

# A Quadtree Approach to Parallel Image Processing

by

Hany S. Saleeb

B.S. Electrical Engineering and Computer Science 1993  
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and  
Computer Science in partial fulfillment of the requirements for  
the degree of


Masters of Engineering

January 12, 1995

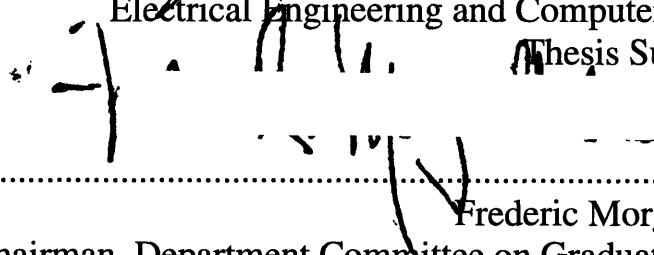
© Massachusetts Institute of Technology, 1995. All Rights Reserved.



Author .....  
Electrical Engineering and Computer Science  
January 12, 1995



Certified by .....  
Professor Steve Ward  
Electrical Engineering and Computer Science  
Thesis Supervisor



Accepted by .....  
Frederic Morgenthaler  
Chairman, Department Committee on Graduate Theses  
Electrical Engineering and Computer Science  
MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

AUG 10 1995

LIBRARIES

Barker Eng

# **A Quadtree Approach to Parallel Image Processing**

by

Hany S. Saleeb

Submitted to the Department of Electrical Engineering and Computer Science on January 12, 1995, in partial fulfillment of the requirements for the degree of the Masters of Engineering.

## **Abstract**

This thesis work is intended through a C simulation to show that color image processing using quadtrees is a task suited to parallel implementation. Also, it explores the strengths and shortcomings of a mesh prototype system like NuMesh to provide recommendations for future work. Issues involving storage setup, hardware, parallel algorithms and image resolution are explored. A balance of hardware, software and systems perspectives is achieved in order to interrelate the design of an image-oriented application on a mesh architecture.

Thesis Supervisor: Steve Ward

Title: Professor

# Table of Contents

Chapter 1: Introduction .....	8
Chapter 2: Background .....	10
Section 2.1: Characteristics of Quadtrees .....	10
Section 2.2: Image representation using quadtrees .....	14
Section 2.3: NuMesh .....	16
Section 2.3.1: NuMesh modules .....	17
Section 2.3.2: CFSM Structure .....	17
Section 2.3.3: CFSM Programming .....	18
Section 2.3.4: Comparison With Systolic Arrays .....	18
Section 2.3.5: The NuMesh Programming Environment .....	19
Chapter 3: Implementation of the Simulation .....	20
Section 3.1: Binary arrays .....	20
Section 3.2: Raster images to quadtrees.....	21
Section 3.2.1: Building a quadtree from a raster .....	21
Section 3.2.2: Merge .....	26
Section 3.2.3: Execution time .....	29
Section 3.3: Rasters to quadtrees .....	29
Section 3.3.1: Building a raster from a quadtree .....	30
Section 3.3.2: Finding a Neighbor .....	33
Section 3.3.3: Execution Time .....	36
Section 3.4: Summary .....	37
Chapter 4: Simulation input and output .....	38
Section 4.1 Input data stream .....	38
Section 4.2 User Options .....	40
Section 4.2.1 Windowing .....	40
Section 4.2.2: Rotation and scaling of the image .....	43
Section 4.3: Output Examples .....	43
Section 4.4: Color Images .....	46
Section 4.5: Summary .....	47
Chapter 5: Data Retrieval .....	48
Section 5.1: Introduction .....	48
Section 5.2: System Architecture .....	49
Section 5.2.2: Storage Management .....	52
Section 5.3: Issues of parallel processors in real-time applications .....	53
Section 5.3.1: Parallel Processing and Real-time Control .....	53
Section 5.3.2: Communication of Processors with Image Interface .....	55
Section 5.3.4: Summary .....	57



Chapter 6: Parallel Processing of Quadtrees .....	58
Section 6.1: Introduction .....	58
Section 6.2: Pointer-less versus Pointer-based Quadtrees .....	59
Section 6.3: Basic parallel algorithms .....	62
Section 6.4: Building Quadtrees .....	64
Section 6.4.1: Routing Data .....	66
Section 6.4.2: NuMesh analysis .....	67
Section 6.5: Summary .....	68
 Chapter 7: Conclusion .....	 69
 Appendices .....	 71
Appendix 1: User Interface .....	71
Appendix 2: Simulation .....	74
 References .....	 92



# List of Figures

Figure 2-1: A quadtree example. a) pixel raster values b) maximal block representations c) corresponding quadtree.....	11
Figure 2-2: A Checker board spatial data image .....	13
Figure 2-3: The NuMesh Diamond Lattice .....	16
Figure 2-4: The internal module of a NuMesh processor .....	17
Figure 3-1: Order of node formations .....	20
Figure 3-2: A quadtree example. a) pixel raster values b) maximal block representations c) corresponding quadtree .....	22
Figure 3-3: Intermediate quadtrees in the process of obtaining a quadtree for the first half of the first row .....	23
Figure 3-4: The quadtree after processing one full row of pixel data .....	23
Figure 3-5: The image transformations due to merges. ....	27
Figure 3-6: Quadtree example. a) binary array representation b) maximal decomposition c) quadtree representation .....	30
Figure 4-1: a) an 8x8 original image b) the windowed image .....	41
Figure 4-2: a) a 4x4 image to be shifted by 3 to the right and 1 up. b) the shifted image represented by the broken lines .....	42
Figure 4-3: An initial image of a sunset given to the user .....	44
Figure 4-4: The updated image for the southeastern quadrant .....	44
Figure 4-5: Successive requests for details from the original to the northwestern to the southeastern quadrants. ....	45
Figure 4-6: A slightly distorted image of Fantasia .....	46
Figure 5-1: A proposed image system architecture .....	49
Figure 5-2: Image access times for two and four communication links respectively .....	51
Figure 5-3: A setup for the NuMesh processor array .....	55
Figure 5-4: Internal architecture of a PE .....	56
Figure 6-1: Indexing: a) row-major b) snake-like c) shuffled row-major .....	62
Figure 6-2: a) original mesh b) merging pairs of 2x2 .....	64

## List of Tables

Table 3-1: Adjacent (I, O).....	34
Table 3-2: Reflect (I, O) .....	35
Table 6-1: Leaf node counts .....	61



# Chapter 1: Introduction

Computer manipulation of images is gaining a great deal of interest and enthusiasm. Advances in processor power, networks and communication have helped drive widespread applications in image processing, computer graphics, geographic information systems, computer vision, robotics, virtual reality and other areas. These advancements may be aided by new techniques of processing video and spatial data.

Digital image processing is widely used in digital television, video communications, pattern recognition for compression and enhancement of images, extraction of their essential parameters and characteristics, transformations of images, filtering, analysis and estimation. When one wants to perform these algorithms in real time and with a very large set of data, one must use parallel computing capabilities which usually consist of many relatively small and simple processing elements (PEs) connected in a regular way by local connections. A lot of the new research being performed is aimed at taking advantage of a computer system's hardware for the purposes of parallel manipulations of images and spatial data like rasters. Some examples of such systems are MPP [Batcher 1980], GAPP [Davis and Thomas 1984], CHIP [Snyder 1982], systolic and wavefront processors. Each PE can range from a simple 1-bit ALU which performs the local operations on binary pixels to complicated 32-bit processors which perform filtering operations and orthogonal transforms with floating point arithmetic.

Graphic and multi-media user interfaces promote the use of computers for visualizing pixmap images. In the fields of scientific modelling, medical imaging and cartography there is an urgent need for huge storage capacities, fast access and real-time interactive visualization of pixmap images.

While processing power and memory capacity double every two years, disk bandwidth only increases ten percent per year. Interactive real-time visualization of full color image data requires a throughput of a few megabytes per second. Parallel input and output devices are required to access and manipulate at high speed image data distributed on disk arrays. A high performance high capacity image server should allow users located on local or even public networks with a set of adequate services for immediate access to images stored on disks. Basic services include real-time extraction of image parts for panning pur-

poses, resampling for zooming in and out, browsing through 3D image cuts and accessing sequences at the required resolution and speed.

This thesis deals with how color images can be partitioned into extents and decomposed through the use of a data structure called quadtrees. It analyzes the performance of the system according to various parameters such as the size of the image, resolution requirements, storage facilities and hardware setup. The implementation and tests were conducted on the NuMesh [Ward et al. 1993] system which is developed under the direction of Steve Ward of the Computer Architecture Group at MIT's Laboratory for Computer Science. It aims at establishing a flexible yet fast and powerful system for communication between processors in a mesh based parallel processing system. Algorithms and techniques for image decomposition will be implemented in ways to take advantage of the hardware's architecture, efficiencies and routing capabilities.

The work will be described in different parts. Chapter 2 will describe some background information about quadtrees and their uses in existing systems and imaging techniques. It will also describe the NuMesh system for which this work is intended. Chapter 3 will detail the implementation of the decomposition. It will explain ways of obtaining quadtrees from a set of pixel data points making up a raster image and how to obtain an image from a quadtree. Also, various criterion for resolution and lossless data compression will be discussed. Chapter 4 will explain the user interface and options which may be chosen. How the original image data is stored and the representation of a quadtree in memory is discussed. Chapter 5 will discuss some of the issues involved in storage facilities and hardware parallel methodology for efficient and time-intensive implementations. Chapter 6, finally, will try to show the feasibility and practicality of using this implementation. Some built-in hardware algorithms are described which are used to make the image decomposition techniques more time efficient and minimizing processor overhead and calculations. Chapter 7 concludes with an overall summary of past chapters and a depiction of the practicality of image decomposition on mesh architectures.

## Chapter 2: Background

There are many hierarchical data structures in representing 2D spatial data like digital images made up of pixels. A very common technique in this area of research is based on the divide and conquer decomposition known as quadtrees [Samet 1990]. Its uses have evolved in a number of different fields. One of the many attractive features of quadtrees is its ability by aggregating data having similar characteristics to save storage and minimize execution time of the process. These are two aspects which are ideal in developing any kind of system: speed and memory. Ideally, one would like a design whose speed is as fast as possible while using as little memory and hardware to achieve the desired output.

### Section 2.1: Characteristics of Quadtrees

The term quadtree is used to describe a type of hierarchical data structure whose common property is that it is based on the principle of recursive decomposition of an image. There are different kinds of quadtrees based on the following:

- type of data being represented
- principle of decomposition
- resolution

Presently, quadtrees are used for point data, areas, surfaces and volumes. The decomposition may be into equal parts on each level or it may be determined by the input. The resolution may be fixed beforehand or it may be defined by properties of the data being used.

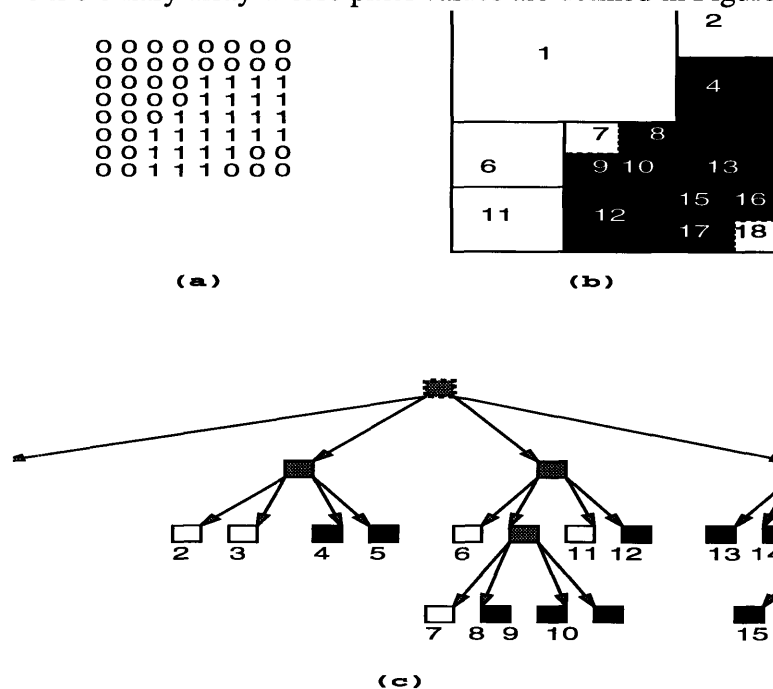
An image may be thought of as a collection of image elements. This collection may be defined in a number of different ways like arrays, lists or trees. Furthermore, sub-collections of similar image elements may be grouped together into blocks and such a collection of blocks can as well be represented as arrays, lists or trees. After one decides on a representation of the image, the procedure to order individual image elements with respect

to each other needs to be addressed.

The array is the most common and simplest way of representing an image. For large images, though, the amount of memory needed for storage may be quite excessive and a raster representation may be used instead. A raster image is an image with a list of row data. The image is processed row by row and can be improved by decomposing the rows into blocks of identically valued pixels.

The simplest and most widespread approach to quadtrees is referred to as region quadtree [Samet 1990] which is based on successive subdivision of a bounded image array into four equal sized quadrants. If all the pixels in each quadrant are not made up of the same value, it is subdivided into quadrants, subquadrants, etc. This process continues until blocks are obtained which consist entirely of the same value. This value is the pixel value. Through a lot of past research in raster and image processing, an image space was a grey scale depiction made up of pixel values of 0's and 1's. A zero represented a white block while a one was a black one. This thesis extends this idea to color images. So, for the latter part of this document, a pixel value will represent a set of three bytes made up of red, green and blue components of the color. Some of the examples initially given, for simplicity, will still use binary scale just to illustrate the concepts and techniques involved in image decomposition.

Figure 2-1 shows an example of a quadtree representation of a binary raster image. It represents an 8 x 8 binary array whose pixel values are defined in Figure 2-1a.



**Figure 2-1: A quadtree example. a) pixel raster values b) maximal block representations c) corresponding quadtree**

The resulting blocks (also referred to as nodes) are combined to an image shown in Figure 2-1b. Finally, taking this new block maximal block structure a quadtree is formed. It is of level 4 ranging from the root node at the top to the third level of leaf nodes.

The tree representation shows a root node which corresponds to the entire array of pixel values. Each node may have zero or four children. These children may be thought of as the NW, NE, SW and SE quadrants of the region represented by that node. As can be seen from Figure 2-1c, the leaf nodes correspond to blocks of uniform homogenous pixel values where no subdivision is needed. A leaf node is depicted as either a white or black square depending whether it is a white or black pixel representation. A number is labeled next to each leaf node so the reader may observe the translation from the maximal block representation to the quadtree one.

The quadtree is a refinement of the array representation of an image which tries to save storage by taking advantage of some regularity in the image by decomposing the array into homogenous disjoint blocks centered at some prespecified positions. The quadtree can be implemented as a list or tree. The list representation facilitates sequential access but is inefficient for random access to specific image elements. *The tree representation is an attempt to balance the costs of sequential and random access.*

A shortcoming of the tree representation of a region quadtree is that a considerable amount of overhead is associated with it. Each node is in need of additional space for each of its four children, parent and pixel value. This is a problem arising with a large image because its quadtree representation may not be fully placed in core memory. One may be able to use binary image trees which is an alternative decomposition defined in a manner analogous to the region quadtree except that at each subdivision stage one subdivides the image into two equal sized parts. In two dimensions, at odd stages, one partitions along the x coordinate and at even stages along the y coordinate. The binary tree can be used to reduce overhead since a node needs memory for only two children instead of four. Thus, its attractiveness increases with higher dimensions since less space is wasted on nil pointers to each of the children of the leaf nodes.

Reducing the overhead costs associated with the use of pointers leads to a considerable amount of interest in pointerless quadtree representation in the form of lists. They can be grouped into two categories: the first treats the image as a collection of leaf nodes while the second represents the image in the form of a traversal of the nodes of the quadtree.

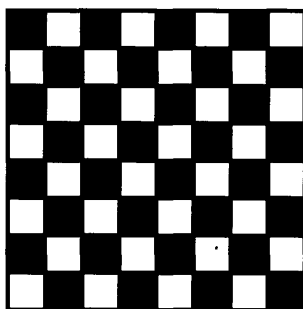
The region quadtree is a variant on the maximal block representation. It requires the blocks to be separate and have some kind of standard size. The motivation for its use was a need to produce a systematic means of representing homogenous sections of an image. Figure 2-1's criterion for homogeneity is that a block of pixels all have the same pixel value. There are other possibilities. For gray scale or color images, the standard devi-

ation of a group of adjacent pixels may need to be below some threshold. Using this criterion, the image array is subdivided into quadrants, subquadrants, subsubquadrants, etc. until homogenous blocks are obtained. If one thinks that each leaf node contains the mean gray or color level of its block, the resulting quadtree will then completely specify a piecewise approximation of the image where each homogenous block is represented by its mean. If the threshold is equal to zero then we revert to the case that the pixel values be the same for a set of adjacent pixels to be reconstructed as one larger block.

The homogeneity criterion can be chosen to guide the subdivision process. It depends on the type of data represented. This will be a major issue in later parts of this document as one considers color pixel values and the trade-offs between memory efficient and distortionless representations of an image.

The example given in Figure 2-1 has the decomposition process based on a standard size of each pixel representing a square. Theoretically, any geometry may be used. Of course, if one were using triangles and attempts were made to decompose a square or rectangle, the nonsymmetry will pose many problems. Squares were used because of two basic characteristics. First, squares produce an infinitely repetitive pattern so that it can be used for images of any size. Images of video or spatial data are intrinsically squares or rectangles. Second, squares yield a partition which is infinitely decomposable into smaller subdivisions. This resolves to better resolution of the image.

The major reason for using quadtrees is to reduce the amount of space necessary to store data by trying to aggregate the largest homogenous blocks possible. This important quality will allow the reduction of execution time of many operations. Of course, quadtrees may not be very efficient in all cases. For example, Figure 2-2 shows an exam-



**Figure 2-2: A Checker board spatial data image**

ple where quadtrees would be a very inefficient means of representing the image. This is because there are no homogenous blocks which are larger than the leaf nodes. So, if this were an  $N \times N$  square, the represented quadtree needs  $N^2$  leaf nodes plus all the inner

nodes of the quadtrees which are

$$(B + W) \times \frac{4}{3} \quad \text{Eq. (2.1)}$$

where  $B$  is the number of black pixels and  $W$  is the number of white ones. On the other hand, a binary array using  $2^{2n}$  bits would be more efficient than a quadtree. Of course, one is assuming that the an image as in Figure 2-2 is an anomaly and will almost never occur in a real digital image. Obviously, the amount of space required is a function of the resolution (the number of levels in the quadtree) and the size of the image (perimeter).

Having a quadtree like the one of a checker board may be inefficient but the number of nodes in a quadtree corresponds directly to the resolution of the image. This is quite significant because increasing the image resolution of a quadtree leads to a linear growth in the number of nodes. This is in contrast to the binary array representation where doubling the resolution leads to quadrupling of the number of pixels.

The number of nodes also affects the execution time of an image decomposition. Most algorithms that execute on a quadtree representation of an image instead of an array representation have an execution time proportional to the number of blocks in the image rather than the number of pixels. This basically means that the use of a quadtree algorithm in  $d$ -dimensions will perform as well as one of a  $(d-1)$  dimensional space for an array.

These distinctions between an array and quadtree representations have allowed this researcher to conclude that the latter is a more suitable technique in image processing for a parallel system. The characteristic that quadtrees are recursive makes it an easy and very powerful tool.

## Section 2.2: Image representation using quadtrees

The quadtree data structure is commonly used in image coding to decompose an image into separate spatial regions. Klinger and Dyer [1976] provide a good bibliography of the history of quadtrees. Their paper reports examples on the degree of compaction of picture representation which may be achieved with tree encoding. Their experiments show that tree encoding can produce memory savings. Horowitz and Pavlidis [1976] show how to segment a picture traversal of a quadtree. They segment the picture by polygonal boundaries. Tanimoto [1979] discusses distortions which may occur in quadtrees for pictures.

The uses of quadtrees are apparent as well in networking. Zhang et al. [1993] describe an image-coding technique in which an adaptive quadtree scheme is used to encode motion compensated difference images. A mechanism was identified within the encoding process that allows a predetermined level of image quality to be selected by the user at setup time and then maintained by the encoder. The context which this work was

performed is video services especially videophone and videoconferencing. They believe that quadtrees are quite useful in these applications due to their ability to control the resolution level. If a connection between two servers needs high quality images then more resolution will be needed, more data will be processed and the costs will be greater.

