SUPERQUADRIC DESCRIPTION ON LARGE ARRAYS OF BIT-SERIAL PROCESSORS

By

RONALD ELLISON DANIEL JR.

Bachelor of Science in Electrical Engineering

Oklahoma State University

Stillwater, Oklahoma

1985

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
July, 1987

SUPERQUADRIC DESCRIPTION ON LARGE ARRAYS OF BIT-SERIAL PROCESSORS

Thesis Approved:

Keith A. Teague
Thesis Advisor

Richard L. Cummins

C. M. Bacon

Norman N. Durham
Dean of the Graduate College

1282684

PREFACE

This study describes the parallel implementation of a new computer vision technique, superquadric description. The use of superquadric primitives to extend the power of Constructive Solid Geometry for Computer Aided Design purposes was first proposed in [BARR 84]. The application of this technique for machine vision purposes was first published by Alex Pentland in [PENTL 86b].

This study developed a parallel least-squares solution technique to solve a slightly modified form of the regression equations originally derived in [PENTL 86b]. This technique is intended for execution on large arrays of bit-serial processors. Several ways have been suggested to interconnect the processing elements in such arrays, therefore the performance of this technique was estimated for three interconnection networks.

An effort such as this study is not possible without help. I would like to thank Dr. Alex P. Pentland, Dr. Steven L. Tanimoto, Dr. Steven P. Levitan, and Dr. Charles C. Weems for their courteous responses to my requests for information. I would also like to thank Dr. Dave Ballew, of OSU and AT&T Technologies, and Dr. Ron Rhoten, of OSU, for their help with particular portions of this study.

The members of my advisory committee, Dr. Charles M. Bacon, Dr. Richard L. Cummins, and Dr. Keith A. Teague deserve special mention for their guidance, enthusiasm, and friendship during this project as well as during previous semesters. The School of Electrical and Computer Enginering also has my deep appreciation for the constant employment which made this study possible.

I would also like to thank my parents for their love and support through all of the trying times children inflicted upon them. Honorable mention in this area goes to my brother and sisters. Most especially, I want to thank my fiance Laura for her patience with me throughout this period of insanity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Purpose and Motivation

This study has two purposes. The first is to investigate the parallel implementation of a new computer vision technique, superquadric description [PENTL 86b]. This technique is described more fully in Chapter Two, but the basic idea is to model imaged objects in terms of Boolean combinations of easily recognized solid primitives. This CSG (Constructive Solid Geometry) approach to object modeling is a standard CAD/CAM (Computer Aided Design / Computer Aided Manufacturing) technique. The use of CSG in machine vision was first investigated by Marr and Nishihara in [MARR 78]. Introductory information about this problem may be found in [BALLA 82]. Superquadrics are a recently suggested enhancement for CSG based systems. They provide a very general three dimensional modeling primitive which includes the sphere, cube, and cylinder as special cases. Figure 1, reproduced from [PENTL 86b], illustrates some of the possible superquadric forms. Almost any physical object can be represented as a CSG form, but the representation may become arbitrarily large. This problem can be reduced if we allow the superquadrics to be deformed as in Figure 1b [PENTL 86b, BARR 84].

Image processing is computationally intensive. Many specialized architectures have been suggested to handle this burden at higher speed than a serial processor is capable of. Since parallel implementations of a particular procedure will be highly dependent upon the architecture they are designed for, the second purpose of this study is to examine how the choice of architecture affects the execution speed of this task. Three specialized image processing architectures are compared for speed of executing the superquadric recovery algorithm. Some aspects of their physical construction are also examined. Image processing architectures can be divided into several major classes. Some of the metrics used to divide the machines into classes are the granularity of the processing elements, the interconnection network used, and the type of parallelism, i.e. SIMD or MIMD (Single Instruction stream Multiple Data streams or Multiple Instruction streams Multiple Data streams). All of the architectures considered in this study are members of what I will call the highly parallel class. This term will be used to identify machines that have at least one bit-serial processing element (PE) for each pixel in the image. A standard image resolution is 512 pixels by 512 pixels. Machines for this size image would have more than a quarter of a million PEs! The massive parallelism of these designs allows very high performance on a broad range of image processing algorithms. These are very fine-grained SIMD machines. Three interconnection networks are studied.

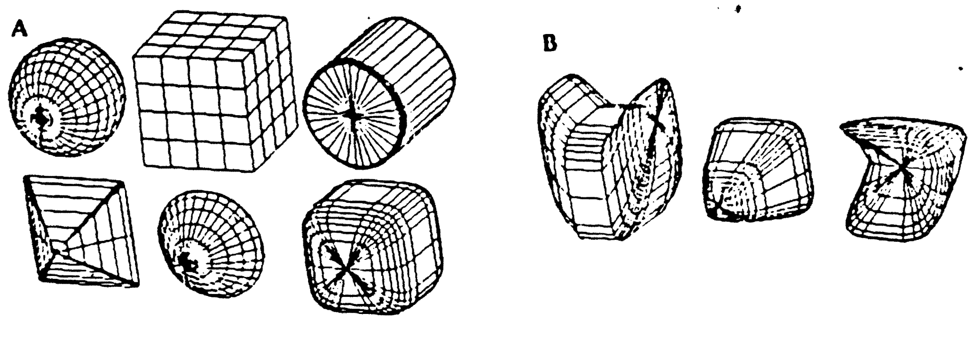


Figure 1: Unit and Deformed Superquadrics

This study is motivated by the limitations of current computer vision systems. A typical task for a current industrial vision system is to determine the exact location and orientation of parts whose location is limited to the width of a conveyor belt. Contrast this with the general purpose vision needed by a household robot! Bridging this huge gap is several years away. Both hardware and software are responsible for the limitations in current vision systems. Computer vision algorithms have not been developed that approach the generality of human vision. Even the limited routines that have been developed are computationally intensive, requiring many operations to be performed for each pixel in the image. The sheer amount of data compounds this problem. For example, a common resolution is an image of 512 x 512 pixels. The typical refresh rate is 30 frames per second. If we assume that the processing for each pixel will be the equivalent of 1000 floating point multiplies [REDDY 78], our machine must execute the equivalent of 7.8 billion multiplies per second! There is a great deal of continuity in visual input, the later stages of a vision system could take advantage of this continuity to reduce the workload. However, these later stages will be dependent upon a great deal of preprocessing of the image. This preprocessing stage cannot be skipped and will have the majority of the arithmetic operations.

Computers capable of one billion multiplies per second have only recently become available, at a cost of millions of dollars. In order for affordable hardware to complete a vision task in an acceptable time, the software to perform the task must be made very simple. Typically, the environment and/or the task are restricted in order to simplify the problem. The vision software is then optimized for these restrictions. Unfortunately, the very restrictions which make the problem solvable in an acceptable time make the solution technique inapplicable for tasks with slightly different requirements. The non-portability of vision software is considered a major problem [REDDY 78].

The ultimate in portable software would be a system with the capabilities of human vision. While this goal is still far in the future, progress is being made. As can be expected, the powerful systems being researched impose a tremendous computational burden. A great deal of work has gone into the development of architectures which are capable of shouldering this burden. A wide variety of architectures have been proposed, with different trade-offs between the factors of price, performance, and programming ease. Chapter Two provides a brief review of some of the major architectures proposed. This study compares three similar architectures, the main difference between them being their interconnection networks. In addition to estimating execution times, we will also examine some of the characteristics of their VLSI and system-level fabrication.

The next section of this chapter sketches the outline of the procedure used to accomplish these purposes, while the subsequent section notes the limitations of this study. Chapter Two reviews the literature, while details of the procedure are in Chapter Three. Results are given in Chapter Four, and conclusions are offered in Chapter Five. The algorithms used for the superquadric estimation are presented in Appendix B. These algorithms are composed of call to microcoded subroutines which perform arithmetic and data transfer operations. The algorithms for these fundamental operations are given in Appendix A, along with the format of the pseudocode used to specify all the algorithms.

Overview of Procedure

Selection of Architectures

As noted above, many parallel architectures have been suggested for use in image processing. Some of these architectures are surveyed in Chapter Two. For reasons described more fully there, three architectures were chosen for comparison. The first architecture is the Content Addressable Array Parallel Processor (CAAPP), designed at the

University of Massachusetts at Amherst [LEVIT 84, LAWTO 84, WEEMS 85]. This machine has recently undergone some major design changes [WEEMS 87]. We will examine the older version, circa 1984. The second is the pyramid machine [TANIM 84, DYER 81]. These two machines were chosen because of the differing approaches they take in trading off speed for VLSI area. The other architecture examined provides an intermediate position in terms of complexity and expected performance. This final machine is the simple Four Nearest Neighbor (4NN) mesh. All of these are massively parallel machines with at least one PE for each pixel. In order to keep the comparisons of the architectures fair, all machines are assumed to have equivalent PEs, the only difference being in the connections with neighboring PEs. The PE used is described in detail in the third section of Chapter Two. The interconnection networks are described in Chapter Two, Section 4, while their implementation is considered in Chapter Two, Section 5.

Design of the Superquadric Recovery Software

Once the architectures were selected, the system to recover the superquadric parameters was designed. Pentland, in [PENTL 86b], derives linear regressions to solve for estimates of the superquadric parameters given estimates of the surface normals of the imaged objects as well as their derivatives along the x and y image axes. This study presents a parallel least-squares solution technique, and estimates its execution time. Procedures to obtain the estimates of the surface normals have been described in the literature [LEE 85, PENTL 84, 86a]. A technique to calculate the derivatives without excessive noise amplification is described in [HARAL 81,82].

Performance Estimation

Execution speed was estimated by breaking the least-squares procedure into major stages, such as matrix inversions and multiplies. The major stages were then decomposed into fundamental operations, such as data movements and arithmetic operations. Formulas for the execution time of the major stages of the least-squares procedure were then derived as functions of the execution time for the fundamental operations. The formulas for the execution time of the least-squares solution are given in Appendix B, along with the parallel algorithms. The times for the fundamental operations were also estimated. These formulas and algorithms are given in Appendix A.

Limitations

Superquadrics are not a cure-all for computer vision. They cannot explain many important perceptual phenomena, such as character recognition or the fractal branching structures of trees. However, they are a much more general modeling primitive than those currently used by CSG based vision systems. As will be seen in Chapter Two, they do appear to satisfy some of the needs of the higher levels of vision processing.

By allowing deformations such as shearing and twisting (Figure 1b), superquadrics become a much more powerful modeling primitive. The principal limitation of superquadric description is that currently, only undeformed superquadrics can be recovered. To my knowledge, the regression equations to recover deformed superquadrics have not been derived. Another problem is that obtaining a CSG representation of imaged objects using superquadric primitives has not been investigated. Previous work [MARR 78] has been done in obtaining CSG descriptions using cylindrical primitives. The use of superquadrics should be a straightforward extension.

A limitation of this study is that the algorithms were not implemented, so no characterization of the errors of the recovery procedure is available. Another limitation is that only one class of parallel architectures was examined. Within this class, the machines studied span a broad range of the speed vs. area trade-off space, but this work could be extended to examine other types of parallel architectures. I hope to address all these limitations in future studies.

No claims are made for optimality of the least-squares algorithms used. They are simple, straightforward implementations. Some possible ways to improve the performance of the algorithms are suggested in Chapter Five. While not optimal, neither are these procedures grossly inefficient. The matrix operations are quite conservative in the number of multiply and divide instructions issued. A special version of the least-squares algorithm was used for the Pyramid machine, which resulted in a marked improvement in speed, at the cost of a loss in the resolution of the solution. If the loss of resolution is unacceptable, the Pyramid can implement the same algorithms as the other two machines, with comparable performance.

CHAPTER II

REVIEW OF LITERATURE

The application of superquadrics to machine vision is quite new, very little has been published at this time. Much more has been published on specialized architectures for image processing. The purpose of this study is not to survey the literature in either area, however, some background material is appropriate in order to gain perspective on the role of superquadrics in machine vision. This chapter is quite brief, hitting only the major highlights. The interested reader is referred to the references for more complete information. The first section of this chapter provides a brief overview of computer vision. This is followed by sections on superquadrics and surface normals estimation. Subsequent sections deal with the hardware side of the study. The PE used by all the machines is described in section three. Section four discusses the interconnection networks examined in this study, while section five deals with some of the considerations of the actual physical implementation of these networks.

Computer Vision

Computer vision is a fast growing area of research and development. The end goal of this research is to create a machine vision system embodying the power and generality of human vision. Since much of the power of human vision is intimately tied to our learning, planning, and reasoning abilities, computer vision is considered to be a subset of Artificial Intelligence (AI). The high level reasoning capabilities provided by AI routines must have an input, they do not pull solutions from thin air! What is the form of this input, and how is it obtained from an array of intensity values? This is the fundamental question of computer vision. The first part of the question is easy to answer in general terms; the input to high level cognitive processes should be a concise description of the scene in a form suited to the cognition mechanism(s). The second part of the question, obtaining the description, is much more difficult to answer. One thing is certain, the recovery of the description is a computationally intensive task. In Chapter One we estimated the processing power needed for real-time computer vision to be on the order of 10 Gigaflops. Given this estimate, the need for a highly specialized machine is clear.

This processing burden is typically described as having three levels: characterized as low, intermediate, and high level. High level vision refers to the AI capabilities mentioned above. It is concerned with the manipulation of abstract units such as 'tree' or 'cloud'. Low level vision is also known as image processing. This stage of vision computation operates on the image, and on image-like data structures. The intermediate level of vision is concerned with providing the interface between the two other stages. Superquadric description is on the border between the low and intermediate levels. The estimation of the parameters is low level, obtaining the CSG grouping is intermediate.

High Level Vision

The purpose of this study is not to compare the merits of various AI techniques for high level vision processing. However, before we can draw any conclusions about the suitability of superquadrics for a task we must know what the task is. Our task is to derive a description of the imaged scene in a format convenient for a high level reasoning mechanism. This format will be constrained by the needs of the high level mechanisms. The next few paragraphs attempt to show some of these needs. The format will also be constrained by what is possible to compute from the image. The characteristics of low level image processing algorithms will be examined in a subsequent sub-section.

The first restriction on the format is that it be amenable to processing by a variety of cognitive mechanisms. Learning, planning, belief maintenance, as well as formal and common sense reasoning will all be needed in a sophisticated vision application. Many of the mechanisms that have been developed so far are based on a linguistic analogy [SOWA 84]. The goal is to obtain and manipulate the semantic content of syntactic structures. For the case of vision, the imaged objects and their visual attributes can be viewed as forming the nouns and adjectives of a visual language. The role of verbs and adverbs is filled by relations, both spatial and temporal, between the objects. The structure of the grammar is imposed by the laws of physics. Seen in this light, the desired 'chunks' input to the cognition mechanisms would need to be at the complexity of words and phrases, or simple parsed structures.

While the main idea of this visual language would seem very attractive, we must be careful about carrying this analogy too far. Most linguistic structures cannot compactly describe natural objects, their attributes, and their physical relations. An additional problem with most linguistic structures is that they are not isomorphic to the physical world, that is, they do not unambiguously describe real scenes. It has been shown that many of the reasoning capabilities of humans cannot be explained unless we use an isomorphic representation [FISCH 78]. CAD/CAM data structures are an example of an isomorphic representation. They unambiguously describe a physical object.

Once we have the input data, how do we process it in order to extract it's meaning? The reasoning mechanisms would certainly find it much easier to work with more abstract entities, such as 'human' or 'tree', rather than a raw description of the parts in an image and their spatio-temporal relations. This leads us to the requirement that the format of the description be easy to match with known objects. This is where the need for a concise description comes into play. It would also be nice if the format of the raw description matched the format of the more abstract description. Having the representation be able to span a range of abstractions would allow the high level mechanisms to operate directly on the unmatched description if it became desirable to do so.

Conceptual structures [SOWA 84], also known as relational structures or semantic nets, are one representation that has been suggested for the high levels of vision processing [BALLA 82, CHARN 86]. This representation offers many advantages for AI and vision. Reasoning can be done at varying levels of abstraction, matching is relatively easy, the structures map easily into linguistic forms, and they offer powerful abilities to model physical objects. CSG data structures are a natural extension of relational structures, so we can have an isomorphic representation. Therefore they meet the requirements set forth by the previous paragraphs. An introduction to the use of relational structures for vision is given in [BALLA 82] and [CHARN 86], where additional references may be found.

We will assume that the inputs to the high level section of the vision system should be at least a partial CSG description of the objects in the scene. The primitive solids used in this CSG model must now be selected. The basic requirement for these solids is that they be computable from the outputs of the image processing stage. An additional constraint is that the number of different primitives used be small and that their expression be compact.

Superquadrics

Superquadrics [BARR 81, PENTL 86b] appear to meet both these requirements. Unit superquadrics are a two parameter family of solids which include spheres, cylinders, and cubes as special cases (Figure 1a). The surface position vector $\underline{X}$ and surface normal vector $\underline{N}$, as functions of latitude η and longitude ω are given by

$$\underline{X}(\eta,\omega) = \begin{pmatrix} a_1\, C_\eta^{e1}\, C_\omega^{e2} \\ a_2\, C_\eta^{e1}\, S_\omega^{e2} \\ a_3\, S_\eta^{e1} \end{pmatrix} \qquad \underline{N}(\eta,\omega) = \begin{pmatrix} 1/a_1\, C_\eta^{2-e1}\, C_\omega^{2-e2} \\ 1/a_2\, C_\eta^{2-e1}\, S_\omega^{2-e2} \\ 1/a_3\, S_\eta^{2-e1} \end{pmatrix} \qquad (1,2)$$

where $C_\eta = \cos(\eta)$, $S_\omega = \sin(\omega)$; a_1, a_2, a_3 are scaling factors for x,y, and z; and the exponents e_1 and e_2 control the roundness or squareness in the latitudinal and longitudinal directions. By allowing deformations such as scaling, shearing, tapering and twisting, a very large set of shapes can be obtained (Figure 1b). The constants a_1, a_2, and a_3 implement scaling, which is a simple example of a deformation. Even with the more complex deformations, only a few parameters are needed to describe any of the possible shapes [BARR 84, PENTL 86b], therefore the requirements of a small number of primitives which have a compact representation is easily met. The parameters controlling the superquadric shapes are computable from a map of the surface normals of the imaged objects. The techniques to do so are still quite new, and further work is needed. However, the technique appeared promising enough that an investigation of its parallel execution seemed in order.

Comments on the Estimation of Surface Normals

Before the superquadric parameters can be estimated, knowledge of the surface normals in the image, as well as their derivatives along the x and y axes, is required. Several techniques to recover this information have been proposed in the literature. The output of these procedures is an estimate of the x, y, and z components of the surface normal of an imaged object in the area covered by a pixel. These components are aligned with the x, y, and z axes of the image plane. This is illustrated below in Figure 2. Frequently these estimates are expressed as the tilt and the slant. The tilt is the image plane component of the normal vector. In other terms, it is the polar combination of the x and y components. The slant is the perpendicular, or z, component. Both tilt and slant are usually expressed as angles.

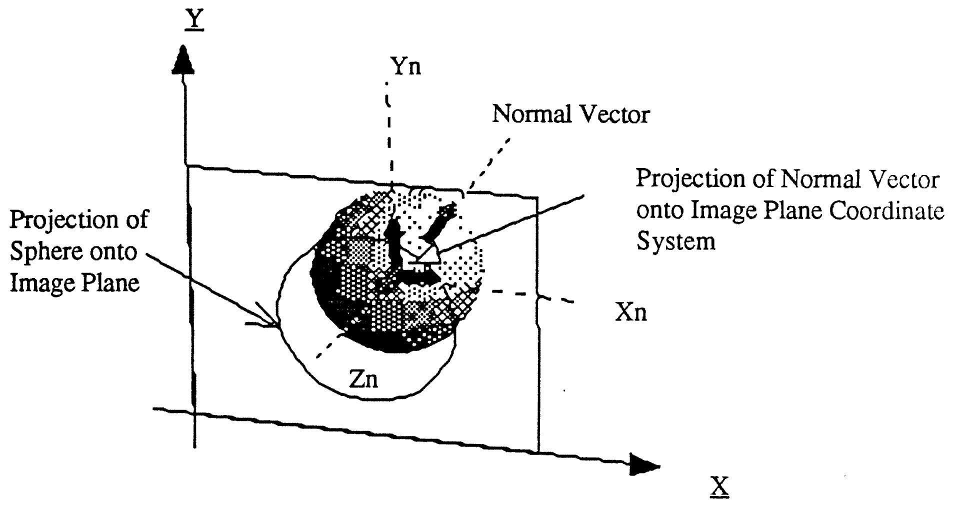

Y
Yn
Normal Vector
Projection of Normal Vector onto Image Plane Coordinate System
Projection of Sphere onto Image Plane
Xn
Zn
X

Figure 2: Projection of Normal Vector onto Image Coordinates

Early work in the area concentrated on obtaining shape information from a single type of information, such as shading [HORN 75] or texture [WITKI 81]. More recently, Pentland developed a method that uses both texture and shading information [PENTL 84, 86a]. Extending his technique to include motion was covered in [FERRI 85]. A more accurate estimator is described in [LEE 85]. This estimator uses a coordinate system that has the slant oriented with the direction of the main illuminant, rather than being perpendicular to the image plane.

All of these estimates are subject to error. Quantitative description of the error is difficult, but it can be qualitatively described as a bias which is a function of the actual values of the surface normals, with an additive noise component. The shape of the imaged object strongly influences the results. The error is quite small for a spherical object, but becomes greater as the object shape becomes more ellipsoidal. Tables I and II are reproduced from [LEE 85]. Table I gives the absolute value of the error in estimates of tilt and slant for Lee's estimator applied to an artificially generated sphere. Table II gives the absolute value of the error of Lee's technique on an artificially generated ellipsoid with a 3:1 ratio of major axis length to minor axis length. The top row of each of the tables gives the eight tilts being tested. The first column gives the nine slants tested. Each element of the table has two entries, the first is the error in slant and the second is the error in tilt. As can be seen, the performance degrades as the assumption of equal curvatures is violated.

TABLE I

ACCURACY OF LEE'S ESTIMATOR FOR SPHERE

	Tilt															
Slant	0		22		44		66		88		110		132		154	
0	6	0	6	*	6	*	6	*	6	*	6	*	6	*	6	*
11	1	0	3	0	3	0	3	0	1	0	3	1	3	1	3	1
22	2	0	2	1	2	0	2	1	2	0	2	2	2	1	2	0
33	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1
44	2	0	2	0	2	0	2	0	2	0	2	1	2	1	2	1
56	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1
67	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1
78	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1
89	7	0	7	0	7	0	7	0	7	0	7	1	7	1	7	1

TABLE II

ACCURACY OF LEE'S ESTIMATOR FOR ELLIPSOID

	Tilt															
Slant	0		22		44		66		88		110		132		154	
0	0	*	0	*	0	*	0	*	0	*	0	*	0	*	0	*
11	11	0	11	16	11	18	11	10	11	0	11	9	11	16	11	15
22	22	0	22	16	22	17	22	11	22	0	22	9	22	16	22	15
33	1	0	3	16	11	18	19	10	25	0	19	9	11	16	3	15
44	8	0	6	16	2	18	10	10	14	0	10	9	2	16	6	15
56	16	0	12	16	2	18	8	10	11	0	8	9	2	16	12	15
67	15	0	15	16	5	17	5	10	9	0	5	9	5	16	15	15
78	4	0	4	15	4	17	4	10	8	0	4	9	4	16	4	15
89	7	0	7	15	7	17	7	10	7	0	7	9	7	16	7	14

In addition to the normals information, we must know the derivatives of the normals along the x and y axes. Since the normals information is subject to noise, the calculation of the derivatives must be done with care. The facet model, [HARAL 81,82] seems to offer an approach that will allow the derivatives to be calculated without excessive problems due to the noise in the normals data. This technique assumes that in each K by K neighborhood of the image, the pixel values can be expressed as a polynomial function f of the row and column coordinates. The coefficients of the polynomial are obtained from linear combinations of the values in the KxK neighborhood. Knowing the coefficients of this polynomial, we can easily calculate the derivatives of interest. The convolution masks used to estimate the polynomial coefficients and the derivatives will depend on the size of the neighborhood, the order of the fit, and the basis functions of the polynomial. For the normals information the calculation would be performed three times, once for each of the components of the normal. Convolution on the architectures considered is quite fast, the execution time being proportional to the area of the mask rather than the image size. The convolution procedure is described later in this chapter.

Description of the Processing Element

The PE used in this study is essentially the one proposed for use in the CAAPP [LEVIT 84, LAWTO 85, WEEMS 85], circa 1984, with the number of inter-PE communication links being modified for whichever interconnection network is being studied. The memory was also extended from the original 128 to reflect the advances in fabrication technology. The PE is equipped with five register bits, 512 bits of local memory, a simple bit-serial ALU, and communication links with its neighboring PEs. A content addressable broadcast and response mechanism is also provided. These components of the PE, as well as the array controller and microcode format, are described below. Figure 3 gives the functional layout of a PE.

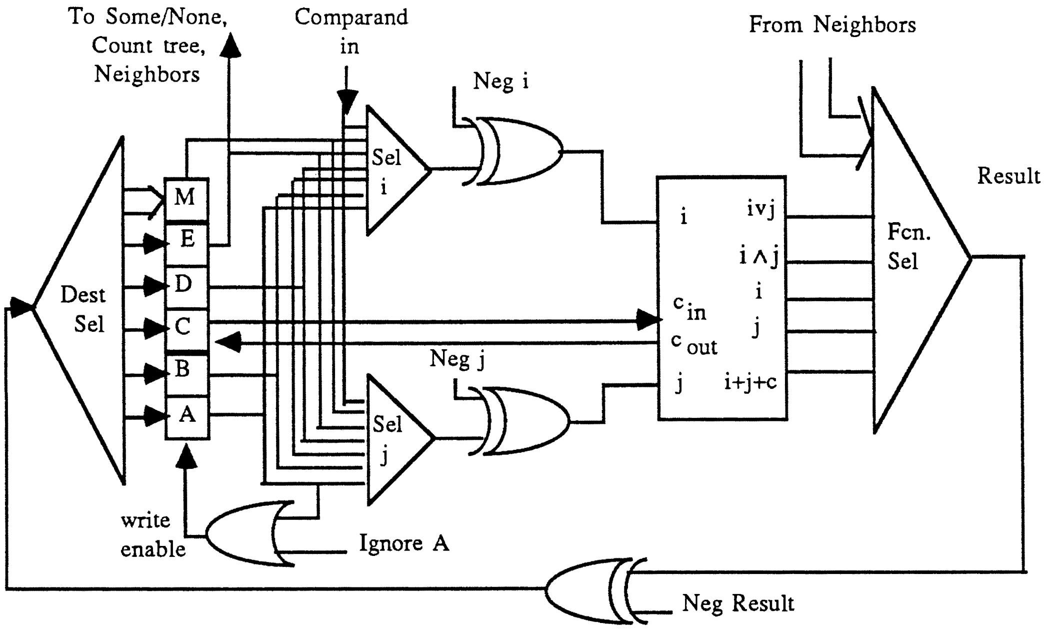


Figure 3: Functional Block Diagram of a Single PE

Registers and Memory

Each PE has five one-bit registers, denoted A through E. Registers A, C, and E have special purposes, while registers B and D are general purpose. Register A is also known as the activity bit. When this register is cleared, no results can be stored in registers or memory of the PE, effectively disabling the PE. The activity bit can be overridden by setting the Ignore Activity (IA) bit in a micro-instruction. The C register is used as the carry register in the addition instruction. The PE can perform a one-bit add with carry in one clock cycle. The E register is the response register of the content addressable mechanism. The use of these three registers is dealt with below in greater detail.

Each PE has 512 bits of local memory, denoted by the block labeled M in figure 3. Content Addressable Memory (CAM) cells are usually used in a machine which has a content addressable broadcast and response mechanism. For static memory, such cells involve three extra gates per bit of memory [WESTE 85]. However, the bit serial nature of the content addressable search mechanism for this PE, which performs the comparisons in the ALU instead of memory, allows us to use standard RAM cells instead of the larger CAM cells. Additional VLSI area could be saved by the use of dynamic memory cells, but the refresh cycle would hurt the speed of operation due to the width of the local memories. The timing estimates presented in the results section do not include the time needed for DRAM refresh.

