] requires the same number of PEs as Stout's algorithm but allows the concurrent

simulation of multiple levels, as long as the base level is not involved in the set of pyramid levels that need to be simulated at the same time. This low-cost algorithm yields higher performance through high utilization of PEs. However it performs slightly worse than Stout's algorithm when only one level is active at a time. Patel and Ziavras' algorithm performs much better than Stout's algorithm when all levels, excluding the leaf level, are active concurrently. The comparative analysis of these two algorithms is based on the incorporation of simulation results for some image processing algorithms which are perimeter counting, image convolution, and segmentation.

PERFORMANCE ANALYSIS OF PYRAMID MAPPING ALGORITHMS FOR THE HYPERCUBE

by
Jing-Chiou Liou

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

Department of Electrical and Computer Engineering

January, 1993

APPROVAL PAGE

Performance Analysis of
Pyramid Mapping Algorithms for the Hypercube

Jing-Chiou Liou

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

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

Dr. Dennis Karvelas, Committee Member
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Jing-Chiou Liou

Degree: Master of Science in Electrical and Computer Engineering

Date: January, 1993

Undergraduate and Graduate Education:

- Master of Science in Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ, 1993
- Bachelor of Science in Electrical Engineering, National Taiwan Institute of Technology, Taipei, Taiwan, R.O.C., 1983

Major: Electrical and Computer Engineering

This thesis is dedicated to
my dear wife Susan

ACKNOWLEDGMENT

The author wishes to express his sincere gratitude to his supervisor, Professor Sotirios G. Ziavras, for his guidance, friendship, and moral support throughout this research.

Special thanks to Professors John D. Carpinelli and Dennis Karvelas for serving as members of the committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 The Hypercube Network

The hypercube network has widely been used in the field of parallel computing because it offers a small diameter, high degree of fault tolerance, and rich interconnection structure that permits fast communication at a reasonable cost [1,8]. A d-dimensional hypercube H_d is composed of 2^d nodes with d edges per node (i.e., each node in such a hypercube has d neighbors) [1]. A unique d-bit address is assigned to each node of the hypercube. An edge connects two nodes if and only if the address of these two nodes differ by a single bit. An edge is a communication link between two neighboring nodes which makes the hypercube a distributed memory machine, where information is passed in the form of messages.

The hypercube topology has several important properties. First, it is homogeneous. This means that for any dimension d, given any two vertices p,q in H_d, there is a graph isomorphism of H_d onto itself which maps p onto q. To see this, let r = label(p) XOR label(q) (all logical operations are performed bitwise) [4]. The mapping which maps a vertex s to the vertex labeled r XOR label(s) is one such isomorphism. Homogeneity implies that all nodes can be treated equally, and in particular it means that in a computer implementation it is natural to allow input/output to all nodes. It also means that if an algorithm treats a node specially (for example, if node 0 is used as the root of a tree), then by using XOR the algorithm can be "translated" so that any other desired node is the special one. Some other

structures such as pyramids and meshes are not homogeneous, since the apex is unique and corners can only be mapped to other corners.

Routing messages between nodes is particularly simple in a hypercube. A message from one node p to another node q has to travel along at least as many edges as the number of bits by which the addresses of p and q differ (i.e., number of 1's in the result of the XOR operation between the binary addresses of p and q). A message from p is sent to a neighboring node r whose address differs in only the i^{th} bit from the address of p (i.e., where the i^{th} bit of the result of the XOR operation is 1) and so on until the message reaches q. This process produces a path of minimum length. Notice that there are many such paths of minimum length. The diameter of a topology is defined as the largest distance between pairs of nodes. The diameter of H_d is $d=\log_2$(number of nodes). For comparison, the diameter of the 2-dimensional mesh is the square root of the number of nodes, while in a pyramid it is $2x\log_2$(number of nodes in the base).

Each node in H_d has degree d, meaning that it has d edges. In a physical implementation the degree of some nodes must be d+1 to allow communication to the outside world, so if communication is homogeneously implemented then all nodes will have degree d+1. The hypercube is a modular structure. Hence, hypercubes are eminently partitionable into smaller hypercubes. For example, H_{d+1} can be partitioned into two disjoint hypercubes H_d. One copy consists of all nodes having 0 in a particular bit position of d+1 bit addresses and the other consists of all nodes having 1 in that coordinate. For example, as shown in Figure 1.1, a 3-dimensional cube H_3 consists of two distinct copies of H_2 with one copy having 0 in the most significant bit and the other copy having 1.

Thus, any number of hypercubes of smaller dimensionality d can be mapped simultaneously into the hypercube with a larger dimensionality D provided $2^D \geq 2^k$, where k = the sum of the dimensions of all such small hypercubes to be mapped into H_D. Hence, the hypercube provides an environment with a great deal of flexibility for dynamic allocation of cubes. Due to its highly regular and dense structure, the hypercube has also been proven to be a highly fault-tolerant network.

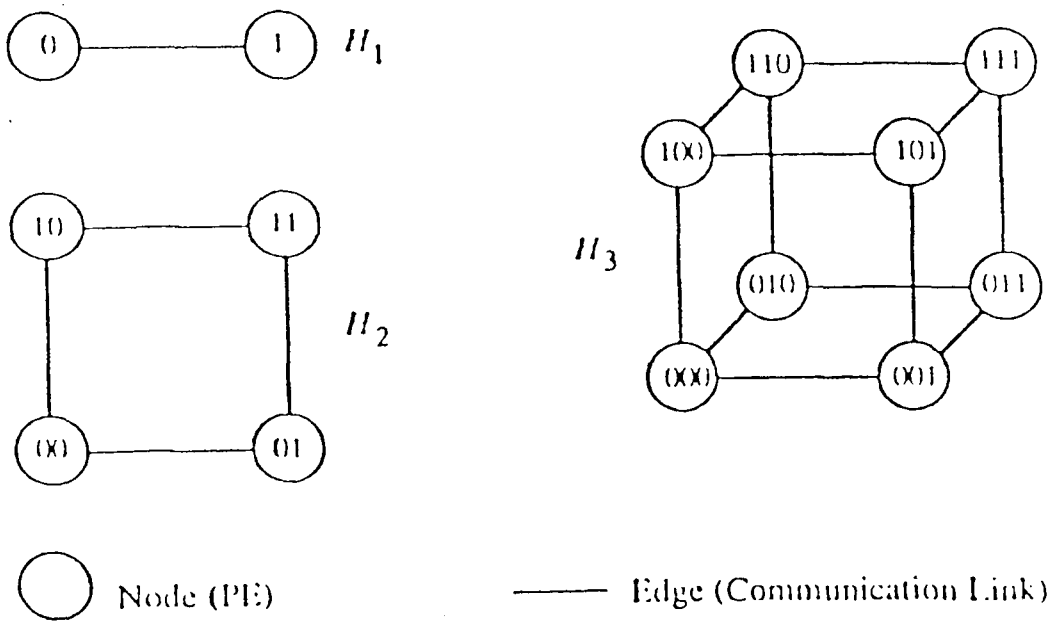

Figure 1.1 Small Hypercubes

1.2 Multilevel Systems

A multilevel system is a hierarchically-structured array of processors, which implements most of the variations of pyramidal systems. The basic structure of the multilevel system is pyramid-like [13]. Hence, these systems are composed of successive layers of mesh-connected arrays of PEs. Each PE is a

processor along with some local memory. The number of PEs in the arrays decreases with the increase of the level number, where the lowest level number corresponds to the leaf level. In addition, the size of the leaf level is $2^n x 2^n$ and the reductions between pairs of neighboring levels are $2^m x 2^m$, where m are natural numbers, and m may have different values for different pairs of neighboring levels. Only pairs of neighboring levels can communicate directly with each other. PEs are connected to each other by point-to-point bidirectional communication channels and the number of data transfer registers (DTRs) of any PE is equal to the number of its communication channels. The characteristics of these system are as follows: (1) They are composed of identical PEs. (2) They are not necessarily single-rooted systems. (3) There is a single controller per level (i.e., each system operates in the MSIMD mode of computation).

The pyramid is a special case of multilevel systems with a single apex and the reductions between successive arrays are 2x2. In the standard pyramid configuration, each processor at any level, except for the processors at the lowest level, is directly connected to four children located at the immediately lower level, and the size of each array is 1/4 the size of the array at the immediately lower level. In the rest of the discussion P_n denotes a standard pyramid with $2^n x 2^n$ nodes at its leaf level. Such a pyramid has n+1 levels. Figure 1.2 shows the P_2 pyramid with base size $4^2=16$.

In general, the nodes on level i, $0 \leq i \leq \log n$ (the base nodes are on level 0) are connected as an $n/2^i$ x $n/2^i$ mesh-connected network. As shown in Figure 1.2, on each level, we denote by (r,s) the node in position (r,s) on that level $0 \leq r,s \leq (n/2^i)-1$. Notice that a pyramid with base size n^2 has no more than $4n^2/3$ nodes.

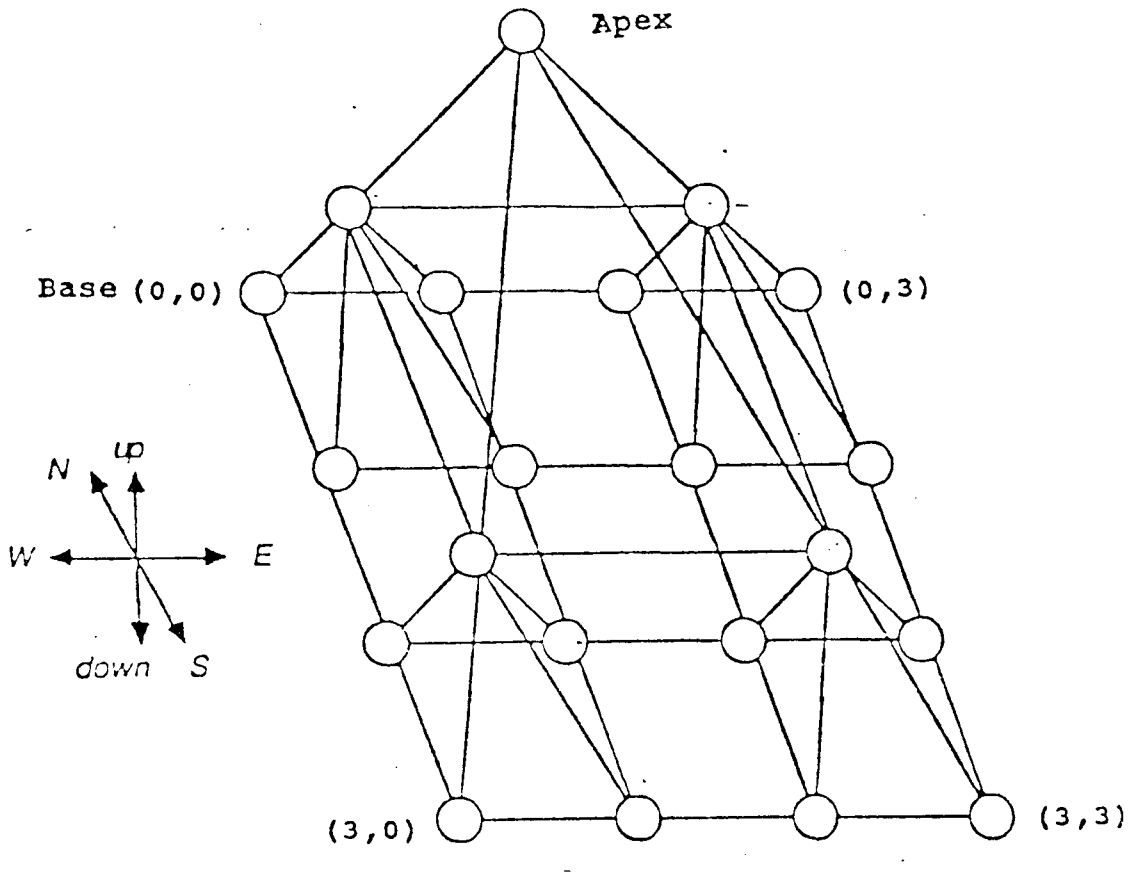

Figure 1.2 The P_2 pyramid with base size 4^2=16

Standard pyramids with very powerful PEs, having 10 or 11 levels and being used to process images of size 512x512 or 1024x1024, are impossible to efficiently build with the current technology. Therefore, alternative hardware solutions need to be investigated [9]. For example, the total number of levels could be reduced by increasing the reductions between neighboring levels. Sometimes, a speedup of computation is achieved by using pyramid-like systems that have small reductions at lower levels to enable the application of standard multiresolution techniques, while larger reductions at higher levels allow for the fast collection of information extracted at lower levels.