Other than in networking applications, quadtrees are useful in compression of image data. Quadtree decomposition is a simple technique for image representation at different resolution levels. This representation is successfully used in binary image compression algorithms. Recently, quadtree decomposition has been used as part of image sequence compression algorithms. For gray level images, it is attractive for a number of reasons:

- Relative simplicity compared to other methods like DCT-based coding. This makes it attractive for applications such as video and HDTV compressions.
- The adaptivity of the decomposition. The decomposition divides the image into regions with size depending on the activity in the region. The compression performance is adapted to the various image regions.
- The useful output of the decomposition. The decomposition achieves an image segmentation which is useful in various applications.

It ought to be quite apparent that quadtrees are an exceptional data structure for image processing. Many of the intrinsic details of its efficiency and distortion due to compression will be described in the next few chapters. However, it should be noted, that there is a great deal of work already performed in image processing and it would be quite invaluable to adhere much of what is already known and developed to parallel systems. There are clear advantages of this decomposition technique. Systems for video and images are numerous and quite common in today's highly technological era.

The MIT Media Laboratory, for example, has explored at great length details of digital video coding. They developed the Cheops imaging system [Bove and Watlington 1994] which is a compact platform for acquiring, processing and displaying images. These images may be video or movie frames. Observing the hardware and software design in order to achieve flexible video processing, a number of useful requirements were pinpointed:

- The system should be real time.
- The system should be easily programmable.
- The system should be easily upgradeable as technology improves.
- The system should be simple and compact.

The Cheops system uses parallelism in achieving its image coding goals. It takes advantage of parallelism through individualized processing units which are separate computing elements and multiple stream processors which operate on a single module. These qualities of parallelism are quite common on many digital systems under development.



In recent years, massively parallel processing architectures have shown excellent performance in different scientific applications and in image and signal processing. In particular, there exist arrays of identical processors executing the same instruction in each time unit each operating on its own local data and further equipped with a regular inter-connection network that allows them to communicate with one another. For example, a mesh connected array is made up of processors on a square lattice with each processor able to communicate with its four nearest neighbors. Valuable additions to this basic hardware may exist. Such systems are well suited for problems requiring simultaneous processing of very large amounts of data and have several exemplars in the commercial world including the DAP, MPP, the Connection Machine and MasPar. Another one that is under development at MIT is the NuMesh system. It is the hardware which this image simulation is to be used.

### **Section 2.3: NuMesh**

Abstractly, NuMesh [Ward 1993] consists of modules, or nodes, that may be connected together to form a three-dimensional mesh. For example, Figure 2-3 shows a simplified view of a small mesh of current prototype nodes. This figure depicts each module as a unit whose peripheral connectors provide signal and power contacts to four immediate neighbors.



**Figure 2-3: The NuMesh Diamond Lattice**

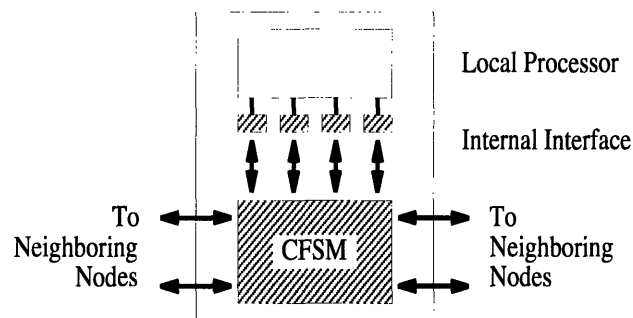
Each node in the mesh constitutes a digital subsystem that communicates directly with each of its neighbors through dedicated signal lines. During each period of the globally-synchronous clock, one datum may be transferred between each pair of adjacent modules. Currently, the prototype runs approximately at 1.2 Gbits/second per port; next-

generation modules will be clocked faster.

### Section 2.3.1: NuMesh modules

The approach in the design of the NuMesh communication substrate involves standardizing the mechanical, electrical, and logical interconnect among modules that are arranged in a three-dimensional mesh whose lowest-level communications follow largely precompiled systolic patterns. The attractiveness of this scheme derives from the separation of communication and processing components, and the standardization of the interface between them. By making the communications hardware as streamlined and minimal as possible, and requiring the compiler to do almost all the work for routing data within the mesh, it should be possible to maintain high-bandwidth, low-latency communications between the processing nodes distributed throughout the NuMesh.

An idealized NuMesh module is a roughly rectangular solid with edge dimension on the order of two inches. A node is logically partitioned into two parts: a *local processor* that implements the node's particular functionality, and a *communications finite state machine* (CFSM), replicated in each node, that controls low-level communications and interface functions. A node's local processor may consist of a CPU, I/O interface, memory system, or any of the other subsystems out of which traditionally-architected systems are constructed. A node's CFSM consists of a finite state machine, data paths for inter-node communication, and an internal interface to the local processor. A typical module is depicted schematically in Figure 2-4.



**Figure 2-4: The internal module of a NuMesh processor**

### Section 2.3.2: CFSM Structure

The core of the CFSM control path is a programmable finite state machine. The

transition table, held in RAM, is programmed to control all aspects of routing data to other nodes and interfacing with the local processor. A small amount of additional hardware reduces the required number of states by providing special-purpose functionality such as looping counters.

The CFSM data path consists of a number of ports connected through a switching network allowing data from one port to be routed to another. Most of the ports are for communication to other CFSMs; however, one port supports CFSM-local processor transfers and allows the CFSM to move data between the processor and the mesh. This port may be wider than the network ports, and provide out-of-band signals in addition to the ordinary data path. Optimally, any combination of ports may be read or written on each clock cycle, but this flexibility may be constrained in a given implementation.

Each CFSM also contains an oscillator to generate the node's clock (the local processor has the option of an independent clock), and circuitry to control the phase of the oscillator. The clocks can be kept globally synchronized by any of a number of methods. The clock cycle time is constrained by the time necessary to transfer a data word between adjacent nodes, which can be made quite short because of the prescribed limited distances between nodes, the point-to-point nature of the links, and the use of synchronous communications.

### **Section 2.3.3: CFSM Programming**

The transition table of the CFSM is typically programmed to read inputs from various neighbors or the local processor into port registers and send outputs from various port registers to other neighbors or the local processor on each clock cycle. It may be thought of as a programmable pipelined switch.

The CFSMs in a mesh, operating synchronously at the frequency of the communication clock, follow a *compiler-generated preprogrammed, systolic communication* pattern. The aggregate CFSM circuitry constitutes a distributed switching network that is customized for each application; its programmability allows this customization to be highly optimized.

### **Section 2.3.4: Comparison With Systolic Arrays**

Arrays of synchronous processors connected together in fixed interconnection topologies offer a high performance environment for certain kinds of applications. Code which can be partitioned into a tightly coupled set of computations exchanging data in fixed communication patterns fits very well into this model of distributed computing. Reconfigurable communication paths have not been a part of traditional systolic array designs. However, the IWarp (iwarp) computer does allow 1D and 2D mesh connectivity between computing units. Software technology developed for this kind of environment may offer a good starting point for further development of compilation technology which

predicts and schedules communication in the broader set of topologies and applications currently being studied in the NuMesh project.

### **Section 2.3.5: The NuMesh Programming Environment**

Previous work on the NuMesh programming environment has centered on a static programming model. A high performance approach relies on a precise cycle count of the computation performed on each node's processor to efficiently schedule communication between the nodes. This work has resulted in the specification of an intermediate language to which programs written in higher level programming languages can be compiled and executed on the Numesh.

Language and user interface issues associated with providing the programmer with an environment in which to develop programs whose dynamic behavior does not depend on run-time data have also been investigated in some detail. Results include a pictorial stream-based front end implemented for the development of real time applications such as digital signal processing for speech and video applications.

A C language interface for writing programs with flow controlled communication requirements has also been implemented. The user of this system is able to write programs in a restricted subset of C and produce compiled processing element code as well as binary executables for CFSMs.

## Chapter 3: Implementation of the Simulation

The quadtree is an excellent approach to color images because of its hierarchical nature. However, most images are represented by such methods as binary arrays, rasters, chain codes or polygons. The criterion for choosing one over the other may be for many reasons like efficiencies and speed of the hardware. For example, rasters are useful for raster-like devices like televisions. Therefore, techniques which can switch among these methods would be of great importance.

The region quadtree for the simulation being implemented is developed on a pointer-based technique. The details of the image processing simulator will be discussed including transformations from raster to quadtree and vice versa as well as the merging criterion.

### Section 3.1: Binary arrays

The most common type of image representation is a raster. There are a number of different ways of transforming a raster to a quadtree. The simplest approach is one which converts an array to a complete quadtree and then repeated attempts at reducing the size of the tree through merging is performed. This merging may be quite extreme if it required that a group of four pixels be of the same uniform color or lenient if they have to be within some threshold of each other. The transformation to a quadtree may be very inefficient because many nodes may be created and then deleted due to merging. It is possible that a computer may run out of memory when employing this algorithm.

One can avoid the creation of nodes by visiting the elements in a specific order. For example, let the raster pixels be read as depicted in Figure 3-1a for a 4 x 4 image. This

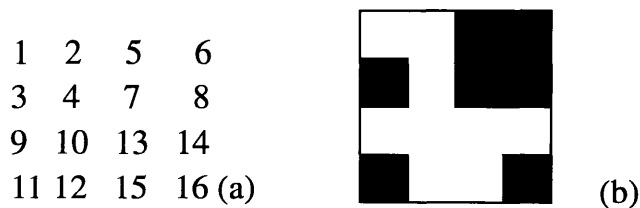


Figure 3-1: Order of node formations

order is known as the Morton Order [Morton 1966]. By using this method, a leaf node is never created until it is certain to be maximal. This basically means that a situation does not arise where four leaf nodes require merging to their parent node. For example, leaf nodes 5, 6, 7 and 8 in Figure 3-1b are all of the same pixel color so no quadtree nodes were created for them. A parent node representing all four blocks of pixel color black was formed to span that part of the image.

The array to quadtree algorithm [Samet 1980] examines each pixel in the raster image only once and in a manner analogous to a preorder tree traversal.

## **Section 3.2: Raster images to quadtrees**

An image may be thought of as a sequential file where the pixel data is defined one row at a time. From Chapter 2, one may remember that a raster representation constitutes a list of rows made up of a specified number of pixels. Each row is made up of the pixel values for the columns in the image under consideration.

With any kind of image processor or user interface, a raster needs to be displayed and thus there is a need for a method to convert a quadtree back to a row-by-row raster. In the following few sections, methods are presented for the conversion between quadtrees and raster images.

### **Section 3.2.1: Building a quadtree from a raster**

The key idea for the raster to quadtree conversion is that at any point, a valid quadtree exists with all unprocessed pixels to be defaulted to the color white. Thus, as the quadtree grows, nodes are merged to yield maximal blocks. This contrasts with another approach taken by some researchers which makes a complete quadtree with one node per pixel and then attempts merges. Such an approach is quite wasteful of memory and time.

The main procedure for the raster to quadtree transformation is called *raster\_to\_quadtree*. It is invoked with a pointer to the first row of data and the width of the image. The width may be perceived as the number of pixels in each row. Let's assume that the image has an even number of rows. If this is not true, then a row of white pixels is added on. The quadtree represents a square image of a side length that is a power of 2. All pixels added such that this is true are defaulted to be white.

A pixel node is defined as a structure containing nine fields. The first three represent the values for each of the red, green and blue values of the pixel color. The next five fields contain pointers to the node's parent and its four children which correspond to the four quadrants. If a node is a leaf node, then it will have four pointers to the empty (Nil) element. The last field represents the level of that node in the tree. Leaves are of level 1, their parents level 2, etc. This is true until one reaches the root node. The complete struc-

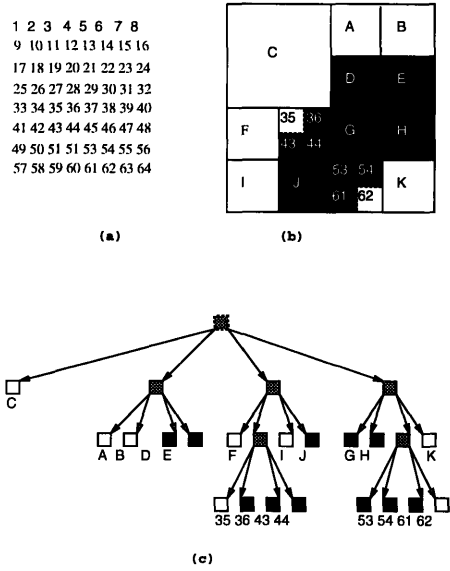
ture may be defined like

```

struct NODE {
    int pixel_red;
    int pixel_green;
    int pixel_blue;
    struct NODE *parent;
    struct NODE *NW;
    struct NODE *NE;
    struct NODE *SW;
    struct NODE *SE;
    int Level;
};

```

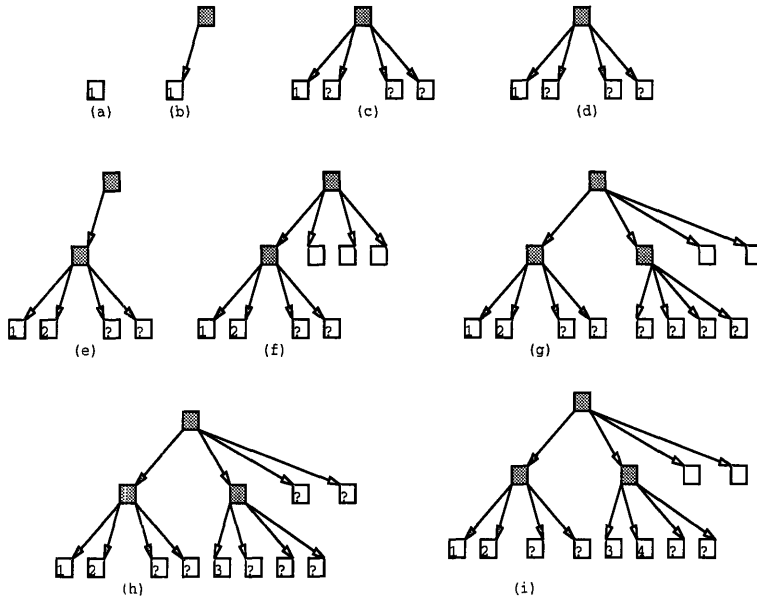
As an example of the application of the algorithm, consider the region given in Figure 3-2a which is a decomposition of an image into rows. Figure 3-2b is the block decomposition while Figure 3-2c is the quadtree representation. All nodes resulting from merging have been labeled with letters and the alphabetical order represents the order in which the merged nodes have been created.



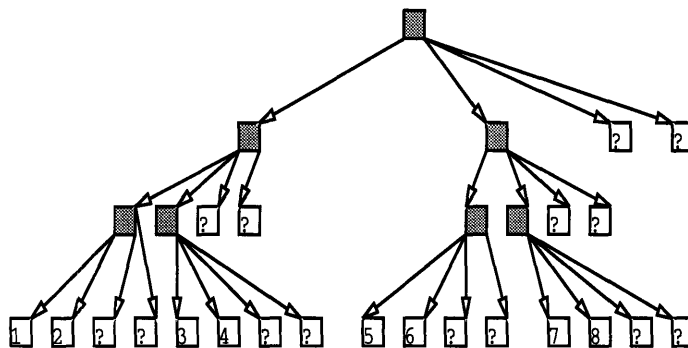
**Figure 3-2: A quadtree example. a) pixel raster values b) maximal block representations c) corresponding quadtree**

Figure 3-3 shows the steps in the construction of the quadtree corresponding to the first four pixels of the first row (pixels 1 through 4). Figure 3-4 shows the tree after the first

full row has been processed.



**Figure 3-3: Intermediate quadtrees in the process of obtaining a quadtree for the first half of the first row**



**Figure 3-4: The quadtree after processing one full row of pixel data**



The *raster\_to\_quadtree* procedure uses a number of different functions. Ones exist for processing even rows, *evenrow*, and odd rows, *oddrow*. Another procedure, *add\_edge\_neighbor*, locates neighboring nodes on the tree. A function, *merge*, is responsible for combining four children of the same pixel color by a single larger block node with the same color. The reason that one needs to differentiate between even and odd rows is apparent when merging. Merges are only performed on even rows so as soon as an even row is processed, an attempt at merging is initiated. The amount of work to process a row depends on whether an odd numbered or even numbered row is under consideration. Odd numbered ones do not require as much work as even ones.

In processing odd numbered rows, the row of pixel nodes are added to the tree from left to right. As the quadtree is constructed, nonleaf nodes must also be added. Wanting to have a valid quadtree after processing each pixel, whenever a nonleaf node is added, its children are as well added. Their pixel values may be defaulted to white until the correct ones are processed from the data.

From Figure 3-3, it may have been noticed by the reader that consecutive pixels 2 and 3 in the first row are not placed in the same subtree as the quadtree is developed. This is because pixel 2 belongs to the northwestern quadrant of the original northwestern quadrant while pixel 3 belongs to the northeastern quadrant of the original northwestern quadrant. As the tree is developed, a methodology for the location in the tree where pixels need to be placed is important. Procedure *add\_edge\_neighbor* achieves this purpose by traversing the tree until encountering a nearest common ancestor. Once this is achieved, a descension along the path that is reflected about the axis formed by the common boundary between the two pixels is performed. An exception to this rule is when the pixel whose neighbor is sought is currently at the extreme right of the row. Here, the nearest common ancestor does not exist. In such a case, a nonleaf node is added with its three remaining children having the color white (Figures 3-3c and 3-3f).

Once the nearest common ancestor and its three children are added, one descends again along a path reflected about the axis formed by the boundary of the pixel whose neighbor is needed. During this descent, a leaf node is transformed into a non-leaf one and four children are added (Figure 3-3g). Finally, the leaf node is colored correctly (Figures 3-3d and 3-3h).

Even numbered rows, on the other hand, require more work because merging will occur. At each even column of each even row, a check for merging needs to occur. Once a merge is processed, further checks of merging need to be conducted at higher levels of the tree. This is done at pixel positions  $(x \ 2^i, y \ 2^j)$  where

$$x \bmod 2 = y \bmod 2 = 1 \quad \text{Eq. (3.1)}$$

In this case, if  $i$  and  $j$  are greater than 1, the maximum number of merges is the minimum of  $i$  and  $j$ . For example, in Figure 3-2, pixel 28 would need a maximum of two

merges until the quadtree is using the maximal blocks possible.

It should be noted that the implementation described for quadtrees is bottom-up because it consists upon decisions to merge. In this way, construction is based on the smallest possible block size in the quadtree. If all the relevant subblocks have been combined into a larger block, then a decision is made whether to combine the larger blocks into even larger ones. The other approach is called top-bottom where a judgement is first made as to whether the entire block can be represented as one leaf or whether four subblocks need to be formed. If the block is divided then a decision is made for each subblock whether it needs to be divided into more subblocks.

