NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

THE PAIR TREE:
A PARALLEL ARCHITECTURE FOR IMAGE REPRESENTATION BASED ON SYMMETRIC RECURSIVE INDEXING

by

William Paul Kastelic
B. Sc., Simon Fraser University, 1982
Extended Studies Diploma, Simon Fraser University, 1984

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
January, 1989

ISBN 0-315-59351-2

APPROVAL

Name: William Paul Kastelic

Degree: Master of Science

Title of Thesis: The PAIR Tree — A Parallel Architecture for Image Representation based on Symmetric Recursive Indexing

Examining Committee:

Chairman: Dr. S. Pilarski

Dr. T. W. G. Calvert
Senior Supervisor
Professor, School of Computing Science

Dr. R. Hobson
Director and Associate Professor, School of Computing Science

Dr. F. W. Burton
External Examiner
Professor, School of Computing Science

Date Approved: January 17, 1989

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

The PAIR Tree - A Parallel Architecture for Image Representation Based on Symmetric Recursive Indexing

Author:

(signature)

WILLIAM PAUL KASTELIC

(name)

Feb. 27/89

(date)

Abstract

Significant technological advances in such fields as computer architecture and very large scale integration have made it feasible to consider firmware-based alternatives to some systems traditionally implemented in software. Some form of representation can usually be found for the underlying basis of such systems. If these representations can be implemented using the new designs and techniques, increased system performance can be achieved.

We investigate one such representation, based on the 2^N-ary tree, which has been successfully implemented at the software level. Its broad scope of applicability motivates the investigation of possible firmware alternatives. Since another feature of this representation is the inherent parallelism of its operations, and extensive work is also being performed in the area of concurrent architectures, a primary objective was to design three such architectures for this representation. As secondary objectives, a set-theoretic representation from which our scheme can be transformed was described and its parallel characteristics were analysed.

Dedication

To Mom, Dad and Tania.

For their love, encouragement and support.

V zahvalo in poklon Mom, Dad ter Tania.

Za njihovo ljubezensko podporo in potrpežljivost, pri uspehu tega dela.

"Research is what I'm doing when I don't know what I'm doing"

—Wernher von Braun

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Tom Calvert, for the numerous suggestions and comments that he has made during the course of this research and in the proofreading of the thesis. The patience that he has exhibited throughout this experience is appreciated. The constructive criticisms made by the other members of the Examining Committee, Drs. Warren Burton, Slawomir Pilarski, and Rick Hobson, provided further insight into the research problem and solution considerations.

The assistance provided by the School of Computing Science's system administrators Ed Bryant and Keith Vincent simplified my programming task. The School's administrative assistant, Mrs. Elma Krbavac, with her sense of humour, personality, and optimism, made the daily jaunts to the General Office interesting. Those frequent trips to the Pub with other graduate students, where the intent was to discuss the relevant issues of computing science, were always enjoyable.

Finally, acknowledgement must be made to the Department of Computer Science at the University of Victoria for allowing me to use their facilities in continuing with this research.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

Two issues which must be considered by the systems[1] designer in planning computer-based applications are the data representation to be used (particularly from the points of view of storage and operations) and its implementation. A representation must be chosen which fulfills the requirements of the application. This representation may take the form of some previously defined scheme, a modification of another representation, or the task may be such that an entirely new representation is needed.

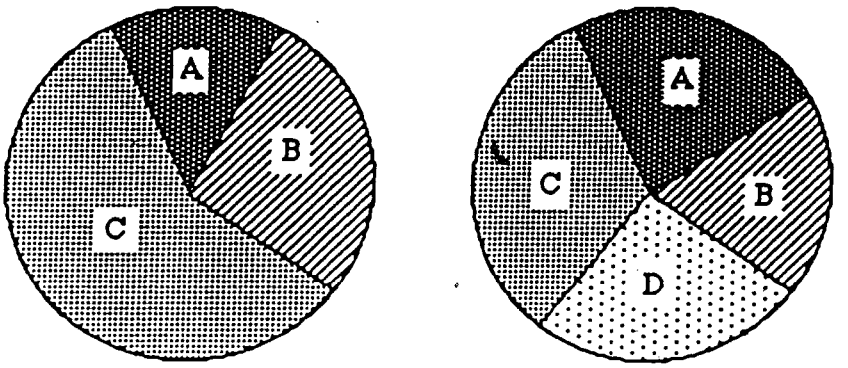
Traditionally, it was only necessary to consider software implementation since hardware implementation was not feasible. In recent years, the focus of study between software versus hardware techniques has shifted in the direction of the latter. Processors have evolved from the ungainly units of the 1950's and 1960's, to the single 32-bit microchip of the 1980's. There has been a progression from ferrite core memory to CD ROM[2]. VLSI[3] has provided a means of placing more of the computer's hardware components onto the microchip, while at the same time improving upon component performance. Developments such as these have made it necessary for the systems designer to become aware of the hardware architectures which are available.

It has been suggested that prior to these advances, it was possible to divide system functionality into three distinct classes: application specific software;

[1] Let a *system* refer to the data and set of tools applicable to the task at hand.

[2] CD ROM - Compact Disc Read Only Memory.

[3] VLSI - Very Large Scale Integration.



Traditional Trends Anticipated Future Trends

A - General purpose hardware
B - General purpose software
C - Application specific software
D - Application specific hardware

Table 1.1: Relative Division of System Functionality

general purpose software, and general purpose hardware [17]. The anticipated future breakdown would involve an additional category, that of application specific hardware. Table 1.1 presents an estimation of the relative contribution of each class to this functionality considering present and anticipated trends.

System performance analysis is another issue that the designer is required to address. Candidate solutions must be evaluated on various criteria, and a decision made based on these results. The task facing the designer is compounded by the fact that the two extremes of the solution, that is, an entirely software or VLSI-based approach, may in fact only provide a partial solution to the overall problem. Another point to consider is that neither of these extremes may provide the most efficient solution. A compromise may be necessary to provide the the most attainable and efficient overall solution.

This dissertation addresses two specific tasks. The first involves the development of a set-theoretic representation scheme. The second considers three system architectures which are designed for one form of this scheme and its associated properties. As will become apparent in the following section and chapters, a representation defined using a set-theoretic approach is very powerful in terms of its scope, and in the number of operations possible. It will be shown that the 2^N-ary tree scheme is a transformation of this set representation.

A further implication of this set-based representation concerns the order in which operations are executed on the entities of a particular application. Tasks can be executed in a sequential or parallel manner. The operations on the 2^N-ary tree are inherently parallel, and it is this fact which is exploited in developing viable architecture alternatives to the representation's software implementation.

The remainder of this chapter discusses the criteria necessary for designing and evaluating a representation scheme, followed by an expansion of some of the reasons presented earlier for the importance of hardware and VLSI technologies in systems design. A particular focus is the motivation behind developing an alternative to the software approach to 2^N-ary trees. The chapter concludes with an informal definition of the 2^N-ary tree, and a presentation of typical algorithms for these trees within the scope of computer graphics.

The dissertation continues with Chapter 2 which presents a formal definition of the set-theoretic representation scheme. The 2^N-ary tree representation is then defined as a transformation of this scheme. Given this transformation, a more concrete basis to the earlier quad- and oct-tree definitions can be extrapolated. Chapter 3 deals with a time complexity analysis of binary operations on quadtrees. This type of analysis allows generalizations to be made for the 2^N-ary tree. The first of three architectures is given in Chapter 4. This architecture attempts to map the logical 2^N-ary scheme on an array of interchanging rows of processing elements and interconnection networks. In Chapter 5, the mapping of a binary tree onto a VLSI array is given, in addition to a justification for using such

a mechanism as an alternative to the more general 2^N-ary tree. The third architecture can be considered a linear arrangement of processor elements which access a bank of shared memory modules in which instances of trees are stored and this description is given in Chapter 6. A simulation was implemented for this last architecture and this is described in Chapter 7. The dissertation concludes with a discussion of additional issues which must be considered in all three architectures.

1.1 Properties Associated With Any Representation Scheme

A primary consideration in developing a solution to a problem is the choice of a foundation or representation which will facilitate generation of a solution. A number of factors exist which must be considered when defining or evaluating a representation scheme [31]. Some formal properties inherent in any scheme are:

Domain

The *domain* is an indication of the *descriptive power* of a representation. In particular, the domain is the set of objects or entities that can be represented by some scheme S;

Validity

The range of a representation scheme S is the set of syntactically *valid* or *correct* representations that are images of elements of the domain of S. The application of certain algorithms on invalid representations may lead to inconsistent results which may or may not be detected by the user;

Completeness

A representation R in the range of some scheme S is *complete* if it corresponds to a single object in the domain. The scheme S is itself complete if all of its valid representations are themselves complete. Each R contains sufficient information to distinguish one object from all other

entities of the same domain. This is a critical property if S is to be used over a wide range of applications;

Uniqueness

A representation R in the range of some scheme S is *unique* if the objects that it defines cannot be represented by any other representation in S. Any S is unique if each of its valid representations are unique. Uniqueness is important when considering whether two or more representations are responsible for the same object.

In addition to these formal considerations, there also exist some properties which are informal in nature:

Conciseness

This property concerns itself with the storage requirements of a representation in a scheme. The ease with which a representation is stored and manipulated is dependent upon the concise nature of the structure itself. Redundant information about a representation is also minimized if its definition is concise.

Ease of creation

A valid representation should be reasonably easy to create. In general, the more concise a representation is, the easier it is to create, since there is less data redundancy.

Effectiveness

This point considers effectiveness in terms of the context of applications. Algorithms must be developed in such a way that the representations themselves can be considered as data for these algorithms. A representation scheme is effective if the algorithms applicable to the scheme are correct, efficient (both in terms of storage and computational complexity), and reliable when subtle errors in the representation are encountered.

These factors provide some guidelines for representation development. It is understandable that certain schemes may be relatively deficient in some of these points when compared against other representations. However, the opposite may also occur — these same schemes may be more favourable in terms of some other factors. In many cases, the representation chosen will provide a compromise between all of these points. The goal of the designer should be to strive for the most complete scheme which provides the best performance.

1.2 Consideration of Hardware Techniques in Applications/System Design

As indicated earlier, systems development has typically followed a software approach. However, in recent years there has been a shift from these traditional methods to solutions incorporating hardware/VLSI-based techniques. Some reasons for this progression include the following:

- advances in VLSI technology have resulted in cost reductions, while increasing performance and chip density characteristics;
- the use of ever-improving computer-aided design (CAD) techniques in VLSI development has simplified the task of designing chips;
- an increased research effort in the area of developing VLSI-based alternatives to existing software functions/structures has produced interesting results;
- certain applications, such as real-time sensing and control systems, require response times which current software methods cannot provide.

The claim here is that software solutions can frequently be supplemented with application specific, as opposed to general purpose, hardware architecture.

1.3 An Informal Definition of the 2^N-ary Tree

The following definition of the 2^N-ary tree representation involves references to entities such as *N-dimensional objects*, *unit cells*, and *volume*. This reflects the influence that the scheme has had in the area of computer graphics. It was this computer graphics theme which was the motivation for proceeding with this dissertation. The general nature of the representation provides for a very broad domain to which this scheme can be applied.

Objects exist within some *universe* U, which is of order N, and is a finite section within some N-dimensional space. This space is defined by N orthogonal axes. An object that exists in U is of the same order as the universe. A 2–D object cannot exist within a 3–D universe. The smallest object in such a universe would be the smallest resolvable unit of space.

In a typical situation involving the representation of some N-dimensional object, the universe is a 2^{MN} array of unit cubes or cells. Associated with each of these cells is a value from the domain of some distinguishing property (for example, colour, radiation intensities, material type, density, land-use characteristics). The *diameter*, D, of such an *object array* is 2^M, where $M > 0$, and may be divided into 2^N non-overlapping cube-shaped arrays of diameter 2^{M-1}. When $N = 2$, the universe is a square, a cube when $N = 3$, and a hypercube for $N > 3$.

The symmetric recursive indexing process subdivides an object array into 2^N subarrays, each of equal volume.[4] Every subarray is classified as being either homogeneous or heterogeneous, based on the predefined distinguishing property. Each heterogeneous subarray is further divided into 2^N additional subarrays. This continues until all subarrays are homogeneous. The subdivision technique may eventually have some subarrays being composed of a single element.

[4]Here, *volume* is a general term used to refer to the form that a subarray, or hypercube, will take when $N > 2$.

For any object array, the entire procedure generates a 2^N-ary tree representation. A node in the tree is either a leaf (representing a homogeneous subarray) of indegree 1, or an internal node (representing a heterogeneous portion of the object) with an indegree of 1 and an outdegree of 2^N (each child corresponds to one of the 2^N subarrays). Every level in the tree is identified by some integer i, $0 \leq i < M$. The root node of such a tree represents the entire object array, while each of its children represents one of the subarrays of the object. The root node is at level 0. The nodes at level i completely describe the object to the resolution of that level. M is commonly referred to as the *resolution* of the universe (*maximal resolution* of the object). Depending upon the amount of information needed, it is only necessary to display part of the tree. Thus, the root node presents the coarsest display of the object, while the display of levels near the bottom of the tree lead to finer representations. The information in this tree is implicit, and requires the application of the representation's operations for its retrieval.

Figure 1.1 (modified from Dyer *et al.* [11]) presents a 2-D object ($N = 2$) that is encoded into a 2^2-ary or quadtree.[5] The region is shown in Figure 1.1 (a). The object is considered to be part of some universe. This encapsulation is evident in Figure 1.1 (b). The figure also shows the divisions that have been made in the universe to decompose each quadrant into its homogeneous state. Finally, the quadtree for Figure 1.1 (b) is shown in Figure 1.1 (c). It is important to note that the quadtree represents the decomposition of the universe, and that object representation is due to the object being a subset of the universe. In this way, the region is represented as a union of maximal units, where each unit is of a standard size and position (powers of 2). The simplest property that can be used on an object is based on whether or not a subunit is associated with the object. **VOID** indicates that the subunit does not contain any part of the object. **FULL** is used if the subunit is part of the object. If the subunit is a composite of the two types, then **PARTIAL** is used. The tree's interior nodes are circular (representing

[5] An extensive survey on the quadtree can be found in [35].

PARTIAL subunits) while leaves are square (**FULL** or **VOID**).

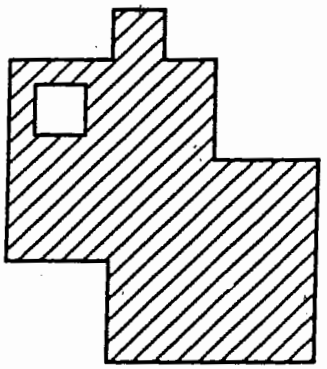
When dealing with a 3–D object, $N = 3$, and the tree (or *octree* [25]) generated is of degree 8 (2^3). Likewise, each of the eight subarrays is called an *octant.* To access some point (X,Y,Z) in an octree, it is necessary to compute the binary representations of X, Y, and Z as $x_0x_1x_2 \ldots x_i$, $y_0y_1y_2 \ldots y_i$, and $z_0z_1z_2 \ldots z_i$, respectively. The object in Figure 1.2 (a) (modified from Srihari [42]) spans the width of the limiting universe ($D = 8$). Therefore, the maximum number of levels in the corresponding octree will be three ($M = 3$). Figure 1.2 (b) shows one of the many ways that the octants within a block may be traversed. Figure 1.2 (c) presents the octree generated from the successive division of the universe.

1.4 Some graphics-based 2^N-ary tree algorithms

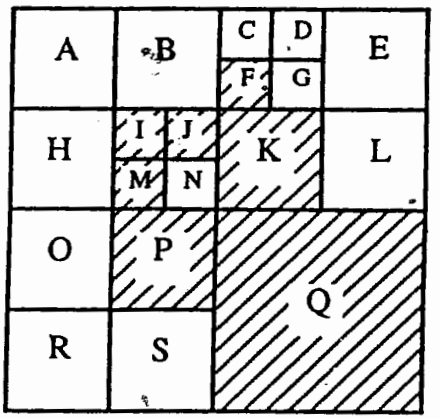
In computer graphics, images can be stored as 2^N-ary trees, where quad- and oct-trees are the most common forms of the representation. Algorithms have been developed for operations such as:

- area, volume, complement, intersection and union [39]
- transformations such as rotation, translation, and scaling [20,25]
- outline determination, and nearest neighbour identification [19,33]
- stereographic projections onto multiple planes [15]
- display of images [10]

Schneier's parallel algorithms for the INTERSECTION and AREA operations on quadtrees are presented here to provide an indication of the representation's simplicity. The algorithms for greater values of N are straightforward extensions of the quadtree routines. Doctor's algorithm for the display of octrees is also given.



(a) Region



(b) Subdivision of (a)

(c) Quadtree representation of (b)

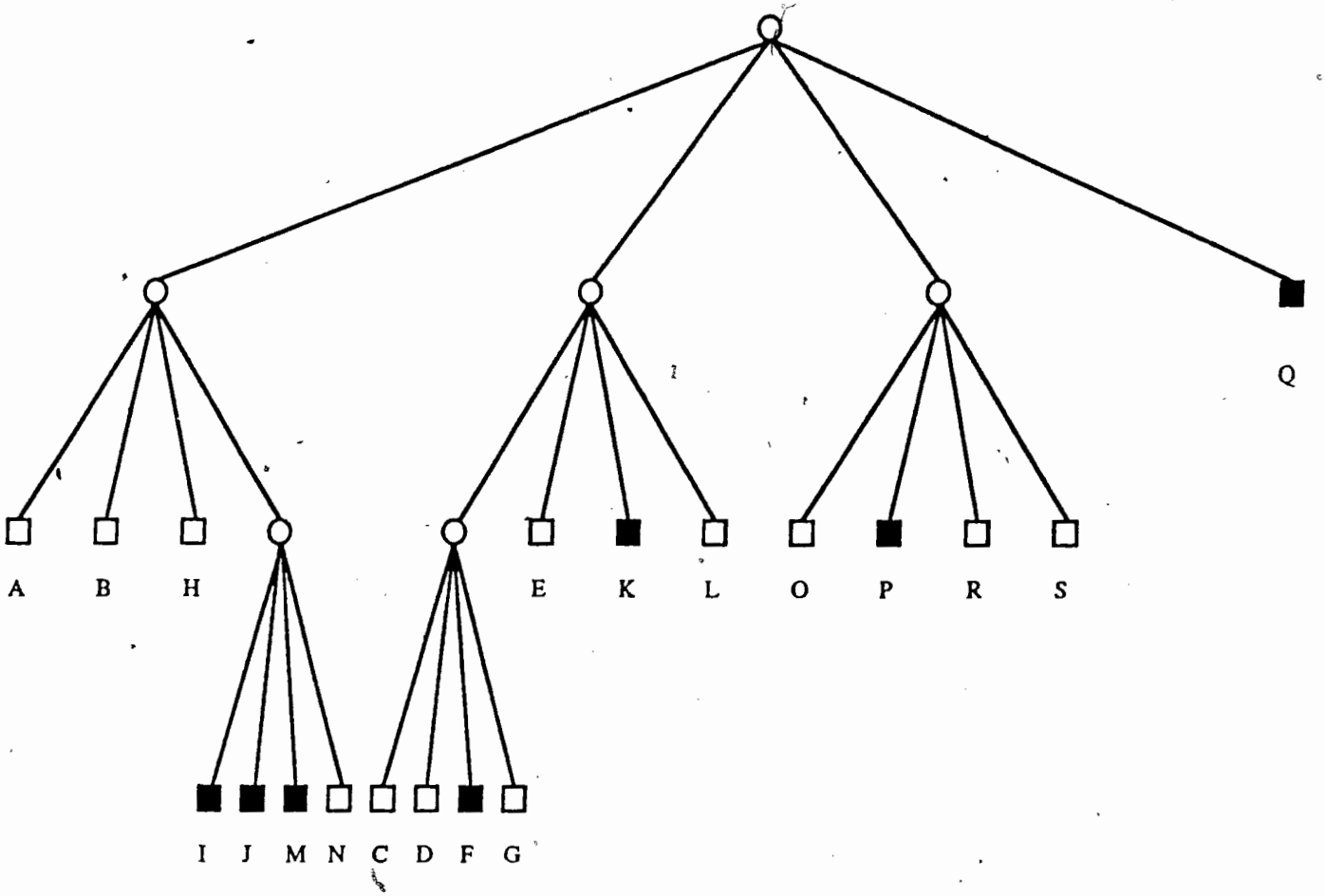
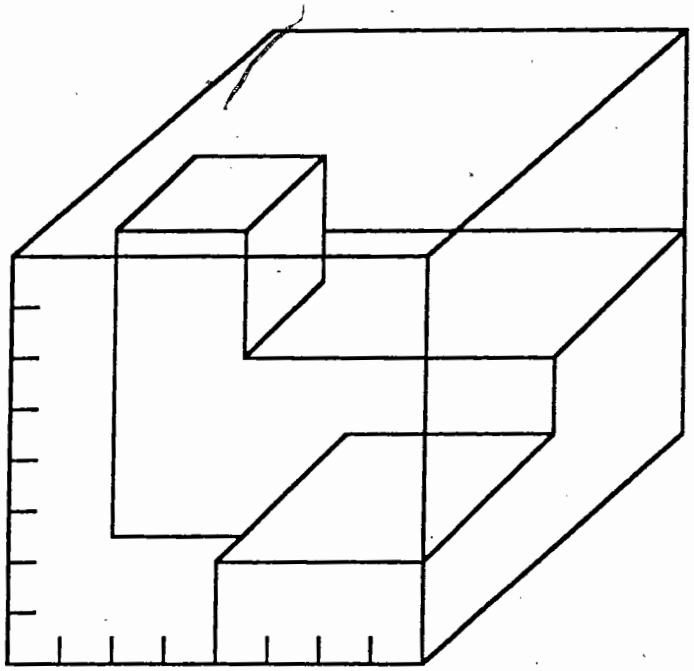
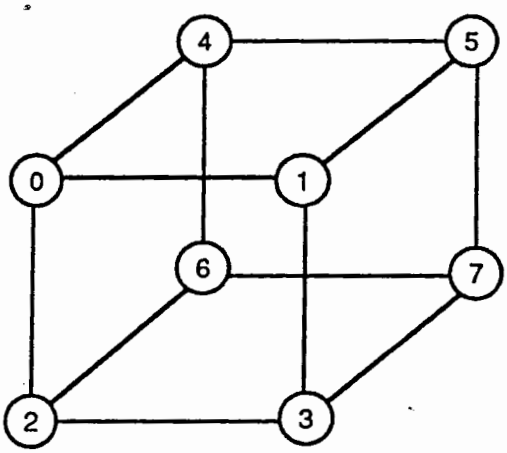


Figure 1.1: Quadtree encoding



(a) Object in 3-Space



(b) Numbering convention

(c) Octree representation of (a)

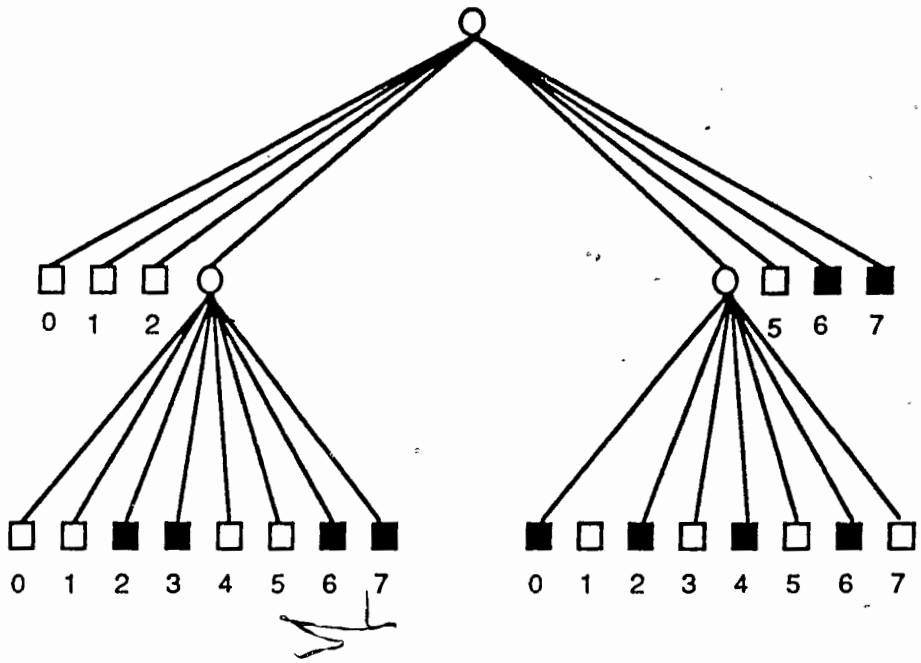


Figure 1.2: Octree encoding

The following conditions exist in performing these operations. It is assumed that the leaves represent **BLACK** and **WHITE** areas within the images, where **WHITE** is the background value. Interior tree nodes are given the value **GREY**. **BLACK**, **WHITE**, and **GREY** are equivalent to the **FULL**, **VOID**, and **PARTIAL** values of our informal definition. It is assumed that the quadrants of a region can be addressed as **NW**, **NE**, **SW**, and **SE**. The auxiliary routines **GREY**, **BLACK**, and **WHITE** return the value of the parameter node. The **SET_QCHILD** routine adds a child node at some given quadrant to its parent. The **PARENT** function creates a node which represents the parent of a given child node. **COPY** generates a structure which is identical to the given tree argument. The procedure **SET_AND** stores the result of performing an **AND** operation on the children of a quadtree in said tree.

1.4.1 The INTERSECTION operation

This algorithm returns a quadtree result which represents the common regions of two quadtrees. If a leaf is **BLACK** and its equivalent node in the other tree is non-**BLACK**, the result node will be a copy of the non-**BLACK** node's subtree. If a leaf is **WHITE**, then its corresponding result leaf will also be **WHITE**.

```
function INTERSECTION(TreeA, TreeB : quadtree) : quadtree;
  begin
    TreeAND : quadtree;
    I       : quadrant;

    if BLACK(TreeA) or WHITE(TreeB) then
       INTERSECTION := COPY(TreeB);
    else
       if BLACK(TreeB) or WHITE(TreeA) then
          INTERSECTION := COPY(TreeA);
    TreeAND := CREATENODE()          /* create root node */
    for I in {NW, NE, SW, SE} do     /* in parallel */
        begin
          SET_QCHILD(TreeAND, I, INTERSECTION(QCHILD(TreeA, I),
                                              QCHILD(TreeB, I)));
          end;
```

```
SET_AND(TreeAND);
INTERSECTION := TreeAND;
end;
```

1.4.2 Find the AREA of a quadtree

The following algorithm determines the area of an image, where the number of black nodes in a quadtree represents this area. The parameters to the function **AREA** include the quadtree and diameter of the image.

```
function AREA(Tree : quadtree; D : integer) : integer;
  begin
    TempAREA : integer;
    I : quadrant;

    TempAREA := 0;
    if GREY(Tree) then
       for I in {NW, NE, SW, SE} do
           TempAREA := TempAREA + AREA(QCHILD(Tree, I), D-1);
    else
       if BLACK(Tree) then
          TempAREA := TempAREA + 2**(2*D);
    AREA := TempAREA;
    end;
```

1.4.3 Display algorithm for an octree

An interesting consideration in the presentation of an object represented as a three-dimensional octree is that the display device is usually two-dimensional. Quadtrees provide a very efficient means of representing two-dimensional objects. By using the transformation of an octree to a quadtree, very effective display algorithms for problems such as hidden-surface removal may be developed. Additional algorithms using different shading techniques, illumination, and semitransparent objects have also been developed using the octree-to-quadtree transformation [10].

By traversing the nodes in a specific front-to-back order, the hidden surface removal procedure is very straightforward. The quadtree generated can then be sent to the display device. In the following algorithm, the front four octants are numbered 0 through 3, while the back four are 4 to 7. The two parameters are the octree to be displayed, and the intermediate quadtree, which is initially a NULL tree. The function **OCHILD** is analagous to the **QCHILD** routine for quadtrees. **HOMOGENEOUS** is a function which determines whether a node is a leaf or interior node. **COLOUR** returns a quadtree of one node whose value field represents the display colour. The **MAKETREE** function creates a quadtree with a new root node that has as its children the four quadtree parameters.

```
function SHOW(Octree : octree; Quadtree : quadtree) : quadtree;
  begin

    if HOMOGENEOUS(Octree) then
       if not WHITE(Octree) then
          SHOW := COLOUR(VALUE(Octree));
       else
          SHOW := Quadtree
    else
       SHOW := MAKETREE((OCHILD(Octree, 0),
                    SHOW(OCHILD(Octree, 4), QCHILD(Quadtree, 0)),
                       SHOW(OCHILD(Octree, 1),
                    SHOW(OCHILD(Octree, 5), QCHILD(Quadtree, 1)),
                       SHOW(OCHILD(Octree, 2),
                    SHOW(OCHILD(Octree, 6), QCHILD(Quadtree, 2)),
                       SHOW(OCHILD(Octree, 3),
                    SHOW(OCHILD(Octree, 7), QCHILD(Quadtree, 3))));
    end;

function COLOUR(Value: colour_values) : quadtree;
  begin
    COLOUR := CREATENODE(Value);
    end;
```

1.5 General observations of the quad- and oct-tree representations

Extensive effort has gone into investigating the properties of the quad- and oct-tree representation schemes. Computer graphics, pattern recognition, and image processing have been the primary areas of application. The use of octrees in solid object modelling has been shown to be very beneficial [31]. Considerable success has been achieved in applying this representation to geographic information systems [1,27]. This research has concluded that there are a number of advantages to using the representation. As the three algorithms of the previous section show, the operations are reasonably straightforward to develop. Numerous algorithms have been generated to convert between quadtrees and other representations[33,36,37,34]. One feature which makes this representation useful is that there is only one primitive object, such as the square or cube, to contend with. The size of this primitive determines the level of representation for any given object. The operations defined for this representation are only required to deal with this primitive. Ordering is implied in this representation, which further simplifies algorithm development.

Situations do exist where the more common 2- and 3-D subdivision methods are not sufficient to fulfill certain requirements. Application of this representation to 4-D leads to a hextree, a tree with 16 subarrays. It is then possible to represent time-varying 3-D images. Two examples of a 4-D image include the dynamic, spatial reconstruction of a beating heart and the breathing lung [44]. The generalization of this method to dimensions greater than four is also useful. One such application involves robot motion planning and multidimensional configuration spaces [49].

Chapter 2 Set Theoretic Basis of the 2^N-ary Tree

Set theory can provide the mathematical justification for the 2^N-ary tree's extensive domain of applicability. It can be safely stated that most computer-based applications deal with data which are naturally related. However, there are situations where the relationships are not obvious, yet require that this data be expressed in such a way as to maintain these relationships. A method must be developed which can accommodate both cases. Set theory fulfills this requirement in that these relations can be expressed as a set of ordered pairs, while at the same time providing a significant collection of operations. An added feature of following this approach is that these operations can be executed rapidly.

2.1 Populations and Attributes

The following definitions provide the set theoretic foundation upon which the 2^N-ary tree is developed.

Definition 2.1 *Let the* population Φ *be a finite collection of items.*

Definition 2.2 *Each* Φ *has associated with it a set of attributes* A_Φ.

Definition 2.3 *Each* A_Φ *has associated with it a set of values* V_{Φ_α}. *In addition, there is a special value* VOID (I-DON'T-CARE) *which is part of each* V_{Φ_α}. *As such, each* γ *of* Φ *can be assigned a value from* V_{Φ_α}.

Φ_0 = collection of pixels making up an image

Φ_1 = a map of British Columbia

Φ_2 = a population of people

$A_{\Phi 0}$ = {colour}

$A_{\Phi 1}$ = {landuse, population, economic activity}

$A_{\Phi 2}$ = {education, marital status, salary, sex}

$V_{\Phi 0 \text{colour}}$ = {RED, WHITE, BLACK, VOID}

$V_{\Phi 1 \text{landuse}}$ = {FARMING, LOGGING, MINING, VOID}

Table 2.1: Examples of populations, attributes, and values

The definitions of Φ, A and V permit the accommodation of additional items, attributes and values, respectively, as the need arises.

Table 2.1 presents a number of examples involving populations, attributes, and associated values.

Definition 2.4 *The* overall picture *for some population Φ consists of the set $\overline{\mathbf{P}}_{\mathbf{\Phi}} = \{\mathbf{P}_{\mathbf{\Phi}_\alpha} : \alpha \in A_\Phi\}$. $\overline{\mathbf{P}}_{\mathbf{\Phi}}$ can be considered as a family of sets, and $\mathbf{P}_{\mathbf{\Phi}_\alpha}$ is a* subpicture *of Φ for $\alpha \in A_\Phi$. A more precise definition of $\mathbf{P}_{\mathbf{\Phi}_\alpha}$ is to follow.*

To this point, sets have been exclusively used in the definition of Φ, A_Φ, V_{Φ_α}, and $\overline{\mathbf{P}}_{\mathbf{\Phi}}$. Given this fact, any valid set operation can be applied to these various sets. These operations can be applied to all of $\overline{\mathbf{P}}_{\mathbf{\Phi}}$. Two example definitions involving set operations and $\overline{\mathbf{P}}_{\mathbf{\Phi}}$ are shown in Table 2.2.

An interesting fact here is that any result generated from a set operation involving $\overline{\mathbf{P}}_{\mathbf{\Phi}}$ or any of its elements, is a new subpicture $\mathbf{P}_{\mathbf{\Phi}_\nu}$. The examples of $\bigcup \overline{\mathbf{P}}_{\mathbf{\Phi}}$ and $\bigcap \overline{\mathbf{P}}_{\mathbf{\Phi}}$ are two such cases. The new attribute ν is defined as a function of the operations and attributes used to generate this new subpicture/result.

The unary union of $\overline{\mathbf{P}}_{\Phi}$:

$$\cup\overline{\mathbf{P}}_{\Phi} = \{e : (\exists \mathbf{P}_{\Phi_\alpha} \in \overline{\mathbf{P}}_{\Phi})(e \in \mathbf{P}_{\Phi_\alpha})\}$$

The unary intersection of $\overline{\mathbf{P}}_{\Phi}$:

$$\cap\overline{\mathbf{P}}_{\Phi} = \{e : (\forall \mathbf{P}_{\Phi_\alpha} \in \overline{\mathbf{P}}_{\Phi})(e \in \mathbf{P}_{\Phi_\alpha})\}$$

Table 2.2: Two set operation definitions given with respect to pictures

Definition 2.5 *Let $\overline{\alpha}$ be the set of attributes from A_Φ involved in a query, and the cardinality of α is greater than one ($\#\overline{\alpha} > 1$). Given any query involving two or more attributes from A_Φ, a new attribute ν can be defined based on the consolidation of these attributes. The values of this ν belong to the set $V_{\Phi_\nu} = \{\text{TRUE}, \text{FALSE}, \text{VOID}\}$. A_Φ now becomes $A_\Phi \cup \{\rho(\overline{\alpha})\}$ where $\rho(\overline{\alpha})$ returns a singleton set containing a unique label/attribute based on the elements of $\overline{\alpha}$.*

If $\#\overline{\alpha} = 0$, then no attributes are involved in the query. In the case of $\#\overline{\alpha} = 1$, the result is the set of elements from $\mathbf{P}_{\Phi_\alpha}$whose attribute values match those of the queried attribute value. When $\#\overline{\alpha} > 1$, the query will access those pictures for which the attributes are being considered. The result will the be taken from these pictures. The elements of this result will have values taken from the set V_{Φ_ν}.

2.1.1 An Example

Table 2.1 presented a number of different populations, together with each collection's set of attributes and associated values. The following example considers $\Phi = \Phi_2 = \{people\}$, where the specific query asks:

Find all females with post-secondary education

The newly-generated attribute will involve "sex and education". The next step is to categorize each element of Φ as either having or not having this property. A

subpopulation Φ_ν, where $\Phi_\nu \subseteq \Phi$, exists such that each element of Φ_ν either fulfills this query or it does not.

