1-31-1993

Translation of images on the hypercube using leaf codes

Bhavesh Patel
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

Translation of Images on the Hypercube Using Leaf Codes

by

Bhavesh Patel

Image processing is used for manipulation of pictorial images. Image analysis applications are typically characterized by the need to process large quantities of image data. Some of the important transformations or operations which are carried out by image processing systems are translation, scaling, superposition and rotation. Algorithms have been developed to carry out these transformations on image regions represented by quadtrees. Gargantini introduced an algorithm to translate an image region represented by a linear quadtree or leafcodes. A linear quadtree is a space efficient data structure used for storing digital images. Ziavras et.al. have proposed a modification of Gargantini's algorithm which makes it much more efficient. Ziavras's algorithm translates as many leaves as possible without splitting them. This thesis carries out a comparative analysis that involves these two algorithms. The comparison is based on results obtained from simulation of these algorithms for a hypercube parallel computing system. Simulation results are obtained for a single pixel and multiple pixels per processing element (PE) of a hypercube parallel computing system. In the case where multiple pixels are stored in each PE, a binary image of size $2^p \times 2^p$ is subdivided into quadrants of equal size and then stored in an n-dimensional hypercube. It is shown that Ziavras's algorithm performs much better than Gargantini's algorithm when p is larger than n. Gargantini's algorithm may perform better when a single pixel is assigned to each PE.

TRANSLATION OF IMAGES ON THE HYPERCUBE USING LEAF CODES

by
Bhavesh Patel

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

Translation of Images on the Hypercube Using Leaf Codes

Bhavesh Patel

Dr. Sotirios Ziavras, Thesis Adviser
Assistant Professor of Electrical and Computer Engineering,
New Jersey Institute of Technology

Dr. John Carpinelli, Committee Member
Assistant Professor of Electrical and Computer Engineering,
New Jersey Institute of Technology

Dr. Dennis Karvelas, Committee Member
Assistant Professor of Computer and Information Science,
New Jersey Institute of Technology

This thesis is dedicated to
my parents

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my advisor Prof. Sotirios Ziavras for his guidance and friendship and for always motivating me to reach higher standards. I would like to thank him also for leading me to areas I had never known before.

Special thanks are due to Professor Herman Estrin for providing help in technical documentation for thesis writing.

I also thank Professor John Carpinelli and Professor Dennis Karvelas for serving as members of the committee.

Finally, I would like to thank Deven Shah for his support and valuable suggestions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

Region representation is an important issue in image processing and computer graphics; consequently a number of representations are currently in use. Pixel-based regions derived from binary pictures are often represented in terms of square blocks or quadrants centered on a $(2^n \times 2^n)$ -array $(n \geq 1)$ called the *raster*. These blocks are generated by recursive subdivision of an initial quadrant into four $(2^g \times 2^g)$-squares, g=n–1, n–2,. . . ,0: in a binary picture, blocks that participate in the representation of a given object are colored *black*, and those which do not participate are colored *white*. Such a recursive process is known as the regular decomposition by quadrants and its related data structure as the *quadtree*.

1.1 Representation of Images by Leaf codes

A quadtree is a form of picture encoding which is compact and easily handled . A quadtree encoded image exploits two-dimensional coherence by recursively decomposing the image into square areas in a particular way and thereby reducing storage space requirements. The code is notionally a tree structure with the root corresponding to the whole image. Unless the image is homogeneous, it is subdivided into four quadrants, each represented in the tree by a node joined by a branch to the root. These nodes are *leaves* when the quadrants they represent are themselves homogeneous; otherwise, they carry further branches to nodes representing successively smaller subdivisions. The subdivision, and hence the tree growth stops when all the nodes are leaves.

Quadtrees may be implemented as pointer structures or as linear structures. Although a pointer structure may simplify any operation on the tree and speedup access to its leaves, in general the memory requirements are unacceptable. Gargantini introduced inquadtree called the **leafcode** which is an ordered sequence of

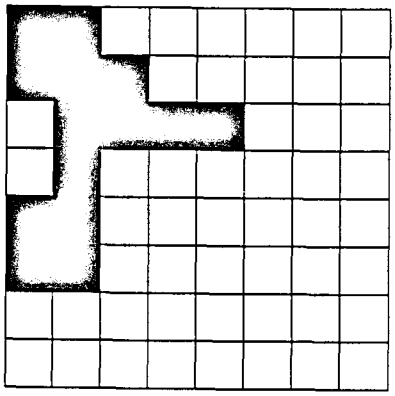

Figure 1 Image representation

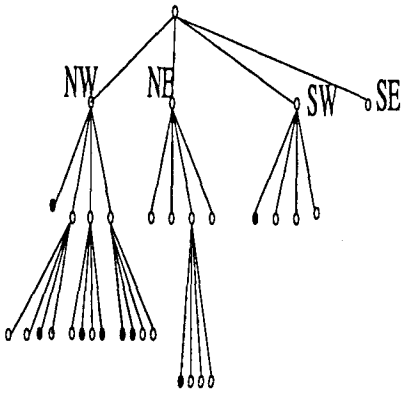

Figure 2 Quadtree representation

encoded black leaves [1,2]. The linear quadtree is a pointerless data structure which saves more than one-third of memory space used by regular quadtrees. The linear quadtree differs from the regular quadtree as follows:

- Only the leaves that contain information are stored.
- A unique encoding is used for each leaf which incorporates adjacency properties in the four principal directions.
- A region is represented as an ordered sequence of codes.

Each pixel is represented by a quaternary code, each digit of which corresponds to a subdivision of the previous quadrant according to the following scheme:

0 → for northwest (NW),

1 → for northeast (NE),

2 → for southwest (SW),

3 → for southeast (SE).

For instance, for $n = 3$ a block which belongs to the SE-quadrant in the first subdivision, to the SW-quadrant in the second subdivision, and to the NW-quadrant in the third subdivision is represented by 320. The left-to-right order of the three-digits reflects the larger-to-smaller subdivision order. Thus, each black pixel is encoded with digits 0, 1, 2, and 3 in base 4. Consider the image region shown in figure , which consists of $N_p = 20$ black pixels and 3–digit quaternary codes (N = 3). The quaternary codes after condensation are: 0XX, 20X, 100,and 102. Figure shows a image region mapped to quadrant with resolution 2.

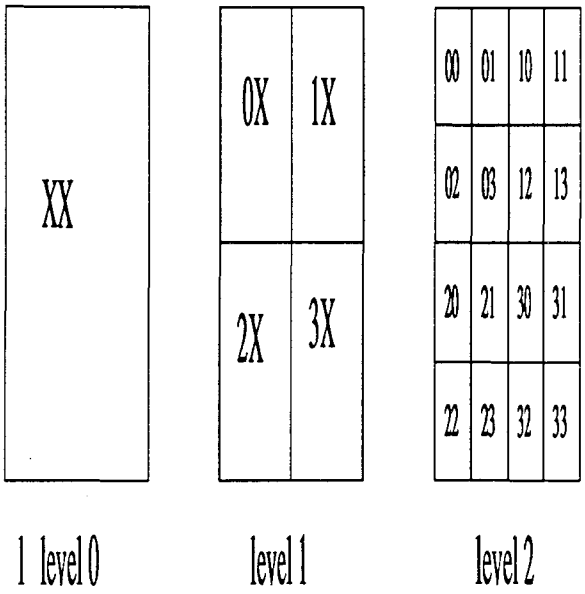

Figure 3 Image quadrant with resolution 2

We shall consider a $(2^n \times 2^n)$-image for the purpose of discussing the encoding used for the black pixels in an image region. Let M and N represent the row and column numbers for the black pixel under consideration, respectively. The encoding procedure consists of mapping the position (M,N) of the black pixel into its corresponding weighted quartenary code (O), which represents successive partitioning of the quadrants. Let the binary representation of M and N be

