USC SIPI Report 115

The SCOOP Pyramid: An Object-Oriented Prototype of a Pyramid Architecture for Computer Vision

by

Herbert Scott Barad

December 1987

Signal and Image Processing Institute
University of Southern California, MC-0272
Los Angeles, California 90089-0272

Research Supported in Part by

Air Force office of Scientific Research (AFSC)
Electronics and Materials Science Division
under Grant No. AFOSR-84-0181

The United States Government is authorized to reproduce and distribute reprints for Governmental purpose notwithstanding any copyright notation hereon.

THE SCOOP PYRAMID:

AN OBJECT-ORIENTED PROTOTYPE OF A

PYRAMID ARCHITECTURE FOR COMPUTER VISION

by

Herbert Scott Barad

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOHY

(Electrical Engineering)

December 1987

Copyright 1987 Herbert S. Barad

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT UNLIMITED
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) USC-SIPI Report # 115	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Southern Calif. Signal and Image Processing Inst. — 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research
6c. ADDRESS *(City, State, and ZIP Code)* University Park/MC-0272 Los Angeles, CA 90089	7b. ADDRESS *(City, State, and ZIP Code)* AFOSR/NE Bolling AFB Washington, DC 20332
8a. NAME OF FUNDING / SPONSORING ORGANIZATION AFOSR — 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR-84-0181
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
The SCOOP Pyramid: An Object-Oriented Prototype of a Pyramid Architecture for Computer-Vision

12. PERSONAL AUTHOR(S)
Herbert Scott Barad

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical Report	FROM 1986 TO 1987	87/08/08	173

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Pyramid architecture, SCOOP pyramid, computer vision, LANDSAT aerial data

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report describes a working software prototype of a pyramid architecture, known as the SCOOP pyramid, to investigate its use and effectiveness in computer vision. The pyramid architecture is shown by simulation to be an effective architecture for a wide range of computer vision tasks from low level pixel-oriented operations to segmentation to high level symbolic operations. Results also show that processing overhead for a task can take more time than the task itself. This processing overhead includes the loading of convolution kernels and morphological look-up tables. An object-oriented methodology for modeling the individual processors and ports is used. The method of modeling and constructing the prototype is efficient and flexible. This encourages the fine-tuning of the architecture design. The prototype is used as as a testbed for simulations of computer vision tasks and the results of these simulations are presented. The simulations include isolated tasks: convolution, edge detection, and segmentation. Also, a complete scenario to find bridges in LANDSAT aerial data is studied. This scenario is controlled by a simple knowledge base. The SCOOP architecture provides an environ-

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* — 22c. OFFICE SYMBOL

ment to model architectures of arbitrary topology, complexity, and composition.

Dedication

The use of technology without careful thought as to its ultimate purpose and effect upon all is irresponsible and must be avoided. It is the responsibility of all engineers, scientists, and others who work in technological fields to insure that their work be productive and peaceful.

This dissertation is dedicated to those individuals who work for a peaceful and just world.

They shall beat their swords into ploughshares,
and their spears into pruning hooks;
nation shall not lift up sword against nation,
neither shall they learn war any more.

Isaiah 2:4

Acknowledgments

I would like to start by thanking my thesis advisor, Alexander Sawchuk, for his guidance during the years of preparation, research, and writing. I would also like to thank my thesis committee, Alexander Sawchuk (chairman), Cauligli Raghavendra, and Russell Johnson for their help in the preparation of this thesis.

I would also like to thank many other staff members of the Signal and Image Processing Institute: Allan Weber, Jerene Brooks, Gloria Bullock, Linda Varilla, Toy Mayeda and Ray Schmidt, whose help was invaluable. Also, I would like to thank the Data Systems Lab of TRW, Inc. for their financial support during much of the work. I would also like to thank the following for their help during the work: Richard Leahy, Rama Chellappa, Mark Thomsen, Dan Antzoulatos, Dimitrios Kalivas, Shankar Chatterjee, Ted Broida, Kung Huang, Keith Jenkins, Keith Price, and Bob Frankot.