The *raster\_to\_quadtree* procedure is quite simple. It requires two parameters: a pointer to the row of data, *p*, and the number of pixels in each row, *width*. The pseudocode is defined as:

```
RASTER_TO_QUADTREE (pointer p, integer width)
begin
  pixel array Q[1:width];
  global pointer node Newroot;
  pointer node First;
  integer I;    /* row of data being processed */

  Q = Row (p);
  First = Create_node(nil, nil, Q[1]);
  Oddrow(width, Q, first);
  p = Next(p);
  I = 2;
  First = Evenrow (I, width, null(Next(p)),
    Row(p), Add_edge_neighbor(First, 'S', 'White'));

  while not(null(p=Next(p))) do
    /* Assume even number of rows */
  begin
    Oddrow(width, Row(p), First);
    p = Next(p);
    I = I + 2;
    First = Evenrow (I, width, null(Next(p)),
      Row(p), Add_edge_neighbor(First, 'S', 'White'));
  end;
```

```

        return(Newroot); /*Return the quadtree root*/
    end;

```

### Section 3.2.2: Merge

Each iteration of *evenrow* will call the *merge* function to check for any possible combinations of leaf nodes to form larger blocks. In the original implementations of quadtrees, a merge only occurs if a group of four pixels have the exact same color. The function accepts a pointer to a node in the tree and two integers representing the row and column positions of the image data. This may be seen by the following pseudocode.

```

Merge(integer i, integer j, pointer p)
begin
    while ((I mod 2) = 0) and ((J mod 2) = 0)
        and pixel_val(child(p, 'NW'))=
            pixel_val(child(p, 'NE'))=
            pixel_val(child(p, 'SW'))=
            pixel_val(child(p, 'SE')) do

        /* Since we start with a pixel-sized node,
           it is impossible for the four children to
           be nonleaf nodes */
        begin
            I = I/2;
            J = J/2;
            pixel_val(p) = pixel_val(child(p, 'NW'));
            free-children(p);
            p = father(p);
        end;
    end;
end;

```

However, this assumes that the criterion for merging is that the pixel values be exact. It should be noted that in reality any given pixel value is made up of three integers representing the red, green and blue components. The above code may be made more abstract and general if the criterion for merging is defined by some other function. So, a pointer to a tree node is given to a function which would decide if merging is possible or not. If the function were defined such that the pixel values of each of the children needs to be the same to allow merging, we revert back to the above example. However, the criterion for merging may depend on the difference from the mean of each of the pixels. This allows more versatility and freedom. The more functional and general merging procedure looks like

```

merge(integer i, integer j, pointer p)
begin
  while ((I mod 2) = 0) and ((J mod 2) = 0) and
    merging_ok (p)) do
    begin
      I = I/2;
      J = J/2;
      pixel_val(p) = pixel_val(child(p, 'NW'));
      free-children(p);
      p = father(p);
    end;
  end;
end;

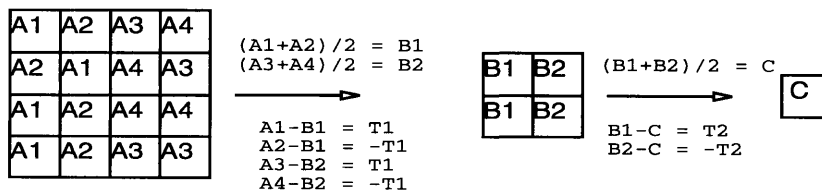
```

The question as to what is a suitable merging criterion arises. A simple implementation is to merge a group of four nodes if each node's pixel value is less than the mean of the pixels by some threshold value, T. This may be expressed by the following:

$$| \text{Pixel}_q - \text{Pixel}_{\text{mean}} | < \text{Threshold} \quad q = \{\text{NE}, \text{NW}, \text{SE}, \text{SW}\}$$

The problem with this condition is that all levels of the tree will be based on this. This may not be practically useful. Leaf nodes of a single pixel ought to be merged on less strenuous needs than higher leveled nodes. These higher leveled nodes represent large blocks of the image and thus may cause a lot of distortion due to merging. This distortion results from errors between the merged blocks and the original image. For example, a group of four nodes having pixel values of 2, 3, 5 and 6 will have a mean of 3.5. If the threshold is 2, then a larger block of 2 x 2 will be formed with the pixel value of 4. Distortion has arisen due to an alteration of the original pixel values. It should be noted that color pixels will have three values for red, green and blue and so each one will have to pass the test in order for merging to occur. Also, it is possible that the threshold value for red be different than blue.

An example of this criterion for merging is depicted in Figure 3-5. The  $A_i$ ,  $B_i$  and  $C_i$  are pixel values.



**Figure 3-5: The image transformations due to merges.**

This thesis improves the basic criterion by first suggesting the use of different threshold values at each resolution level of the quadtree. Nodes near the leaves will be judged differently than ones near the root. The procedure to determine the threshold values will be described below. Let  $T_i$  denote the threshold value at level  $i$ .

The mean square error (MSE) of the subimage representation by a leaf at quadtree level  $i$  is upper bounded by:

$$MSE(i) \leq \sum_{j=1}^i T_j^2 \quad \text{Eq. (3.2)}$$

For further discussion, the following variables are defined:  $N_i$  is the number of pixels at level  $i$  that did not move to level  $i+1$ . The probability of finding a pixel at level  $i$  for a  $2^n \times 2^n$  image is

$$P_i = \frac{N_i}{4^{n-i}} \quad \text{Eq. (3.3)}$$

The pixel propagation probability to level 0 (root) is defined as  $q_0 = 1$ . The empirical probability from level  $i$  to level  $i+1$  is defined by

$$q_i = \begin{cases} 1 & i = 0 \\ q_{i-1} - p_{i-1} & i = 1, \dots, n \end{cases}$$

So, the total MSE of the reconstructed image is defined by:

$$MSE_{tree} = \sum_{i=1}^n p_i MSE_i \leq \sum_{i=1}^n p_i \sum_{j=1}^i T_j^2 \quad \text{Eq. (3.4)}$$

$$MSE_{tree} \leq \sum_{i=1}^n q_i T_i^2 \quad \text{Eq. (3.5)}$$

The goal is to achieve a set of threshold values,  $T_i$ , that minimize the MSE under a constraint of a fixed number of leaves. However, the relation between the threshold values and the MSE is complicated. One will assume that the number of leaves is constant in order to minimize the MSE.

Let's define the number of leaves in the compressed tree,  $L$ , as

$$L = \sum_{i=0}^n 4^{n-i} p_i \equiv 4^n \left( 1 - 3 \sum_{i=1}^n \frac{q_i}{4^i} \right) \quad \text{Eq. (3.6)}$$

which will simplify the MSE criterion to be:

$$MSE_{tree} \leq \frac{4}{3} \left( 1 - 4^{-n} L_{tree} \right) T_1^2 + \sum_{i=2}^n q_i \left( T_i^2 - 4^{1-i} T_1^2 \right) \quad \text{Eq. (3.7)}$$

The first term above is greater than zero and if  $L_{qt}$  is defined, the term depends only on  $T_1$  which may be a given parameter of the simulation. So, the MSE is minimized if the second term is minimized. A very interesting aspect of the above inequality is that the second term is not positive for  $i$  greater than 1

$$T_i \leq 2^{1-i} T_1 \quad \text{Eq. (3.8)}$$

This is the criterion for threshold selection which is used on the simulation. It depends on a given parameter  $T_1$  that controls the distortion tradeoff. At each level, the threshold is a factor of 2 less than the previous level. This ought to make sense because the contribution of the image area represented by a leaf at level  $i$  to total distortion is proportional to the size of the area,  $4^i$ . The most optimal MSE is obtained if the distortion due to merging is distributed uniformly among the various levels of the quadtree. So, at higher levels (closer to the root), distortion is less.

### Section 3.2.3: Execution time

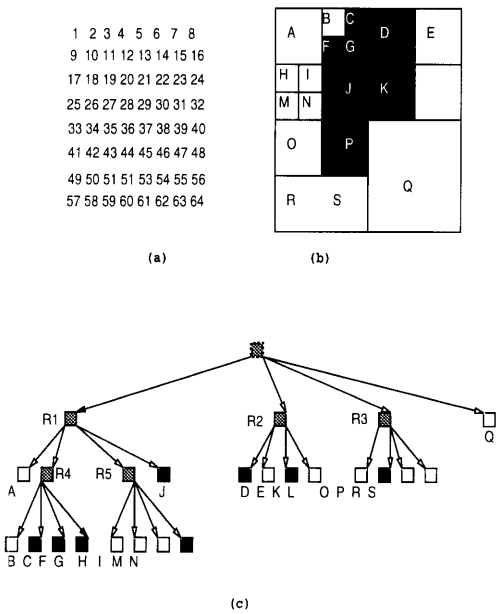
The time for the raster to quadtree construction is measured by the number of nodes visited. Thus, we only need to analyze the amount of time used to find the neighboring pixel (*Add\_edge\_neighbor*) and merge (*Merge*). The number of nodes visited to find a neighbor is bounded by four times the number of pixels in the image. The number of nodes checked for merging is upper bounded when all of the pixels are of the same color. Samet [Samet 1982] claims that the total number of nodes examined for merging is bounded by four-thirds the number of pixels ( $2^{2n}$ ). This occurs when all of the pixels are of the exact same color.

### Section 3.3: Rasters to quadtrees

Once a quadtree is built, one would like to be able to transform the tree back into a raster for output purposes. The most obvious method to build a raster representation from

a quadtree is to generate an array corresponding to the tree but this method may need a large amount of memory which may not necessarily be available. In [Samet 1984], a number of quadtree to raster algorithms are described. All of the algorithms traverse the quadtree by rows from left to right and visit each quadtree node once for each row that intersects it.

Figure 3-6a, for example, shows for pixels 1 through 8 of the first row, node A (Figure 3-6c) is encountered first then nodes B, C, D and E. Each block whether it is black or white at level  $x$  in the tree is visited  $2^x$  times. Each visit results in an output of length  $2^x$  pixels. So, node D which has color black will cause an output of 2 consecutive pixels of color black to represent the node.



**Figure 3-6: Quadtree example. a) binary array representation  
 b) maximal decomposition c) quadtree representation**

**Section 3.3.1: Building a raster from a quadtree**

There are two different approaches, similar to generation of a quadtree, called top-down and bottom-up to achieve a raster from a quadtree. The top-down algorithm starts at the root each time it visits a node that participates in a row while a bottom-up algorithm visits adjacent blocks in a row by the use of neighbor-finding techniques.

Both algorithms make use of a procedure called *Find\_2D\_Block* to find the loca-

tion of the block containing the segment of the row being output. Its parameters are the coordinates of the upper left corner of the leftmost pixel of the segment that is to be output and the coordinates of the lower right corner of a block in the image containing the segment. *Find\_2D\_Block* divides this block recursively until the smallest block which is made up of a leaf node is obtained. The pseudocode is:

```

Find_2D_Block(pointer node p, int x, int xfar, int y,
              int yfar, int w)
begin
  quadrant Q;

  while (node is not a leaf) do
    begin
      w = w/2;

      /* Identify the quadrant of a nonleaf node
       that contains a block corresponding to the
       segment to be output */
      Q = Get_Quadrant (x, xfar-w, y, yfar - w);
      xfar = xfar - xf[q]*w;
      yfar = yfar - yf[q]*w;
      p = child (node p, quadrant q);
    end;
  end;

```

The function *child* basically returns the child of node *p* which represents the quadrant block defined by *q*. So, if *child* (*p*, "NW") were initiated, it would be equivalent to the C instruction of "p->NW".

In locating a block containing the segment starting at row 0 and column 0 in Figure 3-6a, *Find\_2D\_Block* divides the image recursively into blocks having lower right corners at (8,8), (4,4) and (2,2). The result is a block representation of node A.

The actual output of the runs is accomplished by a function named *Output\_Run*. For each leaf node of width *x* pixels that is part of a specific row, the function outputs a run of length *x* of the appropriate color (RGB representation).

The pseudocode for *Output\_Run* looks like:



```

void output_run (int pixel_red, int pixel_green,
int pixel_blue, int length)
{
    int i = 0, counter = 0;

    for (i = 0; i < length; i++) {
        counter = counter + 1;
        print pixel_values to memory storage
    }
}

```

If one were to take the top-bottom approach given in Figure 3-6, the  $R_i$ 's represent non-leaf nodes. The leaf nodes in Figure 3-6a are labeled in the order they are processed. For this image, the output for the first row would be W332. This refers to the first pixel having a white color of length 3 followed by 3 pixels of color black and finally ending with a pair of white pixels. So, an output function would perform the result for each of the rows to obtain the raster image. When outputting the first row, the algorithm needs to start at node R0 and visit nodes R1 and A; R1, R4, B; R1, R4, C; R2, D; R2, E. For the second row, it visits R1, A; R1, R4, F; R1, R4, G; R2, D; R2, E. This goes on for the rest of the rows. The total number of nodes visited in this image is 68.

The top-bottom approach visits each segment of each row numerous times starting at the root of the tree. The bottom-up technique tries to avoid this by using the structure of the tree to find the successive adjacent blocks. For example, in Figure 3-6, once block B is output the next node that is visited is one representing block C. It can be located by traversing links corresponding to nodes R4 and C as opposed to R1, R4 and then C with a top-bottom design.

Using the bottom-up approach to Figure 3-6, the same output will result. However, the difference arises in the order in which the nodes are encountered. For example, the first row will come across the nodes as R0, R1, A; R1, R4, B; R4, C; R4, R1, R0, R2, D; R2, E; R2, R0. The last pair signifies that the end of the row has been reached and no neighbor can be found. The number of nodes encountered here are 96 which may seem to signify that this approach is inferior to the top-bottom one. In reality though, the opposite is generally true and this example is an exception. This will be discussed later on.

The total pseudocode for the quadtree to raster ( $2^n \times 2^n$ ) transformation may be perceived in the following way:

```

Quadtree_to_image (node root, int n)
  /* Obtain a 2nx2n image from a quadtree at root */
  {
    node p = nil;
    int diameter, distance, y;

    diameter = 2n;

    /* Process each row one at a time */
    for (y = 0; y < diameter; y++) {

      p = root;
      distance = diameter;

      /* Find the leftmost block containing row Y */
      Find_2d_block (&p, 0, diameter, y, diameter,
                    &distance);
      out_row (p, y, log2 distance);
      printf("\n");
    }
  }

```

### Section 3.3.2: Finding a Neighbor

In the bottom-up technique described above, adjacent blocks are found without reverting back to the tree root and traversing downwards. Given two nodes, *X* and *Y* whose blocks don't overlap, and a direction *d*, one can define a function *adjacent* such that *adjacent* (*X*, *Y*, *d*) is true if there exist two pixels *x* and *y* in *X* and *Y* respectively such that either *y* is adjacent to *x* in the direction *d* or *y* is adjacent to corner *d* of *x*. So, two blocks may be adjacent if they both belong along the same edge or the same vertex.

The neighbor finding technique is not mathematically based. It is based on obtaining a common ancestor between the present node and the neighbor to be found. Neighboring blocks do not need to be of the same size. If the neighbor is larger, only the part from the nearest common ancestor is retraced. Otherwise, the neighbor corresponds to a block of equal size and a pointer to a node of equal size is returned.

Adjacent nodes are found using the *Find\_2D\_Block* and *Edge\_Neighbor* procedures. For some node *x* and an edge *e*, *Edge\_Neighbor* locates the edge-neighbor for *x* of size greater than or equal to *x* in direction *e*.

If a neighboring block exists, a pointer to its node is returned. If it is a non-leaf node then procedure *Find\_2D\_Block* is used to determine the block that intersects the row currently being processed (the right most block of A in Figure 3-6 is B for the first row and F for the second). The pseudocode for *Edge\_Neighbor* looks like:

```

Edge_neighbor (node p, direction i, node q, int l)
begin
    I = I + 1;

    if (parent(p) != NULL) and (adj (direction i,
        quadrant p))

        /* Find a common ancestor */
        Edge_neighbor (parent(p), i, q, l);
    else
        q = parent(p);

    if (q != NULL)
        if (q not leaf-node) do
            begin
                q = child(q, reflect(direction i, quadrant));
                l = l - 1;
            end
        end
    end
end

```

*Edge\_Neighbor* returns in q the edge neighbor of node p of equal or greater size in direction i. L denotes the level of the tree at which node p is initially found and the level of the tree at which node q is ultimately obtained.

The function *Adj* and the function *Reflect* aid in the operations needed to obtain the neighbor. *Adj* accepts two parameters i and o returning true if quadrant o is adjacent to the i<sup>th</sup> edge or vertex of o's block. For example, *Adj* (*west*, *southwest*) is true. The relation can also be described as true if o is of type i or i is a subset of o. Table 3-1 shows the functionality of the *Adj* procedure.

**Table 3-1: Adj(I,O)**

I direction	O = NW	O = NE	O = SW	O = SE
N	TRUE	TRUE	FALSE	FALSE

I direction	O = NW	O = NE	O = SW	O = SE
E	FALSE	TRUE	FALSE	TRUE
S	FALSE	FALSE	TRUE	TRUE
W	TRUE	FALSE	TRUE	FALSE
NW	TRUE	FALSE	FALSE	FALSE
NE	FALSE	TRUE	FALSE	FALSE
SW	FALSE	FALSE	TRUE	FALSE
SE	FALSE	FALSE	FALSE	TRUE

*Reflect (i, o)* produces the child type (NW, NE, SW or SE) of the block of equal size that shares the  $i^{\text{th}}$  edge of a block having child type value o. For example, *Reflect (North, Southwest)* results in Northwest and *Reflect (Southwest, Southwest)* produces Northeast. Table 3-2 shows the functionality of the Reflect procedure.

**Table 3-2: Reflect (I,O)**

I direction	O = NW	O = NE	O = SW	O = SE
N	SW	SE	NW	NE
E	NE	NW	SE	SW
S	SW	SE	NW	NE
W	NE	NW	SE	SW
NW	SE	SW	NE	NW
NE	SE	SW	NE	NW
SW	SE	SW	NE	NW
SE	SE	SW	NE	NW

Some analysis of this approach would be of great use. The average number of nodes that must be visited in locating the nearest common ancestor when seeking an edge neighbor is no greater than 2. This can be proved by the following: Define a node p at level i and direction I which may be in any of the  $2^{n-i}(2^{n-i} - 1)$  locations. The '-1' is because one of the rows or columns will not have a neighbor in direction I. Of these positions,  $(2^{n-i})2^0$  have their nearest common ancestor at level n,  $(2^{n-i})2$  at level n-1, etc. Starting at node level i, reaching the nearest common ancestor at level j, j-1 nodes must be visited. The average then is

$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^n 2^{n-i} 2^{n-j} (j-i)}{\sum_{i=0}^{n-1} 2^{n-i} (2^{n-i} - 1)} \quad \text{Eq. (3.9)}$$

which may be simplified [Samet 90] to

$$2 + \frac{-6(n-1)2^n + 6}{2^{n+2} - 6(2)^n + 2} \leq 2 \quad \text{Eq. (3.10)}$$

### Section 3.3.3: Execution Time

The neighbor finding techniques were used to develop a bottom-up method for reproducing a raster image. It was mentioned before that the bottom-up approach was more time efficient than the top-bottom one. This claim will be discussed here.

Given a  $2^n \times 2^n$  image with  $x_i$  blocks of size  $2^i$ , the number of nodes visited by the top-bottom algorithm is

$$\text{Top-Bottom: } \sum_{i=0}^{n-1} (n-i) x_i 2^i \quad \text{Eq. (3.11)}$$

This is true because for each block of size  $2^i$  there are  $2^i$  rows, each being visited once starting at the root of the tree. But each block of size  $2^i$  is at a distance  $n-i$  nodes from the root. The above result is quite interesting because it points to the fact that the number of nodes visited only depends on the number of blocks for a given image and their sizes.

For bottom-up, the analysis is more complicated. The leftmost blocks in an image require visiting  $n$  nodes starting from the root. For each row, the algorithm uses a neighbor finding technique as discussed earlier.  $2^0$  nodes need to be visited to find common ancestors at level  $n$ ,  $2^1$  at level  $n-1$ ,  $2^2$  at level  $n-2$ , etc. After finding the common ancestor, a traversal of an equal number of nodes needs to be performed to find the neighbor. The rightmost block has no neighbor. In this case, traversal to the root is performed and then the realization of a non-existent neighbor is achieved. Each row visits a total number of

nodes defined by

$$n + 2 \sum_{i=1}^n i 2^{n-i} + n \quad \text{Eq. (3.12)}$$

which can be simplified to

$$2^{n+2} - 4 \quad \text{Eq. (3.13)}$$

For a  $2^n \times 2^n$  image, the total number of nodes visited will be  $2^{2n+2} - 2^{n+2}$ . Compared to the original image which is made up of  $2^{2n}$  pixels, there is a factor of 4 bound. This makes sense because any two adjacent nodes have to be within 4 nodes of each other. So, for the bottom-up approach, the number of nodes visited for a  $2^n \times 2^n$  image made up of  $x_i$  blocks of size  $2^i$  is

$$\text{Bottom-Up:} \left( \sum_{i=0}^n x_i 2^i \right) 4 \quad \text{Eq. (3.14)}$$

Comparing the expressions of bottom-up and top-bottom, one can see that the bottom-up approach has little dependence on the resolution of the image,  $n$ . When  $n$  is greater than 4, bottom-up performs at a higher level of efficiency when compared to its competitor. As  $n$  grows large, the depth of the tree increases and the neighbor finding techniques become invaluable in saving time.

### Section 3.4: Summary

A quadtree implementation for images was discussed here. Descriptions of both the raster to quadtree and quadtree to raster transformations were described in some detail. The code written in the C programming language is included in the appendix for curious researchers and implementors. It should be quite apparent that the quadtree data structure's divide and conquer decomposition is attractive. An image is a perfect example where a recursive implementation will take advantage of the parallel architecture on which this simulator is to be used and implemented.

## Chapter 4: Simulation input and output

The quadtree may be thought of as a hierarchical data structure. Hierarchical data structures are useful because of their ability to focus on the interesting subsets of data in a given image. This focusing results in efficient representations and in improved execution times.