$$\mathrm{M} = x_{n-1} \ldots x_1 x_0$$

$$\mathrm{N} = y_{n-1} \ldots y_1 y_0$$

The corresponding quaternary code representation is

$$\mathrm{O} = z_{n-1} \ldots z_1 z_0$$

with

$$z_i = d_i + 2c_i \qquad i = 0, 1, \ldots, n-1. \tag{1}$$

Equation 1 suggests a faster way to encode pixels into its quaternary code and vice-versa. The quaternary code is obtained by expressing M and N in its binary form, by multiplying independently every binary digit in M by 2 (to the base four) and later by adding the resultant product to binary representation of N to the base four. The above operation for computing the quaternary code has the advantage of having no carry propogation during addition operation. For example, for $\mathrm{M} = 3 = (0011)_2$ and $\mathrm{N} = 8 = (1000)_2$, O is given by

$O = (N + 2M_4)_4 = (1000 + 0022)_4 = 1022.$

This method of encoding produces quaternary codes for which $z_i \leq 3$.

Various papers have been written on geometrical transformations of pictures that are encoded as pointer-structured quadtrees [4,5,6,7]. Since we are interested in space-efficient implementations of quadtrees, we concentrate on linear structures and look for efficient algorithms that perform geometrical operations on pictures represented by linear quadtrees. In the next section we briefly describe algorithms proposed by Gargantini and Ziavras for translation of linear quadtrees.

1.2 Gargantini Algorithm

The algorithm proposed by Gargantini operates as follows: To translate a pixel O (O is the quaternary code of the black pixel) by i rows, we first express its row and column binary representations by the following mapping rule:

$$0 \rightarrow (\ 0 \text{ or } 2); \quad 1 \rightarrow (1 \text{ or } 3).$$

Decoding is an inverse operation; given O, N is found as follows:

$$0 \rightarrow 0; \quad 1 \rightarrow 1$$
$$2 \rightarrow 0; \quad 3 \rightarrow 1$$

for all O-digits. For example, if O = 3211, N = 1011_2 = 11, Equation 1 gives

$$2\text{M} = (\text{O - N})_4$$

from which we can find M. In the above example,

$$2\text{M} = (3211 - 1011)_4 = 2200_4 = 1100_2 = 10$$

Next, we add the binary number i to the binary number M, multiply the obtained sum by 2 (to the base four), and sum this resultant product to N (to the base four). The translated pixel O', i.e. for $i = 3 = 0011_2$, is given by

$$O' = (\text{N} + (2 * (\text{M} + i)_2)_4)_4$$

Translation by j columns can be achieved in a similar way. The above procedure can be extended to compute translated quaternary code Q',using bith vertical and horizontal translations. In the Gargantini algorithm the time taken to translate a pixel is proportional to n. For the translation of regions, each individual pixel is first translated and then sorting and condensation of the resultant pixel codes is carried out [1]using heapsort to sort the quaternary codes requires $O(N_p \log_2 N_p)$ or $O(nN_p)$ time. Condensation takes $O(N_p)$ time, so the total time is $O(nN_p)$.

1.3 Ziavras Algorithm

In the Gargantini algorithm we carry out pixel-by-pixel translation of the image. Ziavras et. al. have proposed a modification of the Gargantini algorithm to enhance

its performance in terms of computing speed. The performance outweigh's the Gargantini algorithm when both the translation coordinates have even length. This algorithm translates blocks of multiple pixels simultaneously. Assume that a region represented by a linear quadtree is translated by

$$\text{rows} = x \times 2^a,$$

$$\text{cols} = y \times 2^b$$

pixels in the vertical and horizontal direction respectively, with a and b having the largest possible value. Let

$$\tau = \min(a, b) \tag{2}$$

then only blocks (i.e. quadrants) of sidelength greater than 2^τ need to be decomposed. Therefore, this algorithm translates as many leaves as possible without splitting them. Sorting and condensation are then applied to the resultant code [1]. Let N_s be the number of quaternary codes which are translated and N_i be the number of input leaf codes in the region represented by linear quadtree. The results show improvement in performance when the difference $N_i - N_s$ gets larger [3]. The expected degree of improvement over the Gargantini algorithm is $O(nN_s)$ for a uniprocessor.

1.4 Motivations and Objectives

The high computational requirement for transformations of images have resulted in the increased need for parallel computation. Several parallel architectures have been proposed to carry out the low-level and intermediate-level image processing tasks. Of these, the hypercube is of great interest as it is communication efficient. Several authors have proposed algorithms to carry out image transformations on parallel architectures. These algorithms have different behaviours and time complexities. For example Lee et al.developed parallel algorithms for image translation, rotation, and scaling [8]. Their algorithms are for a mesh connected multicomputer. Rosenfeld-pyramid algorithms for shrinking and expanding [9]. A common feature of most

algorithms is to assume that the number of processors exactly matches the number of pixels. In real life, this may not be true. The common case will be one in which the number of processors is less than the number of pixels.

Our objective is to carry out a comparative analysis of the algorithms proposed by Gargantini and Ziavras et al., for image translation. The motivation for this work was not to develop new algorithm but to evaluate the relative performance of these algorithms by simulating them for hypercube parallel computing systems. The analysis is presented for single pixel mapped per PE, as well as for multiple pixels mapped per PE.

CHAPTER 2

PARALLEL IMAGE PROCESSING

Image processing tasks are typically characterized by the need to process a large amount of data. The image computing tasks are generally time consuming in nature. A uniprocessor system might spend several minutes or hours dividing an image into a set of regions or classifying each pixel of the image. On the other hand, in multiprocessor systems each pixel can be processed in parallel, thereby, scaling down the processing time to a millisecond or less. This sort of parallelism is often considered to be *massive* and has been refered to as *image parallelism.*

2.1 Parallel Implementation of Image Processing Algorithms

Parallel image processing exploits the two fundamental modes of parallelism in image processing tasks: image parallelism and function parallelism. Image parallelism means that the same operation is repeated on each pixel or subregion throughout the image frame so that the image may be partitioned into subframes, which can be processed simultaneously by multiple PEs. On the other hand, function parallelism means that an image processing task (function) consists of several levels of processing. Here we divide an image processing function into subfunctions and use the pipelining approach. This method is useful when a sequence of images must be processed.

2.1.1 Problem Requirements

Given a problem for an image of size $N=(n\times\ n)$ pixels, which can be solved in an optimal sequential time of T(N) units, the problem will require $\Omega(T(N)/p)$ time on a parallel organization with $p \leq N$ processors. A parallel algorithm for a given problem is said to be *processor-time optimal* if the product of the number of processors

and the parallel execution time is equal to the sequential complexity of the problem. In parallel and distributed processing, efficiency of exploiting image parallelism is determined by communication overhead. A substantial amount of time is usually spent in routing messages among the processors. Therefore, efficient techniques should be developed for partitioning the image and moving data among the processors. Careful analysis of the problems is needed to derive such techniques.

Based on their communication requirements, image problems can be classified into two categories: local, and global

Local computation: The computation performed on a certain pixel p is a function of the pixels in a relatively small neighborhood of p. Examples include operations such as smoothing, deblurring, edge detection, texture analysis, and labeling of connected components.