V_{Φ_ν} = {TRUE, FALSE, VOID}

$\Phi_\nu = \{\gamma : \gamma \in \Phi, \text{setval}_\nu(\gamma, \Upsilon(V_{\Phi_\nu}, \text{val}_{sex}(\gamma) = \text{FEM}, \text{val}_{educ}(\gamma) = \text{PSEC}))\}$

The attribute ν is equivalent to "female with a post-secondary education". This may seem unnatural since most attributes deal with single categories rather than in more complex relationships. The function Υ has $n + 1$ parameters, where n is the number of attributes under consideration in the current query. The purpose of this function is to return one element from the set V_{Φ_ν} for each γ in Φ.

2.2 n-tuples and E_{Φ_α}

Definition 2.4 refers to the subpicture $\mathbf{P}_{\Phi_\alpha}$ as a component of $\overline{\mathbf{P}}_{\Phi}$. The majority of abstractions considered to this point (for example, Φ, $\overline{\mathbf{P}}_{\Phi}$, A_Φ, and V_{Φ_α}) have been based on set constructions. As such, set operations on these abstractions are perfectly valid. By defining it at a gross level, $\mathbf{P}_{\Phi_\alpha}$ can also be considered as a set-theoretic abstraction.

Definition 2.6 *A* subpicture $\mathbf{P}_{\Phi_\alpha}$ *is an ordered pair of the form* $\mathbf{P}_{\Phi_\alpha} = < V_{\Phi_\alpha}, E_{\Phi_\alpha} >$, *where* $\alpha \in A_\Phi$, $E_{\Phi_\alpha} = \{\gamma : \gamma \in \Phi, \text{val}_\alpha(\gamma) \in V_{\Phi_\alpha}\}$

The function *val* maps an element γ of Φ_α to a single value from the range V_{Φ_α}. *Val* is *not* a one-to-one function as it is highly probable that different elements from the domain may have the same image via this mapping. At this level of abstraction, the ordered pair $\mathbf{P}_{\Phi_\alpha}$ can still be represented on a set-theoretic basis[5].

Each γ is also an ordered pair, where the first component is a unique identifier within Φ, and the second is an n-tuple, $val_\alpha(\gamma)$ ($\#\alpha = n$). This second term can

[5]Here, an ordered pair is defined as $< a, b >= \{\{a\}, \{a, b\}\}$[23].

be considered as a vector of values that a γ possesses for each attribute involved in the picture. The appropriate selector, recognizer, and constructor functions are defined to process the first or second components of γ. The same types of functions are available to access the n-tuple.

2.3 Partitioning of Φ

For any $\mathbf{P}_{\Phi_\alpha}$, its corresponding E_{Φ_α} may prove to be very inefficient in terms of storage considerations. It is necessary to store each component of E_{Φ_α} in memory. Therefore, if the cardinality of Φ is great, a significant cost in terms of this storage will be accrued. There will be an even greater storage requirement if the cardinality of A_Φ is also large.

Of the two components A_Φ and E_{Φ_α}, it is most likely the latter which contributes most unfavourably to the storage costs. The primary reason is that each object in Φ is an identifier for an element in E_{Φ_α} ($\#\Phi = \#E_{\Phi_\alpha}$). A significant reduction in $\#E_{\Phi_\alpha}$ can be made if each element of Φ is placed into a subset with other elements of Φ that have the same value for the attribute being considered.

For any $\mathbf{P}_{\Phi_\alpha}$, it is possible to reduce the cardinality of E_{Φ_α} to a maximum of $\#V_{\Phi_\alpha}$, or to a minimum of 1.

Definition 2.7 *Let E_{Φ_α} be represented as:*
$E_{\Phi_\alpha} = \{e : (\exists\gamma)(v \in V_{\Phi_\alpha})(\gamma \in \Phi)(\mathrm{val}_\alpha(\gamma) = v)(\gamma \in e)\}$

A partition on Φ is formed where each element of E_{Φ_α} will be a non-empty set and represents a $v \in V_{\Phi_\alpha}$. An item $\gamma \in \Phi$ will belong to that set for which $\mathrm{val}_\alpha(\gamma) = v$. In the worst case, E_{Φ_α} will consist of $\#V_{\Phi_\alpha}$ non-empty sets. If for all $\gamma \in \Phi$, $\mathrm{val}_\alpha(\gamma)$ is the same, then $\#E_{\Phi_\alpha} = 1$.

In the above redefinition of E_{Φ_α}, it is evident that each $\gamma \in \Phi$ will have to be tested to place γ in the appropriate element of E_{Φ_α}. Any decrease in storage

requirements will be offset by the considerable amount of time needed to complete the partitioning of Φ in a sequential manner.

An interesting property of any partition is that any two sets from this family of sets are disjoint. This implies an inherent parallelism in the processing that may be performed to generate the sets that make up E_{Φ_α}. As every $\gamma \in \Phi$ is independent of other elements of Φ, each γ can be processed independently. One extreme situation would involve processing each such γ in parallel with the other elements of Φ. Obviously the other case would require a strictly sequential approach in generating the elements of E_{Φ_α}. A compromise between these two limits must be made.

The compromise also involves deciding upon a partitioning that makes use of a granularity which is acceptable in terms of the processing power available. Another point which is considered in attaining this compromise is that data tends to be grouped together. For example, if our Φ is a database of vegetation for a specific province, there will be regions represented which will consist predominantly of one form of vegetation. Of course, there will be cases where this similar neighbour effect will not apply, and it will be necessary to consider a finer partitioning.

Rather than attempting to consolidate similar elements of Φ into distinct subsets, an arbitrary partitioning approach can be taken. Some function M maps the elements of a set Φ' to the subsets comprising a partition on Φ, without regard for attribute value homogeneity within a subset. The values that these elements $\gamma' \in \Phi'$ assume are determined by the values of the subsets that they map onto.

To simplify the notation, the following two definitions assume that the operands and functions refer to specific attributes $\alpha \in A_\Phi$. For example, γ'_α is equivalent to γ'.

Definition 2.8 M *is a one-to-one function which maps* $\gamma' \in \Phi'$ *to a subset* $S'_\gamma \subseteq \Phi$ *(the partitioning of* Φ *is implied by* M*'s functionality).*

Definition 2.9 *For every* $\gamma' \in \Phi'$,

if $\#\{\text{val}(\gamma) : \gamma \in S'_\gamma, M(\gamma') = S'_\gamma\} = 1$, $\text{setval}(\gamma', \text{val}(\gamma \in S'_\gamma))$

else $\text{setval}(\gamma', \text{PARTIAL})$.

This new population Φ' can have a similar mapping scheme M' applied to generate Φ''. This mapping implies a series of levels where each level is composed of a *finer* partition than its predecessor level. A partition A is *finer* than partition B if each subset of A is a subset of one of B's subsets. It follows that B is *coarser* than A. The coarsest level consists of one subset which represents the entire population Φ. The first level consists of $\#\Phi$ subsets in its partition – one per $\gamma \in \Phi$. Figure 2.1 presents a partitioning on some population Φ such that each subset of the partition consists of elements with the same attribute value. This requires just one mapping. Figure 2.2 is an example of random partitioning, where defined subsets may not be homogeneous in terms of attribute value. Consequently, additional partitions are needed on subsequent levels of Φ to get to its most compact state, that is, the state in which further partitions are not possible.

2.4 The Well-Ordered 2^N-ary Tree Mapped onto Φ

The mappings described in the previous sections are not restricted in terms of the number and size of the subsets in a particular partition. This can lead to irregular mappings. If the number of subsets in a partition, and the number of objects in a subset is consistent, an implementation is much easier to devise. This consistency can be considered as a regular mapping. A tree-based representation scheme is one method which provides this regularity and consistency.

A number of interesting properties exist for any tree. A tree is a connected, acyclic directed graph where the root node has indegree 0, while other nodes have indegree 1. This implies a linear ordering between the root node and its successors.

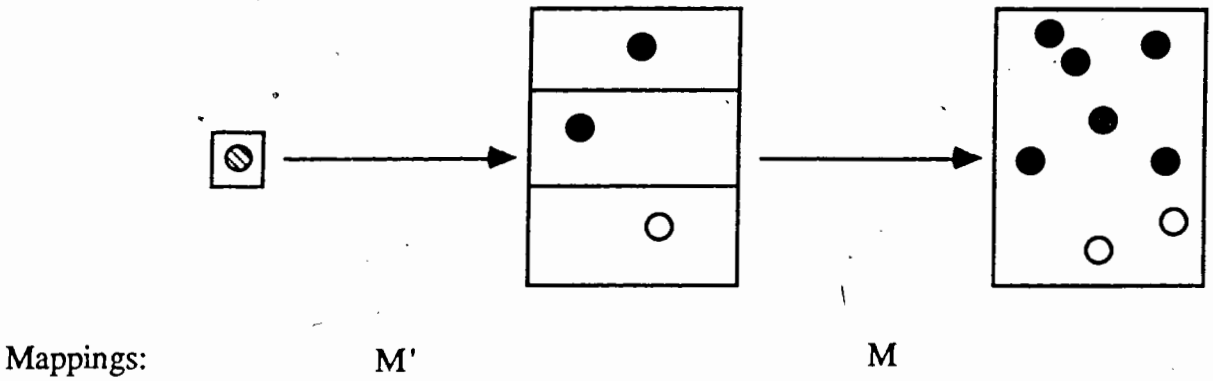


Figure 2.1: Efficient partition of a population Φ

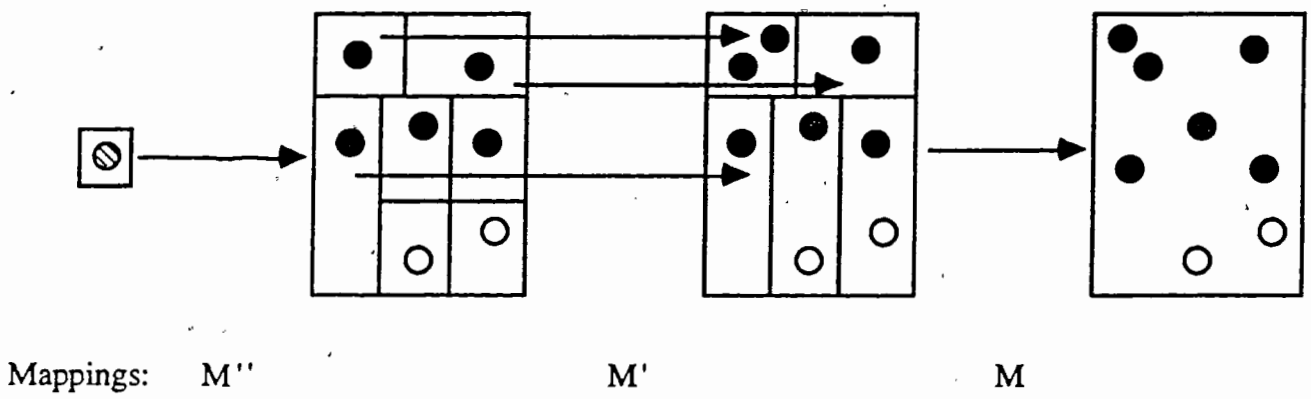
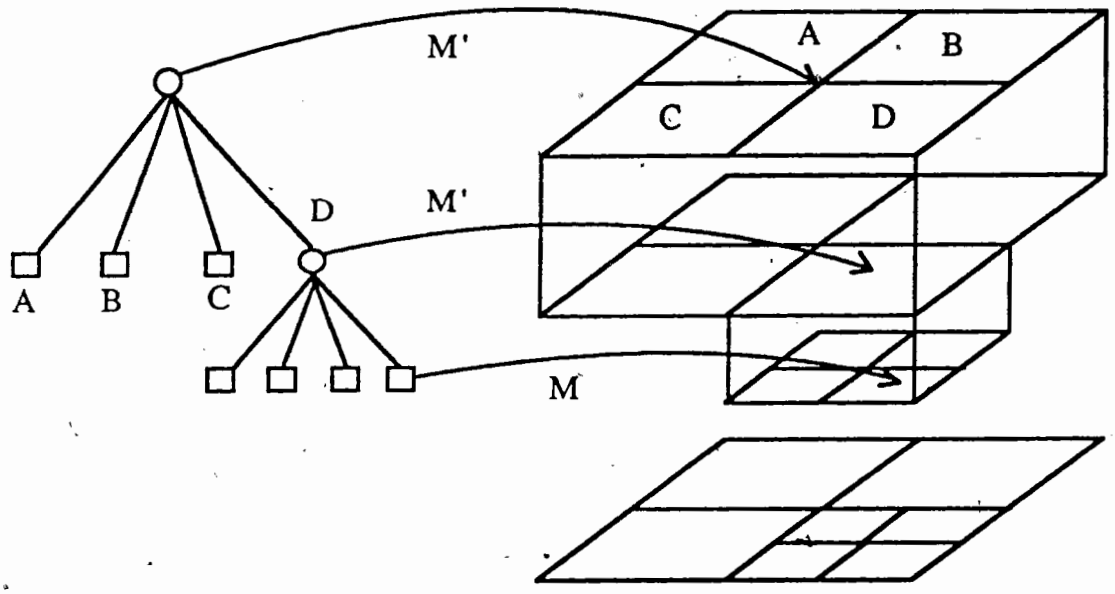


Figure 2.2: Random partition of some population Φ

In general, the direct successor nodes of any nodes are also linearly ordered. Each subtree in the tree also has a first element. Trees are then well-ordered structures.

Consider the case of some 2^N-ary tree T where all interior and leaf nodes are of indegree 1, and the root and interior nodes are of outdegree 2^N. If there exists some mapping M from the leaves of T to the partition of Φ for some $\mathbf{P}_{\Phi_\alpha}$, then T is a tree for $\mathbf{P}_{\Phi_\alpha}$. The value of each leaf in T is based upon the value of its particular partition component ρ (where $M(leaf) = val_\alpha(\rho)$). The interior nodes of T map to those partitions ρ where there is no homogeneity of $v \in V_{\Phi_\alpha}$ via the mapping function M'. The root of T maps to all of Φ. Function M is the restriction of M' to the leaves of T. Figure 2.3 is an example of such a regular mapping on a population. It is essentially the same as that of Figure 1.1. However, in the latter case, it was assumed that the 2^N-ary tree was valid only as a graphic representation scheme. The subsequent discussion has shown the scheme as a very general technique which can function through general or specific mappings.

The existence of such a mapping between the well-ordered 2^N-ary tree and Φ implies that Φ is also well-ordered. It is this property of well-order which is a critical factor in the fast execution times offered by the set representation.



Mappings: M : leaf nodes ---> population (object level)
M' : interior nodes --> population (non-object levels)

Figure 2.3: Mapping of tree T to a population Φ

Chapter 3 Analysis of Parallel Time-Steps for the Quadtree

The following analysis concerns itself with determining the number of time slices that are required to execute a binary operation involving two quadtrees, and a collection of processors. This study exploits the parallel nature of the quadtree. Although it is only the quadtree that is considered, the same type of reasoning can be applied to other 2^N-ary trees.

Some preliminary notation and conventions must be assumed. Given any quadtree, there are a total of M nodes in the structure. The number of processors in the system is P. Let i correspond to a particular leaf level. L_i denotes the nodes at level i in the tree. Figure 3.1 presents a particular situation where there are $l+1$ levels in the tree, and numbering of levels begins at the root node (L_0). The leaves are represented by L_l.

There are two cases which must be considered in this analysis.

- $P \geq 4^l$ (the number of nodes at leaf level l);
- $1 \leq P < 4^l$.

3.1 Worst case on a binary operation

A double traversal of the tree is necessary, from the root node to the leaves, and back to the root. This latter traversal is required to process any waiting tasks that

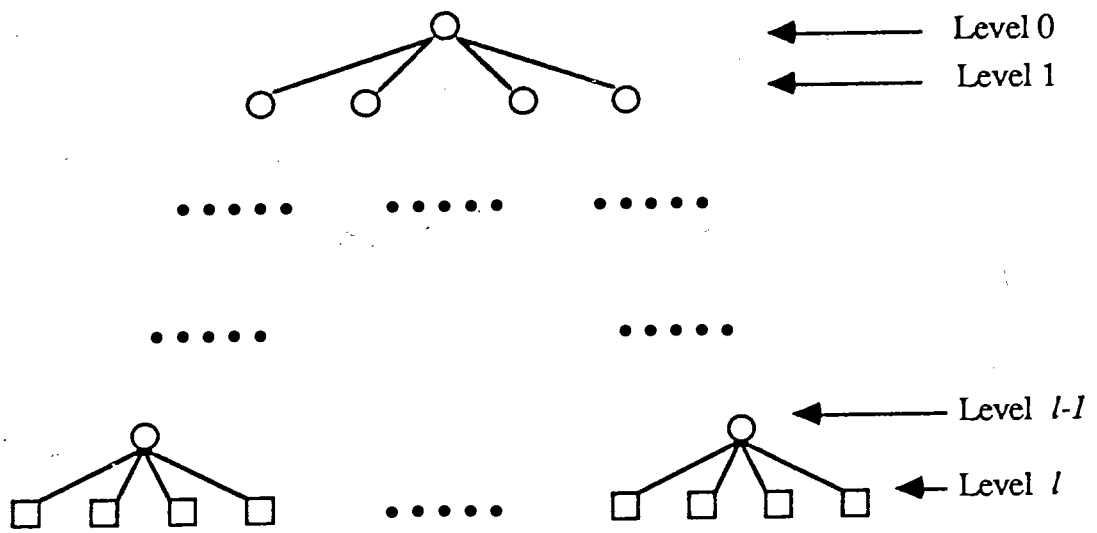


Figure 3.1: Typical quadtree for the time-step analysis

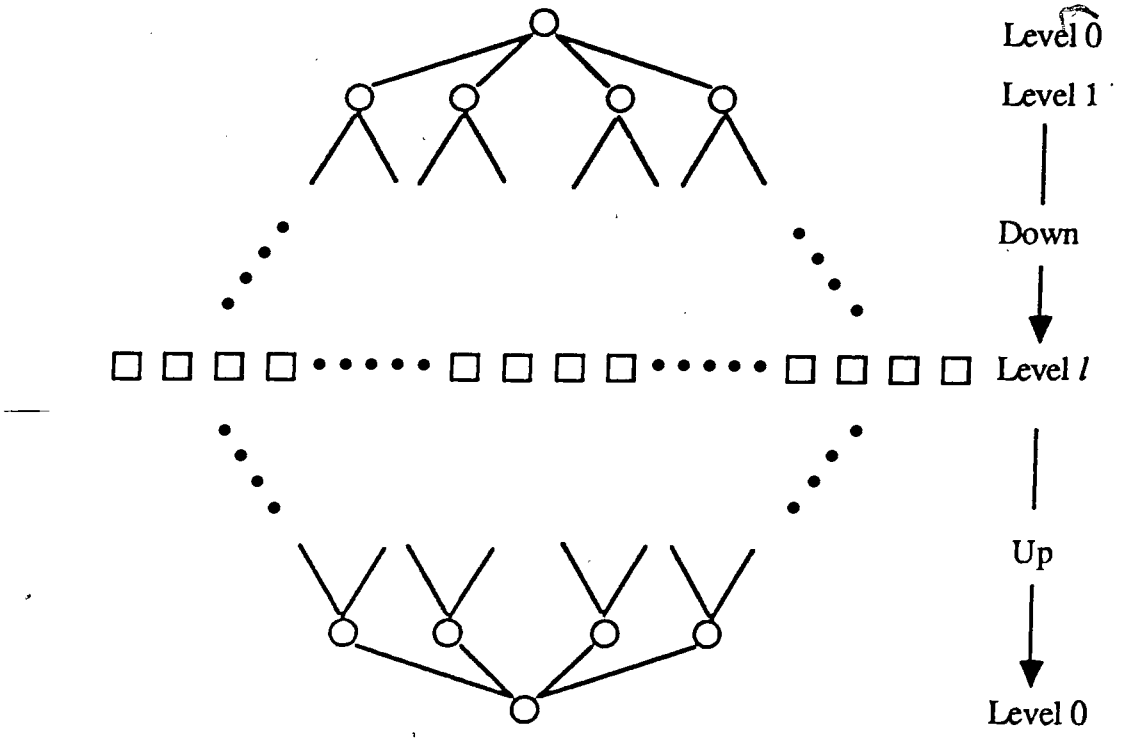


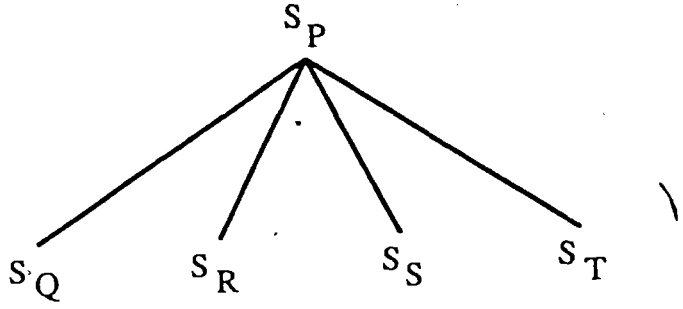
Figure 3.2: Effective double traversal of a tree for a binary operation

are dependent upon the results at lower levels in the tree. In general, tasks at level $i, 0 \leq i < l$, must wait for the results at level $i+1$ to become available. This worst case situation can be envisioned as shown in Figure 3.2. In going from L_0 to L_l, tasks are created as the processing proceeds from the root to the leaf nodes. The bottom half of the figure represents the progression back to the root node after the leaf processing has been completed.

Let $S_{P,l}$ correspond to the time required to finish a complete double traversal of an l-level tree with P processors. The start time begins with the root node at time 0. Let S_j denote the time when some node j in the tree is finished with its processing. The first claim is that the minimum amount of time $\overline{S}_{P,l}$ required to perform this two-way traversal is:

$$\overline{S}_{P,l} = 2l + 1. \tag{3.1}$$

In a tree, or subtree, with nodes P, Q, R, S, and T, the times S_Q, S_R, S_S, and S_T may be equal. However, S_P may not equal S_Q, S_R, S_S, or S_T . In other words, a



parent node must complete processing before its child nodes can begin their processing. The minimum time required to process any level will be attained if all nodes at a level can complete their processing at the same time. There are $(l+1)$ time levels going down the tree, and l time levels going back up. These two values provide the minimum time $\overline{S}_{P,l}$ of Equation 3.1.

Taking this result for minimum time, the minimum number of processors P_{min} to achieve this result is equal to the number of leave nodes M_l:

$$P_{min} = 4^l, \; M_l = 4^l \tag{3.2}$$

The next task is to find $\overline{S_{P,l}}$ for any quadtree with l levels, $l \geq 0$:

$$S_P = \mathcal{F}(P, l) \tag{3.3}$$

Only those situations where $P = 2^x$, for any integer $x \geq 0$, are considered here. The first case involves $M_l = 4^l = 2^{2l}$. If $x \geq 2l$, $S_{P,l} = S_{2^x,l} \triangleq 2l + 1$. At level l, only one time unit is required to process tasks at that level. Therefore, for all $m < l$, $M_m < M_l$. This provides the minimum time of $2l + 1$.

In the second case, $P = 2^x$, for any integer $0 \leq x \leq 2l$, Equation 3.3 still applies, but now consists of two components as given in Equation 3.4. Note that the case of $P = 2^{2l}$, present in the first situation, can also be considered here.

$$S_{P,l} = 2\mathcal{G}(P, l-1) + \mathcal{H}(P, l) \tag{3.4}$$

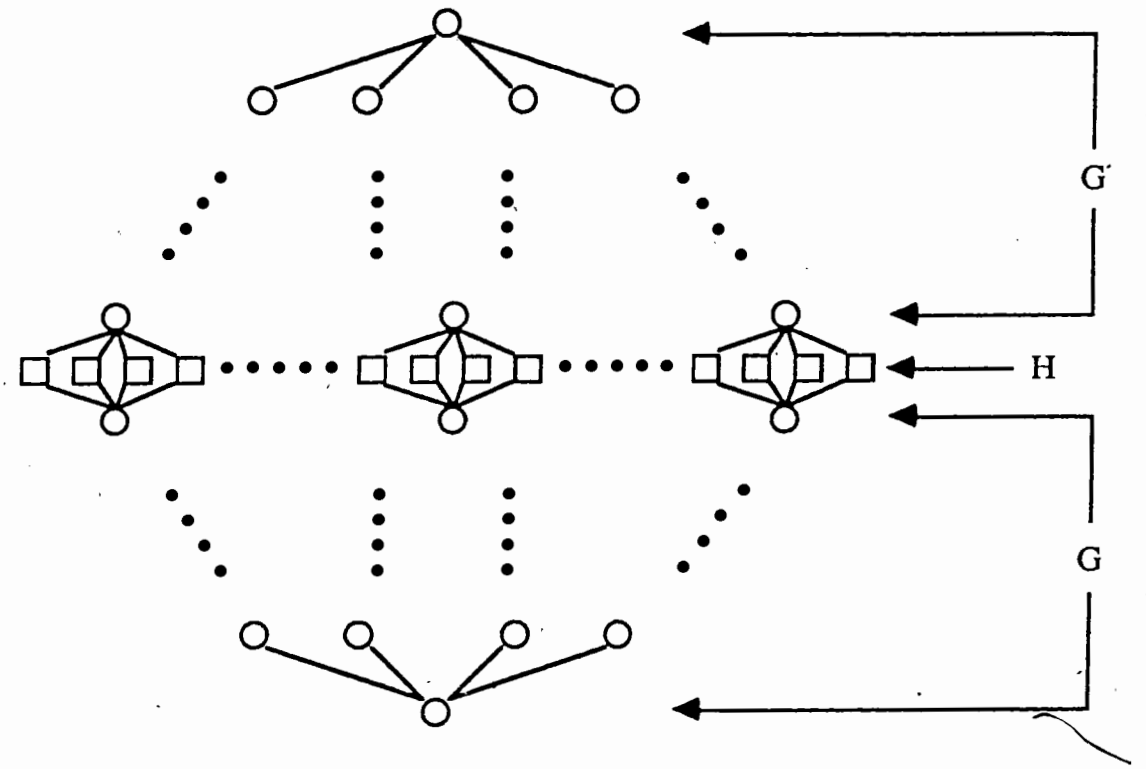


Figure 3.3: Situation for the case of $P = 2^x$, $0 \leq x < 2l$

This presents the situation given in Figure 3.3. Now two additional cases arise from this scenario. The first involves some odd integer x, $0 < x < 2l$:

$$\begin{aligned} S_{P,l} &= 2\sum_{i=0}^{l-1}\left\{\frac{4^i}{P}\right\} + \left\{\frac{4^l}{P}\right\} \\ &= 2\left(\sum_{i=0}^{\frac{x-1}{2}} 1 + \sum_{i=\frac{x+1}{2}}^{l-1} 2^{2i-x}\right) + 2^{2l-x} \\ &= 2\left(\frac{x-1}{2} + 1 + \frac{2^{2l-x}-2}{3}\right) + 2^{2l-x} \\ &= x + 1 + \frac{2\left(2^{2l-x}-2\right)}{3} + 2^{2l-x} \\ &= x + 1 + \frac{2}{3}\left(2^{2l-x}\right) - \frac{4}{3} + 2^{2l-x} \end{aligned}$$

$$S_{P,l} = x + \frac{5}{3}\left(2^{2l-x}\right) - \frac{1}{3}, \text{ for } 0 < x < 2l, \text{ and } x \text{ is odd.} \tag{3.5}$$

In the case of some even integer x, $0 \leq x \leq 2l$, the following exists:

$$\begin{aligned} S_{P,l} &= 2\sum_{i=0}^{l-1}\left\{\frac{4^i}{P}\right\} + \left\{\frac{4^l}{P}\right\} \\ &= 2\left(\sum_{i=0}^{\frac{x}{2}} 1 + \sum_{i=\frac{x+2}{2}}^{l-1} 2^{2i-x}\right) + 2^{2l-x} \\ &= 2\left(\frac{x}{2} + 1 + \sum_{i=\frac{x+2}{2}}^{l-1} 2^{2i-x}\right) + 2^{2l-x} \\ &= x + 2 + 2\sum_{i=\frac{x+2}{2}}^{l-1} 2^{2i-x} + 2^{2l-x} \\ &= x + 2 + \left(\frac{2}{3}2^{2l-x} - \frac{8}{3}\right) + 2^{2l-x} \end{aligned}$$

$$S_{P,l} = x + \frac{5}{3}2^{2l-x} - \frac{2}{3}, \text{ for } 0 \leq x \leq 2l, \text{ and } x \text{ is even.} \tag{3.6}$$

The only difference between Equations 3.5 and 3.6 involves the constant terms $\frac{1}{3}$ and $\frac{2}{3}$, respectively. A general equation for $P = 2^x, 0 \leq x \leq 2l$ can then be defined:

$$S_{P,l} = \frac{5}{3}2^{2l-x} + \frac{x \bmod 2}{3} + x - \frac{2}{3} \tag{3.7}$$

Figure 3.4 presents a plot of Equation 3.7 for $1-, 2-, 3-$, and $4-$ level trees, where the number of processors $P = 2^i$, $0 \leq i \leq 8$.

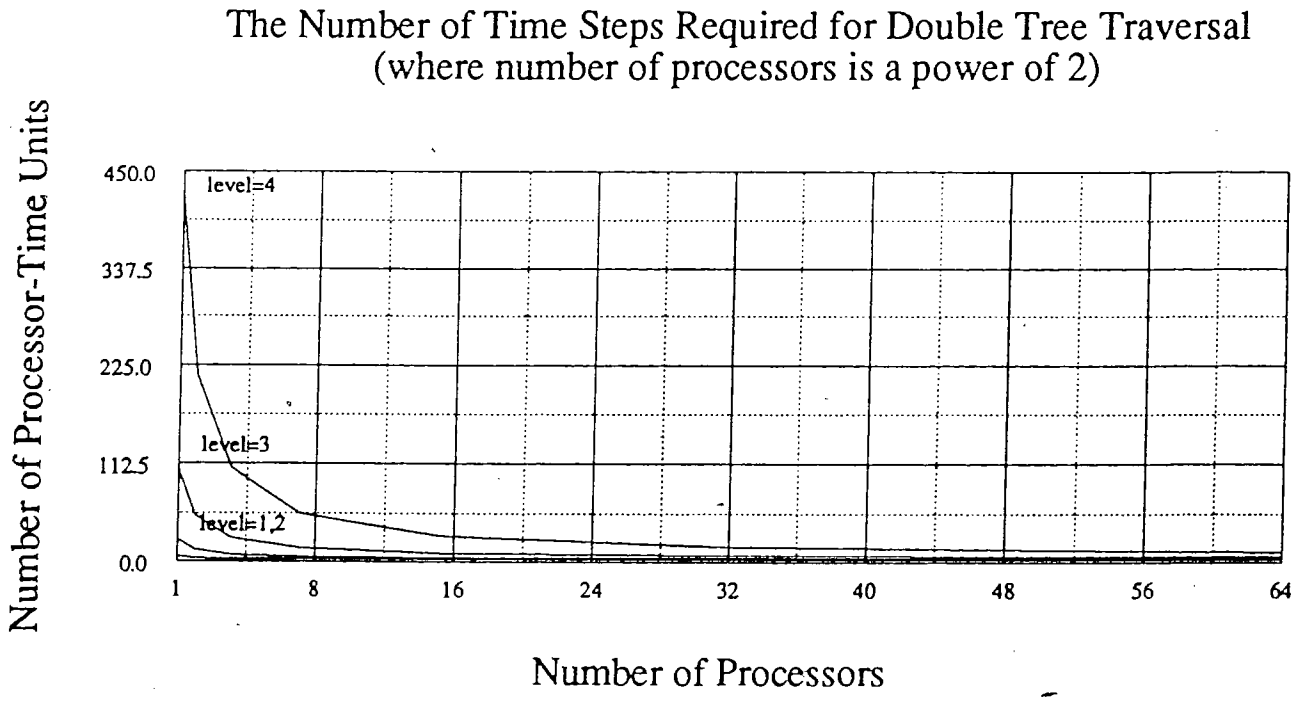


Figure 3.4: Number of time steps for 2^i processors

3.2 The case of $P \neq 2^x$, for any integer $0 \leq x < 2l$

In the case of $P \neq 2^x$, for any integer $0 \leq x < 2l$, and $l+1$ corresponds to the number of levels in the tree ($l \geq 0$),

$$S_{P,l} = \mathcal{G}(P, l-1) + \mathcal{H}(P, l) + \mathcal{E}(P, l) \tag{3.8}$$

One of the parameters of $\mathcal{G}$ is the number of levels in the tree where $P \geq 4^i$, $0 \leq i \leq l-1$. $\mathcal{G}$ provides the number of processor-time units necessary for the traversal of levels 0 to $l-1$. $\mathcal{H}$ returns the number of time units required to complete the processing of level l. Functions $\mathcal{G}$ and $\mathcal{H}$ do not consider the situation where time units can spread over adjacent levels in the tree. An earlier condition stated that for any subtree, if the root node of this subtree was executed at T_i, the children of this node could not execute at T_j, if $T_i \geq T_j$. This implies that those nodes which do not belong to a particular subtree, can execute at some time T_j, where T_i may or may not be less than, or equal to T_j. The functions $\mathcal{G}$ and $\mathcal{H}$ will provide exact results in the number of time units necessary to process a level in the tree, but in terms of final time count, the carry-over to the next level will not be considered. This carry-over may come about in the following fashion. If there are X processors available, and the current processing level m requires Y processors, such that $X > Y$, then it is possible to use the remaining $X - Y$ processors for the level $m+1$ at the same T_i as the Y processors at level m. This would only occur if there are $X - Y$ nodes at level $m+1$ which do not belong to the subtrees whose root nodes are being processed by the $X - Y$ processors. To accommodate this situation, it is necessary to apply an error function $\mathcal{E}$ which returns the number of levels in the tree where there may have been some overcompensation in the functions $\mathcal{G}$ and $\mathcal{H}$.

$$\mathcal{E} = \overline{S}_{P,l} - l_{P_*} \tag{3.9}$$

Here, l_{P_*} corresponds to the number of levels in the double traversal where the number of processors P exceeds L_i, for some level i.

The components of Equation 3.8 are defined as follows:

$$\mathcal{G}(P,l-1) = 2\left(\sum_{i=0}^{\lfloor \log_4 P \rfloor} 1 + \sum_{i=\lfloor \log_4 P \rfloor + 1}^{l-1} \left\lceil \frac{4^i}{P} \right\rceil\right) \tag{3.10}$$

$$\mathcal{H}(P,l) = \left\lceil \frac{4^l}{P} \right\rceil \tag{3.11}$$

$$\mathcal{E}(P,l) = \overline{S}_{P,l} - 2\lceil \log_4 P \rceil$$

$$\mathcal{E}(P,l) = 2l - 2\lceil \log_4 P \rceil + 1 \tag{3.12}$$

After consolidating the terms in Equations 3.10, 3.11, and 3.12, a general equation for $S_{P,l}$ can be presented, with the restriction that $P < 4^l$:

$$S_{P,l} = 4\lceil \log_4 P \rceil + \left\lceil \frac{4^l}{P} \right\rceil + 2\sum_{i=\lceil \log_4 P \rceil}^{l-1} \left\lceil \frac{4^i}{P} \right\rceil - (2l+1) \tag{3.13}$$

Figure 3.5 presents a plot of Equation 3.13 involving four trees of different levels. When compared to Figure 3.4, Figure 3.5 indicates that the error function provides a reasonable approximation of two-level time-unit carry-over.

3.3 Cumulative idle processor time

Another criterion which must be considered in determining the effectiveness of a parallel-oriented 2^N-ary tree representation is the amount of time that is consumed by processor inactivity. As in the previous section, a tree of $l+1$ levels, and P processors are involved in the analysis. The double traversal is still required.