A user, for example, is able to choose among the four quadrants of the image to focus on. An analogy is a zooming of a camera in order to take a picture of a specific area of interest. This simulation allows for zooming in and out which focusses an image into that part of the picture. For example, if one were given a rough and sketchy picture of the world then a choice of the North American continent will produce a full display of the continent showing countries like the United States and Canada. A refocus onto the north-eastern part of America will show cities found in this part of the United States. This may go on as long as more resolution or zooming is possible for a given image. This may be perceived as a geographical information system on a parallel multiprocessor mesh architecture.

Description of how image data is available and sample user outputs will be described. This will give some kind of context where quadtree transformations described in the previous chapter are used.

### Section 4.1 Input data stream

The input to the simulation is the pixel data representations of the image under consideration. For a 256 x 256 pixel image there will be  $(256)^2$  pixels defined. Each one would be made up of three values representing the red, green and blue components. For the purposes of the simulation it is assumed that each component can be as large as wanted or needed. This is a generalization to actual hardware implementations which would allow one byte of 8 bits for each color representation. So, in the latter case, there would be  $(1 \text{ byte})^3$  or  $(256)^3$  variations of the color for each pixel. Approximately 17 million shades of the primary colors are allowed. Of course, a large amount of the possibilities would be indistinguishable from others due to human inability at times to detect slight variations in the pixel color.

In any regard, the issue of the variations of colors will be discussed later. The basic data may look like a series of values for each pixel. This can be perceived in the following way:

Pixel1_Red	Pixel1_Green	Pixel1_Blue
Pixel2_Red	Pixel2_Green	Pixel2_Blue
Pixel3_Red	Pixel3_Green	Pixel3_Blue
Pixel4_Red	Pixel4_Green	Pixel4_Blue
...	...	...

So, for a 256 x 256 image, there will be  $(256)^2$  pixel values. If less than that amount is available, the extra needed pixel values would be assumed to be the color white. Also, it should be noted that an image needs to be a power of 2 on both dimensions, vertical and horizontal. If this is not the case, the simulation automatically augments the image to the closest approximation to a square of power 2 dimension and sets the extra pixel positions to be the color white. For example, an initial image which is defined by 320 x 300 pixels will be simulated as one of 512 x 512. But the image when shown to the user will still be depicted as one of 320 x 300.

Let's go back to the example of looking at the world map and focusing into interested areas. The first image of the world will be processed through the quadtree approach described in the previous chapter. Depending on the user's needs, different parts may be zoomed in and thus more detail about that specific area is needed. So, a simulation ought to be able to figure out how to obtain the next stage of detail or resolution for this geographical viewing system. If one were to take the example of a compact disk, it can be thought that each segment of the image is defined in a different sector of a given track. For example, it may be that the most high leveled and general image be found in the first sector of the first track while its northwest depiction is detailed in the second sector of the first track and the northeast in the third sector of the first track, etc.

So, given that one is at some parent image, traversal to the image representing one of the quadrants may be achieved by the following equation:

$$4(i - 1) + j + 1 \qquad \text{Eq. (4.1)}$$

Given that one is at a parent node represented by  $i$ , the node represented by the child of  $i$  in quadrant  $j$  is defined above. So, for example, let's assume that the northwest quadrant is needed when  $j = 1$ ; northeast when  $j = 2$ ; southwest when  $j = 3$  and southeast when  $j = 4$ . Initially, the highest leveled image is in sector 1. To obtain the image of the northeastern quadrant, one would use the image depicted in sector  $4(1-1) + 1 + 1 = 2$ . Once that image is simulated through a quadtree implementation, choosing the southeastern quadrant would result in depicting the image defined in sector  $4(2-1) + 4 + 1 = 9$ . And



this continues for as long as there is more zooming possible in the image. Once the highest level of detail is achieved, a choice of one of the quadrants for more detail will not result in anything. For image depictions defined by n levels including the root image, the number of sectors needed to define each possible detail that may be accessed is

$$\sum_{i=0}^{i=levels} 4^i \quad \text{Eq. (4.2)}$$

Once one zooms into an image, it is possible to revert back to the parent image. This would be found in sector position

$$\left\lfloor \frac{i+2}{4} \right\rfloor \quad \text{Eq. (4.3)}$$

where i is the present sector that is depicted. So, an image at sector 3 would have its parent at sector 1. Once we revert back to the parent node which has already been represented using quadtrees, no more effort is wasted in performing this action again.

## Section 4.2 User Options

Given an image, the user may choose among a number of options. The basic ones are made up of the following:

- Focus on the NW quadrant
- Focus on the NE quadrant
- Focus on the SW quadrant
- Focus on the SE quadrant
- Return to parent quadrant
- Exit the simulation

Some of these options were described above in some detail. Examples of these options will be given in the next section in order to visually understand exactly what is perceived by the user.

### Section 4.2.1 Windowing

One of the more complicated techniques in geographical information retrieval is windowing. It is a process of extracting a rectangular subsection from an image where the subsection can cover parts of different quadrants. An algorithm designed to achieve this effect for a square window of size  $2^k \times 2^k$  at an arbitrary position in a  $2^n \times 2^n$  image that is represented by a quadtree is described in Rosenfeld et al [Rosenfeld 1982]. Simply, the

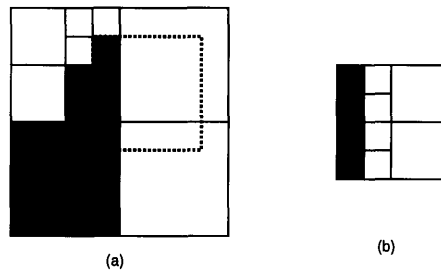
new quadtree is described as the input quadtree is decomposed and relevant blocks are copied into the new quadtree.

The algorithm proceeds in the following manner. Let's assume that the window chosen is smaller than the image,  $k < n$ .

- 1) Find the smallest subtree of the input quadtree that contains the window.
- 2) If the subtree is a leaf node, done. Otherwise, subdivide the window into four quadrants and reapply steps 1 to 3 for each quadrant.
- 3) Try to merge the children of the current node.

The time to produce the window is directly related to two factors: the relative position of the center of the window with respect to the center of the input quadtree and the sizes of the blocks in the input quadtree that overlap in the window.

In Figure 4-1a, an 8 x 8 image is depicted with 10 leaf nodes; 3 black and 7 white. A window represented by borders of broken lines is overlaid. Let's call the original image O and the windowed one W.



**Figure 4-1: a) an 8x8 original image b) the windowed image**

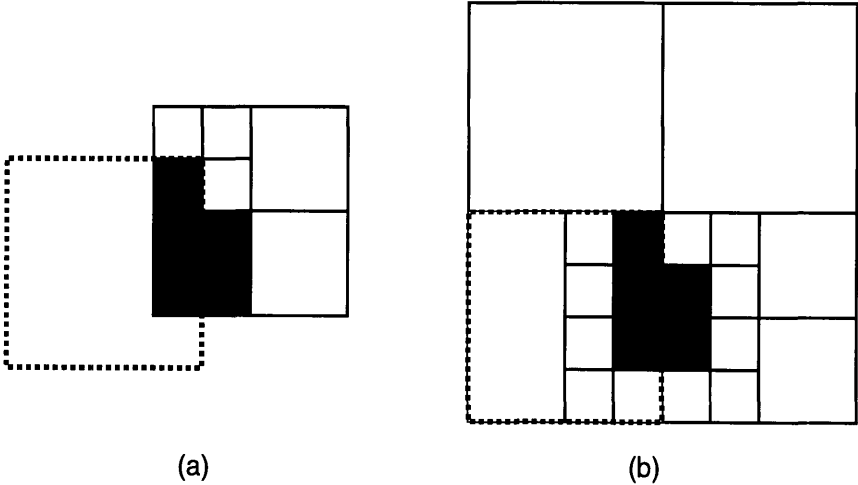
Since W is not contained in a single quadrant of O, we need to subdivide the window and reapply the algorithm to the four resulting windows:  $W_{NE}$ ,  $W_{NW}$ ,  $W_{SW}$  and  $W_{SE}$ . Since  $W_{NW}$  is not completely part of one quadrant of O, further subdivisions would occur. The next stage of subdivisions would result in each child being contained in one node of O.  $W_{NE}$  is contained in one node of O and so  $W_{NE}$  is made up of a white block.  $W_{SW}$  is not contained in one node of O so subdivisions are needed. After this, each child will be made up of one node of O. Finally,  $W_{SE}$  is also not contained in one node of O. We subdivide it and now all its children are made up of nodes of O. Since all the nodes of  $W_{SE}$  are the same color, white, they are merged to form a larger block. The result can be seen in

Figure 4-1b.

Windowing may also be perceived as an intersection request. This is achieved by treating the image and the window as two distinct images like  $I_1$  and  $I_2$ . So, for windowing,  $I_1$  is the image from which the window is being extracted and  $I_2$  is a black image with the same dimensions as the window to be extracted. The quadtree representing the result of the windowing operation has the same dimensions as  $I_2$ . Each pixel of  $I_2$  will have the same value as its representative in  $I_1$ .

Once windowing is perfected, a simple extension is a quadtree translation. Because there is a correspondence between windowing and intersection, translating an image is analogous to extracting a window that is larger than the input image with a different origin from that of the input image. If the image to be translated has its origin at  $(x, y)$  then translating it by  $\text{deltax}$  and  $\text{deltay}$  means that the window is a block whose origin is located at  $(x - \text{deltax}, y - \text{deltay})$ .

Figure 4-2 gives an example of translation.



**Figure 4-2: a) a 4x4 image to be shifted by 3 to the right and 1 up.  
b) the shifted image represented by the broken lines**

As can be seen, Figure 4-2a shows a 4 x 4 image which is shifted by 3 east and 1 north. Assuming the origin is located at the bottom left corner means that an intersection of the image with a 4 x 4 window whose origin is located at  $(-3,-1)$  is to be conducted. This window is overlaid in Figure 4-1a with broken lines. The quadtree of the translated image is shown in Figure 4-2b and is shown with broken lines.

### **Section 4.2.2: Rotation and scaling of the image**

The simplest transformations are those in which the translation distance is a power of two and is greater than or equal to the width of the largest block in the image, the rotation is a multiple of ninety degrees or the scaling factor is a power of two (doubling or halving).

In order to make an image depicted by a quadtree one half the original size, a simple procedure is followed. Create a new root and give the root three children having the color white and one child that represents the original tree. To make the quadtree twice as large, choose one of the subtrees like the Northwest to serve as a new root. This eliminates the remaining three quadrants. This technique is applied recursively for any scale factor that is a power of two.

Rotations of ninety degrees are quite simple. The goal can be achieved by rearranging the pointers at all levels. The algorithm traverses the tree in a preorder manner and rotates the pointers at each node. This basically means the following series of actions will be performed until the leaves are reached.

- 1) The pixels in the northwestern quadrant of the image become the southwestern quadrant.
- 2) The pixels in the northeastern quadrant of the image become the northwestern quadrant.
- 3) The pixels in the southeastern quadrant of the image become the northeastern quadrant.
- 4) The pixels in the southwestern quadrant of the image become the southeastern quadrant.
- 5) Each of the quadrants in its new position appears as if had been locally rotated clockwise by ninety degrees.

### **Section 4.3: Output Examples**

Some examples of user actions and consequences are given here in order to make the simulation more concrete. It should be noted that all images are of size 256 x 256 but may at times be represented in different dimensions. As one may remember, an image is given initially and the user may choose to concentrate on any of the four quadrants. Given a user choice of a specific quadrant, a more detailed depiction of that part is produced by accessing data from some source like a CD-ROM where each sector will have information about a specific area of the initial image.

The first example, Figure 4-3, is one of the ocean. This is given to the user initially and a response from the user is expected.



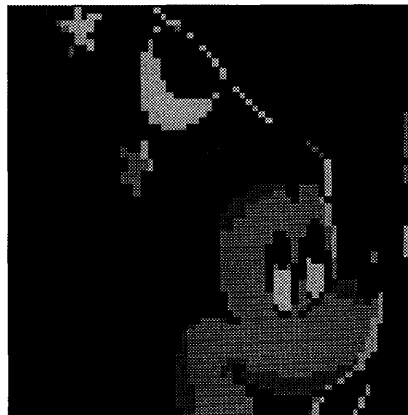
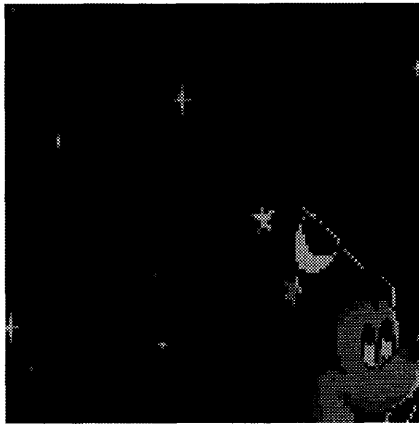
**Figure 4-3: An initial image of a sunset given to the user**

Let's say that the user wants more detail of the bottom right corner of the image. After choosing this as his option, the updated image, Figure 4-4, is formed.



**Figure 4-4: The updated image for the southeastern quadrant**

The next example uses an image of Fantasia. The user is given the initial image and successive inquiries are answered by requests for the northwestern quadrant and then the southeastern quadrant. We finally have a detailed 256 x 256 image of Mickey Mouse.



**Figure 4-5: Successive requests for details from the original to the northwestern to the southeastern quadrants.**

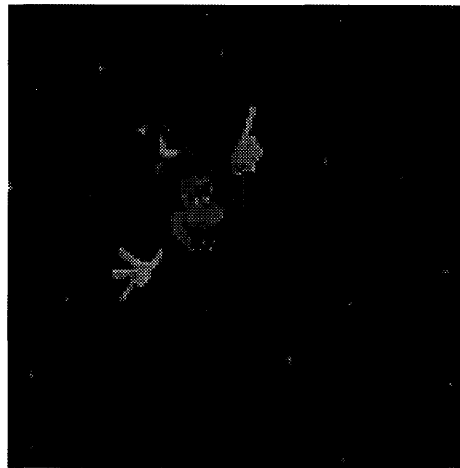
There are a couple of important things to take note of in this approach to image resolution:

- Within a given image, quadtree nodes closer to the root will have a more difficult criterion to merge. This is to allow as little distortion in higher levels which would affect larger percentages of the image viewed by the user.
- As the user zooms into different quadrants, the criterion for overall discrepancy for a given image increases. This is to allow better detail and focus which was the sole reason of zooming in the first place.

Thus, resolution is dependent on the node level within an image as well as the number of zooms requested. The threshold may be thought of in the following way:

$$\text{Threshold} = f(\text{zoomlevel}, \text{nodelevel}) \quad \text{Eq. (4.4)}$$

The examples given before only merged if a set of four neighboring pixels were equivalent in the pixel color. So, there was no distortion whatsoever. Of course, this is at a price of large memory utilization and time. A more efficient but a slightly distorted image may be produced in retrospect. The original image of Fantasia with some distortion may look like Figure 4-6.



**Figure 4-6: A slightly distorted image of Fantasia**

#### **Section 4.4: Color Images**

The images shown earlier were depicted in grey scale even though they are originally defined in the red, green and blue spectrum of colors. Images are depicted by tuples of RGB (red, green and blue) for each pixel. Each pixel is made up of 24 bits to define the specific color. This may be an excessive amount of memory. One approach is to use a sin-

gle value which is similar to the luminosity found on television sets. This component represents the color image which is a weighted sum of the RGB values with the weights in decreasing order for G, R and B values. This is in accordance with the color sensitivities of the human eyesight [Sureschandran and Warter 1993]. The other components, chrominance, can be added with less precision than the luminosity.

So, one can develop a quadtree representation of the three components by a single weighted value like the luminosity used for televisions. Thus, the weighted sum is the sole criterion to drive the development of the quadtree. The advantage of this method is that there is one tree to code.

Some researchers have developed some algorithms for quadtree representations of images which build three different trees, one each for the R, G and B components. The merging thresholds for the three colors would be in the same proportion as the noticeable differences for these colors. Based on Cowlshaw (1985), the merging threshold values should be in the ratio of 4:1:4 or 4:1:8 for red, green and blue colors. In this method, there is the overhead of coding three different trees but there is more flexibility in exploiting the relative sensitivities of the human eye to the three colors. Color components may be merged better than grey scale images because color images compress much more easily. The latter approach, from some simple experimentation, will result in better quality images with the drawback that it requires a great deal more memory. The simulation developed has only one tree but allows for different criterion for each color. This feature is one of the unique characteristics of this work.

Quadtrees implemented for color images are a simple extension for grey scale ones. It is this reasoning that makes them quite attractive for color image processing on parallel mesh architectures. Much work has been performed in the area of computer hardware and image processing. This thesis combines and interrelates the seemingly disparate worlds in order to achieve an inherent relationship which takes advantage of each's qualities and developed algorithms.

## **Section 4.5: Summary**

Some of the simple techniques available to the user were described here. The simulator has the tradeoff of being quite time efficient or distortionless. The criterion for threshold which was used is quite similar to the one described in the previous chapter.

The image data ought to be available on some kind of large memory storage facility like a hard drive or CD-ROM which is to be accessed when needed by the simulator. The location of the required image data must be available beforehand in the manner of pointers to memory locations. This application is quite attractive in geographical information retrieval systems, virtual reality or space exploration.



## Chapter 5: Data Retrieval

Up to now, no discussion or elaboration of any hardware, parallelism or even I/O interface has been dealt with. So far, the actual simulation details have been elaborated in detail, examples of the functionality have been proposed. Also, some measure of time and memory utilization have been produced.

Considering that this simulation will be a user friendly system, it needs to perform in real-time. This basically refers to a reasonable amount of time for images to be implemented as quadrees and perceived by the user. This chapter will discuss architectural issues in the design of hardware for real time parallel manipulations of high bandwidth images.

### Section 5.1: Introduction

Graphic and multi-media user interfaces promote the use of computers for visualizing pixmap images. In many fields like medical imaging, geographical retrieval systems, biology and scientific modelling there is an *urgent need for a large amount of storage capacities, fast access and real-time interactive visualization of images.*

While processing power and memory capacity approximately double every two years, disk bandwidth only increases by about ten percent per year [Hersh 1993]. Interactive real-time visualization of full color pixmap image data requires a throughput of two to ten megabytes per second. Parallel input and output devices are required in order to access and manipulate, at high speed, image data distributed on a disk like CD-ROM. A high performance high capacity image system should provide processors located on a mesh topological network with a set of services for immediate access to images stored in memory. Basic services include real-time extraction of image parts for panning purposes, resampling for zooming in and out, browsing through three dimensional image cuts and accessing image sequences at the required resolution and speed.

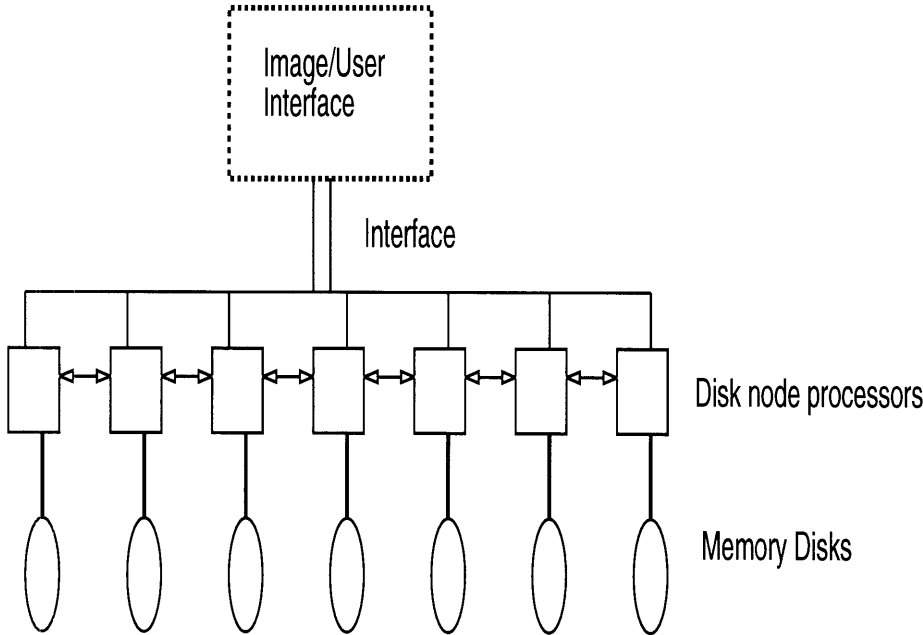
Previous research dealing with LAN's was focussed on increasing transfer rates between CPU and disks by using Redundant Arrays of Inexpensive Disks (RAID) [Patterson 1988]. Access to disk blocks was parallelized, but block and file management continued to be handled by a single CPU with limited processing power and memory bandwidth.