Global computation: The computation performed on a certain pixel p is a function of other pixels at a relatively large distance from p. However, these pixels lie in predetermined (data-independent) locations within the image. Examples include image transforms such as the Fourier transform and the Walsh-Hadamard transform.

Another feature used to characterize image computation is image representation. Images naturally divide into subregions representing objects, shades and lines. Such regions can be represented by smaller amounts of data using border representation, run-length codes and quadtrees. An image represented in such a compressed form can be handled by a reduced number of processors.

2.2 Parallel Architectures for Image Computation

Image processing algorithms, from low-level to high-level, exhibit varying characteristics and demand different architectural features. While the low-level image processing tasks exhibit fine-grain parallelism at the pixel level, the high-level image processing tasks are associated with coarse-grain parallelism at the object or the segment levels.

The former tasks are traditionally known as SIMD algorithms; the latter ones fall into the MIMD category. Efficient concurrent processing relies upon the suitable design of data structures, decomposition of the problem for the allocation to processors, and choice of interconnection patterns. The advent of VLSI technology has enabled us to build parallel SIMD machines as well as MIMD machines to perform specific image-computing applications. Normally, MIMD machines fall into two categories: shared memory and distributed memory machines. Within these categories, multiprocessor systems are distinguished according to the interconnection pattern (tree, cube, mesh). In the following subsections we shall discuss a few parallel architectures and describe their advantages and shortcomings with respect to solving image processing problems.

2.2.1 Mesh Connected Computer

Mesh-connected computers (MCCs) have long been proposed for image processing. Images can be naturally mapped onto an MCC so that neighboring pixels are mapped onto neighboring (or the same) processing elements. The mesh connected computer (MCC) is a Single Instruction stream Multiple Data stream (SIMD) computer. It consists of n^2 processing elements arranged in a $n \times n$ lattice. The mesh computer has been used mainly for low-level *local* image processing. However, for global or distant computation, meshes do not perform well because communication across large distances is expensive and inefficient. To exploit efficiently the mesh computer the data size should match the processor size, which may be a severe limitation. Figure shows a MCC.

2.2.2 Pyramid Computer

The pyramid computer was initially proposed for performing high speed low-level and intermediate level image processing. The pyramid computer is a combination of

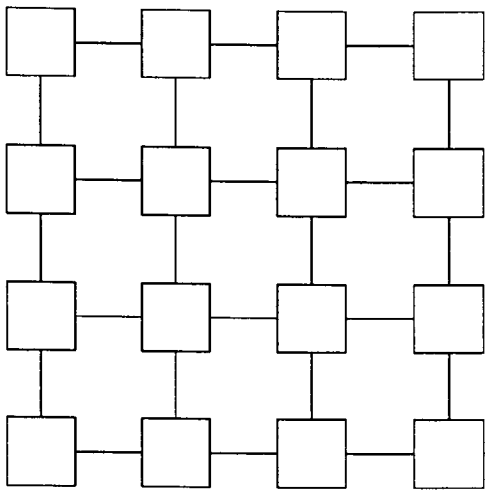

Figure 4 Mesh of size 4×4

the tree and mesh structures. A pyramid of the size n has $n^{1/2} \times n^{1/2}$ mesh-connected computer at its base and log_4(n) levels of mesh-connected computers above. A processing element at level k is connected to 4 siblings at level k, 4 children at level $k-1$, and a parent at level $k+1$ (see Fig 5). The levels are numbered so that the base is level 0 and the apex is level log_4(n). The pyramid computer architecture provides straightforward implementation of the divide-and-conquer strategy. However, pyramid processors are more difficult to build than meshes because of the complex arrangement of the communication links and requires almost twice the number of processing elements for the same image resolution. Also, the architecture is inflexible because of rigid interconnections. Therefore, a failure normally results in the failure of the entire system, or the performance degrades tremendously.

2.2.3 Hypercube Machine

Hypercube computers have gained popularity in a variety of scientific applications. The Caltech Cosmic Cube project [10] demonstrated many of the practical advantages of implementing a hypercube network. Having realized the potential of hypercube machines, the image-processing community is using them for a variety of low-level and high-level image processing applications. A hypercube of dimension d

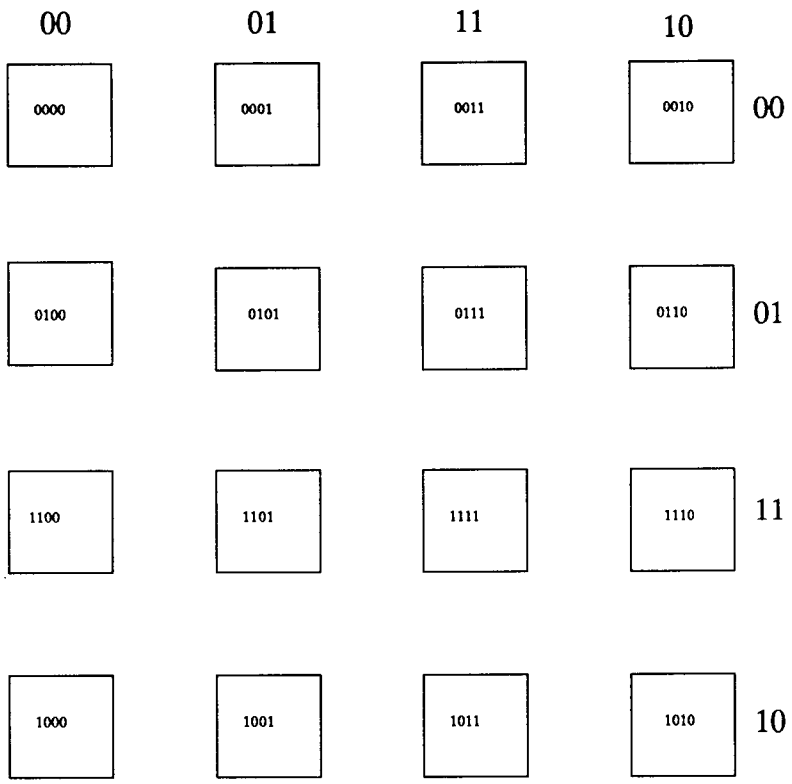

Figure 6 Two dimensional mesh view of a hypercube of dimension 4

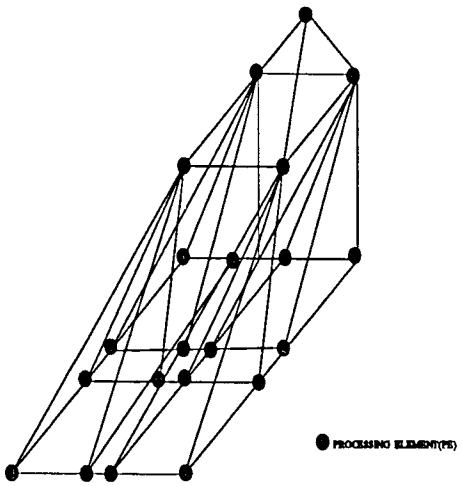

Figure 5 Pyramid(P_2)

($d \geq 0$), has 2^d nodes with unique d-bit binary string used as labels, where there is a link connecting two nodes if and only if their labels differ by a single bit. Several commercially available machines have been built that use hypercube topology. Both SIMD and MIMD types of machines have been built. Examples include, the Connection Machine [11], which is a bit-serial SIMD machine, and the NCUBE [12], which is a MIMD machine.