In general, the amount of idle or wasted processor time T_W is given by:

$$T_W = S_{P,l} \cdot P - S_{1,l} \cdot 1 \tag{3.14}$$

In the case of $P \geq 2^x$, $x \geq 2l$, T_W is given by:

$$T_W = (2l+1)P - \left(\frac{5}{3}4^l - \frac{2}{3}\right) \tag{3.15}$$

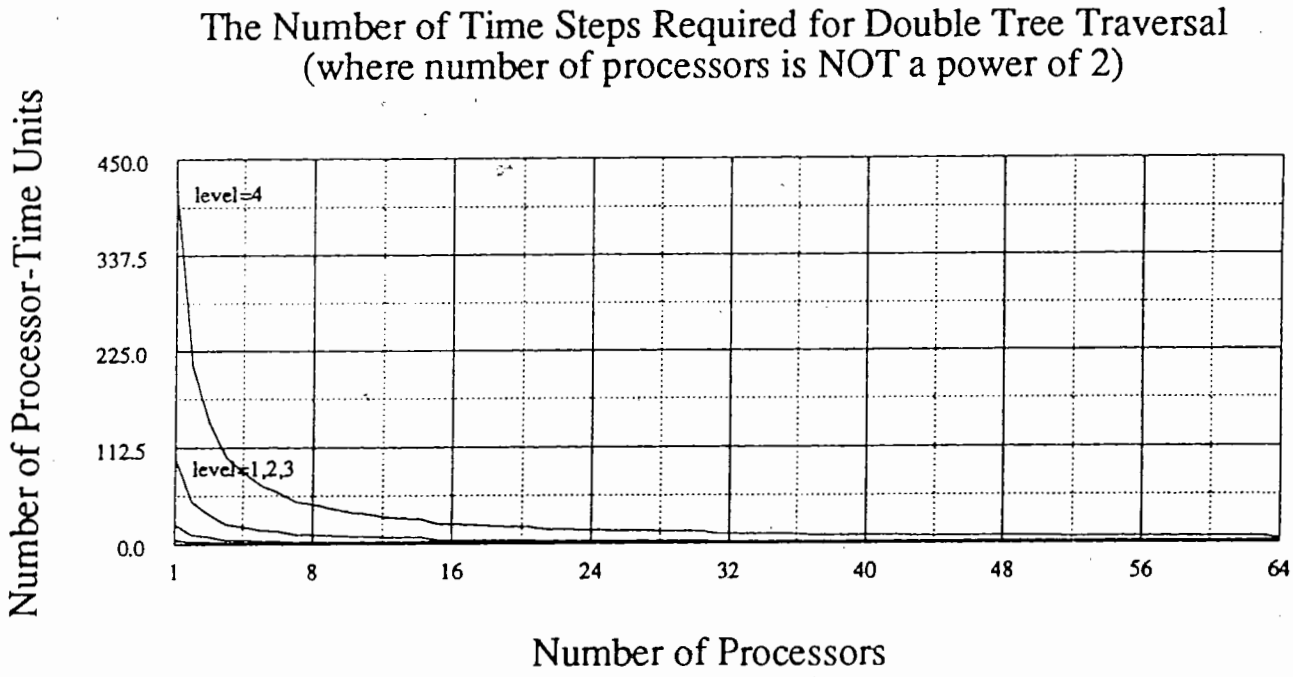


Figure 3.5: Number of time steps for $P \neq 2^i$

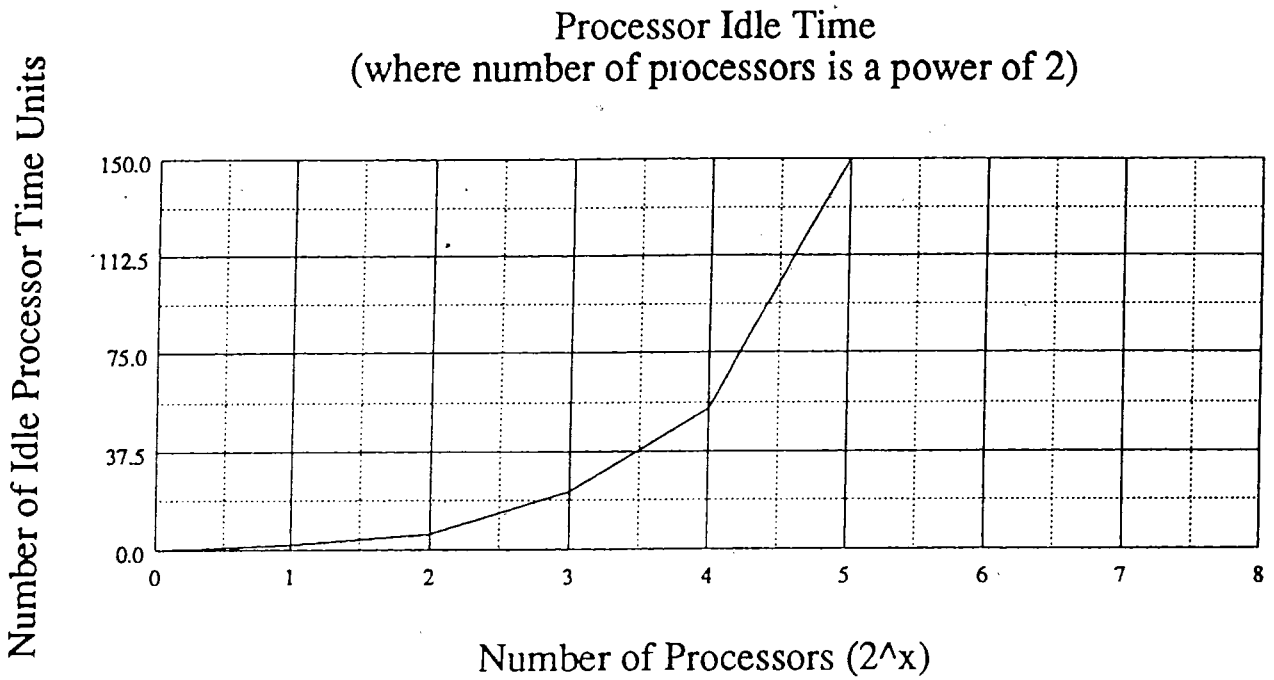


Figure 3.6: Processor idle time for $P = 2^i$

With $P = 2^x$, $0 \leq x < 2l$, T_W is given by:

$$T_W = 2^x \left(x + \frac{(x \bmod 2)}{3} - \frac{2}{3} \right) + \frac{2}{3} \tag{3.16}$$

One interesting point about Equation 3.16 is that the number of levels in the tree is not involved in the calculation. This is a reflection of the fact that $P = 2^x$, and that there will be processors idle at those levels in the tree where $L_i < P$, $0 \leq i < x - 1$. Figure 3.6 presents a plot of Equation 3.16 for $P = 2^i$, $0 \leq i \leq 5$.

In the case involving $P \neq 2^x$, Equation 3.14 can again be used as the basis for the calculation. The first term of the equation can be replaced with Equation 3.13. However, the presence of ceiling and logarithmic terms in the latter makes reduction of the equation difficult.

Chapter 4 A Topologically-Derived Architecture for the 2^N-ary Tree

The structure of this first architecture is based on the 2^N-ary tree topology. The hardware incorporates layers of processing elements and interconnection networks in a manner similar to the levels of a tree. It is the software that generates the child tasks which must be sent to the slave processors for execution. References involving a computer graphics application are again used.

4.1 System Components

Figure 4.1 presents an overall view of the components which make up this architecture. Since it is the array construction which is of primary concern here, only a general overview of the other components is provided.

4.1.1 User Interface UI

The **UI** provides a user friendly environment from which commands are formatted in a manner acceptable to the Master Controller **MC**.

4.1.2 Staging Memory SM

The **SM** functions as a buffer between the external data source and the primary component of the architecture, the Processor-Interconnection Network Array

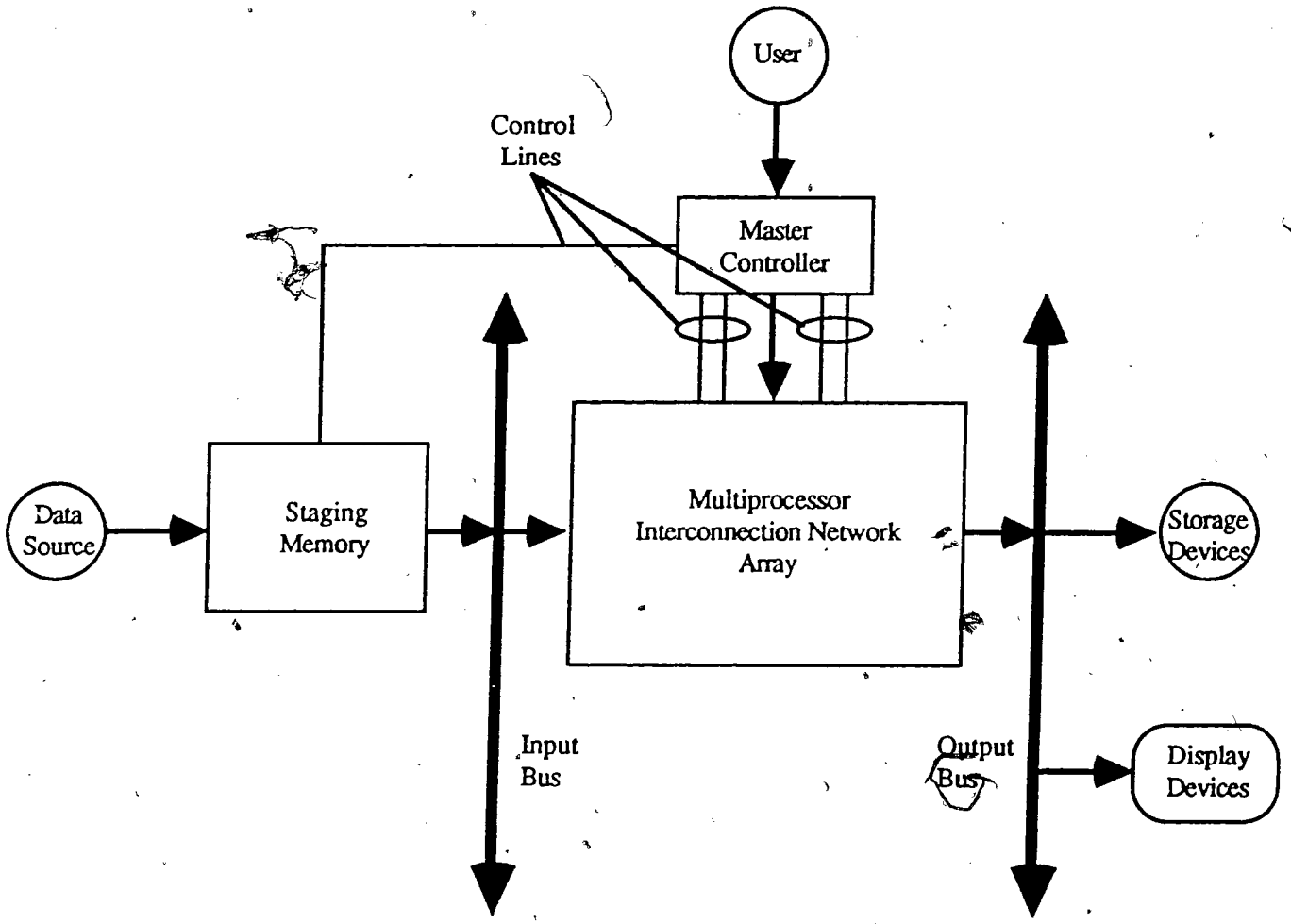


Figure 4.1: Overall system view of the architecture

P-INA. This **SM** formats raw data into a sequence which can be accepted and stored in the array. If the data has already been formatted, it is passed directly to the array.

The unit consists of a memory component and a central processing unit (**CPU**). Due to the wide variety of applications which may use this architecture, there are numerous external data formats. It is unreasonable to assume that one formatting mechanism can accommodate all possible cases. The tremendous flexibility of software control makes it the desired alternative over some hardware-based scheme. This requires that random access memory be available to the **CPU** to retain the necessary formatting instructions.

A question now arises based upon economics. In most cases, the users of such a system view it as a database of information, and are concerned with the speed at which queries can be resolved. The time required to set up the database is not as critical. Of course, unbearably long delays in set-up time are unacceptable, regardless of the situation. With this in mind, the user may require fast throughput for all components. If cost is not a concern, cache memory can be included in the **SM**. The instructions necessary for the current formatting operation can be brought into this cache, along with blocks of the input stream.

The control signals from the **MC** to the **SM** provide information relating to the type of formatting required, the complexity of the 2^N-ary tree, and data size. The **SM** uses this information to format the data into blocks of values which reflect the 2^N-ary nature of the data at its lowest level (in the graphics application this would be the picture element, or *pixel*). This step generates the finest level of subtree groupings in the tree. As formatting continues, these packets are sent out onto the array's input bus. For any image and configuration, there is one row in the array which represents the actual $(l-1)^{st}$ level of its tree, where l is the number of levels in the tree. It is this row's Processor Controller **PC** which takes the packets off the bus, and further formats the data to reflect the actual sibling node/processing element correspondence.

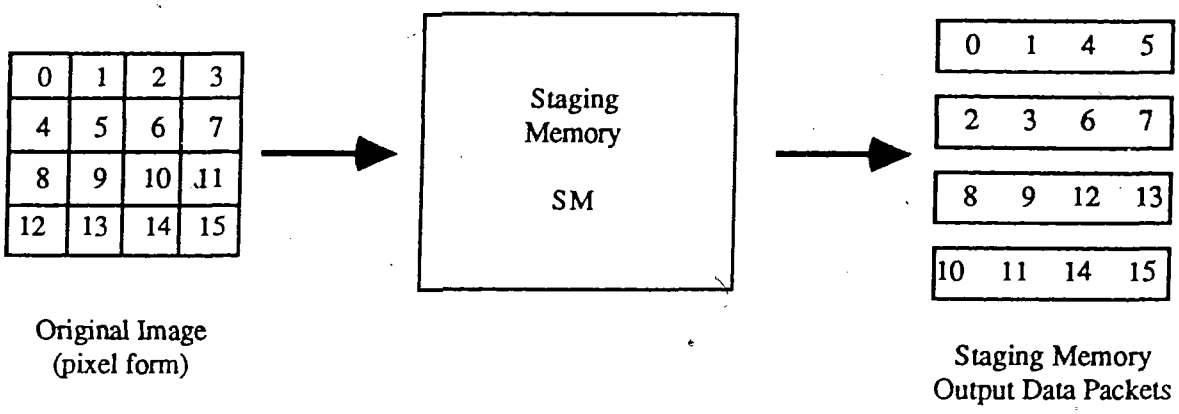


Figure 4.2: Data input sequencing.

Figure 4.2 shows the case of a simple 4 pixel by 4 pixel digitized input image that is stored on tape as a sequence of bit-intensity values. With the given numbering convention, and a request for a sequence acceptable for quadtree generation, the SM provides the given output packets which are then placed on the **P-INA** input bus.

4.1.3 Processor-Interconnection Network Array P-INA

The array consists of alternating rows of processors elements Ps and interconnection networks ICNs. A general view of the array is shown in Figure 4.3. The basic premise is that each row of processors represents a level of the 2^N-ary tree. The nodes of the tree are then mapped to the processors in this array. The interconnection networks, which link consecutive rows of processors together, are analogous to the edges which link a parent node in a tree to its child nodes.

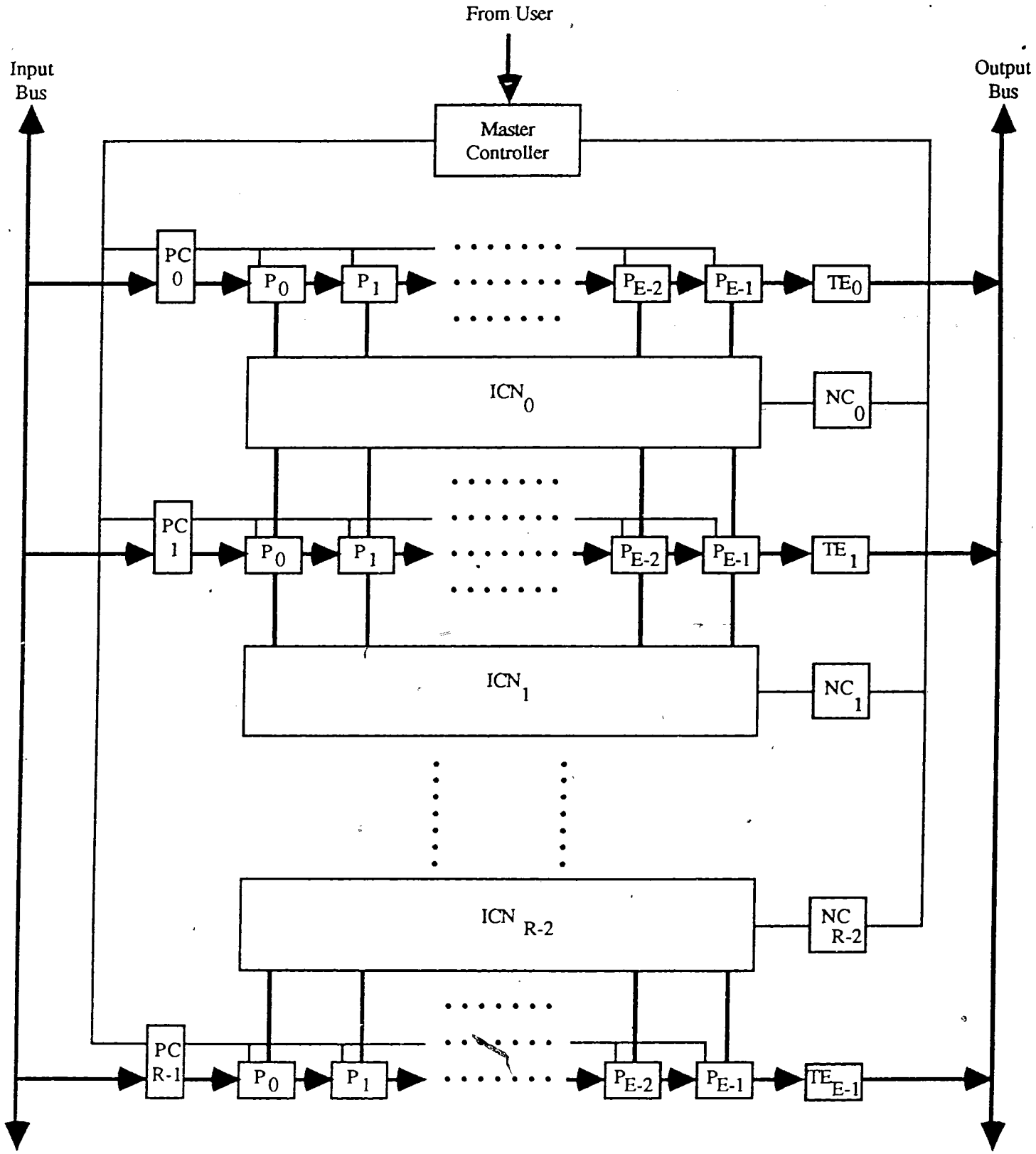


Figure 4.3: The processor-interconnection network array

Master Controller MC

The **MC** is a processor which, although not part of the array per se, provides a significant level of control to the entire structure. Once a request enters the system via the user interface, the **MC** can initiate any number of activities. If data formatting is necessary, control signals from the **MC** are sent to the **SM**'s processor which, in turn, activates its own set of routines.

The **MC** also provides the array with sufficient control information to build and manipulate the trees. Some of this information involves tree size, processor-row entry level, network switch settings, and synchronization markers.

Associated with the **MC** is primary memory that contains all of the tree manipulation routines, standard network switch settings, and functions to generate new settings. Some of these operations are specific for the network controllers, the processor controllers, or for the processing elements themselves.

Processor Controller PC

Rather than have the **MC** communicate directly with the processing elements of the array, which would lead to a communication bottleneck, a level of **PC**s is introduced to the system. There is one **PC** per row of **P**s. This allows the **MC** to allocate tasks to a manageable number of **PC**s rather than to the numerous row processors. Synchronization of tasks between the **MC** and the row processors is also simplified by the inclusion of the **PC** component. Each **PC** has local memory which is used to store the tree operations. However, this local memory does not contain all functions, but only those instructions which are necessary to solve the current user request. These are copied to each **PC**'s memory from the **MC**. This serves a number of purposes. Contention is not a factor here when each **PC** has its own memory to store instructions. The algorithms for 2^N-ary tree manipulation are reasonably straightforward such that their programs are not very large. If speed is a major concern, then cache memory can be used by each **PC**. Instruction

retrieval and execution will be very fast. Each row's **PC** broadcasts the necessary instructions to the **P**s under its control for subsequent execution by these **P**s.

Control lines exist between each **PC** and the **MC**, and between adjacent **PC**s. This allows for the necessary synchronization between levels of row processors.

Row Processor P

The simple tree algorithms require that the **P**s need not be complex. For the majority of cases involving this application, stock microprocessors with the hardware capabilities implied by the processing element of Figure 4.4 are sufficient. Chapter 4.3 presents a case where these simple microprocessors may be replaced with more powerful elements.

Each **P** consists of the following components. A logic gate controls the direction of data through the processor. Data can flow from "top" to "bottom" or vice versa in the array. The logic gate maintains this directional consistency. The gate also channels data to the arithmetic logic unit **ALU**.

The operations taking place within a processor are under the direction of the control unit **CU**, which receives its signals from the row's **PC** via the row control lines. When a program segment from the **PC** is to be stored in the element's local memory, it is the **CU** which oversees the storage process. The logic gate receives the necessary signals from the **CU**, as do the **ALU**, and accumulator.

At the **ALU**, the appropriate operations are applied to the data entering the element to satisfy the requirements of the user query. For example, with a binary operation on a 2^2-tree, the four child values of a node enter the element sequentially, and have the operation applied to them. The intermediate result of applying this operation on two of the child values is routed to the accumulator and a latch **L**, which is used to hold a value indefinitely until it receives the appropriate signal from the **CU**. The temporary result in the accumulator is passed to the **ALU** for the execution of the next instance of the operation. Once all four input

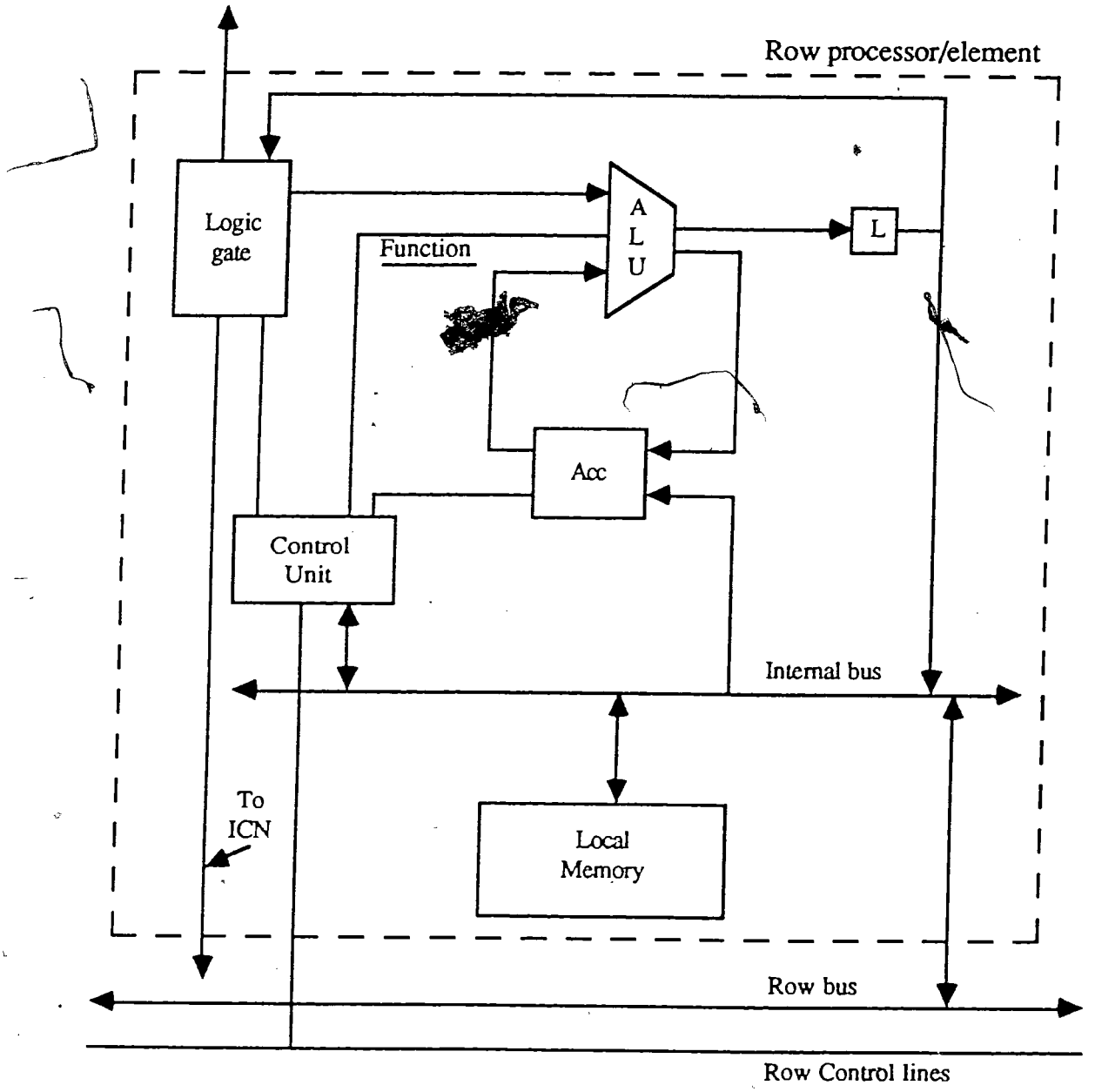


Figure 4.4: The row processing element

values have been processed, **L** releases the result to an internal bus from where the value will be stored in the element's local memory as a node of the resultant tree.

The presence of this internal bus permits data to travel between the local memory, accumulator, and the external row data bus. The external bus is critical in that program segments can be passed from the **PC** to each **P** under its control. In the case of data realignment between **P**s of the same row, node values can be moved from one processor to another.

Interconnection Network Controller NC

Each **NC** provides the necessary control mechanism for its respective interconnection network. This mechanism is implemented through the use of the appropriate control programs, which are initially stored in the **MC**. In this respect, the **NC** is similar to the **PC** in that it is not necessary for the unit to store all network control programs. Instead, the **MC** provides only those programs which are needed to complete the current user request. The **MC** also makes available to each **NC** the switch settings that are required for the network under its influence. This implies than an **NC** has local memory. To speed up the execution process of the **NC**, the switch settings reside in a separate memory with the unit, apart from the program memory.

Each **NC** receives control signals from the **MC**, in addition to sending and receiving signals to and from the network.

Switching Element SE

The switching elements which make up the **ICN**s are simple 2x2 switches that can be in any of eight routing states. The four states permitting the passage of data from top to bottom of the array are shown in Figure 4.5. The remaining four states are mirror images of these states, which allow for data transfer from bottom to top. An **SE**'s state is determined by the settings which it receives from its **NC**.

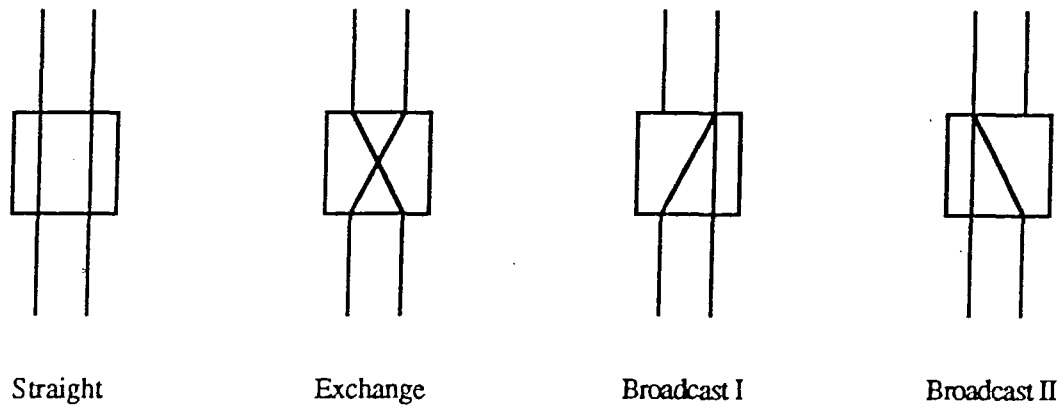


Figure 4.5: Switching states

The broadcast stages are needed to provide the mapping of root and child nodes of the logical tree structure to the Ps of the array. The broadcasting from successive levels of SEs in the ICN eventually generates the communication links between a root P and its 2^N child Ps in the adjacent row. With this broadcasting, only one cycle is needed to send out the data values. However, in the case of data entering a SE, the element can only read one value at a time. Therefore, for the two-input case, two cycles will have to be used to read these values through the element.

4.2 The Interconnection Network ICN

The previous section described the components of the P-INA. However, it is the interconnection scheme linking adjacent rows of processors that allows this architecture to succeed as a hardware alternative for the 2^N-ary tree representation. The nature of the representation dictates that the network be reconfigurable, that it support broadcasting of information, and that two-way communication exist between levels of processors.

One class of networks which fulfills these requirements is the delta network [9,47]. The components of delta networks include:

- 2^T input links and 2^T output links;
- T stages $S_0, \ldots, S_{T-1}$ and;
- for each stage t, $0 \leq t < T$, there are 2^{T-1} 2x2 crossbar switches.

4.2.1 Network candidates

The simple ExF crossbar switch is a candidate for the **ICN** since it has some attractive features, the primary of which is its $O(\log E)$ or $O(\log F)$ gate delays for the switch setting process. A ExF crossbar permits the connection of E processors to F other processors. It is usually discussed in the context of linking each of E processors to F memory units. The one-to-one and one-to-many mappings between stages is also favourable. Unfortunately, the number of interconnections are extensive, and the complexity of the switch increases as E and F become large. The number of gates is proportional to EF.

Of the delta-class network, three interesting options are the Beneš network [6], the Batcher sorting network [5], and the omega Ω network [22]. The former has the same capability as the crossbar switch but only requires $O(E \log E)$ gates for a ExE network. The time required to pass data through the network is $O(\log E)$. The negative aspect of the Beneš scheme is the difficulty in setting up a particular alignment, the complexity of which is $O(E \log E)$.

Like the Beneš network, the Batcher sorting network has similar capabilities to the crossbar switch, but only requires $O(\log^2 E)$ time units for data passage through the network since it is set up as the data flows through it. However, this network requires $O(E \log^2 E)$ gates.

The Ω network is the preferred scheme since it is relatively inexpensive to implement and has only $O(\log E)$ gate delays to control and transmit data. There

are $O(E \log E)$ gates in the network. The topology rules for the Ω network can be stated by the following definition.

Definition 4.1 *Given* $E = 2^T$ *input lines, represent each line in its binary encoded form,* $E_{T-1}E_{T-2} \cdots E_0$. *Then its corresponding output link is* $E_{T-2}E_{T-3} \cdots E_0E_{T-1}$.

The ExE omega network consists of $T = \log E$ identical stages, each of which is made up of $\frac{E}{2}$ switching elements. One of the primary differences between this and the previous two networks involves the number of stages. The Ω network requires $\log E$, while the Beneš and Batcher networks need $\frac{\log E(\log E+1)}{2}$, and $2 \log E - 1$ levels, respectively. The logic involved in the three networks is different but the complexity of the switches is similar.

Figure 4.6 shows an example of an Ω network with $E = 8$ and $T = 3$ where the interconnecting links and a candidate configuration for a quadtree ($N = 2$) are shown.

4.2.2 Network configuration

From the previous subsection, there are $T = \log E$ stages, labelled $S_0, \cdots, S_{T-1}$, and $\frac{E}{2}$ switches per stage for an ExE Ω network. Under most circumstances, the one-to-one mapping which the Ω network provides is sufficient. However, with the 2^N-ary tree scheme, it is necessary for a one-to-2^N mapping. This can be accomplished by the broadcasting capabilities of the switching elements. Broadcasting for a 2^N-ary tree begins at S_{T-N}, and continues to S_{T-1} ($\prod_{T-N}^{T-1} 2 = 2^N$). The mapping from S_0 to S_{T-N-1} is one-to-one.

The number of possible mappings on an array is extensive. As an example, for some row r in the array where E processors map to E nodes, there are $\frac{E}{2^N}$ **P**s in row R_{r-1} which are the fathers of these processors. There are then $E - \frac{E}{2^N}$ **P**s in row R_{r-1} that are available for other mappings. To further increase the

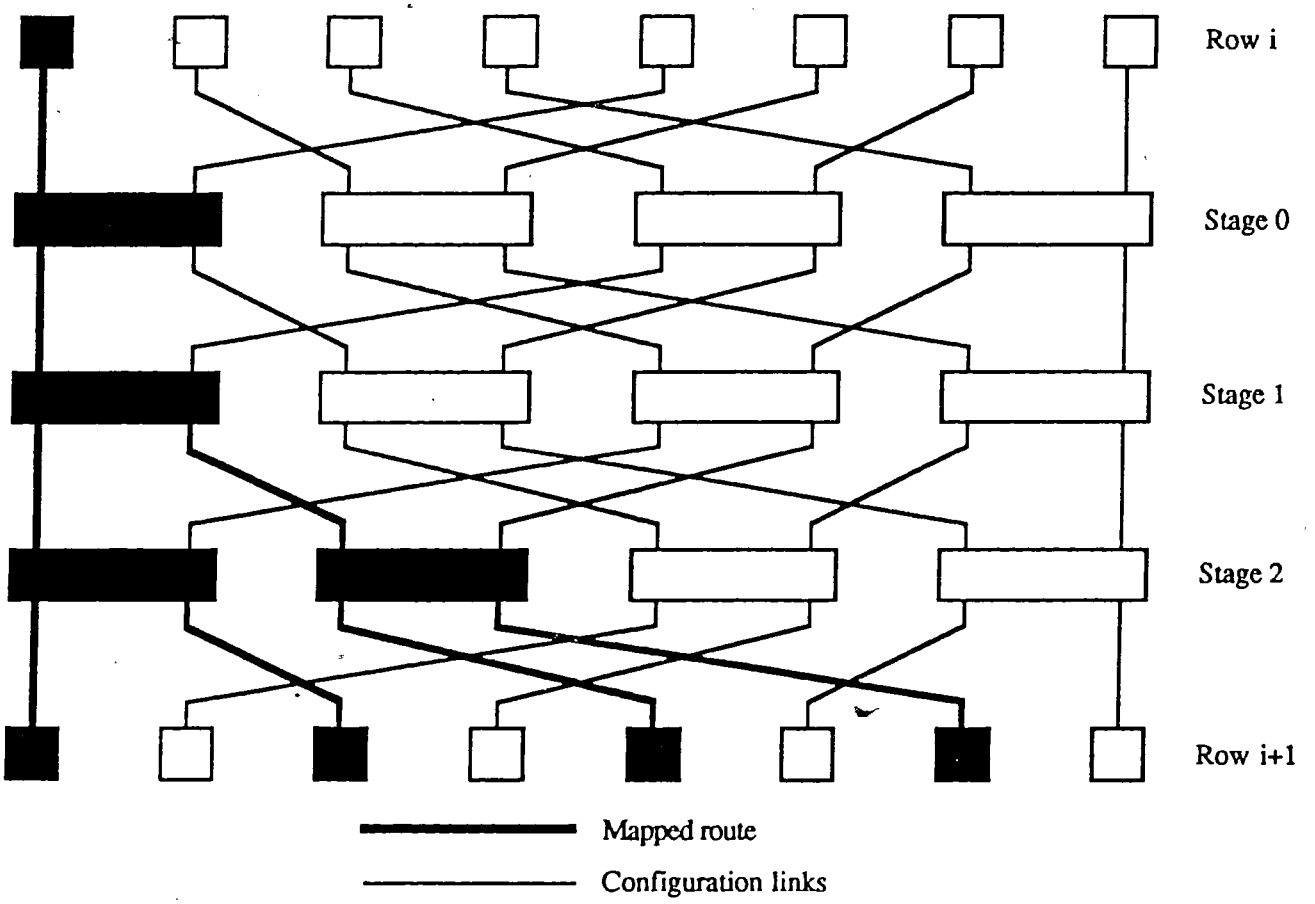


Figure 4.6: Example of a 8-processor/row Ω network

randomness of selection, any $\frac{E}{2^N}$ **P**s can be used. In effect, there are $\begin{pmatrix} E \\ \frac{E}{2^N} \end{pmatrix}$ possible choices for these parent processors.