In order to access large quantities of image data at a throughput of two to ten megabytes per second, a multiprocessor multi-disk approach is proposed. In order to ensure high throughput, image extents are stored on a parallel array of disk nodes. Each disk node may include one disk node processor, cache memory and one large storage disk. This proposed architecture offers a low price-performance ratio since it is composed of standard low cost mass produced components such as processors, memory or disks. The processors in this case will be the ones developed for the NuMesh computer system at the Computer Architecture Group at the Massachusetts Institute of Technology.

This chapter will discuss how images can efficiently be distributed among disk nodes and a proposed system architecture. It will analyze their performance of the system according to various parameters such as the number of disk nodes, the size of the image and the communicating capabilities.

### Section 5.2: System Architecture

A system developed for image processing is made up of a server interface which includes the network interface, disk node processors used for disk access and an array of disks which are connected to the processors, Figure 5-1.



**Figure 5-1: A proposed image system architecture**

Image access performance is heavily influenced by the way in which images are distributed onto a disk array. User workstations like a Sun Sparc generally need rectangular image files so an image data file is partitioned into rectangular groups. To simplify the

file system managing parallel storage of data files on a number of disk nodes, groups are numbered sequentially from left to right and from top to bottom. The  $n$  disk memory nodes selected for storing an image file are defined from 0 to  $n-1$ . The image file data is mapped sequentially in modulo- $n$  mode to disk nodes.

### **Subsection 5.2.1: Paralleizing disk node accesses**

Of great interest to achieving good real-time behavior for the user is the number of channels available to the memory disks. A simulated architecture as the one described earlier where the server interface is linked by two or four channels to a number of interconnected channels is researched. To understand parallelism better, the behavior of an intelligent disk array in which disk node processors execute local processing operations in combination with disk accesses is analyzed.

For these purposes, a simple simulation using the C programming language and Matlab was written. It took into account time which is needed to compute the location of the needed data in the disk array, communication time between server and disk node processors, transfer of image through two or four channel links, receipt and assembly of the image by the server interface.

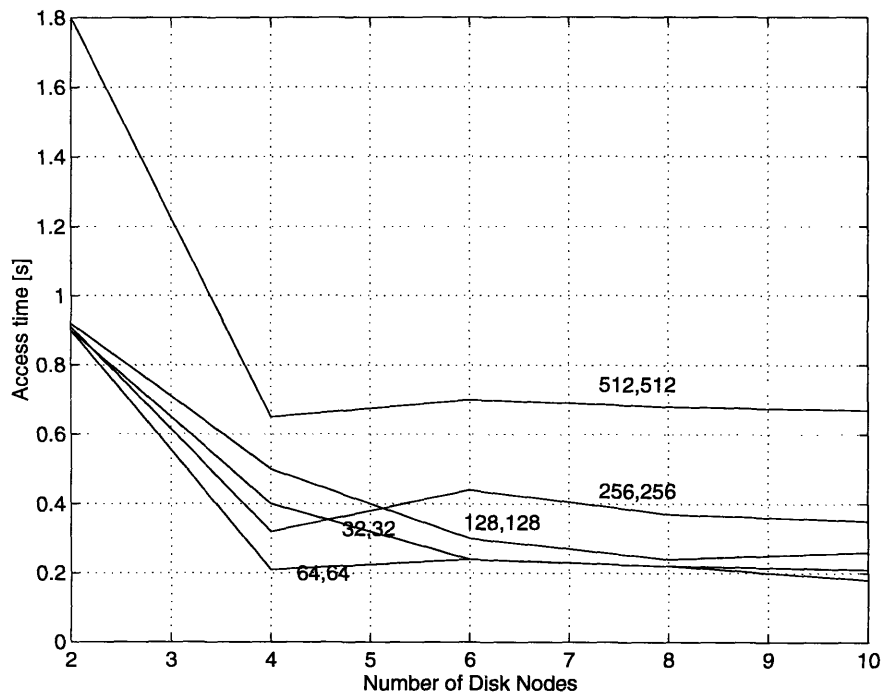
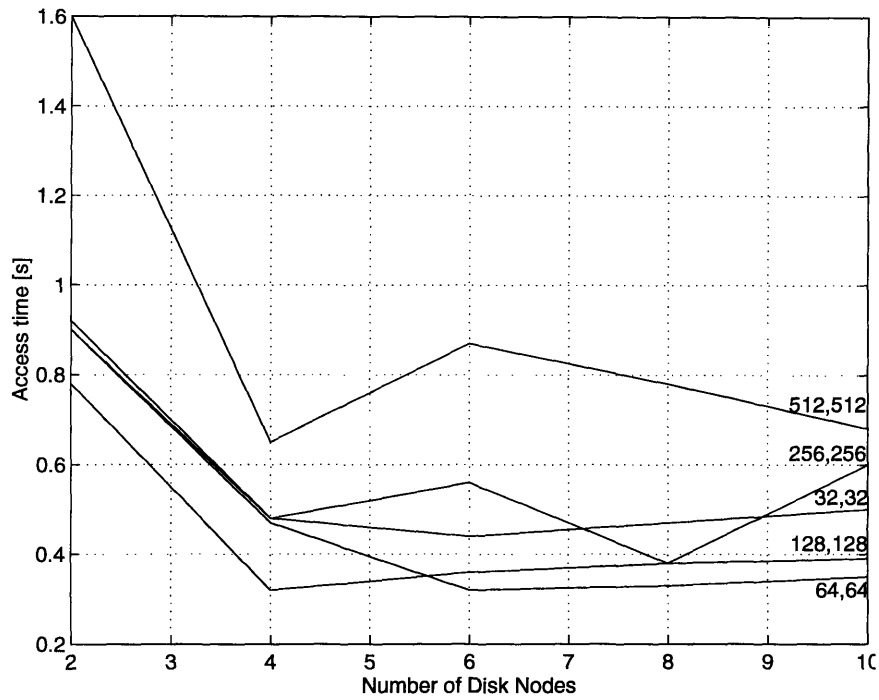
The following values were used in the simulation:

- Throughput of communication links: 1.6 Mbytes/sec
- Disk transfer rate: 2.4 Mbytes/sec
- Track to track access time: 4 msec
- Average rotation time: 8.3 msec

The simulation was carried out for normal 512 x 512 images, small 128 x 128 images, disk nodes numbering from 1 to 10 and either two or four communication links between the server interface processor and the disk nodes. Each image used 3 bytes per pixel to represent the red, green and blue components

Figure 5-2 shows that performance is directly proportional to available communication bandwidth. With two communication links, using two nodes produces a linear speedup for accessing images. With four communication links, using four nodes results in a linear speedup. It seems that the optimal image block size is either 64 x 64 or 128 x 128.

Disk node processors may be idle when image data extraction is occurring. This need not be the case because they may use their processing power to communicate with the server interface. Combining disk accesses and local processing operations, parallelization becomes very efficient and high speedups can be obtained with a relatively large number of disk nodes.



**Figure 5-2: Image access times for two and four communication links respectively**

The major bottleneck in the implementation is the limited communication through-

put between disk nodes and image server interface. The other option is to have a large array of disk nodes in direct memory access to the image server interface. In this case, the system's bandwidth is only limited by the throughput of the disks and by the processing ability of a single server interface processor. This approach's bottleneck stems from the limited ability of the interface to buffer the image data destined to the image server. As the disk nodes used in the architecture proposed increases, the overall performance will be much better than the latter one described. This is because the latter's bottleneck will inhibit a great transfer of data while the proposed will be able to pass the information to the interface with some delay.

### **Section 5.2.2: Storage Management**

Image and video files require large storage spaces. However, the size is not the only problem. Video objects are time-based continuous data streams of information. They must be delivered at a constant rate with bounded delay between source and target in order to preserve human perception. Continuous data stream handling capabilities are needed to provide real-time control and synchronization.

In a distributed multiprocessor system, the previous requirements must be achieved to provide an effective management of concurrent access both to storage devices and to single image or video files. A strategy for mass storage management is to take into account both requirements of image coding format:

- *Throughput*
- *Size of media unit*

and the qualities of the mass storage devices:

- *Capacity*
- *Transfer rate*
- *Access time*
- *Sector size*
- *Sector placement*

An optimal organization of mass storage represents the crucial issue for the realization of large concurrent archives. To achieve effective management, the architecture of the system has to fit the following requirements:

- *Scalability*: to cope with the progressive growth of an archive
- *Hierarchical structure*: to support storage management strategies that exploit different types of devices in order to achieve short access time and reduce storage costs
- *Adaptability*: to operate both in a local and a wide area environment

## **Section 5.3: Issues of parallel processors in real-time applications**

In recent years, massively parallel architectures have shown excellent performance in different scientific applications and image processing. This section will be a simple overview of the issues that are important in the application of multiple processors performing the same action (SIMD) in the same time unit. In the case of quadtrees, this will occur when different processors will be in charge of different segments of the image, e.g. quadrants, or even a single row of raster pixel data. Among the topics to be discussed are the requirements for effectively utilizing massively parallel processors and possible approaches to achieve these goals.

### **Section 5.3.1: Parallel Processing and Real-time Control**

Many times, massively parallel processors are deemed as changing compute time bounded problems into I/O time bounded problems. Actually, though, massively parallel processors also raise the issues of real time control. A SIMD hardware model emphasizes translation invariance and locality.

Image processing algorithms are often naturally translation invariant and local. These qualities are targeted by SIMD machines. However, the following, list of problems are raised:

- Achieving information transfer bandwidth as fast as the high processing capability of electronic parallel processors is developed
- Real time control
- Requiring high rates of information transfer even though the amount exchanged between the SIMD array and external processors is not much

The memory access problem has received considerable attention in past research. The extreme positions taken are either fully parallel access to external memory resulting in one extra pin per bit per processor or to have only connections internal to the processor array. This results in access time to external memory proportional to the number of processing elements (PE),  $x$ , along the perimeter of the mesh array. Between these extremes are the bus network solutions such as a mesh-of-tree-like router which has  $y$  PE's that are constructed as branches of a tree to go into a single node. This results in a constant access time for any processing element. The pin count and the overall access time thus become proportional to  $x/y$  and  $y$  where  $1 < y < x$ .

Real time control is important in most cognitive systems. Humans continually make decisions in real time. In computer systems, the issue raises itself in very high speed machines because the usual methods of program control dealt by the software is not a speed bottleneck for slower sequential machines. In visual cognition applications, for example, where data rates are high, solutions with hardware additions need to be found if

the parallel processor is not to be left idle while program changes and decisions are being performed. The controller ought to be developed with enough adaptability.

A simple controller can be developed as an extension of the Von Neumann machine. This basically refers to a hierarchical system in which higher levels represent strategists relative to the lower levels which are the tacticians. If delays are to be avoided completely, then each step of the implementation by the tactical component must be long enough for the strategist to be able to devise the next plan. By a similar token, the strategist plan cannot take too long on the part of the tacticians to implement, or be too detailed, or the strategists will be left idling. These considerations are independent of the requirement that the processors not be idle, which is to be regarded as a crucial requirement the controller must be able to fulfill.

There are two major qualities to this: software is reconfigurable in real time and dynamic memory management. They are needed to accomplish real time control on the mesh array of processing elements. Software reconfigurable in real time is needed to accomplish various levels of program switching and dynamic memory management is required because access to a large amount of memory is not possible on the mesh of processors.

Software that is reconfigurable in real time thus requires both global broadcasting and dynamic interpretation. Adaptive template matching is an example of dynamic reinterpretation. An initial template is applied to the data, the best matched value and its locations are extracted, using global broadcasting. Then, a new template is cut and passed back to the controller. The program is changed according to the new template using micro-level reinterpretation.

Memory management is a method to optimize memory usage. The limited predictability of memory requirements depending on the program paths followed often needs the memory management hardware to perform run time allocation and releasing of memory through symbolic addressing and data address manipulation. Because of the unpredictable need for memory, run time allocation and releasing of memory will be performed. This is the case when a user keeps zooming in and out of various areas of interest. In such a way, each image has been implemented as a quadtree structure and kept in memory just in case requests for it are reinitiated. However, as zooming and panning is augmented a large number of images are kept track of and a lack of memory may occur. In such a situation, some of the memory allocated for specific images may be released. The criterion for which ones to release may be similar to the criterion for releasing elements from a cache. One possible approach is least recently used (LRU) which may mean the highest level image in memory.

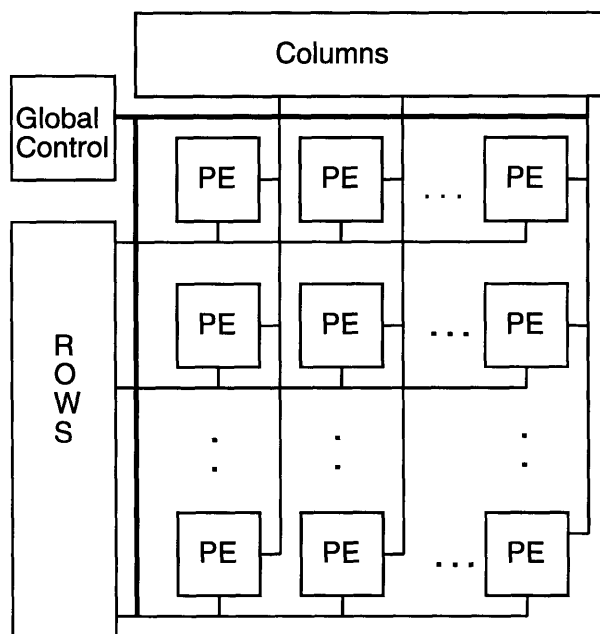
Handling the memory allocation and deallocation in software will not produce the mesh array of processors to function at its full clock speed. Therefore, a structured or modularized program is almost impossible since all of the local memory address within each subroutine is fixed. However, one possible way of alleviating the problem is by hav-

ing the software module pre-assign the addresses to local memory within each subroutine, prior to run-time. This has the disadvantage that the number of memory allocations as a result of conditional branching must not exceed the total memory available. If there are  $n$  times at which requests for image zooming or panning is performed, then the program has to make  $2^n$  passes so that all the memory addresses that are allocated at each user request can be assigned.

### Section 5.3.2: Communication of Processors with Image Interface

The problem of communication between the processors and the host needs to be addressed. Extracted information often needs to be exchanged between these units for further processing or decision making. For example, when windowing, the needed information may only be extracted from some of the processors in the array mesh. An efficient scheme of broadcasting information between the units is vital.

By providing row/column selectability, information from any row, column, or processing element can be sent to the host interface via external buffering. Figure 5-3 gives an example of this.



**Figure 5-3: A setup for the NuMesh processor array**

The question of what exactly does a processing element (PE) perform and what are its duties is to be discussed next. Well, let's assume that each one is in charge of a different segment of the image area and that no two PEs share any common pixels in their calculations. So, if the mesh array were a  $2 \times 2$  setup of PEs then each one would be responsible for one quarter of the image block, a  $4 \times 4$  setup then one-sixteenth of the image, etc. In this way, each PE can essentially develop a quadtree representation for its block of pixel



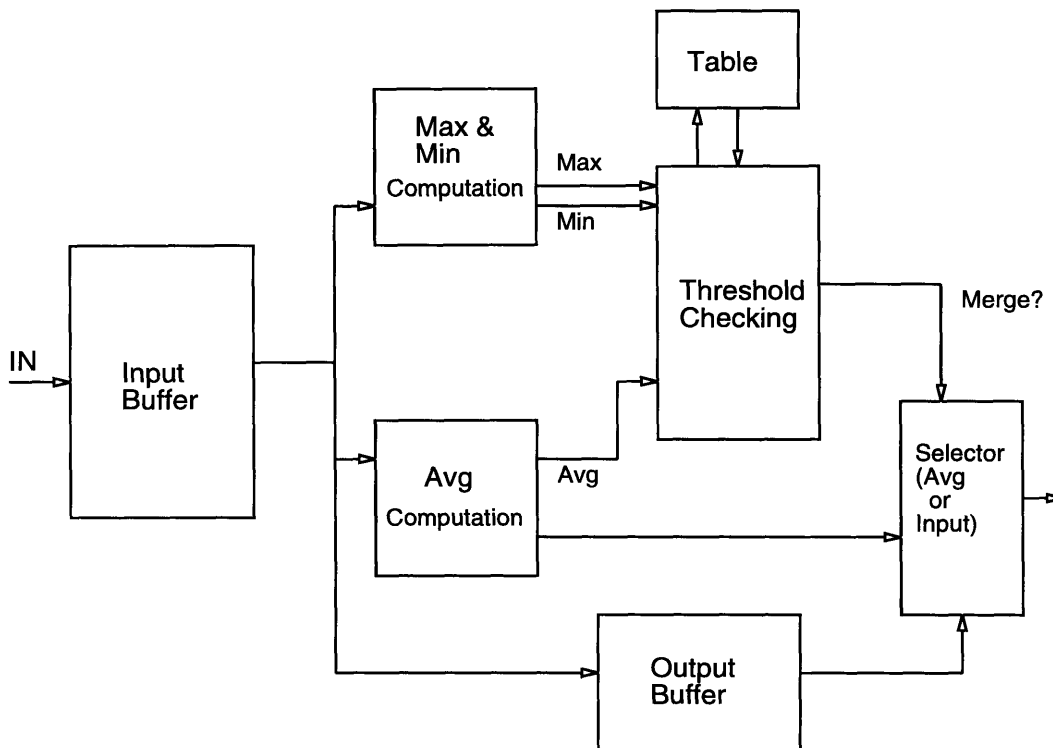
data which is then combined with the results obtained by neighbor processors to achieve an overall quadtree representation of the desired image to be perceived by the user.

For the sake of this discussion, let the threshold criterion be defined in the following way.

$$\text{Max-Min} \leq 2\text{Threshold(Average)} \quad \text{Eq. (5.1)}$$

Each PE processes groups of four elements received over two scan lines of the image and executes the merging criterion on them. Identifying the elements to compress is easily done in this scheme purely by timing, as the system has completely synchronous data flow. In case merging is successful, the elements are marked with a new header. If four elements of appropriate size are not encountered, merging should not be carried out. However, to keep the data flow timing regular, it is convenient to let the elements pass through the pipeline in this case too, only making sure that compression is not indicated.

The operations performed inside a PE are additions and subtractions, computation of the maximum and minimum of four values and table lookup. A typical architecture for the PE is shown in Figure 5-4.



**Figure 5-4: Internal architecture of a PE**

Serially, the computation of the maximum and minimum can be done in two different ways. One approach is to compute without buffering by using a network of two stages

of compare exchange switches which route the input streams towards the maximum and minimum locations. The other technique is to buffer the four elements beforehand and making use of simple logic based on the carry results of respective subtractions to identify the maximum and minimum.

The decoder for the architecture of the PE needs a buffer. This is to allow delay when the input stream is not merged and the average stream of data will be the output of the PE. So, if a merge is allowed then the average set of data will be passed; otherwise the original set of data is going to be used in the image representation.

### **Section 5.3.4: Summary**

Some of the detail dealing with input/output issues in parallelism were dealt in this chapter. In order to develop a system that may retrieve data from a given memory source, there are two options: using one processor which has sole control to accesses to memory but losing any parallelism in the application or using many processors which have to share retrievals from memory thus causing congestion and bandwidth problems. Ideally, one would like to use as many processors as possible in order to minimize time and take advantage of the mesh's hardware capabilities. However, this may be associated with a robust protocol to deal with memory congestion and contention of data retrieval.

# Chapter 6: Parallel Processing of Quadrees

## Section 6.1: Introduction

A quadtree is a hierarchical data structure that is useful for storing digital images. Let's review some of the concepts described in Chapter 2. Given an  $n \times n$  ( $n = 2^k$  for some  $k$ ) color image, its quadtree representation is a tree of degree four which can be defined in a top-bottom fashion. The root node of the tree represents the image. If the entire image has only one value, we label the root node with that value and stop. Otherwise, four descendants to the root node, representing the four quadrants of the image are added. The process is then repeated recursively for each of the four nodes. If a block has a single value, then its corresponding node is a leaf node; otherwise its node has four descendants which are called the NW, NE, SE and SW representing the northwest, northeast, southeast and southwest quadrants respectively. The four descendants are called siblings of each other. A node at level  $h < k$  is made up of a square block of size  $2^{k-h} \times 2^{k-h}$ . For this discussion, the term *node* and *block* will be used interchangeably.