I also cannot forget to mention the people of ParcPlace Systems: Nanette Harter, Ted Goldstein, Glenn Krasner, and Ron Carter, for their tremendous assistance with Smalltalk-80. I must also thank Evelyn Van Orden of Xerox Special Information Systems for the use of the HUMBLE expert system shell.

I would like to thank my sister, Karen Barad, and my cousin, Steven Sandler, for their advice during much of the final months. Finally, I would mostly like to thank my parents, Harold and Edith Barad, for their complete and unwavering support and love.

Table Of Contents

Dedication ii

Acknowledgments iii

Table Of Contents iv

List Of Figures viii

List Of Tables x

Abstract xi

Chapter 1 Introduction 1
- 1.1 Computer Vision 3
  - 1.1.1 Low Level Tasks 4
  - 1.1.2 Image Analysis 4
  - 1.1.3 High Level Tasks 5
- 1.2 Specialized Architectures 6
  - 1.2.1 Mapping of Algorithms onto Architectures 7
    - Index Transformation 7
    - Algorithm Modification 8
    - Partitioning 8
  - 1.2.2 Architecture Design 8
  - 1.2.3 Data Structures 9
  - 1.2.4 Parallel Architectures 10
    - SISD 10
    - SIMD 11
    - MISD 11
    - MIMD 11
    - Systolic Arrays and Wavefront Processors 11
    - Data Flow 12
  - 1.2.5 Hybrid and Hierarchical Systems 12
    - 2-D MCN with Broadcasting 12
    - Hierarchical Mesh Architecture 13
    - Digital Optical Architectures 13
    - Pyramid Architecture 14
  - 1.2.6 Data Routing via Interconnection Networks 14
    - Permutations 15
    - Broadcasting 15
    - Fixed Connections 15
    - Optical Interconnection Networks 15
  - 1.2.7 Performance Evaluation 17

	Simulation	17
	Benchmarks	18
	1.2.8 Considerations	19
1.3	Motivation	21
1.4	Research Contributions	23
Chapter 2	The Pyramid	25
2.1	The Structure	25
2.2	Pyramid Data Structure	28
2.3	Processing and Recognition Cones	30
2.4	Hierarchical Processing and Multiresolution Processing	32
Chapter 3	The Prototype	36
3.1	Building a Prototype	36
	3.1.1 Theoretical Analysis Method	37
	3.1.2 Software Simulation	38
	3.1.3 Software Prototype	38
	3.1.4 Hardware Prototype	39
	3.1.5 The Actual Architecture	39
3.2	Object Oriented Programming	40
	3.2.1 What is Object Oriented Programming?	40
	3.2.2 Message Passing	43
	3.2.3 Overriding Inherited Methods	44
	3.2.4 Advantages of Object Oriented Programming	45
3.3	Protocols for Multiple Processes and Simulations	46
	3.3.1 Class Process	47
	3.3.2 Class Semaphore	47
	3.3.3 Class Delay	47
	3.3.4 Class SharedQueue	47
	3.3.5 Class Simulation	48
	3.3.6 Class SimulationObject	49
	3.3.7 Class EventMonitor	49
3.4	SCOOP Class Protocols	49
	3.4.1 Class UnidirectionalPort	52
	3.4.2 Class PyramidEventMonitor	53
	3.4.3 Class Processors	53
	3.4.4 Class LowLevelProcessor	56
	3.4.5 Class TopLevelProcessor	59
	3.4.6 Class Pyramid	62
	3.4.7 Class ConvSimulation	64
	3.4.8 Class NBSimulation	65
	3.4.9 Class OPRSimulation	66
3.5	The Methods	66