4.3 Overview of the scheme

This architecture exploits two features of the 2^N-ary tree. For any subtree **S** with a root node r at level l in the tree, **S** is independent of the other subtrees with root nodes at level l. As we progress down through each level of the tree, a finer representation of the image is obtained. Therefore, for any image **I**, and its L-level tree representation, the subtree defined by the root node and the next i, $i < L$, levels still provides a valid definition of **I**.

The scheme takes some 2^N-ary tree **A**, and maps it onto a **P-INA** consisting of R processor rows (labelled $0, 1, \cdots, R-1$), each made up of E processors. Separating consecutive rows of processors are T stages ($E = 2^T$) of switching elements which make up the local **ICN**. T 2x2 crossbar switches ($E = 2T$) are present in each stage.

For any tree **A**, its root node (level L_0 in the tree) maps to a processing element in row R_0 of the array. 2^N processing elements in R_1 correspond to the 2^N children of the root node. The **ICN** between R_0 and R_1 utilizes the appropriate switch settings to realize this mapping. The nodes of L_2 of A are then mapped to the appropriate elements of R_2. Each of the elements of R_2 are linked to its parent processing element in R_1. This procedure continues until the entire tree is mapped onto the array. Of course this complete mapping of the tree will only occur if $L \leq R$.

Two features of such an organization are that the mappings for different instances of trees with equal N are the same, and synchronization of processing element execution is simplified. With the former, this leads to processing elements representing the same relative node for all similar instances. In comparing the

values for these different instances, it is not necessary to access some form of shared memory which can lead to contention, but instead, the element's local memory contains these values. Only two memory accesses are needed to load the appropriate values into the data registers of the processing element. In terms of synchronization, execution of the tree takes place one level at a time. Therefore, the processors of an active row begin the execution of their instructions at the same time. The next row of processors can then initiate their sequence of instructions once the processors of the previous row have completed their last instruction.

4.3.1 The incomplete mapping of the tree

When $L > R$, there are two approaches which can be taken. First, an incomplete tree mapping can be made on the array. The top R levels of the tree are mapped onto the array. However, instead of containing node information as in the previous $R-1$ rows, each of the mapped elements of the $(R-1)^{st}$ row of the array contain all of the information of their respective node's subtree for which it is the root node. Figure 4.7 has an example of just such an incomplete mapping, where $R = 3$, and $L = 4$. For clarity, the **ICN**s have not been included in the figure. This storage of information in the last row of the array is possible due to the existence of local memory for each element.

This solution is not as straightforward as expected. In the first $R-1$ rows, the processing elements perform relatively simple tasks such as the routing and queueing of data, and register comparisons. This can all be performed with the type of processing elements described in Chapter 4.1.3. However, the elements which occupy the $R-1^{st}$ row of the array now must process each level of its stored subtree. This increased complexity in processing can be accommodated in two ways. All of the processing elements can be replaced with more powerful microprocessors. This maintains the homogeneity of the array, at the expense of

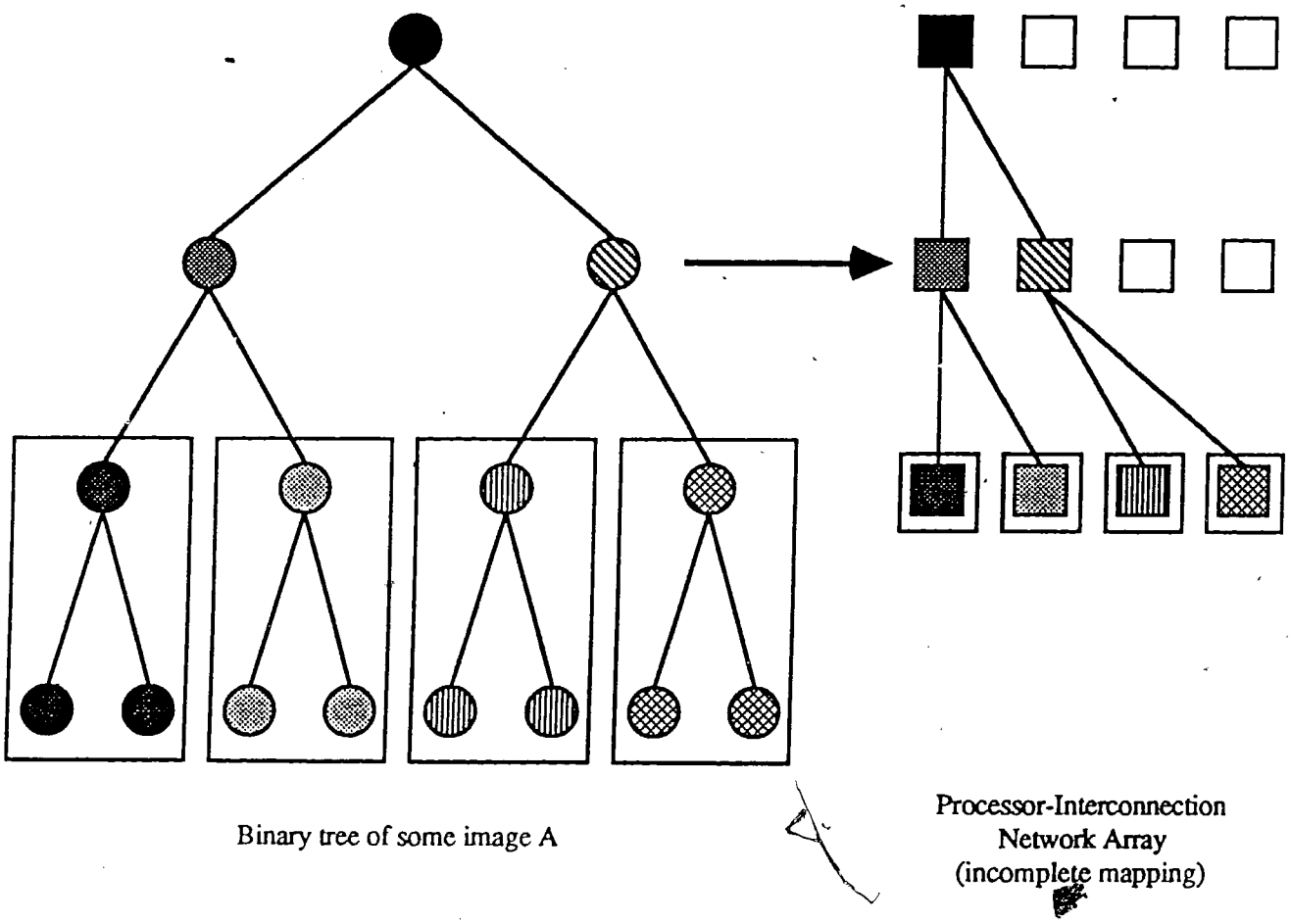


Figure 4.7: Incomplete mapping of a 2^1-tree onto a **P-INA**

wasted power, as it is only the last row which requires these enhanced processors. Array maintenance is simplified since only one type of element is needed.

Alternatively, the economic solution is to replace only the last row of elements with the more powerful microprocessors. These processors receive programs from the **MC** just as the other processors in the array do, albeit the complexity of the routines is greater.

4.3.2 Folding of the 2^N-ary tree

An alternative to the incomplete tree mapping involves the folding of subtrees with root nodes at level $R-1$ onto the array in a bottom-up fashion. The nodes at level R of the tree would be mapped onto the appropriate elements in row $R-2$ of the array. The nodes at level $R-2$ map to row $R-3$. This continues until the entire tree is folded onto the array. This will necessitate multiple foldings if $L \geq 2R$. Figure 4.8 shows the case of an $L=4$ binary tree mapped onto a $R=3$ **P-INA**. To simplify the figure, the **ICN**s are not shown.

With this approach, the elements in the array are all of the same complexity, so there is no extra cost incurred with using different processor types. However, the folding of subtrees back onto the array brings about a problem with processor contention. This is not present in the mapping of the first R levels of the tree. In the initial mapping, each element that represents a node in a level of the tree is part of a single subtree. However, upon folding, an element may represent a node in two or more different subtrees. The processing element has to store the information for these different subtrees, and process them sequentially.

4.4 Execution of operations on the P-INA

There are two components involved in the execution of operations on an array. The first concerns that of the elements making up each processor row. The second

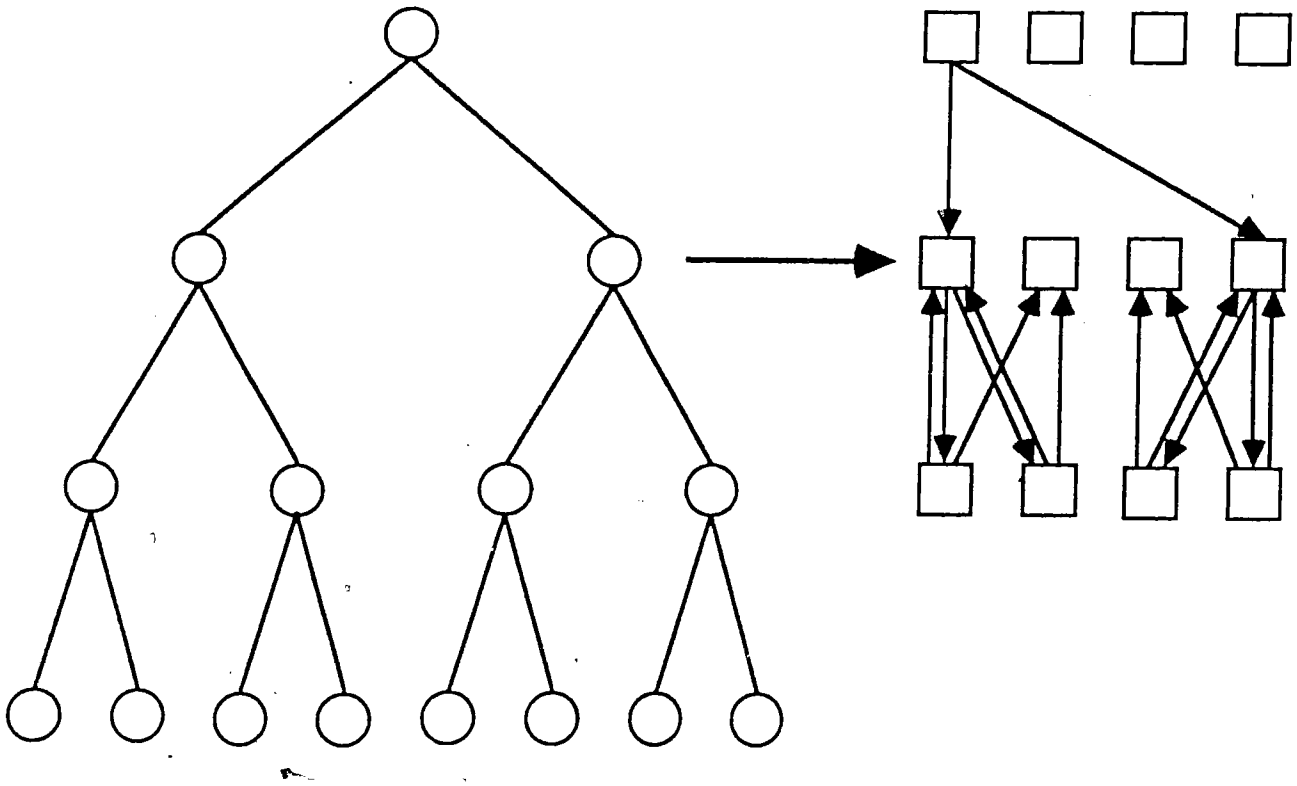


Figure 4.8: Folding of a binary tree onto a **P-INA**

involves the **ICN** between adjacent processor rows.

4.4.1 Row processors and execution of operations

As shown in Chapter 3, it is necessary to traverse the tree in both directions in evaluating a binary operation. With the **P-INA**, this requires that the appropriate switch settings be retrieved and set for the **ICN**s. Once this has been completed, the **PC**s can initiate the appropriate actions for each row. The **P** in row R_0, which represents the root node in the trees, performs its sequence of instructions, and sends any of the necessary values to its 2^N children **P**s. The critical point here is that all **P**s in a row must complete execution before the next row of processors are activated. This is reasonable as the timing restraint given in Chapter 3.1 still applies. Synchronization between rows is accomplished by having the **PC**s communicate with each other. For example, once $\mathbf{PC}_0$ has received all of the required signals from the **P**s under its control, it sends a signal down to $\mathbf{PC}_1$, which in turn will activate the processing elements under its control.

However, in traversing back up the tree/array, we have a situation where some **P**s have to sequentially read in more than one value. Therefore, each row which is affected by this type of multi-valued input has to have a longer execution time before control is passed to its parent row.

This last point must also be considered in the case of the array of sufficient size and the incomplete mapping scenario. However, the necessary synchronization is not as complex as this current situation with the folding tree. As indicated earlier, the folding tree scheme can become ungainly to control with multiple subtrees mapped onto the same subset of processors. The time at each row obviously increases as the number of mapped subtrees increase. In addition to the greater number of values which must be read in by a processing element, every new folding requires that a complete set of switch settings be modified to accommodate this folding. The **ICN** delay can be mininized if the new switch

settings for the folding stage between rows R_i, and R_{i+1} can be set immediately after the current pass through the **ICN** has been completed.

Chapter 4.5 presents an analysis for all three cases, and the delays associated with folding will become obvious.

4.4.2 ICNs and execution of operations

The situation involving **ICN**s has direct parallels to that of the processor rows. The **NC** which controls each **ICN** provides the necessary synchronization to allow for the efficient passage of data. The standard switch settings which are stored in the memory accessible by the **MC** are sent to the **NC**, which then relays this information to the **SE**s.

The rows of processors in the array are mapped in such a way so as to represent a tree. The stages within an **ICN** also map out a tree, but this is a modified binary tree. In the first pass through the array, there is no difficulty with switch elements broadcasting values. However, the problem associated with multiple values entering a processing element is also present with the **SE**s. The solution involves switch synchronization with the **NC**. The data passing from some row R_{r+1} to its parent row R_r first goes through S_{T-1}, where the last broadcast step takes place in the initial pass. Two cycles are necessary to pass these two values through this stage. As the data approaches S_{T-N}, additional cycles are necessary to process this data through the stages. This is comparable to a pipeline effect, in that a value is sent through a **SE** at the same time as a similar value is sent through an **SE** at a lower stage. Since the number of stages per **ICN** is the same throughout the array, the time for data to pass through any **ICN** is consistent.

4.5 · Time frame analysis of the architecture

This analysis of the **P-INA** architecture takes a similar approach to that of Chapter 3. The task is to determine the time complexity of binary operations on trees mapped on the array. The following conventions are used:

- I_A and I_B refer to the two images that are involved in some binary operation of the form I_A **OP** I_B;
- w is the pixel width of the image (therefore, there are w^2 pixels in the image);
- there are 2^N children per parent node in the tree;
- $L = \log_{2^N} w^2 + 1$ = number of levels in tree (labeled $L_0, L_1, \cdots, L_{L-1}$);
- R refers to the number of rows in the **P-INA**;
- E is the number of processors **P** per row of the **P-INA**, and $R = \log_{2^N} E + 1$;
- the time necessary to traverse a 2^N-ary tree with L levels mapped on an E-processor/row R-row **P-INA**is $S_{N,E,R,L}$.

The analysis also assumes that a parallel time step unit consists of the processing time for a row in the array and the **ICN** setup time between rows of the array. This setup time can be considered constant.

There are two situations which must be considered here. The first involves the case where $L \leq R$ and $E \geq w^2$. The second deals with $L > R$.

4.5.1 Scenario involving P-INA of sufficient size - $L \leq R$

The number of parallel time steps necessary to complete a binary operation on a pair of trees is given by the following components:

$$S_{N,E,R,L} = S_{down} + S_{up} \tag{4.1}$$

$$S_{down} = L \tag{4.2}$$

$$S_{up} = (L-1)(2^N+1) \tag{4.3}$$

$$S_{N,E,R,L} = 2L + 2^N L - 2^N - 1 \tag{4.4}$$

With sufficient rows and processors in the array to map the given 2^N-ary tree, there is no contention for any processor P. The L levels in the tree require L time units for the downward traversal (Equation 4.2). In going back up the tree, it is necessary for multiple values to be passed back to the parent processors. However, only one value can be read by a processor from the network at a time. This accounts for the $2^N + 1$ term in Equation 4.3. There are 2^N values going to the parent node of a subtree, in addition to the processing time needed to complete the operation in the parent processor itself — in a fashion similar to pipeline processing. The overall complexity of the worst case situation is then $O(2^N L)$.

4.5.2 Scenario involving P-INA of insufficient size - $L > R$

When the **P-INA** does not have the capacity to map a 2^N-ary tree in one pass of the array, we can use the two approaches specified in Chapters 4.3.1 and 4.3.2.

Analysis of incomplete tree mapping

The time step analysis for an incomplete mapping can be broken down into three components:

- S_{down}, the time needed to process the first $R-1$ levels of the tree;
- $S_{incomplete}$, the time needed to process the last $L-R+1$ levels of the tree and;
- S_{up}, the time needed to traverse back up the tree from level $R-2$.

The number of time steps to complete a binary operation is then:

$$S_{N,E,R,L} = S_{down} + S_{incomplete} + S_{up} \quad (4.5)$$

$$= (R-1) + S_{incomplete} + (R-2)(2^N + 1) \quad (4.6)$$

$$S_{N,E,R,L} = (R-2)(2^N + 2) + S_{incomplete} + 1 \quad (4.7)$$

One problem with Equation 4.7 involves the fact that $S_{incomplete}$ is essentially the contribution of a software-based processing step, and is a function of N, L, and R. There is an obvious disparity between the execution times of an operation implemented in software versus its hardware equivalent. This requires that the contribution of $S_{incomplete}$ be reduced to minimize its effect on the **P-INA**. From Equation 4.7, if $S_{N,E,R,L}$ is to be kept constant, decreasing the contribution of the software-implemented incomplete mapping requires that R or N be increased. Intuitively, this compensation for reducing the software-dependence on the mapping is what is to be expected. Of the two alternatives, adding additional rows of processors is the easiest (modifying N will require entirely new mappings and switch settings). As $R \rightarrow L$, $S_{incomplete} \rightarrow 0$, and Equation 4.7 reduces to Equation 4.4 if the array's last row is accounted for. Its contribution to the expression is $2^N + 2$ time steps.

Analysis of the tree folding alternative

The number of time steps necessary to complete a binary operation using this method are:

$$S_{N,E,R,L} = S_{down} + S_{up} \quad (4.8)$$

$$S_{N,E,R,L} = S_{dpass} + S_{fold} + S_{unfold} + S_{upass} \quad (4.9)$$

S_{dpass} refers to the number of time units required for the first traversal of all R rows of the array. S_{upass} is the time needed to traverse row $R-1$ to row 0 of the array. To fold levels $R, \cdots, L-1$ onto the array calls for S_{fold}. To unfold these $L-R$ levels from the array requires S_{unfold} time.

With no contention for processors in the first pass of the array,

$$S_{dpass} = R \tag{4.10}$$

$$S_{upass} = (R-1)(2^N + 1) \tag{4.11}$$

The reasoning here follows the same as that given in Chapter 4.5.1

An indication of the time required for the folding of a tree onto an array is the number of effective subtrees which remain to be mapped after the first $R-1$ levels of the tree have themselves been mapped. This value is given by

$$\sum_{i=1}^{L-R} 2^{Ni}$$

However, this term does not consider the processing overlap which may occur when two subtrees are not mapped onto the same set of processors. This reduction in the number of subtrees is represented by

$$\sum_{i=1}^{L-R-1} (2^{Ni} - 1)$$

Taking these two terms, the effective folding time is

$$S_{fold} = \sum_{i=1}^{L-R} 2^{Ni} - \sum_{i=1}^{L-R-1} (2^{Ni} - 1)$$

which can be reduced to

$$S_{fold} = 2^{N(L-R)} + L - R - 1 \tag{4.12}$$

Again, by following our intuition, we would expect that the amount of folding necessary, and thus the time for folding, decreases as the number of rows in the array increase. This is exactly what Equation 4.12 predicts. As $R \rightarrow L$, $2^{N(L-R)} \rightarrow 1$, and $S_{fold} \rightarrow 0$.

Combining Equations 4.10 and 4.12, we arrive at:

$$S_{down} = 2^{N(L-R)} + L - 1 \tag{4.13}$$

The final term is S_{unfold}. There are three contributing factors in the S_{unfold} expression. The first provides an approximation of the total amount of time needed to process the last level of subtrees folded onto the array, and is given by

$$\frac{2^{N(L-2)}(2^N+1)}{E}$$

As always, 2^N+1 is the number of time units required to process the children of a parent node in going back up the tree. The level at which these critical subtrees exist is $L-2$. The division by E reflects the fact at level $R-1$ in the tree, there are E nodes, and associated subtrees. Each of these subtrees can be processed independently of the remaining $E-1$ subtrees.

The second component of the S_{unfold} term,

$$\sum_{R}^{L-3}(2^N+1),$$

determines the contribution of the subtrees between the second to last level of the tree, and the R^{th} level of the tree.

Finally, if $L \geq R+2$, then the third component is 2^N. This is the contribution to the final value due to the subtrees rooted at level $R-1$. S_{unfold} can then be expressed as

$$S_{unfold} = \frac{2^{N(L-2)}(2^N+1)}{E} + \sum_{R}^{L-3}(2^N+1) + \begin{cases} 2^N & \text{if } L \geq R+2 \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

Using Equations 4.11 and 4.14, expressions for S_{up} can be derived. In the case of $L \geq R+2$,

$$S_{up} = (2^N+1)\left((L-2) + \frac{2^{N(L-2)}}{E}\right) - 1 \tag{4.15}$$

or for $L < R+2$,

$$S_{up} = (2^N+1)\left((L-3) + \frac{2^{N(L-2)}}{E}\right) \tag{4.16}$$

Finally, by combining these expressions with Equation 4.13, two equations for $S_{N,E,R,L}$ can be generated. For the case of $L \geq R+2$, we have

$$S_{N,E,R,L} = 2^{N(L-R)} + (2^N+1)\left(L-2+\frac{2^{N(L-2)}}{E}\right) + L - 2 \tag{4.17}$$

Otherwise,

$$S_{N,E,R,L} = 2^{N(L-R)} + (2^N + 1)\left(L - 3 + \frac{2^{N(L-2)}}{E}\right) + L - 1 \qquad (4.18)$$

Unfortunately, the nature of these equations makes simplification difficult.

4.6 Reducing the Number of Inactive Nodes

If an application requires extremely large trees (as would be the case in a geographical information system), the tremendous overhead in interconnection settings between adjacent processor rows makes the folding scheme very inefficient. The incomplete mapping does not present as complicated a situation as the folding. However, there will be a large number of inactive row processors if there are many processors per row, and many rows in the array. At the root level of the array which has E processors and mapped for a 2^N-ary tree, only $\frac{1}{N}$ are actively used. The second level utilizes $\frac{2^N}{E}$ of these processors. In addition, those row levels mapping to the top portion of a tree will not have as much intra-row processor communication to contend with, as those levels lower in the array. A more effective use of the available processing power in the array can be made if the tree nodes are mapped more evenly. This consistent distribution is possible considering that there exist communication links between adjacent rows, and amongst processors of each row. The algorithms of the representation are unchanged but it will be necessary to modify the control algorithms maintaining the system. They will be more complex than those of the incomplete mapping method as additional synchronization steps are needed. Even with these modifications, the benefit of utilizing more of the available processing power is obvious..

Chapter 5 Embedding of Restricted 2^N-ary Trees on VLSI Arrays

The emphasis to this point has been on trees involving any valid N. In this chapter, we present an architecture which is based on the 2^1-ary or binary tree. At the abstract level, dealing with $2^{N>1}$-ary trees is more attractive than the simple binary tree. However, significant advances have already been made in the development of binary tree-based architectures. It has even been suggested that the binary tree is a natural method in which to approach problem solving, and that it can be used as a helpful computational structure [24]. With the types of operations which the 2^N-ary tree scheme provides, the binary tree architecture may offer such significant advantages that it is not necessary to consider $2^{N>1}$-ary trees at the moment.

5.1 Dictionary Machines

Extensive effort has gone into the development of a class of architecture called the *dictionary machine*. The research undertaken by Bentley and Kung has provided the basis for further investigation into this design [7]. Some favourable results can be found in [2,13,30]. The machine can be considered as a means of solving search-type problems. For example, given M records, the machine may be queried to locate the record containing the key k. Another type of query may request the number of records with key k. Dictionary machines can be used in those cases

where it is necessary to apply some function $\mathcal{F}$ over every stored record. These types of problems occur in many different applications such as information processing, statistics, and in areas requiring set manipulation. It is these types of applications that can use the 2^N-ary tree representation.

For the most part, these machines are based upon the binary tree organization. The leaves of the tree contain the records, where each consists of two fields, one containing the key k, and the other, the actual data d. Some valid operations are *insertion*, *deletion*, and *search*. Queries are passed to the machine via the root node, and progress down to the leaf level. The partial solution is then passed back up the tree. This two-pass mechanism is similar to that described earlier involving the parallel time-step analysis.

5.2 The Architecture

It is this binary tree topology which allows the dictionary machine and *restricted* 2^N-ary tree representation to make use of developments in the area of embedding trees in VLSI arrays of processing elements[6]. The term *restricted* is used in the context of the special case where $N = 1$. Each node in the tree corresponds to a **PE** in the array. The edges of the tree correspond to the communication links between pairs of **PE**s. The simple and regular interconnections that comprise the tree make it a prime candidate for VLSI implementation, in part because communication is a major consideration in VLSI design. The advantages inherent with the tree abstraction are realized in its implementation. In the worst case example, to access the **PE**s which represent the M leaves in the tree would require $\log_2 M$ time steps. In addition, it is possible to pipeline queries through the tree machine. This leads to a greater utilization of the **PE**s within the array.

A question arises as to why is there a restriction on only using binary trees. Could a similar mapping be applied to the 2^N-ary tree in general? The efficiency

[6]Let **PE** denote one processing element

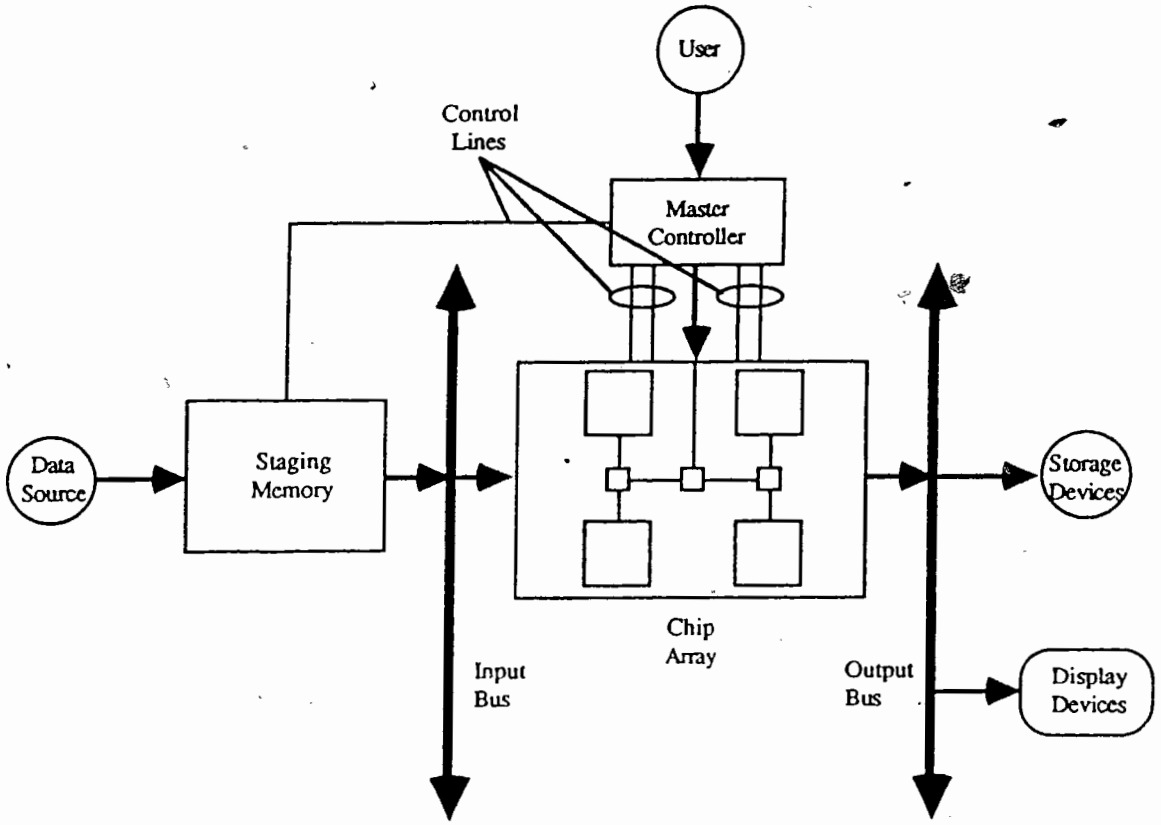


Figure 5.1: The overall architecture using binary tree-mapped chips

of the machine is dependent upon the PE utilization and the extent of the communication between these PEs. The binary tree provides a compact structure with the least fan-out of the 2^N-ary trees. Fan-out refers to the number of children that a node has. The smaller the fan-out, then the more compact and regular the mapping on the array. This also reduces the number of unused PEs. Figure 5.1 presents the entire architecture. The mapping mechanism, and descriptions of the major components follow.

5.2.1 Mapping of the Binary Tree

Two important evaluation criteria that can be used for considering different VLSI interconnection schemes are area efficiency and propagation delay. The former refers to the ratio between the actual number of PEs that are mapped from the

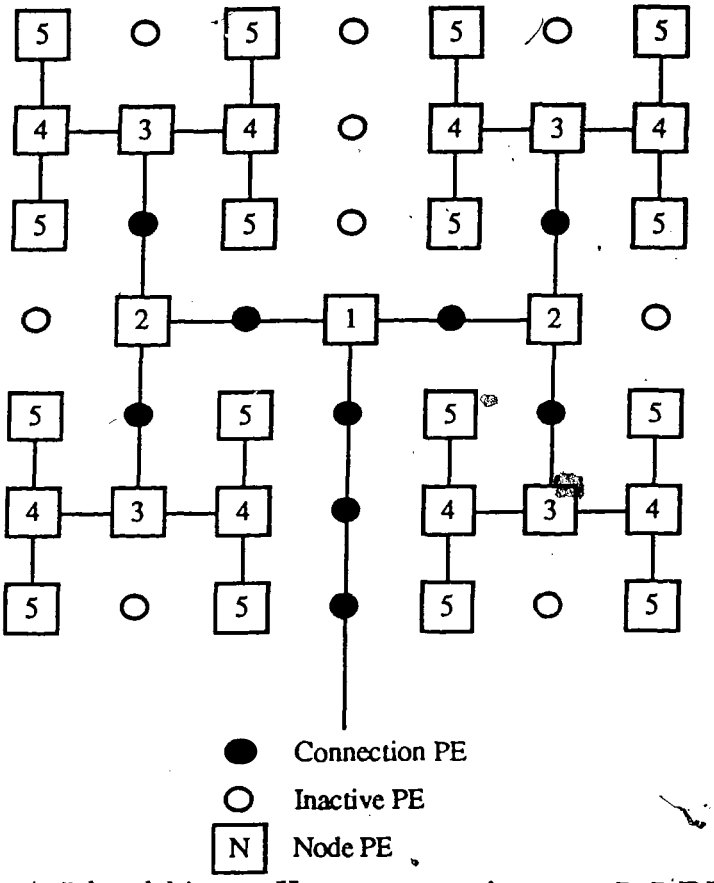


Figure 5.2: A 5-level binary H-tree mapped onto a 7x7 **PE** array

abstraction and the total number of **PE**s on the VLSI chip. With the binary tree, the mapped **PE**s are those which represent nodes of the tree. Propagation delay refers to the distances between any two mapped **PE**s in terms of the number of interconnections between them (including connection **PE**s).

An example of a binary tree mapping includes the H-tree method of Horowitz and Zorat [18]. The connections are rectangular, have unit width, and occupy an area proportional to their length. The rectangular connection restriction prevents corner connections being made. Figure 5.2 shows the mapping of a 5-level binary tree (32 nodes) on a 7x7 **PE** chip array. The basic unit is the 3-level tree. The area efficiency of this mapping is approximately 65%. The maximum propagation delay is 7 units.

A second type of mapping is the hexagonal array proposed by Gordon *et*

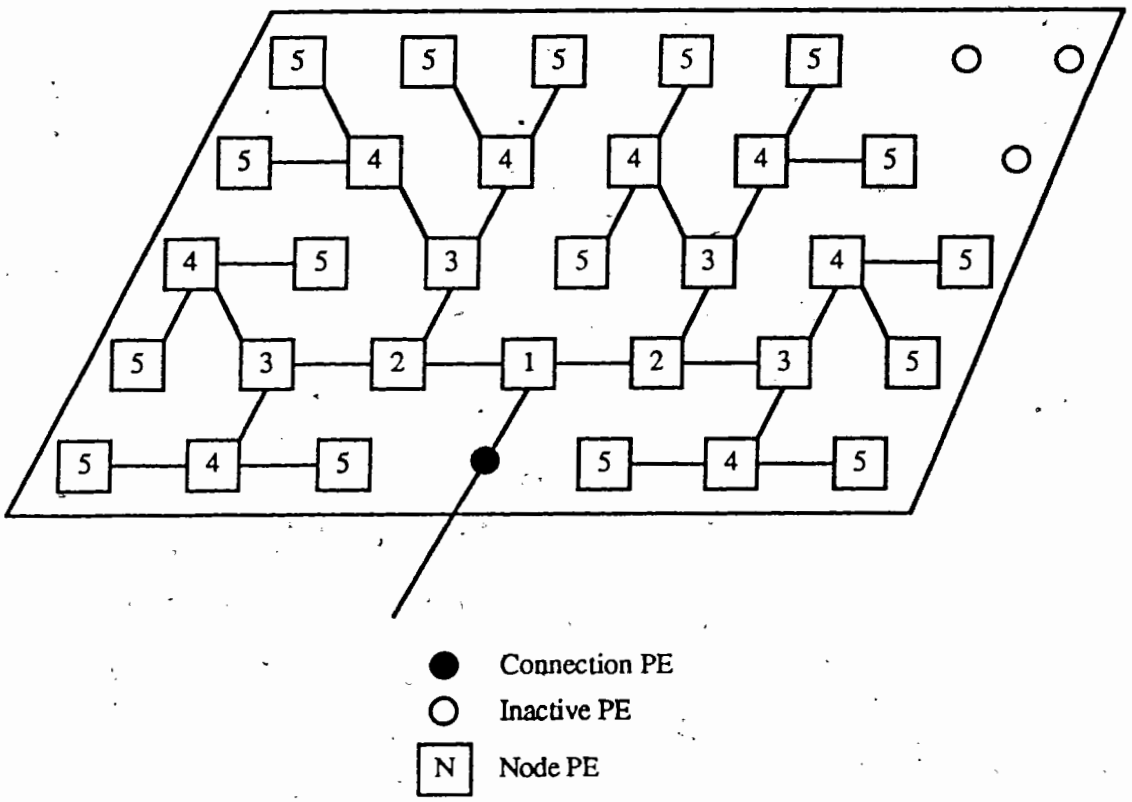


Figure 5.3: A 5-level binary Hexagonal-tree mapped onto a 5x7 PE array

al. [16]. A benefit of this mappings is that two additional connections are now possible (six versus the four of the H-tree). A 5-level binary tree mapped onto a 5x7 grid is shown in Figure 5.3. One observation between the two different mappings is the smaller number of inactive PEs in the hexagonal approach (an area efficiency of approximately 90%, and a propagation delay of five units). It is possible to take the 5- or 6-level mapping as a basic unit to build higher level trees.