A quadtree can be represented by explicitly storing the tree structure. But the space required for the pointers from a node to its children is not trivial [Samet and Tamminen 1985]. Moreover, it may take  $O(n)$  steps for a node to access a descendant on a  $n \times n$  mesh computer and parallel operations such as accessing neighbors in a given direction for all nodes can be very difficult. Thus, pointer-based representation may hinder processing on a mesh-connected computer. To keep the data in non-pointer notation, a preorder traversal of the nodes of the pointer quadtree ought to be performed. Transforming from a pointer-based quadtree to a pointer-less one is very simple and much of the work already discussed is valid. The resulting pointer-less representation is usually referred to as a linear quadtree.

Some parallel algorithms for computing properties of digital images can be found in [Dyer and Rosenfeld 1981, Stout 1987]. This chapter concerns processing quadtrees on a mesh computer. Yubin and Rosenfeld [1989] deal with processing of quadtrees for binary images on a mesh computer. Some of their work is expanded and enhanced in order to deal with parallel color data images.

This chapter will be divided into a few parts. The first will discuss in more detail pointer-less versus pointer-based quadtrees. The second deals with some simple parallel

algorithms which will be used later on. The third describes the format of a quadtree construction for a parallel quadtree, concurrent neighbor finding techniques and different techniques for routing data.

## Section 6.2: Pointer-less versus Pointer-based Quadtrees

Pointer-less quadtrees, at times, are less memory intensive than their counterpart. However, this is not true in all cases and this section will explain when and in what circumstances the pointer-less technique may be a better approach.

A pointer quadtree has two types of nodes: nonleaf and leaf. Non-leaf nodes consist of  $2^d$  pointers where  $d$  is the dimension of the image with a maximum side length of  $2^h$ . To distinguish between the two types, one bit of memory will be needed. This bit will be part of the pointer field that points at the node being described rather than in the node being described. No memory is attributed to the leaf nodes. The number of non-leaf nodes, where the leaves are  $L$ , is

$$\frac{L-1}{2^d-1} < \frac{L}{2^d-1} \quad \text{Eq. (6.1)}$$

Each pointer field only needs to be wide enough to differentiate among the numerous nodes in a particular tree. The possible number is

$$\left(1 + \frac{1}{2^d-1}\right) \times L \quad \text{Eq. (6.2)}$$

The number of bits needed to represent the quadtree is

$$\left(\frac{L}{2^d-1}\right) \times \left(2^d \times \left(1 + \log\left(L \times \frac{2^d}{2^d-1}\right)\right)\right) \quad \text{Eq. (6.3)}$$

which is the number of non-leaves \* (pointers \* (leaf bit + node)).

The linear quadtree would require bits on the order of

$$L \times (d \times h + \log(h+1)) \quad \text{Eq. (6.4)}$$

In order for a linear quadtree to be more compact than the pointer quadtree, the following relation must hold:

$$L \times (d \times h + \log(h + 1)) \leq \frac{L}{2^d - 1} \times \left( 2^d \times \left( 1 + \log \left( L \times \frac{2^d}{2^d - 1} \right) \right) \right) \quad \text{Eq. (6.5)}$$

Letting  $m = 2^d$  and  $n = 2^d - 1$ , L is equivalent to

$$L > \frac{n}{m} \times 2^{\frac{n}{m} \times (d \times h + \log(h + 1)) - 1} \quad \text{Eq. (6.6)}$$

As long as the number of leaf nodes, L, is less than or equal to the right side, the pointer quadtree requires at most as many bits as the linear quadtree. Let's define the value where the left side equals the right as the critical point.

The number of bits utilized must be an integer so this restriction changes the last equation to be

$$L > \frac{n}{m} \times 2^{\frac{n}{m} \times (d \times h + \lceil \log(h + 1) \rceil) - 2} \quad \text{Eq. (6.7)}$$

Practically, nodes in computer architecture start on byte boundaries. Assuming 8-bit bytes, let  $\{x\}$  denote  $8(\text{roundup}(x/8))$ . This changes relation 6.5 to

$$L \{d \times h + \lceil \log(h + 1) \rceil\} \leq \left( \frac{L}{2^d - 1} \right) \left\{ 2^d \times \left( 1 + \left\lceil \log \left( L \times \frac{2^d}{2^d - 1} \right) \right\rceil \right) \right\} \quad \text{Eq. (6.8)}$$

which using the above values for m and n results in

$$\left\{ m \left( 1 + \left\lceil \log \left( L \frac{m}{n} \right) \right\rceil \right) \right\} \leq 8 \left( m \frac{\left( 1 + \left\lceil \log \left( \frac{m}{n} L \right) \right\rceil \right)}{8} + 1 \right) \quad \text{Eq. (6.9)}$$

Finally, using the a simple approximation, the inequality results

$$\frac{n}{m} \times 2^{\frac{n}{m} \times \{d \times h + \lceil \log(h + 1) \rceil\} - \frac{8}{m} - 2} \leq L \quad \text{Eq. (6.10)}$$

Table 6-1 [Samet 1984] shows the cutoff values for equations 6.5 and 6.10 for

images whose sizes range from  $2^3 \times 2^3$  to  $2^{15} \times 2^{15}$ . The table also shows the maximum number of leaf nodes in the quadtree.

**Table 6-1: Leaf node counts**

Depth	Relation 6.5	Relation 6.10	Max number of leaves
3	24	3	64
4	80	192	256
5	260	192	1024
6	826	192	4096
7	2583	12288	16384
8	7981	12288	65536
9	24431	12288	262144
10	74221	12288	1048576
11	224085	786432	4194304
12	673021	786432	16777216
13	2012390	786432	67108864
14	5994178	786432	268435456
15	17794925	50331648	1073741824

Taking a look at the table, a quadtree of depth 9 must contain at least 12288 leaf nodes in order for the linear quadtree to be more compact than the corresponding pointer approach. Since, the maximum number of leaf nodes is 262144, the number of leaf nodes must be at least 4.7% of the maximum in order for the linear quadtree to be more compact for images of depth 9.

Putting this in perspective, a simple geographical image from [Scherson 1987] needed 5266 leaf nodes (2%). Thus, the pointer quadtree requires 12285 8-bit bytes while the linear quadtree needs 15798 8-bit bytes.

The significance of the comparison of the storage requirements of these alternative implementations lies in the issue of which approach is more likely to store a larger amount of quadtree nodes given some specified memory storage size. Of course, the size of the

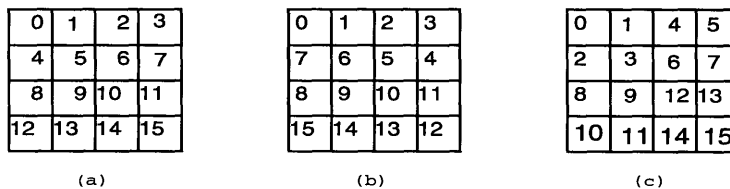
image and the resolution level will affect the number of nodes needed. It ought to be obvious that near the critical points, the two approaches are equally good. Much research is being performed now to implement parallel quadtrees in the linear approach but the outcome is quite uncertain. Believing that positive outcomes of the linear approach may result, the rest of this chapter will deal with parallel quadtrees in that manner. It should be remembered that once a simulation or code has been implemented in a pointer format, a linear quadtree may easily be obtained by an inorder traversal. This is what is done in the NuMesh simulation. The stored memory nodes for a given image are stored in a pointerless manner in order to minimize memory.

### Section 6.3: Basic parallel algorithms

Some simple parallel algorithms will be described. Their use will become apparent in the next section.

#### Sorting

Given a list of  $n^2$  elements on a  $n \times n$  mesh, assignment of processors ranging from 0 to  $n^2 - 1$  is done. The goal is for the elements to be in sorted order with the smallest in PE position (0,0). There are a number of ways to index the PEs. The most popular ones are row-major indexing, snake-like indexing and shuffled row-major indexing. If the mesh were an array of  $n \times n$  and given a PE with coordinates  $(r, c)$  would represent the one defined by  $r \times n + c$ ,  $r \times n + c'$  where  $c'$  equals  $c$  if  $r$  is even and  $n - c - 1$  otherwise. If the binary representation of  $r$  and  $c$  were  $r_{l-1}..r_1r_0$  and  $c_{l-1}..c_1c_0$  then the index of the PE will be equivalent to  $r_{l-1}c_{l-1}..r_1c_1r_0c_0$ . Examples of the three indexing schemes are shown in Figure 6-1.



**Figure 6-1: Indexing: a) row-major b) snake-like c) shuffled row-major**

#### Prefix

Given a sequence of  $m$  elements defined as  $a_0, a_1, \dots, a_{m-1}$  and an associative operator like '\*', the problem is to compute  $a_0, a_0*a_1, a_0*a_1*a_2, \dots, a_0*a_1*\dots*a_{m-1}$ . This problem has been worked on in great detail. If the number of elements,  $m$ , were  $n^2$  and they were stored in a row-major sorted order in the array then the prefixes can be computed in  $O(n)$  time. A binary tree approach would have a result of  $O(\log n)$  instead. The

algorithm can be altered to obtain parallel prefixes in column-major or shuffled major sorted order.

### Broadcast

Given a sequence of records such that records from the same sequence reside in consecutive PEs, assume that each sequence has a distinct label. The problem is to pass information from one record to the rest of the sequence. This can be done by a parallel algorithm in  $O(n)$ .

### Compression

Many algorithms for mesh architectures are naturally recursive. At each step, the amount of data decreases by some constant factor. Typically, the number of recursive steps is  $O(\log_a n)$  where  $a^2 = b > 1$  is the constant factor. Define the initial mesh architecture to be of 2D array of size  $n \times n$  which is also the size of the image under consideration. If the recursive step is performed in  $O(n)$  time units, the algorithm will take a total time of  $O(n \log_a n)$  to execute. In order to improve the performance of the algorithm, compression of the data after each recursive step will be performed to produce a smaller working mesh. This way, the algorithm will take  $O(n)$  time. A problem of size  $n^2$  can be solved on a mesh of size  $n \times n$ . Define  $T(n)$  to be the time for the algorithm to execute and assume that each recursive step including the compression part is performed in  $O(n)$  with the data being decreased at each step by a factor of  $b$ . Then,  $T(n) = O(n) + T(n/a) \sim O(n)$ . This compression may be found in [Hung and Rosenfeld 1986].

### Unshuffling

The objective is to get data in processors moved from shuffled row-major order to row-major indexing. This can be seen in Figure 6-2.

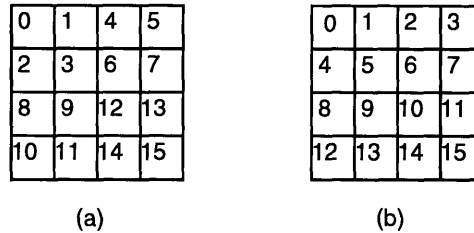
The original data in the PEs are relocated in such a way that they are stored in the same quadrant from left to right and from top to bottom according to the shuffled row-major index of the PE in which each element is located. For each horizontal pair of consecutive rows, the data in the rows are moved in such a way that successive rows in the same square become a single row of twice the length. In this way, the entire mesh will be unshuffled.

The algorithm merges horizontal pairs of  $m \times m$  squares where the  $m$ th merge ( $1 < m < \log n - 1$ ) will be processed with the given data,  $D$ , and some temporary variable defined as  $t$  in the following way:

- 1) For the PEs in the right square, move the data from  $D$  to  $t$ .
- 2) Move  $D$  in the odd rows of the left square  $m$  steps to the right. Store the result in  $D$ .
- 3) Move  $t$  in the even rows of the right square  $m$  steps to the left. Store the result in  $t$ .



- 4) Move D in both squares to the first  $m/2$  rows.
- 5) Move t in both squares to the last  $m/2$  rows. Then for these rows, move t back to D



**Figure 6-2: a) original mesh b) merging pairs of 2x2**

## Section 6.4: Building Quadrees

A quadtree is built by reading the image sequentially. Assume the image is a color  $n \times n$  image which is stored in a mesh architecture where one pixel is found per PE. As the quadtree is being processed, new nodes are created and old ones split or merge depending on the color of the pixel scanned and the configuration of the tree. This is exactly what was performed in the earlier description of Chapter 2.

A number of parallel algorithms exist. Dyer and Rosenfeld [1981] developed a bottom-up approach which builds a quadtree by merging blocks of the same color. Their work is expanded to allow utilization on a mesh connected processor setup like NuMesh. The data structure is defined like

$$\text{Block} = (\text{length}, \text{row}, \text{column}, \text{color}, \text{leaf}) \quad \text{Eq. (6.11)}$$

The *row* and *column* represent the position of the block in the image, *color* is the color of the block and *length* is the side length of the homogenous block having pixel (row, column) as its representative. Lastly, *leaf* is a boolean to signify whether the block is a leaf node of the final quadtree or not.

For the purposes of this discussion, each processor (PE) starts off with one pixel in its initial state. The following algorithm can be generalized with each PE being responsible for  $n$  consecutive pixels of the image as this may be more realistic and practical. For example, for a 1K x 1K image, it is ridiculous to expect 1 million processors to be utilized in representing an image into its quadtree representation.

The parallel quadtree building algorithm (QBA) algorithm proposed is as follows:

- 1) Each PE defines a block and initializes the length to 1, the row and column to its coordinates, color to the pixel color value and flag to be a non-leaf.
- 2) For each of the sibling blocks of size  $2^i \times 2^i$  where  $i$  ranges from 0 to  $\log n$ :
  - (2.1) The PE representing a  $2^i \times 2^i$  block initializes a local variable,  $x$ , to its pixel color if the entire block is of a homogenous color; otherwise -1 to signify that it is a non-leaf.
  - (2.2) The representatives of the NW, SW and SE blocks send  $x$  to the representative of the NW block. If the four  $x$ 's fit the merge criterion, the change the *length* field of the NW representative to  $2^{i+1}$ . Otherwise, set the leaf flag to be true for the four blocks with length  $2^i$ .

Hung and Rosenfeld [1989] proposed a different approach which utilizes a number of the functions described in Section 6.2.

- 1) Compute the length of pixel runs and the relative position of its constituents using the Prefix algorithm. Define  $m$  to be this value.
- 2) Broadcast the length of each run to its constituents.
- 3) Each PE computes the number of pixels that follow it in the same run. Define this as  $n$ .
- 4) Each PE determines the number of pixels in the maximal block its pixel can represent. Define this as  $o$ .
- 5) For  $o$  and  $n$ , each PE determines the number of pixels in the largest homogenous block its pixel represents.
- 6) Delete blocks which are contained in blocks of the same color. This is a merging criterion where the pixels need to be exactly the same. Whatever criterion may be used instead.
- 7) Compress the resulting blocks not deleted into PEs in the first rows of the mesh.

In order to find the maximal set or the perimeter of a block, techniques to find neighbors would be of great value. The most basic type of neighbor is one with a common edge. The sequential version to find neighbors in pointer-based quadtrees was described in Chapter 2.

Define a block,  $A$ , that has eastern neighbors of equal or smaller size. Let  $B$  be the topmost eastern neighbor. If one sorts the blocks in (row, column) order where the row and column will be the coordinates of the representative of the block, then block  $A$  will be the immediate predecessor to block  $B$ .  $B$  can obtain the needed information from  $A$  and passes this data to the other eastern neighbors of  $A$ , if they exist. All the eastern neighbors of  $A$  have the same column number so sort the blocks on (column, row).  $B$  will be the first of the run of eastern neighbors. If  $A$  had western neighbors, let  $C$  be the topmost western neighbor. After the first sort,  $A$  will be the immediate successor of  $C$  in the sorted list and is able to pass information to it. In the second run, if one sorts on (column +  $b$ , row) where  $b$  is the side length of a block, then the second sort will produce a run of western neigh-

bors. The north and south neighbors can be found very similarly. For finding the western neighbors, the procedure looks like:

Parallel Western Neighbor Procedure:

- Sort all blocks using (row, column).
- Each block examines its predecessor. If they have the same row and the predecessor has an equal or larger size, the block marks itself the leader and obtains the needed information (color, size, etc.). If they have the same row but the predecessor is smaller, the block marks itself as 'none' because it has no western neighbor of equal or greater size.
- Sort using (column, row).
- Each leader passes the information about the common western neighbor to the rest of the run.

### **Section 6.4.1: Routing Data**

Sorting, in the above procedures, was essentially used for routing purposes. Given a fast algorithm for sorting, it is usually easy to derive a fast algorithm for routing. This is because routing data from one PE to another is often trivial if the coordinates of the processors have been presorted. For more information about sorting algorithms, see [Leighton 1992].

Getting the needed data from one PE to another within a reasonable amount of time is one of the most challenging and important tasks of any large-scale general purpose parallel machine. This is because the processors comprising a parallel machine need to communicate with each other in a tightly constrained fashion in order to solve the image processing application in a realistic fashion. This may be expensive both in terms of the hardware and time.

Most parallel machines use some form of methodology in order to route data. One approach is the store-and-forward approach of routing where each data transfer consists of a packet that moves through a mesh architecture, one PE at a time, until it reaches its final destination. In general, one piece of data may pass between one node and another at any given time. Packet switching varies among architectures. Some will permit each node to only have one packet of data at a time while others will allow packets to pile up.

Another approach to routing data among processors is circuit-switching. This model of routing establishes and dedicates a specific path from the origin to the destination so that data may be transmitted as an uninterrupted stream of bits. Of course, it may not always be possible to establish disjoint paths through the mesh for all messages that need them and sometimes data will not be allowed to be sent.

Routing algorithms perform well if they route data from some given origins to

final destinations as quickly as possible using as small an amount of the network resources as possible. The degree to which an algorithm achieves a certain level of performance can be measured in several ways.

In a dynamic approach, the goal is to minimize the latency in sending each message to its final PE destination. This basically means that locality needs to be exploited in the algorithms used. Also, a state where messages can't move any further (deadlock) needs to be avoided.

In static scenarios, as the NuMesh one, the goal is to minimize the total time it takes to route all data to their final PE destinations. In static approaches where messages are dropped in circuit switching, the goal is to maximize the number of data transfers that successfully reach their destinations within a fixed amount of time. Much work has been accomplished for solving optimization problems to statically allocate resources on the NuMesh architecture [Minsky 1993].

### Section 6.4.2: NuMesh analysis

The NuMesh architecture is made up processors in a multi-dimensional setup. This parallel design consists of an array of processing elements (PEs) each connected to its neighbors. PEs can be enabled in ways such that instructions may be executed separately on each processor at a given time or information may be shared among a number of processors. There is some kind of control unit which dictates the instruction performed and the processors which are involved.

For this analysis, assume that the layout is two dimensional. If a higher one exists then a subset of the processors will actually be used. Also, let a color image of dimension  $n \times n$  be stored in the mesh, one pixel per PE. If the image is larger than the mesh, say of size  $m \times m$ , where  $m=kn$  for some integer  $k$ , then load  $k^2$   $n \times n$  subimages into different locations of the mesh. Build the quadtrees for the subimages separately. Note that further merging is possible if there is at least a set of four sibling  $n \times n$  blocks which satisfy the merging criterion.

The QBA described earlier will produce an  $O(n)$  time for completion for an  $n \times n$  mesh of processors. Step (2.1) decides whether a PE contains a representative of a  $2^i \times 2^i$  block. This can be determined by examining the least significant  $i$  bits of the binary representations of  $r$  and  $c$ . They must all be zero. In step (2.2), the quadrant of a  $2^i \times 2^i$  block is determined by the  $(i+1)^{\text{st}}$  bit of the row number and the column number of its representative. To be precise, let the two bits be  $a$  and  $b$ . Then the quadrant is NW, NE, SW and SE if the value of  $ab$  is 00, 01, 10 or 11 respectively. In step (2.2), passing data among the representative PEs can be achieved by  $O(2^i)$  horizontal and vertical shifts; the  $i$ th merging step takes only  $O(2^i)$  time. Consequently, this quadtree building algorithm can be completed in  $O(n)$  time.

The converse of building an image from its quadtree takes very similar time. Given a quadtree representation of an image on a  $n \times n$  mesh, the algorithm is simple. Dispatch each node to a PE according to the location of the upper left corner of the block it represents. Then, instruct all PEs in the same block of their color by a sequence of horizontal and vertical shifts. The time required for this algorithm is  $O(n)$ .

## Section 6.5: Summary

This chapter described a number of algorithms for generating and processing pointerless quadtree for digital images on a mesh architecture like NuMesh. These algorithms portray efficient ways that nodes can be represented and how using shuffled row-major sorting to index the pixels of the image.

Samet [1985] describes some algorithms that compute geometric properties without finding the neighbors. These algorithms use a data structure called the active border which keeps track of the state of the processing. But, this approach is very sequential in nature. Plus, in a mesh architecture, there is no shared memory in which to keep this data structure.