Hypercubes can efficiently carry out mesh calculations. Also, the hypercube machine supports efficient long distance communication that is absent in meshes or pyramids. Of most parallel architectures hypercubes have proved to be the most effective machines for research and development of scientific as well as vision applications. Several image computations require the processors of the hypercube to be arranged as a two-dimensional mesh. This can be done by assigning indices to the mesh nodes according to *reflected gray-code* numbering [13]. More specifically, the reflected gray code is used to encode the rows and columns of the mesh. The corresponding hypercube address is obtained by concatenating the encoded row and column numbers as shown in figure 6. The i-binary refected gray code S^k is defined recursively as follows

$$S^1 = 0, 1; S^k = 0S_{k-1}, 1[S^R_{k-1}],$$

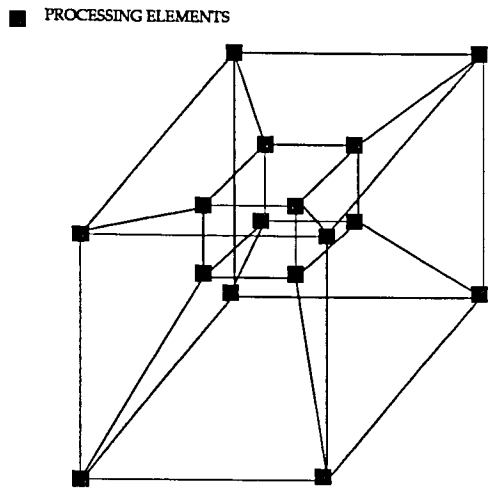

Figure 7 hypercube of *dimension 4* formed from two *3 - dimensional* hypercube

where $[S_{k-1}^R]$ is the reverse of the $k-1$-bit code S_{k-1}. Therefore, S^2 = 00, 01, 11, 10 and S^3 = 000, 001, 011, 010, 100, 101, 111, 110. This guarantees that pixels sharing an edge, or adjacent mesh entries, are in adjacent PEs. This also guarantees that a hypercube of size N can simulate any mesh algorithm for an image of size N.

To summarize, the salient features of a hypercube are:

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 called the *diameter* of the hypercube.
4. For two processors in the d - cube to communicate, data has to travel at least a distance which is equal to the number of 1s in the XOR result between the addresses of these PEs. This is same as the Hamming distance H(X,Y) between the two binary addresses.

CHAPTER 3

TRANSLATION OF IMAGE REGIONS REPRESENTED BY LEAF CODES ON HYPERCUBE MACHINES

An image can be represented using various data structures. A linear quadtree is one of them. In this thesis, we are interested in the translation of regions represented by linear quadtree on hypercube parallel computing system. Some parallel algorithms for computing geometric properties of digital images can be found in [14,15]. Before we present the implementation of the algorithms and their execution times, let us first discuss the data structures that will be employed by both algorithms. We define a structure *node* comprising of following important elements:

gcode: represents binary address of the PE.

pixel:[ ] is an array which is used to store the pixel value of the image region mapped to this PE.

quat:[ ] is an array which is used to store the quaternary code of each pixel i.e. black or white.

Let us use the notation *snode* for the source node, *imnode* for the intermediate node, and *dnode* for the destination node. During the translation of a pixel, if an *imnode* receives data from more than one *snode* which go to the same *next imnode*, then the data is stored in the structure *buffer*. Each node will have a FIFO output *buffer* to store the data. The *buffer* is necessary because each PE can transmit only to one PE in a unit time. The structure *buffer* has the following important elements:

adr:[ ] is an array to store addresses of the destination node.

foq: pointer to front of the buffer.

eoq: pointer to end of the buffer.

If the image size is $i \times i$ and the processor size is $p \times p$, then $sf = i/p$ is called the *scaling factor*. A block of size $sf \times sf$ is mapped to each PE. In the case where

multiple pixels are mapped per PE, the translation coordinates for the hypercube are scaled down as shown below:

$$scaledrow = \frac{row}{sp} \tag{3}$$

$$scaledcol = \frac{col}{sp} \tag{4}$$

where *scaled row* and *scaled col* are the quotients of the above operation.

3.1 Algorithm I (Gargantini)

In algorithm I, we carry out pixel-by-pixel translation of the image region. The algorithm I is as follows:

```
Begin
{translation coordinates: mrows, locls}
{time parameters:timetran, rtimetran, Total_time,
where, timetran: gives time to translate pixel information,
rtimetran: gives time to translate for remainder of row and column
divisions, "Total_time" gives total_time for translation of region}
{scaling factor: mc}
var tmrows, tlcols, mrowsrem, lcolsrem;
        tmrows = mrows/mc;
        tlcols = lcols/mc;
        mrowsrem = mrows % mc; { % is modulus operator}
        lcolsrem = lcols % mc;
        {Begin the mapping of the image on hypercube}
        Map_image();
        {Begin translation of pixel information}
        timetran = 0;
        timetran = timetran + cdt(tmrows,tlcols);
```

```
{Begin translation for remainder part translation coordinates}
if((mrowsrem != 0) || (lcolsrem != 0))
{
  rtimetran = rcdt(mrowsrem,lcolsrem);
}
Total_time = timetran + rtimetran;
End
```

where cdt() is a function which carries out parallel routing of pixel information from the source nodes to the destination nodes. The function cdt() is as follows:

```
Begin
{sr[ ]: stores binary address of the source nodes}
{dest[ ]: stores binary address of the destination nodes}
{hamdist[ ]: stores hamming distance value for each source node- destination node pair}
{stage: gives the number of stages for communication phase to complete}
{maxcount: starting count of PEs involved in the translation of pixel information}
var maxcount, ist, temp, maxdist, time_counter;
{Compute hamming distance for each source-destination pair}
maxdist = 0;
for(i = 0; i < maxcount; i++)
{
    temp = sr[i] d̂est[i];
    hamdist[i] = 0;
    hamdist[i] = ham_dist(temp);
    {hamdist(): function to compute hamming distance for each node}
    if(hamdist[i] > maxdist){
        maxdist = hamdist[i];
    }
    } stage = maxdist − 1;
```

```
for(ist = 0; ist <= stage; ist++)
{
        {Sort the buffer queue for each PE so that pixel
        going farthest is at the front of the queue}
        Sort_buffer_queue();
        {Communication involving parallel routing of data begins}
        Parallel_routing();
}
return time_counter;
End
```

In the routine cdt(), each source node may have to send the pixel information to same "*imnode*", in such situation, the pixel going to the farthest node is kept at the front of the buffer. The routine cdt() also returns the value of the time needed to carry out routing of data.