1.3 Applications of Multilevel Systems

Multilevel systems have been widely used in the low-level and intermediate-level phase of image processing and computer vision (IP & CV). The main goal of the low-level and intermediate-level phases of IP & CV is to locate objects present in images and then produce a description of them; this description is then used by the high level image understanding tasks to identify individual objects and their spatial relationships in the given scene. The low-level and intermediate-level phases of IP & CV are characterized by both local and global operations, when the two-dimentional array structure of an image is considered, with the majority of the operations being local. Multilevel systems support efficiently both local and global operations; they are also suitable for divide-and-conquer techniques [8]. As a consequence, various algorithms that utilize such systems have been proposed [9,2].

We use for perception images input into the retina-like base array of multilevel systems (typically, the pyramid). From that point, a number of different approaches can be taken.

- The system could first find edges, regions, and other features, using local array operations. These can then be successively averaged and / or grouped together by linking them moving up, through, and down the system.
- Intermediate-level and higher-level processors could be used in parallel algorithms to find contours, regions and intrinsic images, build up 3-dimensional images, and try to match sub-regions of this abstract image with models of objects stored in memory.
- A whole hierarchy of abstractions could be built, each level transforming the results of other levels.
- The system could find features and / or segment the image and directly process regions of interest.

• Features could be extracted from the image to generate abstract feature images, and also collected, compounded and converged into higher-level abstract images.

1.4 Motivations and Objectives

The hypercube network has achieved a marked popularity in the field of parallel computing. Some systems such as Intel iSPC, NCUBE, and Connection Machine are commercially available. In contrast, powerful pyramid machines are not cost-effective, difficult to build with current technology, and have limited applications. However, the hypercube is a general purpose topology which is capable of efficiently emulating a wide variety of networks, such as the mesh [14], the pyramid [4,5,6], and the hyper-pyramid [15]. Thus, the problem of simulating the pyramid on the hypercube is very important. Several algorithms like Stout's [4], Lai-White's [5,6], and Patel-Ziavras'[7] have been developed to embed pyramids into hypercubes.

Studying these algorithms reveals the fact that Lai-White's algorithms need a (2n+1)-dimensional hypercube H_{2n+1} to simulate a P_n pyramid with $2^n x 2^n$ PEs at its leaf level while Stout's and Patel-Ziavras' algorithms need an H_{2n} hypercube. This means that Stout's and Patel-Ziavras' algorithms require only half the number of PEs needed by Lai-White's algorithm to simulate the same pyramid. However, Lai-White's algorithms allow the concurrent simulation of all levels of the pyramid while Stout's algorithm allows only one level of pyramid to be active at a time. On the other hand, Patel-Ziavras' algorithm also allows the concurrent simulation of all levels excluding the leaf level of the pyramid.

With the need of the H_{2n} hypercube to simulate a P_n pyramid and the capability of simulating all levels simultaneously except for the leaf level of the pyramid, Patel-Ziavras' algorithm is a compromise between Stout's algorithm and Lai-White's algorithm. Therefore, although Lai-White's algorithms achieve higher performance than Stout's algorithm when multiple levels of the pyramid need to concurrently activate, they will not be considered useful algorithms due to their higher cost and lower utilization (Notice that some PEs are never used in Lai-White's algorithms).

Thus, the main objective of this research is to explore a comparative analysis based on analytical techniques involving Stout's algorithm and Patel-Ziavras' algorithm for the mapping of the pyramid onto hypercube. In addition, this thesis also shows the mapping of multiple pyramids and overlapped pyramids onto the hypercubes. Simulation results for some important image processing algorithms such as finding the perimeter of an object, 2-D convolution, and Segmentation are also included.

1.5 Thesis Outline

This thesis is organized as follows. Chapter 2 discusses existing algorithms that map pyramids onto hypercubes. The mapping of overlapping pyramid structures onto the hypercube is discussed in the last two sections of Chapter 2. Comparative analysis of these existing algorithms is also included. Various simulation results are presented, and the mapping algorithms are compared in Chapter 3. Conclusions are presented in Chapter 4.

CHAPTER 2
EXISTING MAPPING ALGORITHMS

2.1 Performance Measures

The analytical technique being used in this research incorporates three measures of the cost of graph mappings, namely expansion, dilation and congestion. The function h: G $\rightarrow$ G' represents the mapping of the source graph G onto the target graph G'. It is a mapping of the vertices on G into the vertices of G' in a one to one fashion. The three measures are then defined as follows [6]:

Expansion: The expansion of h is the ratio of the size of V(G') to the size of V(G) (i.e., $|V(G')|/|V(G)|$, where V(G) and V(G') are th 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 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. The smaller the value of the dilation, 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 images in G' is the maximum value of the congestion for the chosen mapping h. The smaller the value of the maximum 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.

2.2 Mapping Algorithm I

The first mapping algorithm was presented by Stout [4]. Stout's algorithm 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 $2^n x 2^n$ contains a total of $\lfloor 2^{2(n+1)/3} \rfloor$ nodes, the expansion is less than 1.

The n-bit Reflected Gray Code is used to transform the row and column numbers in the base of the pyramid with a one-to-one mapping. Hence, each PE in the base of the pyramid is mapped onto a single PE in the hypercube by obtaining a PE address through interleaving of the bits in the transformed row and column numbers. This process produces a perfect mapping for the base of the pyramid. Thus, all PEs of the hypercube are used to simulate the nodes in the base (i.e., level 0) of the pyramid. To simulate the next level PEs of the pyramid, 1/4 of the hypercube's PEs are employed. As a matter of fact, one of the PEs in each sequence of four children will simulate their parent, and one of the children will have to send data to its parent over two communication links. The PEs which have the least significant bits 0 in the transferred row and column numbers are used to represent the parents in the next higher level. 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. The two main advantages of this mapping are the small resultant dilation (i.e., the dilation of such a data transfer is equal to two) and the relatively small number of hypercube processors required.

Figure 2.1 shows the mapping of the P_3 pyramid onto the H_6 hypercube; the numbers within the squares represent level numbers. By 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 maximum congestion of this mapping is equal to two. As mentioned earlier, the total number of PEs in the target hypercube is smaller than the total number of nodes in the source pyramid.

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. Thus, if multiple levels of the pyramid need to be active simultaneously, not only will some hypercube PEs not be capable of simulating nodes from several levels of the pyramid simultaneously but also may spend some extra time in switching from one simulation to the next; in addition, the storage 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 [10]. However, this mapping does not consume a prohibitively long period of 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.

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 2.1 Mapping the P_3 pyramid onto the H_6 hypercube with Algorithm I (RGC: 3-bit Reflected Gray Code)

2.3 Mapping Algorithm II

2.3.1 Mapping the Pyramid

Similar to Algorithm I, the mapping algorithm proposed by Patel and Ziavras [7], Algorithm II herein, maps the P_n pyramid onto the H_{2n} hypercube. However , in contrast to Algorithm I, Algorithm II allows multiple levels of the pyramid to be active simultaneously. More specifically, Algorithm II allows any subset of levels, excluding the leaf level (i.e., level 0), to be active at one time. The simulation of the leaf level excludes the simultaneous simulation of other levels in the pyramid because the total number of leaf nodes is the same as the number of PEs in the hypercube.

The embedding algorithm proceeds as follows. Similarly to Stout's algorithm, the n-bit Reflected Gray Code is used to independently encode the row and column numbers of the leaf level of the P_n pyramid. A perfect mapping is then produced for this level and the H_{2n} hypercube 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 Algorithm I. More specifically, every PE of the next level of the pyramid has four children at the leaf level, so one PE is chosen from each square of four PEs to represent the parent PE. The PEs of the hypercube chosen to simulate these parents are those for which both the transformed column and row numbers have their least significant bits equal to 0 (as in Stout's algorithm).

For each set of four PEs which represent sibling nodes at this level of the pyramid, a PE is again chosen to represent their parent at the next level. The PE chosen to serve as the parent is the neighbor of one of the PEs representing the children and all the parent PEs for level 2 form mirror images in squares outlined by the children. This procedure is repeated until the apex of the pyramid is reached.

For example, as shown in Figure 2.2, the leaf 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 2x2 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 level 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.

We can see that the maximum dilation of the embedding for an edge connecting a parent at level 1 and one of its children at level 0 is 2 (as for Algorithm I). However, the maximum dilation for higher levels is equal to three. The maximum congestion for lower and higher levels is 2. In general, both the maximum dilation and the maximum congestion associated with this mapping algorithm are 3 and 2 respectively.

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 2.2 The Mapping of the P_3 pyramid onto the H_6 hypercube with Algorithm II (RGC: 3-bit Reflected Gray Code)

The above algorithm is generalized in the following subsection for the mapping of multilevel structures onto the hypercube.

2.3.2 Mapping Multilevel Structures

The algorithm developed by Patel and Ziavras that maps the pyramid onto the hypercube can be extended for the mapping of multilevel systems. Multilevel systems have reductions $2^m x 2^m$, where m are natural numbers, instead of 2x2 as in the pyramid. In addition, the reductions between different pairs of neighboring levels may differ. In general, the mapping of a level with total reduction $2^t x 2^t$ with respect to the base of the multilevel structure is identical to that of level n-t of the pyramid.

The generalized algorithm to map a multilevel structure onto the hypercube is presented in mathematical form below.

The introduction of the following variables is pertinent.

- f(i,x,y)=(j,k) is a mapping function which maps the PE(i,x,y) of the P_n pyramid onto the PE of the H_{2n} hypercube with transformed row and column addresses j and k respectively.
- l: for a P_l.
- m(i,i+1) : $2^{m(i,i+1)}x2^{m(i,i+1)}$ is the reduction between levels i and i+1.
- $Gray_k(m)$: k-bit Reflected Gray code of m.
- k: auxiliary variable.

The algorithm is as follows.

i=0; k=0; $f(l,x,y) = (Gray_l(x),Gray_l(y))$.

for i < l then

i=i+1; k=k+m(i-1,i);

$f(i,x,y) =(Gray_{(l-j)}(x).(j0s),Gray(l-j)(y).Gray_j(k-1))$;

$$\text{where } j = \sum_{z=0}^{i-1} m(z,z+1)$$

is the total reduction between levels i-1 and i.