Although Gordon's hexagonal mapping is reasonably efficient, a third mapping, which has recently been developed by Youn and Singh, provides for greater PE utilization with some improvement in propagation delay [50]. The number of PEs involved as intermediate connectors has decreased. Figure 5.4 shows both the 4- and 5-level mappings. Note that the latter is formed from two 4-level units. Youn has also summarized the area efficiencies and propagation

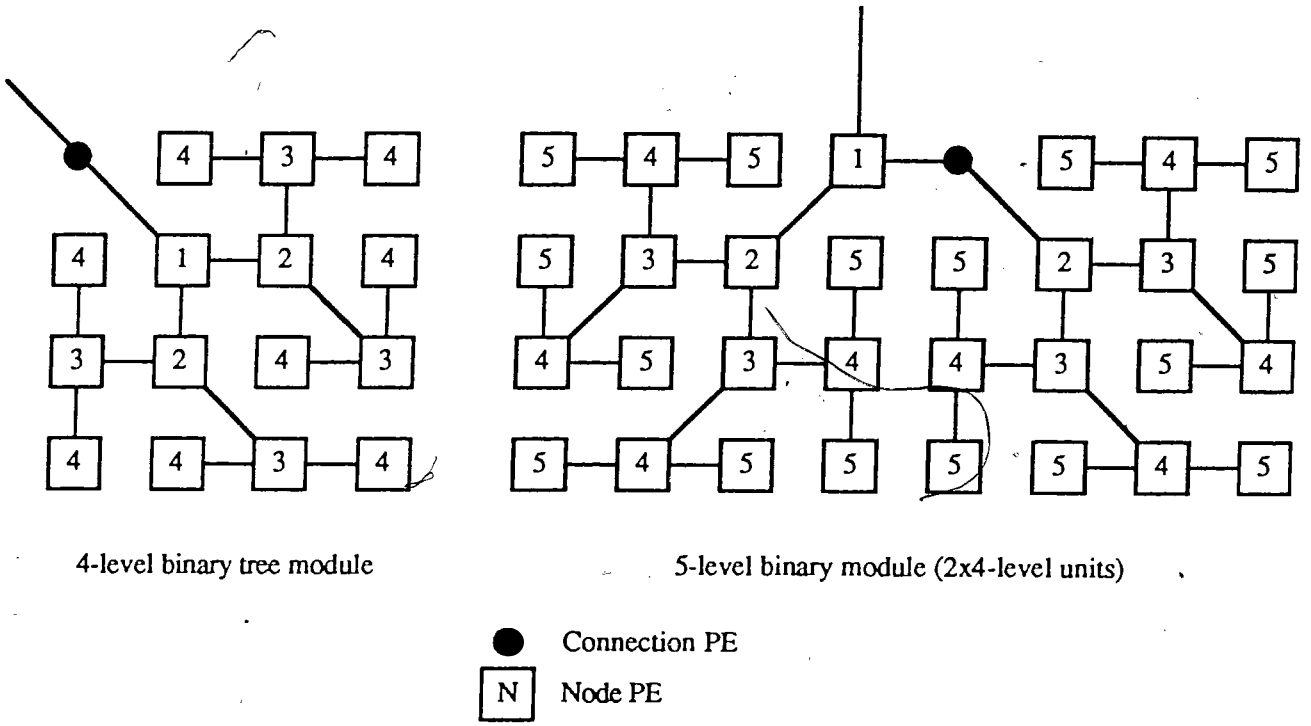


Figure 5.4: Youn's mapping of 4- and 5-level binary trees

delays for all three methods. They present a series of equations which indicate that this new mapping scheme is more area efficient, and provides a considerable reduction in propagation delay than the hexagonal tree. The H-tree is the least efficient method of the three mappings for both criteria.

Some additional benefits of Youn's mapping scheme will be presented when the three 2^N-ary tree architectures are evaluated in terms of fault tolerance.

All three tree mappings make use of a basic unit consisting of $3-6$ levels. It is the replication and connection of these units which allows for the mapping of larger trees.

5.2.2 The Processing Element

Once a mapping scheme has been selected to embed a full binary tree onto a **PE** array, it is necessary to consider other requirements of the representation to make this architecture more efficient. One mechanism which provides an increase in this desired efficiency is bit-serial processing [4]. Instead of processing an array of 100 32-bit numbers sequentially one number at a time, the bit-serial method only requires 32 time units instead of the expected 100, since bit-slicing is used to access the values. Other advantages are that data items can be of any length (it is not necessary to pad them with null values to fill the machine word); considerable time savings can be realized if only part of an operand is required for an operation. For example, given a set of positive and negative integers, one query may be to return the number of negative integers. It is only necessary to access the sign bit instead of the entire number. The design of the **PE** attempts to make use of this bit-serial processing feature wherever possible. Rather than defining a time unit as a function of a complex instruction's execution time, a time unit can be considered in terms of a single cycle.

Figure 5.5 presents the major components of the **PE**. Each **PE** can be identified by a unique address. The address of the root **PE** is the L-bit value $0_0 \cdots 0_{L-1}$, where L is the number of levels in the largest tree that can be mapped onto the array architecture. The nodes/**PE**s are sequentially identified in some consistent manner across a level and towards the $L - 1^{st}$ level. For example, the left and right children at level 1 could be accessed via the addresses 0001 and 0010 for a 4-level tree. The children at level 2 are addressed as 0011, 0100, 0101, and 0110. The **PE**-address identifier is stored in the L-bit shift register **PEid**.

There is also a status register **FLAG** which is just an array of RAM. The elements of this array are used to indicate specific states of the **PE**. For example, one bit can be used to identify the **PE** as being active or just a connection element. In the case of a *connection* **PE**, data is passed directly through the element. There are two general purpose 32-bit parallel-shift registers, **RA** and

$\mathbf{R_B}$ that provide temporary storage. Between these two registers exists an **ALU** which executes all instructions between $\mathbf{R_A}$ and $\mathbf{R_B}$ in parallel by shifting the contents of these registers through it. The result of an operation usually goes into $\mathbf{R_A}$. There is also a link between $\mathbf{R_A}$ the element's $\mathbf{PE_{id}}$.

The actual data for the representation scheme is stored in a parallel shift register which consists of 32 32-bit words. The 32 words may actually be increased, as may the word length. The logic necessary to support the types of operations that are performed on 2^N-ary trees is relatively simple because the operations themselves are simple. This 32-bit width allows for the storage of multiple node values per word. For example, if the number of values that a tree node can have is three (**BLACK**, **WHITE**, and **GREY**), two bits are needed per node. Therefore, sixteen complete nodes can be stored per word (without having overlap onto the next word). As indicated earlier, the bit-serial method provides for efficient selection of specific bits from a word. A mask can be stored in $\mathbf{R_A}$ while $\mathbf{R_B}$ contains the target word. The appropriate function can be applied to $\mathbf{R_A}$ and $\mathbf{R_B}$ via the **ALU** to isolate the desired two bits. Selection of the necessary word from this memory is done through a multiplexor, and requires 5 bits ($2^5 = 32$).

5.2.3 Data/Instruction Buses

Communication between the **PE**s of the VLSI tree is accomplished through the data and instruction buses. These can be considered as the edges connecting the nodes between adjacent levels in the tree. An instruction is passed from the **MC** to the root **PE**, which can forward/broadcast the instruction to its child nodes. In a tree with $L = \log M + 1$ levels, where M is the number of data items, and the log function is of base 2, the leaf **PE**s can receive instructions from the root **PE** in L cycles. We had previously made reference to high level "time units" and "time steps" which consisted of additional execution steps/cycles. With the bit serial

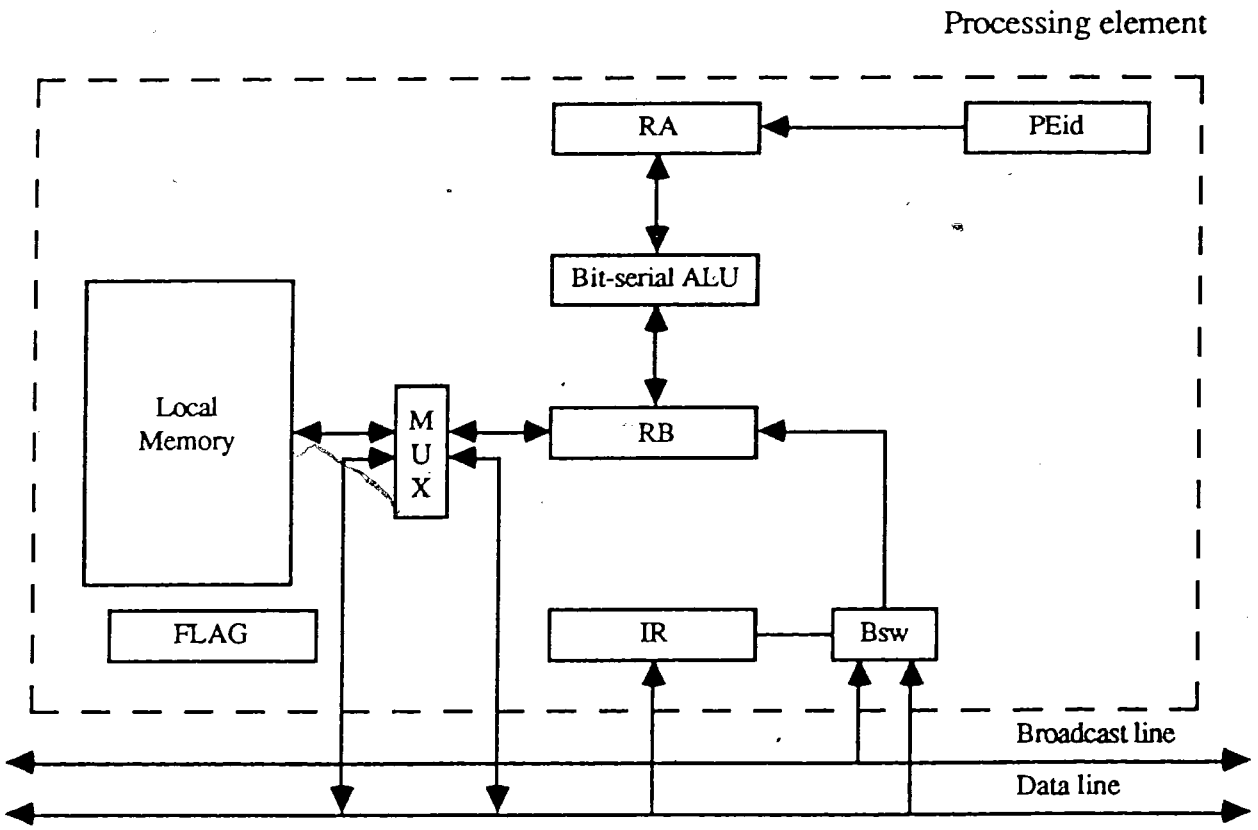


Figure 5.5: A processing element in the VLSI array

processing capabilities of the architecture, it is now possible to use these cycles as a frame of reference. Data can also be passed back and forth between parent and child nodes in time consistent with its bit width.

With these instruction and data transfers, it is also necessary to consider transmission delay between **PE**s. In the analysis of a particular VLSI model, one criterion for model effectiveness is the bit transfer time across the connections between **PE**s [28]. One school of researchers assume a delay of $O(\log D)$, where D is the connection length. Yet another uses an $O(D)$ delay. Due to the predominantly direct links between nodes at successive levels in the tree offered by Youn's tree mapping scheme, it is possible to assume an $O(1)$ delay between successive **PE**s.

5.3 Analysis of Two Operations on the 2^1-ary Tree

5.3.1 Building a Tree

The parallel time step analysis of Chapter 3 presented a worst case scenario where it was necessary to effectively traverse the tree twice. An example of such an operation is the storage of data (the values which are stored in the leaves of a tree), and subsequent building of the tree. The initial pass down through the tree requires $M + \log_2 M - 1$ major time steps for M data items. If pipelining is not used, this time increases to $M \log M$ units.

The loading of an image with M values into the array of chips requires that the **MC** compute the number of levels, $L = \log M + 1$, and the starting leaf **PE** address (the remaining leaf addresses follow sequentially after this first address) for the tree. The **MC** then sends an instruction to each **PE** via the root **PE** requesting that its particular identification key be copied into its $\mathbf{R_A}$. Once the starting address is stored in the $\mathbf{R_B}$ of each **PE** a comparison requiring only $O(1)$ cycle time is made with the contents of $\mathbf{R_A}$. The leaf data is sent out on the data

bus, and bit-shifted into the destination **PE**.

For some X-bit value to be shifted into $\mathbf{R_B}$, the complete value can be stored in X cycles, as each bit requires one cycle. In the storage of **PE** addresses, the maximum tree depth offered by a chip configuration also provides the bit length of the largest **PE** address. If the maximum number of leaf nodes is M, then the maximum address size is given by L bits. L cycles are needed to load this value into a **PE**'s register.

In building a tree of maximum depth from a set of M data values, the storage of the leaf values requires $O(ML)$ minor cycles. Each pair of child **PE**s at level $L-1$ must pass their corresponding values to their parent **PE** on level $L-2$. In effect, a total of $2L$ bits must be sent to $\mathbf{R_A}$ and $\mathbf{R_B}$ of the parent, which requires $O(2L) = O(L)$ minor cycles. The bit-serial application of the appropriate query function generates a value for the parent node (O(1) time). This result is stored in the **PE**'s data memory in $O(L)$ time. Storing this value can be done in conjunction with passing the result to the next highest level so that an additional $O(L)$ cycles are not required.

As shown earlier, $M + \log M - 1$ time steps are needed to store M values in the leaf **PE**s. These leaf **PE**s must pass their values back to their parent nodes, which will then compare the items and store the result. These steps require $2L + 2$ cycles for each of the remaining $L - 1$ levels back to the root **PE**. Storage of the root value needs an additional L cycles. The entire procedure requires $3\log^2 M + (5 + M)\log M + M$ or $O(M \log M)$ cycles. Using a conservative estimate of a 10 megahertz (MHz) clock rate, approximately 1 millisecond would be needed to store 1000 values.

5.3.2 Double Pass Query/Operation

Operations involving two or more objects usually involve double passes through the trees. It is necessary to copy the contents of entity A into $\mathbf{R_A}$, and of B into

$\mathbf{R_B}$ (O($2L$) cycles). The compare and branch operations require 2 cycles. These $2L + 2$ cycles are performed for the L levels of the downward pass of the tree. At level $L - 1$, it is necessary to save the result at each leaf **PE**. The storage of leaf values can be done while the result is passed back up the tree. This presents a savings of L cycles. Returning to the root node requires $2L + 2$ cycles per level for $L - 1$ levels. The final root value needs a separate L cycles for its storage. The total number of cycles is $4L^2 + 3L - 2$, or O($\log^2 M$).

5.4 The Case of Insufficient Chip Levels

The analysis in the previous section assumes that there are sufficient tree levels in provided by the VLSI mapping to accommodate any request presented to it. Unfortunately, there will be situations where this assumption will not be valid. The simplest solution would be to add additional chips to our array, with the appropriate interconnections. A situation will be reached where the packaging of these binary chips becomes the limiting factor to a successful implementation. However, with the phenomenal developments taking place in VLSI chip technology, **PE** densities are increasing at a significant rate, as are the **PE** complexities. With the 2^1-ary representation, this increase in **PE** complexity can be sacrificed for greater chip densities.

Essentially, most of the alternatives which are available to our first two architectures cannot be used here. Some involve network reconfiguration which are difficult to achieve when **PE**-to-**PE** links are hard-wired into the chip.

One feasible alternative follows an approach taken by the reconfigurable processor-interconnection scheme. The **PE**s on the last level of the array can be of greater complexity than those of other levels. Those chips that map to the lower levels of a tree may consist of **PE**s which contain greater amounts of local memory, program stores, and control logic. The function of these **PE**s is to process the lower subtrees of these large trees, rather than single nodes of the tree.

Chapter 6 A Multiprocessor System for the 2^N-ary Tree

This particular architecture for the 2^N-ary tree has properties which are consistent with other multiprocessor systems, which include [12]:

- the sharing of common memory by all processors;
- accessibility to input/output channels, devices, and control units by each processor, and;
- system control through a single operating system.

At a high level, a multiprocessor system can be represented as shown in Figure 6.1.

The flexibility offered by the multiprocessor organization makes it suitable for a large number of applications. Architectures such as the array computer, which is classified as belonging to the **SIMD** class, are most effective in dealing with vector-type computations. These computations can be mapped onto a **MIMD** system by modifying the necessary algorithms. The same cannot be said in the opposite case. A non-vector computation that can be processed by a **MIMD** arrangement may be such that it cannot be reworked into a form consistent with the array computer's requirements.

To distinguish between different multiprocessor designs, the type of processor unit (PrU)-memory unit (MU) interface, the homogeneity of the PrUs, and PrU

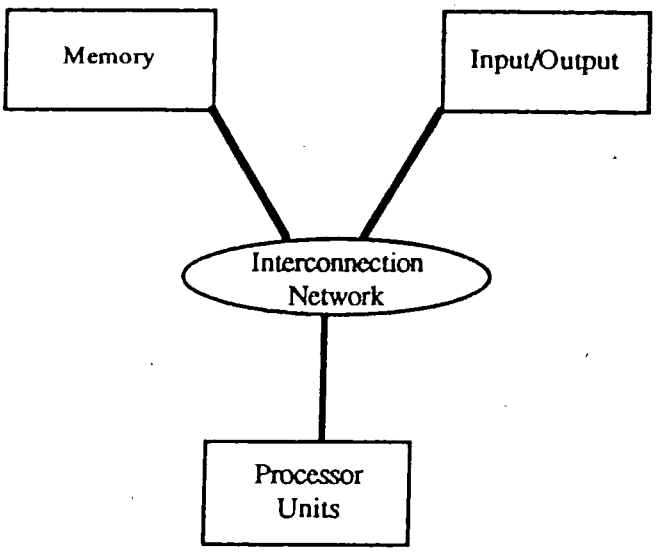


Figure 6.1: A High-level view of multiprocessor architectures

intercommunication are most often considered [3]. In terms of the former, a *tightly coupled* multiprocessor system is one in which all of the PrUs share common memory. The number of PrUs can be fixed, and under the control of a strict supervisory mechanism. Some features of such a system include dynamic load balancing and fault tolerance. An important advantage of utilizing the tight coupling along the critical path between the PrUs and MUs is that it is possible to incrementally increase the processing power of the system. Another advantage is that all of the PrUs are equivalent, so there is no need to distinguish between them in assigning tasks. Since the data is in one location, only one data accessing mechanism is needed.

In a *loosely coupled* system, each PrU has its own local memory. Communication between these PrUs is accomplished via message passing. This action is slow since it is processed at the subroutine level with software. The recipient PrU must also be prepared for the incoming message. Extra overhead is incurred if additional PrUs must act as intermediaries for messages between two distant PrUs.

The system to be described makes use of a compromise between these two approaches.

Both the tightly and loosely coupled systems can be considered as *local* systems in that the PrUs, MUs, and supporting components are defined as one unit. A third type of interconnection scheme can be considered as linking independent computer systems through some form of network [40]. This last situation is beyond the scope of the dissertation.

6.1 Components and Issues of Multiprocessor Systems

We begin by considering the forms of contention which may affect the components of a multiprocessor system. This is followed by a further description of these same components, additional issues associated with such systems, and possible methods to alleviate the contention problem.

6.1.1 Contention in Multiprocessor Systems

There are both hardware and software limitations in the design of multiprocesor systems [29]. In the case of the former, these include the number of processors, memory bandwidth, and the interconnection bandwidth. Software limitations deal with data sharing amongst many PrUs. There are also four types of contention which must be addressed in designing multiprocessor systems [32]:

- processor-to-memory interconnection;
- the interconnection mechanism;
- the memory module, and;
- the memory location.

Contention between the PrUs and MUs

Adding additional PrUs will generally increase the processing capacity of the system up to the point where the effects of contention through the interconnection negate these increases. If the number of MUs is also increased, there will be a greater number of paths between the PrUs and MUs. The interconnection scheme can be designed to provide the means by which many PrUs can read/write the shared memory in parallel.

Contention through the Interconnection

While there may be sufficient paths between the processors and memory units, there still exists a possibility that two or more PrUs can attempt to access the same path through the interconnection. Two possible actions can be used to resolve this problem. In the case of a *blocking* approach, only one request is allowed to use the path. The remaining requests are queued at the point of contention, and continue only after the initial request reaches its destination. The path then becomes available for further use. With the *non-blocking* interconnection, a request is sent through, while the other contending requests are aborted, to be sent again by the processors. One critical disadvantage of the former method is that blocked requests actually maintain and block already traversed nodes and their resources in the switch. This is an example of *switch saturation*. The non-blocking switch does not suffer from this.

Memory Module Contention

A non-uniform distribution of references to the MUs results in greater contention problems compared to a uniform distribution. Unfortunately, program and data locality make the former case the predominant one.

If more PrUs are added to the system, this memory contention will also increase. As the bandwidth of a memory module is fixed, adding memory to a

module will not reduce this contention to accommodate this PrU increase. Instead, it is necessary to add additional MUs. The next concern is to consider program locality and how the code segments are to be distributed amongst the MUs to lower the amount of contention.

Memory Location Contention

This type of contention occurs when two or more PrUs attempt to access the same memory location. Examples leading to this contention include the accessing of a semaphore for a critical section, and the index variable of a repetitive language construct such as a parallel **DO** or **FOR**.

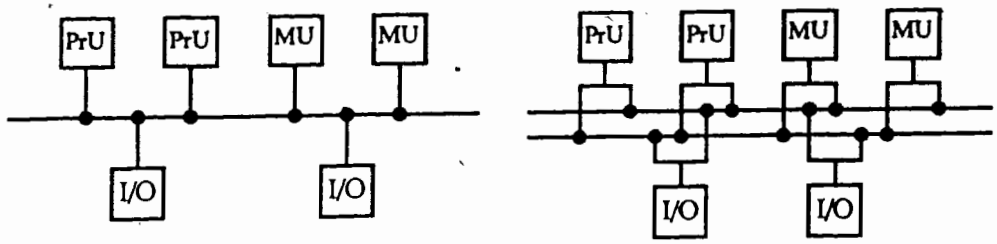
6.1.2 The Processor-to-Memory Switch

The relationship between PrUs and MUs requires a processor-to-memory switch. Figure 6.2 presents three basic topologies which provide such a foundation in multiprocessor systems.

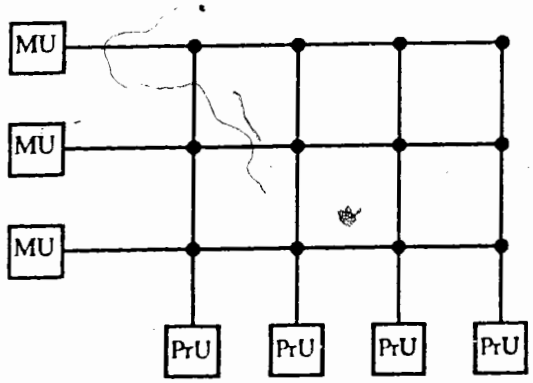
The Time-Shared Multiplex Bus

The simplest connection mechanism between components of a multiprocessor system involves the single time-shared multiplex bus. It is very cost effective and reliable because of the low logic, switching, and control function requirements needed to allow the bus system to operate. Unfortunately, this single bus can also cause the entire system to become inoperational if it malfunctions. The overall system processing power can be considered a function of the bus' capabilities, such as bandwidth and speed. As only a single transaction is allowed on the bus at a time, the system performance level will be very low. A compromise between this low processing power and minimal system cost is necessary.

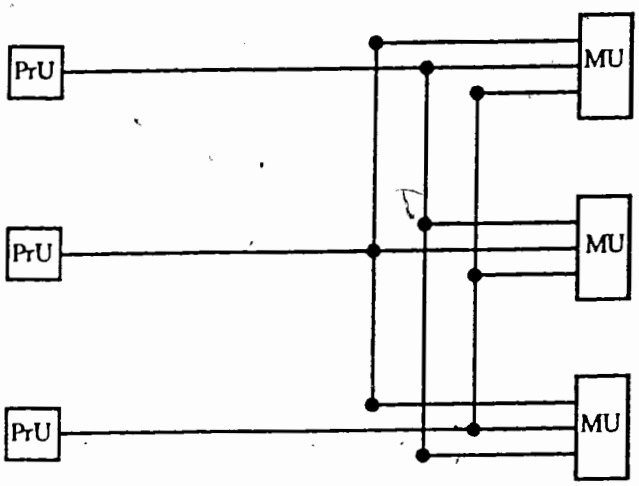
These performance levels can be improved by increasing the number of uni- or multi-directional buses. Another approach is to provide separate bus sets for each



a) Time-shared Multiplex Bus



b) Cross-bar Switch



c) Multiport Memory Bus

Figure 6.2: Basic switch topologies in a multiprocessor system

logical component of the system, and then have additional interconnection mechanisms between these sets. These bus sets can be used for processor groups, memory arrangements, and input/output devices. This increase in system performance is at the expense of increased system cost.

In the multiple bus, time shared scheme shown in Figure 6.2.a, memory connections between the PrUs and MUs involve a considerable amount of redundancy. With P PrUs, M MUs, and B buses, $B(M+P)$ connections are required in the complete configuration. However, alternatives have been developed which reduce the number of connections, while at the same time maintaining similar throughputs [21]. A reduction of 25% in the number of connections can be realized if $B = \frac{M}{2}$ and $P = M$. Reducing just the number of connections in the memory-bus component while using a fully connected PrU-bus arrangement can lead to the minimal number of connections.

The Crossbar Switch

If the number of buses added to the system results in each MU having a separate bus, we essentially have a crossbar switch arrangement. This switch was considered as a possible mechanism for the 2-D processor/interconnection network, but as indicated in our earlier discussion, its expensive nature as the number of PrUs and MUs became larger made it an unattractive solution. Another difficulty with the crossbar involves the access conflicts to the memory units (memory contention). Since the switch supports simultaneous transfers for all MUs, it must be able to resolve multiple requests to a single MU. A priority scheme may be in place which the switches use to arbitrate all memory access requests. This requires more complex switches, which results in increased system cost.

A study has investigated three different systems that utilize the crossbar switch [41]. These consist of:

- P PrUs and M fully interconnected global MUs, where each PrU has access

to its own MU that contains the code that it is to execute, and the PrUs occasionally access other MUs;

- P PrUs and 1 shared MU : each PrU has its own local memory LM, which reduces the number of access to the MU. It is less tightly coupled so that a linear increase of P in performance level is expected. A state is reached for some P where no further increase in performance is possible;
- a combination of the first two systems : P PrUs and M MUs, where each PrU has its own LM. The benefits of both previous systems are realized here (the former provides a dynamic environment, with no performance saturation, while the second offers low memory contention).

The primary drawback of the first system is the exponentially increasing cost per processor ratio. With the second, the single global MU prevents dynamically allocated memory, as most of the programs and data are stored in the private LMs of the PrUs. For a relatively high access-to-MU probability, and a low number of consecutive memory access, the cost per processor ratio makes this system favourable. If these limits are exceeded, the ratio approaches infinity. This is not the approach that high performance applications would use. In the case of the third system, the cost per processor ratio increases linearly with no performance saturation regardless of P. However, proponents of tightly coupled systems may find this arrangement unacceptable.

Expandability of the switch is another concern placed upon the designer. A switch which exceeds the present requirements of the system can have a number of inactive nodes. These nodes can enter the active state whenever additional PrUs and/or MUs are made available to the system.

One of the earliest systems utilizing a crossbar interconnection mechanism was the Carnegie-Mellon multi-miniprocessor C.mmp [40,48]. It consists of sixteen PDP-11 minicomputers, and sixteen memory modules. This is an example of a very large multiprocessor system, and the cost of the crossbar switch is

significantly lower than the minicomputers being used. It was designed as a general purpose multiprocessor. Our system is used for a much simpler, more specific application which does not require all of the supporting components of the C.mmp.

The Multiport Memory Bus Scheme

The third organization is the multiport memory system. Each PrU has its own bus that allows access to all MUs. The switching mechanism is located at each MU. Memory access conflicts are resolved by assigning fixed priorities to each port such that specific PrUs can preferentially access certain MUs. Throughput is intermediate between the single bus and crossbar systems.

The hardware requirement is similar to the crossbar system, and the level of concurrency is the minimum of the number of PrUs and MUs. One disadvantage of such a system is that the number of ports that a MU has also limits the number of PrUs that can access it. The complex control and switching mechanism of each MU makes this arrangement very expensive.

6.1.3 Memory Considerations

Issues such as the usage of local and/or shared memory, and the types of memory mapping schemes must be considered in designing a multiprocessor system for a particular application.

Local memory at each PrU can be used to reduce the dependency on the interconnection mechanism. These local memories contain data which are used most frequently by the PrUs. This will result in greater overall performance as an effectively higher bandwidth is achieved, and the data is more readily available to the PrU.

Two problems exist with the use of local memory. If a request for data from PrU_j's local memory is made from PrU_i, the value returned may not be correct,

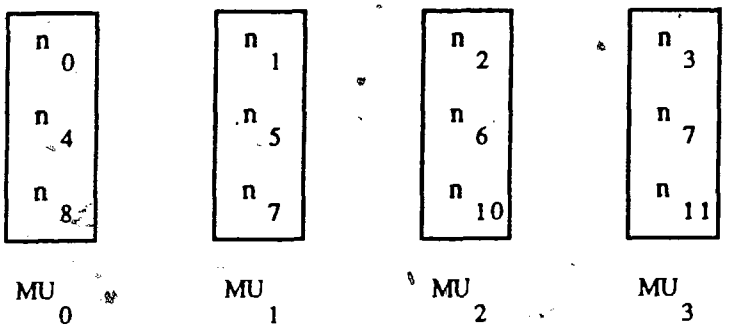


Figure 6.3: Storage of a 2-D array of numbers

particularly if the data is volatile. The second problem deals with data which is heavily used by more than one PrU. The affected local memories will contain copies of this data which may be manipulated by their respective PrUs, leading to data corruption.

The use of shared memory can reduce the levels of this corruption by providing only a single set of data to all of the PrUs. The PrU-MU interconnection scheme will determine one level of contention in using shared memory. An effective *memory mapping mechanism* can reduce the amount of memory contention. This mapping is similar to the mapping/paging practices present in operating systems using virtual memory. These maps translate a PrU-generated logical address into a physical address that corresponds to some location in a particular MU. If the application is considered in defining a mapping scheme, exploiting properties of the application's data structures can further reduce the levels of memory contention.

For example, one application may require the storage of a 2-dimensional array of N numbers. The simplest arrangement of these values distributed over M MUs would be as shown in Figure 6.3.

Only one memory cycle is required to access any row or diagonal of values over

the MUs. However, accessing the numbers in a single MU leads to contention, and $\lceil \frac{N}{M} \rceil$ memory accesses will be needed to process every value in the MU. It is obvious that this type of distribution of values is unacceptable for most operations. If the array values can be uniformly distributed throughout the MUs, the number of accesses to any one MU will decrease, thus reducing the extent of contention. This *memory interleaving* can be of the following types:

coarse consecutive blocks of memory, addressed sequentially, are found in each MU. The high order bits of an address indicate the module;

fine consecutive memory addresses are located in consecutive modules. The low order bits of an address are used here to specify the module;

mixed both methods can be combined to provide an intermediate form of interleaving. Hardware may control the coarse interleaving, while the fine interleaving could be accommodated through software.

Information such as the type of PrU-MU interconnection scheme, and the application will provide some assistance in determining which form of interleaving to use. If the crossbar switch is being used, coarse interleaving will provide the lowest level of contention. For the most part, independent processors will access individual MUs. If fine interleaving is used, these seemingly independent PrUs are required to cycle through most of the MUs to get the requested data. If a common bus structure is the interconnection, fine interleaving will provide a lower level of memory contention than the coarse alternative.

The process of interleaving can be dealt with at the hardware or software level. In the case of the latter, facilities may be built into compilers so that the memory distribution is made transparent to the user. Conversely, functions can be defined which allow the applications programmer to explicitly specify the arrangement of the data structures in memory.

6.1.4 Software Considerations

Some references to software techniques were alluded to in the discussion of multiprocessor system components. Three additional issues which must be considered in such a system are control, synchronization, and scheduling of the processors. There are three different control organizations [12]:

- master-slave;
- separate and;
- symmetric.

The easiest to implement is the former, although its major disadvantage is that once the master processor fails, the entire system also fails. A major advantage of such a system is that specialized hardware is much easier to add to the system, thereby reducing the executive's overhead. An example of this specialized hardware is associative memory. With the second, each PrU has its own copy of the executive. The fault tolerant benefits of this organization exceed those of the former. With the symmetric approach, each PrU has access to the master executive, and has the ability to schedule itself. The fault tolerance of this organization is far superior than the first two, as it its reliability. This scheduling feature permits functioning PrUs to compensate for any failed processors.

Synchronization between processes/tasks is another major issue which must be addressed in multiprocessor systems. The accessing of shared variables by the PrUs must also be considered. The use of semaphores, different priority levels, and guaranteed processor execution via fair scheduling practices are all techniques which can be used to provide the necessary synchronization.

6.2 An Architecture Applied to the 2^N-ary Tree

The above has been considered in designing a multiprocessor-based architecture for the 2^N-ary tree application. The design, which is shown in Figure 6.4, depends on the properties and characteristics of the representation. The following subsections present each system component, and justifies its particular configuration.

6.2.1 The Shared Memory Module Units

The system's shared memory component consists of a bank of memory modules that are used to store the data and tree representations for any user applications. The type of memory interleaving that is used to store some instance of the representation is a critical consideration. The preferred scheme calls for the use of coarse memory interleaving for a number of reasons. The concurrent processing of each subtree of a tree is dependent upon the disjoint property of each subtree at a given tree level. Consider the case of a simple four MU system where we are evaluating some operation involving two quadtrees. This arrangement is shown in Figure 6.5. A processor can access the equivalent node values in both trees with two read-instructions from the same MU. By storing equivalent subtrees for different instances in the same MU, there is no need to follow a second interconnection path to another module. In one case, the path to a particular module is held for the two reads, and one write-back if there is a need for a resultant third subtree to be stored. As for possible conflicts involving the nodes at different tree levels, one requirement which was specified in Chapter 3 is that children of a node cannot be processed until the parent has been accessed. This prevents memory contention by processors attempting to access a parent and child node of the same subtree, both of which are stored in the same MU. Even with the possibility of traversing back up the tree, the accessing of nodes at some level $i+1$ must be completed prior to the parent at level i.