3.2 Algorithm II (Ziavras)

In algorithm II, we condense the pixel information as much as τ (Equation 2), and then carry out translation of pixel information. Algoirthm II is as follows:

```
Begin
{translation coordinates: mrows, locls}
{time parameters:timetran, rtimetran,Total_time,icondtime,
econdtime,edectime,idectime,
where, timetran gives time to translate pixel information,
rtimetran: gives time to translate for remainder of row and column
divisions, Total_time gives total_time for translation of region
icondtime gives time for internal condensation,
```

```
econdtime gives time for external condensation,
edectime gives time for external decomposition,
idectime gives time for internal decomposition}
{scaling factor: mc}
{processor_level: signifies level on which the parent node is one
with 'number of least significant zeros' equal to 'processor level'}
var tmrows, tlcols, mrowsrem, lcolsrem;
{Scale down the translation coordinates for multiple
pixel mapping}
        tmrows = mrows/mc;
        tlcols = lcols/mc;
        mrowsrem = mrows % mc; { % is modulus operator}
        lcolsrem = lcols % mc;
        {Map the image on hypercube}
        Map_image();
        {Compute v = MIN{p,t}, where 'v' is the condensation value
        used for internal and external condensation}
        v = 0;
        p = -1;
        t = -1;
        while((mrows == 0) || (lcols == 0))
        {
          if(mrows == 0)
          {
            do{
            t = t + 1;
            }while((lcols) % power(2,(t+1))) == 0);
```

```
    {power(x,y) computes x^y}
    v = t;
  }
  if(lcols == 0)
  {
    do {
    p = p + 1;_
    }while((mrows) % power(2,(p+1))) == 0);
    v = p;
  }
}
{Compute the sidelength of the pixel quadrant}
sidelength = 0;
tmc = mc;
do {
   sidelength++;
}while((tmc = tmc/2) != 1);
{Begin internal condnesation}
if(v) {
     icondtime = int_cindensation(v);
}
processor_level = 0;
if(v > sidelength){
    do {
       processor_levell++;
    }while((processor_level + sidelength) != v);
    econdtime = ext_condensation(processor_level, sidelength);
```

```
}
{Begin translation of condensed as well as non-condensed
pixel information within the image frame}
if((tmrows != 0) && (tlcols != 0)){
    timetran = cdt(tmrows, tlcols);
}
{Begin external decomposition of pixel information}
if(v){
  edectime = ext_decomposition(sidelength);
  {Begin internal decomposition of pixel information}
  idectime = int_decomposition(sidelength);
}
{Begin translation for remainder part translation coordinates}
if((mrowsrem != 0) || (lcolsrem != 0))
{
  rtimetran = rcdt(mrowsrem,lcolsrem);
}
Total_time = icondtime + econdtime + timetran
+ rtimetran + idectime +edectime;
End
```

We show below an example which gives the PEs spanned by pixel information during parallel routing of data. The image array size is 32×32 and the processor array size is 8×8.

```
00000011110000000000000000000000
00000011110000000000000000000000
00000011110000000000000000000000
00001111111111110000000000000000
```

```
00001111111111111000000000000000
00001111111111111000000000000000
11111111111111111110000000000000
11111111111111111110000000000000
11111111111111111000000000000000
11111111111111111000000000000000----->IMAGE
00001111111111111000000000000000
00001111111111111000000000000000
00001111111111111000000000000000
00001111111111111000000000000000
00001111111111111000000000000000
00001111111111111000000000000000
00000011110000000000000000000000
00000011110000000000000000000000
00000011110000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