ALU Operations and Control

The control inputs to the ALU select two sources, a function, and a destination. Sources may be registers, memory, or a value broadcast from the controller. The two sources are denoted as SRC i and SRC j. Destinations are either the registers or memory of the PE. Instructions that access memory can specify only one memory address per instruction. The functions that can be selected are listed in figure 4. They include logical operations, communication with neighboring PEs, and one bit addition with carry. The addition operation automatically uses and updates the carry bit, register C. All functions are computed in parallel and their results sent to the function select multiplexer. The function select field of the micro-instruction determines which of the results is placed on the result line for storage. Control lines are provided to allow the negation of any combination of the source and result bits.

Communication with neighboring PEs is accomplished with the E register and the function select block. Each PE continuously broadcasts the value in its E register to all PEs it can communicate with. When communication with neighboring PEs is desired, the function select lines of the destination PE choose which of the broadcast values will be placed on the result line. This value is then routed to its destination within the PE, either register or memory. The slow-down of the communication operations due to the high fanout of the E register is ignored in this study, under the assumption that it will not be a critical limitation on the clock frequency. The number of control lines needed for the function select multiplexer will depend on which interconnection network is being used. The pincounts for the various architectures are given in the last section of this chapter.

Content Addressable Broadcast and Response Mechanism

A notable feature of the PE is the content addressable broadcast and response mechanism. This mechanism allows the host or the controller to interrogate the PE array about characteristics of the image, avoiding the need for the image data to be dumped to the host for sequential evaluation. The controller broadcasts a one bit value to each PE on the Broadcast Comparand (BC) line. The source selected by the SELECT i multiplexer is compared with this broadcast value. If they are the same, a one is placed on the result line. This value will usually be stored in either the PE's response bit, register E, or the activity bit, register A. The response bits of all the cells are OR'ed together and made available to the controller as the SOME/NONE flag. At any time, the number of responding cells can be obtained by use of the COUNT RESPONDERS function. An adder tree is used to obtain the count. A description and timing analysis of the adder tree is given in [LEVIT 84]. The time to obtain the response count for a 512 x 512 image is 2.5 microseconds. Due to the organization of the adder tree, response count requests can be issued every 1.6 microseconds [LEVIT 84]. The algorithms developed in this study do not make use of either the SOME/NONE or COUNT RESPONDERS functions, therefore we will not consider them further.

It will frequently be useful to restrict processing to only those PEs with a particular bit pattern in some field of memory. The activity bit, register A, allows subsets of the PEs to be disabled. By storing the result of a comparison in the activity bit, all PEs that failed the comparison can be disabled. The simple algorithm below will leave enabled all cells that exactly meet some search criterion, leaving all others disabled.

ALGORITHM 1: SELECTIVE PROCESSING

```
SELECT all cells
for each bit in the pattern
    BROADCAST bit value.
    IF ACTIVE, COMPARE with memory location, store result in activity bit
next bit and memory location
```

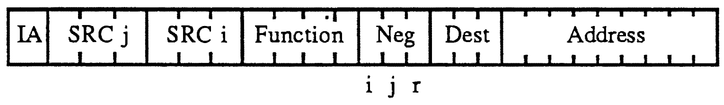
Note that this algorithm's execution time is proportional to the length of the pattern, not to the size of the image. This is a frequent occurrence in these architectures. If an exact match is not desired, Foster [FOSTE 76] gives algorithms for several other matching criteria such as minimum, maximum, greater than, less than, closest to, etc. The algorithms developed in this study frequently select PEs based on their position in the array. These comparisons require that the coordinates of the PE be stored in its local memory.

Microcoded Array Controller

The array controller is a microcoded sequencer which receives assembly language level instructions from the host and interprets them as a call to a microcode subroutine that the PEs can execute. The same micro-instruction is broadcast to all PEs at the same time, thus the architectures are of the Single Instruction stream / Multiple Data stream (SIMD) variety. As mentioned earlier, the PE will execute the instruction only if its activity bit is set, or if the micro-instruction's IA, Ignore Activity, bit is set.

The format for the portion of a micro-instruction broadcast to the PEs is given in Figure 4, along with the basic micro-operations that can be performed by the PE. Micro-instructions are of the form: select two sources, perform some operation on them, then store the result in some destination. The local memory is fast enough that a read-modifywrite takes a single clock cycle. Instructions which access the local memory must use the same address for all memory operands. In other words, local memory to memory transfers take two instruction cycles per bit and must proceed through an intervening register. Transferring a data value in the memory of one PE to the memory of another PE also takes two instructions, since the data must be moved from the memory of the first PE to its E register. This takes one instruction cycle. During the second instruction cycle the value of the E register is automatically broadcast to all neighboring PEs. The function select of the destination PE puts this value onto its result line and enables writing to the memory location whose address is specified in the second micro-instruction.

The microword will also have additional fields for use within the array controller to support looping, microcode subroutines, etc. The form of these additional fields will depend on the design of the array controller and will not be considered in this study.



IA	
0	Use Activity Bit
1	Ignore Activity Bit

Neg i,j,r	
0	True
1	Complement

Function	
0	i
1	j
2	CMP i,j (XNOR)
3	ADD i,j,C
4	NAND i,j
5	NOR i,j
6	Read N
7	Read S
8	Read E
9	Read W
10	Read NE*
11	Read NW*
12	Read SE*
13	Read SW*
14	Read Parent*
15	Read NE Child*
16	Read SE Child*
17	Read NW Child*
18	Read SW Child*

SRC i,j	
0	None
1	M
2	A
3	B
4	C
5	D
6	E
7	Comparand

Dest	
0	None
1	A,C (ADD,SUB)
2	A
3	B
4	C
5	D
6	E
7	M

* Only for Pyramid

Figure 4: Microinstruction Format

Parallel Image Processing Architectures

As noted above, many parallel architectures have been suggested for use in image processing. The majority of image processing algorithms apply the same operations to all pixels in an image, or a subset of the image. This makes them suitable for implementation on Single Instruction stream, Multiple Data stream (SIMD) machines. This is the simplest type of multiprocessor, where all PEs execute the same instruction at the same time. This greatly simplifies the programming task, especially for system software. Another important characteristic of image processing algorithms is that most calculations depend upon the values of a few neighboring pixels. This has important implications for the network interconnecting the processing elements.

Too many architectures have been suggested to give a complete survey here. However, the architectures proposed can be divided into several classes. SIMD vs. MIMD is a major distinction, as is the granularity of the processing ensemble. The interconnection network used to connect the PEs is important. Also important is the communication between the host and the parallel architecture. The majority of specialized image processing architectures that have been proposed are coarse-grained SIMD machines. A linear or mesh interconnection is frequently used, although hypercube and tree topologies are also used on occasion. Few have used content-addressable memory, due to its high cost compared to standard DRAM. The coarse grained approach is used to take advantage of the widely available microprocessors and specialized DSP chips. Many fine-grained architectures have been proposed on paper but few have been fabricated.

Of all the SIMD architectures which have been suggested for image processing, the class of highly parallel architectures offers perhaps the greatest potential for performance. These architectures have at least one PE for each pixel in the image. These PEs are usually bit serial ALUs with a small amount of local memory, typically less than 1024 bits, and connections to a small number of neighboring PEs. The massive parallelism of these fine-grained designs more than compensates for the slow speed of bit serial operation. By having one PE for each pixel in the image, the dependance on the size of the image in many image processing algorithms is removed. As an illustration, consider the problem of adding a constant to every pixel in an image. On a standard serial machine, ignoring the overhead of instruction fetches, the time to perform this function can be calculated from

$$N(t_{read} + t_{add} + t_{store}),$$

where N is the number of pixels in the image. For a large array of bit-serial processors the time can be calculated from

$$B(t_{read} + t_{add} + t_{store}),$$

where B is the number of bits to be added in each PE. If we assume that the standard processor can read a word, add the constant, and store the resulting word in the same time that the bit-serial processor can perform the same operations on a bit, then the array will work faster anytime there are more pixels in the image than bits in a word. If we use 16-bit words and the image size is 512 x 512, the array will be 16,384 times faster.

To provide another illustration of these machines, let us compute an approximate figure for floating point arithmetic speed. Assume that a 32-bit floating point multiply takes 2000 clock cycles. If the clock frequency is 10 MHz., each PE has a speed of 5 KFlops. This is quite slow compared to the performance of dedicated arithmetic coprocessors such as the Intel 80387, which can perform approximately 300 KFlops. However, the 5 KFlop figure was for one PE. There are 64 PEs on a chip, giving 320 KFlops per chip. For a 512x512 image, there are more than a quater million PEs. The floating point performance for the entire array is 1.25 GFlops!

Several factors go into determining the speed with which a parallel processor can execute a particular algorithm. Obviously, the speed of the PE is an important factor.

However, the fit between the interconnection network and the communication needs of the task to be performed is at least equally important. This latter factor is what we will be comparing in this study. In order to keep the speed and area comparisons of the interconnection networks meaningful, the three machines are considered to have equivalent PEs. The difference between the architectures in this study is the type of interconnection network used to communicate with neighboring PEs. The interconnection networks studied are described in the next section.

The interaction between the host and the processing array is handled by the array controller. The content addressable mechanism described earlier provides a powerful way to control the operations of the array, as well as obtain results without dumping the entire image for slow sequential evaluation by the host.

Interconnection Networks Studied

Three interconnection networks were chosen for comparison in this study. All use essentially the same processing element, the only differences being in the number of communication lines and their corresponding control lines. Each of these interconnection networks will be described below. In addition to the interconnections between the PEs, each interconnection network has communication links with the array controller. As mentioned earlier, each of these architectures is of the SIMD type. All the instructions are broadcast from a microcoded central controller. The central controller can also broadcast data to all the PEs. This data broadcast is done bit-serially. As well as being able to send data to the array of PEs, the controller can receive data from the array by means of a content addressable response mechanism. The content addressable mechanism was discussed in a previous section.

Four Nearest-Neighbor Mesh

The first interconnection network is the standard four nearest neighbor (4NN) mesh. In this interconnection network the PEs are arranged in a two dimensional mesh, one PE for each pixel in the image. Each PE that is not on an edge of the mesh has communication links with its four nearest neighboring PEs, as illustrated below in Figure 5. Communication for those PEs that are on the edge of the mesh can be handled in several fashions, as shown in Figure 6.

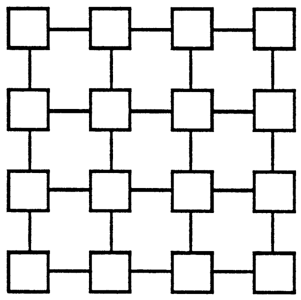


Figure 5: Four Nearest Neighbor Interconnection Network

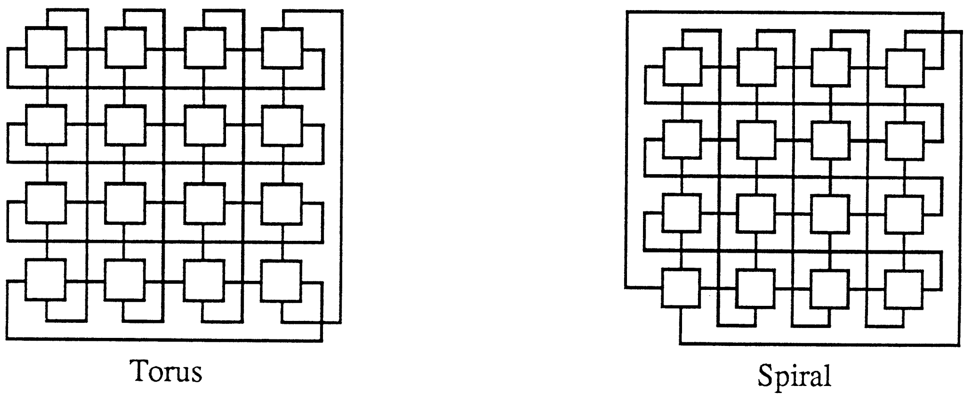


Figure 6: Sample Edge Treatments

This interconnection network has recieved a great deal of attention for image processing purposes, due to direct mapping of pixels onto PEs. The 4NN mesh can execute many of the algorithms used in the lower levels of vision. As an illustration, consider the task of convolving the image with a three by three mask. We will denote the coefficients of this mask as a_{11}, a_{12}, ... , a_{33}, and will ignore edge effects. The first step is to broadcast a_{11} to all the PEs. They will multiply a_{11} by the image value they contain, then move the product to their southern neighbor. The value of a_{21} will then be broadcast and multiplied by the image value, then added to the product from the previous step. We will continue the broadcast and accumulate process in a spiral fashion. At the last step, the value of a_{22} is broadcast, multiplied by the image value, then added to the accumulated value read from the PE's northern neighbor. The data movements of this algorithm are shown in Figure 7. The important thing to note about this algorithm is that the execution time will be proportional to the area of the mask and the numeric precision of the operands, not the area of the image. [LEVIT 84] gives sample times for convolution with varying size masks. Assuming 8-bit pixels, the time for the 3x3 mask was 0.7 msec., the time for the 7x7 mask was 4.0 msec. These times are independent of the size of the image. Multiplies are quite slow on this machine, since their execution time is proportional to the square of the number of bits multiplied. If the convolution mask is symmetric, the time for the convolution can be reduced. Keep in mind that these times are for the CAAPP. As will be seen in the next section, the CAAPP is not as fast as the 4NN. The times for the 4NN would be roughly half those for the CAAPP.

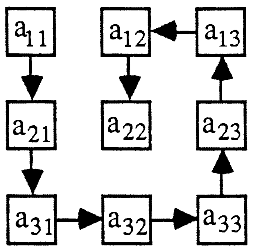


Figure 7: Data Movements for Convolution

Content Addressable Array Parallel Processor

The second interconnection network is the one proposed for use in the Content Addressable Array Parallel Processor (CAAPP), circa 1984. This architecture has recently undergone major design changes, one of which was to use a full 4NN mesh. The original version uses an interconnection network which is organized as a mesh of submeshes. Referring to Figure 8, we see that each four by four submesh has the standard mesh connection, with the edges being treated as a double spiral. However, communication between the submeshes is restricted to one bidirectional link between neighboring submeshes. When data must cross a submesh boundary this will increase communication time by a factor of four, plus overhead, compared to the 4NN mesh. The advantage of this submesh organization is that the pincount of the VLSI devices is reduced considerably. For example, assume each package has 64 PEs organized as an eight by eight portion of the array, only needing communication links to neighboring packages. For the case of the 4NN mesh, each PE on the perimeter needs a communication link off-chip, and the corner PEs need two. This gives a count of 32 pins needed simply for inter-chip communication. For the CAAPP, this is reduced to eight. The savings in pin count is very important when the board level pin count is calculated. Assuming each board has 64 chips arranged in an eight by eight array, the number of pins needed to communicate with other boards in the system is reduced from 256 to 64. Since connections are a major reliability problem, this savings is quite important.

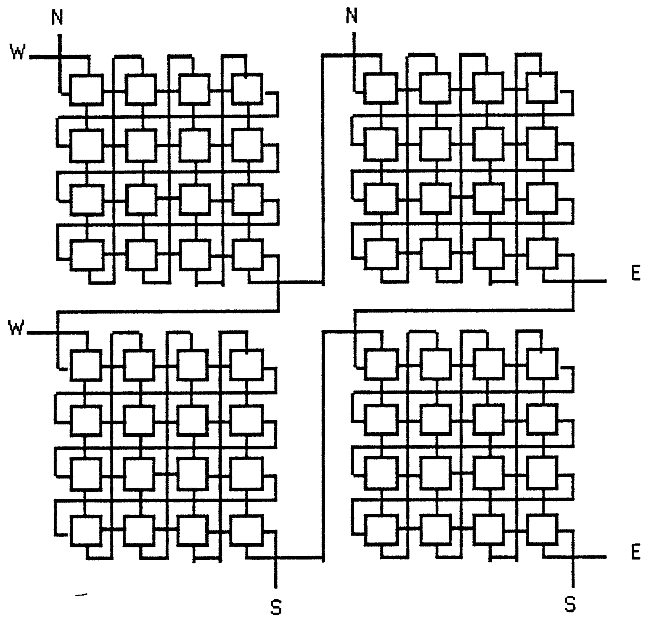


Figure 8: Interconnection Network for One Chip of the CAAPP

The Pyramid

The final architecture studied is the pyramid [TANIM 84, DYER 81]. This architecture uses more PEs than the others, as well as a more complex interconnection network. The interconnection network used is illustrated below in Figures 9a and 9b. In this scheme, the processing array is organized into L layers of 8NN meshes. Each mesh has four times as many PEs as the mesh above it. The top layer has a single PE, while the bottom layer has as one PE for each pixel in the image. The number of layers, L, can be found from the relation $L = \log_2(r)+1$, where r is the number of pixels per row or column in the base layer. Each PE not on the edge of a layer or in the top or bottom layers communicates with 13 neighboring PEs, one on the layer above, four on the layer below, and eight on the same layer. These 13 PEs are referred to as the pyramidal neighborhood. The pyramidal neighborhood is shown in Figure 9b. The eight nearest neighbors on the same layer are referred to as the lateral neighborhood. The connections at the root, base, and edge cells can be handled in fashions similar to those for the 4NN. Additionally, a border register is provided for high speed I/O between the array and either the image digitizer, image display unit, or mass storage. The border register has 2^{L-1} one-bit registers which connect to the PEs on one edge of the base level of the pyramid. The overall organization of the pyramid machine is shown in Figure 9a.

This organization is an implementation of the resolution pyramid data structure, which has been shown to have several advantages for vision processing [BALLA 82, HANSO 78]. For serial machines, the upper levels of the Pyramid can be processed, and the results of that processing used to guide the processing of the lower levels. This has the advantage of reducing the computations expended on the lower layers, as well as providing some measure of noise immunity. Region growing algorithms based on a split and merge technique also use the Pyramid data structure for efficiency gains. More information on the use of pyramids can be found in [TANIM 80].

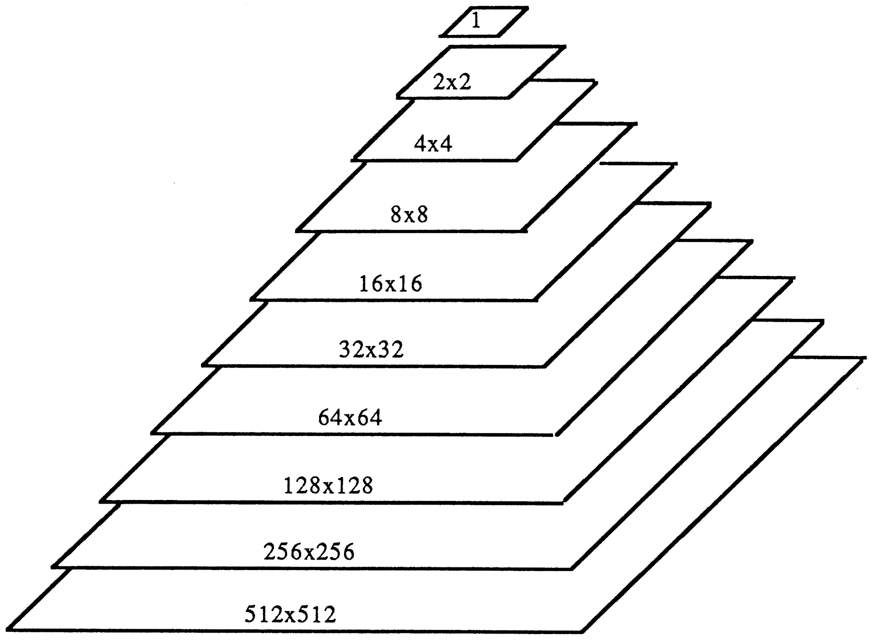


Figure 9a: The Pyramid Interconnection Network

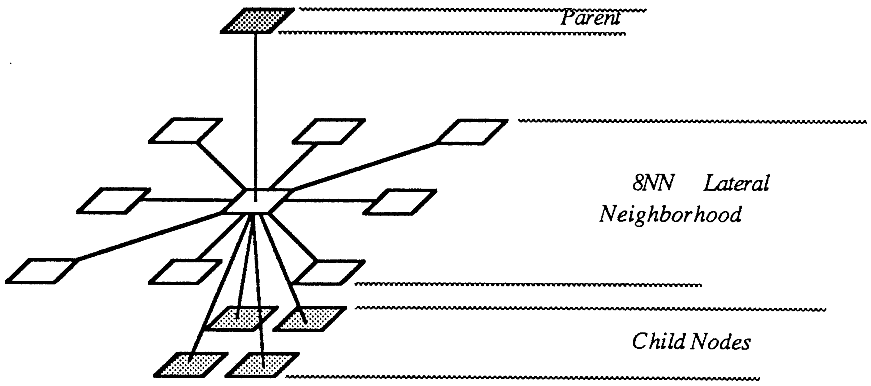


Figure 9b: The Pyramidal Neighborhood

Edge Treatments

The subsection on the 4NN briefly discussed possible methods of handling communication for those PEs on the edges of the processing arrays. These edge treatments are also applicable to the CAAPP and Pyramid machines. Various algorithms make up the image processing stage of computer vision, not all of which will necessarily use the same edge treatment. Edge treatments can be handled by the use of programmable switches. The communication lines at the edge of the processing array would be routed to these switches, which could then provide spiral, torus, wired-1, or wired-0 connection schemes. These switches might be located on the controller board, or on a seperate board.

Another common factor in all the interconnection networks is the need for I/O with the image acquisition, image display, and mass storage hardware. All the interconnection networks will be augmented with a border register to provide this functionality. The border register was briefly described in the Pyramid subsection.

VLSI Fabrication Considerations

At this point, it seems appropriate to consider some of the details of the VLSI implementation of the interconnection networks studied. The following sub-sections discuss the package and board level implementation of these networks for a 512 x 512 image. Several simplifying assumptions were used. The first was that the area required for the memory and ALU of the PEs would dominate the size of the die, therefore no comparisons of the differences in area of the interconnection networks were made. No gate counts were computed, however, current limits on gate count suggest that there be no more than 64 PEs, each with approximately 512 bits of memory, integrated into one package.

Despite the ever-increasing numbers of devices that can be fabricated onto a chip, pin counts are expected to remain limited to around one or two hundred pins per package until the advent of wafer-scale integration. Therefore, the number of PEs that can be fabricated onto a chip will be limited by practical pin count as well as by the total gate count. The CAAPP is the only interconnection network that directly addresses this limitation. However, by arranging the 64 PEs on each chip into an eight by eight array, the number of communication lines needed for each chip is not a major problem. The 4NN would require 32 pins for communication, the CAAPP would need eight. Even with the additional pins for instructions, power, ground, and clock, the pincounts are within current limits.

While the 4NN and CAAPP are within the capabilities of current fabrication technology, the limits on pin and gate count make the implementation of the pyramid impractical today. Therefore, the limits on the number of PEs were relaxed somewhat for the pyramid. Device sizes should soon be at a level that will allow more gates per device, without much help in the way of pincount limits.Pin counts do become restrictive for the Pyramid, as will be discussed later. By allowing 85 PEs per die, a practical configuration for the Pyramid was achieved which met the limits on pincount.

4NN and CAAPP Pincounts

The 4NN and CAAPP, using 64 PEs per package, will require a total of 4096 packages for a 512x512 image. If 64 packages are mounted on each printed circuit board, 64 such boards would be needed. Obviously, such a system would be expensive to construct. The submesh organization of the CAAPP would make it a cheaper and more reliable system. This is due to the reduced number of pins needed on the packages and boards. For the 4NN, each PE on the perimeter of the package will need a communication pin, corner PEs will need two. This gives a total of 32 communication pins for each package. For the CAAPP, only eight communication lines will be needed per package. An additional 25 pins will be needed for instructions, two for power and ground, one for the clock, and four for the content addressable broadcast and response mechanism. Thus, the number of pins for each package is 40 for the CAAPP and 64 for the 4NN.

Additional pins might be used for chip or quadrant enables. This would speed selections based on row-column coordinates by having all but a few of the least significant bits handled by chip and board enables instead of the ALU of the PEs. However, the benefit of the enable pins is unclear. For a 512x512 image there are only 18 bits in the row-column address. While such selection is a frequently used operation, its execution time by either method is quite small compared to a multiply or divide. For this study, the algorithms for superquadric estimation all execute on small portions of the array, allowing many estimations to be accomplished in parallel. Selections in these algorithms only depend on a few of the least significant bits in the row-column address, therefore the enable pins would not appreciably impact their execution. For this reason, no enable pins will be considered, and all row-column selections will be accomplished by a bit-serial associative search into memory.

Pincount savings at the package level will be especially important when the pincount of the printed circuit boards is considered. Assuming 64 packages per board, arranged in an eight by eight mesh, each board of the 4NN will require 256 connections to the backplane for communication lines while the CAAPP boards will only require 64. We again will need 32 additional lines for instructions, power, etc. Board level response counts must be provided, at the expense of an additional 17 lines. The total board level pincount is thus 113 for the CAAPP and 305 for the 4NN. The package and board level counts for the 4NN and CAAPP are presented with those for the Pyramid in Table V at the end of this section.

Pyramid Pincounts

The twin limitations of gate and pin counts are especially acute for the Pyramid. Given that we cannot integrate the Pyramid into a single package, we must decide on an appropriate way to distribute the PEs. This will require that the machine be composed of two type of chips; those which contain leaf, or base, cells and those that do not [DYER 81]. The first type is called a leaf chip, the other is a non-leaf chip. Several levels of the machine would be provided in each type of chip.

The number of levels that can be integrated onto a non-leaf chip will be limited by the pincount necessary for the lowest level of the non-leaf chip to communicate with its children. For example, consider the top of the pyramid. One line will be needed for communication above the root cell. If three levels of the pyramid are to be integrated into a single package, there will be 16 PEs on the bottom level. Each of these PEs must be provided with links to its four child nodes. This brings the pincount to 65 for communication with parent and children alone. To this must be added 33 more lines for instruction lines, power, ground, clock, and associative response. If we consider the chips which will make up the interior of the Pyramid, we see that we must also provide lines for communication within the lateral neighborhood. For the chips on the edges of the array, these lines will be wrapped around to the opposite side of the array to handle edge treatments as described earlier in this chapter. Since the lateral neighborhood is an 8NN mesh, the pincounts here will be greater than those for the 4NN or CAAPP. There are several ways to handle this 8NN connection, two are diagramed below in Figure 10. The first is the simple-minded solution. This requires three communication pins for each PE on the edge of a chip, while corner PEs require five pins. This is shown in the figure by highlighting a corner and edge PE, and showing their off-chip communication links with bold lines. The pincount can be reduced by adding two-bit buffers to each end of the communication lines. One of the buffer bits would be dedicated to broadcasting the value from the E register of the closest PE on the same chip. This is denoted as the S, or Send, bit in Figure 10. The second bit would hold the E register value from the PE on the other chip. This value would then be routed to the necessary PEs on the first chip. This bit is denoted by R, or Receive, in the figure. Using this scheme, the pincount can be reduced to one pin for each edge PE, with corners needing three pins. This is a penalty of only four pins, one at each corner, compared to the 4NN and CAAPP interconnection networks.