Chapter 4 The Algorithms 68
4.1 Low Level Image Processing 68
4.1.1 Convolution 69
4.1.2 Morphological Operations 74
4.1.3 Nevatia-Babu edge detector 76
4.2 Image Segmentation 77
4.2.1 Color Segmentation 79
4.2.2 Ohlander-Price-Reddy Segmentation 80
4.2.3 Phoenix Segmentation 82
4.2.4 Histogram of Arbitrary Shaped Regions 83
4.3 High Level Symbolic Computation 87
4.3.1 Rule Chaining within a Production System 87
4.3.2 Parallelism During Chaining Process 89
4.4 Sequence of tasks 92

Chapter 5 Results 94
5.1 The Prototype 94
5.1.1 Prototype Construction 95
5.1.2 Monitoring of Prototype During Execution 96
5.1.3 Limitations of SCOOP 102
5.1.4 Algorithm Setup Considerations 104
5.2 Convolution 105
5.2.1 Algorithm Performance 105
5.2.2 Setup Overhead 106
5.2.3 Results 107
5.3 Nevatia-Babu Edge Algorithm 109
5.3.1 Algorithm Performance 109
5.3.2 Setup Overhead 110
5.3.3 Results 110
5.4 Segmentation 111
5.4.1 Simultaneous Histogram Computation of Arbitrary Shaped Regions 112
5.4.2 Ohlander-Price-Reddy Segmentation (Phoenix Version) 114
5.4.3 Results 118
5.4.4 Space Tradeoffs 119
5.5 The Expert System Shell 120
5.6 A Scenario: Finding Bridges in Aerial Scenes 121
5.7 Scenario Results 124
5.7.1 MSS Scenario 125
5.7.2 MSS Timing Data 129
5.7.3 TM Scenario 130
5.7.4 TM Timing Data 135
5.7.5 Processing Overhead 137

5.8 Comparison to Other Architectures ... 138

Chapter 6 Discussions ... 139
6.1 Methodology of Constructing Prototypes ... 139
6.1.1 Aid in Hardware Implementation ... 141
6.1.2 Modelling Other Topologies ... 142
6.1.3 Optical Architectures ... 146
6.1.4 Aid in Programming Architectures ... 149
6.2 Comparison of Pyramid to Other Architectures for Vision ... 150
6.3 Conclusion and Future Work ... 151

Bibliography ... 153

List Of Figures

1.1. Association of Classes of Algorithms to Data Abstraction 2
1.2. Optical Interconnection Network 16
1.3. Separate architectures for separate tasks 22
2.1. Pyramid Architecture 26
2.2. A Pyramid Data Structure 29
2.3. Processing Cone Model 31
3.1. Class structure of GeometricFigures 42
3.2. Abstract representation of objects passing messages 44
3.3. Class hierarchy of processors 50
3.4. Class hierarchy of simulations 51
4.1. Roberts gradient kernels 70
4.2. Directional edge kernels 71
4.3. Knowledge of pixel's 3x3 neighborhood is acquired 72
4.4. Knowledge of pixel's 5x5 neighborhood is acquired 74
4.5. Ordering of 3x3 Morphological Coefficients 75
4.6. Convolution kernels for Nevatia-Babu edge detector 77
4.7. Accumulating bin counts in the bottom level processors 85
4.8. Goal Tree of Sample Rules 89
5.1. Object table usage in 7 level SCOOP pyramid 98
5.2. Memory usage for a 7 level SCOOP pyramid 99
5.3. Memory usage for 5 level SCOOP pyramid. 102
5.4. Memory usage for 7 level SCOOP pyramid. 102
5.5. Gray level image before convolution. 108
5.6. Image after convolution with smoothing kernel. 108
5.7. Image after convolution with edge enhancement kernel 108
5.8. Smoothing kernel. 109
5.9. Edge enhancement kernel. 109
5.10. Gray level image before Nevatia-Babu algorithm 111
5.11. Processed with Nevatia-Babu algorithm 111
5.12. Number of bins each processor is responsible for 115
5.13. Utilization of the pyramid during each iteration of the OPR algorithm 116
5.14. Processing time for each step of OPR iteration—3 data bands, 6 bits/pixel 117
5.15. Processing time for each step of OPR iteration—1 data band, 8 bits/pixel 117
5.16. Red data band. 118
5.17. Green data band. 118
5.18. Blue data band. 119
5.19. Region boundaries overlaid with red data band. 119
5.20. Region boundaries overlaid with green data band. 119