Many of the techniques utilized in parallel algorithms are quite suitable for image representations. Approaches for sorting, broadcasting, compressing, etc. are valid techniques in parallel architectures. Parallel processing of quadtrees was implemented in such a way to take advantage of these opportunities and hardware optimizations which are inherent on high speed multiprocessors and mesh architectures.

## Chapter 7: Conclusion

Hierarchical data structures are vital data representation techniques in areas such as computer graphics, image processing, computational geometry, geographic information systems, and robotics. There is an urgent need for large storage capacities, fast access and real-time interactive visualization of images. Through the emergence of workstations, image processing and GIS techniques for single users may be a great asset. Quadrees are a natural data structure to represent digital color images. An attractive characteristic is their ability to save space. A wide range of operations can be performed on them directly without having to transform them to the original digital color images. In addition, hierarchical data structures allow us to be able to focus on specified local areas since the data representation is based on decomposition. A number of key ideas about quadrees are important to remember:

- The characteristic that quadrees are recursive makes it an easy and very powerful tool.
- The number of nodes in a quadtree correspond directly to the resolution of the image. The level of allowable distortion may be achieved with a larger sized quadtree representation.

With the emergence of parallel multiprocessor machines, building parallel implementations of quadrees has been important. This is especially true on a mesh connected architecture like NuMesh. There has been some research on parallel algorithms for various machines which perform neighbor finding, parallel prefix and routing. Some of this work was extended to deal with a mesh architecture.

This work dealt with image decomposition on mesh architectures and the many systems issues involved. Because of the growing technological market in parallel and distributed systems and the natural interdisciplinary effects, this thesis combines a number of different yet related fields:

- Image Processing
- Geographic Information Systems
- Parallel Data Storage
- Parallel Algorithms for Mesh Architectures

Issues involving congestion, I/O bandwidth and real-time control were discussed in detail. In order to develop a system that may retrieve data from a given memory source, there are two options: using one processor which has sole control to accesses to memory but losing any parallelism in the application or using many processors which have to share retrievals from memory thus causing congestion and bandwidth problems. Ideally, one would like to use as many processors as possible in order to minimize time and take advantage of the mesh's hardware capabilities. Thus, memory accesses need to be set in a distributed fashion. This is to minimize time for data to be transmitted to the processors.

Methods of transforming a pixmap or raster file of data into a quadtree representation and vice versa were formulated. Threshold criterion was discussed in much detail. There are a couple of important things to take note of in this approach to image resolution:

- Within a given image, quadtree nodes closer to the root will have a more difficult criterion to merge.
- As the user zooms into different quadrants, the criterion for overall discrepancy for a given image increases.

Storage capacity versus distortion was the key element in formulating a tree representation from the original pixel depiction. Given a parallel mesh architecture, software and algorithms need to take advantage of that. For example, a sequential algorithm that attempts to locate a sibling may take  $O(n)$  time steps but in a parallel system this will produce large time delays. Realizing that certain algorithms like Prefix, Broadcast and Unshuffling are optimized in the hardware, simulation techniques may be adhered for a parallel mesh architecture.

Sample parallel techniques were described for the NuMesh architecture. Given an  $n \times n$  2D mesh of processors, the time to build a quadtree is  $O(n)$ . This result is also true of obtaining an image from a quadtree representation. This is quite good compared to past techniques which would take  $O(n)$  just to locate neighboring nodes. The total time involved will of course depend on the number of processors available in the mesh. As more processors are utilized more parallelism will result. Each processor will be able to handle its own distinct subimage of the original image.

To achieve success with a mesh architecture of processors, code must be able to take advantage of the hardware capabilities. The code should be partitioned into a tightly coupled set of computations exchanging data in fixed communication patterns. This is quite true of the code represented in this work for quadtree manipulations. It fits very well into the model of distributed computing and thus will take advantage of the inherent parallelism of the NuMesh architecture. Real-time image applications are very valuable and a mesh layout allows a high degree of parallelism and inter-processor communication.

# Appendices

## Appendix 1: User Interface

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

        /* Variable Declarations */
#define CHILDREN 4
#define DEPTH 16+4+1
#define MAXSIZE 1024
#define ROWS 256
#define WIDTH 256

        /* The node structure */
struct NODE {
    int pixel_red;
    int pixel_green;
    int pixel_blue;
    struct NODE *parent;
    struct NODE *NW;
    struct NODE *NE;
    struct NODE *SW;
    struct NODE *SE;
    int LEVEL;
};
typedef struct NODE Node;
typedef Node *node;

        /* The visual spectrum for the analysis */
node visual[DEPTH+1];
int maxvalues [DEPTH+1];

        /* The level of the visual spectrum we are perceiving */
int level = 1;

        /* Output file */
FILE *fptr;

        /* The pid number for a child process */
int pidnumber = 0;
```



```

        /* Function Prototypes */
int visual_parent (int);
int visual_child (int, int);
int fork(void);
extern inorder (node);
extern node quadtree;
extern void image_to_quadtree (int *);
extern void quadtree_to_image1 (node, int);
extern void print_quad_tree(node);
extern int maxvalue;

/* ===== */
void main ()
{
    int i,j, k, temp;
    int stop = 1;
    int choice;
    int array[ROWS][WIDTH][3];
    FILE *infile;
    int counter = 0;

    infile = fopen (".data", "r");

    /* Reading in a file made up of a 256 x 256 image */
    for (i=0; i < 256; i++)
        for (j=0; j < 256; j++)
            for (k=0; k < 3; k++) {
                fscanf (infile, "%d", &temp);
                array[i][j][k] = temp;}

    fclose (infile);

    image_to_quadtree (&array[0][0][0]);
    visual[level] = quadtree;

    fptr = fopen ("/tmp/output8.pnm", "w");
    quadtree_to_image1 (visual[level], ceil((log10 (WIDTH)) / 0.30103));
    fclose (fptr);
    system ("xv /tmp/output8.pnm -geometry 256x256+200+200 ");

    maxvalues[level] = maxvalue;

    while (!stop) {
        printf ("\n\nThe Following are your choices in this Visual Exploration: \n");
        printf (" 1. Focus on the NW Quadrant\n");
        printf (" 2. Focus on the NE Quadrant\n");
        printf (" 3. Focus on the SW Quadrant\n");
    }
}

```

```

printf (" 4. Focus on the SE Quadrant\n");
printf (" 5. Go up one level\n");
printf (" 6. Exit the VE\n");
printf ("Your choice is: ");
scanf ("%d", &choice);

if (choice == 1) {
    level = visual_child (level, 1);
    quadtree = NULL;
    image_to_quadtree (&data1[0][0][0]); }

if (choice == 2) {
    level = visual_child (level, 2);
    quadtree = NULL;
    image_to_quadtree (&data2[0][0][0]); }

if (choice == 3) {
    level = visual_child (level, 3);
    quadtree = NULL;
    image_to_quadtree (&data3[0][0][0]); }

if (choice == 4) {
    level = visual_child (level, 4);
    quadtree = NULL;
    image_to_quadtree (&data4[0][0][0]); }

visual[level] = quadtree;

if (choice == 5)
    if (level != 1) {
        level = visual_parent (level);
        maxvalue = maxvalues[level];
    }

if (choice == 6) stop = 1;

if (choice != 6) {
    maxvalues[level] = maxvalue;
    fptr = fopen ("output1.pnm", "w");
    quadtree_to_image1 (visual[level], 3);
    fclose (fptr);
    system ("xv output1.pnm -geometry 256x256+200+200 ");
    print_quad_tree (visual[level]); }

}

}

/* Finds the jth child of node i; j >=1 */
/* NW = 1; NE = 2; SW = 3; SE = 4 */
int visual_child (int i, int j)

```

```

{
    return (CHILDREN * (i-1) + j + 1);
}

/* Finds the parent node to node i */
int visual_parent (int i)
{
    return ((int) floor( (i-2+CHILDREN) / (double)CHILDREN));
}

```

## Appendix 2: Simulation

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MAXSIZE 1024      /* Max size of raster */
#define ROWS 256         /* Number of Rows of raster */
#define WIDTH 256        /* Number of Columns of raster */
#define T1 80            /* Parameter needed for merge criterion */

/* The node structure */
struct NODE{
    int pixel_red;        /* The pixel is divided into Red, Green */
    int pixel_green;     /* and blue parameters. */
    int pixel_blue;
    struct NODE *parent; /* parent node */
    struct NODE *NW;     /* The four children of a node */
    struct NODE *NE;
    struct NODE *SW;
    struct NODE *SE;
    int LEVEL;           /* hierarchial level of a node in its binary tree */
};
typedef struct NODE Node;
typedef Node *node;     /* pointer to the NODE structure */

/* Function Prototypes */
extern void image_to_quadtree (int *);
void row (int *, int [WIDTH][3], int);
node create_node (node, char *, int, int, int);
void oddrow (int, int [WIDTH][3], node);

```

```

node evenrow(int, int, int, int [WIDTH][3], node);
node add_edge_neighbor (node , char *, int, int, int);
void merge (int, int, node);
node child (node, char []);
node father (node);
char * opedge (char *);
char * cedge (char *);
char * ccedge (char *);
void print_quad_tree(node);
char * opquad (char direction[]);
char * cquad (char direction[]);
char * ccquad (char direction[]);
char * quad (char *, char *);
int adj (char *, char *);
char * reflect (char *, char *);
char * sontype (node);
extern void quadtree_to_image1 (node, int);
void find_2d_block (node *, int, int, int, int, int *);
int get_quadrant (int, int, int, int);
void out_row (node, int, int);
void qt_gteq_edge_neighbor2 (node, char *, node *, int *);
void output_run (int, int, int, int);
void inorder (node);
int merging_ok (node, double *, double *, double *);
int find_max(int *);

```

```
extern FILE *fptr;
```

```
node quadtree=NULL; /*quadtree for the root of the tree
representing the image data. */
```

```
extern level; /* level of image interested in */
```

```
int xf[4] = { 1, 0, 1, 0 };
int yf[4] = { 1, 1, 0, 0 };

```

```
int maxvalue;
```

```
void image_to_quadtree (int *data)
```

```
/* Construct a quadtree corresponding to the image whose bin representation is contained in a list
of rows, WIDTH pixels wide, pointed at by P. Procedure ADD_EDGE_NEIGHBOR uses global
variable NEWROOT to keep track of the root of the quadtree as it is built in procedures ADD_-
EDGE_NEIGHBOR and CREATE_QNODE. MAXSIZE is the maximum length of a side of an im-
age for which the quadtree can be built and must be a power of two. It is used to enable merging
white nodes at the extreme right and bottom of an image that is not a square and of side length that
is a power of two */
```

```
{
    int Q[WIDTH][3];
    int i, j;
```

```

node first = NULL;

for (i =0; i<3; i++)
  for (j=0; j<WIDTH; j++)
    Q[j][i] = 0;

maxvalue = find_max (data);

row (data, Q, 0); /* get the first row of pixel data */

    /* create the first node for tree */
first = create_node (NULL, NULL, Q[0][0], Q[0][1], Q[0][2]);

odddrow (WIDTH, Q, first); /* create the first row of data */

i = 1;

row (data, Q, i);

i = i + 1;

/* call evenrow to place the next row in the tree */
first = evenrow (i, WIDTH, (i>ROWS), Q, add_edge_neighbor(first,"S",0,0,0));

/* loop for all the rows left to add into the tree structure */
while (i<ROWS) {
  row (data, Q, i);
  oddrow (WIDTH, Q, first);
  i = i + 1;
  row (data, Q, i);
  i = i + 1;
  first = evenrow (i, WIDTH, (i >= ROWS), Q, add_edge_neighbor(first,"S",0,0,0));
}
}

```

*void row (int \*data, int Q[WIDTH][3], int i)*

*/\* This function obtains a row of data from the array DATA and places it into the array Q. The row value is defined by the variable I \*/*

```

{
  int counter;

  for (counter = 0; counter < WIDTH; counter++)
  {
    Q[counter][0] = *(data + counter*3 + i*WIDTH*3);
    Q[counter][1] = *(data + counter*3 + i*WIDTH*3 + 1);
    Q[counter][2] = *(data + counter*3 + i*WIDTH*3 + 2);
  }
}

```

*node create\_node (node root, char \*t, int red, int green, int blue)*

*/\* Create a node with color red,green and blue corresponding to child T of node ROOT and return a pointer to it. When ROOT is nil, the transmitted actual paramter value corresponding to T is nil and ignored. \*/*

```
{
    node p = NULL;

    /* allocate a memory node for variable p */
    p = (struct NODE *) malloc (sizeof(struct NODE));

    /* depending on which child defined by variable T, that subchild
       of node ROOT is created */
    if (root != NULL){
        if (strcmp (t, "NW") == 0)
            root->NW = p;
        else if (strcmp (t, "NE") == 0)
            root->NE=p;
        else if (strcmp (t, "SW") == 0)
            root->SW=p;
        else if (strcmp (t, "SE") == 0)
            root->SE=p;}

    /* Created node has parent */
    p->parent = root;
    p->pixel_red = red;
    p->pixel_green = green;
    p->pixel_blue = blue;
    p->LEVEL = 0;

    p->NW = NULL;
    p->NE = NULL;
    p->SE = NULL;
    p->SW = NULL;

    return p; /* return the created node */
}
```

*void oddrow (int w, int Q[WIDTH][3], node root)*

*/\* Add an odd-numbered row of width W, represented by Q, to a quadtree whose node ROOT corresponds to the first pixel in the row \*/*

```
{
    int i;

    root->pixel_red = Q[0][0];
```

```

root->pixel_green = Q[0][1];
root->pixel_blue = Q[0][2];
root->LEVEL = 0;

/* create a new node for each pixel and add it to the tree */
for (i = 1; i < w; i++) {
    root = add_edge_neighbor (root, "E", 0,0,0);
    root->pixel_red = Q[i][0];
    root->pixel_green = Q[i][1];
    root->pixel_blue = Q[i][2];
}
}

```

*node evenrow(int i, int w, int lastrow, int Q [WIDTH][3], node first)*

*/\* Add row I, an even numbered row of width W represented by Q, to a quadtree whose node FIRST corresponds to the first pixel in the row. During this process, merges of nodes having four sons of the same color are performed. The value returned is a pointer to the node corresponding to the first pixel in row I+1 unless row I is the last row (denoted by LASTROW) in which the value returned is irrelevant. For the last column in the row, merging is attempted for the white nodes at the extreme right of the image by setting the column to MAXSIZE. For the last row in the image, merging is attempted for the white nodes at the bottom of the image by setting the row number to MAXSIZE \*/*

```

{
    int j;
    node p,r;

    p = first;

    if (!lastrow) /* Remember the first node of the next row */
        first = add_edge_neighbor (p, "S", 0,0,0);
    else
        i = MAXSIZE; /* Enable merging white nodes on the bottom of image */

    for (j = 0; j <= (w-2); j++) {
        r = add_edge_neighbor (p, "E", 0,0,0);
        p->pixel_red = Q[j][0];
        p->pixel_green = Q[j][1];
        p->pixel_blue = Q[j][2];
        father(p)->LEVEL = p->LEVEL + 1;

        if (((j-1)% 2) == 0)
            merge (i, j-1, father(p));
        p = r;
    } /* Don't invoke add_edge_neighbor to the extreme right of the image */

    p->pixel_red = Q[WIDTH - 1][0];
    p->pixel_green = Q[WIDTH - 1][1];
}

```

```

p->pixel_blue = Q[WIDTH - 1][2];

    /* Merge white nodes at the extreme right of the image */
    merge (i, MAXSIZE, father(p));

    return (first);
}

```

*node add\_edge\_neighbor (node Q, char \* direction, int red, int green, int blue)*

*/\* Return a pointer to a node corresponding to the pixel that is adjacent in the direction of DIRECTION to the pixel represented by node Q. This is done by finding the nearest common ancestor of the two nodes and creating one if it does not exist. Whenever a nearest common ancestor or other nodes are created, the color of all created sons is set to PIXEL. They are later reset to gray, black or white as appropriate. NEWROOT is used to keep track of the root of the quadtree as it is constructed. \*/*

```

{
    node p;
    char sontypeofQ[2];
    node up, up2;

    if (Q->parent == NULL) { /* Nearest common ancestor does not exist */
        p = create_node (NULL, NULL, -1, -1, -1);
        quadtree = p;
        Q->parent = p;
        p->pixel_red = -1;
        p->pixel_green = -1;
        p->pixel_blue = -1;
        p->LEVEL = Q->LEVEL + 1;

        strcpy(sontypeofQ, quad(ccedge(direction), opedge(direction)));

        /* Depending on which subdivision defined by SONTYPEOFQ, that child
           is created and defined by the node Q */
        if (strcmp (sontypeofQ, "NW") == 0)
            p->NW = Q;
        else if (strcmp (sontypeofQ, "NE") == 0)
            p->NE = Q;
        else if (strcmp (sontypeofQ, "SW") == 0)
            p->SW = Q;
        else if (strcmp (sontypeofQ, "SE") == 0)
            p->SE = Q;

        /* Create 3 children */
        create_node (p, opquad(sontypeofQ), red, green, blue);
        create_node (p, opquad(reflect(direction,sontypeofQ)), red, green, blue);
        return (create_node(p, reflect(direction, sontypeofQ), red, green, blue));
    } /* Parent is null ? */
}

```



```

    /* if parent exists */
else if (adj(direction,sontype(Q)))
    p = add_edge_neighbor(father(Q), direction, red, green, blue);
else p = father(Q);

/* Trace a path from the nearest common ancestor to the adjacent node creating
white children and relabeling nonleaf nodes to gray as needed */

if (child (p, reflect(direction,sontype(Q))) == NULL) {
    p->pixel_red = -1;
    p->pixel_green = -1;
    p->pixel_blue = -1;

    create_node (p, "NW", red, green, blue);
    create_node (p, "NE", red, green, blue);
    create_node (p, "SW", red, green, blue);
    create_node (p, "SE", red, green, blue);

    p->LEVEL = child(p, "NW")->LEVEL + 1;

    up2 = p;
    while (up2 != quadtree) {
        up = up2->parent;
        up->LEVEL = up2->LEVEL + 1;
        up2 = up;
    }

} /* if child is NULL */

return (child (p,reflect(direction,sontype(Q))));
}

int merging_ok (node p, double *avg_red, double *avg_green, double *avg_blue)
/* Function to decide whether to merge four children to one parent or not */

{
    double RESOLUTION;

    RESOLUTION = (double) pow (2, - level) * (1.0/(p->NW->LEVEL) * T1;

    /* Find the mean of the four children in order to compare each child's relative variation
from the mean */
    *avg_red = (child(p, "NW")->pixel_red + child(p, "NE")->pixel_red
                + child(p, "SW")->pixel_red + child(p, "SE")->pixel_red) / 4.0;
    *avg_green = (child(p, "NW")->pixel_green + child(p, "NE")->pixel_green
                 + child(p, "SW")->pixel_green + child(p, "SE")->pixel_green) / 4.0;
    *avg_blue = (child(p, "NW")->pixel_blue + child(p, "NE")->pixel_blue

```

```

+ child(p, "SW")->pixel_blue + child(p, "SE")->pixel_blue) / 4.0;

```

*/\* If the relative difference of each pixel from the mean is less than or equal to the desired resolution value then we may merge. The higher the resolution the less detailed and inaccurate the image becomes \*/*

```

if ( (child(p, "NW")->pixel_red != -1) && (child(p, "NE")->pixel_red != -1) &&
    (child(p, "SW")->pixel_red != -1) && (child(p, "SE")->pixel_red != -1) ) {
    if ( (fabs (*avg_red - child(p, "NW")->pixel_red) <= RESOLUTION) &&
        (fabs (*avg_red - child(p, "NE")->pixel_red) <= RESOLUTION) &&
        (fabs (*avg_red - child(p, "SW")->pixel_red) <= RESOLUTION) &&
        (fabs (*avg_red - child(p, "SE")->pixel_red) <= RESOLUTION) &&
        (fabs (*avg_green - child(p, "NW")->pixel_green) <= RESOLUTION) &&
        (fabs (*avg_green - child(p, "NE")->pixel_green) <= RESOLUTION) &&
        (fabs (*avg_green - child(p, "SW")->pixel_green) <= RESOLUTION) &&
        (fabs (*avg_green - child(p, "SE")->pixel_green) <= RESOLUTION) &&
        (fabs (*avg_blue - child(p, "NW")->pixel_blue) <= RESOLUTION) &&
        (fabs (*avg_blue - child(p, "NE")->pixel_blue) <= RESOLUTION) &&
        (fabs (*avg_blue - child(p, "SW")->pixel_blue) <= RESOLUTION) &&
        (fabs (*avg_blue - child(p, "SE")->pixel_blue) <= RESOLUTION) )
        return 1;
    else return 0; }
else return 0;