With our example, contention will arise when processing the nodes at the third

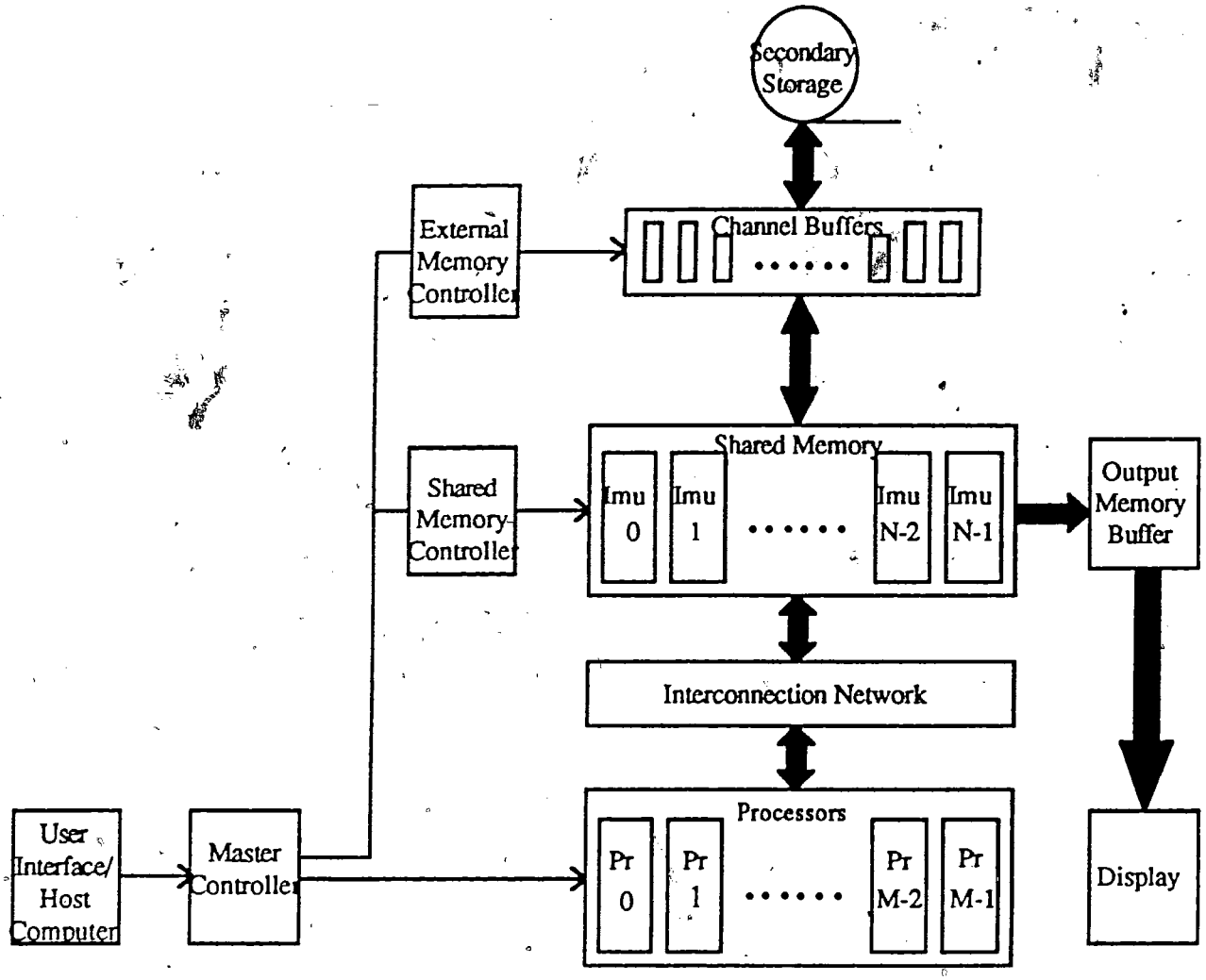


Figure 6.4: A multiprocessor architecture for the 2^N-ary tree

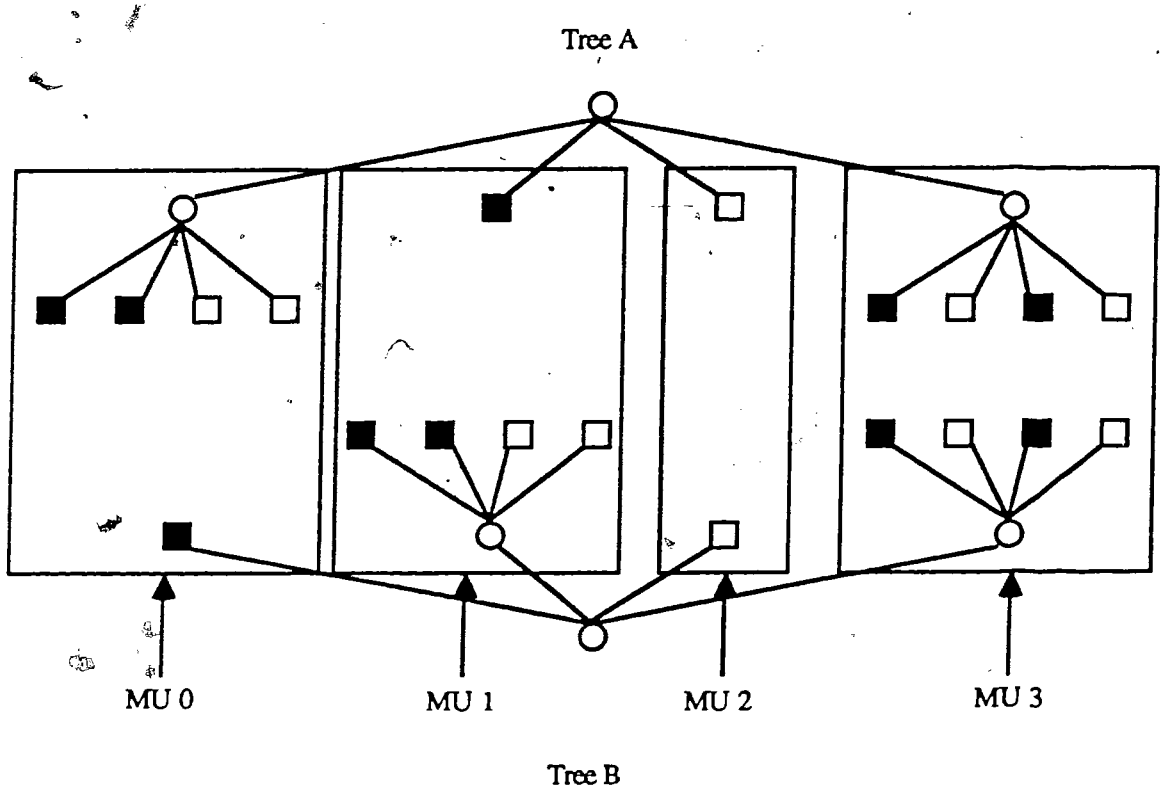


Figure 6.5: Memory interleaving for the tree representation

and subsequent levels of the tree. The primary cause of this contention is the small number of MUs (4) being used. By adding more MUs, memory location contention can be reduced. If twenty MUs are used in the system, memory contention will not be a concern in the processing of the first three levels of the quadtrees. Of course, the extreme case would provide one MU per node of a tree. Even with inexpensive memory, there are some factors which make the one MU/node approach unfeasible. The principal of which is the increased cost incurred by the interconnection scheme, that must now become much more complex. Ineffective use of the storage is also a problem in that few nodes are stored per MU.

A compromise would involve a reduced number of MUs, and an appropriate memory mapping scheme which functions on the premise that storing non-consecutive tree levels in an MU will not lead to an increase in memory contention. This is a consequence of the parent–children node processing restriction. The premise assumes that if processing is taking place at some level i in the tree, there is no concurrent processing being done on level $i \pm \epsilon$ of the same subtree, where ϵ is some value sufficient to make this a valid assumption. Obviously, values of 0 and 1 are inappropriate for ϵ. This approach is similar to the folding of trees presented in Chapter 4.

6.2.2 The Interconnection Scheme

The interconnection arrangement in this architecture is a multistage network which uses the binary tree topology. This is convenient in that the order of the application is some power of 2. Consistent with the observations concerning the mapping of data in the previous section, there exist paths through the M-level network to 2^M MUs, where $M \geq 1$. Each of these MUs will store the equivalent subtrees for a number of tree instances. A simple example of such a network utilizes a 3-level network and is given in Figure 6.6. In the example, the two processors PrU_0 and PrU_1 can access eight MUs. To provide the necessary routing paths for these PrUs and MUs, seven switches are needed in the network tree.

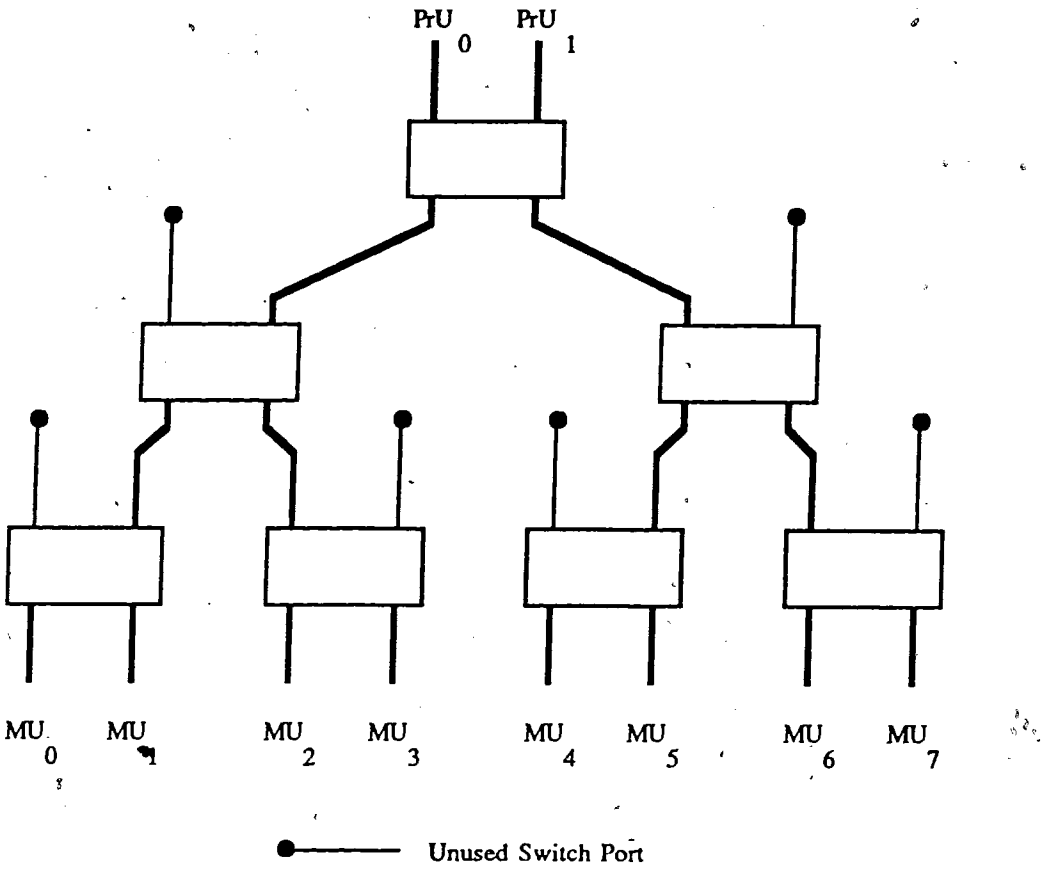


Figure 6.6: A 3-level interconnection network

The figure also shows that there are six unused switch ports. The addition of extra switches will make use of these ports. All ports will be accounted for if eight PrUs and twelve switches are used.

The nodes of the network are actually simple 2x2 crossbar switches. It was stated earlier that the crossbar switch is very expensive to implement. However, with the binary tree topology that is being used here, the node count is reduced from $O(n^2)$, in the case of a regular nxn switch, to $O\left(\frac{n}{2}\log_2 n\right)$, where n is the number of PrUs/MUs in the system, $\log_2 n$ the number of node levels, and there are $\frac{n}{2}$ nodes at the bottom of the network tree. The fan out of each node is the

base of the log function. With the binary tree, the base is 2. In general, if the fan out of each node switch is b, the node count is $O\left(\frac{n}{2}\log_b n\right)$.

There will be network contention through the network if two or more PrUs attempt to access data in the MUs through the same path. The frequency of such conflicts can be reduced if the number of candidate paths are increased. One method of achieving this is to increase the fan out b of each switch, which also reduces the number of nodes in the network. The effect for the special case of $b = 2^M$ is an environment similar to a complete crossbar switch with its associated disadvantages. As has been stated a number of times throughout this dissertation, a compromise between extremes is the most effective solution.

6.2.3 The Processor Units

Given the type of processing power, and local memory requirements needed by the functions of the representation, certain stock microprocessors can be used. Alternatively, an entirely new processor could be developed for this application. Given the nature of the operations, and cost efficiency considerations, existing designs are more than adequate.

One such class of stock processor is the MC68Q00 family, whose favourable characteristics include fast instruction times, sufficient data-path width, reasonable memory size, and a rich instruction set. The memory size is critical since each PrU has its own local memory, which reduces the number of accesses to shared memory. Rather than have all of the representation's programs in shared memory for the PrUs to access, each has its own set of routines. At first, this may seem to be redundant. However, the advantages of using local memory far outweigh the disadvantages of the redundancy. With a large number of PrUs, accessing the shared memory for each each instruction block will lead to a significant contention problem. These requests for instructions will also have to contend with requests for actual representation data. The simple algorithms for

the operations make it practical to store these instructions in local memory. The problem of contention due to instruction fetch is eliminated. If the amount of local memory available to each PrU is insufficient to contain all of the programs, only those functions which are required by the current user request may be stored in this memory. The idea here is to have the entire operation set in shared memory, and broadcast to the PrUs, for local storage, the programs needed for the user query. This approach is again a compromise between two extremes.

Each of the PrUs are identical and can function as independent units. The fault tolerant benefits of such an arrangement are obvious. If a PrU becomes inoperable, there will be a slight deterioration in overall system performance. Scheduling of tasks remains as before, the only difference being that there is one fewer PrU to process these tasks. The system will not come to a halt as would be the case if the PrUs were dependent upon each other.

6.2.4 The Master Controller

There is one processor which serves as the master controller MC for the entire architecture. The MC receives queries/instructions from the User Interface and passes these requests to any available PrU. Until this initial request is fulfilled, the MC is involved in the scheduling of any task requests that are sent to it by the PrUs. This scheduling responsibility requires that the MC has sufficient local memory to maintain the necessary task queues and stacks. The task specifications contain no PrU dependencies such as PrU identification, and can then be considered as autonomous entities. The availability of PrUs for task processing must also be known to the MC. Each PrU can be represented by a single bit in a register stored in the MC — a 0 indicates a PrU currently in use; a 1 for available. Interrupt/acknowledge control lines between the MC and PrUs are used to send special signals to the MC, such as for a defective PrU. In this case, the corresponding PrU register bit is set to 0 until a replacement PrU is added. As with the PrUs, microprocessors such as those available in the MC68000 class

would be appropriate here because of their support of interrupt handling.

Task scheduling is the primary responsibility of the MC. If for some reason the MC becomes inactive through some component malfunction, the entire system becomes inoperable. For this reason it may be advisable to provide an auxiliary MC which can replace the primary MC upon its failure.

6.2.5 Additional Components

There are some additional less critical components that make up this system:

- The User Interface UI provides a user-friendly environment to which user queries/instructions are encoded and directed to the MC.
- To test the integrity of the shared memory modules, and to reduce the need for the MC to monitor the status of these modules, there is a shared memory controller whose primary purpose is to maintain these modules. Another function of the SMC is to coordinate the passage of formatted and unformatted data between the system and the outside environment.
- To facilitate the entry of data into the system, there exist a series of channel buffers which accept data from some secondary storage device such as a tape, and perform some preliminary preformatting on the raw data, if necessary. These buffers direct this data to the shared memory modules. Of course, instances of the representation can be stored on secondary storage devices by going through the buffers. These channel buffers are under the control of the external memory controller EMC. The EMC and SMC both cooperate in the data traffic between the external environment and the system.
- In the application on which this dissertation was initially based, that is, as a representation technique for computer graphics, there is an obvious need for a display device. An instance of the representation is first routed to a memory buffer which can be loaded in parallel from the shared memory

modules. Once the buffer is filled, it is presented to the display device in a single step. This could be thought of as loading the background buffer of a two-buffer device. Switching of buffer planes renders the effect of instant display.

Chapter 7 Simulation of the Multiprocessor Architecture

This chapter presents the results of a simulation of the multiprocessor architecture developed in the previous chapter. The term *simulation* is not used in the truest sense of the word. Typically, the synthesized behaviour of some system or real-time process is generated, and observations are made of this history to develop inferences or hypotheses concerning the characteristics of the system. With respect to the dissertation, the type of software facilities available has made it possible to implement a pseudo-simulation which uses multiple processes to represent the processors of our system, and message passing to represent the network communication mechanisms. In effect, a software-based equivalent to the architecture has been developed which allows various statistics to be obtained on the behaviour of our system.

The chapter is organized as follows. First, some of the implementation details of the simulation are presented. A number of different processor and data configurations are used in the actual experiments. These follow the implementation details. Finally, the results of these experiments are given in the form of graphs and observations.

7.1 Implementation Details

7.1.1 The Processors

The implementation of the simulation makes use of various features made available by the UNIX[7] operating system with its System V enhancements, such as facilities for using shared memory, semaphores, and message passing.

The architecture specifies that the operations of the representation are stored in each PrU's local memory. There is no need to consider accessing some shared memory for the required instructions. Therefore, this implementation only has one set of operations which are readily available to all processors. The code does not have to be considered a critical section — which would require the use of semaphores or other control constructs as a means of mediating mutual exclusion.

Each processor in the architecture is represented by an independent process that has its own identification — to facilitate processor-to-processor (process-to-process in the implementation) communication. Unless otherwise indicated, the simulation's *process* is equivalent to the architecture's *processor*. The control functions and data structures available to the slave processes are very simple. The main control loop for a slave essentially consists of:

```
slave: process
       for TRUE {
           RECEIVE ( task )
           cmd := SELECT ( task )
           EXECUTE ( cmd, task )
           SIGNAL ( FREE, MASTER )
       }
```

The **RECEIVE** operation places the processor in a waiting state, until some instruction/task is sent to it by the master processor MC. Upon receipt of such a

[7] UNIX is a registered trademark of AT&T Bell Laboratories.

task, the appropriate function is invoked by the processor with the included arguments. The **task** contains the function identifier, and its arguments. Once the operation has been completed, the MC is sent a signal indicating that this particular processor is available for further task acceptance.

Although the responsibilities of the MC exceed those of the slave processors, the control structure of the implementation has a similar internal arrangement:

```
master: process
        for TRUE {
            RECEIVE ( user_request )
            slave := SCHEDULE ( available )
            SEND ( user_request, slave )
            for ~ DONE ( ) {
                RECEIVE ( task )
                cmd := SELECT ( task )
                if cmd in {slave_commands}  {
                   slave := SCHEDULE ( available )
                   SEND ( task, slave )
                   }
                else
                   EXECUTE ( cmd, task )
                }
            }
```

The outer control loop permits the MC to **RECEIVE** requests from the user interface. This **RECEIVE** function is similar to that available with the PrU operations in that the processor is in a waiting state until a request is received. Once a request has been accepted by the MC, the **SCHEDULE** function is used to select an available slave processor, after which, this initial **task** is sent to the slave.

The internal loop continues until the requirements of the user request have been fulfilled. This completion state is determined by the function **DONE** which tests various control data structures such as the task queues and stacks. Obviously, if the task queue is not empty, the **user_request** has not been

completed. As an example, a request may be for the generation of a tree from the application of the binary operation **AND** on two trees. The processing within the inner loop continues until the result tree has been derived. This processing involves the receiving of subsequent slave tasks, further scheduling, and the dispatching to the next available processor these tasks.

The data structures required for the MC include: a processor-availability queue containing the identification of each slave processor currently idle; a request queue that is used to recall multistep tree operations — for example, in building a tree, the two steps involved are to load the tree's leaves with data values, and the actual generation of the tree, and; a task stack that holds the tree tasks which are created by the slaves upon execution of the various operations of the representation. This last structure is the most volatile of all the structures since it has to accept all of the child processes which are generated in going down a level of a tree. For example, in the case of a quadtree, if it is necessary to process the children of some node, four tasks have to be created and sent to the MC, to be placed on the task stack. This structure is a stack rather than a queue to preserve the integrity of the tree nesting as processing continues from level to level.

7.1.2 Shared Memory

A major component of the architecture is the bank of shared memory modules that contain the instances of the tree representation. System V's shared memory can be accessed by independent processes using memory identifiers — similar to the idea of process identifiers in UNIX. These memory blocks can be cast as any valid data structure. In this case, the implementation sets these blocks as two-dimensional arrays of the character data type. This type permits byte addressing. By having a series of these blocks, we can simulate the architecture's bank of shared memory modules. It is then possible to access any byte-addressable memory location by providing a module identifier, and a local or relative memory address.

7.1.3 The Interconnection Network

The interconnection network is simulated with a series of simple variables which are identified by means of their row in the network tree, and their relative position within a row. A value of 0 or 1 in a network variable indicates whether the node that it represents is currently being used or not used in a memory access. For each memory address <module#,memory#> generated, the network tree path is also determined. If each of the network variables that is required for a successful access is available, the memory value is either read or written. If not, the task is sent back to the MC and placed back on its task stack.

7.1.4 Contention Considerations

In this implementation, there are two areas of contention which are considered: network path and memory module conflicts. Actual memory location is not considered. This is a valid assumption since at the leaf level, there is never a case of two tasks accessing the same memory location -- a property of the representation. The network and module conflicts carry through at the implementation level since actual sharing of memory (the System V shared memory and the network variables) is also taking place. To accommodate this, semaphores are used. In fact, the network variables are actually semaphores. With the memory modules, each has its own semaphore. The module is only accessible if no other processor has set the module semaphore. Once a processor has finished accessing a module, the module semaphore is reset to indicate its availability, as are the network node semaphores.

7.2 Simulation Configurations

The simulation has been configured to accept anywhere from one to nine processors, and eight shared memory modules. This requires that there be three

levels of nodes in the interconnection network, each level with four nodes. Simulation runs utilizing one to nine processors have been performed. Given the nature of the interconnection network, that of a binary tree topology, eight processors would be the maximum permitted. However, this implementation has been set up for the required eight PrUs, in addition to an extra PrU. The intent is to provide as much data as possible for later interpretation within the constraints of the implementation environment. This environment makes extensive use of a limited number of sockets, pipes, and file descriptors. Nine PrUs is the maximum number of processors which can be simulated with these limitations. The use of one slave processor is a special case in that it most closely resembles the sequential uniprocessor situation where a simple construct such as a **FOR** loop allows each child of a tree node to be processed. Of course, these two cases are truly not equivalent since some processing overhead is accumulated through the message passing between the MC and the slave processor.

The size of the data set, and the type of trees which make up the simulation sample set, requires that only four shared memory modules be used. Even with four modules, it is still necessary to have three levels of nodes to allow for the eight PrUs. All of the experiments use the quadtree as the tree configuration. In terms of data size, statistics are obtained for images which are 8x8, 16x16, 32x32, and 64x64 pixels in size. Each data size involved three different types of image, one being what would be considered an average case where the quadtree is not a complete tree, a best case in which only one level of the tree must be processed, and the worst case where the image is of a checkerboard type. These various images serve as input for a number of the representations operations. Tests are performed using the operations: **LOAD UNFORMATTED**, which takes unformatted data and generates a quadtree; and **AND**, which is applied to two quadtree, creating a third. A binary **OR** operation has also been defined, essentially using the same routines as the **AND**. For this reason, tests have not been run with the **OR**. The application of these different data sets with the two

operations allows comparisons to be made between trials that require both complete and partial tree traversal. It is this which provides the best, average, and worst case scenarios. For each different set of circumstances, ten trials are run.

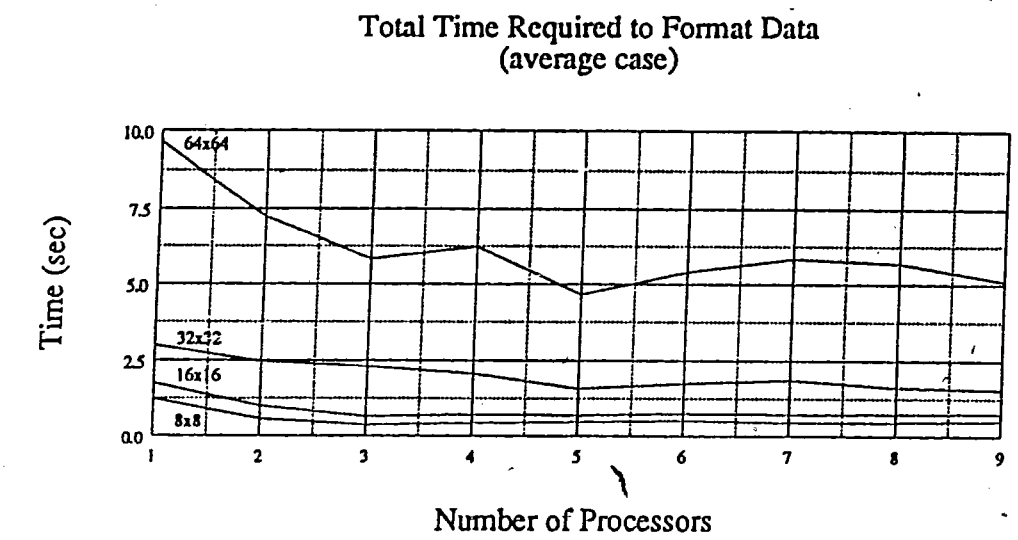
7.3 Presentation of Simulation Results

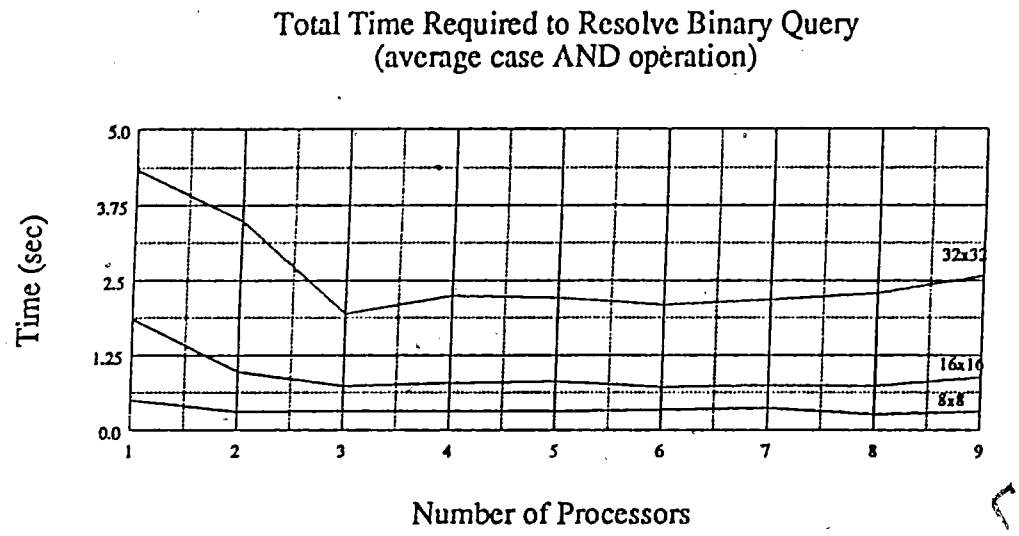
The results obtained from the following experiments provide some indication of the architecture's behaviour. To facilitate comparisons between the different operations, this section is organized in a manner where all of the results for a particular criterion are presented together. In the following six subsections, the results involving the MC are given independently of the slave processors. The results and observations for each statistic are preceded by a brief description of the statistic itself. The last three subsections deal with the slave processor results.

7.3.1 Total Execution Time

Of significant concern to the user is the amount of time required to complete a request, from its submission to the notification of the job's termination. Figure 7.1 presents the three graphs for the **LOAD UNFORMATTED**, average case **AND**, and best case **AND** operations.

In general, the **LOAD UNFORMATTED** and average case **AND** graphs show a substantial improvement in completion times as the number of processors increase. As expected, the case of the number of slave processors $P = 1$ requires the most time to complete a query. The results obtained here are similar to those generated mathematically in Chapter 3. The absolute time to complete the **LOAD UNFORMATTED** operation is expected to require the greatest time since it requires that the entire tree be generated — a case of two complete tree traversals being necessary. The time needed to complete the best case **AND** is significantly less than the previous two cases. This is understandable in that only





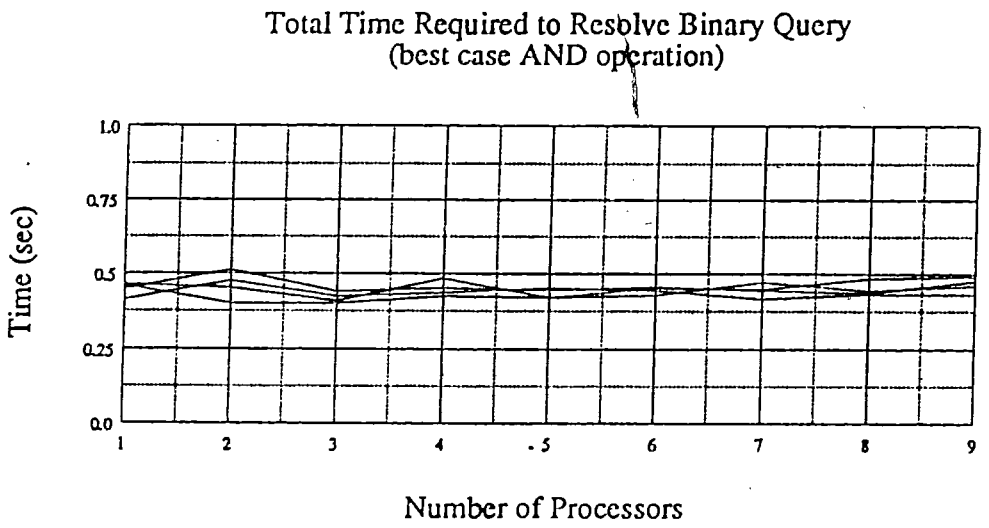


Figure 7.1: Total execution time of a user query

one level of the two trees have to be processed. As the remaining criteria are presented, it will become obvious that this last case provides a base from which the other operations can be considered. One interesting point which can be made about all three sets of results involves the trial where $P = 4$. After showing consistent decreases in processing time for $P = 2, 3$, there is a slight increase for $P = 4$. This is most evident with the 64x64 pixel image.

The results of Figure 7.1 can be used to determine the figure of merit for each operation. This value is the ratio

$$F_M = \frac{T_1}{N\,T_N}$$

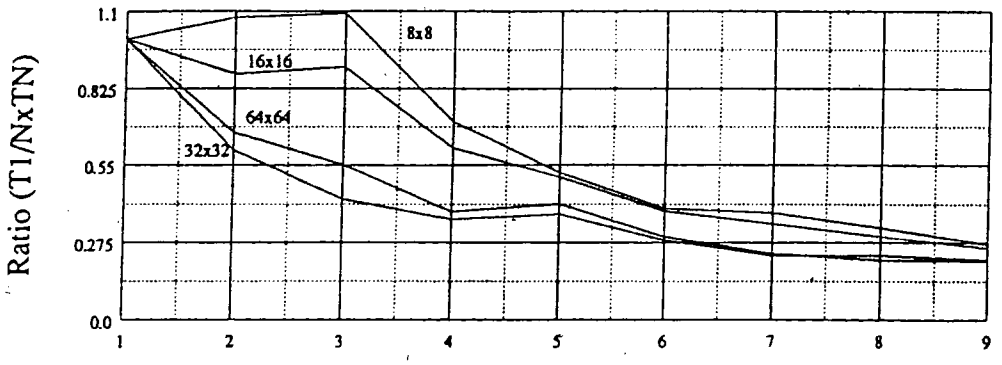
where T_1 is the time needed for one processor to complete an operation. The number of processors used in a particular run is given by N, and T_N is the completion time for these processors. If the $F_M = 1$ for some range $1 \ldots N$, then we have linear speedup. Figure 7.2 presents the figure of merit graphs for the three operations studied here.

The situation presented by the best case **AND** operation is as expected considering that the minimum processing time can be accommodated by one processor. Additional processors contribute nothing to operation speedup. With the remaining operations, the F_M reduction is not as severe over a span of the first five processors.

Two additional graphs are included here which deal with tree generation times. Both consider image sizes which are much larger than those actually tested. These sizes range from 128x128 to 1024x1024 pixels. Using Equation 3.7, and a constant processing rate of 1.5 msec/node, we arrive at the graph given in Figure 7.3.

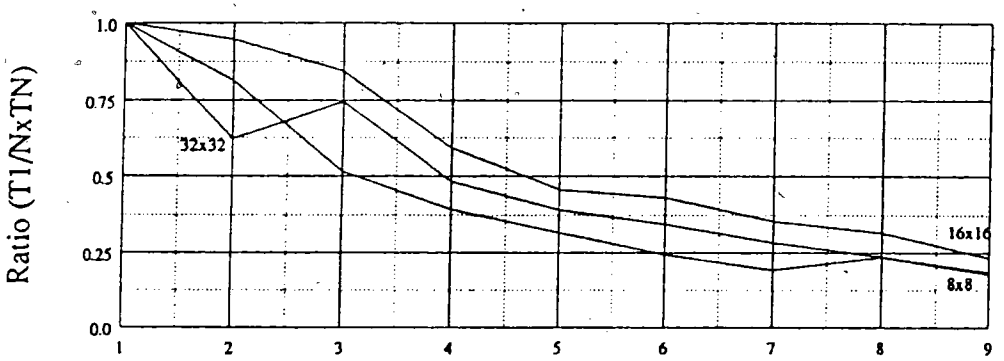
It is also possible to extrapolate execution times for these large images using the simulation results. The processing rate per node is much higher, as is indicated in Figure 7.4.