5.21. Region boundaries overlaid with blue data band. 119
5.22. 3x3 Laplacian Kernel 125
5.23. MSS data band #5. 125
5.24. MSS data band #6. 125
5.25. Band 5, edge detection, threshold 32. 126
5.26. Band 5, edge detection, threshold 32, isolated pixel removal. 126
5.27. Band 6, threshold 50 127
5.28. Invert pixels of previous 127
5.29. Overlay of processed bands 5 & 6. 128
5.30. Inverted and magnified image of Fig. 5.29 128
5.31. Isolated pixels removed from Fig. 5.29 128
5.32. Inverted and magnified image of Fig. 5.31 128
5.33. MSS utilization (includes overhead). 129
5.34. MSS utilization (without overhead) 130
5.35. TM band #4. 131
5.36. TM band #5. 131
5.37. Band 4, threshold 32 131
5.38. Band 4, threshold 32, inverse, erosion 131
5.39. After 10 skeletonizations 132
5.40. After 50 skeletonizations 132
5.41. Band 5, edge detection 133
5.42. Band 5, edge detection, threshold 32 133
5.43. After isolated pixel removal 133
5.44. Overlay of Fig. 5.38 and Fig. 5.43 134
5.45. Inverted and magnified image of Fig. 5.44 134
5.46. Overlay using medial-axis transform 134
5.47. Inverted and magnified image of Fig. 5.46 134
5.48. Overlay after erosion and skeletonization 135
5.49. Inverted and magnified image of Fig. 5.48 135
5.50. TM utilization with overhead 136
5.51. TM utilization without overhead. 136
5.52. Processing times shown with associated overhead during scenario 137
6.1. Processor class hierarchy for SCOOP pyramid 144
6.2. Processor class hierarchy for SCOOP hypercube. 145
6.3. Optical Interconnection Network 147

List Of Tables

5.1. Size of pyramid for different levels. 96
5.2. Performance for convolution 105
5.3. Performance for convolution 107
5.4. Performance for Nevatia-Babu Edge Detection 110
5.5. Performance for OPR iteration 114

Abstract

This dissertation describes a working software prototype of a pyramid architecture, known as the SCOOP pyramid, to investigate its use and effectiveness in computer vision. The pyramid architecture is shown by simulation to be an effective architecture for a wide range of computer vision tasks from low level pixel-oriented operations to segmentation to high level symbolic operations. Results also show that processing overhead for a task can take more time than the task itself. This processing overhead includes the loading of convolution kernels and morphological look-up tables. An object-oriented methodology for modeling the individual processors and ports is used. The method of modeling and constructing the prototype is efficient and flexible. This encourages the fine-tuning of the architecture design. The prototype is used as a testbed for simulations of computer vision tasks and the results of these simulations are presented. The simulations include isolated tasks: convolution, edge detection, and segmentation. Also, a complete scenario to find bridges in LANDSAT aerial data is studied. This scenario is controlled by a simple knowledge base. The SCOOP architecture provides an environment to model architectures of arbitrary topology, complexity, and composition.

Chapter 1 Introduction

The field of computer vision includes tasks that range in complexity from simple low level arithmetic computation to higher level symbolic processing [1]. The low level tasks are usually local (i.e. they deal with small spatial neighborhoods) and involve simple computations done repetitively throughout the image. Also, the output is usually a data structure of similar size. This data structure is often an image of labeled pixels. These tasks are viewed as "image in, image out" algorithms.