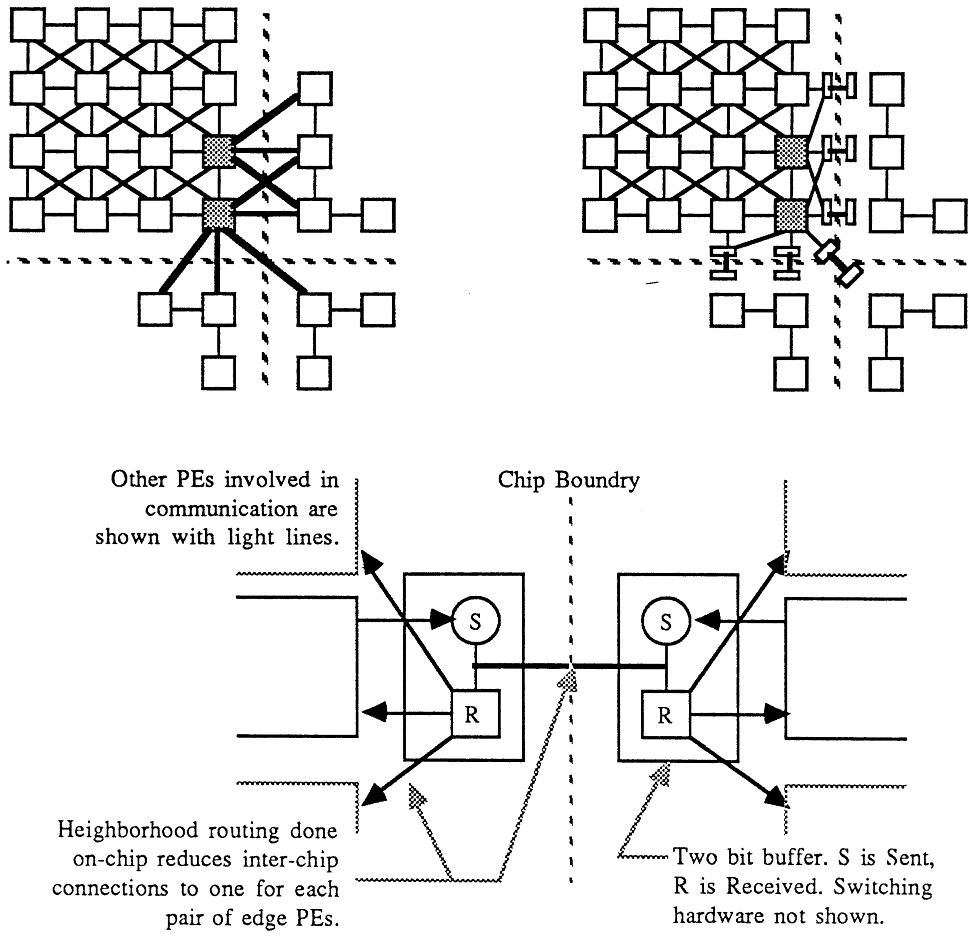

Other PEs involved in communication are shown with light lines.
Chip Boundry
S
S
R
R
Heighborhood routing done on-chip reduces inter-chip connections to one for each pair of edge PEs.
Two bit buffer. S is Sent, R is Received. Switching hardware not shown.

Figure 10: Inter-Chip Communication for 8NN Meshes

For the three-level non-leaf package discussed above, this will add another 40 pins, bringing our total to 138. If only two levels of the pyramid are fabricated onto a non-leaf chip, the total pin count is reduced to 70. Four levels gets quite expensive with 365 pins per chip. The pin counts as a function of the number of levels per package are tabulated below in Table III. This table also gives information about how many chips and boards would be required for a complete 512 x 512 system. This will depend upon how many levels are integrated into the leaf chips. The numbers in Table III are based on the assumption of four levels in the leaf chips. They also assume a single PE at the top of each non-leaf chip.

TABLE III

NON-LEAF CHIP PARAMETERS AS A FUNCTION OF THE NUMBER OF LEVELS PER PACKAGE

Levels per chip	Number of PEs	Comm. Pins per chip	Total Pins per chip	Number of Chips for 512x512 Image
2	5	37	70	273
3	21	105	138	65
4	85	332	365	17

The pin limit is not so severe for the leaf chips, since they do not need four lines for each terminal cell. Assuming a single PE at the top, a three-level leaf chip will have 21 PEs per package. Such a chip will implement an 4 x 4 section of the base level. There will be 40 lines for communication within the lateral neighborhoods, one for communicating with the parent PE, as well as 33 pins for instructions, etc. Such a device would have 71 pins.

There is no need for the top level of the leaf cell to have a single PE. If we do away with this restriction we can put more PEs on each leaf cell, reducing the number of packages needed. Table IV summarizes the pin, package and board counts versus the number of levels and the number of PEs that occupy the top level.

TABLE IV

LEAF CHIP PARAMETERS AS A FUNCTION OF THE NUMBER OF LEVELS PER PACKAGE AND THE NUMBER OF PES ON THE TOP LEVEL

Levels per Chip	Number of PEs on Top Level	Number of PEs per Chip	Communication Pins per Chip	Total Pins per Chip	Number of Chips for 512x512 Image
2	1	5	21	54	65,536
2	4	20	36	69	16,384
2	16	80	72	105	4,096
3	1	21	41	74	16,384
3	4	84	72	105	4,096
4	1	85	77	110	4,096
5	1	341	145	178	1,024

System Level Assembly

Assembling the 4NN and CAAPP into a system would be relatively straightforward. The designer of a Pyramid has several decisions to make which the other two machines do not present. The decision about how many levels per chip is one instance. In a similar fashion, we may wish to put both types of chip onto a single board to reduce pincount. Another minor problem to address concerns the root cell. The number of levels used for the leaf and non-leaf chips may not allow the root cell to appear at the top of the topmost package. In other words, the root PE will have parents! This is not as serious as it sounds, but a slight amount of special handling would be necessary when dealing with the top of the pyramid. For Table V, the pyramid was assembled from 4 level leaf cells and three-level non-leaf cells. This gives a balanced pyramid, i.e., the root PE has no parents. The decision was also made to put 64 leaf chips and one non-leaf chip onto the low level boards. The single remaining non-leaf chip could then be easily placed on the controller board. This reduces the number of board-level connections, while also reducing the board count by one. The board eliminated would have held 64 non-leaf cells, each with 16 PEs on the bottom layer. Since each PE on the bottom layer of the non-leaf cell needs four communication lines to its children, eliminating this board reduces the complexity of the backplane considerably. The counts for such a system are presented below in Table V, along with those for the 4NN and CAAPP.

The final column gives an estimated MTBF (Mean Time Between Failures) for each machine. Typical failure times for backplane connections are on the order of 1E8 to 1E9 hours. For complex VLSI chips, the failure time is approximately 1E7 to 1E8 hours. The failure rates of validated printed circuit boards and solder joints are so low that they can be neglected[1]. For the table below, the failure times were taken to be 5E8 hours for the connections and 5E7 hours for the chips. It should be noted that the failures may not be fatal, or even seriously degrade results. The massive parallelism of the interconnection networks considered provides a degree of fault-tolerance. Since most image processing algorithms are local, the effects of a hardware failure may remain confined to a small area of the image. Detecting such areas and deciding on an appropriate error-recovery technique is another question, and beyond the scope of this study.

[1] Reliability estimates courtesy Dr. Dave Ballew, AT&T Technologies

TABLE V

SYSTEM LEVEL COMPARISON

Network	PEs per 512^2 image	Chips per image	Boards per system	Backplane lines per board	Backplane lines per system	MTBF (weeks)
Pyramid	349,525	4,161	65	570	36,480	38
4NN	262,144	4,096	65	286	18,304	50
CAAPP	262,144	4,096	65	94	6,016	63

CHAPTER III

PROCEDURE

If we agree that CSG combinations of superquadrics seem to form a desirable data structure to bridge the gap between low and high level vision, the next question we are faced with is how to recover the superquadric parameters from the image. Fortunately, there exists a dual relation between the surface position and normal vectors of a superquadric that allows this information to be recovered if we have knowledge of the surface normals throughout the image. The first section of this chapter presents the derivation of regression equations that will allow the recovery of the superquadric parameters given knowledge of the surface normals. The second section of the chapter discusses the solution of the regression equations and presents algorithms suited for the parallel machines studied. Equations for the execution time and memory requirements of each stage of the solution process are presented with the algorithms. The final section discusses the performance estimation techniques used and limitations of the regression equations derived.

Derivation of the Regression Equations

Pentland, in [PENTL 86] presents a derivation showing how knowledge of the image plane component of the surface normal, as well as its derivatives, may be used to estimate the center of the superquadric form and the shape parameter e_2. In the interests of completeness this derivation is presented below, terminating with equation (23a). Equations (24) to (26) modify equations (23 a,b) for a least-squares solution.

The surface position vector $\underline{X}$ and surface normal vector $\underline{N}$, as functions of longitude ω and latitude η are given by

$$\underline{X}(\eta,\omega) = \begin{pmatrix} a_1 C_\eta^{e1} C_\omega^{e2} \\ a_2 C_\eta^{e1} S_\omega^{e2} \\ a_3 S_\eta^{e1} \end{pmatrix} \qquad \underline{N}(\eta,\omega) = \begin{pmatrix} 1/a_1 C_\eta^{2-e1} C_\omega^{2-e2} \\ 1/a_2 C_\eta^{2-e1} S_\omega^{2-e2} \\ 1/a_3 S_\eta^{2-e1} \end{pmatrix} \qquad (1,2)$$

where $C_\eta = \cos(\eta)$, $S_\omega = \sin(\omega)$ and a_1, a_2, a_3 are scaling factors for x,y, and z. The surface normal vector $\underline{N} = (x_n, y_n, z_n)$ can be written as a function of the surface position vector $\underline{X} = (x, y, z)$,

$$\underline{N}(\eta,\omega) = \begin{pmatrix} 1/x\ C_\eta^2 C_\omega^2 \\ 1/y\ C_\eta^2 S_\omega^2 \\ 1/z\ S_\eta^2 \end{pmatrix} \qquad (3)$$

From equations (1) and (3) we have

$$x_n = \frac{1}{x} C_\eta^2 C_\omega^2 \qquad \text{and} \qquad y_n = \frac{1}{y} C_\eta^2 S_\omega^2, \qquad (4,5)$$

so

$$\frac{y_n}{x_n} = \frac{x}{y}\tan^2\omega, \qquad \text{or} \qquad \left(\frac{yy_n}{xx_n}\right)^{\frac{1}{2}} = \tan\omega. \qquad (6,7)$$

Alternative expressions for tan ω may be derived as follows:

$$x = a_1 C_\eta^{e1} C_\omega^{e2}, \qquad y = a_2 C_\eta^{e1} S_\omega^{e2}, \qquad (8)$$

so

$$\frac{x}{y} = \frac{a_1}{a_2}\left(\frac{C_\omega}{S_\omega}\right)^{e2} \qquad \text{or} \qquad \left(\frac{y\ a_1}{x\ a_2}\right)^{\frac{1}{e2}} = \tan\omega. \qquad (9,10)$$

Combining, we obtain

$$\left(\frac{yy_n}{xx_n}\right)^{\frac{1}{2}} = \left(\frac{ya_1}{xa_2}\right)^{\frac{1}{e2}} \qquad \text{or} \qquad \frac{y_n}{x_n} = \left(\frac{y}{x}\right)^{\frac{2}{e2-1}} \left(\frac{a_1}{a_2}\right)^{\frac{2}{e2}}. \qquad (11,12)$$

Letting $\tau = \frac{y_n}{x_n}$, $k = \left(\frac{a1}{a2}\right)^{\frac{2}{e2}}$, and $\zeta = \frac{2}{e2-1}$, we find

$$\tau = k\left(\frac{y}{x}\right)^{\zeta}, \quad \frac{d\tau}{dy} = \frac{k\zeta}{x}\left(\frac{y}{x}\right)^{\zeta-1}, \quad \text{and} \quad \frac{d\tau}{dx} = \frac{-k\zeta y}{x^2}\left(\frac{y}{x}\right)^{\zeta-1}. \qquad (13,14,15)$$

These equations can be combined to give two equations which relate the unknown shape parameters to image measurable quantities:

$$\frac{\tau}{\frac{d\tau}{dy}} = \frac{y}{\zeta} \qquad \text{and} \qquad \frac{\tau}{\frac{d\tau}{dx}} = \frac{-x}{\zeta} \qquad (16,17)$$

The two equations above are rather limited, since they assume the superquadric is centered at the origin, and has no rotation. For the case of rotation and translation in the image plane, the equations relating the coordinate systems become:

$$x^* = C_\theta\,(x - x_0) + S_\theta\,(y - y_0), \qquad y^* = -S_\theta\,(x - x_0) + C_\theta\,(y - y_0) \qquad (18)$$

where θ is the rotation, x_0, y_0 the translation, and (x^*, y^*) the new rotated and translated coordinate system. The tilt τ then becomes

$$\tau = \frac{y_n^*}{x_n^*} = \frac{-S_\theta x_n + C_\theta y_n}{C_\theta x_n + S_\theta y_n}, \qquad (19)$$

and the derivative of (19) is

$$\frac{d\tau}{dy^*} = \left(-S_\theta \frac{dx_n}{dx} + C_\theta \frac{dx_n}{dy}\right)(C_\theta\, x_n + S_\theta\, y_n)^{-1}$$

$$-\left(C_\theta \frac{dx_n}{dy^*} + S_\theta \frac{dy_n}{dy^*}\right)(-S_\theta\, x_n + C_\theta\, y_n)\,(C_\theta\, x_n + S_\theta\, y_n)^{-2}$$

$$= (C_\theta\, x_n + S_\theta\, y_n)^{-2}\left(x_n \frac{dy_n}{dy^*} - y_n \frac{dx_n}{dy^*}\right). \qquad (20)$$

Noting that

$$\frac{dx_n}{dy^*} = \frac{dx_n}{dx}\frac{dx}{dy^*} + \frac{dx_n}{dy}\frac{dy}{dy^*} = \frac{-dx_n}{dx}S_\theta + \frac{dx_n}{dy}C_\theta ,$$

$$\frac{dy_n}{dy^*} = \frac{dy_n}{dx}\frac{dx}{dy^*} + \frac{dy_n}{dy}\frac{dy}{dy^*} = \frac{-dy_n}{dx}S_\theta + \frac{dy_n}{dy}C_\theta , \qquad (21\ a,b)$$

we now rewrite (16) as

$$(C_\theta x_n + S_\theta y_n)(-S_\theta x_n + C_\theta y_n) = \frac{1}{\zeta}\big(-S_\theta (x-x_o) + C_\theta (y-y_o)\big)$$

$$\times \left(x_n\left(-S_\theta \frac{dy_n}{dx} + C_\theta \frac{dy_n}{dy}\right) - y_n\left(-S_\theta \frac{dx_n}{dx} + C_\theta \frac{dx_n}{dy}\right)\right) . \qquad (22)$$

Finally, collecting the image measurable quantities in square brackets, and the unknown compound parameters in parenthesis, we obtain:

$$\begin{aligned} 0 = {} & \left[x_n^2 - y_n^2\right] (\xi C_\theta S_\theta) + \left[x_n y_n\right] (\xi(S_\theta^2 - C_\theta^2)) \\ & + \left[xx_n \frac{dy_n}{dy} - xy_n \frac{dx_n}{dy}\right] (-C_\theta S_\theta) + \left[xx_n \frac{dy_n}{dx} - xy_n \frac{dx_n}{dx}\right] (S_\theta^2) \\ & + \left[yx_n \frac{dy_n}{dy} - yy_n \frac{dx_n}{dy}\right] (C_\theta^2) + \left[yx_n \frac{dy_n}{dx} - yy_n \frac{dx_n}{dx}\right] (-C_\theta S_\theta) \\ & + \left[x_n \frac{dy_n}{dy} - y_n \frac{dx_n}{dy}\right] (C_\theta S_\theta x_0 - C_\theta^2 y_0) \\ & + \left[x_n \frac{dy_n}{dx} - y_n \frac{dx_n}{dx}\right] (C_\theta S_\theta y_0 - S_\theta^2 x_0), \end{aligned} \qquad (23a)$$

Using a similar development, (17) becomes:

$$\begin{aligned}0 = {} & \left[\, y_n{}^2 - x_n{}^2 \,\right] (\xi C_\theta S_\theta) + \left[\, -x_n y_n \,\right] (\xi(S_\theta{}^2 - C_\theta{}^2)) \\ & + \left[\, x y_n \frac{dx_n}{dy} - x x_n \frac{dy_n}{dy} \,\right] (-C_\theta S_\theta) + \left[x x_n \frac{dy_n}{dx} - x y_n \frac{dx_n}{dx} \right] (C_\theta{}^2) \\ & + \left[\, y x_n \frac{dy_n}{dy} - y y_n \frac{dx_n}{dy} \right] (S_\theta{}^2) + \left[\, y y_n \frac{dx_n}{dx} - y x_n \frac{dy_n}{dx} \,\right] (-C_\theta S_\theta) \\ & + \left[y_n \frac{dx_n}{dx} - x_n \frac{dy_n}{dx} \,\right] (C_\theta S_\theta y_0 + C_\theta{}^2 x_0) \\ & + \left[y_n \frac{dx_n}{dy} - x_n \frac{dy_n}{dy} \right] (C_\theta S_\theta x_0 + S_\theta{}^2 y_0) \ . \end{aligned} \tag{23b}$$

As presented above, the two equations are not well suited to constructing a linear regression, since they would give rise to a system of the form $A\underline{x} = 0$. For this system the A matrix must be singular if we are to have anything but the trivial solution. However, note that there are a number of common terms in the two equations. If we rewrite 23a and 23b as:

$$a_1 x_1 + a_2 x_2 + \ldots + a_8 x_8 = 0 \tag{24a}$$

$$b_1 y_1 + b_2 y_2 + \ldots + b_8 y_8 = 0 \tag{24b}$$

we may take the sum and difference of these two equations, use the fact that $\sin^2\theta + \cos^2\theta = 1$, and obtain a different set of equations which are suited to a least-squares solution. The first equation, from the sum of 24a and 24b, reduces to:

$$-(a_4 + a_5) = a_7(x_7 - y_8) + a_8(x_8 - y_7) \ , \tag{25a}$$

while the second , from the difference of 24a and 24b, becomes:

$$\begin{aligned}-(a_4 - a_5) = {} & 2[a_1 x_1 + a_2 x_2 + a_3 x_3 + a_6 x_6] + 2[a_5 - a_4]\, x_5 \\ & + a_7(x_7 + y_8) + a_8(x_8 + y_8)\end{aligned} \tag{25b}$$

In terms of the image measurable quantities, these are:

$$- \left[xx_n \frac{dy_n}{dx} - xy_n \frac{dx_n}{dx} + yx_n \frac{dy_n}{dy} - yy_n \frac{dx_n}{dy} \right]$$

$$= \left[y_n \frac{dx_n}{dy} - x_n \frac{dy_n}{dy} \right] (y_0) + \left[y_n \frac{dx_n}{dx} - x_n \frac{dy_n}{dx} \right] (x_0) , \tag{26a}$$

and

$$- \left[xx_n \frac{dy_n}{dx} - xy_n \frac{dx_n}{dx} \right] + \left[yx_n \frac{dy_n}{dy} - yy_n \frac{dx_n}{dy} \right]$$

$$= 2 \left[x_n^2 - y_n^2 \right] * \left(\frac{\xi C_\theta S_\theta}{S^2_\theta - C^2_\theta} \right) + 2 \left[x_n y_n \right] (\xi)$$

$$+ 2 \left[xx_n \frac{dy_n}{dy} - xy_n \frac{dx_n}{dy} + yx_n \frac{dy_n}{dx} - yy_n \frac{dx_n}{dx} \right] * \left(\frac{-C_\theta S_\theta}{S^2_\theta - C^2_\theta} \right)$$

$$+ \left[x_n \frac{dy_n}{dy} - y_n \frac{dx_n}{dy} \right] * \left(\left(\frac{2 C_\theta S_\theta}{S_\theta^2 - C_\theta^2} x_0 \right) + y_0 \right)$$

$$+ \left[x_n \frac{dy_n}{dx} - y_n \frac{dx_n}{dx} \right] * \left(\left(\frac{2 C_\theta S_\theta}{S_\theta^2 - C_\theta^2} y_0 \right) + x_0 \right) \tag{26b}$$

The two equations above may be used to construct a linear regression to solve for the unknown parameters in parenthesis. Since we have seven unknowns, we will need at least seven equations. Each pixel gives rise to two equations, thus we can solve for estimates of the superquadric parameters over regions as small as two by two pixels. For better noise immunity it may be that we will want to use larger regions. A technique to solve the regression matrix will be discussed in the next section.

Parallel Regression Algorithms

Now that the regression equations have been derived, we must have a technique to solve them. This section discusses an implementation of the standard least squares technique suited for the parallel architectures studied. The actual algorithms used are collected in Appendix B; this section describes the behavior of the solution technique. We first discuss the notation and terminology used in the algorithms. We also discuss how they map onto bit-serial arrays. We then give the main procedures that will be used for the flat and Pyramid machines. The differences between the flat and Pyramid algorithms are then discussed. Each stage of the procedures is described in separate sub-sections. Formulas used to calculate the execution time of the algorithms are given in Appendix B, along with the algorithms. The data movements and operations of each stage are described in the appropriate sub-section of this chapter. For those stages that dominate memory consumption, the contents of memory are diagrammed. The execution times as functions of the numeric precision, size of the estimation region, and architecture are presented in the next chapter.

The notation we will use for the least-squares solution is:

$$\underline{x} = (\mathbf{A}^T\mathbf{A})^{-1}\,\mathbf{A}^T\underline{b} \tag{27}$$

where $\underline{x}$ is the vector of parameters to be estimated, **A** is the matrix formed from the coefficients on the right side of equations (26), and $\underline{b}$ is the vector of coefficients from the left side of (26). T and -1 indicate the matrix transpose and inverse, respectively. Recall that we can estimate the parameters from a region as small as two pixels by two pixels. Such a region will be called the *estimation region.* The *dimension* of the estimation region is the square root of the number of PEs it contains. This study only considers square estimation regions. Times will be provided for regions of dimension two, four, and eight.

The algorithms are capable of solving for the estimated parameters of hundreds or thousands of these estimation regions at the same time. For the case of the 2x2 estimation region and a 512x512 image, 4096 of these systems can be solved in parallel! The matrix manipulations used do not execute within one PE. Instead the elements of a matrix are distributed, one per PE, among a group of neighboring PEs. For the case of the 2x2 estimation region, there will be eight rows and seven columns in the **A** matrix. This will be mapped onto an eight by eight submesh of the architectures. Such a submesh will be referred to as the *solution submesh.* Note that the solution submesh is larger than the **A** matrix. The solution submeshes are square matrices of size n by n, where n is four times the dimension of the estimation region. In other words, there are 16 estimation regions in each solution submesh. This was done to ease the selection of elements within the solution submesh. For example, with an eight by eight submesh, the top left PE in all submeshes can be enabled by performing a content-addressable search to determine if the three least-significant bits of the row and column coordinates in each PE are all zero. This is a total of six comparisons. If the submeshes were seven by seven and packed tightly together, we would have to search on many different keys and the length of the keys would be the full size of the row and column coordinates. This submesh technique is why only 4096 systems can be solved in parallel when there are 65,536 two by two estimation regions in a 512 by 512 image. The solution algorithm must be repeated 16 times to obtain the total solution. This factor of 16 holds true for estimation region dimensions that are a power of two, on a flat architecture. Depending on the needs of the high level vision component, this factor of 16 may not be the case for the Pyramid. Differences in programming the Pyramid are discussed below, after an overview of the procedure.

Main Procedure

The calculation of $\underline{x} = (A^T A)^{-1} A^T \underline{b}$ will be broken into stages and algorithms presented for each stage. Separate sections of this chapter will discuss the operation of each stage. This section discusses the main procedure, and presents two algorithms. The first is for the flat machines, the second is for the Pyramid.

The first step is to calculate the coefficients $a_1, a_2, \ldots, a_7$, b_1, and b_2 of the regression equations. Once this is complete, we must move the coefficients to form the **A** and $\underline{b}$ matrices within the solution submesh. The products $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\underline{b}$ are then calculated, and $\mathbf{A}^T\mathbf{A}$ is inverted. Next, $(\mathbf{A}^T\mathbf{A})^{-1}$ and $\mathbf{A}^T\underline{b}$ are multiplied to give the vector of the parameter estimates, $\underline{x}$. The elements of the $\underline{x}$ vector are then moved back to the estimation region that gave rise to them, where they will be stored in an unused field of the local memory until all 16 iterations have completed. Formulas to estimate the execution time were derived for each stage. These formulas are presented along with the corresponding algorithm in Appendix B. Variables in these formulas are the size of the estimation region and the execution time of fundamental arithmetic and data transfer operations. The time for the fundamental operations will be a function of the format and precision of the numeric operands. This study assumes a two's-complement, fixed-point format and considers precisions of 16, 32, 48, and 64 bits. Algorithms and execution times for the fundamental operations are presented in Appendix A.

The estimation procedure does not use inordinate amounts of memory, all calculations can be accomplished within the nine words needed to store the regression coefficients at each pixel. Additional memory must be allocated to hold the row and column addresses of the PEs within the array. Space must also be provided for storing the estimated parameters from an estimation region while the other 15 estimation regions are being processed. Since we have seven parameter estimates to store, the four by four and eight by eight estimation regions will need one extra word in each PE. For the two by two estimation region we must have two words per PE of temporary storage. These memory needs are shown below in Figure 11. Figure 11a shows the input data for the superquadric estimation procedure. All PEs are assumed to have this information in their local memory at the begining of the superquadric recovery procedure. Figure 11b shows the maximum amount of memory needed by the procedure, which occurs when we must store the regression coefficients at each pixel. The space for storing results while other estimation regions are being processed is also shown. The field marked with an asterisk is only needed for the two by two estimation region. Figure 11c shows the contents of memory once the estimation procedure has finished.

Row,Col	Xn	Yn	dXn/dX	dXn/dY	dYn/dX	dYn/dY	Unused

A: Initial contents of all PE's local memory

Row,Col	a1	a2	a3	a4	a5	a6	a7	b1	b2	t1	t2 *	Unused

B: Space needed to store all regression coefficients in PE

		(ς)	(-T)			(-Yo)	(-Xo)	
Row,Col	x1	x2	x3	x4	x5	x6	x7	Unused

C: Final contents of all PE's local memory

Figure 11: PE Memory Contents on the CAAPP and 4NN Machines

Despite being reasonably memory-efficient, the small size of the local memories presents a problem when the precision of the operands is greater than 32 bits. For the case of 64 bit operands and a two by two estimation region, the total memory required is 722 bits. To simplify the algorithms, we will assume that the PEs have enough local memory to avoid disk swaps while processing a single estimation region. The memory contents are undefined in all PEs that do not hold one of the results, so we will account for the time to restore the regression coefficients from disk as they are needed at the beginning of each of the 16 iterations. The algorithm below is the main procedure for the two flat interconnection networks. It shows that we must perform 16 iterations of the solution procedure, as well as showing the restoration of the regression coefficients from disk at the start of each iteration.

ALGORITHM 3 MAIN PROGRAM FOR FLAT NETWORKS

```
calculate regression coefficients
save regression coefficients to disk
for i = 0 to 3                          ; For each estimation region in the
    for j = 0 to 3                      ; solution submesh
        move elements of estimation region to their
           position in the solution (sub)submesh
        fill the submesh
        Calculate $A^T\underline{b}$
        Calculate $A^TA$
        Invert $A^TA$
        Calculate $\underline{x}$
        Move parameter estimates back to estimation region and save
        if $i \neq 3$ and $j \neq 3$, restore regression coefficients from disk
    next j
next i
distribute parameters to fields of all PEs in estimation region
done
```

A final topic that needs consideration is how to program the Pyramid. It can, of course, be programmed the same as the flat architectures. Such a technique offers no speed gain compared to a flat solution, but we will have calculated on all the layers large enough to hold the solution submesh for the dimension of the estimation region. The extra information provided by a wide range of scales could be quite useful to a sophisticated vision system which employed scale-space filtering.

Another way exists. If we are willing to give up the solutions for the two lowest levels, we can avoid the 16 iterations needed by the flat machines and obtain a solution for all layers large enough to hold one or more estimation regions in a single iteration. This approach moves the coefficients of the regression equation down two levels, to where there are 16 times as many PEs. The **A** matrix is formed, the matrix operations carried out, and the $\underline{x}$ vector calculated, giving the superquadric parameters. The parameters are then moved up two levels to all the PEs of the estimation region.

Another benefit of this technique is that it provides a mechanism for reducing the noise in the surface normals information. Since the pixel values two levels up are the mean of 16 pixels on the lowest layer, much of the noise in the image will have been averaged out. Most of the information that would be lost in the Pyramid algorithm would be texture information. To illustrate this, imagine that the camera is photographing a close-up of a basketball. If we use a two by two estimation region, the superquadric parameters on the lowest level will give the center and shape of the small lumps that provide texture to the ball. If we want to know the center and shape of the ball itself, we must move up a few layers into the machine. The times for the Pyramid in the charts in Chapter Four are for this approach. The times for the flat machines reflect their factor of 16 penalty. Times for the flat algorithm running on the Pyramid do not differ significantly from those of the 4NN, despite the 8NN lateral neighborhood of the Pyramid. This is because very little communication takes place between diagonal PEs. In fact, this study was originally going to consider four interconnection networks. The fourth would have been the 8NN mesh, but it was dropped when it became apparent that it offered no appreciable advantage over the 4NN.