Figure of Merit Plot for Building a Tree



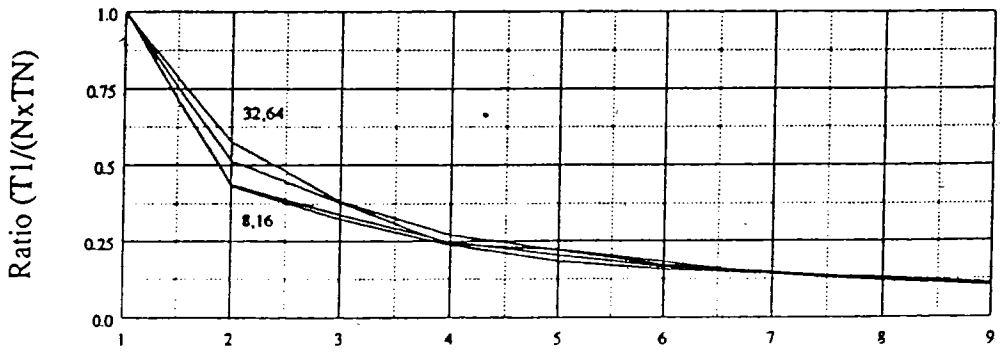
Number of Processors

Figure of Merit Plot for a Binary Query
(average case AND operation)



Number of Processors

Figure of Merit Plot for a Resolve Binary Query
(best case AND operation)



Number of Processors

Figure 7.2: Figure of Merit ratios of tree operation execution times

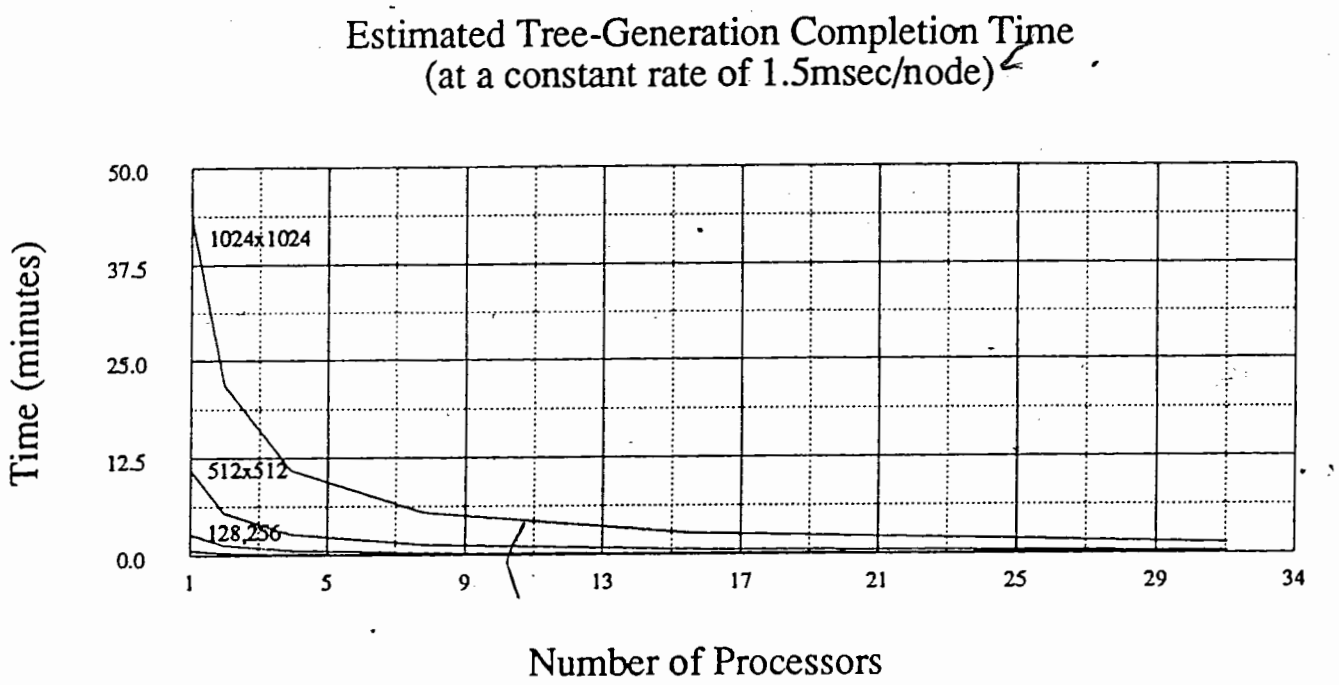


Figure 7.3: Estimated tree generation times for large images

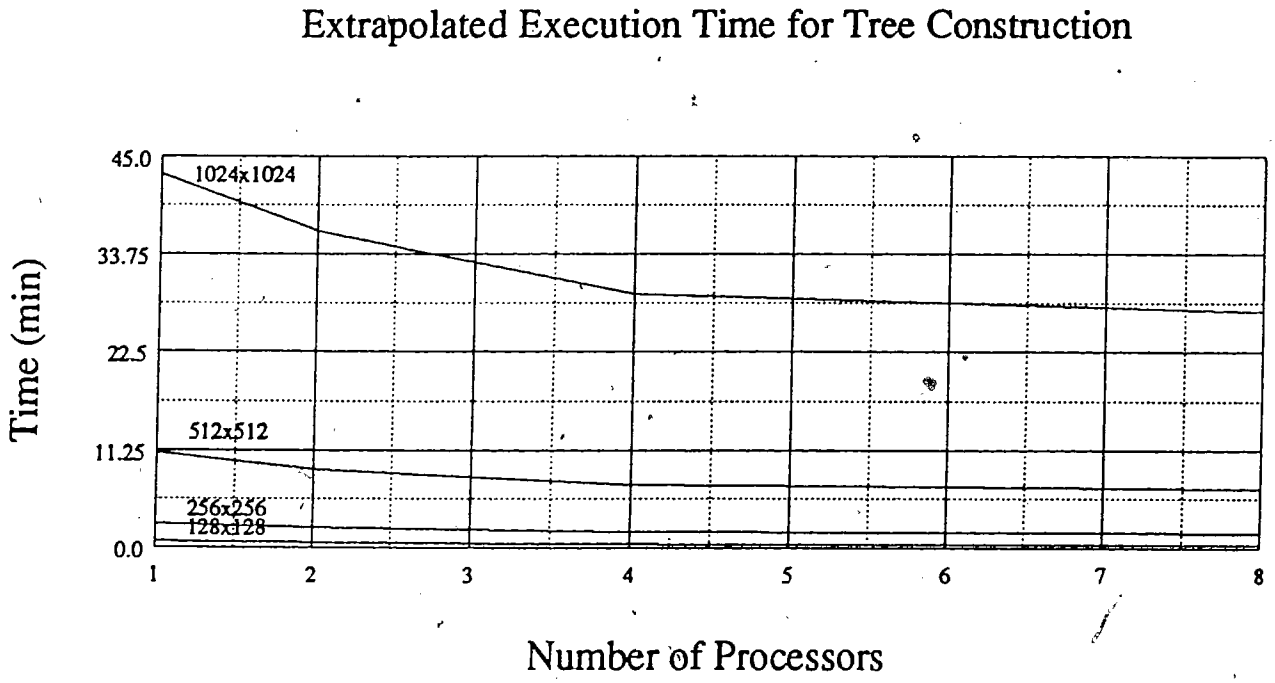


Figure 7.4: Extrapolated tree generation times for large images

7.3.2 Absolute Active Time

In initiating, mediating, and terminating a query, the MC is usually in one of two states, that of being busy or waiting. A reflection of the MC's busy state is its absolute active time, which is defined here as being the number of iterations of the inner processing loop that it completes to resolve a query. The results for the three operations are given in Figure 7.5.

The most noticeable observation from these graphs is the significant drop in operations executed from $P = 1$ to $P = 3$, and then the subsequent stabilization for $P \geq 3$, with the 64x64 pixel image. This indicates that although there are more tasks to schedule, by keeping the PrUs busy with job processing, the MC can stay idle for a period of time. With the other three images, the reduction in operations executed is not as significant. The best case **AND** situation requires the least number of MC operations, as expected. There is also some fluctuation in operation count as $P \rightarrow 9$, but this is acceptable considering the scale reduction. Comparing this figure to that of Figure 7.1, it is possible to see that the active times for the MC follow a similar pattern to the total execution time for the entire system.

7.3.3 Absolute Idle Time

During the course of resolving a user query, there are periods of time when the MC is in a waiting state performing no activity. For example, in a situation where all of the slave processors are busy executing their own independent tasks, the MC essentially just waits idle until some request or signal is sent by the other processors in the system. With the best case **AND** operation, there is very little processing necessary with the MC and slave processors since only a very few PrUs are required to complete the entire comparison. Although there seems to be a slight upward trend in idle times as $P \rightarrow 9$, the overall processing required with this case is very fast, and it is very difficult to comprehend that a stable environment is attained so quickly. The results can really be considered a baseline

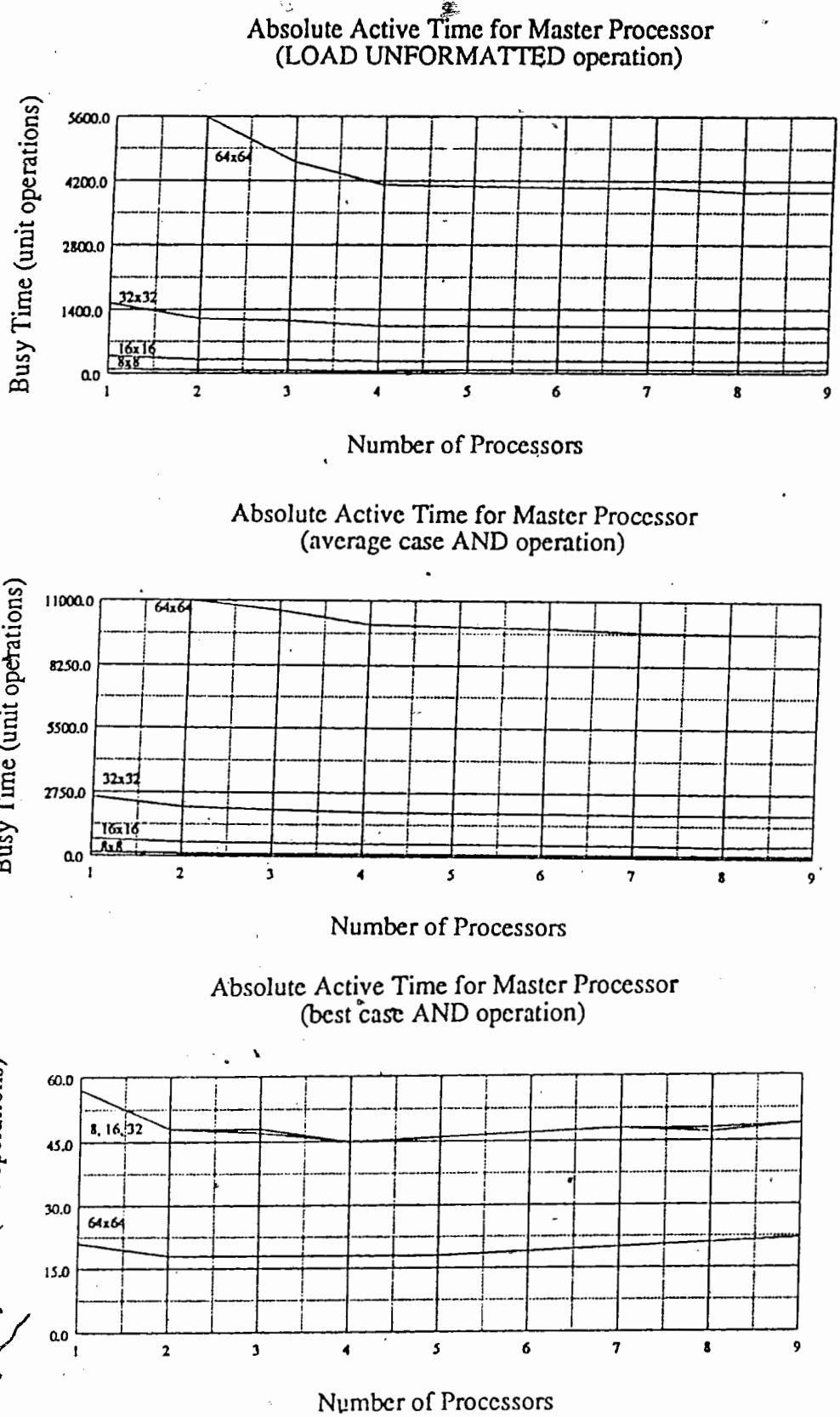


Figure 7.5: Absolute active time of a user query for the Master Controller

profile of the MC's activities, from which more complex examples can be compared.

With the **LOAD UNFORMATTED** query, there is another slight upward trend noticeable. However, in comparing the magnitude of these idle times with that of the absolute active times given in Figure 7.5, the increase is not significant. The idle time is essentially constant in the average **AND** case.

7.3.4 Relative Active Time

A more meaningful indication of how busy the MC is in coordinating the various system activities is the relative active time, which is the ratio of active to overall time based on unit operations. As Figure 7.7 indicates, the MC is busy for most of the time which is available to it. The inconsistent nature of the results for the best case **AND** with the 64x64 pixel image can again be explained by the fact that only one level of the trees has to be processed, and not all slave processors have to be used to fulfill these processing requirements.

7.3.5 Relative Idle Time

Figure 7.8 shows the results of the relative idle time for the MC. With the exception of two runs, most of the results are consistent. The best case **AND** operation with the 64x64 image, and the **LOAD UNFORMATTED** 8x8 query have relative idle time values which are slightly higher than the results of the other experiments.

7.3.6 Maximum Task Stack Length

This statistic provides some information as to the rate that the MC dispenses with the tasks which are sent to it by the slave processors.

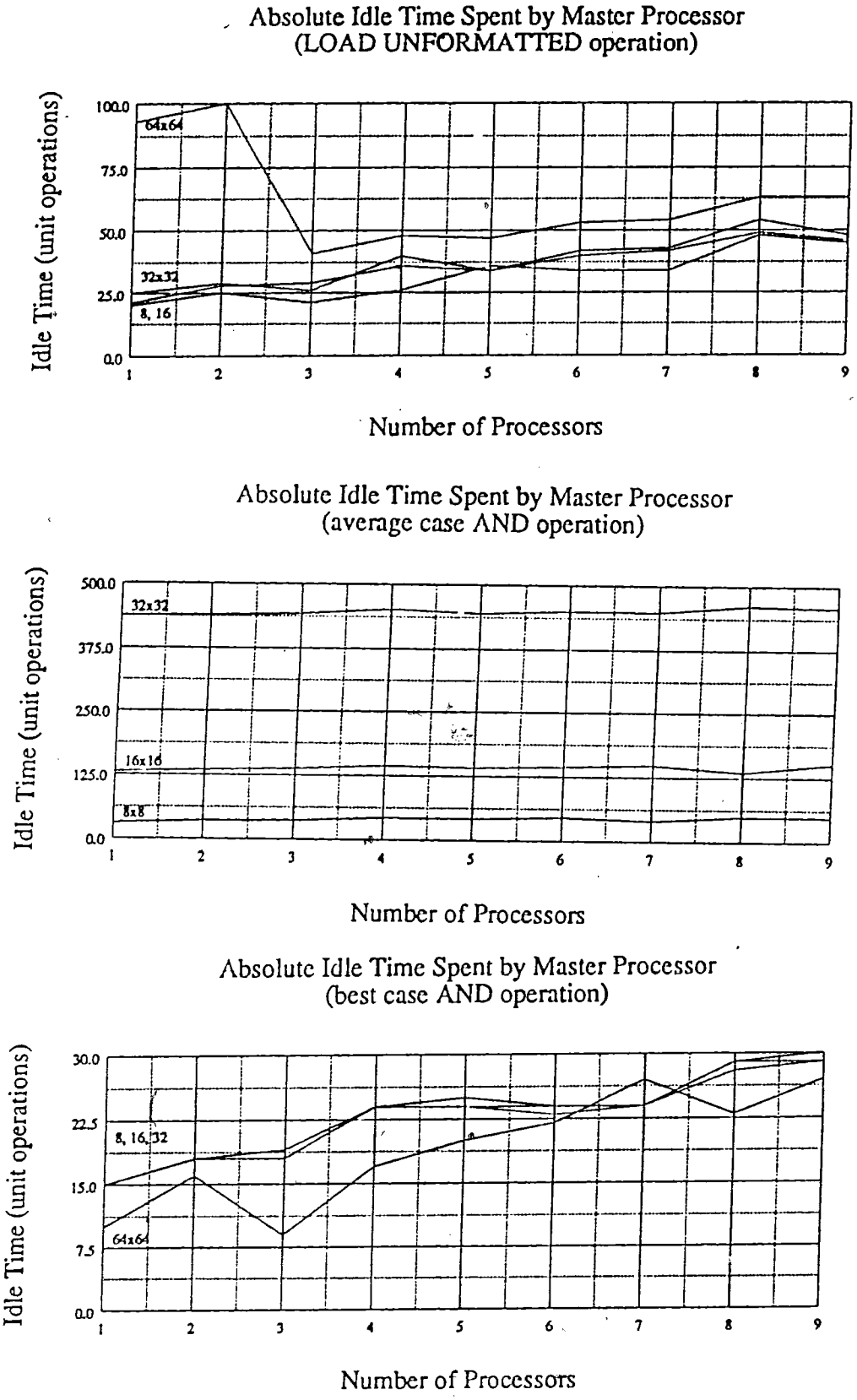


Figure 7.6: Absolute idle time of a user query for the Master Controller

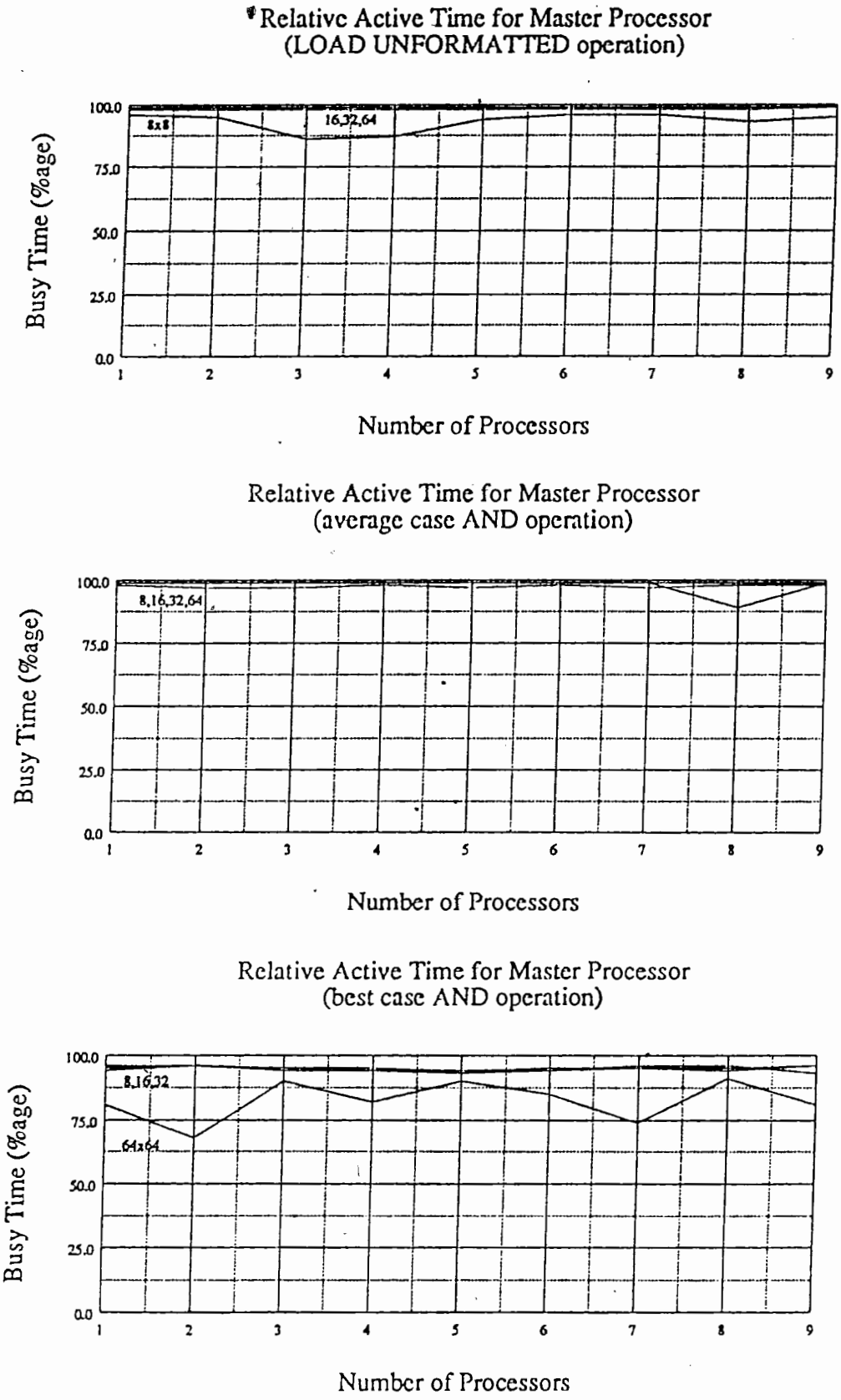


Figure 7.7: Relative active time of a user query for the Master Controller

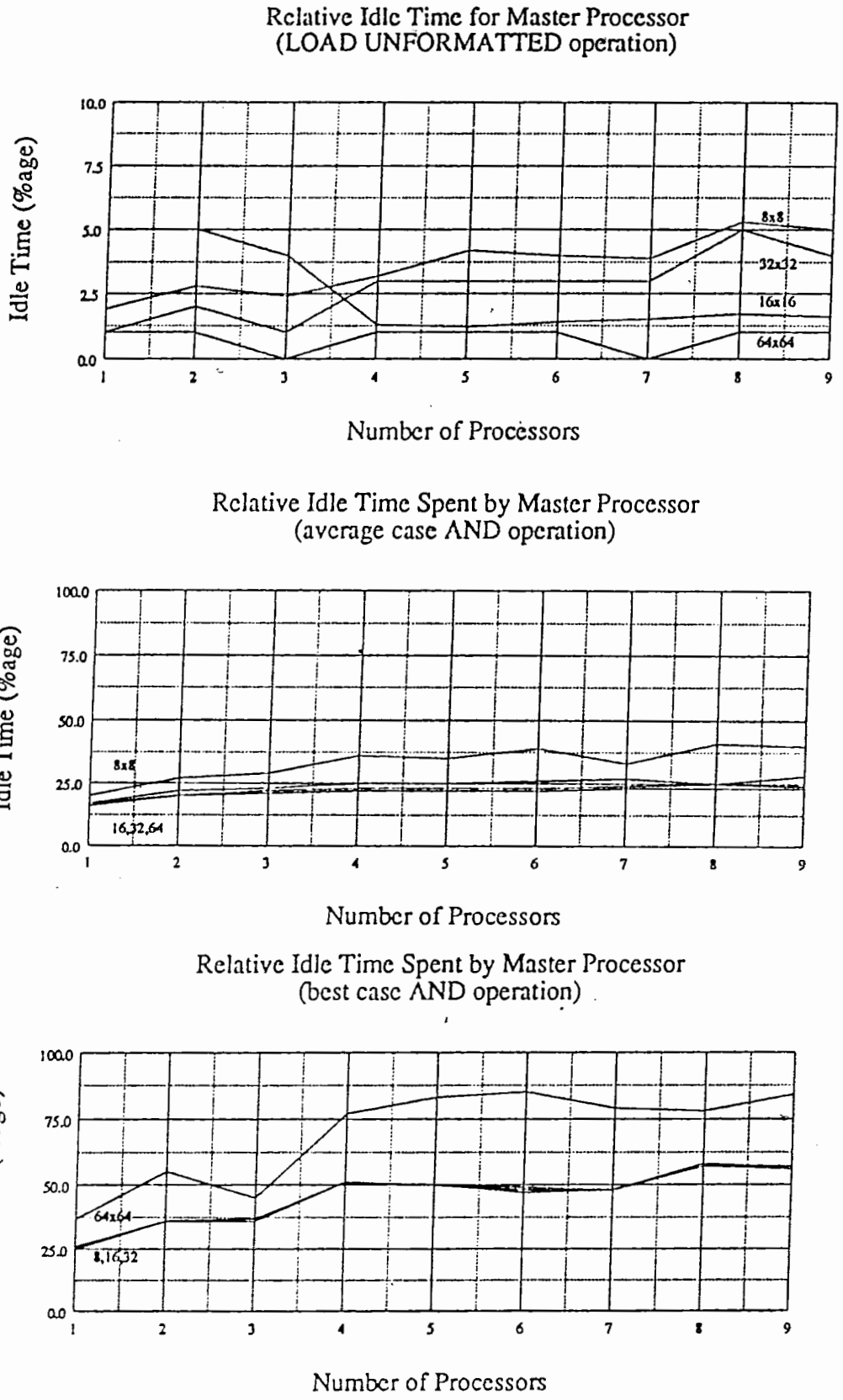


Figure 7.8: Relative idle time of a user query for the Master Controller

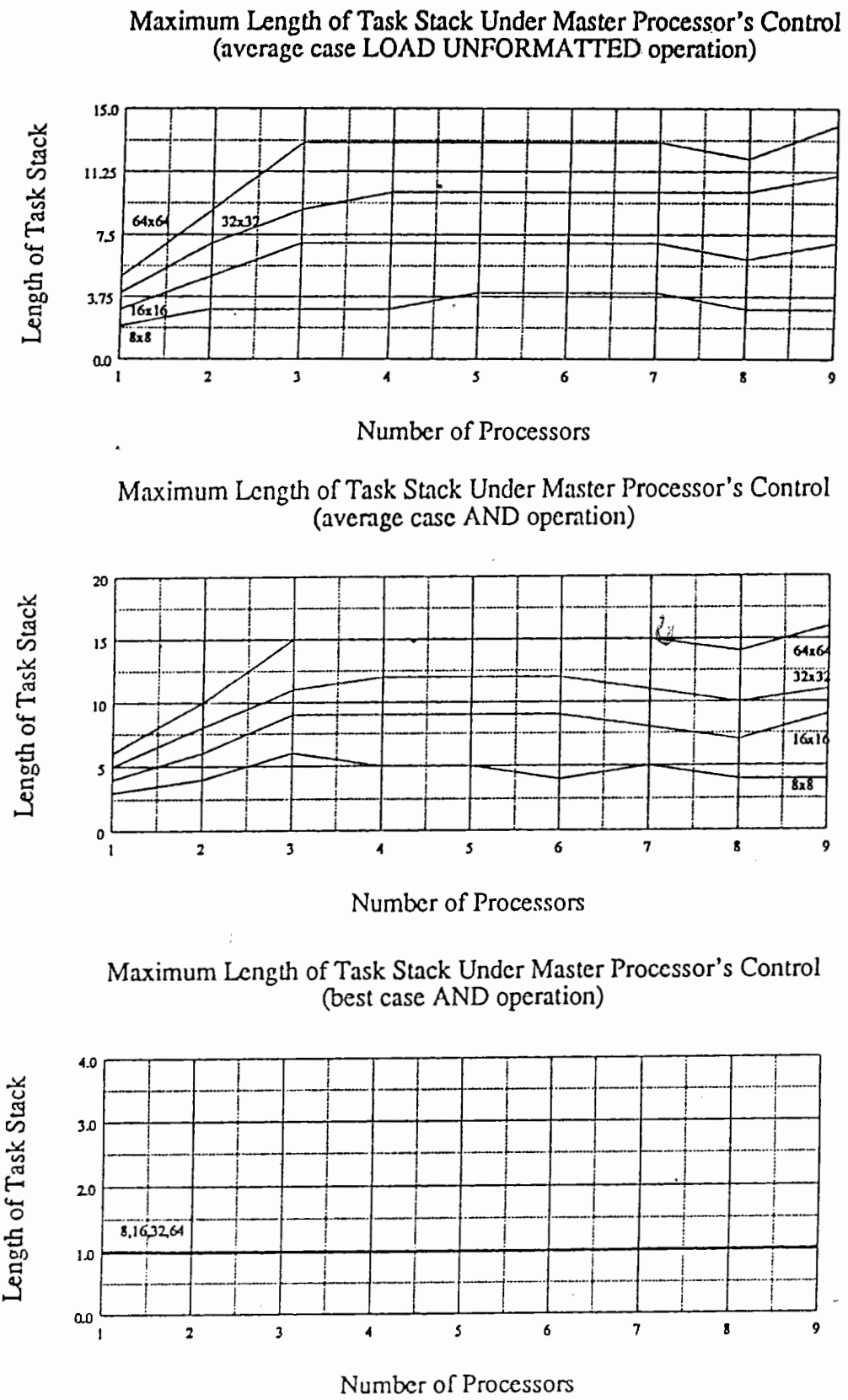


Figure 7.9: Maximum task stack size in the Master Controller for a user query

Figure 7.9 presents these results. In the best case **AND** query, the consistent $LENGTH = 1$ result for all processor and image combination can be explained by the fact that there is only one task which must be created and processed. This is the task which compares the root nodes of the involved trees. With the other two queries, there is a steady increase in length of the stack until $P = 3$, at which point there is very little, if any, change. This increase, and levelling off can be explained by realizing that as the number of slave processors increase, there will be an appreciable increase in the number of tasks received by the MC. With the given image sizes, a state is reached where the number of tasks being received by the MC can be accommodated by the availability of slave processors.

7.3.7 Average Utilization Time for Slaves

The results for the slave processors' average utilization time is given in Figure 7.10. With the **LOAD UNFORMATTED** and average case **AND** tests, there is an obvious decrease in average processing time being performed by the slaves as their number increase. One interesting observation is that with the former, the utilization of the processors does not decrease significantly for $P > 3$ when compared to $P = 3$. Of course, this is a reflection of the image sizes that we are using. As indicated in Chapter 3, this special value of P will increase as the image size increases. The results of the best case **AND** query are essentially linear if experimental error is taken into account. This case can really be considered as a lower limit on the amount of time that is needed by the processors

7.3.8 Average Number of Tasks Processed by Slaves

Each slave processor generates and processes tasks which are needed for the representation's operations. The purpose of this criterion is to show the direct correlation between processor utilization time and the number of tasks that the slave processors must execute. This is done by comparing Figure 7.11 with that of

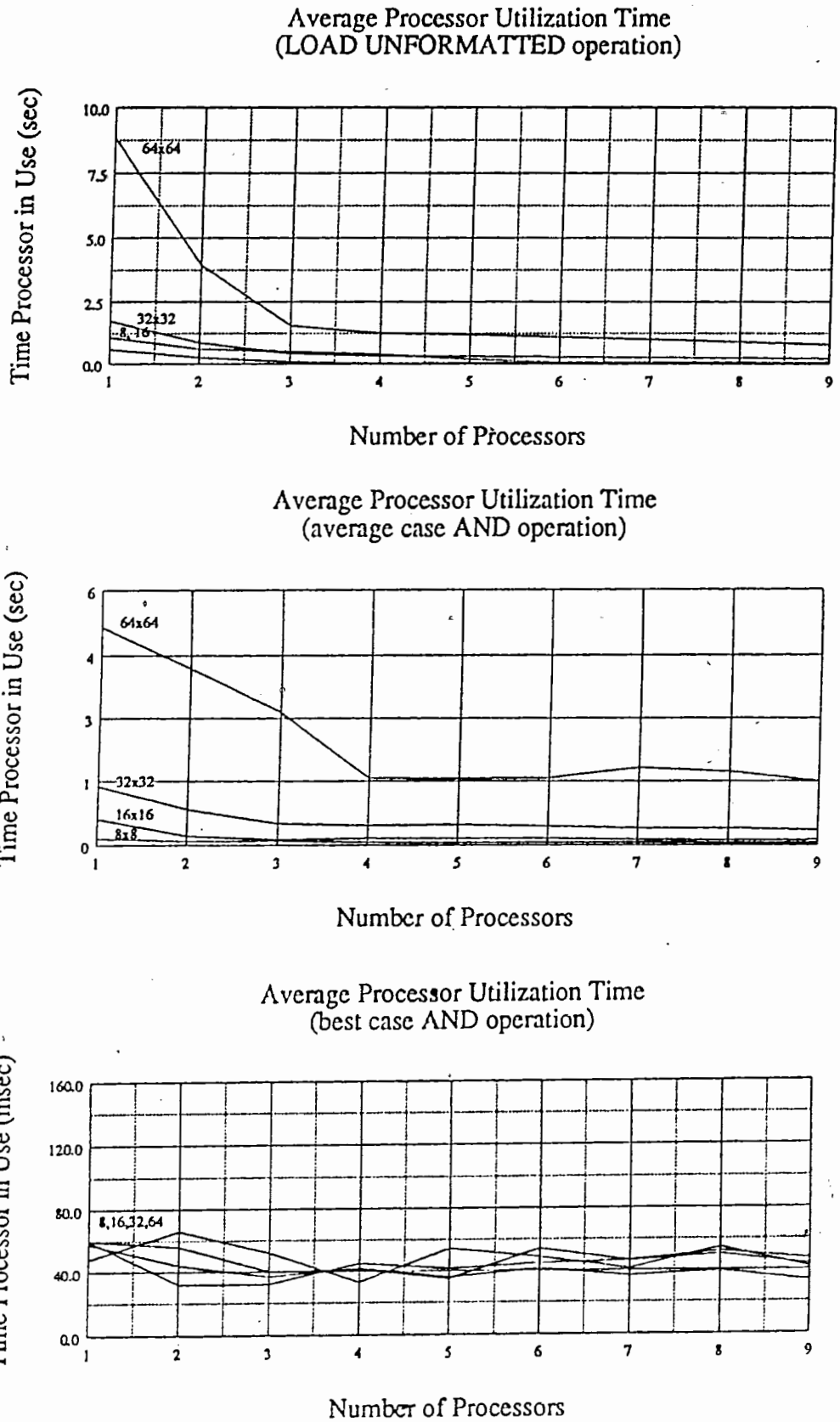


Figure 7.10: Average slave processor utilization time for a user query

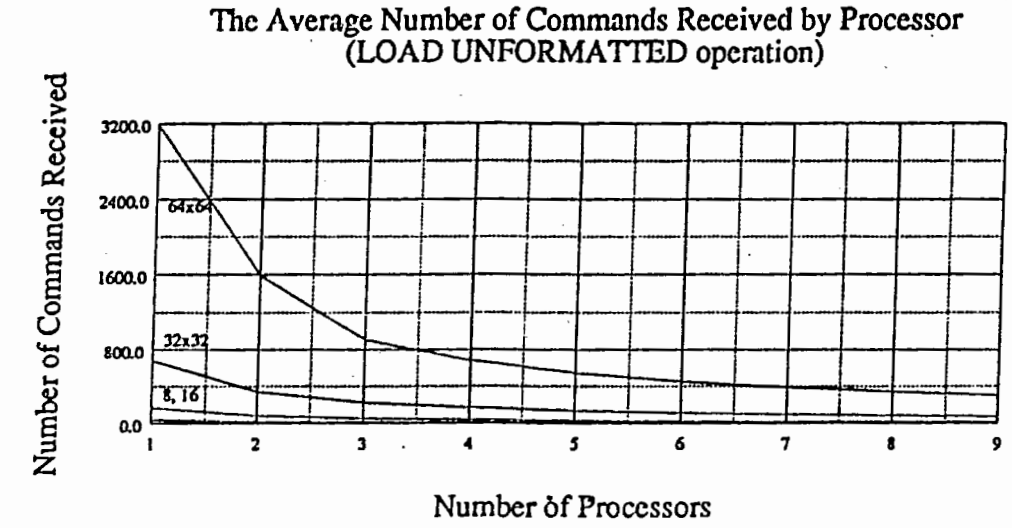
Figure 7.10. As expected, the trends evident in the first two graphs of both figures are consistent. Comparing the appropriate graphs for the best case **AND** case, there does not seem to be anything in common . The graph in Figure 7.11 is consistent with that of the **LOAD UNFORMATTED**, and average case **AND**.

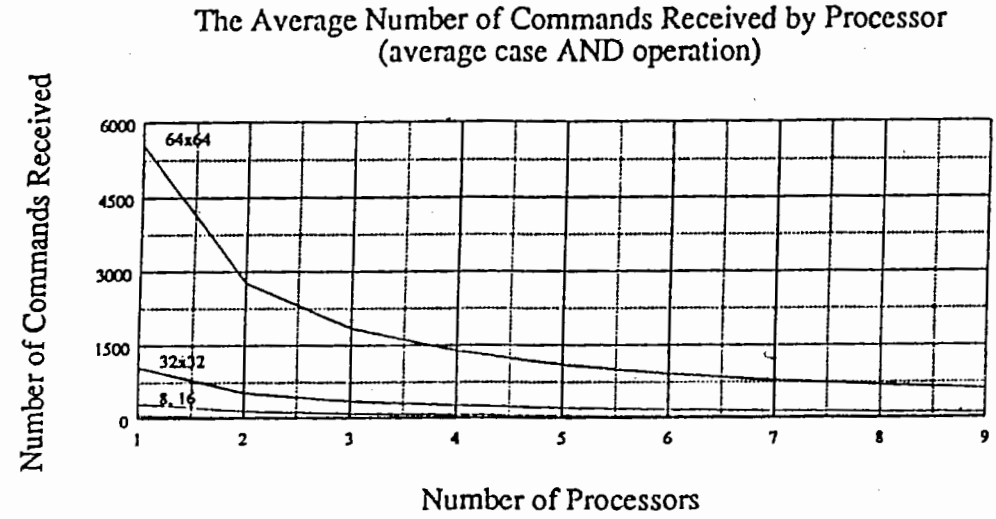
7.3.9 Average Number of IMU Accesses for Slaves

The distribution of shared memory accesses by the slave processors is presented in Figure 7.12. In all cases, the trend is for a decrease in access frequency as P gets larger. This is expected since the number of shared memory accesses is a function of a PrU's processing load. The more tasks that a processor must execute, the greater the probability that some of these tasks require information from the shared memory.

7.4 Discussion of Simulation Results

Of the various statistics which have been accumulated during these experiments, it is the total execution time which is of most interest. The other criteria may provide very favourable results which confirm the architecture's mechanism of action, but if the execution times are excessive, the architecture would be unacceptable. Chapter 7.3.1 presented some observations about the execution time, the principal one being that there is a marked decrease in completion time between $P = 1$ and $P = 3$ on the two cases (**LOAD UNFORMATTED** and average case **AND**) which require a reasonable amount of processing. For values of $P > 2$, the execution times vary by no more than 10% from the average. Given the image sizes which were used, there really is no need to use large values of P. Using the equations of Chapter 3, and considering the trends shown in Figure 7.1, queries involving larger image sizes will follow the same pattern as here, with an appropriate P providing a lower limit of consistent execution time.