As a consequence, the following are true:

- Level 1 of P_l is mapped onto PEs of H_{2n} having row and column addresses with 0 in their least significant bit.
- Level 2 of P_l is mapped onto the PEs of H_{2n} with row and column addresses equal to $Gray_{(l-1)}$ (0).0 and $Gray_{(l-1)}$(1).0 respectively and its mirror images.

The maximum dilation for edges that connect parent and children from levels i and i-1 respectively is 2m(i-1,i)+1. Stout's algorithm can also be extended for the mapping of multilevel structures. The resultant maximum dilation is equal to 2m(i-1,i).

2.3.3 Mapping Overlapped Multilevel Structures

Overlapped multilevel structures are similar to standard multilevel structures except that each parent is also linked to children from neighboring groups. For example, the overlapped pyramid structure is the same as the pyramid structure except that instead of four children each parent has 16 children. It is obtained from the standard pyramid by extending the area occupied by the children by 50 % in each direction. Hence, each child has four parents. Such a structure is appropriate for some segmentation algorithms in image processing [3]. As a consequence, it becomes imperative to develop algorithms for mapping such structures onto hypercube.

The algorithms of this chapter which map the pyramid or multilevel structures onto the hypercube are also applicable for the mapping of the corresponding overlapped structures onto the hypercube. However, the dilation and congestion will increase as the number of children which communicate with the same parent increases. For example, for the

overlapped pyramid the maximum dilation and maximum congestion will be 4 and 8 respectively.

2.4 Comparison with Other Existing Algorithms

There are four existing algorithms that map pyramids onto hypercubes. Two algorithms, other than Algorithms I and II discussed earlier, were proposed by Lai and White [5,6]. Both Algorithms I and II need an H_{2n} hypercube to embed a P_n pyramid. In contrast, the algorithms presented by Lai and White need a H_{2n+1} hypercube to map a P_n pyramid. Therefore, the cost associated with the mapping algorithms of Lai and White is much higher. As a result, the mapping algorithms proposed by Lai and White will not be discussed in this thesis.

We should remind that, Algorithm II presented by Patel and Ziavras for the mapping of the pyramid onto the hypercube has maximum dilation and maximum congestion 3 and 2 respectively, while Stout's algorithm, i.e. Algorithm I has 2 and 2 respectively for these metrics. Thus, the Algorithm II will be inferior Algorithm I with respect to the communication delay as the dilation is increased by 1 in Algorithm II. On the other hand, Algorithm II is superior to Algorithm I with respect to the total execution time when several levels of the pyramid are considered to be active at the same time. This is due to the fact that Algorithm I does not allow concurrent simulation of multiple levels of the pyramid. However, the only type of concurrency not allowed by Algorithm II is the concurrent simulation of the leaf level along with other levels.

It can be seen that, four pyramids could be simulated at the same time with the same dilation and congestion of 2 when Stout's mapping algorithm is used. These pyramids will have the same base, which will be simulated by

all PEs of the hypercube. One of four PEs in each group of the base simulates a parent at the next level. The remaining three PEs in each group can be used to concurrently simulate three more pyramids of the same size.

In contrast, Algorithm II can simulate only two such pyramids concurrently. Since different PEs of the hypercube simulate different levels of the pyramid, only one more pyramid can be simulated at the same time with the remaining PEs of the hypercube, for the same maximum dilation and maximum congestion of 3 and 2 respectively. All levels of both pyramids (except for their leaf level) will be active simultaneously. PEs marked with prime numbers in Figure 2.3 simulate the second pyramid.

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	0,1'	0,2'	0,3'	0,1'	0,1'	0	0,2'	0,1'
011	0,1'	0	0	0,1'	0,1'	0	0	0,1'
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	0,1'	0	0	0,1'	0,1'	0	0	0,1'
101	0,1'	0,2'	0	0,1'	0,1'	0	0,2'	0,1'
100	0,1	0,2	0	0,1	0,1	0	0,2	0,1

Figure 2.3 The mapping of two P_3 pyramids onto the H_6 hypercube with Algorithm II.

CHAPTER 3
COMPARATIVE ANALYSIS

3.1 Image Processing Algorithms

This chapter carries out a comparative analysis using of simulation results. It involves the two mapping algorithms of the previous chapter. In fact, simulation results are derived for three important image processing algorithms which are perimeter counting of objects, 2-D convolution, and segmentation of an image.

3.1.1 Perimeter Counting of Objects

This application algorithm assumes the existence of a single object and the assignment of a single pixel with a value of 0 or 1 to each node at the leaf level of the pyramid. PEs containing 1 from the previous assignment correspond to boundary pixels. Hence, a bottom-up process is applied to count the total number of boundary pixels. More specifically, nodes at the leaf level (level 0) of the pyramid that contain a boundary pixel send 1 to their parent at the next level (level 1), while the others send 0 to their parent. Each parent at the next level sums the four values it receives from its children and transmits the result to its parent at the next level (level 2). This process is repeated until the topmost level (apex PE) is reached. After the addition of the values received by the apex PE, the perimeter of the object is obtained.

3.1.2 2-D convolution

Two-dimensional convolution is a common operation in the area of image processing. The 2-D convolution algorithm using the pyramid structure convolves a kxk window of weighting coefficients with a 2^n x 2^n image matrix at the leaf level. In practice, k is much smaller than n. Let $X = \{ x_{i,j} \}$ and $W = \{ W_{i,j} \}$ be the image matrix and the window respectively. The 2-D convolution problem is to compute $Y = \{ y_{r,s} \}$ where

$$y_{r,s} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} W_{i,j} * X_{r+i,s+j} \qquad (3.1)$$

with $0 \le r,s \le 2^n-k$

We assume that the $2^n x 2^n$ image matrix has been loaded into the leaf level nodes, one pixel per node. Therefore, the 2-D convolution algorithm is divided into three phases:

1. The smallest integer r is found for which $2^r \ge k$. Then the leaf level of the pyramid is partitioned into square blocks of size $2^r x 2^r$. Each such partition contains the leaves of a subpyramid whose apex is at level r.
2. After partitioning the base nodes into blocks, the weighting coefficients are 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. The rest of the PEs in each partition receive a zero as the weighting coefficient.
3. The pyramid then computes the 2-D convolution. The results are stored in the base nodes. The result $y_{r,s}$ is stored in a register of the base node (r,s).

It should be noted that phases 1 and 2 are not included in the total execution time of the presented results. For phase 3, more detail follows. The PEs at the leaf level multiply the weighting coefficients with the pixel values they contain, and send the results to their parents at next level (level 1). Parents at level 1 sum the four values they receive from their four children and send the result to their parent. This process is repeated until the apexes of the subpyramids at level r are reached. Each apex 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 leaf level that contains the weighting coefficients is shifted to the right once, multiplications are performed as above, the results are then shifted to the left once, and the values are sent to the parents at level 1. Then the bottom-up and top-down processes described above are applied with the result now stored in the PE with offset (0,1) in the partition. It is obvious that the 2-D convolution algorithm involves lateral shifts and multiplications at the leaf level, bottom-up additions of products, and finally top-down transmissions of final results.

No matter what the window size k is, those steps described earlier are repeated 2^{2r} times which is equal to the total number of PEs in each partition. For instance, these steps are repeated 16 times for window sizes 3x3 and 4x4 because $2^{2r}=16$, with $2^r \geq k$.

3.1.3 Segmentation of Images

Segmentation is the process which partitions the image into regions with more or less homogeneous property; but the process which estimates these properties should be confined within individual regions. Segmentation and image properties are computed in a cooperative, iterative fashion. The

results obtained for each task at one iteration are used to adjust and improve the performance of the other task at the next iteration. This approach uses an overlapped pyramid where each node in the pyramid has four parents and 16 children.

A father-son relationship is defined between nodes in adjacent levels but this relationship is not fixed and may be redefined at each iteration. In each iteration the node is linked to a single one of these four higher level candidate father nodes. The father-son links then define windows in the image and ultimately the image segments. The window for a given node is just the sum of its son's windows, although the actual size and shape of windows will vary from node to node at a given level and from iteration to iteration for a given node.

There are four time dependent variables associated with each node:

- C[i,j,l][t]: the value of the local image property;
- a[i,j,l][t]: the area over which the property was 3computed;
- 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 values for the entire segment containing the node;

Here time is the iteration number, a positive integer.

For each node [i,j,l] with l>0 (l is the level number), there is a 4x4 subarray of candidate son nodes at [i', j', l-1] where

i' = 2i-1, 2i, 2i+1, 2i+2

j' = 2j-1, 2j, 2j+1, 2j+2

On the other hand, each node below the top level has four candidate father nodes at [i", j", l+1] where

i"={ (i-1)/2 } or { (i+1)/2 }

j"={ (j-1)/2 } or { (j+1)/2 }

Here {.} indicates the integer part of the fraction enclosed.

In the initial iteration, the value of C for each leaf level node is set equal to the corresponding image sample value, while the C value for each higher level node is the average of all 16 of the node's candidate sons. All iterations following initialization (t>0) are divided into three phases:

1. Father-son links are established for all nodes below the top of the pyramid. The way used to choose the father-son link is as follows:
   The m^{th} parent is chosen, where d[m] is the smallest absolute difference between the C value of node [i,j,l] and all of its candidate parents. The decision is made at random if the value of d[n] for two or more of the candidate fathers are equal.

2. The C and S values are computed bottom up on the basis of the new son-father links.
   For l=0 a[i,j,l][t]=1
   For 0<l<L a[i,j,l][t] is the sum of areas over those sons of node [i,j,l] assigned in phase 1
   If a[i,j,l][t] >0 then $C[i,j,l] = \Sigma(a[i,j,l-1][t]*C[i',j',l-1][t])/a[i,j,l][t]$

3. Segment values are assigned top down.
   At the top most 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 lower level l<L, each node's value is just that of its father.

At the end of phase 3, the level 0 segment values represent the current state of the smoothing-segmentation process. Any changes 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 repeated until no changes occur from one iteration to the next.

3.2 Simulation Results

Simulation results for the aforementioned image processing algorithms using the two mapping algorithms that map the pyramid onto the hypercube will be discussed in this section. Some definitions used for the calculation of the execution time are expressed in machine cycles. The scanning delay is 2; it is the time needed to load the values of pixels into the corresponding PEs at the leaf level. The communication time for a single value is 2, the set up time to receive or transmit a single value is 1. The addition time is 1, and both the multiplication and division times are 2.

Table 3.1 shows simulation results for the algorithm of perimeter counting, for only one level of the pyramid being active at a time. Stout's algorithm performs better than Ziavras' Algorithm. This is because of its smaller dilation (D=2 compared, to D=3 in Ziavras' algorithm) and hence reduced communication delay between adjacent levels. However, the pyramid machine is more efficient than the hypercube machine for this algorithm. Since the communication time between adjacent simulated levels increase on the hypercube due to increased dilation and congestion of the mapping, the better performance of the pyramid should be expected.

For multiple levels being active simultaneously, only Ziavras' algorithm can be implemented on the hypercube machine. As expected, a pyramid machine performs better than Ziavras' algorithm for the hypercube.

For different mapping algorithms, the total numbers of PEs used may differ. As shown in Table 3.1, the average utilization of hypercube PEs for the two algorithms are different. It must be emphasized that communication times are not included in the calculation of utilization because they correspond to pure overhead.