Image analysis is defined as "classifying segments or features of the image into known classes" [2]. Image analysis tasks are often composed of many lower level computations. The output is not always a collection of pixels. Instead, more complex data structures are needed for symbolic representation of features. The output is a collection of symbolic descriptions of regions or lines in the image. From this output, object surfaces, boundaries, and possibly depth are inferred. It is during image analysis that much of the pixel-based to symbolic transformation takes place [3].

Higher level symbolic processing then uses the derived symbolic information. Such algorithms often manipulate data in the form of semantic networks. These networks are graphs that represent a relational model of the real world as interpreted from the pixel-based data. The high level processing also can serve to control the lower levels of processing via control strategies. These strategies can influence the rules and computations applied at lower levels. Often, such a system is classified as goal-oriented, goal-directed, or backwards inductive.

The different levels of vision tasks are associated with the levels of abstraction of the data that are being processed (see Figure 1.1). Low level image analysis tasks primarily make use of pixel-based image data. This

data is characterized by large regular data structures of values that represent some local spatial feature (such as intensity). These large data structures are often simple gray level images, but often complex data sets can also be used (e.g. Synthetic Aperture Radar).

Image analysis through high level tasks makes use of symbolic data that represent textural descriptions, regions, boundaries, surfaces, and objects. The symbolic data can also represent a "real world model" to be associated with the higher level symbolic descriptions by the vision system. Symbolic data structures and their algorithms are complex, although there is generally more pixel-based data than symbolic data.

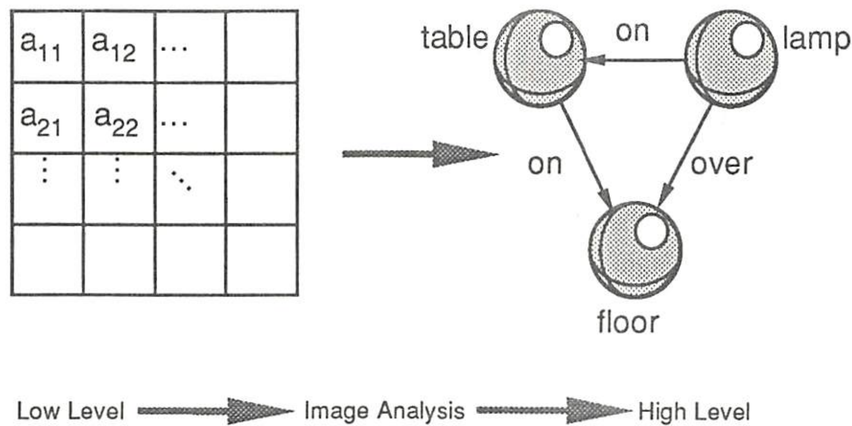

Figure 1.1. Association of Classes of Algorithms to Data Abstraction

Many vision systems use sequential general purpose computers [1, 4, 5, 6]. Their performance do not meet the increasing demands on throughput. For example, the processing of color images with a resolution and frame rate comparable to broadcast television requires a throughput rate of approximately 22.5 MBytes/sec. Certain special purpose architectures can be used effectively to increase performance (discussed in Section 1.2).

The design of parallel architectures often requires a prototype or testbed in which to develop and simulate algorithms. A testbed also provides an environment for collecting performance benchmarks. Chapter 3 presents

methods for constructing such a prototype that is used as a testbed for the simulation of a simple vision system on a pyramid architecture. The algorithms for the vision system are presented in Chapter 4 and the simulation results are presented in Chapter 5. The methods for constructing the prototype provide great flexibility. A prototype built using these methods is easily reconfigured and allows for the construction of other architectures of interest. The SCOOP pyramid uses an object-oriented methodology and allow for the modeling of the architecture at various levels of detail: architecture level, processor level, processor component level, gate level, and so on.

The prototype constructed for this research is configured as a pyramid that is modeled at the processor level. Pyramids are discussed in Chapter 2. The name of the prototype is the SCOOP pyramid. The acronym SCOOP stands for "Southern California Object-Oriented Prototype." Chapter 6 shows that by changing as little as a single method (methods are define in Chapter 3) the prototype can represent a SCOOP architecture of another topology. By changing the description of the processors (or ports), the prototype can represent architectures constructed from different components. Thus, the SCOOP pyramid is a useful tool for researchers who wish to prototype, benchmark, and design algorithms for their proposed architectures.