The algorithm below is the main routine for the Pyramid. All of the stages in this algorithm are the same as for the flat machines except for the formation of the **A** matrix and $\underline{b}$ vector, the movement of the results to the original estimation region, and the lack of a loop to do the 16 iterations. Since all the work can be accomplished in a single pass, there is no need to swap data to and from disk.

ALGORITHM 4 MAIN PROGRAM FOR PYRAMID NETWORKS

```
calculate regression coefficients
move elements of estimation region down two levels
    to their position in the solution (sub)submesh
fill the submesh
Calculate $A^T\underline{b}$
Calculate $A^TA$
Invert $A^TA$
Calculate $\underline{x}$
Move parameter estimates back up two levels to estimation region
Distribute coefficients to all PEs in estimation region
```

Calculation of the Regression Coefficients

The first step in both main routines is to calculate the regression coefficients from the values of the surface normals and their derivatives. Algorithm B1 in Appendix B performs this task. There is no difference in the algorithm or execution time for the three architectures. All PEs will calculate their coefficients at the same time, and these values do not depend on their neighbors. We will assume that the values of the surface normals and their derivatives are stored as p-bit, fixed-point numbers, as shown below in Figure 12. We will also require the coordinates of each pixel in the image. These will be represented by two 9-bit integers, spanning 0 to 511 for both x and y. Also shown on this figure is the space needed to store the results of estimation regions that have been processed while the other estimation regions are being used.

The Pyramid will calculate coefficients for all levels, but will not use those on the lowest two levels. There is no performance penalty for the unnecessary calculations, but the algorithms could be easily modified to eliminate their calculation for hygienic purposes. The Pyramid could benefit from having a special version of this procedure. The algorithm in Appendix B requires 11 multiply instructions to be issued. By moving the data down one level, the Pyramid can get by with four multiply instructions. However, this technique was not pursued. The data movements needed to move the data down one level, without destroying information that will be needed at a later stage, are rather intricate. The benefit of the reduction in multiplies would be swamped by the factor of 16 the Pyramid already enjoys over the flat machines. For these reasons the flat algorithm was used for all machines. Figure 12 gives the contents of the local memory of all PEs both before and after the regression coefficients have been calculated. The differences between this figure and Figure 11 are that the Pyramid needs to store its level as well as its row and column coordinates, but does not need the temporary storage to hold results while other estimation regions are being processed.

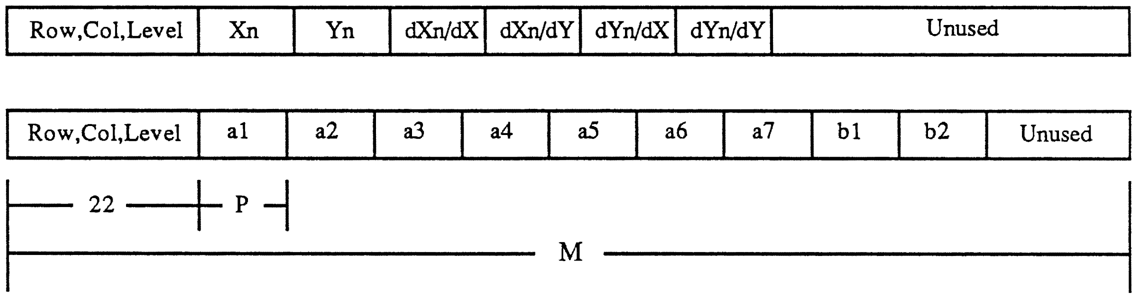


Figure 12: Memory Contents on the Pyramid Machine

Forming the A Matrix and b Vector

Once the coefficients have been calculated, we must move them to form the solution submesh for the A matrix and b vector. The solution submesh will always be a square matrix 16 times the size of the estimation region, so this routine must be repeated 16 times on the flat machines. Moving the coefficients will destroy the contents of memory fields, so we must save the coefficients to disk, then restore them on later iterations of this routine.

The Pyramid has a different algorithm for this process than the flat machines and will not need the disk swaps or 16 iterations.

The work of this stage is broken into two parts. The first part moves two by two blocks of the estimation region to the upper-left corner of each sub-submesh within the solution submesh. These blocks are nine words deep, to hold a_1 through a_7 along with b_1 and b_2. For the two by two estimation region there is only one sub-submesh in the solution submesh. The eight by eight estimation region will have 16 such sub-submeshes. The second part of this procedure fills the sub-submeshes once the estimation region has been broken into two by two sections and moved to the sub-submeshes. The reason for this solution technique will be given shortly, after a discussion of the form of the A matrix and $\underline{b}$ vector. The first part, moving the estimation region to its position within the sub-submesh, differs for the two types of machine. The second part, filling the sub-submeshes, is identical for all machines. Algorithms B2 and B3 in Appendix B present the first part of this stage for the flat and Pyramid machines, respectively. Algorithm B4 gives the second part.

Note that the regression equations are sparse. On a serial processor, we could take advantage of this to reduce the work of the matrix inversion procedure. Each pixel i gives rise to the two equations:

$$b_{i1} = a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{i5}x_5 + 0x_6 + 0x_7 \tag{28a}$$

and

$$b_{i2} = 0x_1 + 0x_2 + \ldots + 0x_5 + a_{i6}x_6 + a_{i7}x_7 \; . \tag{28b}$$

If we collect the versions of 28a at the top of the A matrix, and the versions of 28b at the bottom, the matrix will have groups of zeros as illustrated below in Figure 13. The main advantage of this organization of the matrix would occur when A^TA is inverted. The form of A^TA is also shown in Figure 13. Notice that this matrix is partitioned into 5x5 and 2x2 submatrices. The inverse of the A^TA matrix may be found by inverting the two non-zero submatrices in place. This will be faster to invert than the original 7x7 would have been. However, for the parallel machines studied, the percentage gains of this stage are small compared to those that would be achieved on a serial processor. This is due to the disproportionate share of time it takes to execute multiplies and divides on the bit-serial arrays. Inverting the 2x2 requires that we scale the elements by the norm of the 2x2 matrix. The division it takes to accomplish this considerably reduces the utility of the sparseness of the equations. For the serial processor we want to minimize the number of multiplies that must be *accomplished.* For the bit-serial arrays, we want to minimize the number of multiply instructions that must be *issued.* On a parallel machine, this is not the same thing as the number of multiplies that are accomplished. On the bit-serial arrays we can do one or a quarter of a million multiplies in the same time, the only restriction on doing a quarter of a million is that the data all be arranged properly. The $\mathbf{A}^T\mathbf{A}$ matrix is sparse, symmetric, and positive-definite. Serial processors can take advantage of these properties to reduce the total number of multiply and divide instructions executed. Unfortunately, the complex data movements and lack of parallelism of such procedures make them poorly suited to implementation on the architectures considered in this study. The inversions will be performed by Gaussian elimination, which is well suited to the machines considered due to the regularity of the data movements. Referring to Chapter Four, Figure 26, we see that inverting the 7x7 matrix will take approximately 6.6E4 clock cycles for 32-bit operands on the Pyramid. The 5x5 will take 4.6E4 clocks, the 2x2 requires 1.8E4. The savings by breaking the calculation into two inversions is less than 5%. If the 2x2 submatrix was inverted directly, its execution time could be halved. However, the savings compared to the 7x7 inversion are still less than 10%.

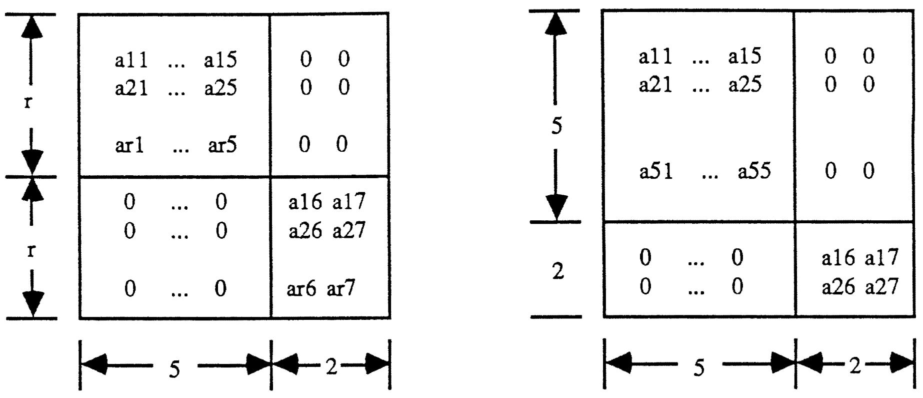
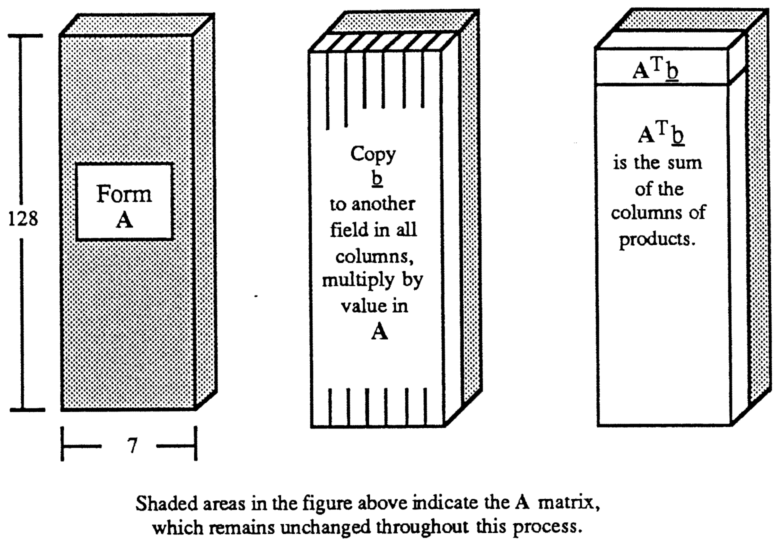


Figure 13: Form of the Sparse A and A^TA Matrices

We now can discuss why the algorithms use a square solution submesh which has been divided into eight by eight sub-submeshes. Consider the 8x8 estimation region. This should have an **A** matrix with 128 rows and 7 columns. Such a long matrix will make moving data up and down the columns quite slow. It is also unsuited to implementation on the Pyramid. The organization of the Pyramid begs that the solution submesh be square in order to avoid wasting resources or incurring prohibitive communication costs. The search for a better routine for the Pyramid paid the unexpected benefit of showing a better way to program the flat machines. Consider how we can calculate the vector $A^T\underline{b}$, where **A** is 128 by 7 and $\underline{b}$ is 128 by 1. A simple approach is to form the **A** matrix, one element per PE, in a 128 by 7 submesh. The $\underline{b}$ vector would be placed in another memory field in the first column of **A**. The $\underline{b}$ vector would then be copied to all the columns of the **A** matrix. The values of a_{ij} and b_i in each PE would be multiplied together, then sums taken up each column. At the end of this process the product $A^T\underline{b}$ will be in the top row of the 128 by 7 submesh. Actually, since $A^T\underline{b}$ should be a column vector, we have computed $(A^T\underline{b})^T$. However, this transpose will turn out to be a benefit rather than a problem.

$A^T\underline{b}$ can also be solved on a 32 x 32 solution submesh by having four groups of size 32 x 8. The large **A** and $\underline{b}$ matrices are simply broken into four, each resulting matrix having 32 rows and 7 columns. The extra columns of PEs are unused. The $\underline{b}$ vector is copied to all the columns within the group, the multiply and column sum are as described before. The final step needed for this arrangement is to collect the column sums from the top row of each 32 by 7 group into a total sum. Figure 14, below, shows the two solution techniques.


128
7
Form A
Copy b to another field in all columns, multiply by value in A
$A^T b$
$A^T b$ is the sum of the columns of products.
Shaded areas in the figure above indicate the A matrix, which remains unchanged throughout this process.

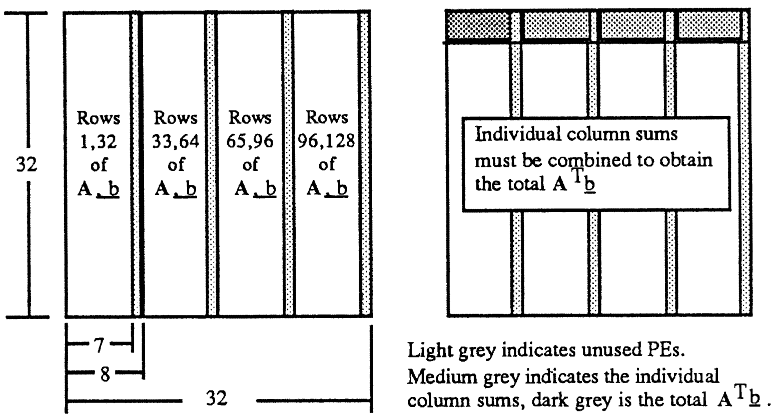

32
Rows 1,32 of A.b
Rows 33,64 of A.b
Rows 65,96 of A.b
Rows 96,128 of A.b
7
8
32
Individual column sums must be combined to obtain the total $A^T b$
Light grey indicates unused PEs. Medium grey indicates the individual column sums, dark grey is the total $A^T b$.

Figure 14: Submesh Organization

Arranging the solution submesh in this fashion reduces the length the data must travel to compute the column sums. The price that must be paid is some additional complexity in the routine to break the estimation region into two by two portions and move those portions to the upper-left corner of the sub-submeshes. Consider the eight by eight estimation region. This will have a solution submesh that is 32 by 32. There will be 16 estimation regions as well as 16 sub-submeshes. Figure 15 shows how the two pixel by two pixel elements of the estimation region (2,2) are moved to the sub-submeshes. Figure 15a shows the solution submesh, with the estimation region at location (2,2) highlighted. The squares within the estimation region are two by two blocks of PEs, not single PEs. Figure 15b shows which blocks must be moved to the left or the right. In Figure 15c, the blocks have been moved to the correct column, we just need to move them to the correct row. The dividing line for those blocks that will move up vs. those that will move down is also shown in this figure. Finally, Figure 15d shows the solution submesh with all two by two blocks of the estimation region moved to the upper-left corner of the sub-submeshes. Each of these blocks is nine words deep, to hold the coefficients a_1, a_2, ..., a_7, b_1, and b_2. After the blocks have been moved to the upper-left corner of the sub-submeshes, the a_1 through b_2 fields of all the other PEs have been destroyed. The second stage of the algorithm will create the **A** matrix and $\underline{b}$ vector by filling the sub-submeshes.

As mentioned above, the Pyramid has a different way of moving portions of the estimation region to the upper-left corner of the sub-submesh. The Pyramid achieves a somewhat cleaner implementation of this procedure than the flat architectures. The algorithm the Pyramid uses is presented in Appendix B as Algorithm B10. Figure 16, below, illustrates the steps of this algorithm.

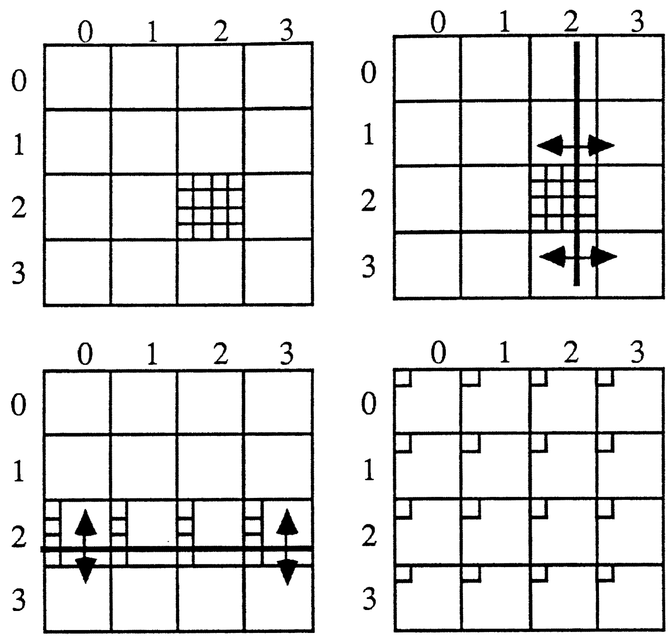
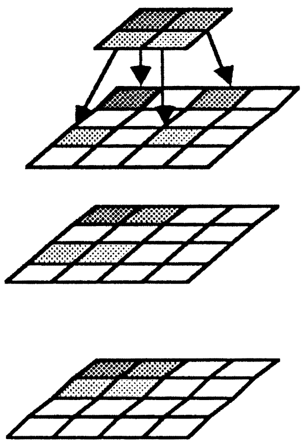


Figure 15 a,b,c,d: Partitioning the Estimation Region on the Flat Interconnection Networks



All cells with row and column LSBs = 0 read a1..b2 from parent.

If column LSBs = 01, read a1..b2 from East.

If Row LSBs = 01, read a1..b2 from South.

Repeat the procedure above, with the comparisons of the LSBs using an additional zero bit at the MS end of the LSBs.

Figure 16: Partitioning the Estimation Region on the Pyramid Interconnection Network

Now we must fill the sub-submeshes. Figure 17 illustrates this process. The initial contents of the memory fields are shown in figures 11b and 17a. Depth in Figure 17 indicates the space available in the local memory of the PEs.

The first step is to move the second column of the two by two portion of the estimation region down two PEs and left one PE. We then move the coefficients a_6, a_7, and b_2 down four PEs, and copy b_2 into the same field used by b_1 in the four top PEs. This is shown in Figure 17b. The coefficients a_2 through a_7 are then marched across the sub-submesh. As they are moved across the sub-submesh, they are also moved forward one field in the local memory, and the value that has been placed into the a_1 field is not moved to the next column of PEs. At the end of this process, the coefficients of **A** are in the a_1 field of all PEs in the first seven columns of the sub-submesh. The coefficients of $\underline{b}$ are in the b_1 field of the first column of each sub-submesh. After a final chore of clearing those elements of the a_1 field that should be zero, we are ready to compute $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\underline{b}$.

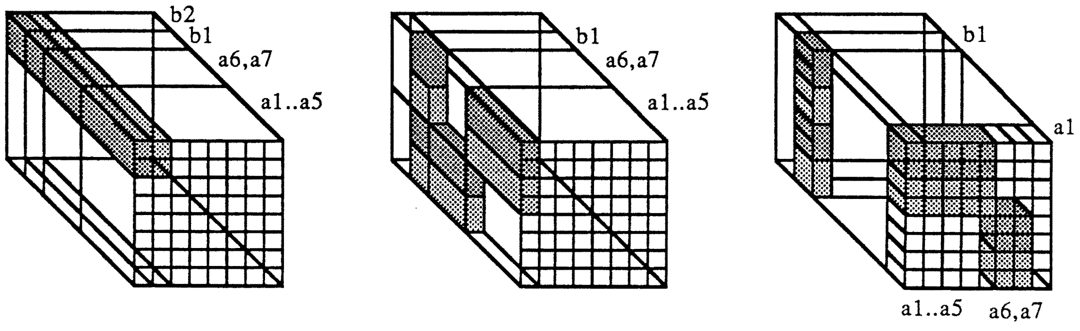


Figure 17: Filling the sub-submeshes

Calculating $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\underline{\mathbf{b}}$

The calculations of this step are simple matrix-vector and matrix-matrix multiplications. For estimation regions other than the two by two, they are complicated only slightly by the need to sum the top rows of each group of sub-submeshes once sums have been taken up the columns (see Figure 14). The matrix-vector multiply was described earlier in this chapter, when we discussed the use of a square solution submesh which was divided into sub-submeshes, and will not be repeated. The routine to multiply $\mathbf{A}^T\mathbf{A}$ is described below. The algorithms are given in Appendix B as Algorithms B5 and B6. It is not necessary for these algorithms to explicitly transpose the **A** matrix. Note that all the architectures use the same algorithms for these two operations.

Calculating $\mathbf{A}^T\mathbf{A}$ is essentially seven iterations of the $\mathbf{A}^T\underline{\mathbf{b}}$ algorithm, one for each column of the **A** matrix. For each column i of the **A** matrix, we copy that column to another field, c, of the local memory of the PEs. These values are then moved to the c field of all the other columns of the **A** matrix. Each PE then calculates c = a1 * c. The columns are then summed, but instead of the sum going all the way to the top of the solution submesh, we stop at row i. Any values above row i are summed downward so that the sum will be complete. At the end of this step, the ith row of $\mathbf{A}^T\mathbf{A}$ has been calculated. We will move this value into another field, d, of the local memory to keep it out of harm's way while the other columns are being processed. After all columns have been processed, if the dimension of the estimation region is two by two, the $\mathbf{A}^T\mathbf{A}$ matrix is in the first seven by seven submesh of the solution submesh. If the dimension of the estimation region is not two by two, we must sum the matrices at the top of each column of the sub-submeshes.

Inverting A^TA

We now must invert $\mathbf{A}^T\mathbf{A}$. To accomplish this we will use the Faddeeva technique[NASH 81, FADDE 63]. This technique uses Gaussian elimination to solve matrix equations. For a uniprocessor there are more efficient schemes to minimize the number of multiplies and divides. However, our primary emphasis is not to minimize the number of multiplies that are done, but rather to minimize the number of multiply and divide instructions that must be issued. Since we have an 8x8 solution submesh, we can do 64 of the multiplies at one time if the data can be put in place. The very regular data movements that come from Gaussian elimination make this technique well suited to the architectures studied.

The Faddeeva technique can be used to invert an n x n matrix **M** by forming the augmented matrix:

$$\begin{array}{c|c} \mathbf{M} & -\mathrm{I} \\ \hline \mathrm{I} & 0 \end{array} \tag{29}$$

where the entries I and -I are the identity matrix with the appropriate sign. Gaussian elimination is used to reduce the order of **M** by one. The resulting matrix is then shifted up and to the left one PE. The identity matrices are then restored and the process repeated a total of n times. Since each step of the Gaussian elimination only uses one row or column of the identity matrices, we can invert an nxn matrix on an n+1 x n+1 submesh. Since our $\mathbf{A}^T\mathbf{A}$ matrix is seven by seven and the solution submesh is at least eight by eight, this presents no problem. The Faddeeva algorithm can be found in Appendix B as Algorithm B7.

The Faddeeva technique can be used for other matrix manipulations besides inversion. These other operations are described in [NASH 81] and [FADDE 63]. The Gaussian elimination procedure in Algorithm B7 does no pivoting. Since the $\mathbf{A}^T\mathbf{A}$ matrix is positive definite, this should not be a major problem in areas of the image that have valid superquadric parameters.

Calculating x

To obtain our vector of superquadric parameters, we must now multiply $(\mathbf{A}^T\mathbf{A})^{-1}$ and $(\mathbf{A}^T\underline{b})$. The vector $\mathbf{A}^T\underline{b}$ is the row vector in the first seven elements of the top row of the solution submesh. The matrix $(\mathbf{A}^T\mathbf{A})^{-1}$ is in the first seven by seven PEs of the submesh. This algorithm is presented in Appendix B as Algorithm B8. It is very similar to Algorithm B4, which was the $\mathbf{A}^T\underline{b}$ algorithm. However, this routine is simpler since it is not dependant upon the size of the estimation region. The top row is moved down to the other seven rows which contain $(\mathbf{A}^T\mathbf{A})^{-1}$. A single multiply instruction is issued, then the products are summed along the rows. When complete, the vector $\underline{x}$ of superquadric parameter estimates is in the first column of the top seven rows of the solution submesh.

Distributing the Results

The seven elements of the $\underline{x}$ vector have now been calculated. We now move these parameters back to the estimation region which gave rise to them. Once there, they will be stored in unused memory fields while the other estimation regions are being calculated. Since there are seven parameters, and a two by two region has four PEs, we must have two temporary fields to store the results in the two by two estimation region. For the larger estimation regions we will only need one temporary field.

The routines to accomplish this movement will depend on both the dimension of the estimation region and the type of interconnection network being used. The first step is to change the organization of the parameters from a seven by one vector to either a 2x2x2 matrix, for the two by two estimation region, or a 4x2 matrix, for the other estimation regions. These organizations of the parameters are shown below in Figure 18. The algorithm for this re-organization is presented in Appendix B as Algorithm B9. This algorithm is used for all the machines.

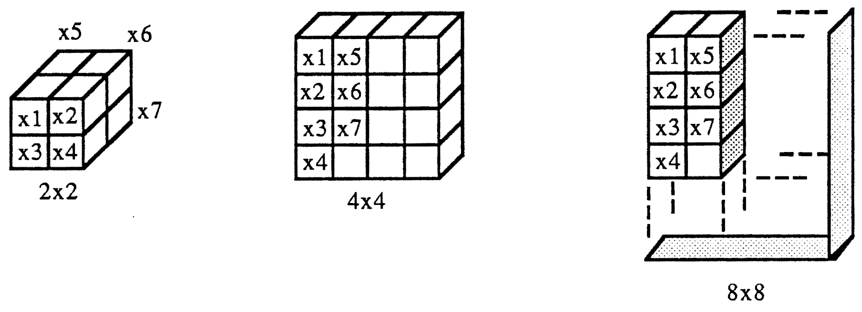


Figure 18: Organization of Parameters for Distribution to the Originating Estimation Region

The second stage of the process is to move the newly re-organized block of parameters to the estimation region from which they sprang. This process will depend upon the machine used. For the two flat machines, once the parameters have been moved back to the estimation region they must be stored in a memory location that will not be destroyed by later iterations of the superquadric recovery procedure. For the Pyramid, all estimation regions are processed at the same time, so this storage is not needed. The algorithms used are presented in Appendix B as Algorithms B10 and B11 for the flat and Pyramid machines, respectively.

The final step is to move each coefficient to a particular memory field in al! the PEs of the estimation region. This algorithm is presented as Algorithm B12 in Appendix B. This routine is used by all the architectures. For the flat machines, it will not be executed until all 16 iterations are complete. The final organization of the memory of the PEs is shown below in Figure 19.

		(ζ)	(-T)			(-Yo)	(-Xo)	
Row,Col	x1	x2	x3	x4	x5	x6	x7	Unused

Figure 19: Final Contents of Local Memories

CHAPTER IV

RESULTS

This chapter presents the performance estimates of the three interconnection networks studied. The first section presents the estimated execution times for each major stage of the least-squares procedure. The second section briefly reviews the memory requirements of the algorithms.

Execution Timings

This section presents the estimated execution time for each major stage of the least-squares procedure. In general, the times for a particular stage will be a function of the numeric precision, the dimension of the estimation region, and the particular interconnection network used. This information is presented as a series of charts. The dependant and independant axes of the charts show time vs. numeric precision. Estimation region dimension is shown with families of curves on a single chart. Three such charts, one for each interconnection network, will be presented on a page. Each stage of the procedure is shown on a seperate page. Any special assumptions used in obtaining the execution times are described in Appendix B. Some of the algorithms' timings do not depend upon all of the variables. The format of these charts is modified accordingly, adhering to the rules regarding time vs. precision and one stage per page. Short notes are included with some of the charts to explain their format, or make particular comments about the algorithm. The final charts in this section show the total time consumed, as well as the percentage of the total execution time consumed by each stage of the procedure. These charts are also functions of the numeric precision, estimation region dimension, and interconnection network.