Results for the two-dimensional convolution algorithm are shown in Table 3.2. Due to the perimeter implementation, as discussed earlier, only result for 2x2, 4x4, and 8x8 (i.e., power of 2) should be presented. The results show again that Ziavras' algorithm has worse performance than Stout's algorithm due to its larger dilation when only one level is active at a time. As in the case of the perimeter counting algorithm, the pyramid performs better than the hypercube. Note that the number of levels in the pyramids is not shown because only levels 0 through r are involved. For 4x4 convolution, only the lowest three levels of the pyramid are used, while for 8x8 convolution, the lowest four levels are used.

When multiple levels are active simultaneously, only Ziavras' algorithm can be applied. The use of pipelining also raises the average utilization of the PEs.

Table 3.3 shows the comparison of the times needed for lateral data transfers at the leaf level and for processing the entire image for 2-D convolution. The results in Table 3.3 indicate that there are no differences for the times of lateral data transfers. The main reason for the different total execution times is the increased communication delay between adjacent levels in Ziavras' algorithm.

Simulation results for the segmentation algorithm that utilizes overlapped pyramids are shown in Table 3.4. Stout's algorithm and Ziavras' algorithm have the same performance with respect to the total execution time for one level being active at a time. They yield higher utilization than the pyramid machine. This is because the total number of hypercube PEs used in the algorithm is smaller than the number used with the overlapped pyramid structure. Results are not presented for concurrent multilevel processing because the algorithm is inherently sequential in nature. Generally, the performance of the pyramid is better than that of the hypercube.

Results of concurrently simulating two pyramids on the hypercube are shown in Table 3.5. Algorithms for perimeter counting and two-dimensional convolution of an image are implemented simultaneously on the same hypercube. The execution time is basically determined by convolution. Here the reduction is 2^2x2^2 for window size of 4x4. Hence, only the lowest three levels of the pyramid are used for 4x4 convolution. For window size of 8x8, the lowest four levels are used. The reductions are either 2^2x2^2 or 2x2, therefore two cases are considered for multilevel structures. For the first case of 8x8 window, the reduction between levels 0 (leaf level) and 1 is 2x2, and it is 2^2x2^2 between levels 1 and 2. On the contrary, the reduction between levels 0 and 1 is 2^2x2^2, and between levels 1 and 2 is 2x2 for the second case. But both cases yield the same performance because of the chosen timings for the simulation. The advantage of Ziavras' algorithms is that it can simulate multiple levels simultaneously. All three image processing algorithms that were simulated in this thesis illustrate a major improvement in performance for this algorithm.

Table 3.1 Simulation results for perimeter counting on Hypercubes
(C: congestion, D: Dilation)

Stout's Algorithm; D=2; C=2; H_{2n}

One Level Active			Multiple Levels Active			# of Levels	# of PEs in Hypercube
Ext.Time	Utilization Avg.	Max.	1/Throughput	Utilization Avg.	Max.		
29	15.89	20.69				3	64
38	12.24	15.79				4	256
47	9.92	12.77				5	1024
56	8.33	10.71				6	4096
65	7.18	9.23				7	16384

Ziavras' Algorithm; D=3; C=2; H_{2n}

37	12.46	16.22	13	35.46	46.15	3	64
50	9.30	12.00	13	35.79	46.15	4	256
63	7.40	9.52	13	35.87	46.15	5	1024
76	6.14	7.90	13	35.89	46.15	6	4096
89	5.24	6.74	13	35.90	46.15	7	16384

Simulation on Pyramids

17	13.15	17.65	5	44.70	60.00	3	85
22	10.20	13.64	5	44.90	60.00	4	341
27	8.33	11.11	5	44.98	60.00	5	1365
32	7.03	9.38	5	45.01	60.00	6	5461
37	6.08	8.10	5	45.03	60.00	7	21845

Table 3.2 Simulation results for convolution on Hypercubes

Stout's Algorithm; D=2; C=2; H_{2n}

One Level Active			Multiple Levels Active			Size of Window	# of PEs in Hypercube
Ext.Time	Utilization Avg.	Max.	1/Throughput	Utilization Avg.	Max.		
88	35.23	48.86				2x2	4
700	28.71	42.71				4x4	16
4668	28.34	47.54				8x8	64

Ziavras' Algorithm; D=3; C=2; H_{2n}

88	35.23	48.86	88	35.23	48.86	2x2	4
764	27.49	40.58	615	34.15	46.18	4x4	16
5172	26.45	43.98	3941	34.71	54.81	8x8	64

Simulation on Pyramids

56	25.71	31.14	33	43.64	54.55	2x2	5
364	15.91	21.43	131	44.20	59.54	4x4	21
1980	11.18	15.61	510	43.40	60.59	8x8	85

Table 3.3 Comparison of lateral data transfer with total execution time for 2-D convolution

Stout's Algorithm; D=2; C=2; H_{2n}

Lateral	Total	Window Size
28	88	2x2
252	700	4x4
2044	4668	8x8

Ziavras' Algorithm; D=3; C=2; H_{2n}

Lateral	Total	Window Size
28	88	2x2
252	764	4x4
2044	5172	8x8

Table 3.4 Simulation results for segmentation on Hypercubes

Stout's Algorithm; D=2; C=2; H_{2n}

One Level Active			Multiple Levels Active			# of Levels	# of PEs in Hypercube
Ext.Time	Utilization Avg.	Max.	1/Throughput	Utilization Avg.	Max.		
1048	27.77	32.25				3	64
1567	19.38	21.57				4	256
2086	14.70	16.20				5	1024
2605	11.79	12.98				6	4096
3124	9.84	10.82				7	16384

Ziavras' Algorithm; D=3; C=2; H_{2n}

1048	27.77	32.25	519	56.08	65.13	3	64
1567	19.38	21.57	519	58.52	65.13	4	256
2086	14.70	16.20	519	59.08	65.13	5	1024
2605	11.79	12.98	519	59.20	65.13	6	4096
3124	9.84	10.82	519	59.22	65.13	7	16384

Simulation on Pyramids

488	21.78	31.97	189	56.25	82.54	3	85
727	15.26	21.46	189	58.71	82.54	4	341
966	11.61	16.15	189	59.32	82.54	5	1365
1205	9.33	12.95	189	59.47	82.54	6	5461
1444	7.79	10.80	189	59.50	82.54	7	21845

Table 3.5 Simulation results for convolution and perimeter counting on Multilevel Pyramid

Stout's Algorithm; D=2; C=2; H_{2n}

One Level Active			Multiple Levels Active				
Ext.Time	Utilization Avg.	Max.	1/Throughput	Utilization Avg.	Max.	Size of Window	# of PEs in Hypercube
700	28.71	42.71				4x4	16
4668	28.34	47.54				8x8	64
4668	28.34	47.54				8x8	64

Ziavras' Algorithm; D=3; C=2; H_{2n}

700	28.71	42.71	700	28.71	42.71	4x4	16
4916	27.09	45.40	3187	41.79	70.03	8x8	64
4916	27.09	45.40	3187	41.79	70.03	8x8	64

Simulation on Pyramids

236	26.32	33.05	129	48.15	60.47	4x4	17
1468	17.33	21.66	491	51.82	64.76	8x8	81
1468	20.34	21.66	491	60.83	64.76	8x8	69

CHAPTER 4
CONCLUSION

This thesis has investigated the performace of two algorithms that map multilevel structures onto hypercubes. Such mappings are very important due to the robustness of the hypercube network with respect to the efficient emulation of several topologies. Ziavras' algorithm performs better than Stout's algorithm when multiple levels of the pyramid are considered to be active simultaneously. This is because only Stout's algorithm does not allow multiple levels of the pyramid to be active simultaneously. On the other hand, when only one level of the pyramid must be active at a time, Stout's algorithm yields better performance than Ziavras' algorithm because of its lower dilation which results in smaller communication time. However, Ziavras' algorithm achieves very good performance when only one level is active at a time. In contrast, Ziavras' algorithm improves the performance dramatically when multiple levels must be active simultaneously.

The mapping of overlapped multilevel structures onto hypercubes was also investigated. Three algorithms from the image processing domain were considered for a comparative analysis.