```
00000000000000000000000000000000
------------------------------
    I     II    III   IV    V   ---->STAGES
------------------------------
    1     0     2     10    26
    2     3     7     15    31
    3     2     6     14    30
    8     9     11    3     19
    9     8     10    2     18
   10     11    15    7     23
   11     10    14    6     22
   14     15    13    5     21  ---->PEs SPANNED
   17     16    18    26    58
   18     19    23    31    63
   19     18    22    30    62
   24     25    27    19    51
   25     24    26    18    50
   26     27    31    23    55
   27     26    30    22    54
   49     48    50    58    42
   51     50    54    62    46
------------------------------
translation coordinates are:
 rows = 9 cols = 9
------------------------------
```

In case of *multiple* pixel mapping we perform *external* as well as *internal* condensation in situations where it is possible. The condensed codes are then translated

by the *scaled* translation coordinates (see equation 3 and equation 4). Later, the codes are decomposed *externally* as well as *internally.* The codes are then translated pixel by pixel if there is non-zero remainder in equation 3 and equation 4. In *internal condensation*, we check each subdivision to see if all the pixel values are *one*, i.e. we check if the pixel count is *four.* If the pixel count is *four*, i.e. all pixels have the same quaternary code representation except for the last digit, then, in that subdivision we make the pixel value of quaternary codes in NE, SW, and SE equal to *zero* and that in NW direction is made *one.* The last digit of NW-quaternary code is replaced by the marker X. The same procedure is performed for all the subdivisions.

External condensation is carried out only if the *block* mapped to each PE is *completely* condensed internally. The processor array is subdivided into quadrants similar to that used for images. In each subdivision we check if the *block count* is *four*, if true, the pixel value of the *child* PE's is made *zero* and that of *parent* PE made *one* as given in [12] for all the subdivisions. Similarly, we do it for *external* decomposition, in each subdivision, the parent node sends a pixel value of *one* to its childrens and the procedure is repeated for each subdivision. The time needed to carry out communication internal to PE, and external to PE is kept as a variable for performance analysis. Figure 8, 9, and 10 illustrate condensation, and translation of an image of size 8×8. We use the following notations for various time parameters:

T_{ic}: Time for internal condensation.

T_{ec}: Time for external condensation.

T_{id}: Time for internal decomposition.

T_{ed}: Time for external decomposition.

T_{rm}: Time for translation of image by remainder part of translation coordinates.

T_{tr}: Time for translation by the scaled translation coordinates.

T_{total}: Summmation of all the above mentioned time parameters.

Thus, for Ziavras Algorithm, we have,

$$T_{total} = T_{ic} + T_{ec} + T_{id} + T_{ed} + T_{tr} + T_{rm}$$

000	001	010	011	100			
002	003	012	013	102			
020	021	030	031				
022	023	032	033				
200	201						
202	203						

Figure 8 Image of size 8 × 8

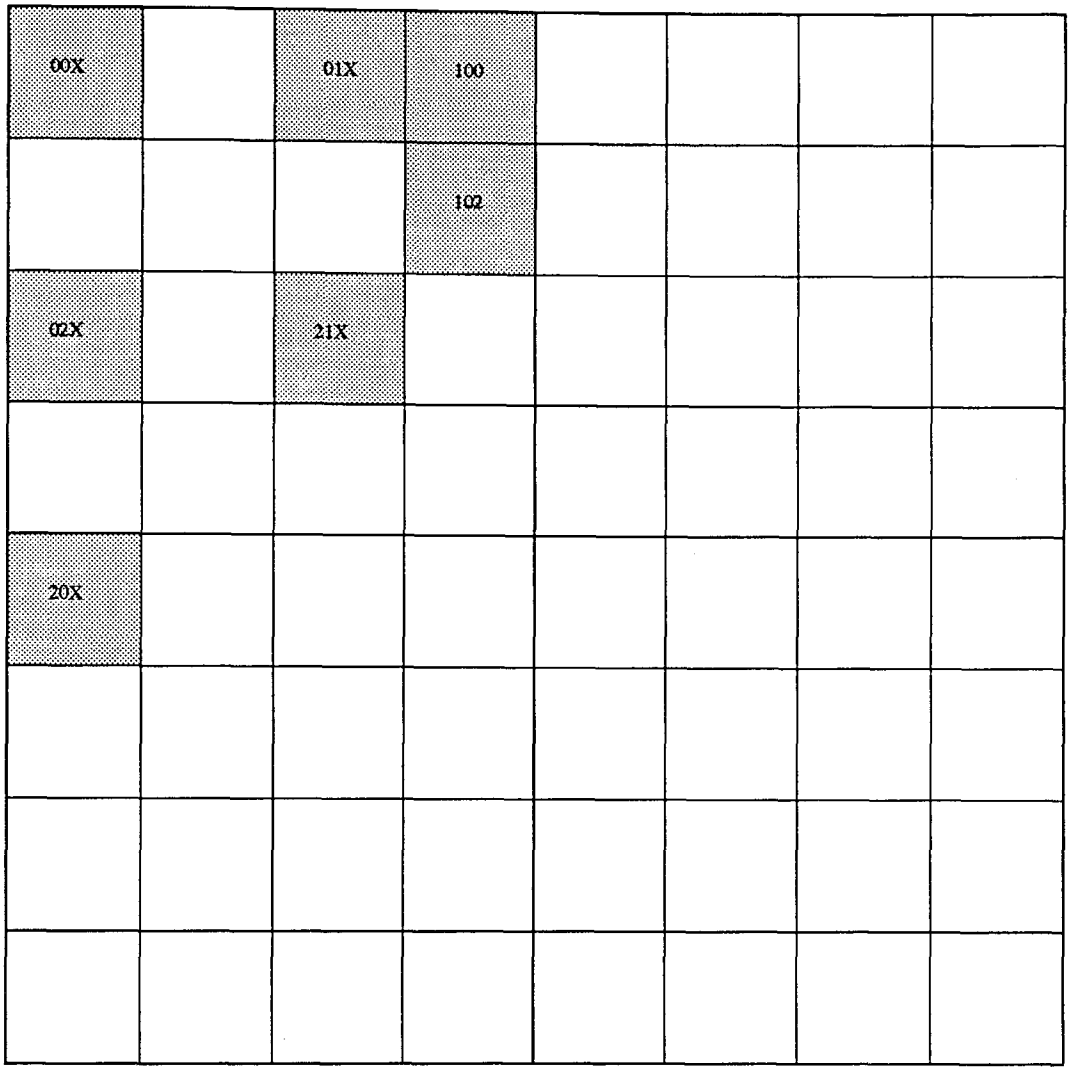

Figure 9 Same image after condensation

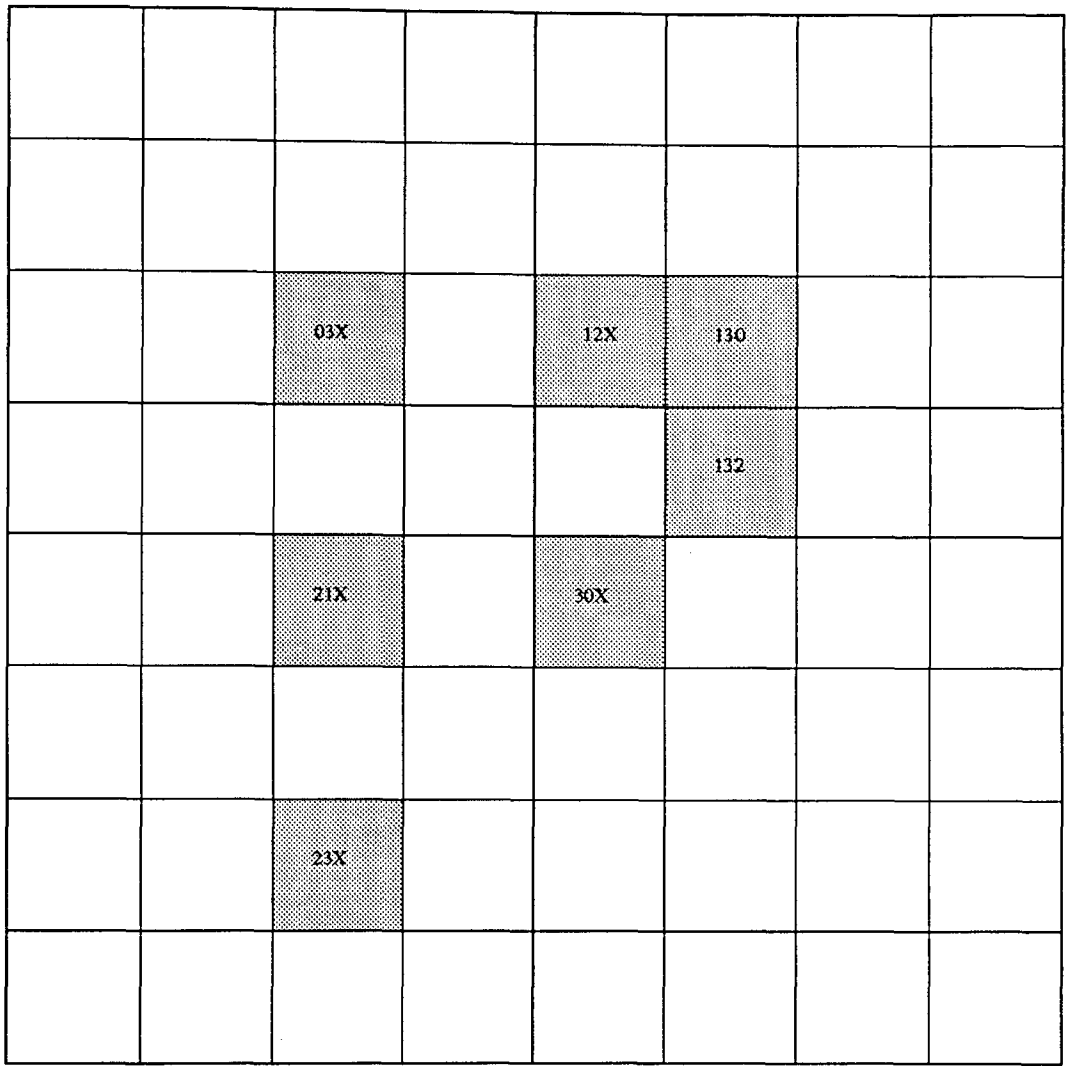

Figure 8 Same image after translation by (2,2)

CHAPTER 4
RESULTS AND DISCUSSIONS

This chapter carries out a comparative analysis of the algorithms discussed in the previous chapter. The Gargantini algorithm shows better performance when there is one-to-one mapping of the image onto the hypercube. Since there is neither condensation nor decomposition of pixel information, the performance of Gargantini algorithm goes on improving as the value of τ increases. The results of single-pixel mapping are tabulated in Table 4.1. The notation used in the table for the various parameters is as follows:

row: It specifies translation in the vertical direction.

col: It specifies translation in the horizontal direction.

Im-size: Image array size.

PE-size: Processor array size.

$\mathbf{N_{bp}}$: Number of input black pixels.

$\mathbf{N_{tc}}$: Number of translation codes.

$\mathbf{T_{tg}}$: Translation time for the Gargantini algorithm.

$\mathbf{T_{tz}}$: Translation time for the Ziavras algorithm.

Table 4.1 Execution times for single-pixel mapping

row	col	Im-size	PE-size	N_{bp}	N_{tc}	T_{tg}	T_{tz}
2	4	16	16	16	4	8	20
16	16	32	32	256	1	8	56
8	16	64	64	436	34	8	44
2	4	128	128	3880	970	8	20
16	32	128	128	3880	106	8	56
64	32	256	256	17936	116	8	68

The Ziavras algorithm shows improved performance when multiple pixels are mapped per PE. In this case, the image is partitioned into equal sized square blocks, where the number of blocks equals the number of processing nodes. The blocks themselves form a mesh, so they are embedded into hypercube nodes as was the case