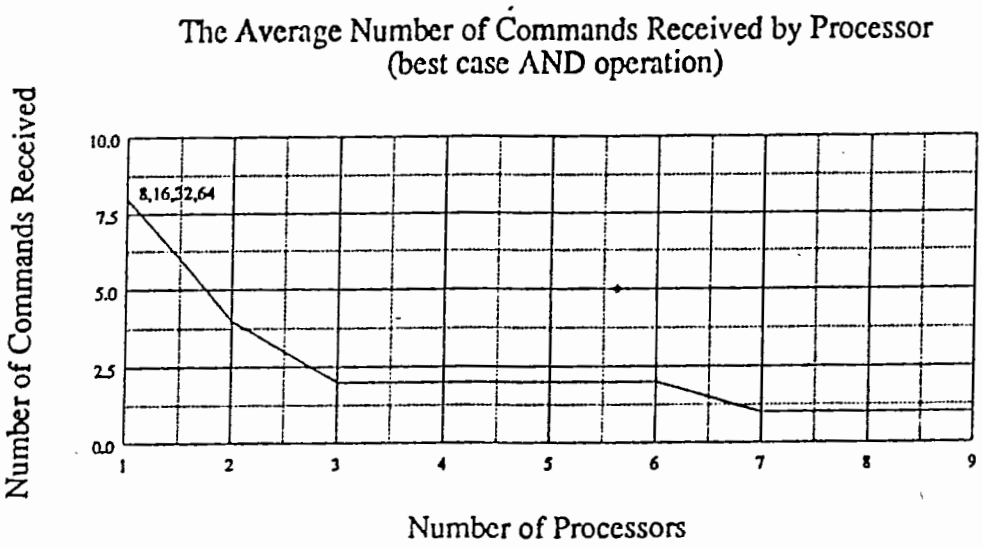
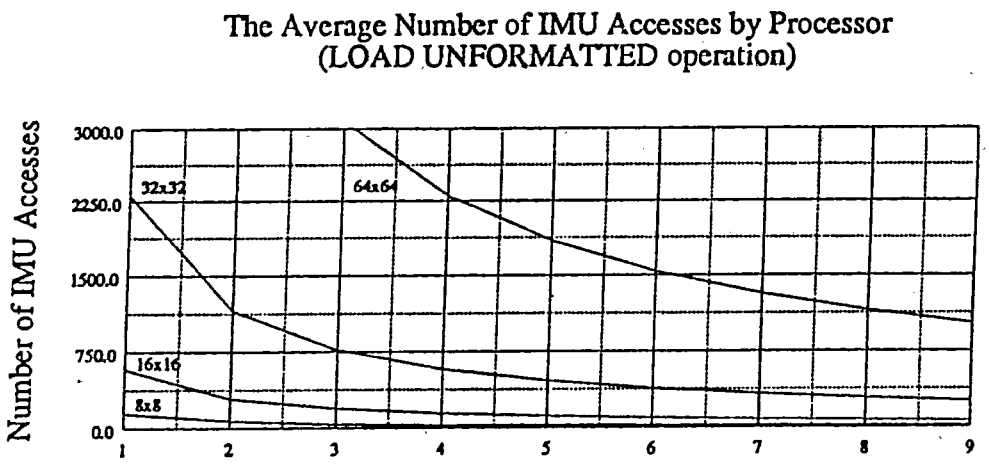
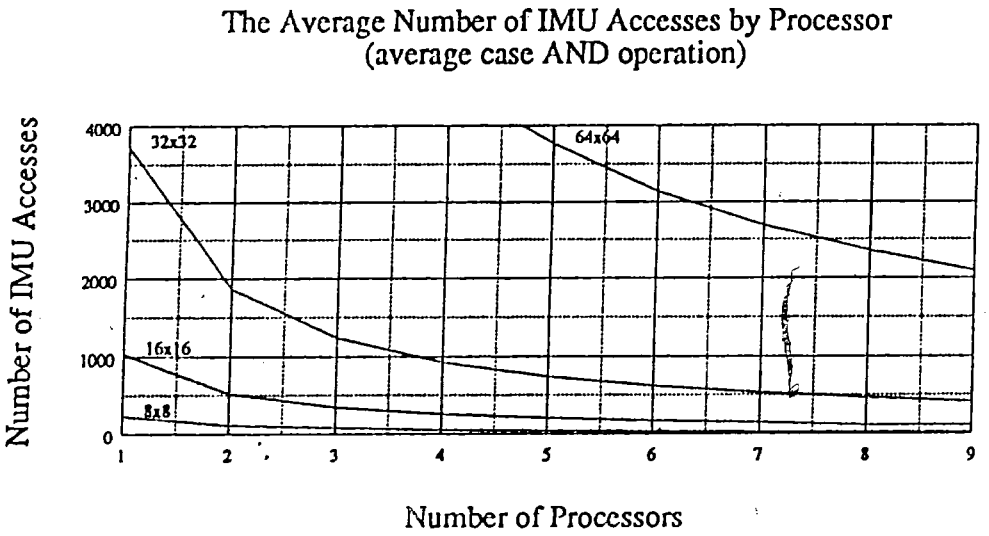


Figure 7.11: Average number of commands executed by a slave processor





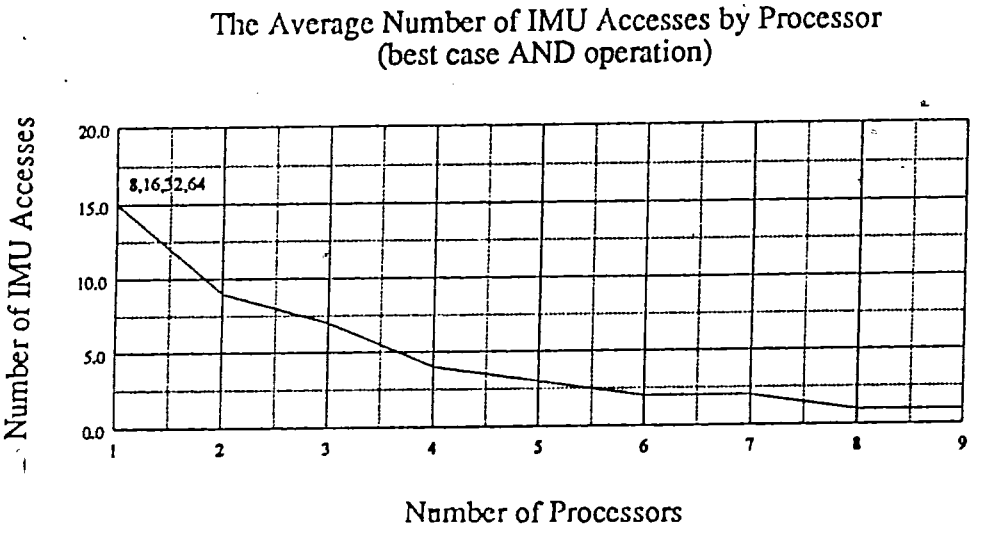


Figure 7.12: Average number of memory accesses by the slave processor

The actual execution times obtained during this simulation are measured in units of seconds, which may suggest that the architecture is very inefficient for even small processing tasks. However, the implementation is entirely software-based, and uses the relatively slow method of message passing to transmit tasks from one process to another. If the number of unit operations given for the absolute active times of the MC are assumed to take 1μsecond each to complete, an operation such as a binary **AND** on a 64x64 pixel image requires a favourable 11 milliseconds. The simulation of the interconnection network also contributes to the inflated execution times. If the magnitude of these quantitative results are ignored, the obvious trends indicated by this simulation do reflect the system's anticipated behaviour under various situations.

Although the network is a major cause of the slow execution times, it itself is dependent upon another component of the system. This is the number of shared memory modules. With such a small number of modules (4), the number of accesses to these modules will be much higher than if there were more modules. By increasing the module count, there will be fewer accesses per memory unit. This reduction in contention will also lower the execution times. The figure of merit ratios can be expected to improve significantly as a result of this. Obviously, if the application involves a large data size, it will be beneficial to have a large number of memory modules.

Figures 7.3 and 7.4 were included to consider more complex images. The differences in execution times can again be attributed to the overhead resulting from contention through the network due to an insufficient number of memory modules.

The increasing size of the task stack for larger data sets being accessed by the MC requires that its local memory be of sufficient size to accommodate such a large data structure. It is highly unlikely that there are a sufficient number of hardware registers on the processing chip to store all of these tasks. One option available to the designer is to place the top R tasks which are on the top of the

stack into the chip's R stack registers. Newly arrived tasks to the MC are placed on this stack, while those tasks at the bottom of these registers are sent to the MC's local memory. The tasks at the top of the stack are the most volatile, and by placing them into fast hardware registers, they can be dispatched to the next available slave processors quickly.

As the results indicate, the extent to which the MC is occupied with its processing, essentially just task scheduling, is dependent upon both the type of query and image size. Large images may be processed quickly if the query is as straightforward as the best case **AND**. Moderately sized images may require more processing time if the query is similar to the average case **AND**. The processing load on the MC can provide a bottleneck if it is proceeding at full capacity. One method of reducing this possibility is to use the alternative suggested earlier, that of allowing the slave processors to do their own scheduling. A second approach is to distribute the processing load between additional MCs. With this latter method, questions such as how will tasks be allocated to the MCs (possibly via a shared memory scheme), and what is the optimum number of MCs are raised. Neither of these alternatives have been simulated for this dissertation.

With respect to the slave processors, observations have been made which indicate that after some value of P for a given image size and query, there is not that great of an improvement in processing times. As to the ramifications of this point, the designer must consider the application's data and operation set. Initially, some minimal value of P may be sufficient for a start-up system. However, as the needs of the application increase, facilities must be in place to allow for the addition of more slave processors. One concern is that of processor cost. If this cost is prohibitive, the number of slaves used should be that of this optimal P.

Chapter 8 Discussion

8.1 The Representation

Before the three architectures designed for the 2^N-ary tree can be discussed, it is necessary to consider the representation itself. By using straightforward set theoretic principles, it has been possible to define a general representation which has a very broad range of applicability. The large number of operations provided by set theory makes this representation even more attractive to the user. If it is possible to define a collection of objects as a population, then it is possible to use our defined set representation on these objects. Another critical property of this representation involves the fact that subsets of objects within an instance of the representation are disjoint entities. Therefore, it is possible to apply any valid operation on these subsets in parallel. Understandably, the parallel execution of an operation will be completed before its sequential version does.

A transformation was performed on this general set representation to give us the 2^N-ary tree representation. The operations and properties of the set representation are also valid within the scope of the 2^N-ary tree. This transformation confers some order upon this new scheme, which simplifies the process of developing simulation models to test these properties and operations.

Each of the three architectures presented provide a significantly different approach in implementing this 2^N-ary tree representation. At one extreme, we have the mapping of binary trees onto a VLSI array of processing elements, while

at the other, the system configuration consists of shared memory and a linear arrangement of high-level processing elements that execute tasks from a sophisticated Master Controller. Intermediate to these two systems is the architecture utilizing linear arrays of processing elements that are linked via 2^N-ary tree mappings on interconnection networks. Each offers certain advantages over the others.

8.2 Performance

Obviously, performance is one feature which must be considered in any comparison or evaluation. The shared memory simulation has provided the only results from which the validity of the theoretical analyses can be determined. Some of the results were discussed earlier in the dissertation. Based on these results, a primary observation is that the number of memory modules in the system is critical. The complexity of the interconnection network is directly related to this number. Therefore, what may be considered as a deficiency in the network capability can actually be attributed to the memory modules. With the system that was tested in the simulation, the interconnection contention was definitely a result of the small number of modules.

With the other architectures, we can compare the time complexity for the execution of binary operations. From Equation 4.4, we see that the complexity for the **PE-ICN** system is $O(2^N L)$, where $L = \log_{2^N} P + 1$, and P is the number of objects in the instance. In the original analysis, the pixel width w was used in place of P, but the relationship $P = w^2$ makes these two entities interchangeable. For the VLSI system, the time complexity of a binary operation is $O(L^2)$. Here, $L = \log_2 P$. For the case of $N = 1$ in the **PE-ICN** system, we are actually mapping a binary tree onto the rows of processors. A direct comparison to the binary tree mapping on the VLSI array shows that the former architecture is faster by a factor of L. A general comparison between these two complexities may

be made with the following approximation:

$$P_{\text{VLSI}} = P_{\text{PE-ICN}} \tag{8.1}$$

$$L^2 = 2^N \log_{2^N} P \tag{8.2}$$

$$\log_2^2 P = 2^N \log_{2^N} P \tag{8.3}$$

$$N \log_{2^N}^2 P = 2^N \log_{2^N} P \tag{8.4}$$

$$N^2 \log_{2^N} P = 2^N \tag{8.5}$$

Using Equation 8.5 as a ratio between VLSI and **PE-ICN** times, as the value of N increases, it will take the latter system significantly more time to complete the same type of operation. The major contributing factor to this relatively inefficient **PE-ICN** system is the time required to pass children values to a parent node in a sequential manner. If this can also be performed in parallel, the complexity of passing through a tree is reduced from $O(2^N L)$ to $O(L)$. The above comparison with the **PE-ICN** system has used the case where the number of levels in the trees is less than the number of **PE** rows. If we have the opposite case, then it is even more obvious that the VLSI system is more time efficient than the **PE-ICN** system.

8.2.1 Multi-operand and More Diverse Operations

The operations that have been considered to this point have either required single or double operands. There are two approaches that can be used in evaluating queries that require more than two parameters. As the representation provides a significant number of set-type operations, it is possible to use principles such as associativity, commutativity, distributivity, and DeMorgan's laws to reconstruct the query as a series of binary operations. For example, the query $A \cup B \cup C$ can be evaluated as $(A \cup B) \cup C$, where the result of $A \cup B$ becomes the other operand for the second union operation. The second approach involves the sequential evaluation for each level of the operand trees, before the lower levels are

considered. With an example as $A \cap B \cap C$, the intersection of the root node for A and B is stored temporarily in the processing element so that the root for C can be obtained and applied with this value on the intersection operation. If necessary, subsequent tree levels can be processed in a similar manner.

The most efficient architecture in this case is that system which stores similar trees in the same memory module, be it in shared memory or as part of a processing element's local memory. The VLSI binary tree and shared memory schemes are significantly more effective than the **PE-ICN** architecture. The binary tree has one root node, so that all tree instances must begin at this **PE**. Each **PE** has sufficient storage to accomodate a number of trees. The bit-serial feature also allows for quick node retrieval of the necessary trees from this local memory. The shared memory system can also process such requests. If equivalent nodes of R operand trees are stored in the same memory modules, the **PE**s maintain their paths through the **ICN** for R read cycles. This allows sufficient time for the R nodes to be passed to the requesting **PE**. The **PE-ICN** system is the most ineffective method of the three architectures in that equivalent tree instances may not be stored using the same type of switch settings through the various **ICN**s. This requires that additional synchronization steps be taken to allow values along a processor row to be passed between **PE**s to bring the required node values together in one **PE**.

The operations that have been used in the simulation and the various analyses are set based . However, there exist other algorithms that use the 2^N-ary tree data structure which are not. Some of these include raster-to-quadtree conversion [34], and location of nearest-neighbours [39]. Features of the representation, such as disjoint subtrees, still exist. The sequential form of the algorithms will have to be modified to take advantage of these properties.

8.3 Consideration of Large Databases

The test cases considered in the simulation have involved relatively small data sets. In applications such as geographic information systems, data representing areas spanning many kilometers must be transformed into the tree scheme. For example, if a region of about $1000km^2$ is mapped to 100 meters, it is necessary to generate a 13-level quadtree. The complete tree requires approximately $90x10^6$ nodes. Attempting to map such a tree onto either of the first two architectures would be very difficult due to the tremendous number of nodes that are involved. The folding tree mapping of the **P-INA** system would most probably be required. Task synchronization between levels would be very difficult to maintain if multiple tree foldings are needed. The shared memory system provides a reasonable solution as the tree can be easily managed in the memory modules. The interconnection network can also be expanded easily if more memory units have to be added.

In the case of systems with a limited number of modules, it is necessary to divide the trees into subtrees of consistent depth. While the top block of subtrees occupies the modules, the next block can be brought into the channel buffers from secondary storage. The transfer of information between these two components is potentially an input/output bottleneck that must be resolved. A high-speed data bus between the buffers and modules can facilitate the swapping of these subtrees.

8.4 Fault Tolerance

One issue to consider is the fault tolerant behaviour of these systems. Fault tolerant computing can be defined as the process by which an algorithm is executed correctly even in the presence of defects in the system [40]. Of course, it is assumed that algorithm has been implemented in the appropriate fashion. A failure can be considered as some physical damage while a fault is generated whenever some logical value differs from its expected value. One approach to

make a system fault tolerant is through redundancy. This can be in the form of repeated calculations, or through extra hardware and software. The more economical option available to the designer in handling permanent faults that are caused by some failure is to use hardware redundancy. However, if we are dealing with faults that are caused by some system inconsistency or external influence, the practical solution involves repetitive calculations. The following discussion on the fault tolerant behaviour of each system is very general, and no attempt is made at providing an in-depth analysis of concepts such as fault detection, diagnosis, isolation and repair.

The shared memory and VLSI array systems provide a greater number of advantages when compared to the alternating **PE-ICN** scheme. From a high level perspective, memory is less prone to malfunction that the more complex processing elements, with components such as the CPUs and ALUs. The shared memory modules of our third architecture provide stability to the system. If any of the memory modules do happen to malfunction, entire subtrees of objects may be lost. This requires that memory backups be done at regular intervals. On a finer scale, data corruption can be controlled using error-correcting code on memory.

With the shared memory system, the **PE**s, and **MC** are the critical components which must be protected with some fault tolerant mechanism. If the **MC** malfunctions, the entire system will shutdown. In the event of an **MC** breakdown, an alternate **MC** may continue in its place. The results of the simulation have shown that after a certain number of processors P in an E **PE** system, where $E \geq P$, increasing P will not result in exceptionally great advances in throughput. These $E - P$ **PE**s can be considered as the redundant component. The **MC**'s scheduling mechanism does not require any significant action in the case of a **PE** malfunction. A signal is received from the defective **PE** to indicate that it is in a failure state, and to allow the **MC**'s table of available slave processors to be modified to reflect this component breakdown. Scheduling of tasks continues with this reduced number of **PE**s.

The shared memory system is very modular, consisting of memory units, and different processor units. Each of these may reside on an individual microchip. If any of these modules become faulty, they can be isolated from the rest of the system. Repair may only require that the unit be replaced by a similar module.

In the case of the VLSI array, the situation presented by Youn [50] calls for interconnection buses to run horizontally and vertically between the **PE**s of a module. At each bus junction there exists a switch which affects the four **PE**s that surround it. Regardless of whether we are dealing with interior or leaf nodes, a switch is used to connect a parent node to its two children, thus leaving one redundant **PE** per four-unit cluster. If a **PE** in a cluster malfunctions, the extra **PE** can be switched on in its place.

The situation presented by the alternating **PE-ICN** architecture is significantly more complex. There are two additional module types to contend with, those which are responsible for controlling the rows of slave processors, and those that control the **ICN**s. The major concern arises with the **ICN**s. There may also be multiple mappings through the **ICN**s in the case of representation instances which require more tree levels than there are processor levels. The **ICN**s must function properly otherwise these mappings will be corrupted. There is some inherent redundancy in the system. In Chapter 4.2.2, it was shown that for the 2^N-ary tree with E slaves per row, we have a situation where for some row r, any $\frac{E}{2^N}$ **PE**s can be used for the row mapping. There are then $E - \frac{E}{2^N}$ **PE**s in this row r which provide adequate redundancy.

8.5 Expandability

Expandability of systems is another feature to consider in comparing these systems. The modularity of the shared memory architecture makes it relatively simple to enhance. If extra processing power is needed, additional slave processors may be added. These extra slaves may require that additional switch levels be

added to the **ICN**. Increasing the number of memory modules may also be necessary. These additions will have little affect on the **MC** and its task scheduling. Of course, the controller will have to be notified of these extra slaves, but the scheduling procedure remains the same.

The **PE-ICN** is also very modular, and the vertical expansion of this system by adding extra rows of slave processors is straightforward. Connection networks between these new rows will also have to be included. However, the situation is more complex if it is decided to add extra slaves to a particular row. This type of horizontal expansion requires that each row of processors also get the same number of additional slaves. This is necessary, otherwise adjacent interconnection networks will be inconsistent in size and complexity.

With the VLSI system, a single microchip may consist of an X-level binary tree, where X is some reasonable value such as 4 or 5. This can be considered as the basic building block. Connecting some of these chips into a cluster allows trees of greater depth to be stored and processed. However, as these clusters become greater in size and occupy more area, the distances between adjacent clusters also increases, resulting in increased communication times. Minimization of these distances through alternative clustering techniques will make expansion of this system worthwhile.

8.6 Other Comments on the Architectures

In designing the VLSI architecture, the use of a binary tree mapping of the processing elements of the array was the most beneficial. The binary tree provides a compact structure because of its small fan-out. In trees with greater fan-out, such as the quadtree and octree, the structure requires more area to connect all children to a parent node. With this greater area, there are more unused processing elements on the array unless some irregular mapping scheme can be developed. Another reason for using the binary tree involved work which has

already been done on binary tree mappings by different research groups. Efficient mapping schemes that incorporate some interesting features have been devised for the binary tree. For example, consideration of fault tolerance has been included in these mapping schemes. Even with these features in favour of the restricted 2^N-ary tree mapping, the question may still arise as to why not use a mapping for $N > 1$ where there will be fewer levels in the tree, and therefore, faster processing times. To this, one may answer that the compactness of the binary tree more than compensates for this need of extra levels in the tree. One consequence of this is that the interprocessor distances are much smaller. Therefore, there will be less delay in travelling between these additional levels. In addition, the advances being made in VLSI technology allows for more complex and faster processing elements.

The discussion to this point can be summarized as presented in Table 8.1. The relative ranking is, for the most part, a subjective measure of the expandability, fault tolerant capability, performance, and complexity of each architecture. The shared memory approach is the most favourable of the three, while the **P-INA** structure is least effective.

A fair indication of the overall cost effectiveness of the three systems is provided by the above rankings and criteria. The shared memory architecture's features make it an attractive system. With the cost of microprocessors and memory decreasing, justification for the use of this system, which relies heavily on these two components, is obvious. The interconnection network can be readily expanded to accommodate any reasonable increase in system requirements. The simple structure of the switching elements also translates to cost efficiency.

With our other two systems, it is more difficult to present arguments in favour of their cost effectiveness. Because of the effort required to develop and set up these systems, one may counter that a powerful uniprocessor with the necessary software implementation of the representation is preferred. The task facing the designer is to arrive at a decision based upon these options. There is no hard and fast rule which stipulates when a multiprocessor based system should be used.

	I	II	III	IV
P-INA	B	C	C	C
VLSI-based	C	B	A	A
Shared memory	A	A	A-B	A

Relative ranking: A - best
B - OK
C - not too good

I ---- Expandability

II ---- Fault tolerance

III ---- Performance

IV ---- Complexity

Table 8.1: Relative ranking of the three systems

8.7 Consideration of Alternative Architectures

The architectures which have been presented are three methods by which the 2^N-ary tree representation can be implemented. Recent advances in parallel processing have introduced further alternatives for the system designer to select from in taking advantage of the 2^N-ary tree's properties. For example, the hypercube topology has been used at Caltech in the development of the Cosmic Cube [38]. General purpose parallel computers, such as Intel's iPSC personal supercomputer with its 32, 64, or 128 processing nodes connected in a hypercube arrangement, are now becoming commercially available. With the iPSC, 16-bit microprocessors are used as nodes, with 512K bytes of memory. Adjacent nodes are connected via ethernet links which allow for 10Mbit per second transfer rates. The nature of the hypercube model allows for different applications to use the same structure. For example, computationally intensive applications such as computer vision have successfully used the hypercube to achieve useful improvements in processing times [26]. Within the context of the 2^N-ary tree scheme, algorithms have been developed to embed trees into hypercubes [46]. The binary tree approach has been found to be one of the simplest structures to embed [8]. To re-emphasize a point made earlier, the ease in using the binary tree topology to solve the current problem justifies its use.

Work at MIT resulted in the development of a massively parallel computer which consists of a SIMD array of 256K one-bit processors [45]. The Connection Machine uses two communication networks, one linking nearest neighbours, and the second allowing for communication between any two arbitrary processors. This second network can then be used to generate a tree composed of processors. The size of the Machine makes its general availability restrictive. For example, it can support eight front-end computers, most of which are either VAX or Symbolics LISP Machines. 4096 microchips are used, each of which contain 16 processors. The processors are arranged as 12-dimensional hypercubes with 16 processors at each vertex. The Machine essentially functions by generating a

solution tree for the problem, and then dynamically pruning the tree through the broadcasting of necessary constraint conditions for the problem to all of the processors. One problem with such a scheme is that the entire tree be first mapped onto the array. New techniques have been developed for the Connection Machine which allows it to selectively grow specific regions of the tree, prune these subtrees, and then generate new levels [14]. Applications for which the Connection Machine has proven to be successful in generating solutions include VLSI circuit simulations, machine learning, modeling of fluid dynamics, pattern recognition, and image processing.

A third type of general purpose architecture which has been successful as a parallel computer is the DADO processor [43]. The current prototype consists of 1023 processors all connected in a complete binary tree. The DADO project builds on the work of Bentley and Kung [7]. Applications for which the DADO architecture has shown to be very efficient include logic programming, relational database, and pattern recognition.

Chapter 9 Conclusion

The original intent of this dissertation was to develop an architecture which utilized the inherent parallelism of the 2^N-ary tree representation. The primary application of this representation and architecture was computer graphics. As the research progressed, two additional architectures were developed, both of which were significantly different in structure. The nature of the representation and its operations are such that a general representation can be defined based on set theory. This new representation can be used in many different applications, such as image processing, computer animation, database processing, and general information systems.

In addition to the wide range of applicability for this representation, its basis in set theory allows for the use of many operations, most of which are derived from equivalent set theoretic functions. Another property of this representation is the manner in which components of an instance are defined as being disjoint subsets of the instance. It is this feature which allows the operations of the representation to be executed in parallel.

Once the 2^N-ary tree representation was defined as a transformation of the general set theoretic representation, it was possible to proceed with an analysis of the effectiveness of executing these operations in parallel. This parallel time step analysis was performed on the quadtree, but the reasoning used in the analysis could be extended to the general 2^N-ary tree case. As expected, the primary results of this analysis have shown that there is indeed an improvement in

performance if multiple processors are used. A significant difference in execution time is evident in just going from a uniprocessor to a dual processor environment. However, one interesting observation that the analysis presented was that after some particular value for P (the number of processors used), the difference in execution times is negligible. This value of P was dependent upon the size of the object being represented, and the operation being performed.

The first architecture which was developed for the representation made use of alternating rows of slave processors and interconnection networks. The basic idea was that the 2^N-ary tree would be mapped on top of this array of slaves and networks. The slave processors represented the nodes of the tree while the networks provided the links or edges between parent and child nodes. It was determined that if the number of rows of slaves exceeded the number of levels in the trees using the system, then the implementation was worth pursuing. However, if multiple mappings were needed, as in those cases where there are insufficient processor rows, then the processor and scheduling overhead necessary to accommodate these mappings far exceeds the benefits of the system. An alternative solution which avoided this multiple mapping made use of more complex slave processors that allowed actual subtrees of instances to be stored at each leaf slave. If primitive slaves were used, then only one node value would be stored in each processor.

In the case of the VLSI-based architecture, a binary tree topology was used in the processing element mapping. The use of the binary tree was justified on a number of points. The architecture was also shown to perform more effectively than the PE-ICN system for large values of N.

Our last system was designed using shared memory modules, a series of slave processors, and a master controller. This approach used the benefits of both current hardware and software developments. Unlike the previous two schemes which stored tree values within the processing elements themselves, this approach used the memory modules for tree storage. The processors simply processed the

tree values. This required that the slaves communicate with the master controller, which handled all of the scheduling duties of the system. The simulation that was developed for this architecture produced results which were consistent with those presented in the preliminary quadtree analysis. Specifically, there was considerable improvement in execution times for binary operations as the number of slaves increased. However, a levelling off in execution time was noticed after a critical number of slaves was reached.

Of these three architectures, the simplest to actually implement would be the latter. The modular nature of the system would permit the use of existing hardware such as microprocessors, and memory microchips. The interconnection network linking the slaves to the shared memory could be constructed from simple 2x2 crossbar switches. The alternating row scheme, although modular in nature, is more complex with the added controllers, and multiple interconnection networks. The VLSI array would require that the actual binary tree be laid out onto the chip. The most likely method would require the use of laser technology to set the appropriate switches.

The Discussion closed with brief descriptions of three powerful general-purpose architectures that permit mappings of trees. The considerable size of these architectures from the standpoint of processing elements indicates that issues such as communication difficulties, and memory contention have been successfully addressed. A representation such as the 2^N-ary tree could be effectively implemented on any of these structures. The multidimensional arrangement of the Connection Machine's hypercube and point-to-point communication facilities makes the actual mapping of a 2^N-ary tree more possible than with the DADO machine, with its restricted topology.

To conclude, it is necessary to re-iterate a number of points which have been explicitly stated or at least implied as the dissertation progressed. The approach to be taken in the implementation of the representation must be one which is both cost effective and sufficiently fast to satisfy typical user requests. If the situation is

such that a fast uniprocessor can fulfill the needs of the application, then it should be considered as a candidate solution. The technology, facilities, and components necessary for the system must be available. Compromise is a term which contributes significantly to the selection process in system design.

Bibliography

[1] D. J. Abel. Some elemental operations on linear quadtrees for geographic information systems. *Computer Journal*, 29(1):73–77, 1985.

[2] M. J. Atallah and S. R. Kosarajo. A generalized dictionary machine for VLSI. *IEEE Transactions on Computers*, C–34(2):151–155, February 1985.

[3] J. L. Baer. *Computer Systems Architecture.* Computer Science Press, Inc., Rockville, Maryland, 1980.

[4] K. E. Batcher. Bit-serial parallel processing systems. *IEEE Transactions on Computers*, C–31(5):377–384, May 1982.

[5] K. E. Batcher. Sorting networks and their applications. In *AFIPS Conf. Proc. SJCC*, pages 307–314, Thomson Books, Washington D. C., 1968.

[6] V. E. Beneš. *Mathematical theory of connecting networks and telephone traffic.* Academic Press, New York, 1965.

[7] J. L. Bentley and H. T. Kung. *A Tree Machine for Searching Problems.* Technical Report CMU–CS–79–142, Carnegie-Mellon University, 1979.

[8] S. R. Deshpande and R. M. Jenevein. Scalability of a binary tree on a hypercube. In *Proceedings Int. Conf. Parallel Processing*, pages 661–668, 1986.

[9] D. Dias and J. R. Jump. Analysis and simulation of buffered data networks. *IEEE Transactions on Computers*, C-30(4):331–346, April 1981.

[10] L. J. Doctor and John G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, 1(7):29–38, July 1981.

[11] C. R. Dyer, A. Rosenfeld, and H. Samet. Region representation: boundary codes from quadtrees. *Communications of the Association for Computing*, 23(3):171–179, March 1980.

[12] P. H. Enslow. Multiprocessor organization – A survey. *Computing Surveys*, 9(1):103–130, March 1977.

[13] A. L. Fisher. Dictionary machines with a small number of processors. In *11th Annual International Symposium on Computer Architecture*, pages 151–156, June 1984.

[14] J. G. Harris A. M. Flynn. Object recognition using the connection machine's router. *IEEE Computer Vision and Pattern Recognition*, 5:134–145, 1986.

[15] I. Gargantini. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20:365–374, 1982.

[16] D. Gordon, I. Koren, and G. M. Silberman. Embedding tree structures in VLSI hexagonal arrays. *IEEE Transactions on Computers*, C–33(1):104–107, January 1984.

[17] A. C. Hartmann. Software or silicon? The designer's option. In *Proc. IEEE*, pages 861–874, June 1986.

[18] E. Horowitz and A. Zorat. The binary tree as an interconnection network: applications to multiprocessor systems and VLSI. *IEEE Transactions on Computers*, C–30(4):247–253, April 1981.

[19] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–1(2):145–153, April 1979.

[20] C. L. Jackins and S. L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14:249–270, 80.

[21] T. Lang, M. Valero, and M. A. Fiol. Reduction of connections for multibus organization. *IEEE Transactions on Computers*, C-32(8):707–716, August 1983.

[22] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):173–183, December 1975.

[23] A. Levy. *Basic Set Theory*. Springer-Verlag, Berlin, Heidelberg, New York, 1979.

[24] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Maine, 1980.

[25] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

[26] T. N. Mudge. Vision algorithms for hypercube machines. *IEEE Computer Architectures for Pattern Analysis and Image Database Management*, 3:225–230, 1985.

[27] G. Nagy and S. Wagle. Geographic data processing. *Computing Surveys*, 11(2):139–181, June 1979.

[28] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt. Efficient VLSI networks for parallel processing based on orthogonal trees. *IEEE Transactions on Computers*, C–32(6):569–581, June 1983.

[29] L. M. Ni and C. E. Wu. Design trade-offs for process scheduling in tightly coupled multiprocessor systems. In *Proceedings IEEE 1985 Intl. Conf on Parallel Processing*, pages 63–70, 1985.

[30] T. A. Ottmann, A. L. Rosenberg, and L. J. Stockmeyer. A dictionary machine (for VLSI). *IEEE Transactions on Computers*, C–31(9):892–897, September 1982.

[31] A. A. G. Requicha. Representation for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[32] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications ACM*, 29(12):1202–1212, December 1986.

[33] A. Rosenfeld. Tree structures for region representation. *Map Data Processing*, 137–150, 1980.

[34] H. Samet. An algorithm for converting rasters to quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–3(1):93–95, January 1981.

[35] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):223–260, June 1984.

[36] H. Samet. Region representation: Quadtrees from binary arrays. *Computer Graphics and Image Processing*, 13:88–93, 1980.

[37] H. Samet. Region representation: Quadtrees from boundary codes. *Communications of the Association for Computing*, 23(3):163–170, March 1980.

[38] C. L. Seitz. The Cosmic Cube. *Communications of the Association for Computing*, 28(1):22–33, January 1985.

[39] M. Shneier. Calculations of geometric properties using quadtrees. *Computer Graphics and Image Processing*, 16:296–302, 1981.

[40] D. P. Siewiorek, C. G. Bell, and A. Newell. *Computer structures: Principles and examples*. McGraw-Hill Book Company, New York, New York, 1982.

[41] K. O. Siomalas and B. A. Bowen. Performance of crossbar multiprocess systems. *IEEE Transactions on Computers*, C–32(8):689–695, August 1983.

[42] S. N. Srihari. Representation of three-dimensional digital images. *ACM Computing Surveys*, 13(4):399–424, December 1981.

[43] S. J. Stolfo. Initial performance of the DADO2 prototype. *IEEE Computer*, 75–83, January 1987.

[44] J. K. Udupa, S. N. Srihari, and G. T. Herman. Boundary detection in multidimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–4:41–50, 1982.

[45] D. L. Waltz. Applications of the Connection machine. *IEEE Computer*, 85–97, January 1987.

[46] A. Y. Wu. Embedding of tree networks into hypercubes. *J. Distributed Computing*, 2:238–249, 1985.

[47] C. Wu and T. Feng. On a class of multistage interconnection networks. *IEEE Transactions on Computers*, C–29(8):108–116, August 1980.

[48] W. A. Wulf and G. C. Bell. C.mmp – A multi-miniprocessor. In *AFIPS Conf. Proc. FJCC*, pages 765–777, AFIPS Press, Montvale, N. J., 1972.

[49] M. Yau and S. N. Srihari. A hierarchical data structure for multidimensional digital images. *Communications of the Association for Computing*, 26(7):504–515, July 1983.

[50] H. Y. Youn and A. D. Singh. *On Area Efficient and Fault Tolerant Tree Embedding in VLSI.* Technical Report CS–87–151, University of Massachusetts, 1987.