APPENDIX

```
                /*************************/
                /***  Stout Algorithm  ***/
                /***   One PE /PIXEL   ***/
                /***    PERIMETER      ***/
                /*************************/


#include "stdio.h"
#include "stdlib.h"
#include "math.h"
unsigned long int k,Texe;
float Uavg,Umax,H,H1;
unsigned long int Oep, max;
int i,j,x,y,a[64][64],n,C[128][128];
int Tload,l,d,Omax,inc,Base,Tcomm,Tset,Tadd;
void Dec_Gray_();

main()
 {
   FILE *fp,*fg;
   Tload=2;
   Tcomm=2;
   Tset=1;
   Tadd=1;
   fg=fopen("peri.sto","w");
   fprintf(fg,"\n n   Processors    Oep    Texe    Uavg
              Umax \n\n");
   for(n=3;n<=7;n++)
   {
     Omax=0;
     Oep=0;
     Texe=0;
     Base=(pow(2,n))-1;
     if((fp=fopen("mat.dat","r")) == NULL) exit (1);
     for(i=0;i<=Base;i++)
     {
    for(j=0;j<=Base;j++)
    {
     fscanf(fp,"%d ",&a[i][j]);
     printf("%d\t",a[i][j]);
     Dec_Gray_();
     C[x][y]=a[i][j];
    }

    printf("\n");
     }

   Texe=Texe+Tload;
   H=(Base+1)*(Base+1);
   Oep=Oep+H;
   inc=2;
   for(x=0;x<=Base;x=x+inc)
     for(y=0;y<=Base;y=y+inc)
```

```
      {
    C[x][y]=C[x][y]+C[x][y+1];
    C[x][y+1]=C[x+1][y+1];
    C[x][y]=C[x][y]+C[x+1][y];
    C[x][y]=C[x][y]+C[x][y+1];
     }
   Omax=Omax+2;
   Oep=Oep+11*H/4;
   Texe=Texe+Tcomm+2*Tset+Tadd;
   Omax=Omax+2;
   Texe=Texe+Tset+Tadd;
   Omax=Omax+2;
   Texe=Texe+Tset+Tadd;
    for(l=2;l<=n;l++)
    {
      inc=pow(2,l);
      d=pow(2,l-1);
      for(x=0;x<=Base;x=x+inc)
        for(y=0;y<=Base;y=y+inc)
        {
          C[x][y]=C[x][y]+C[x][y+d];
          C[x][y+d]=C[x+d][y+d];
          C[x][y]=C[x][y]+C[x+d][y];
          C[x][y]=C[x][y]+C[x][y+d];
        }
        Texe=Texe+Tcomm+2*Tset+Tadd;
        Texe=Texe+Tset+Tadd;
        Texe=Texe+Tset+Tadd;
        H1=H/pow(2,2*l);
        Oep=Oep+11*H1;
    }
          Umax=(float)Omax/Texe;
          i=2*n;
          j=pow(2,i);
          k=Texe*j;
          Uavg=(float)Oep/k;
          fprintf(fg,"%d\t %d\t %lu\t %d\t",n,j,Oep,Texe);
          fprintf(fg,"%5.4f\t %5.4f\n",Uavg,Umax);
   }
          printf("\n Output C0 %d",C[0][0]);
 }

void Dec_Gray_()
{
  x=i>>1;
  x^=i;
  y=j>>1;
  y^=j;
}
```

```
/*****************************/
/***  Ziavras Algorithm  ***/
/***    One PE /PIXEL    ***/
/***      PERIMETER      ***/
/*****************************/


#include "stdio.h"
#include "stdlib.h"
#include "math.h"
unsigned long int Texe;
float Uavg,Umax,Upipmax,k,H,H1;
long int Oep;
double Upipavg;
unsigned  int Tpipe,max;
int i,j,x,y,a[64][64],n,C[128][128],l,d,Omax,inc,Base;
int Tset,Tadd,Tl1,Tcomm,Tload,hostx,hosty;
int x1,y1;
void Dec_Gray_();
void Host_();

main()
 {
   FILE *fp,*fg;
   Tload=2;
   Tcomm=2;
   Tset=1;
   Tadd=1;
   fg=fopen("peri.zia","w");
   fprintf(fg,"\n n   Processors  Tseq   Texe    Uavg");
   fprintf(fg,"    Umax    Tpipe  Upipavg  Upipmax\n\n");
   for(n=3;n<=7;n++)
   {
     Omax=0;
     Oep=0;
     Texe=0;
     Base=(pow(2,n))-1;
     if((fp=fopen("mat.dat","r")) == NULL) exit (1);
     for(i=0;i<=Base;i++)
     {
    for(j=0;j<=Base;j++)
    {
     fscanf(fp,"%d ",&a[i][j]);
     printf("%d\t",a[i][j]);
     Dec_Gray_();
     C[x][y]=a[i][j];
    }
    printf("\n");
     }

   Texe=Texe+Tload;
   H=(Base+1)*(Base+1);
```

```
Oep=Oep+H;
inc=2;
max=pow(2,2*(n-1));
for(x=0;x<=Base;x=x+inc)
{
  for(y=0;y<=Base;y=y+inc)
  {
 C[x][y]=C[x][y]+C[x][y+1];
 C[x][y+1]=C[x+1][y+1];
 C[x][y]=C[x][y]+C[x+1][y];
 C[x][y]=C[x][y]+C[x][y+1];
  }
}
Omax=Omax+2;
Oep=Oep+11*H/4;
Texe=Texe+Tcomm+2*Tset+Tadd;
Omax=Omax+2;
Texe=Texe+Tset+Tadd;
Omax=Omax+2;
Texe=Texe+Tset+Tadd;
Tl1=Texe;
x1=0;
y1=0;
  for(l=2;l<=n;l++)
  {
   Host_();
   inc=pow(2,l);
   d=pow(2,l-1);
   for(x=x1;x<=Base;x=x+inc)
 for(y=y1;y<=Base;y=y+inc)
 {
  C[x+hostx][y+hosty]=0;
  C[x+hostx][y+hosty]=C[x+hostx][y+hosty]+C[x][y];
  C[x+hostx][y+hosty]=C[x+hostx][y+hosty]+C[x][y+d];
  C[x][y+d]=C[x+d][y+d];
  C[x+hostx][y+hosty]=C[x+hostx][y+hosty]+C[x+d][y];
  C[x+hostx][y+hosty]=C[x+hostx][y+hosty]+C[x][y+d];
 }
     x1+=hostx;
     y1+=hosty;
     Texe=Texe+Tcomm+2*Tset+Tadd;
     Texe=Texe+Tcomm+Tset+Tadd;
     Texe=Texe+Tcomm+Tset+Tadd;
     H1=H/pow(2,2*l);
     Oep=Oep+11*H1;
     if (l==2) Tpipe=Texe-Tl1;
   }
     Umax=(float)Omax/Texe;
     Upipmax=(float)Omax/Tpipe;
     i=2*n;
     j=pow(2,i);
     k=Texe*j;
     Uavg=(float)Oep/k;
     k=(float)j*Tpipe;
```

```
        Upipavg=(float)Oep/k;
        fprintf(fg,"%d\t %d\t %ld\t %d\t",n,j,Oep,Texe);
        fprintf(fg,"%5.4f\t %5.4f\t",Uavg,Umax);
        fprintf(fg,"%u\t ",Tpipe);
        fprintf(fg," %5.4f\t %5.4f\n",Upipavg,Upipmax);
    }
        printf("output C[%d][%d] %d",x1,y1,C[x1][y1]);
}

void Dec_Gray_()
{
  x=i>>1;
  x^=i;
  y=j>>1;
  y^=j;
}

void Host_()
{
  if (3<=l) {i=0;j=l-1;}
  else if ((l==4)||(l==7)) {i=1;j=0;}
    else if (l==5) {i=0;j=-2;}
         else if (l==6) {i=0;j=-1;}
           else {i=0;j=1;}
  hostx=i;
  hosty=j;
}
```

```
/*****************************************/
/***   Convolution Stout's Algorithm ***/
/*****************************************/


#include "stdio.h"
#include "stdlib.h"
#include "math.h"

int i,j,k,l,d,colm,row;
int max,Base,shu,shl,x,y,Texe,Tload,temp[10][10];
int Tset,Tadd,Tmul,Omax,W[10][10],inc;
int Base,Tcomm,wsize,a,dx,dy,m;
float Uavg,Umax;
int next1,next2;
int C[10][10],P[10][10],conv[10][10];
long int Oep;

void Product_Wc_();
void Sum_();
void Shift_R_();
void Shift_D_();
void Shift_U_();
void Shift_L_();
void Dec_Gray_();
void Dist_();

main()
 {
   Tload=2;
   Tcomm=2;
   Tset=1;
   Tadd=1;
   Tmul=2;
   Oep=0;
   Omax=0;
   Texe=0;
   printf("Please input wsize");
   scanf("%d",&wsize);
   k=pow(2,wsize);
   Base=k-1;
   printf("Please input window coefficient\n");
   for(i=0;i<=Base;i++)
    for(j=0;j<=Base;j++)
    {
     scanf("%d",&a);
     Dec_Gray_();
     W[x][y]=a;
     temp[x][y]=W[x][y];
     }
   printf("Please input image matrix\n");
   for(i=0;i<=Base;i++)
    for(j=0;j<=Base;j++)
```

```
 {
  scanf("%d",&a);
  Dec_Gray_();
  C[x][y]=a;
 }
shu=0;
for(row=0;row<=Base;row++)
{
  shl=0;
  for(colm=0;colm<=Base+1;colm++)
  {
 if(colm==Base+1){ Shift_R_();
             Texe=Texe-2*Tset-Tcomm;
             Oep=Oep-2*pow(k,2);
           }
 else{ if((row==0)&&(colm==0)){  Product_Wc_();
                      for(l=1;l<=wsize;l++)
                      Sum_();
                      conv[0][0]=P[0][0];
                      }
       else{ if((row!=0)&&(colm==0)){ Shift_D_();
                          shu=shu+1;
                        }
          else{ Shift_R_();
             shl=shl+1;
             if((row==Base)&&(colm==Base))
             { Omax=2*pow(k,2)+Omax;}
           }
         Product_Wc_();
         if((row==Base)&&(colm==Base))
         { Omax=Omax+pow(k,2);}
         if(shu!=0)
        {
         for(m=1;m<=shu;m++)
         {
          Shift_U_();
          if((row==Base)&&(colm==Base))
          { Omax=2*pow(k,2)+Omax;}
         }
        }
          if(shl!=0)
          {
           for(m=1;m<=shl;m++)
           {
             Shift_L_();
             if((row==Base)&&(colm==Base))
             { Omax=Omax+2*pow(k,2);}
           }
          }
 for(l=1;l<=wsize;l++)
 { Sum_();
  if((row==Base)&&(colm==Base)) Omax=Omax+11*pow(max,2);
 }
  i=row;
```

```
        j=colm;
        Dec_Gray_();
        conv[x][y]=P[0][0];
        Dist_();
        Texe=Texe+4*(dx+dy);
        Oep=Oep+2*(dx+dy);
        if((row==Base)&&(colm==Base))
        { Omax=Omax+(dx+dy)*2;}
        }
      }
     }
    }
    Uavg=(float)Oep/(Texe*pow(k,2));
    Umax=(float)Omax/Texe;
/**printf("\n %d\t %5.4f\t %5.4f\t%d\t%d\t
%ld\n",Texe,Uavg,Umax,k,k,Oep);**/
 for (i=0;i<=Base;i++)
   {
    for (j=0;j<=Base;j++)
     {
      Dec_Gray_();
      printf("%d  ",conv[x][y]);
      }
     printf("\n");
    }
}



void Product_Wc_()
 {
   for(i=0;i<=Base;i++)
    for(j=0;j<=Base;j++)
     {
      Dec_Gray_();
      P[x][y]=temp[x][y]*C[x][y];
     }
    Texe=Texe+Tmul;
    Oep=Oep+pow(k,2);
  }


  void Sum_()
  {
     max=pow(2,wsize-1);
     inc=pow(2,1);
     d=pow(2,1-1);
     for(x=0;x<=Base;x=x+inc)
      for(y=0;y<=Base;y=y+inc)
      {
     P[x][y]=P[x][y]+P[x][y+d];
     P[x][y+d]=P[x+d][y+d];
     P[x][y]=P[x][y]+P[x+d][y];
```

```
    P[x][y]=P[x][y]+P[x][y+d];
     }
   Texe=Texe+9;
   Oep=Oep+11*max*max;
}


void Shift_R_()
{
 for(i=0;i<=Base;i++)
  for(j=0;j<=Base;j++)
   {
   Dec_Gray_();
   if(y==0){ next1=temp[x][0];
          temp[x][0]=temp[x][k/2];
        }
   else{ next2=temp[x][y];
      temp[x][y]=next1;
      next1=next2;
    }
   }
  Texe=Texe+2*Tset+Tcomm;
  Oep=Oep+2*pow(k,2);
}


 void  Shift_D_()
{
  for(j=0;j<=Base;j++)
   for(i=0;i<=Base;i++)
    {
     Dec_Gray_();
     if(x==0){ next1=temp[0][y];
         temp[0][y]=temp[k/2][y];
          }
     else{ next2=temp[x][y];
        temp[x][y]=next1;
        next1=next2;
      }
     }
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*pow(k,2);
}


 void Shift_U_()
 {
   for(j=Base;j>=0;j--)
   for(i=Base;i>=0;i--)
   {
    Dec_Gray_();
    if(x==k/2)  { next1=P[k/2][y];
              P[k/2][y]=P[0][y];
             }
```

```
    else{ next2=P[x][y];
          P[x][y]=next1;
          next1=next2;
        }
    }
    Texe=Texe+2*Tset+Tcomm;
    Oep=Oep+2*k*k;
}


void Shift_L_()
{
 for(i=Base;i>=0;i--)
    for(j=Base;j>=0;j--)
    {
     Dec_Gray_();
     if(y==k/2){ next1=P[x][k/2];
                 P[x][k/2]=P[x][0];
               }
     else{ next2=P[x][y];
           P[x][y]=next1;
           next1=next2;
         }
    }
 Texe=Texe+2*Tset+Tcomm;
 Oep=Oep+2*pow(k,2);
}


void Dec_Gray_()
{
 x=i>>1;
 x^=i;
 y=j>>1;
 y^=j;
}

void Dist_()
{
   int d1,d2;
   d1=x;
   d2=y;
   dx=0;
   dy=0;
   while (d1>0){  if((d1&1)>0) dx=dx+1;
                  d1=d1>>1;
               }
   while(d2>0){ if((d2&1)>0) dy=dy+1;
                d2=d2>>1;
              }
  }
```

```
/*****************************************/
/***   Convolution Ziavras' Algorithm ***/
/*****************************************/


#include "stdio.h"
#include "stdlib.h"
#include "math.h"

int i,j,k,l,d,colm,row;
int max,Base,shu,shl,temp[10][10];
int Tset,Tadd,Tmul,Omax,W[10][10],inc;
int Texe,Tload,Tcomm,wsize,a,m;
float Uavg,Umax;
int next1,next2;
int C[10][10],P[10][10],conv[10][10];
long int Oep;
int hostx,hosty,x,y,dx,dy;
int x1,y1;

void Product_Wc_();
void Sum_();
void Shift_R_();
void Shift_D_();
void Shift_U_();
void Shift_L_();
void Dec_Gray_();
void Dist_();
void Host_();

main()
 {
   Tload=2;
   Tcomm=2;
   Tset=1;
   Tadd=1;
   Tmul=2;
   Oep=0;
   Omax=0;
   Texe=0;
   printf("Please input wsize");
   scanf(" %d",&wsize);
   k=pow(2,wsize);
   Base=k-1;
   printf("Please input window coefficient\n");
   for(i=0;i<=Base;i++)
    for(j=0;j<=Base;j++)
    {
     scanf("%d",&a);
     Dec_Gray_();
     W[x][y]=a;
     temp[x][y]=W[x][y];
     }
```

```
printf("Please input image matrix\n");
for(i=0;i<=Base;i++)
 for(j=0;j<=Base;j++)
 {
  scanf("%d",&a);
  Dec_Gray_();
  C[x][y]=a;
 }
shu=0;