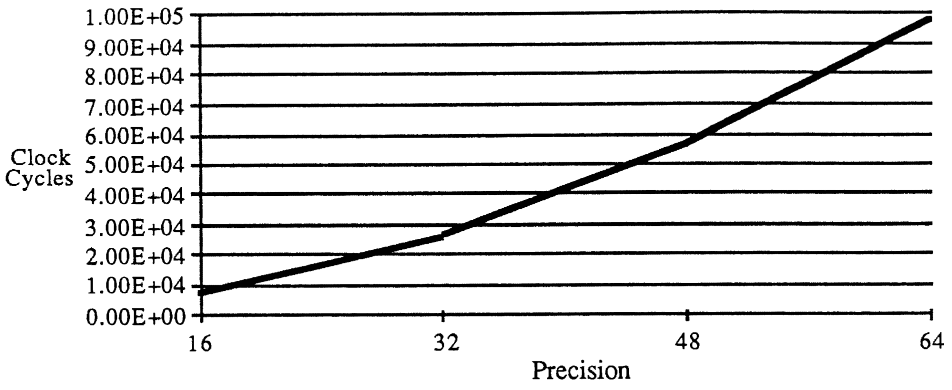


Figure 20: Calculation of Regression Coefficients

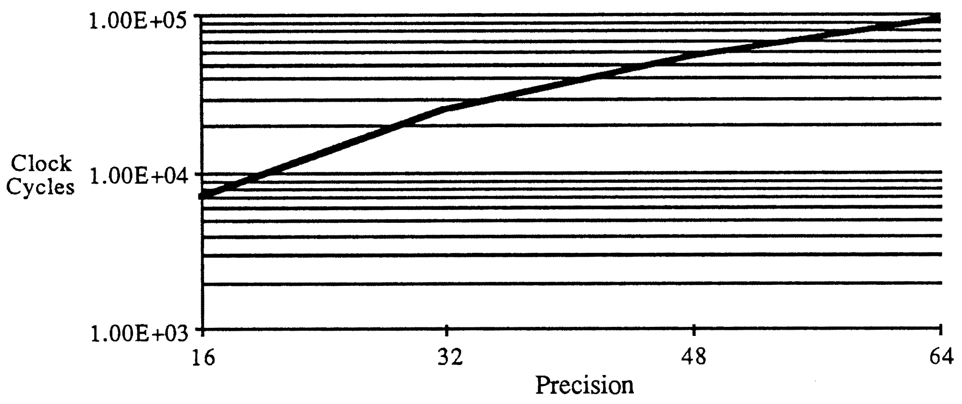


Figure 21: Calculation of Regression Coefficiénts (Log Scale)

This chart is one of the noted exceptions to the rules. The calculation of the regression coefficients is performed entirely within the PEs, so the interconnection network and estimation region dimension do not enter into the problem. To give a clearer indication of the time consumed by the lowest numeric precision, the data was replotted onto a logrithmic scale. This is shown above as Figure 21.

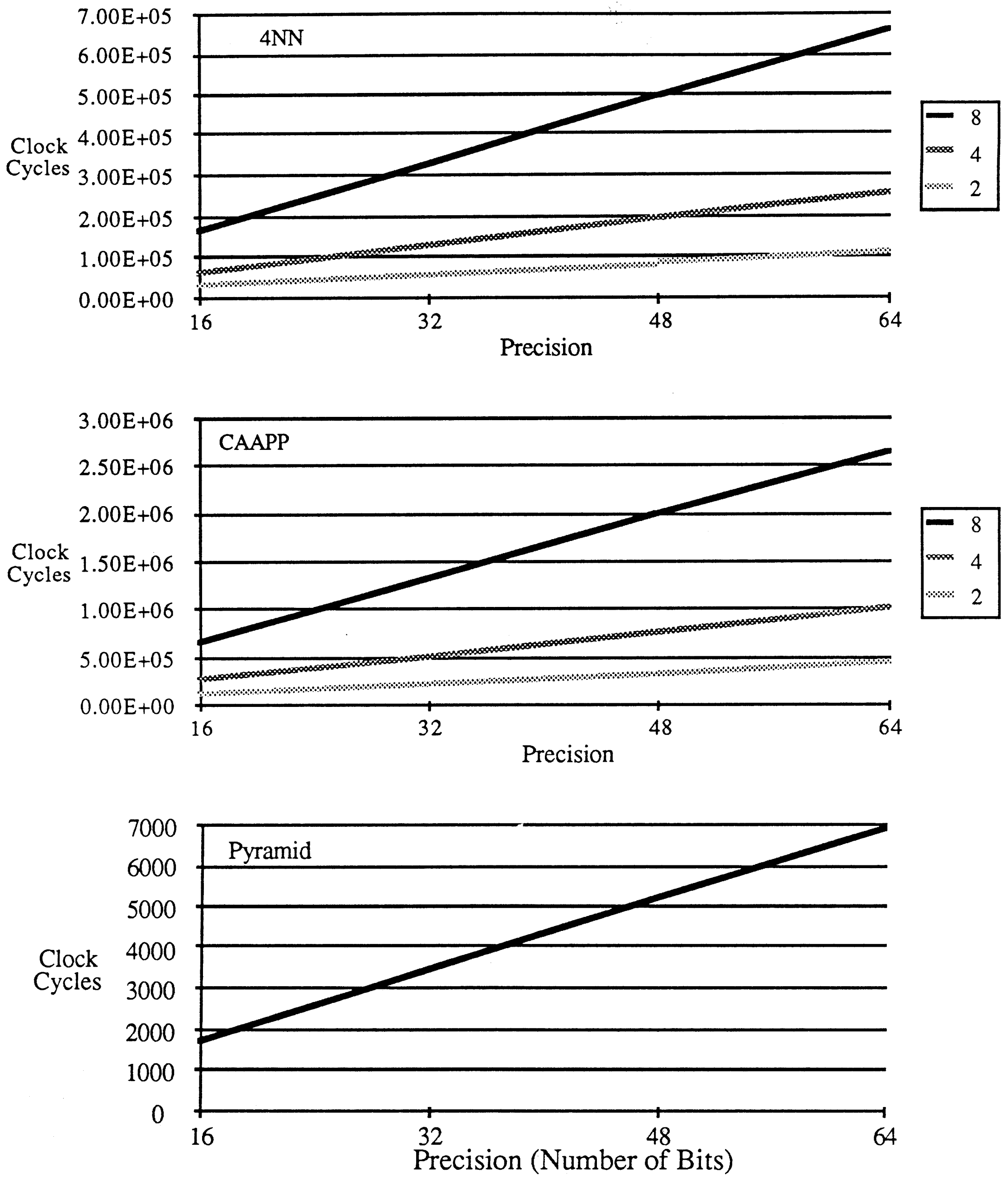


Figure 22: Distribute Estimation Region among Sub-Submeshes

Note that the Pyramid algorithm does not depend upon the dimension of the estimation region. This holds true for all estimation region dimensions that are a power of two.

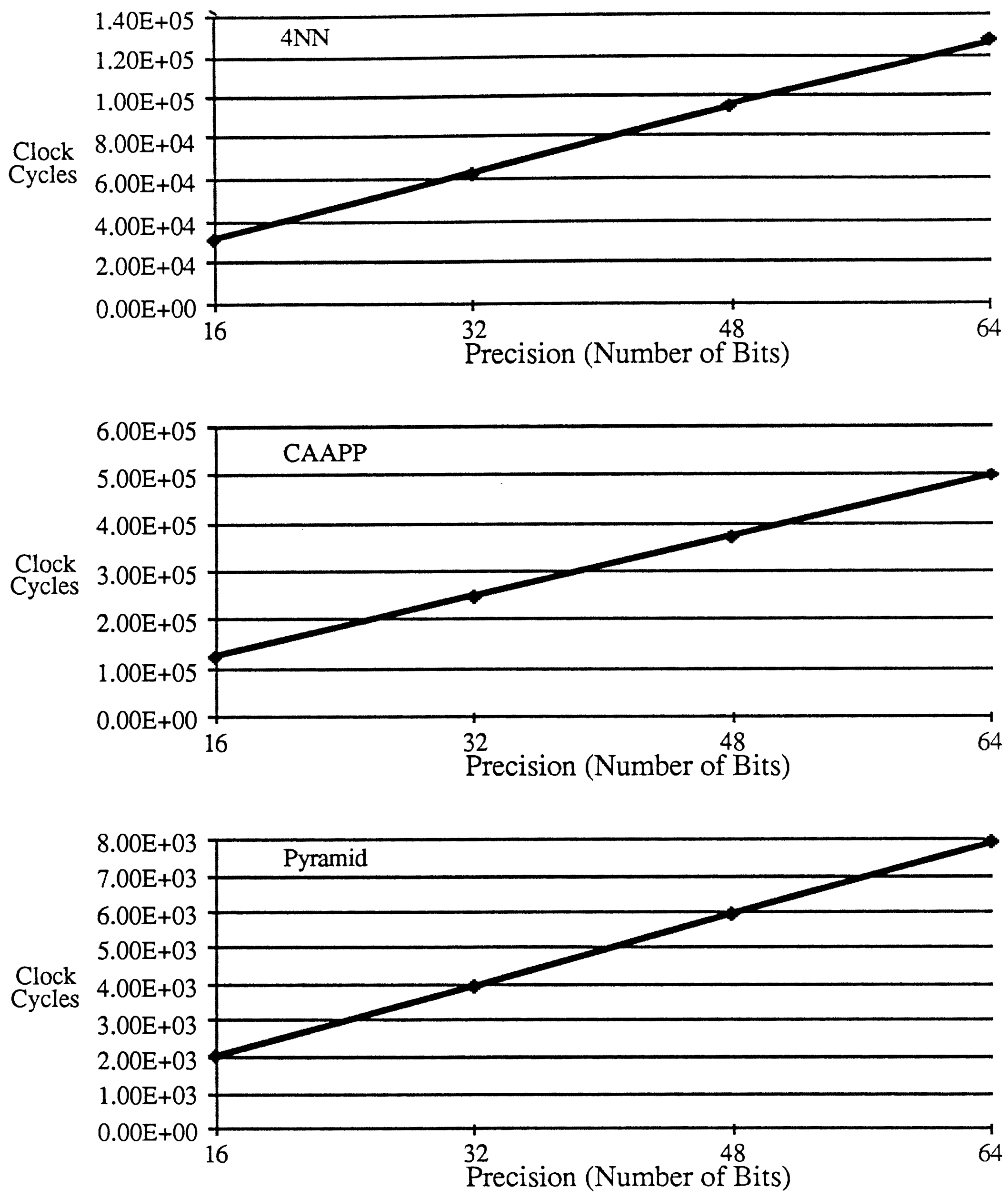


Figure 23: Filling the Sub-Submeshes

The same algorithm is used for all machines. The times for the 4NN and CAAPP interconnection networks reflect the need for 16 iterations.

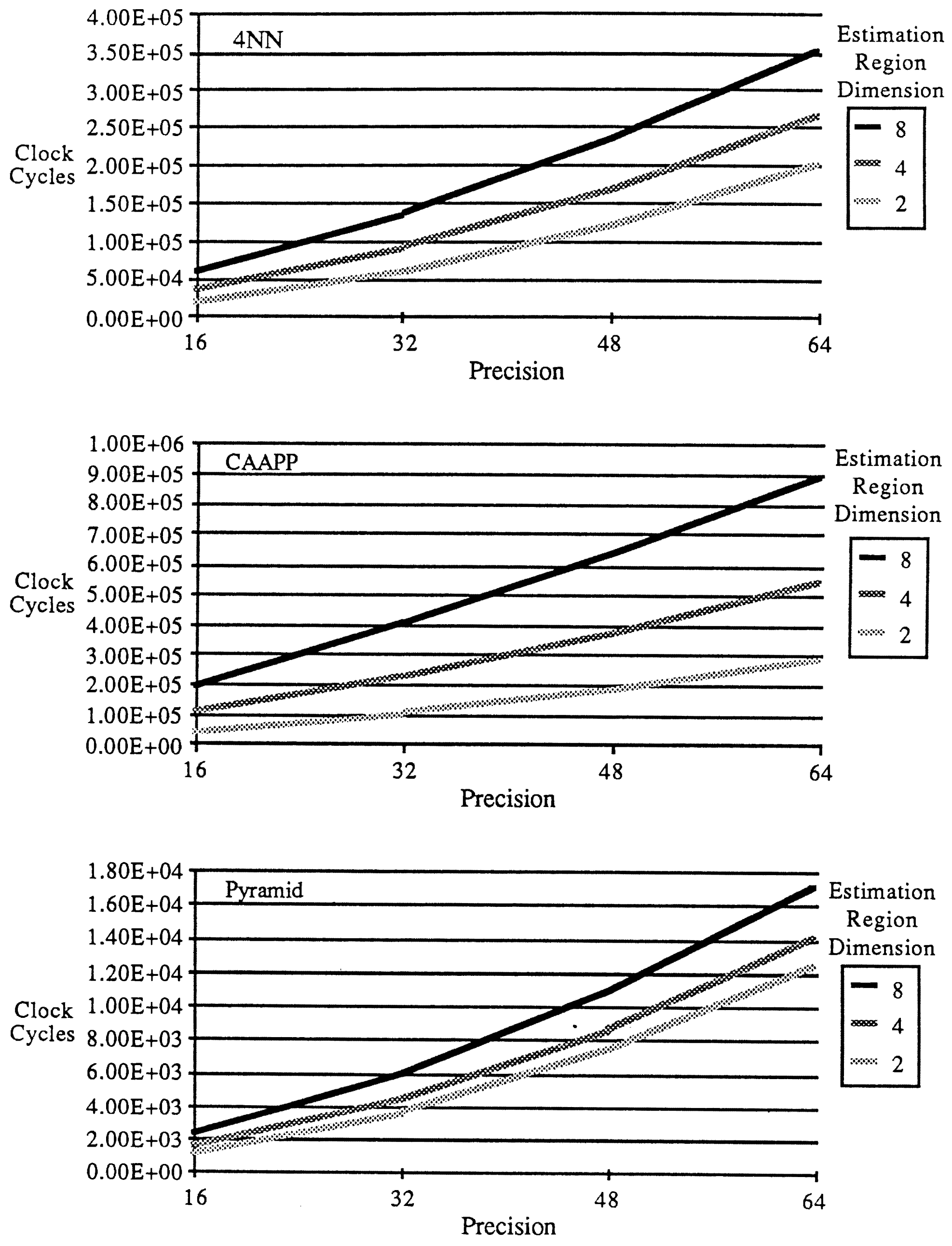


Figure 24: Calculating $A^T\underline{b}$

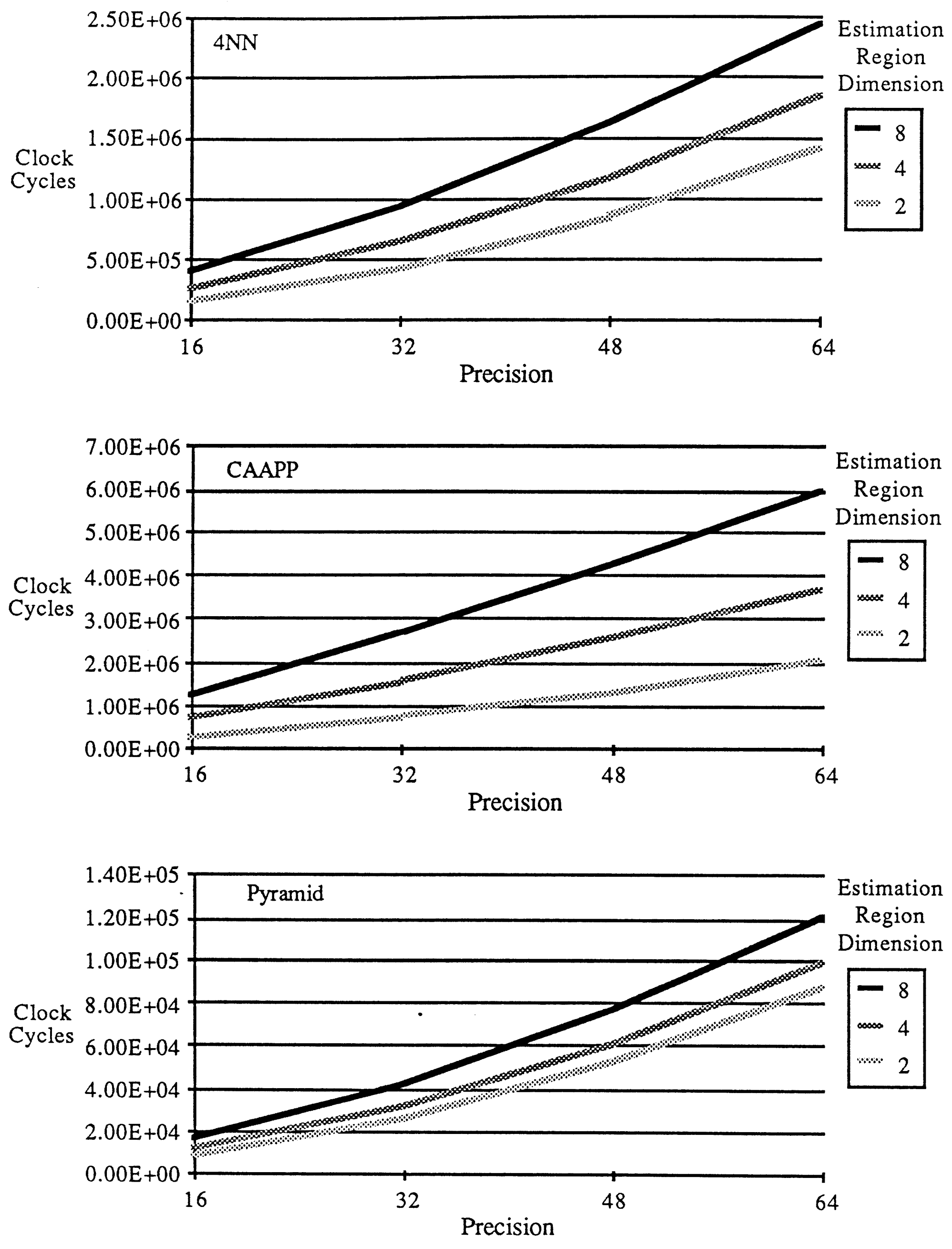


Figure 25: Calculating A^TA

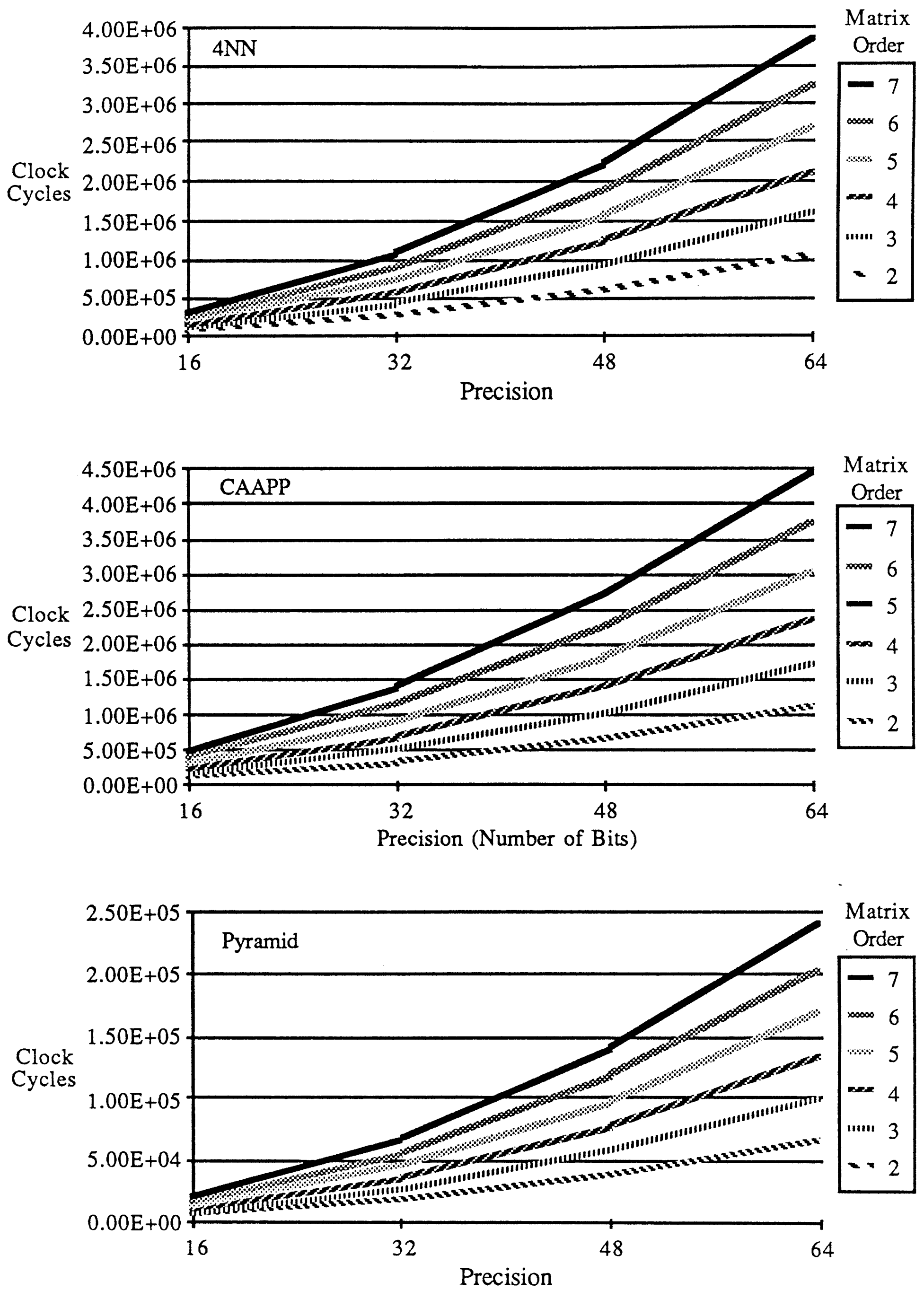
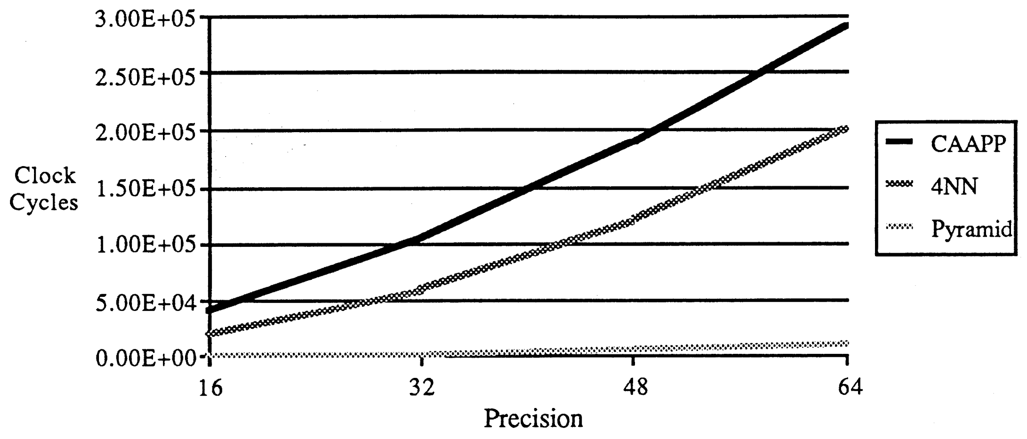


Figure 26: Inverting Matrices by the Faddeeva Technique



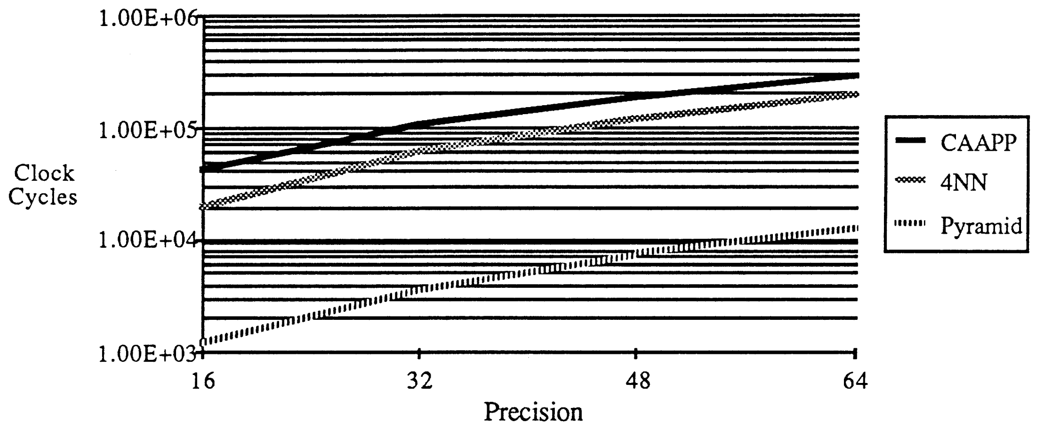


Figure 27: Calculating the Vector of Superquadric Parameters

This calculation is not dependant upon the estimation region's dimension. The first chart shows how the computation time grows with increasing precision. This figure does not show any detail of the Pyramid's execution time. The second chart shows the data replotted on a logrithmic scale to give a more accurate indication of the time for the Pyramid.

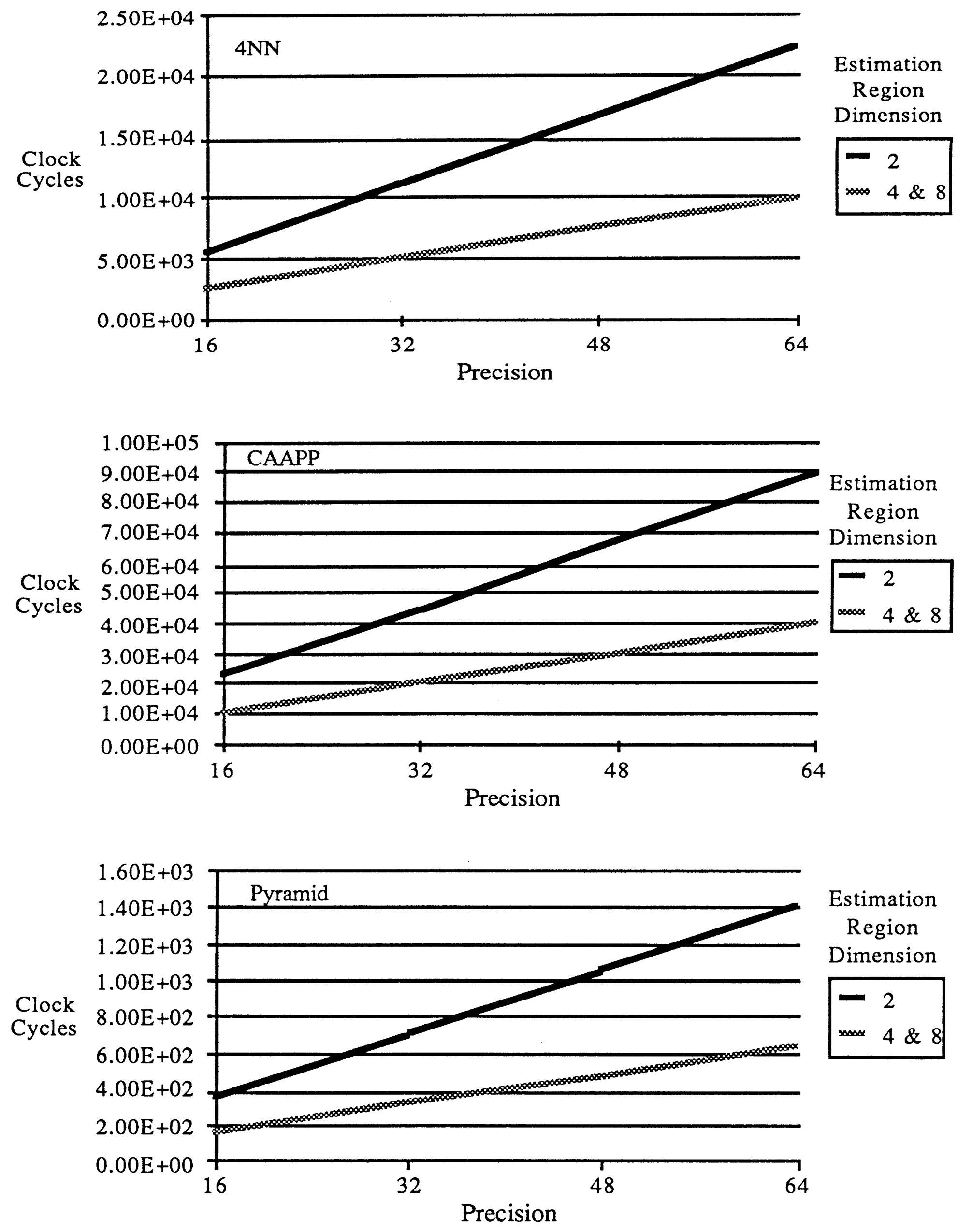
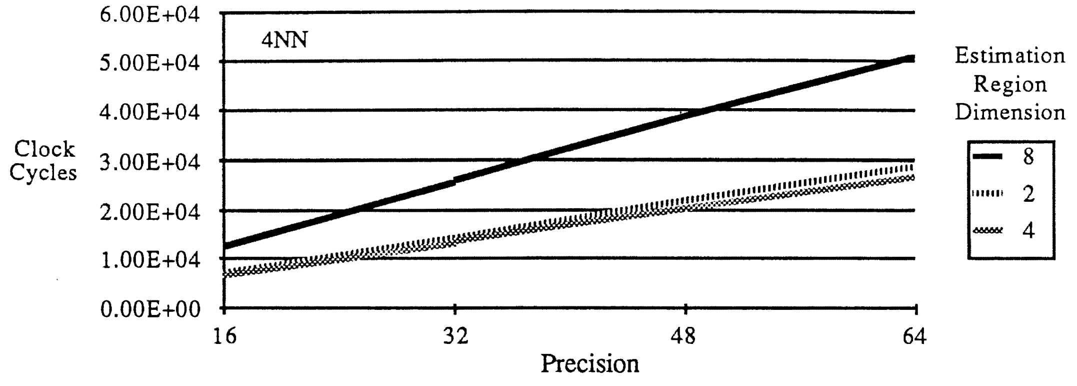
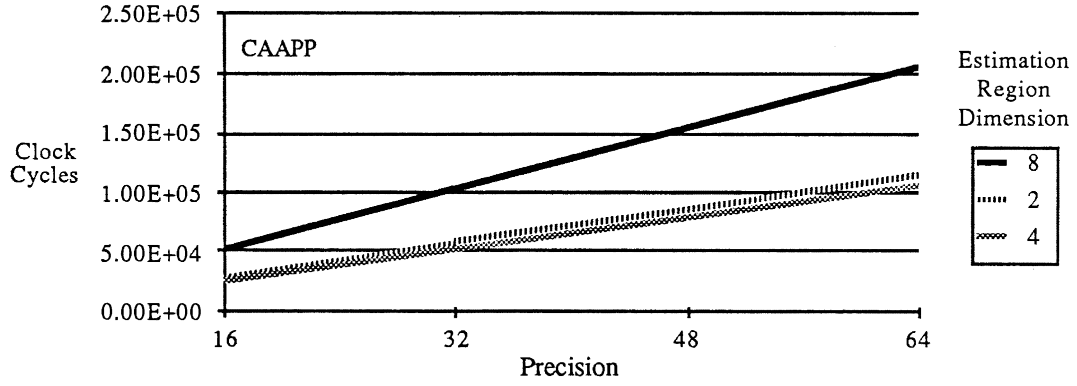


Figure 28: Reorganizing Parameters

The performance differences on these charts between the 2x2 and the other estimation region dimensions occur because of the extra copies and reads needed to form the 7x1 vector x̲ into a 2x2x2 block as opposed to a 4x2x1.