for single pixel per PE mapping. Let us denote by sf the sidelength of the block mapped to each PE. If the image array size is im and the processor array size is pr, then, the sidelength sf is given by

$$sf = \frac{im}{pr}, \quad where \quad im > pr \tag{5}$$

We assume that each node can hold the entire block of pixels in its local memory. Each PE has an output FIFO buffer which holds the message to be transmitted . Message passing is implemented in parallel with each node allowed to transmit a single message in a unit time.

In the Ziavras algorithm the pixel information is condensed internally within the node as well as external to the node if possible. The condensed pixel information which represents the leaf codes are then translated to the destination nodes. At the destination, the condensed pixel information is decomposed both internally and externally if possible. In Gargantini algorithm, there is pixel by pixel translation of the image object. Table 4.2 shows the results for both algorithms when multiple pixels are mapped per PE.

Table 4.2 Execution time for Algorithm I and II

row	col	Im-size	PE-size	N_{bp}	N_{tc}	T_{tg}	T_{tz}
32	28	64	16	368	44	160	90
15	8	64	16	960	960	176	176
0	16	64	8	1024	256	256	142
12	26	64	8	1024	256	584	200
42	36	128	64	988	247	72	24
-22	-32	128	64	4597	1210	56	34
12	38	256	128	16384	4096	72	24
8	32	256	64	22500	378	128	58
8	16	256	64	22500	546	384	242

The results show that the performance of the Ziavras algorithm is better than Gargantini's algorithm for all these cases. The larger the values of sf ($sf > 2$), and τ, the better the performance of the Ziavras algorithm when compared to the Gargantini algorithm. The performance shows marked improvement when the number of

condensed translated codes get smaller. When one of the translation coordinates has odd length (i.e. $\tau = 0$), both algorithms have the same performance. In one of the cases (Im-size = 64 and PE-size = 8), there is an improvement in performance of the Ziavras algorithm when the translation coordinates are changed. This is due to the fact that the pixel information has to traverse larger internode distances.

Graph plots for various values of sf are shown in the appendix. Results for various values of sf follow.

$sf = 2$: The table 4.3 shows the results for an image array of size 256 × 256 being mapped onto a processor array of size 128 × 128. The black object for this case is chosen to occupy the upper left quadrant of size 128 × 128. The number of pixels mapped per PE are 4 for this example. From the graph we see that the execution time for the Ziavras algorithm increases by increasing the value of τ. This is because the black image region is highly regular in shape, i.e. it is a square. Due to this the Ziavras algorithm is able to carry out condensation (internally as well as externally) for higher levels where PE nodes are parents of nodes at levels below it. This condensation involves communication overhead, due to which the execution time for Ziavras algorithm is slightly higher compared to the Gargantini algorithm. In the case of the Gargantini algorithm, we carry out pixel by pixel translation. The execution time for the Gargantini algorithm is constant for this example, because the execution time is dependent on the number of intermediate nodes it spans to reach the destination node, which is the same for all values of τ.

$sf = 4$: For the case when $sf = 4$ the number of pixels mapped per PE node are 16. We find from the table that the execution time of the Ziavras algorithm is better compared to the Gargantini algorithm because more number of pixels are mapped per PE. Due to this, the Gargantini algorithm which carries out pixel by pixel translation takes comparatively more time.

$sf = 8$: For the case when $sf = 8$ the black image region mapped onto the processor

array is of irregular shape with the number of input black pixels being 416. The Ziavras algorithm shows significant improvement in its performance for values of τ shown in Table 4.5.

$sf = 16$ The number of pixels mapped per PE for $sf = 16$ is 256 and the number of input black pixels is 1950. The execution time of the Ziavras algorithm totally outweighs that of the Gargantini algorithm for all possible values of τ.

Table 4.3 Execution time for $sf = 2$

Im-size	PE-size	τ	T_{tg}	T_{tz}
256	128	1	32	12
256	128	2	32	24
256	128	3	32	32
256	128	4	32	44
256	128	5	32	56
256	128	6	32	74

Table 4.4 Execution time for $sf = 4$

Im-size	PE-size	τ	T_{tg}	T_{tz}
256	64	1	128	80
256	64	2	128	96
256	64	3	128	50
256	64	4	128	62
256	64	5	128	74
256	64	6	128	86
256	64	7	128	98

Table 4.5 Execution time for $sf = 8$