```

```

}

```

*void merge (int i, int j, node p)*

*/\* Attempt to merge a node having four sons of the same color starting with node P at row I and column J. \*/*

```

{

```

```

    double avg_red = 0.0;
    double avg_green = 0.0;
    double avg_blue = 0.0;
    int keepit = 1;

```

*/\* If the children all have the same values then can merge them \*/*

```

while ( ((i % 2) == 0) && ((j % 2) == 0) &&
    merging_ok (p, &avg_red, &avg_green, &avg_blue) && keepit) {

```

```

    i = i/2;
    j = j/2;

```

*/\* The average value of the childrens' pixels is replacing the parent's old value. This is not to allow extreme values of the childrens' pixels \*/*

```

    p->pixel_red = (int) avg_red;
    p->pixel_green = (int) avg_green;
    p->pixel_blue = (int) avg_blue;

```

```

    /* Free all the children to this parent due to merging */
    free (child(p, "NW"));
    free (child(p, "NE"));
    free (child(p, "SW"));
    free (child(p, "SE"));

    /* Make sure all the children of the parent are defined to be nil */
    p->NW = NULL;
    p->NE = NULL;
    p->SE = NULL;
    p->SW = NULL;

    p = p->parent;

    if ( merging_ok(p, &avg_red, &avg_green, &avg_blue) && ((j % 2) != 0) )
        j++;
    if (p == quadtree)
        keepit = 0;

} /* while stmt */
}

```

*char \* opedge (char \*direction)*

*/\* This function returns the opposite edge from the DIRECTION edge given \*/*

```

{
    if (strcmp (direction, "E") == 0)
        return "W";
    else if (strcmp (direction, "W") == 0)
        return "E";
    else if (strcmp (direction, "N") == 0)
        return "S";
    else if (strcmp (direction, "S") == 0)
        return "N";
    else return "NULL";
}

```

*char \* cedge (char \*direction)*

*/\* This function returns the opposite edge from the DIRECTION edge given \*/*

```

{
    if (strcmp (direction, "N") == 0)
        return "E";
    else if (strcmp (direction, "E") == 0)
        return "S";
    else if (strcmp (direction, "W") == 0)
        return "N";
}

```

```

        else if (strcmp (direction, "S") == 0)
            return "W";
        else return "NULL";
    }

```

```
char * ccedge (char *direction)
```

```
/* This function returns the opposite edge from the DIRECTION edge given */
```

```

{
    if (strcmp (direction, "N") == 0)
        return "W";
    else if (strcmp (direction, "E") == 0)
        return "N";
    else if (strcmp (direction, "W") == 0)
        return "S";
    else if (strcmp (direction, "S") == 0)
        return "E";
    else return "NULL";
}

```

```
char * opquad (char direction[])
```

```
/* This function returns the opposite quad from the DIRECTION edge given */
```

```

{
    if (strcmp (direction, "NW") == 0)
        return "SE";
    else if (strcmp (direction, "SE") == 0)
        return "NW";
    else if (strcmp (direction, "NE") == 0)
        return "SW";
    else if (strcmp (direction, "SW") == 0)
        return "NE";
    else return "NULL";
}

```

```
char * cquad (char direction[])
```

```
/* This function returns the opposite quad from the DIRECTION edge given */
```

```

{
    if (strcmp (direction, "NW") == 0)
        return "NE";
    else if (strcmp (direction, "SE") == 0)
        return "SW";
    else if (strcmp (direction, "NE") == 0)
        return "SE";
    else if (strcmp (direction, "SW") == 0)

```

```

        return "NW";
    else return "NULL";
}

char * ccquad (char direction[])
/* This function returns the opposite quad from the DIRECTION edge given */

{
    if (strcmp (direction, "NW") == 0)
        return "SW";
    else if (strcmp (direction, "SE") == 0)
        return "NE";
    else if (strcmp (direction, "NE") == 0)
        return "NW";
    else if (strcmp (direction, "SW") == 0)
        return "SE";
    else return "NULL";
}

/* PRINTS tree */
void print_quad_tree(node root)
{
    if(root == NULL)
        return;
    else
    {
        printf("\npixel = %d %d %d level = %d root = %d parent = %d",
            root->pixel_red, root->pixel_green, root->pixel_blue, root->LEVEL,
            root, root->parent);
        print_quad_tree(root->NW);
        print_quad_tree(root->NE);
        print_quad_tree(root->SW);
        print_quad_tree(root->SE);
    }
}

char * quad (char *a, char *b)
{
    if ((strcmp (a, "E") == 0) && (strcmp (b, "N") == 0))
        return "NE";
    else if ((strcmp (a, "E") == 0) && (strcmp (b, "S") == 0))
        return "SE";
    else if ((strcmp (a, "W") == 0) && (strcmp (b, "N") == 0))
        return "NW";
}

```

```

else if ((strcmp (a, "W") == 0) && (strcmp (b, "S") == 0))
    return "SW";
else if ((strcmp (b, "E") == 0) && (strcmp (a, "N") == 0))
    return "NE";
else if ((strcmp (b, "E") == 0) && (strcmp (a, "S") == 0))
    return "SE";
else if ((strcmp (b, "W") == 0) && (strcmp (a, "N") == 0))
    return "NW";
else if ((strcmp (b, "W") == 0) && (strcmp (a, "S") == 0))
    return "SW";
else return "NULL";
}

```

*int adj (char \*a, char \*b)*

```

{
    if (((strcmp (a, "N") == 0) && (strcmp (b, "NW") == 0)) ||
        ((strcmp (a, "N") == 0) && (strcmp (b, "NE") == 0)) ||
        ((strcmp (a, "E") == 0) && (strcmp (b, "NE") == 0)) ||
        ((strcmp (a, "E") == 0) && (strcmp (b, "SE") == 0)) ||
        ((strcmp (a, "S") == 0) && (strcmp (b, "SE") == 0)) ||
        ((strcmp (a, "S") == 0) && (strcmp (b, "SW") == 0)) ||
        ((strcmp (a, "W") == 0) && (strcmp (b, "SW") == 0)) ||
        ((strcmp (a, "W") == 0) && (strcmp (b, "NW") == 0)) ||
        ((strcmp (a, "NW") == 0) && (strcmp (b, "NW") == 0)) ||
        ((strcmp (a, "NE") == 0) && (strcmp (b, "NE") == 0)) ||
        ((strcmp (a, "SW") == 0) && (strcmp (b, "SW") == 0)) ||
        ((strcmp (a, "SE") == 0) && (strcmp (b, "SE") == 0)))
        return 1;
    else return 0;
}

```

*char \* reflect (char \*a, char \*b)*

```

{
    if ((strcmp (a, "N") == 0) && (strcmp (b, "NW") == 0))
        return "SW";
    else if ((strcmp (a, "N") == 0) && (strcmp (b, "NE") == 0))
        return "SE";
    else if ((strcmp (a, "N") == 0) && (strcmp (b, "SW") == 0))
        return "NW";
    else if ((strcmp (a, "N") == 0) && (strcmp (b, "SE") == 0))
        return "NE";
    else if ((strcmp (a, "E") == 0) && (strcmp (b, "NW") == 0))
        return "NE";
    else if ((strcmp (a, "E") == 0) && (strcmp (b, "NE") == 0))
        return "NW";
    else if ((strcmp (a, "E") == 0) && (strcmp (b, "SW") == 0))
        return "SE";
}

```

```

else if ((strcmp (a, "E") == 0) && (strcmp (b, "SE") == 0))
    return "SW";
else if ((strcmp (a, "S") == 0) && (strcmp (b, "NW") == 0))
    return "SW";
else if ((strcmp (a, "S") == 0) && (strcmp (b, "NE") == 0))
    return "SE";
else if ((strcmp (a, "S") == 0) && (strcmp (b, "SW") == 0))
    return "NW";
else if ((strcmp (a, "S") == 0) && (strcmp (b, "SE") == 0))
    return "NE";
else if ((strcmp (a, "W") == 0) && (strcmp (b, "NW") == 0))
    return "NE";
else if ((strcmp (a, "W") == 0) && (strcmp (b, "NE") == 0))
    return "NW";
else if ((strcmp (a, "W") == 0) && (strcmp (b, "SW") == 0))
    return "SE";
else if ((strcmp (a, "W") == 0) && (strcmp (b, "SE") == 0))
    return "SW";
else if ((strcmp (a, "NW") == 0) && (strcmp (b, "NW") == 0))
    return "SE";
else if ((strcmp (a, "NW") == 0) && (strcmp (b, "NE") == 0))
    return "SW";
else if ((strcmp (a, "NW") == 0) && (strcmp (b, "SW") == 0))
    return "NE";
else if ((strcmp (a, "NW") == 0) && (strcmp (b, "SE") == 0))
    return "NW";
else if ((strcmp (a, "NE") == 0) && (strcmp (b, "NW") == 0))
    return "SE";
else if ((strcmp (a, "NE") == 0) && (strcmp (b, "NE") == 0))
    return "SW";
else if ((strcmp (a, "NE") == 0) && (strcmp (b, "SW") == 0))
    return "NE";
else if ((strcmp (a, "NE") == 0) && (strcmp (b, "SE") == 0))
    return "NW";
else if ((strcmp (a, "SW") == 0) && (strcmp (b, "NW") == 0))
    return "SE";
else if ((strcmp (a, "SW") == 0) && (strcmp (b, "NE") == 0))
    return "SW";
else if ((strcmp (a, "SW") == 0) && (strcmp (b, "SW") == 0))
    return "NE";
else if ((strcmp (a, "SW") == 0) && (strcmp (b, "SE") == 0))
    return "NW";
else if ((strcmp (a, "SE") == 0) && (strcmp (b, "NW") == 0))
    return "SE";
else if ((strcmp (a, "SE") == 0) && (strcmp (b, "NE") == 0))
    return "SW";
else if ((strcmp (a, "SE") == 0) && (strcmp (b, "SW") == 0))
    return "NE";
else if ((strcmp (a, "SE") == 0) && (strcmp (b, "SE") == 0))
    return "NW";

```

```

        else return "NULL";
    }

char * sontype (node p)
{
    if (child (father(p), "NW") == p) {
        return "NW"; printf ("1\n"); }
    else if (child (father(p), "NE") == p) {
        return "NE"; printf ("2\n"); }
    else if (child (father(p), "SW") == p) {
        return "SW"; printf ("3\n"); }
    else if (child (father(p), "SE") == p) {
        return "SE"; printf ("4\n"); }
    else
        return "NULL";
}

node father (node child)
/* Returns a pointer to the parent of CHILD. */

{
    return (child->parent) ;
}

node child (node father, char whichchild[2])
/* Returns a pointer to one of the children of PARENT specified by WHICHCHILD */

{
    if (strcmp (whichchild, "NW") == 0)
        return (father->NW);
    else if (strcmp (whichchild, "NE") == 0)
        return (father->NE);
    else if (strcmp (whichchild, "SW") == 0)
        return (father->SW);
    else if (strcmp (whichchild, "SE") == 0)
        return (father->SE);
    else return NULL;
}

void quadtree_to_image (node root, int level)
/* Output image representation of 2^level x 2^level picture corresponding to the quadtree rooted at
node ROOT. For each row, the leftmost block is located by starting from ROOT and then visiting
in sequence the blocks comprising the row by ascending and descending the appropriate links in the
tree using neighbor finding. */

```



```

{
    node p = NULL;
    int diameter, distance, y;

    diameter = (int) pow(2, level);

    fprintf (fptr, "P3\n");
    fprintf (fptr, "%d %d\n", WIDTH, ROWS);
    fprintf (fptr, "%d\n", maxvalue);

    for (y = 0; y < diameter; y++) {
        /* Process the rows in sequence one row at a time */
        p = root;
        distance = diameter;

        /* Find the leftmost block containing row Y */
        find_2d_block (&p, 0, diameter, y, diameter, &distance);
        out_row (p, y, ceil((log10 (distance)) / 0.30103));
        printf("\n");
    }
}

```

*void out\_row (node p, int row, int l)*

```

{
    node q = NULL;
    int dist;
    int total = 0;
    dist = (int) pow(2, l);

    do {
        output_run (p->pixel_red, p->pixel_green, p->pixel_blue, dist);
        total += dist;

        /* Find the leftmost adjacent block containing row ROW */
        l = p->LEVEL;
        qt_gteq_edge_neighbor2 (p, "E", &q, &l);
        dist = (int) pow(2,l);

        if (q != NULL) {
            if (q->pixel_red == -1) /* a non-leaf node */
                find_2d_block (&q, 0, dist, row, row + dist - (row % dist), &dist);
            else dist = (int) pow(2,q->LEVEL);
        }

        p = q;
    }
    while (p != NULL);
}

```

```
}
```

```
void find_2d_block (node *p, int x, int xfar, int y, int yfar, int *w)
```

```
/* P points to a node corresponding to a block of width W having its lower right corner at (XFAR, YFAR). Find the smallest block in P containing the pixel whose upper left corner is at (X, Y). If P is non-gray, then return the values of P, W, and FAR; otherwise repeat the procedure for the son that contains (X,Y). */
```

```
{
```

```
    int q;
```

```
    while ((*p)->pixel_red == -1) {
```

```
        *w = (*w)/2;
```

```
        q = get_quadrant(x, xfar - (*w), y, yfar - (*w));
```

```
        xfar = xfar - xf[q] * (*w);
```

```
        yfar = yfar - yf[q] * (*w);
```

```
        if (q == 0)
```

```
            *p = child(*p, "NW");
```

```
        else if (q == 1)
```

```
            *p = child(*p, "NE");
```

```
        else if (q == 2)
```

```
            *p = child(*p, "SW");
```

```
        else if (q == 3)
```

```
            *p = child(*p, "SE");
```

```
    }
```

```
}
```

```
int get_quadrant (int x, int xcenter, int y, int ycenter)
```

```
/* Find the quadrant of the block rooted at (XCENTER, YCENTER) that contains (X,Y). The origin is assumed to be at the NW-most pixel of the image */
```

```
{
```

```
    if (x < xcenter) {
```

```
        if (y < ycenter)
```

```
            return 0;
```

```
        else return 2; }
```

```
    else if (y < ycenter)
```

```
        return 1;
```

```
    else return 3;
```

```
}
```

```
void output_run (int pixel_red, int pixel_green, int pixel_blue, int length)
```

```
/* Outputs the pixel value of PIXEL for a length of LENGTH */
```

```
{
    int i = 0;
    static int counter = 0;

    for (i = 0; i < length; i++) {
        counter = counter + 1;

        fprintf (fptr, "%d %d %d ", pixel_red, pixel_green, pixel_blue);
        if ((counter % 5) == 0)
            fprintf (fptr, "\n");
    }
}
```

```
void qt_gteq_edge_neighbor2 (node p, char *i, node *q, int *l)
```

```
/* Return in Q the edge-neighbor of node P, of size greater than or equal to P, in direction I. L denotes the level of the tree at which node P is initially found and the level of the tree at which node Q is ultimately found. If such a node does not exist, then return NIL. */
```

```
{
    *l = *l + 1;

    if (p->parent != NULL)
        if (adj(i, sotype(p))) {

            /* Find a common ancestor */
            qt_gteq_edge_neighbor2 (p->parent, i, q, l); }

    else
        *q = p->parent;
    else
        *q = p->parent;

    /* Follow the reflected path to locate the neighbor */
    if (*q != NULL)
        if ((*q)->pixel_red == -1)
            { *q = child((*q), reflect(i, sotype(p)));
              *l = *l - 1;
            }
}
```

```
void inorder (node root)
```

```
{
    if (root != NULL)
```

```

    {
        inorder (root->NW) ;
        if (root->pixel_red != -1)
            printf ("%d %d %d \n", root->pixel_red, root->pixel_green, root->pixel_blue) ;
        inorder (root->NE) ;
        inorder (root->SW) ;
        inorder (root->SE) ;
    }
}

```

```

int find_max(int *data_input)
/* FIND POSITION OF MAXIMUM DATA ITEM */
{
    int i, j;
    int max = -1;

    for (i = 0 ; i < ROWS ; i++)
        for (j = 0 ; j < WIDTH*3 ; j++) {
            if (*(data_input + j + WIDTH*i*3) > max)
                max = *(data_input + j + WIDTH*i*3);
        }
    return max;
}

```

# References

Atallah M., *Graph problems on a mesh-connected processor array*, Journal Association of Computing Machinery. 31, 1984.

Batcher V., *Design of massively parallel processors*, IEEE Transactions on Computers, C-28, 1980.

Bove M. and Watlington J., *Cheops: A reconfigurable dataflow system for video processing*, IEEE Transactions on Circuits and Systems for Video Technology, 1994.

Cowlishaw M., *Fundamental requirements for picture representation*, Proceedings of the SID, Vol. 2612, 1985.

Davis R. and Thomas D., *Systolic array chip matches the pace of high speed processing*, *Electronic Design*. Oct., 1984.

Dyer C. and Rosenfeld A., *Parallel image processing by memory-augmented cellular automata*, IEEE Transactions on Pattern Analysis Machine Intelligence PAMI-3, 1981.

Gersch R., *Benefits of an image parallel file system*, SPIE Vol. 1968, 1988.

Holroyd A., *Raster GIS*, Computers and Geosciences, Vol. 18 No. 4, 1992.

Horowitz S. and Pavlidis T., *Picture segmentation by a tree traversal algorithm*, ACM Vol. 23, April 1976.

Hung Y. and Rosenfeld A., *Parallel processing of linear quadtrees on a mesh connected computer*, Journal of Parallel and Distributed Computing 7, 1989.

Hung Y. and Rosenfeld A., *Processing border codes on a mesh-connected computer*, TR-227 Center for Automation Research University of Maryland College Park, 1986.

Ibarra O. and Kim M., *Quadtree building algorithms on a SIMD hypercube*, Proceedings of the 6th International Parallel Processing, 1992.

Klinger A. and Dyer C., *Experiments on picture representation using regular decomposition*, Computer Graphics and Image Processing, Vol 5., March 1976.

Klinger A. and Rhodes M., *Organization and access of image data by areas*, IEEE Transactions on Pattern Analysis and Machine Intelligence 1, 1979.

Leighton F., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes*, Morgan Kaufman Publishers, San Mateo CA, 1992.

Morton G., *A computer oriented geodetic database and a new technique in file sequencing*, IBM Ltd. Ottawa Canada, 1966.

Minsky M., *Scheduled routing for the NuMesh*, Masters Thesis, MIT, 1993.

Patterson D. et al, *A case for inexpensive disks (RAID)*, Proceedings of ACM SIGMOD Conference, 1988.

Rosenfeld A. et al, *Application of hierarchical data structures to geographical information systems*, TR-1197 University of Maryland College Park, 1982.

Samet H., *Region Representation: Quadrees for boundary codes*, Communications of the ACM 23, 3 1980.

Samet H., *Distance transform for images represented by quadrees*, IEEE Transactions on Pattern Analysis and Machine Intelligence 4, 3 1982.

Samet H., *Algorithms for the conversion of quadrees to rasters*, Computer Vision, Graphics and Image Processing 26, 1 1984.

Samet H., *Applications of spatial data structures*, Addison-Wesley Publishing Co., 1990.

Samet H. and Tamminen M., *Computing geometric properties of images represented by linear quadrees*, IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-7, 1985.

Scherson D., *Data structures and the time complexity of ray tracing*, Visual Computing 3, 4 1987.

Shankar R. and Ranka S., *Hypercube algorithms for operations on quadrees*, Pattern Recognition, Vol. 25 No. 7, 1992.

Snyder L., *Introduction to configurable highly parallel computer*, IEEE Computer Magazine, Jan 1982.

Stout Q., *Supporting divide-and-conquer algorithms for image processing*, Journal of Par-

allel and Distributed Computing 4, 1987.

Sureshandran S. and Warter P., *Algorithms and architectures for real-time image compression using a feature visibility criterion*, IEEE International Conference on Systems, Man and Cybernetics, Vol. 2, 1993.

Tanimoto S., *Pictorial feature distortion in a pyramid*, Computing Graphics and Image Processing, Vol. 5, 1976.

Tanimoto S., *Image transmission with gross information first*, Computer Graphics and Image Processing 9, 1 1979.

Ward S. et al, *The NuMesh: A modular, scalable communications substrate*, Proceedings of the International Conference on Supercomputing, 1993.

Yang S., *Efficient parallel neighbor finding*, Journal of Information Science, Vol. 9 No. 1, 1993.

Zhang X. at al, *Adaptive quadtree coding of motion-compensated image sequences for use on the broadband ISDN*, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 3, No. 3, June 1993.