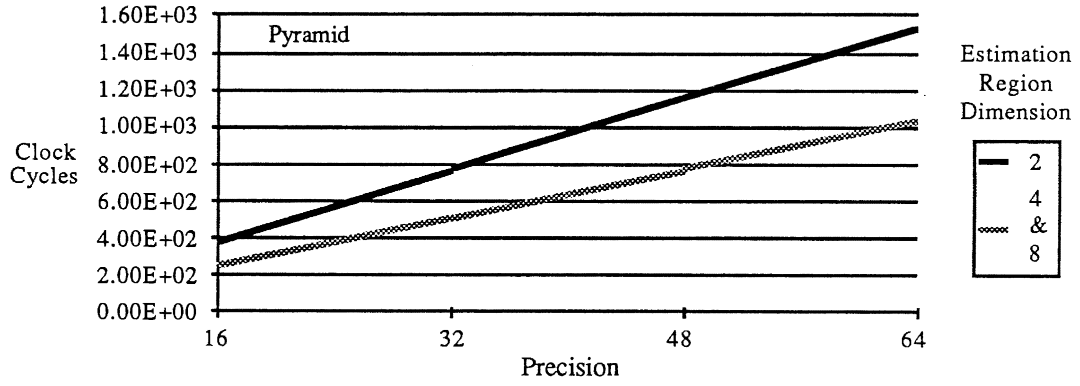
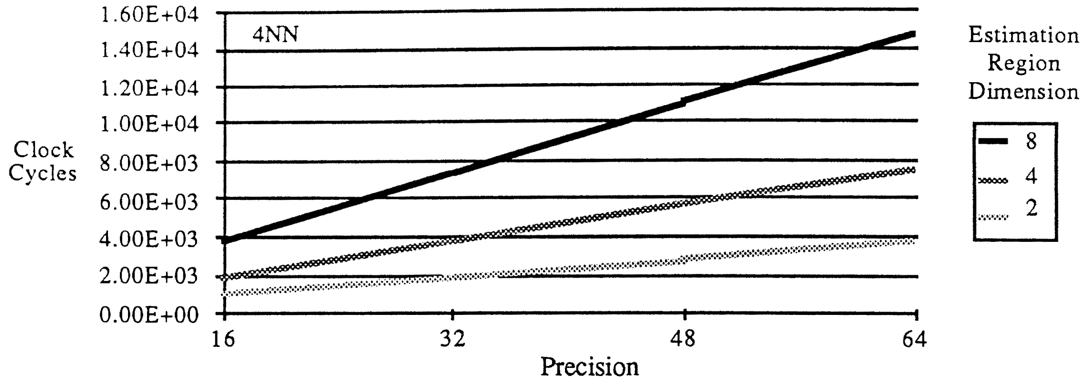
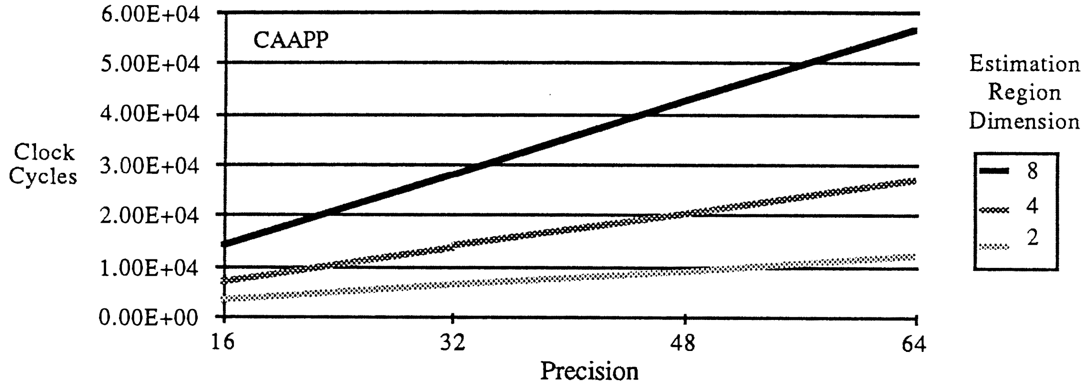


Figure 29: Moving Parameters to Originating Estimation Region





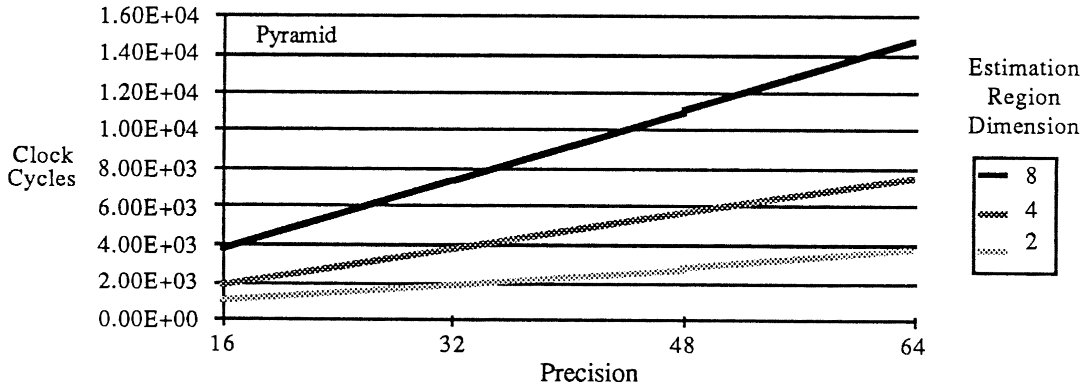
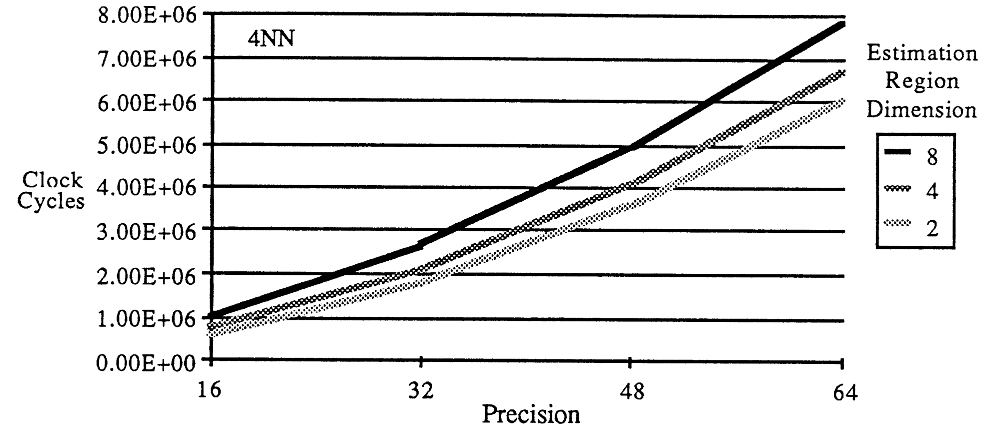
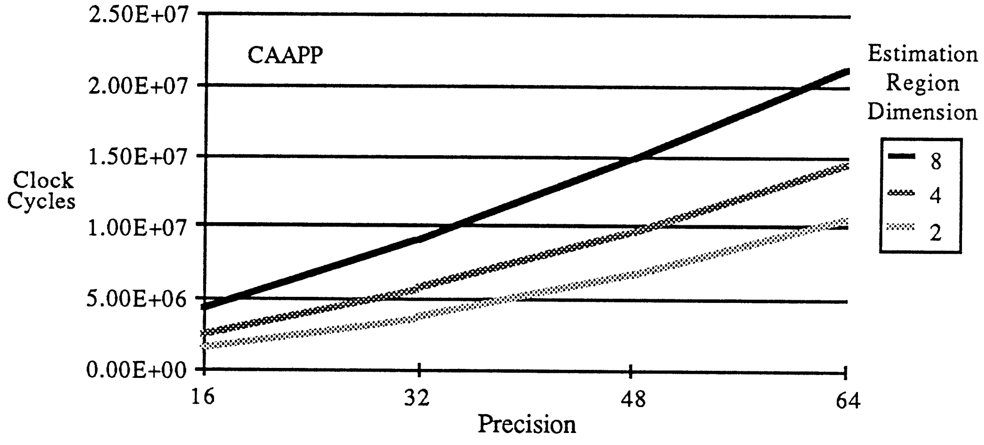


Figure 30: Filling Estimation Region with Parameters





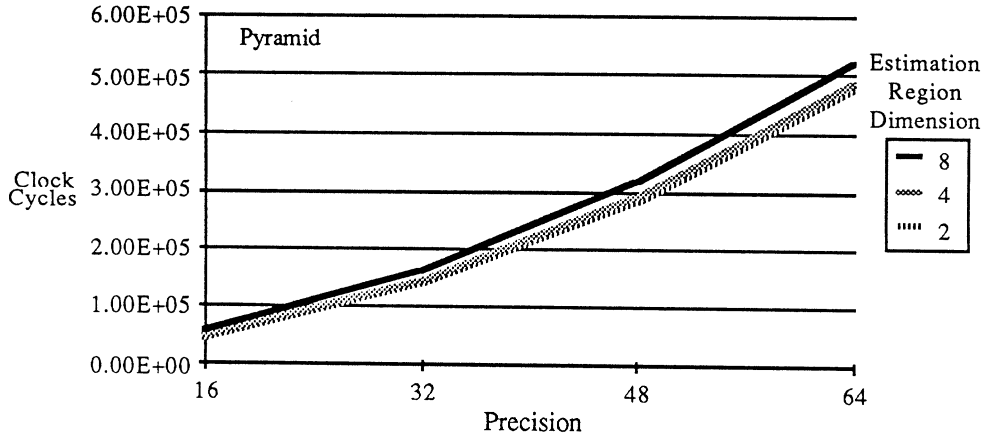
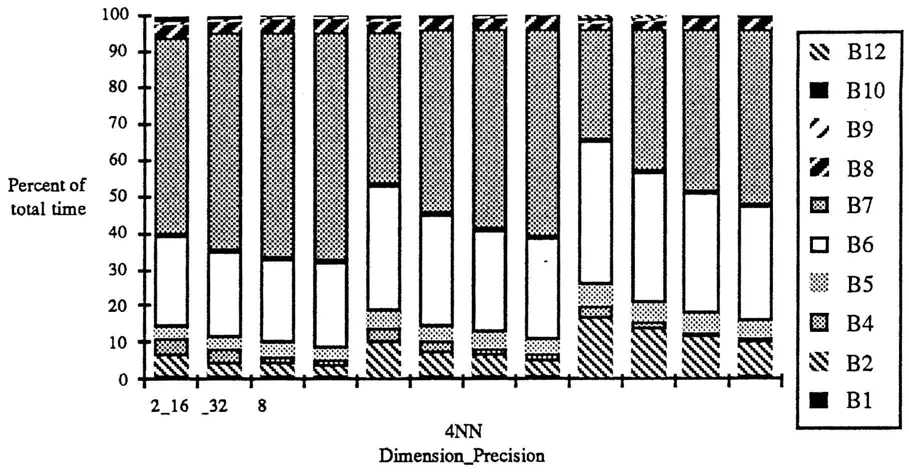
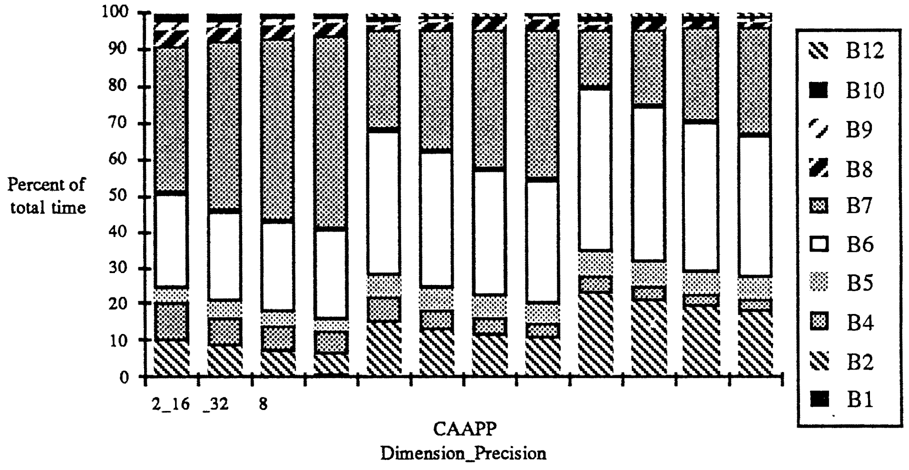


Figure 31: Total Execution Time





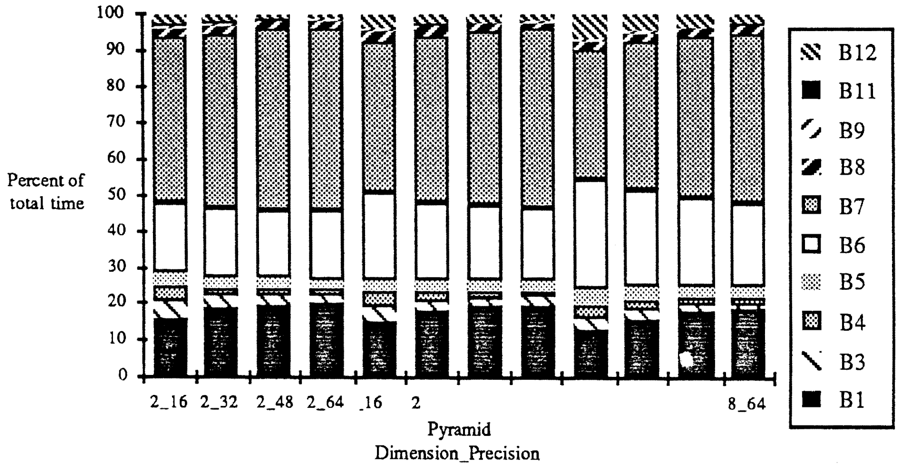


Figure 32: Percentages of Total Time

Memory Requirements

The superquadric procedure developed is fairly conservative in its memory use. By writing over data fields when they are no longer needed, all the calculations can be accomplished within the nine fields originally needed to hold the regression coefficients in each PE. We also need to allocate space to store the PE's location within the array. Temporary storage will be required on the flat machines to hold the parameters while the other estimation regions are being processed. The memory needs of each interconnection network are shown below, under differing conditions of precision, architecture, and estimation region dimension. Note that the Pyramid's memory needs are not functions of the estimation region dimension.

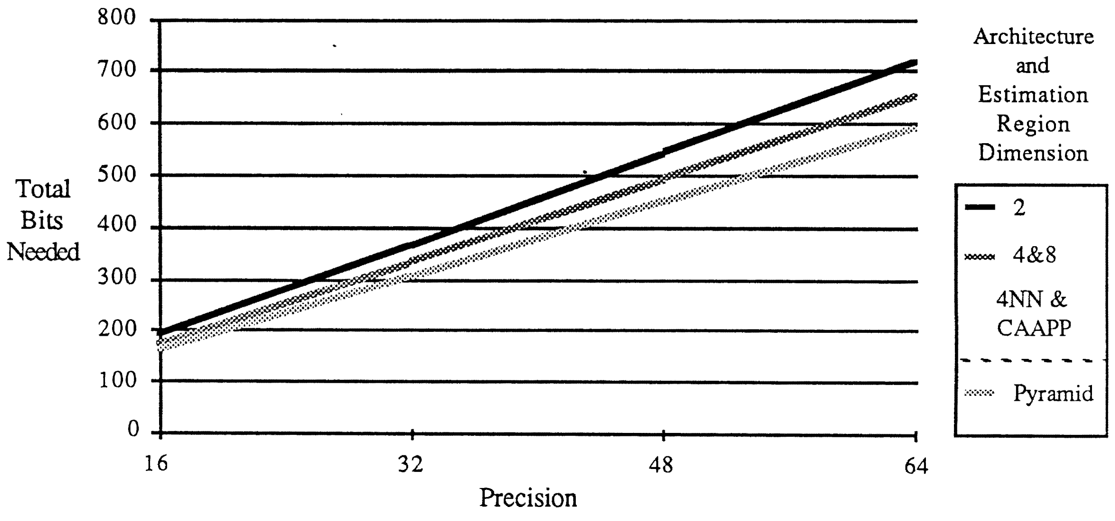


Figure 33: Memory Requirements

CHAPTER V

CONCLUSIONS

Summary

This study investigated the parallel execution of a new computer vision technique, superquadric description. This technique attempts to model imaged scenes as CSG collections of superquadric primitives. Algorithms were obtained to estimate the parameters of the primitives, given a map of the surface normals throughout the image. These algorithms are intended to execute on large arrays of bit-serial processors, which are interconnected in one of three fashions. The execution times were estimated for each of the three interconnection networks, and presented in graphical form in the preceding chapter. These times are functions of the interconnection network as well as the numeric precision and the dimension of the estimation region. The memory requirements of the algorithms, as well as some aspects of the actual implementation of the interconnection networks were also estimated.

Conclusions

Suitability of the Interconnection Networks

All of the interconnection networks studied seem well-suited to the execution of the superquadric estimation algorithm. Assuming a 10 Mhz. clock frequency, even the slowest machine, using the most precise numeric representation considered, achieves processing times on the order of a few seconds. The Pyramid, using the procedure described in Chapter Three and a reasonable precision (32 bits), achieves speeds on the order of one frame time. The 4NN is roughly twice as fast as the CAAPP. The Pyramid is approximately 16 times faster than the 4NN interconnection network, while only requiring 4/3 as many PEs. However, it should be noted that the calculations are not identical. The Pyramid algorithm does not obtain the superquadric parameters for the lowest two levels. Therefore, the resolution of its solution is only 1/16 that of the other two machines. Most of the information that would be lost in the Pyramid algorithm would be texture information. This may not be a great handicap, since we usually will want to describe the position, orientation, and shape of physical objects that are large enough to remain visible at the third and higher levels. However, Pentland [PENTL 86b] has suggested that the fractal dimension of superquadric descriptions of textures may be a powerful technique for classifying natural objects with similar shapes. This loss of information may make the Pyramid algorithm unacceptable.

While the execution speed of the procedures is quite an improvement over a uni-processor, the cost of the architectures must also be considered. These systems will be quite large, expensive, and will consume a great deal of power. The complexity which allows such high processing rates is also an Achille's heel in terms of reliability.

Architecture Enhancements

Several enhancements are possible. One of the most obvious improvements to the PEs would be to have the communication lines selected with the source select multiplexers, rather than the function select. This would cut the time for many of the fundamental operations in half. In fact, the current version of the CAAPP [WEEMS 87] implements this enhancement.

Another enhancement concerns the limited size of the local memories. Recall that the worst case memory requirements of the superquadric procedure is 722 bits. Adding this much local memory to each PE would result in excessively large die sizes and correspondingly low fabrication yields. Segmentation needs will also extend the size of the local memories. The procedures discussed in the literature for recovery of the surface normals information require the computation of several quantities that have widespread use in segmentation. The Laplacian, gradient, difference-of-Gaussians, and directional derivative are examples. Pentland's estimator also requires the computation of a measure of texture energy. To aid in segmenting the image, we might want to keep the most significant bits of each of these computations stored somewhere in the local memory. A co-operative segmentation procedure might then be used to combine the evidence from all the sources in order to achieve a more robust segmentation [DANIE 86]. In the most recent version of the CAAPP, each PE has a double-buffered interface to a 32k-bit external memory. This would be a great benefit to execution speed. The slow disk I/O operations currently needed on the flat machines could be replaced with an essentially transparent read from memory. However, the effect on the pincount is staggering. The submesh interconnection network used in the CAAPP is not well-suited to the needs of communicating with such a backing store. The current implementation of the CAAPP uses a 4NN interconnection network, and provides a byte-wide path to the backing store. Approximately 180 pins are used in the current implementation of each CAAPP package[Weems 87]. Contrast this with the original pincount of 45. The implications of this for the Pyramid are frightening!

A third improvement would be to have two activity bits. A PE would be enabled if both bits were set. This would allow us to easily disable calculations in regions were errors, such as division by zero, have occurred.

Directions for Future Research

Enhancing the Regression Equations

The superquadric estimation technique is quite new and a great deal more work must be done before the technique can achieve widespread use. The regression equations used in this study have several problems. One major problem is that these equations assume we are viewing the superquadric form from some position on its z-axis. This is a violation of the principle of general position, making the procedures developed suitable for few applications. Other limitations of the equations are that they do not allow recovery of the e_1 shape parameter and they will not give a meaningful result in areas of the image corresponding to flat object faces. The spatial derivatives of the surface normals will be zero in such areas, making the coefficients all zero. The equations to correct the problems of general position and e_1 can be derived by following a procedure similar to the one used to derive (23), but the algebra becomes quite tedious. Pentland mentions that these equations will have 15 coefficients instead of 7, but does not present the equations themselves [PENTL 86b]. Obtaining equations that are less sensitive to areas of low curvature is a more difficult problem, in fact, it may not be possible. Another limitation of the equations is that they do not allow recovery of deformed superquadrics. The power added to a CSG modeling system that uses deformed primitives is so great that we cannot ignore the need to recover the deformation information.

Optimizing the Solution Technique

The algorithms for the least-squares solution technique are not optimized. Referring to Figure 32, we see that the matrix inversion is a bottleneck for all the machines. As presented in this paper, the Gaussian elimination technique requires n multiply and n divide instructions to be issued, where n is the order of the matrix to be inverted. Use of the single-division scheme [FADDE 63] for elimination could yield significant reductions in execution time. The use of an inversion procedure other than the Faddeeva technique also bears investigation.

It is possible to reduce the time for the inversion stage for the 4x4 and 8x8 estimation regions. Consider the 8x8 region. There are 16 estimation regions in each solution submesh. Since $\mathbf{A}^T\mathbf{A}$ is 7x7, all the estimation regions can invert their own $\mathbf{A}^T\mathbf{A}$ matrix. This would eliminate the need for 16 iterations of this procedure. The 4x4 estimation region would require four iterations instead of 16. This optimization is possible for both flat machines.

The Pyramid could benefit from a modified routine to calculate the regression coefficients. The number of multiplies needed for this stage could be cut in half by moving the coefficients down one level and arranging them to calculate several products with a single multiply being issued. This would result in 5% to 10% faster execution, depending upon the estimation region dimension and the numeric precision used.

Error Characterization

The algorithms presented in this study have not been implemented. This will be necessary to investigate several of the errors possible in this computation. The sensitivity of the superquadric estimation procedure to noise must be investigated. Image noise will corrupt the surface normals data. Since we must take derivatives of this information, the problem of noise will be compounded. As mentioned above, areas with small derivatives of the surface normals data will not give reliable estimates of the superquadric parameters. Detecting and handling such areas must be investigated. Also needing study is the numeric precision necessary. Since the time for multiply and divide instructions is proportional to the square of the precision, we must take pains to ensure that unwarrented precision is not used.

Interfacing With High Level Vision

The use of superquadric information by high level stages of the vision process has been discussed in general terms, but has not been addressed in depth. Combining related primitives into a CSG object description is the most obvious area which must be addressed. Such combination will require a technique to determine which primitives are "related". A co-operative segmentation procedure was mentioned above when we discussed the memory enhancements. Such a procedure should not be driven solely by the data from the superquadric and other low-level procedures. The ability to suggest possible interpretations for the imaged data, then use the success or failure of matching these possible descriptions with stored object models will be vital to the success of a sophisticated vision system. This work must wait upon improved methods of obtaining the superquadric parameters.

Other Architectures

This study only considered machines within one class of parallel architectures. Other machines within this class, such as the Connection Machine, should be considered in future work. Other classes of architectures should also be considered. The slow speed of multiplication and division on bit-serial processors suggests that a machine composed of a large number of DSP chips, such as the TMS 320 series, could achieve equal or greater processing rates without the need for full-custom VLSI fabrication. Such a system might offer a significant reduction in the number of chips per system. Hypercube architectures should also be considered, due to their commercial availability at "reasonable" prices.

BIBLIOGRAPHY

[BALLA 82] Dana H. Ballard and Christopher M. Brown; Computer Vision; Prentice-Hall; Englewood Cliffs, N.J.; 1982.

[BARR 81] Alan H. Barr; "Superquadrics and Angle Preserving Transformations"; IEEE Computer Graphics and Applications; Vol 1, No. 1, Jan. 1981; pp. 11-23.

[BARR 84] Alan H. Barr; "Global and Local Deformations of Solid Primitives"; Computer Graphics; Vol. 18, No. 3; July 1984; pp. 21-30.

[CHARN 86] Eugene Charniak and Drew McDermott; Introduction to Artificial Intelligence; Addison-Wesley; Reading, Mass.; 1986.

[DANIE 86] Ron Daniel Jr.; "Early Vision Processing on the Content Addressable Array Parallel Processor"; Unpublished Class Report; 1986.

[DYER 81] Charles R. Dyer; "A VLSI Pyramid Machine for Hierarchical Parallel Image Processing"; Procedings of PRIP '81: The IEEE Conference on Pattern Recognition and Image Processing; Dallas, Texas; August 1981; pp. 381-386.

[FADDE 63] D.K. Faddeev and V.N. Faddeeva; Computational Methods of Linear Algebra; R.C. Williams,trans.; W.H. Freeman and Company; San Francisco and London; 1963.

[FERRI 85] F.P. Ferrie and M.D. Levine; "Piecing Together the 3-D Shape of Moving Objects: an Overview"; Proceedings IEEE Conference on Computer Vision and Pattern Recognition; San Francisco; 1985.

[FISCH 78] Martin A. Fischler; "On the Representation of Natural Scenes"; In A. Hanson and R. Riseman (Eds.), Computer Vision Systems; Academic Press; New York; 1978.