for(row=0;row<=Base;row++)
{
  shl=0;
  for(colm=0;colm<=Base+1;colm++)
  {
 if(colm==Base+1){ Shift_R_();
             Texe=Texe-2*Tset-Tcomm;
             Oep=Oep-2*pow(k,2);
            }
 else{ if((row==0)&&(colm==0)){  Product_Wc_();
                      x1=0;
                      y1=0;
                      for(l=1;l<=wsize;l++)
                      Sum_();
                      goto point1;
                      }
       else{ if((row!=0)&&(colm==0)){ Shift_D_();
                          shu=shu+1;
                        }
          else{ Shift_R_();
             shl=shl+1;
             if((row==Base)&&(colm==Base))
             { Omax=2*pow(k,2)+Omax;
             }
           }
         Product_Wc_();
         if((row==Base)&&(colm==Base))
         { Omax=Omax+pow(k,2);
         }
         if(shu!=0)
        {
         for(m=1;m<=shu;m++)
         {
          Shift_U_();
          if((row==Base)&&(colm==Base))
           Omax=2*pow(k,2)+Omax;
         }
        }
          if(shl!=0)
          {
           for(m=1;m<=shl;m++)
           {
             Shift_L_();
             if((row==Base)&&(colm==Base))
              Omax=Omax+2*pow(k,2);
```

```
                }
               }
      x1=0;
      y1=0;
      for(l=1;l<=wsize;l++)
      { Sum_();
       if((row==Base)&&(colm==Base))
        { if (l==1) Omax=Omax+11*pow(max,2);
          else  Omax=Omax+20*pow(max,2);
         }
      }
       point1:
       i=row;
       j=colm;
       Dec_Gray_();
       conv[x][y]=P[x1][y1];
       Dist_();
       Texe=Texe+4*(dx+dy);
       Oep=Oep+2*(dx+dy);
       if((row==Base)&&(colm==Base))
       { Omax=Omax+(dx+dy)*2;}
       }
     }
    }
   }
    Uavg=(float)Oep/(Texe*pow(k,2));
    Umax=(float)Omax/Texe;
/**   printf("\n %d\t %5.4f\t %5.4f\t%d\t%d\t
%ld\n",Texe,Uavg,Umax,k,k,Oep);**/
 for (i=0;i<=Base;i++)
 {
  for (j=0;j<=Base;j++)
  {
   Dec_Gray_();
   printf("%d  ",conv[x][y]);
   }
   printf("\n");
  }
}



void Product_Wc_()
 {
   for(i=0;i<=Base;i++)
    for(j=0;j<=Base;j++)
     {
      Dec_Gray_();
      P[x][y]=temp[x][y]*C[x][y];
     }
    Texe=Texe+Tmul;
    Oep=Oep+pow(k,2);
  }
```

```
void Sum_()
{
   Host_();
   max=pow(2,wsize-1);
   inc=pow(2,l);
   d=pow(2,l-1);
   for(x=x1;x<=Base;x=x+inc)
    for(y=y1;y<=Base;y=y+inc)
    {
     if (1<l)
   {
   P[x+hostx][y+hosty]=0;
   P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x][y];
   }
   P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x][y+d];
   P[x][y+d]=P[x+d][y+d];
   P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x+d][y];
   P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x][y+d];
    }
  x1+=hostx;
  y1+=hosty;
  if (l==1)
    { Texe=Texe+9;
     Oep=Oep+11*max*max;
     }
  else {Texe=Texe+13;
     Oep=Oep+20*max*max;
     }
}


void Shift_R_()
{
 for(i=0;i<=Base;i++)
  for(j=0;j<=Base;j++)
   {
   Dec_Gray_();
   if(y==0){ next1=temp[x][0];
          temp[x][0]=temp[x][k/2];
        }
   else{ next2=temp[x][y];
      temp[x][y]=next1;
      next1=next2;
    }
   }
  Texe=Texe+2*Tset+Tcomm;
  Oep=Oep+2*pow(k,2);
}


 void  Shift_D_()
{
```

```
  for(j=0;j<=Base;j++)
   for(i=0;i<=Base;i++)
    {
     Dec_Gray_();
     if(x==0){ next1=temp[0][y];
         temp[0][y]=temp[k/2][y];
          }
     else{ next2=temp[x][y];
        temp[x][y]=next1;
        next1=next2;
      }
    }
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*pow(k,2);
}


 void Shift_U_()
 {
   for(j=Base;j>=0;j--)
   for(i=Base;i>=0;i--)
   {
    Dec_Gray_();
    if(x==k/2)  { next1=P[k/2][y];
              P[k/2][y]=P[0][y];
             }
    else{ next2=P[x][y];
       P[x][y]=next1;
       next1=next2;
     }
   }
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*k*k;
 }



 void Shift_L_()
 {
  for(i=Base;i>=0;i--)
    for(j=Base;j>=0;j--)
    {
     Dec_Gray_();
     if(y==k/2){ next1=P[x][k/2];
            P[x][k/2]=P[x][0];
          }
     else{ next2=P[x][y];
        P[x][y]=next1;
        next1=next2;
       }
    }
  Texe=Texe+2*Tset+Tcomm;
  Oep=Oep+2*pow(k,2);
 }
```

```
void Dec_Gray_()
{
 x=i>>1;
 x^=i;
 y=j>>1;
 y^=j;
}

void Dist_()
{
  int d1,d2;
  d1=x^hostx;
  d2=y^hosty;
  dx=0;
  dy=0;
  while (d1>0){  if((d1&1)>0) dx=dx+1;
           d1=d1>>1;
         }
  while(d2>0){ if((d2&1)>0) dy=dy+1;
         d2=d2>>1;
       }
 }

void Host_()
{
  if (l<=3) {i=0;j=1-l;}
  else if ((l==4)||(l==7)) {i=1;j=0;}
    else if (l==5) {i=0;j=-2;}
         else if (l==6) {i=0;j=-1;}
           else {i=0;j=1;}
  hostx=i;
  hosty=j;
}
```

```
/*****************************************************/
/***** Simulation of Segmentation Algorithm *******/
/*****************************************************/


#include "stdio.h"
#include "math.h"
#include "stdlib.h"


int i,j,k,l,stop,m,row,col,layer;
int maxl,nc,n,m,iter,a[16][16][3];
int sx[8][8][3][16],sy[8][8][3][16];
int fx[16][16][3][4],fy[16][16][3][4];
int nchild[8][8][3],p[16][16][3];
float c[16][16][3],s[16][16][3];
float sum,psum,min,d[4];
void Cand_Father_();
void Cand_Son_();



main()
{
 FILE *fp;
 fp=fopen("matrix.dat","r");
 printf("Please input layer");
 scanf("%d",&layer);
 maxl=layer-1;
 k=pow(2,layer);
 stop=0;
 for(i=0;i<=k-1;i++)
  for(j=0;j<=k-1;j++)
   a[i][j][0]=1;
 for(i=0;i<=k-1;i++)
 {
  for(j=0;j<=k-1;j++)
   {
    fscanf(fp,"%f",&c[i][j][0]);
    printf("%3.1f\t",c[i][j][0]);
   }
    printf("\n");
 }
 Cand_Father_();
 Cand_Son_();
/****   Initial C Value   ****/
 for(l=1;l<=maxl;l++)
 {
   m=pow(2,layer-1);
   for(i=0;i<=m-1;i++)
    for(j=0;j<=m-1;j++)
     {
```

```
        sum=0.0;
        for(n=0;n<=15;n++)
        {
          row=sx[i][j][l][n];
          col=sy[i][j][l][n];
          sum+=c[row][col][l-1];
        }
        c[i][j][l]=sum/16.0;
        a[i][j][l]=16;
        printf("%6.3f",c[i][j][l]);
         }
         printf("\n");
    }
/****     Iterations Start    ****/
  iter=0;
/****     Find Son-Father Relation    ****/
  while (stop<4)
    {
     for (l=1;l<=maxl;l++)
      {
       m=pow(2,layer-l);
        for (i=0;i<m;i++)
      for (j=0;j<m;j++)
       nchild[i][j][l]=0;
      }
     iter+=1;
     for(l=0;l<=maxl-1;l++)
     {
      m=pow(2,layer-l);
      for(i=0;i<=m-1;i++)
      {
       for(j=0;j<=m-1;j++)
       {
       if (a[i][j][l]>0)
       {
      min=10.0;
      for(n=0;n<=3;n++)
      {
       row=fx[i][j][l][n];
       col=fy[i][j][l][n];
       if (a[row][col][l+1]>0)
       {
       d[n]=c[i][j][l]-c[row][col][l+1];
       if(d[n]<0){d[n]=c[row][col][l+1]-c[i][j][l];}
       if(d[n]<min){min=d[n];p[i][j][l]=n;}
       }
      }
      n=p[i][j][l];
      row=fx[i][j][l][n];
      col=fy[i][j][l][n];
      nc=nchild[row][col][l+1];
      sx[row][col][l+1][nc]=i;
      sy[row][col][l+1][nc]=j;
      nchild[row][col][l+1]+=1;
```

```
      printf("%d %d  ",row,col);
       }
      }
        printf("\n");
      }
    }
/****   Computation of a and c     ****/
   for(l=1;l<=maxl;l++)
   {
     m=pow(2,layer-l);
     for(i=0;i<=m-1;i++)
     {
      for(j=0;j<=m-1;j++)
      {
     sum=0.0;
     psum=0.0;
     nc=nchild[i][j][l];
     if(nc>0)
     { for(n=0;n<=nc-1;n++)
       {
         row=sx[i][j][l][n];
         col=sy[i][j][l][n];
         sum+=(float)a[row][col][l-1];
         psum+=(float)a[row][col][l-1]*c[row][col][l-1];
       }
       a[i][j][l]=(int)sum;
/**    printf("%d a[i][j][l]",a[i][j][l]);**/
       if(sum>0) c[i][j][l]=psum/sum;
     }
       else {a[i][j][l]=0;c[i][j][l]=c[2*i][2*j][l-1];}
     printf("%6.4f ",c[i][j][l]);
     }
     printf("\n");
       }
   }
/****    Segmentation Value    ****/
   for(i=0;i<=1;i++)
    for(j=0;j<=1;j++)
    {
      min=s[i][j][maxl]-c[i][j][maxl];
      if (min<0) min=c[i][j][maxl]-s[i][j][maxl];
      min=100000.0*min;
      if(min>1.0) stop=0;
      else stop+=1;
      s[i][j][maxl]=c[i][j][maxl];
      printf("%6.3f ",c[i][j][maxl]);
    }
    printf("\n");
   for(l=maxl;l>=1;l--)
   {
    m=pow(2,layer-l);
    for(i=0;i<=m-1;i++)
     for(j=0;j<=m-1;j++)
      {
```

```
      nc=nchild[i][j][l];
      if(nc>0){for(n=0;n<=nc-1;n++)
          {
           row=sx[i][j][l][n];
           col=sy[i][j][l][n];
           s[row][col][l-1]=s[i][j][l];
          }
      }
       }
     }
  }
  printf("iter %d\n",iter);
  for(i=0;i<=k-1;i++)
  {
   for(j=0;j<=k-1;j++)
   printf("%5.3f\t",s[i][j][0]);
   printf("\n");
  }
}


void Cand_Father_()
{
  int ta1,ta2,tb1,tb2;
  int maxf,t,max,i,j;
  tb2=0;
  for(l=0;l<=maxl-1;l++)
  {
   maxf=pow(2,layer-l)-1;
   printf("layer[%d]\n",l);
   for(i=0;i<=maxf;i++)
    for(j=0;j<=maxf;j++)
    {
         ta1=(i-1)%2;
         if(ta1<0) ta1=(maxf-1)/2;
         else ta1=(i-1)/2;
         ta2=(i+1)/2;
         if(ta2>(maxf-1)/2) ta2=0;
         tb1=(j-1)%2;
         if(tb1<0) tb1=(maxf-1)/2;
         else tb1=(j-1)/2;
         tb2=(j+1)/2;
         if(tb2>(maxf-1)/2) tb2=0;
         fx[i][j][l][0]=ta1;
         fy[i][j][l][0]=tb1;
         fx[i][j][l][1]=ta1;
         fy[i][j][l][1]=tb2;
         fx[i][j][l][2]=ta2;
         fy[i][j][l][2]=tb1;
         fx[i][j][l][3]=ta2;
         fy[i][j][l][3]=tb2;
```

```
          printf("%d %d %d %d
%d\n",l,fx[i][j][l][0],fx[i][j][l][2],fy[i][j][l][0],fy[i][j
][l][1]);
        }
        }
     }


  void Cand_Son_()
  {
   int ta1,ta2,ta3,ta4;
   int tb1,tb2,tb3,tb4,max;
   int i,j;
    for(l=maxl;l>=1;l--)
    {
/**       printf("layer[%d]\n",l);**/
      max=pow(2,layer-1);
      for(i=0;i<=max-1;i++)
       for(j=0;j<=max-1;j++)
       {
      m=pow(2,layer-l+1);
      ta1=2*i-1;
      if(ta1<0) ta1=m-1;
      ta2=2*i;
      ta3=2*i+1;
      ta4=2*i+2;
      if(ta4>=m) ta4=0;
      tb1=2*j-1;
      if(tb1<0) tb1=m-1;
      tb2=2*j;
      tb3=2*j+1;
      tb4=2*j+2;
      if(tb4>=m) tb4=0;
/**   printf("%d %d %d %d  ",ta1,ta2,ta3,ta4);
      printf("%d %d %d %d \n",tb1,tb2,tb3,tb4);**/
      sx[i][j][l][0]=ta1;
      sy[i][j][l][0]=tb1;
      sx[i][j][l][1]=ta1;
      sy[i][j][l][1]=tb2;
      sx[i][j][l][2]=ta1;
      sy[i][j][l][2]=tb3;
      sx[i][j][l][3]=ta1;
      sy[i][j][l][3]=tb4;
      sx[i][j][l][4]=ta2;
      sy[i][j][l][4]=tb1;
      sx[i][j][l][5]=ta2;
      sy[i][j][l][5]=tb2;
      sx[i][j][l][6]=ta2;
      sy[i][j][l][6]=tb3;
      sx[i][j][l][7]=ta2;
      sy[i][j][l][7]=tb4;
      sx[i][j][l][8]=ta3;
      sy[i][j][l][8]=tb1;
      sx[i][j][l][9]=ta3;
```

```
      sy[i][j][l][9]=tb2;
      sx[i][j][l][10]=ta3;
      sy[i][j][l][10]=tb3;
      sx[i][j][l][11]=ta3;
      sy[i][j][l][11]=tb4;
      sx[i][j][l][12]=ta4;
      sy[i][j][l][12]=tb1;
      sx[i][j][l][13]=ta4;
      sy[i][j][l][13]=tb2;
      sx[i][j][l][14]=ta4;
      sy[i][j][l][14]=tb3;
      sx[i][j][l][15]=ta4;
      sy[i][j][l][15]=tb4;
       }
    }
}
```

```
    /*****************************************/
    /***   Simulation of Stout's Algorithm ***/
    /***         Multilevel Pyramid        ***/
    /*****************************************/

#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#define n 2 /** # of topmost level **/

int i,j,k,l,d,colm,row;
int max,shu,shl,x,y,Texe,Tload,temp[10][10];