Im-size	PE-size	τ	T_{tg}	T_{tz}
64	8	0	352	70
64	8	1	368	72
64	8	2	384	95
64	8	3	512	174
64	8	4	512	174

Table 4.6 Execution time for $sf = 16$

Im-size	PE-size	τ	T_{tg}	T_{tz}
128	8	1	1792	265
128	8	2	1792	300
128	8	3	1792	404
128	8	4	1792	666
128	8	5	1792	718

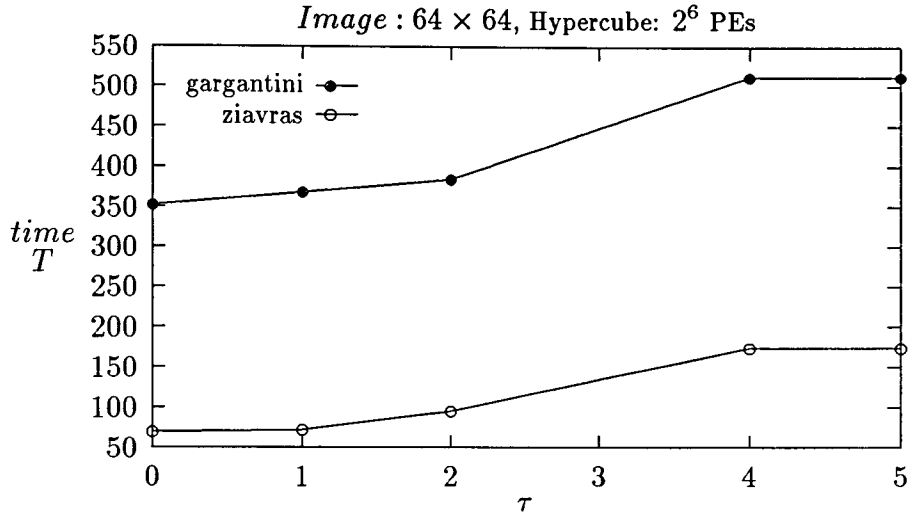

Figure 11 A graph for $sf = 8$

CHAPTER 5
CONCLUSIONS

This thesis carries out a comparative analysis of algorithms that translate an image on a hypercube parallel computing system. The analysis is carried out for two cases:

1. single-pixel mapping.

2. multiple-pixel mapping.

The time for translation of a pixel depends on the maximum number of processor nodes traversed by the pixel information. However, in the case of the Ziavras algorithm the pixel information is condensed if the translation coordinates are even. The execution time of the Ziavras algorithm may be more for single-pixel mapping as not only does it have to perform translation, but also has to perform condensation and decomposition. The time for condensation is also dependent upon the shape of the translated region. If the image is square in shape, then there is a likelihood of increase in time for condensation depending upon the value of the translation coordinates, i.e., the translation coordinates are even and either one of the coordinates is a multiple of two.

In the case of multiple-pixel mapping, the Ziavras algorithm performs better than the Gargantini algorithm. The number of pixels mapped per PE are sf^2, where sf is given by Equation 5 (page 28). Only a single pixel value can be transferred over the communication link in a unit time, as a result, the Ziavras algorithm shows better performance because there is *internal* as well as *external* condensation of pixel information before translation. As a result, the number of translation codes are less compared to the Gargantini algorithm in which the number of input black pixels equals the number of translated codes. The execution time for the Ziavras algorithm is far better for cases where more pixels are mapped per PE. Thus, if N_s is the number of codes which are translated by the modified algorithm and N_p is the total number of black pixels in the region represented by the leaves of the input quadtree,

then the larger the value of the difference $N_p - N_s$, the better the performance of the Ziavras algorithm compared to the Gargantini algorithm.

APPENDIX

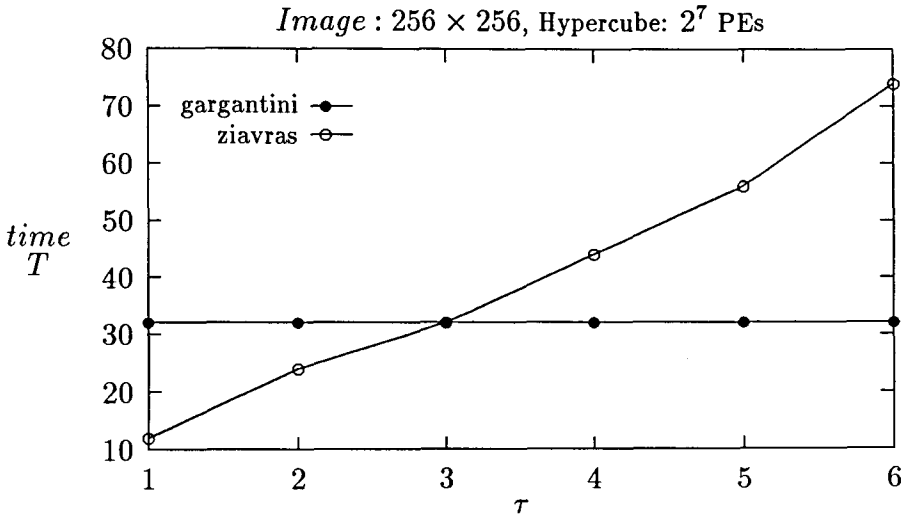

Figure 12 A graph for $sf = 2$

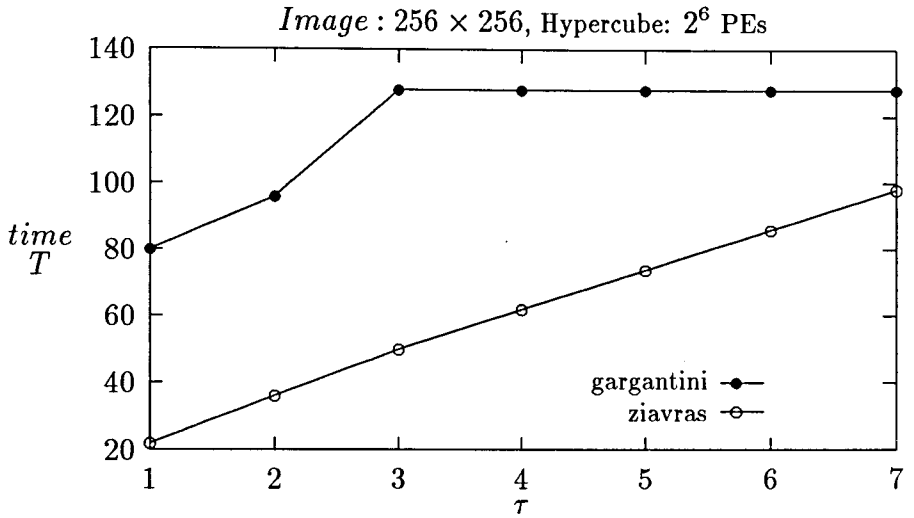

Figure 13 A graph for $sf = 4$

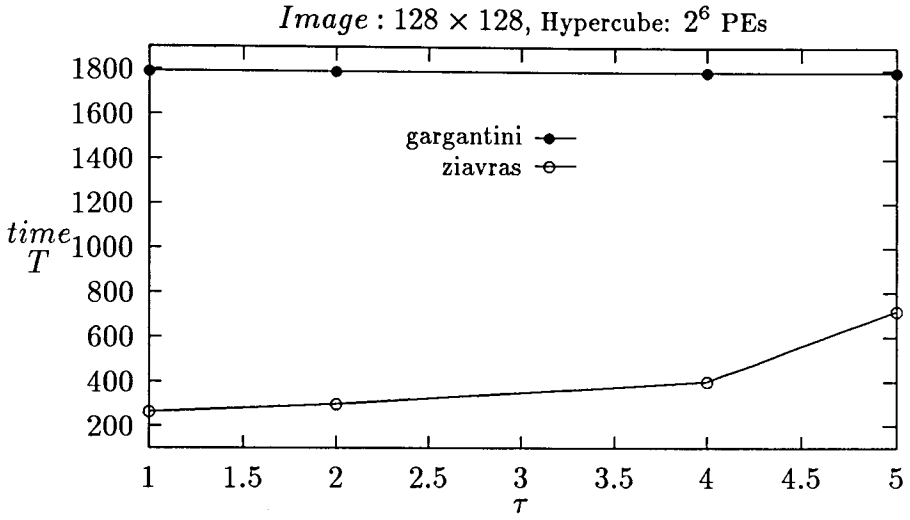

Figure 14 A graph for $sf = 16$

WORKS CITED

1. Gargantini, I. "An Effective Way to Represent Quadtrees." *Commun. ACM*,Vol 25,(1982): 905-910.

2. Gargantini, I. "Translation, Rotation, and Superposition of Linear Quadtrees." *Int J. Man Machine Studies*, Vol 18, (1983): 253-263.

3. Samet. H. "The Quadtree and its Related Heirerchical Data Structures." *Com--put. Surv.*, Vol 16, (1984): 260-274.

4. Hunter, G. M., and K. Steiglitz. "Linear Transformation of Pictures Represented by Quadtrees." *Comput. Graphics Image Process.*, Vol 10, (1979): 289-296.

5. Van Leirop, M. L. P. "Geometrical Transformations on Pictures Represented by Leaf Codes." *Comput. Vision Graphics Image Process.* Vol 33, (1986): 81-98.

6. Samet H., and M. Tamminen. "Computing Geometric Properties of Images Represented by Linear Quadtrees." *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-7*, (1985): 229-240.

7. Ziavras, S.G., and N.A. Alexandridis. "Improved Algorithms for Translation of Pictures Represented by Leaf Codes." *Image and Vision Computing*, Vol 6, no 1, (1988): 13-20.

8. Lee S.Y., Yalamanchali, and J.K. Agarwal. "Parallel Image Normalization on a Mesh Connected Array Processor." *Pattern Recognition*, Vol 20, no 1, (1987): 115-120.

9. Rosenfeld, A. "A Note on Shrinking and Expanding Operations on Pyramids." *Pattern Recognition Letters*, Vol 6, (1987): 241-244.

10. Seitz, C. "The Cosmic Cube." *Communication of ACM*, Vol 28, no 1, (1985): 22-33.

11. Hillis, D. "The Connection Machine." Cambridge, MIT Press, 1985.

12. Ncube Corp, "Promotional Literature." Beaverton, OR, 1985.

13. Stout, Q. F. "Hypercube and Pyramids." In Cantoni. Vi, and Levialdi, S., (Eds) *Pyramidal Systems for Computer vision*, Springer-Verlag, New York/Berlin, (1986): 75-89.

14. Miller R., and S.F Stout. "Geometric Algorithms for Digital Pictures on Mesh-Connected Computer." *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-7*,

(1985): 216-228.

15. Stout Q.F. "Supporting Divide-and-Conquer Algorithms for Image Processing" *J. Parallel Distrib. Comput.*, **4**, (1985): 95-115.

16. Ziavras S.G. "On the Problem of Expanding Hypercube-Based Systems." *J. Parallel Distrib. Comput.*, 16, (1992): 41-53.