[FOSTE 76] Caxton C. Foster; Content Addressable Parallel Processors; Van Nostrand Reinhold; New York; 1976.

[HANSO 78] Allen R. Hanson and Edward M. Riseman;"VISIONS: A Computer System for Interpreting Scenes"; In A. Hanson and R. Riseman (Eds.), Computer Vision Systems; Academic Press; New York; 1978.

[HARAL 81] R.M. Haralick and L. Watson; "A Facet Model for Image Data"; Computer Graphics and Image Processing; Vol. 15; 1981; pp.113-129.

[HARAL 82] R.M. Haralick; "Zero Crossing of Second Directional Derivative Edge Operator"; SPIE vol. 336; Robot Vision; 1982; pp. 91-99.

[HORN 75] B.K.P. Horn; "Obtaining Shape From Shading Information" in P.H. Winston (Ed.) Psychology of Computer Vision; McGraw-Hill; New York; 1975.

[LEVIT 84] Steven P. Levitan, et al; "Signals to Symbols: Unblocking the Vision Communications/ Control Bottleneck"; in VLSI Signal Processing; Peter Cappello, ed.; IEEE Press; 1984.

[LAWTO 84] Daryl Lawton, et al; "Iconic to Symbolic Processing Using a Content Addressable Array Parallel Processor"; SPIE Vol. 504; Applications of Digital Image Processing VII; pp. 92- 111; 1984.

[LEE 85] Chia-Hoang Lee and Azriel Rosenfeld; "Improved Methods of Estimating Shape From Shading Using the Light Source Coordinate System"; Artificial Intelligence; Vol. 26; Elsevier Science Publishers (North-Holland); 1985; pp. 125-143.

[MARR 78] D. Marr and H.K. Nishihara; "Representation and Recognition of the Spatial Organization of Three Dimensional Shapes"; Proc. Royal Soc. London B 200; pp. 269-294; 1978

[NASH 81] J.G. Nash, et al; "VLSI Processor Arrays for Matrix Manipulation"; In H.T. Kung, B. Sproull, G. Steele (Eds.), VLSI Systems and Computations; Computer Science Press; Rockville, Md.; 1981.

[PENTL 84] Alex P. Pentland; "Local Shading Analysis"; IEEE Trans. Pattern Analysis and Machine Intelligence; vol. PAMI-6; pp. 170-187; 1984

[PENTL 86a] Alex P. Pentland; "Shading Into Texture"; In A.P. Pentland (Ed.), Pixels to Predicates: Recent Advances in Computational and Robotic Vision; Ablex; Norwood N.J.; 1986.

[PENTL 86b] Alex P. Pentland; "Perceptual Organization and the Representation of Natural Form"; Artificial Intelligence; Vol. 28; pp. 293-331; 1986.

[REDDY 78] Raj Reddy; "Pragmatic Aspects of Machine Vision"; In A. Hanson and R. Riseman (Eds.), Computer Vision Systems; Academic Press; New York; 1978.

[SOWA 84] J.F.Sowa; Conceptual Structures: Information Processing in Mind and Machine; Addison-Wesley; Reading, Mass.; 1984.

[TANIM 80] S. Tanimoto and A. Klinger, eds.; Structured Computer Vision: Machine Perception Through Hierarchical Computation Structures; Academic Press; New York; 1980.

[TANIM 84] Steven L. Tanimoto; "A Pyramidal Approach to Parallel Processing"; Proceedings of the 11th International Symposium on Computer Architecture; IEEE Computer Society Press; pp. 372-378; 1984.

[YANG 83] Yee-Hong Yang and Tsung-Wei Sze; "An Evaluation Study of Six Topologies of Parallel Computer Architectures for Scene Matching"; Proceedings of the 1983 International Conf. on Parallel Processing; pp. 258-260; 1983.

[WEEMS 85] Chip Weems, et al; "Iconic and Symbolic Processing Using a Content Addressable Array Parallel Processor", Proc. IEEE Conf. Computer Vision and Pattern Recognition; San Francisco; 1985.

[WEEMS 87] Charles C. Weems; personal communication.

[WESTE 85] Neil H. E. Weste and Kamran Eshraghian; Principles of CMOS VLSI Design; Addison-Wesley; Reading, Mass.; 1985.

[WITKIN 81] A.P. Witkin; "Recovering Surface Shape and Orientation From Texture"; Artificial Intelligence; vol. 17; pp.17-45; 1981.

APPENDIX A

FUNDAMENTAL OPERATIONS ALGORITHMS

The performance estimates presented in Chapter Four were obtained by breaking the least-squares procedure into major stages. Formulas for the execution time of the major stages were written as functions of the time for fundamental arithmetic and data-transfer operations. The formulas for the fundamental operations will be functions of the architecture and the numeric precision of the operands. This appendix presents the algorithms and formulas for the fundamental operations. The algorithms and formulas for the major stages are presented in Appendix B. All the algorithms in both appendices are written in a pseudocode that most closely resembles an assembly language with high-level looping and branching capabilities. The format of this pseudocode is explained below, followed by a brief description of all the low-level algorithms called by the high level routines. Algorithms are presented for a representative subset of the fundamental operations, and their execution time characterized. The assumptions used in estimating the execution times are then described, and formulas for the execution time of the other operations are given.

Pseudocode Format

This section discusses the format of the pseudocode used to present the algorithms in this study. The presentation is rather informal. Most of the algorithms are composed of calls to microcoded subroutines that are at the level of assembly language instructions. Higher level constructs such as indexed loops and If / Then / Else statements are also used.

This mimics the organization of the machines studied, which are divided into the array and controller sections. All the microcoded subroutines used by the algorithms in Appendix B are documented in the next section of this appendix. The remainder of this section discusses the format of the algorithms and the higher level controller operations.

The format of the pseudocode reflects the division of the machines into array and controller sections. Array operations are written in upper-case, controller functions are written in lower-case. When memory operands are used, they are referred to by their least-significant bit (LSB). Most array intructions will have an argument, p, which indicates the numeric precision of the memory operands. Thus, a p-bit number referred to by 'label' occupies addresses label to label+p-1. More signicant bits are stored at higher addresses. Comments are denoted by either pairs of braces, { comment }, or from a semicolon to the end of the line. Statements can be labeled to support control transfers. Labels are the first field in an instruction, are written in lower case, and are terminated with a colon.

The microinstruction word of the PE has four fields to change certain aspects of its behavior. These are the IA (Ignore Activity), NI (Negate source I), NJ (Negate source J), and NR (Negate function Result) fields. These options can be enabled for the duration of one microcode call by the use of the prefixes IA:, NI:, NJ:, and NR:. The restriction that code labels be lowercase is to avoid ambiguity in the interpretation of the labels and prefixes. Any number of prefixes may be used for a single statement, however, it does not make sense to use the negate prefixes with most high level statements. The algorithms presented in this appendix will occasionally make use of these prefixes.

A notational convienience used with many SELECT statements is the use of 'intelligent' selections. Since the algorithms for the major stages will be functions of the estimation region dimension, the arguments "collsbs" and "rowlsbs" will automatically have their precision adjusted to the size necessary for whichever estimation region dimension is being used. For example, suppose we wish to enable the PE at location 0,0 within each solution submesh. The statement:

```
        SELECT (collsbs=0 and rowlsbs=0)
```

would compare the p least significant bits of the row and column addresses to 0. For the 8x8 solution submesh, p would be three. Four and five bits would be examined for the 16x16 and 32x32 submeshes, respectivly. The select statement supports tests other than strict equality. An example of this capability is provided by the statement:

```
        SELECT (ptr ≥ collsbs ≥ ptr-6) .
```

The capabilities of the SELECT statement are discussed more fully in the description of this statement.

The PE's registers can also be used as arguments. REGA, ... REGE are used to access the PE's registers A ... E. When a register is used in an instruction in place of a memory location, it is not necessary to provide the number of bits that will be moved, since a register holds only one bit.

A simple 'for, next' construct is used for looping. This is a controller function, therefore written in lower case. The loop statements also allow the use of more than one index variable. An example of a statement using two index variables is:

```
        for (i=srcindx,j=dstindx) to (i=srcindx+n,j=dstindx+n).
```

'If' statements are also provided. These are also controller functions, which compare variables the controller has access to without involving the array. This differs from the SELECT statement, which uses values in the PE's local memory. Thus the SELECT is an array function, the 'if' is a controller function.

Description of the Low Level Procedures

All the low level routines that are called by the algorithms for the major stages of the least-squares procedure are described in this section. The name of each procedure is given, as well as a list of its arguments. This is followed by a description of the procedure and its arguments. Some procedures are minor variations of others. Such routines are grouped together to avoid needless repitition of the argument descriptions. Examples of this are the addition and subtraction procedures. Since the numbers are stored in a two's complement format, the subtraction routines bear a strong resemblence to the additon instructions. Three varieties of the add and subtract instructions are provided. The first is to add the values from a particular memory field of neighboring PEs. The other two implement two and three address addition within the local memory of a PE. All six procedures are described in the same section.

SELECT(p,field-indx,test,comparand,dir)

DESCRIPTION

Compares the p bit comparand with the memory locations field through field+p-1. By default the comparision procedes from the least significant to the most significant bits, but the dir operand allows this to be reversed. Tests supported are =, ≠, >, <, ≥, ≤. More complex testing opertions such as between, not between, etc. can be implemented. See [FOSTE 76] for further information in this area.

OPERANDS

p	Number of bits to be compared.
field-indx	LSB of the memory field that is compared to the broadcast pattern.
test	Type of comparison function. The algorithm presented later in this appendix is for strict equality. Other tests are also possible, see [FOSTE 76].
comparand	Bit pattern known by controller. This is broadcast to all PEs, which compare their values with the pattern and store the result of the comparisons in the A register.
dir	By default, the comparison procedes from the LSB to the MSB. This can be changed by specifing 'M' as the dir argument.

COPY(p,srcindx,dstindx)
READ(p,srcindx,dstindx,dir)

DESCRIPTION

Copy is used to move data from one memory field to another within the same PE. The memory fields should not overlap. This restriction would be quite easy to overcome, but the superquadric algorithms do not need to copy to overlapping fields. READ is used to transfer information from one PE to another. Memory fields for the READ may overlap. If a register is used as the source or destination operand, p is not specified since it must be equal to one.

OPERANDS

p	The number of bits to be copied or read.
srcindx dstindx	The LSB of the source and destination fields, respectively. Can also be a register.
dir	Tells the read instruction which neighbor to read the data from. No dir is needed if srcindx is BC (the Broadcast Comparand line).

SHIFT(p, n, dir, sign, srcindx, reg)

DESCRIPTION

Shifts the p bits specified by srcindx n places. The direction of the shift is specified by dir. Valid arguments for sign are 'Y' and 'N'. These do and do not perform sign extension on the bit field, respectively.

OPERANDS

p	Width of the field to be shifted.
n	Number of bit positions the field is shifted by.
dir	Direction of the shift, L (left) or R (right).
sign	Indicates if sign bit should be preserved. Y (preserve sign) or N (don't preserve sign).
srcindx	LSB of the p-bit field to be shifted.
reg	Register to use for temporary storage. REGE is default.

ADD2(p,srcindx,dstindx)
SUB2(p,srcindx,dstindx)
ADD3(p,src1indx,src2indx,dstindx)
SUB3(p,src1indx,src2indx,dstindx)
ADDN2(p,src1indx,dstindx,dir)
SUBN2(p,src1indx,dstindx,dir)
ADDN3(p,src1indx,src2indx,dstindx,dir)
SUBN3(p,src1indx,src2indx,dstindx,dir)
NEG(p, indx)

DESCRIPTION

These instructions perform bit-serial addition and subtraction. The procedures assume that the operands are p-bit, fixed-point, two's complement numbers. NEG changes the sign of its argument, the operation is performed in place. The difference between the ADD and SUB procedures is that the ADDs initially set the carry to zero, the SUBs set the carry to one. The SUB also negates the source operand before the addition is performed. The combination of these two operations negates the source operand on the fly.

The numbers in the procedure name indicate how many seperate addresses are used. For example, ADD2 has two memory operands, src and dst. The operation performed is dst = dst + src. ADD3 has three operands, src1, src2, and dst. The operation performed is dst=src1+src2. For subtraction, the operations are dst=dst-src or dst=src1–src2. If the procedure name has an N in it, the operation involves neighboring PEs. The particular neighbor that will provide the source operand is specified by using 'N', for North, 'E", for East, etc. The E register is used for communication in the neighbor algorithms. The other procedures use the E register for temporary storage. All procedures use the C register for the carry bit.

OPERANDS

p	Numeric precision in bits.
srcindx (src1indx) (src2indx)	LSB of the source operand(s)
dstindx	LSB of the destination operand
dir	For operations involving neighboring PEs, dir specifies from which neighboring PE to obtain the source operand. For the flat machines, valid arguments are 'N', 'E', 'S', 'W'. The Pyramid uses these, as well as 'NE', 'SE', 'NW', 'SW', 'NEC', 'SEC', 'NWC', 'SWC', and 'P' to provide communication with the lateral and pyramidal neighborhoods.

MUL(p,src1indx,src2indx,prodindx,scratch)
DIV(p,dvdindx,dvrindx,qindx,rindx, scratch)
AFMUL(p, ip,src1indx,src2indx,dstindx,scratch)

DESCRIPTION

These routines provide fixed point multiplication and division. MUL performs signed multiplication of two p-bit, fixed-point numbers. The product is also p-bit, scratch storage is used to build the product which is then reduced to p bits. AFMUL multiplies an integer by a fixed-point number and returns a fixed-point product. This routine is included for multiplying row, column coordinates by fixed-point numbers.

OPERANDS

p	Precision of the fixed-point numbers, in bits
ip	Precision of the integer number. Only used in AFMUL.
src2indx	Address of the LSB of the two source operands.
prodindx	Address of the LSB of the product.
dvdindx	Address of the LSB of the dividend
dvrindx	Address of the LSB of the dividend
qindx	Address of the LSB of the quotient.
rindx	Address of the LSB of the remainder.
scratch	Address of the LSB of a 2p block of memory used for working space.

An additional function used by the main procedure for the flat machines is image I/O. The two routines below provide the capability for image I/O through the border register. This is used by the main algorithm for the flat architectures to save the regression coefficients before processing the first estimation region, and to restore the regression coefficients at the begining of each of the remaining 15 iterations.

IMAGEIN(p, dstindx)
IMAGEOUT(p, srcindx)

DESCRIPTION

Uses the border register for high speed image I/O. The elements of the image are saved or restored one bit-plane at a time. The routines assumes that the device(s) connected to the border register can fill the border register on every clock cycle of the array. For the 4NN and Pyramid machines, this is a data rate of five megabits per second. The CAAPP has one fourth as many elements in the border register, thus its data rate is one fourth that of the other machines.

OPERANDS

p	The number of bitplanes that must be input or output.
srcindx dstindx	The address of the LSB to be output or input.

Representative Algorithms

This subsection presents algorithms for a representative sampling of the low level procedures. The next subsection presents the execution time formulas for all the algorithms.

Algorithm A1: SELECT(p,fieldindx,test,comparand,dir)

{ Compares the p bits, fieldindx to fieldindx +p-1, to the pattern which is broadcast by the controller over the BC line. If the test is passed, the A register will be set, otherwise it is cleared. If IA is specified when the select instruction is issued, it will only be used for the first bit comparison. By default, the comparison procedes from the LSB to the MSB. This can be reversed by specifing 'M' for the dir operand. The algorithm below tests for strict equality, other tests are possible. See [FOSTE 76] for other algorithms. SInce the purpose of this algorithm is more to show the flavor of the SELECT statement than to provide pseudocode ready to microcode, no use will be made of the 'test' operand. Since this procedure does no communication, it is the same for the CAAPP as for the other machines. }

```
        if dir ≠ M
(IA: ?)         CMP fieldindx[0] with comparand[0]    ; The (IA: ?) is for conditional use of
                for (i=1,j=1) to (i=p-1,j=p-1)        ; the IA: option.
                    CMP fieldindx[i] with comparand[j]
                next (i,j)
        else
                i=p-1, j=p-1
(IA: ?)         CMP fieldindx[i] with comparand[j]
                for (i=p-2,j=p-2) downto (i=0,j=0)
                    CMP fieldindx[i] with comparand[j]
                next (i,j)
        endif
```

Algorithm A2: READ(p,srcindx,dstindx,dir)

{ Reads p bits from dir neighbor's srcindx field to dstindx field of the issuing PE. If the IA option is given, all PEs perform the operation. If it is not specified, only those PEs who's A register is set will perform the operation. All E registers are changed, even if the IA option is specified. This routine will need to be changed for the CAAPP to handle the reduced connections between PEs. The time for the CAAPP will be four times as long as for the other two machines. }

```
for (i=srcindx,j=dstindx) to (i=srcindx+p-1,j=dstindx+p-1)
        IA: move memory[i] to register E
            READ from dir-neighbor into memory[j]
next (i,j)
```

Algorithm A3: ADD2(p,srcindx,dstindx)

{ Adds two p-bit, two's-complement, fixed-point numbers. Two address addition is performed, i.e. dstindx = dstindx+srcindx. Since this procedure does no communication, it is the same for the CAAPP as for the other machines. }

```
READ(1,REGC,BC)                          ; Broadcast a zero to clear the carry,
for (i=0,j=0) to (i=p-1,j=p-1)           ; then add the bits.
        IA: move srcindx[i] to register E
        ADD REGE, dstindx[j]
next (i,j)
```

Algorithm A4: MUL(p,src1indx,src2indx,dstindx,scratch)

{ Multiplies the two p-bit, fixed-point, two's-complement numbers. The multiplicand is src1, src2 is the multiplier. The product is stored into the p-bit field with LSB=dstindx. Since the product of two p-bit binary numbers is a 2p-bit number, the scratch field is provided to accumulate the product. Once the calculation is complete, the middle p bits of the product are copied to the destination field. Since this procedure does no communication, it is the same for the CAAPP as for the other machines. }

```
COPY mem[src2indx+p-1] to REGB           ; Keep the sign bits handy for
COPY mem[src1indx+p-1] to REGD           ; corrections.
for i= scratch to scratch + 2p -1
        COPY '0' to mem[i]               ; Zero the scratch area.
next i
for i = 0 to p-1                         ; For each multiplier bit,
        SELECT (1, mem[src2indx+i], =, 1) ; if it is a '1', add the
        ADD2(p, src1indx, scratch+i)     ; multiplicand to the partial product.
        SELECT( REGB=1)                  ; If the multiplier is negative,
        ADD2(p-i, scratch+2p-1-i, 1)     ; do the correction.
next i
SELECT( 1, REGD, =, 1)                   ; If multiplicand is negative,
SUB2(p, src1, scratch+p)                 ; correct for that.
COPY(p, dst, scratch+p/2)                ; Copy p-bit product to destination.
```

Algorithm A5: IMAGEIN (p,src1indx or dstindx)

{ Uses the border register to move large amounts of data into the arrray. One bit plane is moved at a time. Assumes the border register can be filled or emptied as fast as the array can accept or provide the data. Also assumes we have configured the edge-treatment switches so that the border register is the southern neighbor of the PEs on the bottom of the array. The algorithm below does image input, output is a simple modification of this procedure. For the CAAPP, the inner loop must execute four times for each row of the array. The PEs will read data from their eastern neighbor instead of their souther neighbor. }

```
IA: READ( REGA, BC)                  ; BC broadcasts '1', enables all PEs.
for i = 0 to p-1                     ; for each bit plane
    for j = 1 to 512                 ; for each row of bits to be read
        READ  ( S, REGE)
    next j                           ; Register E of each PE holds the bit
    COPY (REGE, mem[dstindx+i])      ; to be moved into memory.
next i
```

Execution Times

This section presents the formulas used to estimate the execution times of the low level procedures. The salient features of the algorithms are described, any special assumptions used are noted, then the formula is given.

SELECT(p,field-indx,test,comparand,dir)

If the test is for strict equality, the SELECT statement will take p clock periods, where p is the number of bits in the comparand. Logical NOT (all bits differ) also takes p clocks. If the test is for >, <, or ≠, the time can be less than p for a particular PE. However, so many PEs will be involved that the time will almost always be p clocks. Other tests, such as 'between', 'min', 'max', or 'closest to' may take additional time.

=, ≤, ≥, != (logical NOT):	p clock cycles
>, <, ≠ :	≤ p clocks

COPY(p,srcindx,dstindx)

The copy takes two clocks for each bit to be moved. This is because only one memory location can be addressed per clock cycle. The formula is thus:

2p clock cycles.

READ(p,srcindx,dstindx,dir)

The READ takes two clocks for each bit to be moved. If the PE is redesigned so that communication is selected with the source select multiplexers instead of the function select multiplexer this could be reduced to one clock per bit, provided that srcindx and dstindx were the same. The formula for the execution time is thus:

2p clock cycles.

For the CAAPP, this must be multiplied by four to account for the reduced communication capabilities of that machine.

SHIFT(p, n, dir, sign, srcindx, reg)

The shift runs one loop p-n times to shift the data bits. Another loop runs n times to set the bits that have been shifted in. Each iteration of the loops takes two clock cycles, thus the total time is:

2p clock cycles.

ADD2(p,srcindx,dstindx)
SUB2(p,srcindx,dstindx)

There is no difference in the execution time between ADD2 and SUB2. It takes one clock cycle to initialize the carry. Another 2p clocks are reqired to perform the p 1-bit additions. The total time is thus:

2p + 1 clock cycles.

ADD3(p,src1indx,src2indx,dstindx)
SUB3(p,src1indx,src2indx,dstindx)

There is no performance difference between these two procedures. The inner loop requires three clock cycles per bit. Another one clock cycle is required to initialize the carry bit. The total time is thus:

$3p + 1$ clock cycles.

ADDN2(p, srcindx, dstindx, dir)
SUBN2(p, srcindx, dstindx, dir)

There is no performance difference between these two procedures. The inner loop requires two clock cycles per bit. Another clock cycle is required to initialize the carry bit. The total time is thus:

$3p + 1$ clock cycles.

For the CAAPP, the formula is $12p + 1$ clock cycles.

ADDN3(p,src1indx,src2indx,dstindx,dir)
SUBN3(p,src1indx,src2indx,dstindx,dir)

There is no performance difference between these two procedures. The inner loop requires three clock cycles per bit. Another one clock cycle is required to initialize the carry bit. The total time is:

$3p + 1$ clock cycles.

For the CAAPP the formula is $12p + 1$ clock cycles.

NEG(p, indx)

One clock cycle is required to initialize the carry register to a '1'. Each bit in the operand requires on clock cycle to be negated. The total time is thus

$p + 1$ clock cycles.

MUL(p,src1indx,src2indx,prodindx,scratch)

Setting up the scratch area and the sign bits takes $2(p+1)$ clock cycles. The loop to perform the multiplication will execute p times. The worst-case time for the interior of the loop is 2.5p + 1 clocks. If all the multipliers are positive, the time would be 2p + 1 clocks, but this will be rare. The factor of .5 is used to model the behavior of the loop that adds ones to the most significant bits in the scratch field. If the multiplier were all ones, this loop would run p, (p-1), ... , 0 times. The .5p thus represents an average time for the negative multiplier correction loop.

One clock is required to determine if the multiplicand is negative. If so, an additional 2p+1 clocks are needed to correct for that. Finally, 2p clocks are needed to move the middle p bits of the product to the destination field. It will be quite rare that the average time is less than the worst-case time. This is because the MUL instruction will be executed on many PEs at the same time. It must be general enough to handle all the possible cases. It would be possible to check the sign bits of the multiplier using the SOME/NONE flag, thus allowing the correction steps to be skipped if all the multipliers and multiplicands were positive. For a 512x512 image this will be so rare that this possibility was not implemented. The total worst-case time is thus:

$$2.5\,p^2 + 7p + 4 \text{ clock cycles.}$$

AFMUL(p, ip,src1indx,src2indx,dstindx,scratch)

The AFMUL is similar to the MUL, however, one of its operands is an unsigned integer, typically representing a row or column address. For a 512x512 image, this would be an unsigned nine-bit integer. Using this as the multiplier lets us achieve considerably better performance than the MUL instruction. Not only will we execute the main loop fewer times, we will not have to correct for a negative multipler. We will still need the correction for a negative multiplicand, but this is much less expensive than multiplier correction. The formula for the execution time is:

$$ip*(2p + 1) + 4p + 2.$$

For p = 32, MUL takes 2788 clocks, compared to 713 for AFMUL.

DIV(p,dvdindx,dvrindx,qindx,rindx, scratch)

Since the purpose of this study was to estimate the execution time of the superquadric algorithms on bit-serial arrays, not to develop a full set of bit-serial microcode subroutines, a simplifing assumption was used to estimate the division time. We will model the time for division as being twice that for multiplication. This is quite conservative, actual performance should be almost equal to the MUL. The formula used is:

$$2*[\ 2.5\ p^2 + 7p + 4\ ] \text{ clock cycles.}$$

IMAGEIN(n, dstindx)
IMAGEOUT(np, srcindx)

These procedures have two loops. The outer loop executes once for each bitplane that must be input or output. The inner loop executes as many times as there are rows in the array of PEs. The formula below assumes 512 rows in the array. The body of the inner loop requires one clock cycle. Once the inner loop is completed, an extra clock cycle is required to place the data into the correct memory location. The execution time of this routine is 513 * n, where n is the number of bits to be retrieved from each PE. For the superquadric algorithms, n = 9p. The form of this equation used in obtaining the total time spent in disk swaps is:

$$16 * 9 * p.$$

The CAAPP will take four times as long to execute this routine as the 4NN and Pyramid machines.

APPENDIX B

SUPERQUADRIC ESTIMATION ALGORITHMS

This appendix presents the algorithms used to obtain the superquadric parameters. The algorithms are written in terms of fundamental operations such as addition, multiplication, and data transfers. The algorithms for the fundamental operations were presented in Appendix A. Each algorithm is presented and a table or formula is provided to determine the number of fundamental operations that are used. Any special assumptions are noted before the table or formula is given. All of these algorithms, along with the ones in the previous appendix, assume that the controller overhead is small enough to be neglected. In other words, we will assume that anyone building a system as expensive as these would design the controller to keep the array working at all times.

Many of the algorithms move the activity bit to a neighboring PE in order to enable a neighboring PE. Another simplifying assumption we will use is to neglect the time to to perform these one-bit transfers. Since normal precisions will be on the order of 32 bits, the error introduced by this assumption will be small.

A notational convienece used in several of the algoritms in this appendix that was not used in Appendix A is to identify memory fields by a letter. Referring to Figure 11 in Chapter Three, we see that we will need 11 p-bit memory fields. The fields that hold a_1, a_2, ... , b_2 will be referred to, in this appendix, as A, B, ... I. The two temporary fields will be referred to as T1 and T2. Some algorithms may use the names of particular coefficents, such as a_1 for clarity.

A final caveat, these algorithms have not been implemented.