int Tset,Tadd,Tmul,Omax,W[10][10],inc;
int Tcomm,wsize,a,dx,dy,m;
float Uavg,Umax;
int next1,next2;
int C[10][10],P[10][10],conv[10][10];
long int Oep;
int Base[n];

void Product_Wc_();
void Sum_();
void Shift_R_();
void Shift_D_();
void Shift_U_();
void Shift_L_();
void Dec_Gray_();
void Dist_();

main()
 {
   Tload=2;
   Tcomm=2;
   Tset=1;
   Tadd=1;
   Tmul=2;
   Oep=0;
   Omax=0;
   Texe=0;
   printf("Please input wsize");
   scanf("%d",&wsize);
   Base[n]=0;
   Base[0]=7;
   printf("Please input reduction ");
   scanf("%d",&a);
   if (a==16) Base[1]=1;
   else Base[1]=3;
   printf("Please input window coefficient\n");
   for(i=0;i<=Base[0];i++)
    for(j=0;j<=Base[0];j++)
    {
     scanf("%d",&a);
     Dec_Gray_();
```

```
   W[x][y]=a;
   temp[x][y]=W[x][y];
   }
printf("Please input image matrix\n");
for(i=0;i<=Base[0];i++)
 for(j=0;j<=Base[0];j++)
 {
  scanf("%d",&a);
  Dec_Gray_();
  C[x][y]=a;
 }
shu=0;
for(row=0;row<=Base[0];row++)
{
  shl=0;
  for(colm=0;colm<=Base[0]+1;colm++)
  {
 if(colm==Base[0]+1){ Shift_R_();
             Texe=Texe-2*Tset-Tcomm;
             Oep=Oep-2*pow(k,2);
           }
 else{ if((row==0)&&(colm==0)){  Product_Wc_();
                      for(l=1;l<=wsize;l++)
                      Sum_();
                      conv[0][0]=P[0][0];
                      }
       else{ if((row!=0)&&(colm==0)){ Shift_D_();
                          shu=shu+1;
                        }
          else{ Shift_R_();
             shl=shl+1;
             if((row==Base[0])&&(colm==Base[0]))
             { Omax=2*pow(k,2)+Omax;}
           }
         Product_Wc_();
         if((row==Base[0])&&(colm==Base[0]))
         { Omax=Omax+pow(k,2);}
         if(shu!=0)
        {
         for(m=1;m<=shu;m++)
         {
          Shift_U_();
          if((row==Base[0])&&(colm==Base[0]))
          { Omax=2*pow(k,2)+Omax;}
         }
        }
          if(shl!=0)
          {
           for(m=1;m<=shl;m++)
           {
             Shift_L_();
             if((row==Base[0])&&(colm==Base[0]))
             { Omax=Omax+2*pow(k,2);}
           }
```

```
                }
      for(l=1;l<=wsize;l++)
      { Sum_();
       if((row==Base[0])&&(colm==Base[0]))
Omax=Omax+11*pow(max,2);
      }
       i=row;
       j=colm;
       Dec_Gray_();
       conv[x][y]=P[0][0];
       Dist_();
       Texe=Texe+4*(dx+dy);
       Oep=Oep+2*(dx+dy);
       if((row==Base[0])&&(colm==Base[0]))
       { Omax=Omax+(dx+dy)*2;}
       }
     }
    }
   }
    Uavg=(float)Oep/(Texe*pow(k,2));
    Umax=(float)Omax/Texe;
/**printf("\n %d\t %5.4f\t %5.4f\t%d\t%d\t
%ld\n",Texe,Uavg,Umax,k,k,Oep);**/
 for (i=0;i<=Base[0];i++)
  {
   for (j=0;j<=Base[0];j++)
    {
     Dec_Gray_();
     printf("%d  ",conv[x][y]);
     }
    printf("\n");
   }
}



void Product_Wc_()
 {
   for(i=0;i<=Base[0];i++)
    for(j=0;j<=Base[0];j++)
     {
      Dec_Gray_();
      P[x][y]=temp[x][y]*C[x][y];
     }
    Texe=Texe+Tmul;
    Oep=Oep+pow(k,2);
  }


  void Sum_()
  {
     int count,k;
     max=Base[l];
```

```
    k=1;
    if (Base[l-1]-Base[l]>2) count=2;
    else count=1;
    while (count>0)
    {
     inc=pow(2,k);
     d=pow(2,k-1);
     for(x=0;x<=Base[0];x=x+inc)
      for(y=0;y<=Base[0];y=y+inc)
      {
     P[x][y]=P[x][y]+P[x][y+d];
     P[x][y+d]=P[x+d][y+d];
     P[x][y]=P[x][y]+P[x+d][y];
     P[x][y]=P[x][y]+P[x][y+d];
    }
   Texe=Texe+9;
   Oep=Oep+11*max*max;
   }
}


void Shift_R_()
{
 for(i=0;i<=Base[0];i++)
  for(j=0;j<=Base[0];j++)
   {
   Dec_Gray_();
   if(y==0){ next1=temp[x][0];
          temp[x][0]=temp[x][k/2];
        }
   else{ next2=temp[x][y];
      temp[x][y]=next1;
      next1=next2;
    }
   }
  Texe=Texe+2*Tset+Tcomm;
  Oep=Oep+2*pow(k,2);
}


 void  Shift_D_()
{
  for(j=0;j<=Base[0];j++)
   for(i=0;i<=Base[0];i++)
    {
     Dec_Gray_();
     if(x==0){ next1=temp[0][y];
         temp[0][y]=temp[k/2][y];
          }
     else{ next2=temp[x][y];
        temp[x][y]=next1;
        next1=next2;
      }
     }
```

```
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*pow(k,2);
}


void Shift_U_()
{
   for(j=Base[0];j>=0;j--)
   for(i=Base[0];i>=0;i--)
   {
    Dec_Gray_();
    if(x==k/2)  { next1=P[k/2][y];
                  P[k/2][y]=P[0][y];
                }
    else{ next2=P[x][y];
       P[x][y]=next1;
       next1=next2;
     }
   }
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*k*k;
}



void Shift_L_()
{
 for(i=Base[0];i>=0;i--)
    for(j=Base[0];j>=0;j--)
    {
     Dec_Gray_();
     if(y==k/2){ next1=P[x][k/2];
                 P[x][k/2]=P[x][0];
               }
     else{ next2=P[x][y];
        P[x][y]=next1;
        next1=next2;
       }
    }
 Texe=Texe+2*Tset+Tcomm;
 Oep=Oep+2*pow(k,2);
}


void Dec_Gray_()
{
 x=i>>1;
 x^=i;
 y=j>>1;
 y^=j;
}

void Dist_()
{
```

```
 int d1,d2;
 d1=x;
 d2=y;
 dx=0;
 dy=0;
 while (d1>0){  if((d1&1)>0) dx=dx+1;
          d1=d1>>1;
         }
 while(d2>0){ if((d2&1)>0) dy=dy+1;
        d2=d2>>1;
      }
}
```

```
/*****************************************/
/***   Simulation of Ziavras' Algorithm ***/
/***        Multilevel Pyramid           ***/
/*****************************************/


#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#define n 2 /**# of the topmost level **/
int i,j,k,l,d,colm,row;
int max,shu,shl,temp[10][10];
int Tset,Tadd,Tmul,Omax,W[10][10],inc;
int Texe,Tload,Tcomm,wsize,a,m;
float Uavg,Umax;
int next1,next2;
int C[10][10],P[10][10],conv[10][10];
long int Oep;
int hostx,hosty,x,y,dx,dy;
int x1,y1,Base[n];

void Product_Wc_();
void Sum_();
void Shift_R_();
void Shift_D_();
void Shift_U_();
void Shift_L_();
void Dec_Gray_();
void Dist_();
void Host_();

main()
 {
   Tload=2;
   Tcomm=2;
   Tset=1;
   Tadd=1;
   Tmul=2;
   Oep=0;
   Omax=0;
   Texe=0;
   printf("Please input wsize");
   scanf(" %d",&wsize);
   Base[0]=pow(2,wsize)-1;
   k=Base[0];
   Base[n]=0;
   printf("Please input reduction");
   scanf(" %d",&a);
   if (a==16) Base[1]=1;
   else Base[1]=3;
   printf("Please input window coefficient\n");
   for(i=0;i<=Base[0];i++)
    for(j=0;j<=Base[0];j++)
```

```
 {
  scanf("%d",&a);
  Dec_Gray_();
  W[x][y]=a;
  temp[x][y]=W[x][y];
  }
printf("Please input image matrix\n");
for(i=0;i<=Base[0];i++)
 for(j=0;j<=Base[0];j++)
 {
  scanf("%d",&a);
  Dec_Gray_();
  C[x][y]=a;
 }
shu=0;
for(row=0;row<=Base[0];row++)
{
  shl=0;
  for(colm=0;colm<=Base[0]+1;colm++)
  {
 if(colm==Base[0]+1){ Shift_R_();
             Texe=Texe-2*Tset-Tcomm;
             Oep=Oep-2*pow(k,2);
            }
 else{ if((row==0)&&(colm==0)){  Product_Wc_();
                      x1=0;
                      y1=0;
                      for(l=1;l<=wsize;l++)
                      Sum_();
                      goto point1;
                      }
       else{ if((row!=0)&&(colm==0)){ Shift_D_();
                          shu=shu+1;
                         }
          else{ Shift_R_();
             shl=shl+1;
             if((row==Base[0])&&(colm==Base[0]))
             { Omax=2*pow(k,2)+Omax;
             }
           }
         Product_Wc_();
         if((row==Base[0])&&(colm==Base[0]))
         { Omax=Omax+pow(k,2);
         }
         if(shu!=0)
        {
         for(m=1;m<=shu;m++)
         {
          Shift_U_();
          if((row==Base[0])&&(colm==Base[0]))
           Omax=2*pow(k,2)+Omax;
         }
        }
          if(shl!=0)
```

```
                {
                 for(m=1;m<=shl;m++)
                 {
                   Shift_L_();
                   if((row==Base[0])&&(colm==Base[0]))
                    Omax=Omax+2*pow(k,2);
                 }
                }
     x1=0;
     y1=0;
     for(l=1;l<=wsize;l++)
     { Sum_();
      if((row==Base[0])&&(colm==Base[0]))
       { if (l==1) Omax=Omax+11*pow(max,2);
         else  Omax=Omax+20*pow(max,2);
        }
     }
      point1:
      i=row;
      j=colm;
      Dec_Gray_();
      conv[x][y]=P[x1][y1];
      Dist_();
      Texe=Texe+4*(dx+dy);
      Oep=Oep+2*(dx+dy);
      if((row==Base[0])&&(colm==Base[0]))
      { Omax=Omax+(dx+dy)*2;}
      }
    }
   }
  }
   Uavg=(float)Oep/(Texe*pow(k,2));
   Umax=(float)Omax/Texe;
/**   printf("\n %d\t %5.4f\t %5.4f\t%d\t%d\t
%ld\n",Texe,Uavg,Umax,k,k,Oep);**/
 for (i=0;i<=Base[0];i++)
 {
  for (j=0;j<=Base[0];j++)
  {
   Dec_Gray_();
   printf("%d  ",conv[x][y]);
   }
   printf("\n");
  }
}



void Product_Wc_()
 {
   for(i=0;i<=Base[0];i++)
    for(j=0;j<=Base[0];j++)
     {
```

```
     Dec_Gray_();
     P[x][y]=temp[x][y]*C[x][y];
    }
   Texe=Texe+Tmul;
   Oep=Oep+pow(k,2);
}


void Sum_()
{
   int count,k;
   Host_();
   max=Base[1];
   k=1;
   if (Base[l-1]-Base[l]>2) count=2;
   else count=1;
   while (count >0)
   {
    inc=pow(2,k);
    d=pow(2,k-1);
    for(x=x1;x<=Base[0];x=x+inc)
     for(y=y1;y<=Base[0];y=y+inc)
     {
   if (l<1)
    {
     P[x+hostx][y+hosty]=0;
     P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x][y];
     }
    P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x][y+d];
    P[x][y+d]=P[x+d][y+d];
    P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x+d][y];
    P[x+hostx][y+hosty]=P[x+hostx][y+hosty]+P[x][y+d];
   }
  x1+=hostx;
  y1+=hosty;
  k++;
  count--;
  if (l==1)
    { Texe=Texe+9;
     Oep=Oep+11*max*max;
     }
  else {Texe=Texe+13;
     Oep=Oep+20*max*max;
     }
  }
}


void Shift_R_()
{
 for(i=0;i<=Base[0];i++)
  for(j=0;j<=Base[0];j++)
   {
   Dec_Gray_();
```

```
    if(y==0){ next1=temp[x][0];
            temp[x][0]=temp[x][k/2];
          }
    else{ next2=temp[x][y];
       temp[x][y]=next1;
       next1=next2;
     }
    }
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*pow(k,2);
}


 void  Shift_D_()
{
  for(j=0;j<=Base[0];j++)
   for(i=0;i<=Base[0];i++)
    {
     Dec_Gray_();
     if(x==0)  next1=temp[0][y];
          [x]  [y]=temp[k/2][y];

     else{ next2=temp[x][y];
        temp[x][y]=next1;
        next1=next2;
      }
    }
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*pow(k,2);
}


 void Shift_U_()
 {
   for(j=Base[0];j>=0;j--)
   for(i=Base[0];i>=0;i--)
   {
    Dec_Gray_();
    if(x==k/2)   { next1=P[k/2][y];
              P[k/2][y]=P[0][y];
            }
    else{ next2=P[x][y];
       P[x][y]=next1;
       next1=next2;
     }
   }
   Texe=Texe+2*Tset+Tcomm;
   Oep=Oep+2*k*k;
 }


 void Shift_L_()
 {
```

```
  for(i=Base[0];i>=0;i--)
    for(j=Base[0];j>=0;j--)
    {
     Dec_Gray_();
     if(y==k/2){ next1=P[x][k/2];
             P[x][k/2]=P[x][0];
           }
     else{ next2=P[x][y];
        P[x][y]=next1;
        next1=next2;
       }
    }
  Texe=Texe+2*Tset+Tcomm;
  Oep=Oep+2*pow(k,2);
}


void Dec_Gray_()
{
 x=i>>1;
 x^=i;
 y=j>>1;
 y^=j;
}

void Dist_()
{
  int d1,d2;
  d1=x^hostx;
  d2=y^hosty;
  dx=0;
  dy=0;
  while (d1>0){  if((d1&1)>0) dx=dx+1;
           d1=d1>>1;
         }
  while(d2>0){ if((d2&1)>0) dy=dy+1;
         d2=d2>>1;
       }
 }

void Host_()
{
  if (l<=3) {i=0;j=l-1;}
  else if ((l==4)||(l==7)) {i=1;j=0;}
    else if (l==5) {i=0;j=-2;}
         else if (l==6) {i=0;j=-1;}
           else {i=0;j=1;}
  hostx=i;
  hosty=j;
}
```