Algorithm B1 Calculate Two Dimensional Regression Coefficients

{ Assumes the memory contents diagrammed in Chapter 3. }

```
    MUL (p, A, E, E, scratch)            ; Calculate common terms
    MUL (p, A, F, F, scratch)            ; scratch uses fields H and I.
    MUL (p, B, C, C, scratch)
    MUL (p, B, D, D, scratch)
    MUL (p, A, B, G, scratch)
    MUL( p, A, A, A, scratch)
    MUL (p, B, B, B, scratch)

    SUB2(p, B, A, EREG)                  ; Calculate a1
    SHIFT(p, 1, L, Y, A, REGE)
    COPY(p, G, B, E)                     ; Calculate a2
    SHIFT(p, 1, L, Y, B, REGE)
    SUB2(p, F, D, E)                     ; Calculate a6
    SUB2(p, E, C, E)                     ;  and a5.

    AFMUL(p, COL, F, C, SCRATCH)         ; Now do a3, AFMUL multiplies an
    AFMUL(p, ROW, E, D, SCRATCH)         ; unsigned integer ( the row,column
    ADD2(p, D, C, E)                     ; coordinates) by a fixed point value.
    SHIFT(p, 1, L, Y, C, REGE)

    AFMUL(p, COL, E, D, SCRATCH)         ;  Now do b1 and b2.
    AFMUL(p, ROW, F, G, SCRATCH)
    SUB3(p, D, G, H, E)                  ; b1 is in field H
    ADD3(p, D, G, I, E)
    NEG(p, I)                            ; b2 is in field I

    COPY(p, F, D, E)                     ; Copy a6 and a5 to a4 and a7
    COPY(p, E, G, E)

    done                 ; All pixels have a1..a7, b1, b2 in their local memory.
```

Obtaining the number of fundamental operations needed for this stage of the computation is straightforward. We have seven fixed-point multiplies, four multiplies that compute the product of an address field and a fixed-point number, four two-address adds or subtracts, two three-address adds or subtracts, three copies, and one negate. We also have three left-shifts which move a p-bit operand one bit to the left and should preserve the sign bit. Using the notation described in Appendix A, the execution time can be written as:

$$7[M] + 4[AFM] + 4[A2] + 2[A3] + 3[C] + 3[ASL] + 1[N] .$$

Algorithm B2 Distributing the Estimation Regions on the Flat Machines

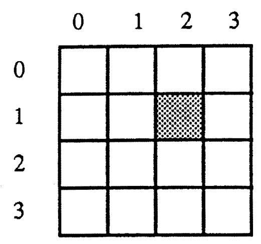
{ This routine moves the elements of estimation region (i,j) to the upper left corner of all solution submeshes or sub-submeshes. Once this has been accomplished, Algorithm B5 is used to fill the submeshes and complete the formation of **A** and b. These steps, as well as the matrix solutions, must be repeated for each estimation region within the solution submesh. This will mean 16 iterations of these procedures. This routine only works for estimation region dimensions strictly equal to 2, 4, or 8. }

```
left:   if j=0, goto right
        ptr = j*dim(er)                        ; Select all columns that must be
        if dim(er) ≠ 2, ptr=ptr+ 1/2 * dim(er) ; moved to the left,
nextl:  SELECT collsbs ≤ ptr                   ; and start moving them.
        READ(9p,right,b2,b2,E)
        ptr = ptr-1
        if ptr=0, goto right                   ; If ptr is in the first column, this part
        if ptrlsbs = 000, ptr=ptr-2,           ; is done. If we are at a sub-
           goto nextl                          ; submesh's first column, leave 2
                                               ; columns there, move
                                               ; the rest further left.
right:  if (j=3 or dim(er)=2 or (j ≥ 1/2*dim(er)), goto up
        ptr=((j+1)*dim(er)) - 1
        if dim(er)=8, ptr=ptr-(2*(2-j))        ; Set the pointer
nextr:  SELECT collsbs ≥ ptr                   ; and move the data
        READ(9p,left,b2,b2,E)
        ptr=ptr+1
        if ((ptrlsbs=1001 and dim(er)=4)       ; until all the data has been moved,
           or (ptrlsbs=11001 and dim(er)=8)), goto up
        if ptrlsbs = 001, ptr=ptr+2            ; or until we need to drop off 2
        goto nextr                             ; columns before moving the rest.

up:     if j=0, goto down                      ; This case is almost identical to
        ptr = j*dim(er)                        ; moving the data to the left. The
        if dim(er) ≠ 2, ptr=ptr+ 1/2 * dim(er) ; down case is also essentially the
nextu:  SELECT rowlsbs ≤ ptr                   ; same as moving the data to the right.
        READ(9p, north,b2,b2,E)
        ptr = ptr-1
        if ptr=0, goto down
        if ptrlsbs = 000, ptr=ptr-2
        goto nextu
down:   if ((j=3) or (dim(er)=2) or (j≥ 1/2*dim(er))
           goto quit
        ptr=((j+1)*dim(er)) - 1
        if dim(er)=8, ptr=ptr-(2*(2-j))
nextd:  SELECT rowlsbs ≥ ptr
        READ(9p,south,b2,b2,E)
        ptr=ptr+1
        if ((ptrlsbs=1001 and dim(er)=4) or (ptrlsbs=11001 and dim(er)=8)),
           goto quit
        if ptrlsbs = 001, ptr=ptr+2
        goto nextd
quit:   done
```

Quantifying the execution time for this routine is awkward, due to the special actions that must be taken for the different sizes of estimation regions. Another complicating factor is that the time is not the same for each of the 16 estimation regions. The differences arising from which estimation region is being moved will become less significant as the dimension of the estimation region increases, but this is little help for the two by two case.

Several simplifying assumptions were used in obtaining the timing estimates below. As mentioned at the begining of the appendix, we will neglect the time needed to select subsets of the PEs. Since the selections are based on only a few bits, while the data we will be moving is nine words deep, this should not cause a significant error. Another simplification is the use of an average time. The number of data movements needed for the estimation region at location 1,2 within the four rows and four columns of estimation regions were calculated for each dimension. This is shown below in Figure 34. The value obtained was multiplied by 16 to give the total time used by this stage in all 16 iterations. The average and total number of data transfers needed for each estimation region dimension are given below in Table VI. Each of these transfers involves nine words of whatever precision is being used, so the last column gives the number of p-bit READ instructions that must be issued.



The estimation region at row 1, column 2 was used since it more accurately represents the average data movements than the estimation regions at 1,1 or 2,2.

Figure 34: Estimation Region Used for Average Performance

TABLE VI

NUMBER OF DATA TRANSFERS NEEDED TO DISTRIBUTE THE ESTIMATION REGION AMONG THE SUB-SUBMESHES ON THE FLAT MACHINES

Dimension of the estimation region	Average number of block transfers	Total number of block transfers	Total number of READs issued
2	6	96	864
4	14	224	2016
8	36	576	5184

TABLE VII

NUMBER OF DATA TRANSFERS NEEDED TO DISTRIBUTE THE ESTIMATION REGION AMONG THE SUB-SUBMESHES ON THE PYRAMID MACHINE

Dimension of the estimation region	Total number of block transfers	Total number of READs issued
2	6	54
4	6	54
8	6	54

Algorithm B3 Distributing the Estimation Regions on the Pyramid Machine

```
{ This routine also places 2 x 2 portions of the estimation region at the upper_left corner of
   the 8 x 8 sub-submeshes. However, only one iteration of this routine will be needed
   using the Pyramid algorithm described in Chapter 3. }

            IA: SELECT((rowlsbs = 0) and (collsbs = 0))  ;move down one level
            READ( 9p, parent, b2, b2)
            IA: SELECT(collsbs = 01)               ; and collect it in the 2x2 block
            READ( 9p, east, b2, b2)                ; at the upper-left corner.
            IA: SELECT(rowlsbs = 01)
            READ( 9p, south, b2, b2)

   ; move down another level, the x in the least significant bits indicate "don't cares".

            IA: SELECT((rowlsbs = 0x0) and (collsbs = 0x0))
            READ( 9p, parent, b2, b2)
            IA: SELECT(collsbs = 001)              ; collect it in the 2x2 block
            READ( 9p, east, b2, b2)                ; at the upper-left corner.
            IA: SELECT(rowlsbs = 001)
            READ( 9p, south, b2, b2)
   done
```

This routine is much easier to characterize than the equivalent routine for the flat machines. It does not depend on the size of the estimation region or the position of the estimation region within the solution submesh. It also does not require 16 iterations. The number of data transfers needed are presented above in Table VII. This table ignores the time to select subsets of the PEs, for the same reason as was given above.

Algorithm B4 Filling the Sub-submeshes

```
{ Once the portions of the estimation region have been distributed to the upper-left corner
  of all the sub-submeshes of the solution submesh, this routine fills them, completing the
  formation of the A matrix and b vector. }

    IA:SELECT(collsbs = 001)                     ; get all coefficents into the first
    READ(9p, north, b2, b2)                      ; column of the submesh.
    READ(9p, north, b2, b2)
    IA:SELECT(collsbs=000 and rowlsbs=01x)
    READ(9p, east, b2, b2)

    IA: SELECT(collsbs=000 and rowlsbs≠000)      ; move a6, a7, and b2 down four
    for n=1 to 4                                 ; PEs to form the bottom half of the
            READ(2p, north, a7, a7)              ; sub-submesh.
            READ(p, north, b2, b2)
    next n
                                                 ; the bottom 4 PEs in sub-submesh
    SELECT(rowlsbs = 1xx)                        ; put the b2 coefficients into the
    COPY(p, b2, b1, Ereg)                        ; same field used to hold a1 in the
                                                 ;top 4 PEs.
    SELECT(collsbs=000)                          ; Now do the A matrix into a1.
    for col = 1 to 6
            IA:READ(1, west, Areg, Areg)            ;make next column active
            READ(p*(7-col), west, a(8-col), a(7-col)) ; read the values, leave one in
    next col                                        ; the a1 field.

    IA:SELECT((collsbs<101 and rowlsbs=101 or 110)     ; Clear the elements in a1 that
                    or (rowlsbs<101 and collsbs=101 or 110)) ; should be zero
    READ(p, BC, a1)

    done
```

This routine is common to all the architectures and estimation region dimensions. Once again, we will ignore the clock cycles required by the SELECT statements. The equation used to estimate the execution time for this routine is:

61[R] + 1[C] .

Algorithm A5 Calculating $\mathbf{A}^T\underline{b}$:

{ Assumes the solution submesh is square, with the elements of **A** and $\underline{b}$ arranged as described in Chapter 3. The LSB of the memory field containing the element of $a_{i,j}$ of **A** is pointed to by a_indx. The elements of $\underline{b}$ are pointed to by b_indx. }

```
        for col = 1 to 6
            IA:SELECT (collsbs = col)              ;Copy b to all other columns
            READ (p,b_indx,b_indx,west)
        next col

        IA:MULT(p,a-indx,b_indx,t1)                ;Do all the multiplies at once

        for row =  [(4*dim(er))-2] downto 0
            IA:SELECT (rowaddr = row)              ; Collect sums at top for result
            RADD2(p,t1,t1,S)
        next row
```

{ If dim(er) =4 or 8, sub-submeshes were used and we must collect the column sums together to get the solution.}

```
        ptr=[(4*dim(er))-2]
        if ptr ≤ 8, goto quit
again:  IA:SELECT(ptr ≥ collsbs ≥ ptr-6)
        COPY(p,bsum,bsum2,E)
        for ctr=1 to 8
            ptr=ptr-1
            IA:READ(1,east,REGA,REGA)
            READ(p,east,bsum2,bsum2)
        next ctr
        ADD2(p,bsum,bsum2,EREG)
        if ptr≤13, goto quit
        goto again
quit:   done
```

; Result vector is in bsum2 field of the top row of solution submesh.

This routine is used by all the machines in this study, although the flat machines must repeat it 16 times. The execution time will depend on the dimension of the estimation region. The equation used to estimate the execution time of this routine is:

$$6[R] + 1[M] + (4*\dim(er)-2)[RA2] + (\dim(er)/2)*(8[R] + [A2]) .$$

For the flat machines, this equation must be multiplied by 16. This equation ignores the time for the SELECT instructions. It also ignores the READ instructions that only move the activity bit to the next row or column. Since numeric precisions will be on the order of 32 activity bit to the next row or column. Since numeric precisions will be on the order of 32 bits, this is an error of less than ten percent.

Algorithm A6 Calculating A^TA

```
        for k=0 to 6                          ; for all columns of A
            IA:SELECT(collsbs=k)              ; copy A value to another field
            COPY(p,afield,a2field,E)          ;in the same column.
            for col = k-1 downto 0
            IA:SELECT (coladdr = col)         ; Read it to previous columns,
                READ (p,west,b_indx,b_indx,E)
            next col
            for col = k+1 to 6
            IA:SELECT (coladdr = col)         ; then to later columns
                READ (p,east,b_indx,b_indx,)
            next col

            IA:MULT(p,a_indx,b_indx,t1)       ;Do the multiplies for column k

            for row =  1 to k                 ; Collect sums at row k for result
                IA:SELECT (rowaddr = row)     ; First do sums that must move
                RADD2(p,t1,t1,S)              ; down to reach row k,
            next row
            for row=[(4*dim(er)) -2] downto k ; then the ones that must move
                IA:SELECT(rowlsbs=row)        ; up to reach row k.
                RADD(p,south,ata_prod,ata_prod,E)
            next row

        ; If dim(er) =4 or 8, sub-submeshes were used and we must collect the sums
        ; that are at row k of each group of eight columns to get the solution. This
        ; solution will be placed in the first seven by seven sub-submesh of the
        ; solution submesh.

            ptr=[(4*dim(er))-2]
            if ptr≤13, goto quit
  again:    IA:SELECT(ptr≥ collsbs ≥ ptr-6)
            COPY(p,ata_prod,,E)
            for ctr=1 to 8                    ; Move the sum to the next group by
                ptr=ptr-1
                IA:READ(1,east,REGA,REGA)        ; enabling the new column
                READ(p,east,ata2_prod,ata2_prod) ; and reading the data  (8 times).
            next ctr
            ADD2(p,ata_prod,ata2_prod,EREG) ; Data has now been moved to the
            if ptr≤13, goto new_col           ; next column of sub-submeshes, so
            goto again                        ; add the values and repeat the process
  new_col:  next k                            ; until done.
```

by all the machines, and exhibits the same dependance on the size of the estimation region that Algorithm A5 does. The formula for its execution time is:

7{[C]+6[R]+[M]+(4*dim(er) -2)[RA2]+(dim(er)/2)*(8[R]+[a2])}.

As in Algorithm B5, we neglect SELECTs and one-bit READs.

Algorithm B7 Inverting $\mathbf{A}^T\mathbf{A}$: The Faddeeva Technique

```
{ Assumes the input matrix is nxn and the solution submesh is n+1 x n+1. For the
regression equations, n = 7. The matrix is stored in the field m_field of the PE's local
memory. Two temporary fields, rtmp and ctmp, are needed, as well as scratch storage for
the multiply. }
        for count = 1 to n                        ; repeat elimination procedure n times
            IA:SELECT(rowlsbs=n or collsbs=n)       ; augment the matrix with the +
            READ(p,BC, m_field)                   ; and - identity matrices
            IA:SELECT((row=0 and col=n) or (row=0 and col=n))
            READ(1,BC,m_field+fractional precision)
            SELECT(row=0 and col=n)
            NEG(p,m_index,Ereg)

            SELECT(collsbs=0)                     ; copy column 1 to all columns
            COPY(p, m_index,ctmp,Ereg)            ; ctmp is the temporary field to
            for col=1 to n-1                      ; hold column copies
                SELECT(collsbs=col)
                READ(p,west,ctmp,ctmp)
            next col

            SELECT(rowlsbs=0)                     ; normalize the top row
            DIV(p,m_index,ctmp,m_index,scratch)

            COPY(p,m_index,rtmp,Ereg)             ; move the normalized top row to
            for row=1 to n-1                      ; the rtmp field of all other rows.
                SELECT(rowlsbs=row)               ; rtmp is the row temporary field
                READ(p,north,tmp,tmp)
            next row

            SELECT(rowlsbs ≠ 0)                   ;eliminate row 1 and column 1
            MUL(p,rtmp,ctmp,ctmp,scratch)
            SUB2(p,ctmp,m_indx,Ereg)
            IA:READ(p,south,m_index,m_index)   ;shift up and left
            IA:READ(p,east,m_index,m_index)
        next count
        done
```

The formula for the execution time is:

$$n\{\ 2n[R] + [Rbc] + 2[C] + [M] + [D] + [A2] + [N]\}\ ,$$

where [Rbc] is the time to read p bits from the Broadcast Comparand line and store them into the local memory. This operation will take one clock per bit, as opposed to the two clocks per bit needed to transfer a value from the memory of one PE to the memory of a neighbor. It is interesting to note that each elimination takes a single multiply and a single divide instruction. Since these are so computationally expensive on our machines, this is quite a benefit.

The algorithm above is not optomized. The columns of zeros which are shifted in from the east will remain zero [NASH 81]. The loop which copies the first column to all other columns could have a termination condition which decremented each time through the outermost loop. This would only be a minor improvement. It also might be possible to implement the single-division scheme for Gaussian elimination [FADDE 59]. This would be a more significant gain than eliminating the unnecessary copying of columns, sice we would not need to issue the multiply instructions. Another problem with this routine is that it makes no provision for matrices which are not well-behaved. Since the A^TA matrix is positive-definite, this should not be a great problem in areas of the image that can give a meaningful solution. Areas that contain edges will proabably cause an.

Algorithm B8 Calculating x

{ Assumes the inverted matrix is in the seven by seven submesh at the upper left corner of the estimation region, and the vector $A\underline{b}^T$ is in the first seven elements of the top row. }

```
for row = 1 to 6                          ;Copy AbT to all other rows
    IA:SELECT (rowaddr = row)
    READ (p,north, bt_indx, bt_indx)
next row

IA:MUL (p,ATA-1_indx,bt_indx,t1)          ;Do all the multiplies

for col = 5 downto 0                      ; Collect sums at left for result
    IA:SELECT (coladdr = col)
    ADDN2(p,t1,t1,E)
next col

done
```

When this routine terminates, the result vector is in the t2 field of the first seven PEs in the first column of the solution submesh. This routine does not depend on either the architecture or the dimension of the estimation region. The equation used to estimate the execution time is:

$$6[R] + [M] + 6[RA2] .$$

Again, this must be multiplied by 16 for the flat interconnection networks.

Algorithm B9 Reorganizing the Coefficients

{ This routine moves the superquadric parameters from the first seven PEs of the first column of each solution submesh to the upper-left corner of the solution submesh. It also changes the data from a 7x1 column vector to the form that will be used when the data is moved to the estimation region which gave rise to it. }

```
if (dim(er) = 2)                        ; For the 2x2 estimation region, the
   IA:SELECT( (4 ≤ rowlsbs ≤ 6) and (collsbs = 000))
   COPY(p, x_indx, xtmp, E)             ; data must be two fields deep, so
   for k= 1 to 4                        ; copy those elements that will be in
       IA: READ(1,south, Areg, Areg)    ; the second field and move them into
       READ(p, south, xtmp, xtmp)       ; the PEs that hold the first field.
   next k
   IA:SELECT((2 ≤ rowlsbs ≤3) and (collsbs=000))
   IA: READ(1,west,Areg,Areg)           ; Now make the 4x1 'complex' vector
   READ(2p, west, tmp, tmp)             ; into  a 2x2 by moving the bottom 2
   IA: READ(1,south,Areg,Areg)          ; entries right one PE and up two
   READ(2p, south, tmp, tmp)            ; PEs.
   IA: READ(1,south,Areg,Areg)
   READ(2p, south, tmp, tmp)

else                                    ; If the dimension = 4 or 8, the data
   IA:SELECT(collsbs = 1 and 4 ≤ rowsbs ≤ 6)
   READ(p,west,tmp,tmp)                 ; only needs to be one field deep, so
   for k= 1 to 4                        ; just make the 7x1 vector into a 4x2.
       IA: READ(1,south, Areg, Areg)    ; (moves the activity bit)
       READ(p, south, tmp, tmp)         ; (then the parameters)
   next k
endif
done
```

Upon completion of this routine, the data will have been reorganized into a block whose upper-left corner is positioned at the upper-left corner of the solution submesh. The execution time is dependent upon the size of the estimation region, but in a manner quite different than the other routines. The times for the four by four and eight by eight estimation regions will be the same, since both have enough PEs to hold the data as organized for the four by four region. The two by two region's routine will take longer to execute, since the data must be formed two memory fields deep. The number of READ instructions that must be issued are given below in Table VIII. This shows how many READS are necessary for one iteration and for 16 iterations. For the two by two estimation region, we must also issue a COPY instruction. This will be modeled as a READ, since it takes the same number of clock cycles as a READ, except on the CAAPP. FOr the CAAPP, the time for a read is approximately four times as long as for a copy, but since only one COPY must be issued, we will neglect this difference.

TABLE VIII

NUMBER OF DATA TRANSFERS NEEDED TO REORGANIZE THE CALCULATED PARAMETER ESTIMATES PRIOR TO THEIR BEING MOVED TO THE ORIGINATING ESTIMATION REGION ON THE FLAT MACHINES

Dimension of the estimation region	Number of READs issued per iteration	Total number of READs issued
2	11	176
4	5	80
8	5	80

Algorithm B10 Moving Parameters to Estimation Region; Flat Machines

{ The estimated parameters have already been moved to form the shape that will be stored (see Figure 18). We now move them from the upper-left corner of the solution submesh to the upper-left corner of the originating estimation region; i,j. I and J are the variables used in the main routine to identify estimation regions within the solution submesh. }

```
start
if (dim(er) = 2),                       ; Select the coefficients
   IA: SELECT (rowlsbs ≤ 1 and collsbs ≤ 1)
   for row=1 to i*dim(er)               ; Move the results down,
       IA: READ (1,north, Areg, Areg) ; first the activity bit, then the values,
       READ (2p, north, x_indx, x_indx)  ; until data is in correct rows
   next row
   for col=1 to j*dim(er)               ; Move the results to the right,
       IA: READ (1,west, Areg, Areg)  ; first the activity bit, then the values
       READ (2p, west, x_indx, x_indx)   ; until data is in right column
   next col
   COPY(2p,x_indx, tmp,Ereg)            ; Now store the results in the
                                        ; temporary fields.

else                                    ; dim(er) = 4 or 8

   IA: SELECT ( rowlsbs ≤ 1 and collsbs ≤ 3)
   for row=1 to i*dim(er)               ; Move the results down,
       IA: READ (1,north, Areg, Areg) ; first the activity bit, then the values,
       READ (p, north, x_indx, x_indx)   ; until data is in correct rows
   next row
   for col=1 to j*dim(er)               ; Move the results to the right,
       IA: READ (1,west, Areg, Areg)  ; first the activity bit, then the values
       READ (p, west, x_indx, x_indx)   ; until data is in right column
   next col
   COPY(p,x_indx, tmp,Ereg)             ; Now store the results in the
                                        ; temporary fields.
endif
done
```

The time for this routine will depend upon the dimension of the estimation region and the values of i,j. Once again, to estimate the execution time we will use the estimation region 1,2 as an indication of the average time consumed by this routine, then multiply by 16 to get the total time. The resulting number of READ instructions that must be issued are shown below in Table IX.

TABLE IX

NUMBER OF DATA TRANSFERS NEEDED TO MOVE THE REORGANIZED BLOCK OF PARAMETER ESTIMATES BACK TO THE ORIGINATING ESTIMATION REGION ON THE FLAT MACHINES

Dimension of the estimation region	Average number of READs issued per iteration	Total number of READs issued
2	14	224
4	13	208
8	25	400

Algorithm B11 Moving Coefficients to Estimation Region; Pyramid Machine

{ This routine moves the results up two levels from the solution submesh to the originating estimation region. It assumes that the parameters have been formed into the different shapes required for the various estimation region sizes and moves them so that each element resides in the north-west corner of a 2x2 submesh. The parents of these submeshes then read the data, and the original shape of the data is restored, but one level up. The 2x2 estimation region must be handled slightly differently than the others, and is considered first. }

```
start
for depth = 1 to 2                           ;Each iteration moves the data up one
                                             ; level
if (dim(er) = 2),
   IA: SELECT(rowlsbs≤1 and collsbs≤1)  ; Select the coefficients,
   IA: READ (1,north, Areg, Areg)       ; copy them to the next row down and
   READ (2p, north, x_indx, x_indx)     ; the next column to the right. This
   IA: READ (1,west, Areg, Areg)        ; makes all the data have different
   READ (2p, west, x_indx, x_indx)      ; parents, and these parents form
   IA: SELECT(rowlsbs≤1 and collsbs≤1)  ; another 2x2 block, on the next
   READ(2p,NEchild, x_indx, x_indx)     ; level up. Now select the parents and
                                        ; readthe data into them.

else                                    ; dim(er) = 4 or 8
   for row=1 to 3                       ; Move the results down
      IA:SELECT(rowlsbs ≥ 2*row)        ; until data are in correct rows.
      READ (p, north, x_indx, x_indx)
   next row
   IA:SELECT(collsbs = 001)             ; Now select the second column and
   READ (p, west, x_indx, x_indx)       ; move it over so that a different
endif                                   ; parent  will be used.
endif                         ; parent  will be used.
next depth
done
```

Since the Pyramid algorithm computes all estimation regions at the same time, this routine's execution time is mush easier to characterize than the corresponding routine for the flat machines, which would have a time dependent upon the position of the estimation region within the solution submesh. The execution time for this routine will depend upon the dimension of the estimation region, since the data for the two by two estimation region will be two fields deep, as opposed to the data for the larger estimation regions. The number of READ instructions that must be issued are given below in Table XI.

TABLE XI

NUMBER OF DATA TRANSFERS NEEDED TO MOVE THE REORGANIZED BLOCK OF PARAMETER ESTIMATES BACK TO THE ORIGINATING ESTIMATION REGION ON THE PYRAMID MACHINE

Dimension of the estimation region	Total number of READs issued
2	12
4	8
8	8

Algorithm B12 Fill Estimation Region with Parameter Estimates

{ This routine is divided into two parts. The first part moves each coefficient to its unique field in the local memory of its current PE, then collects the coefficients into the PE at position 0,0 within each estimation region. The second part copies the seven coefficents in PE 0,0 to all the other PEs in the estimation region. All the machines use this algorithm. }

```
start
for i =1 to 7                              ; For each coefficient
IA: SELECT(xxx)                            ; select the cell that holds it and copy
    COPY(p,tmp,coeff[i])                   ; the coefficient to the ith field of the
next i                                     ; PE's local memory.

if(dim(er) = 2)                            ; For the 2x2 estimation region,
   IA:SELECT(rowlsbs = 0)                  ; move the coefficients from the
   READ(2p, south, coeff[4], coeff[4])     ; second row to the top row.
   READ(p,south,coeff[7],coeff[7])
   IA:SELECT(collsbs = 0)                  ; Now select the top-left PE and read
   READ(p,east,coeff[2], coeff[2])         ; the 4 coefficients into it.
   READ(p,east,coeff[4],coeff[4])
   READ(2p,east,coeff[7],coeff[7])
else                                       ;dim(er) = 4 or 8.
   for row = 2 downto 0                    ; collect the values in the top row
        IA:SELECT(rowlsbs = row)
        READ((3-row)p, south, coeff[3-row],coeff[4-row])
   next row
   SELECT(collsbs = 0)                     ; Now select the top-left PE and read
   READ(4p,east,coeff[4], coeff[8])        ; the 4 coefficients into it.
endif

for col = 1 to dim(er)-1                   ; Now copy from the PE at 0,0 to all
   SELECT(collsbs = col)                   ; the other PEs in the estimation
   READ(7p,west, coeff[7], coeff[7])       ; region. First fill the top row,
next col
for row = 1 to dim(er)-1                   ; then copy that row to all other rows
   SELECT(rowlsbs = row)                   ; in the estimation region.
   READ(7p,north, coeff[7], coeff[7])
next row
done
```

TABLE XII

NUMBER OF DATA TRANSFERS NEEDED TO MOVE THE CALCULATED PARAMETER ESTIMATES TO ALL PES IN THE ORIGINATING ESTIMATION REGION

Dimension of the estimation region	Number of READs issued per iteration	Number of COPYs issued per iteration	Total number of COPYs issued	Total number of READs issued*
2	22	7	112	352
4	52	7	112	832
8	108	7	112	1728

* For flat machines only, Pyramid uses values in "iteration" columns

VITA

Ronald Ellison Daniel Jr.

Candidate for the Degree of

Master of Science

Thesis: SUPERQUADRIC DESCRIPTION ON LARGE ARRAYS OF BIT-SERIAL PROCESSORS

Major Field: Electrical and Computer Engineering

Biographical:

Personal Data: Born in Springfield, Missouri, September 8, 1960, the son of Ron and Barbara Daniel.

Education: Graduated from Putnam City High School, Oklahoma City, Oklahoma, in May, 1978; received Bachelor of Science Degree in Electrical and Computer Engineering from Oklahoma State University in May, 1985; completed requirements for the Master of Science degree at Oklahoma State University in July, 1987. Member of Eta Kappa Nu, Tau Beta Pi, IEEE Computer Society.

Professional Experience: Teaching Assistant, Department of Electrical and Computer Engineering, Oklahoma State University, August, 1984 to December 1985; Research Assistant, Department of Electrical and Computer Engineering, Oklahoma State University, August, 1985 to August, 1986; Lecturer, Department of Electrical and Computer Engineering, Oklahoma State University, August, 1986 to May, 1987. Research Assistant, Department of Electrical and Computer Engineering, Oklahoma State University, May, 1987 to present.