BIBLIOGRAPHY

[1] Seitz, C.L. "The Cosmic Cube." Comm. ACM, vol. 28, No.1 ,Jan. 1985, pp. 22-33.

[2] Rosenfeld, A. "Multiresolution Image Proscessing and Analysis." Spring-Verlag, New York, N.Y. 1984.

[3] Burt, P.J., T.H. Hong, and A. Rosenfeld. "Segmentation and Estimation of Image Region Properties Through Cooperative Hierarchical Computation." IEEE Transactions on System, Man, and Cybernatics, vol. SMC-11, No. 12, Dec. 1981.

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

[5] Lai, T.-H., and W. White. "Embedding Pyramids in Hypercubes." Tech. Rep., Dept. of Computer and Information Science, Ohio State Univ., Nov. 1987.

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

[7] Patel, S.C., and S.G. Ziavras. "Comparative Analysis of Techniques That Map Hierarchical Structures into Hypercubes." in Proc. Parallel Distributed Computing Systems Conf., Washington, D.C., Oct. 1991, pp. 295-299.

[8] Ziavras, S.G., "On The Problem of Expanding Hypercube-Based Systems." Journal Parallel Distributed Computing, vol. 16, Sep. 1992, pp. 41-53.

[9] Ziavras, S.G., "Techniques for Mapping Deterministic Algorithms onto MultiLevel Systems." Proceedings of International Conference on Parallel Processing, vol. I, Chicago, I.L., Aug. 1990, pp. 228-233.

[10] Clermont, P., and A. Merigot. " Real Time Synchronization in a Multi-SIMD Massively Parallel Machine," in Proc. Architectures for Pattern Analysis Machine Intell. Conf., 1987, pp. 131-136.

[11] Hwang, K., and F.A. Briggs. "Computer Architecture and Parallel Processing," Mcgraw Hill Publication. 1984.

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

[13] 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.

[14] Ho, C.T., and S.L. Johnsson. "On the Embedding of Arbitary Meshesin Boolean Cubes with Expansion Two Dilation Two," in Proc. Intern. Conf. Parallel Processing Chieago, IL, August 1987, pp. 188-191.

[15] Johnsson, S.L., "Dilation & Embedding of a Hyper-Pyramid into a Hypercube, "Comm. ACM, 1989, pp. 294-303.