1.1 Computer Vision

The goal of computer vision is to enable machines to interpret two dimensional pixel-based data (captured from some sensory input such as visible wavelength cameras, infra-red cameras, ultra-violet cameras, and synthetic aperture radars) so that a relational model is formed from that data. Humans are natural experts at this and we do not yet have a complete understanding as to how this occurs [7].

The nature of vision is one that suggests special purpose architectures as the human eye contains hundreds of millions of optical receptors and other such biological "hardware" that operate in parallel [8]. A discussion of some special purpose architectures is in Section 1.2. Computer vision tasks can be divided into low level processing, image analysis, and image understanding and are discussed in the following sections.

1.1.1 Low Level Tasks

Some of the most computationally intensive computer vision tasks are the low level image processing procedures. Many architectures have been presented for these tasks [9, 10]. Typical low level tasks are local operations, such as filtering and morphological operations. Most filtering is done by convolution of the image with a kernel. Morphological operations can be performed quickly using a table look-up scheme [11]. A characteristic of these tasks is that the processing time (or number of steps) is usually known in advance. This tasks can also be classified as "image-in, image-out" algorithms.

1.1.2 Image Analysis

Image analysis tasks usually involve extraction of features from an image. The algorithms are more complex and often global. Different data structures result from these operations. Typical image analysis algorithms involve analysis of lines, regions, and areas [12]. At this level, segmentation of the image into distinct objects or surfaces occurs (although not necessarily the identification of the objects). Image analysis tasks can also be used to guide the operation of the lower level tasks. Attention can be focused on selected regions. The image analysis tasks can be guided by the higher level strategies. If such a system is goal-oriented, then the image analysis tasks can concentrate on that goal. Some image analysis

tasks can be adaptive and treat some parts of the image different from others.

1.1.3 High Level Tasks

High level symbolic tasks are less well defined than the previously mentioned tasks. This can be attributed to the lack of understanding about intelligence (both artificial and natural) and about human vision. When a human being observes a scene, what exactly is he looking at? What cognitive processes are going on inside his head that can (consciously or unconsciously) control his thought processes and further efforts to get a better look? In other words, the vision system, like the human being, must be able to focus its attention upon a region of interest depending upon the goal. These are the areas of much research in subjects such as artificial intelligence, cognitive psychology, and philosophy. For the image processing community, much current research has been limited to segmentation, network matching, model correlation, identification of objects in a scene, and firing of rules in a knowledge-base. Typically, they are implemented as goal-oriented systems—thus enabling the machine to perform specified tasks efficiently and economically.

The representation of knowledge (or knowledge base) of such systems should have the following properties [1]

1. represent analogical, propositional, and procedural structures,
2. allow quick access to information,
3. be easily and gracefully extensible,
4. support inquiries to the analogical structures,
5. associate and convert between structures, and
6. support belief maintenance, inference, and planning.

An inference mechanism is necessary in a goal-oriented system. Given precise goals and operating rules, inferences are made as to the operations

needed in order to achieve those goals. Also, inferences can be made in either direction. One can start with facts and infer possible results, or one can start with goals and work backwards towards the facts. Working in both directions simultaneously can produce results that working in either direction alone cannot produce. The synergy of forward and backward inferring can be the key of the success of a goal-oriented system.

1.2 Specialized Architectures

With performance requirements increasing, one must consider parallel architectures for efficient and possibly real-time execution of these tasks. "Real-time" execution of tasks require that the architecture can keep up with the input from the sensor. These sensors produce data at very high rate (on the order of 10 MBytes/sec) and the image data is very regular in structure. Depending on the tasks that are required, the structure of the data, and the processing requirements, a suitable and simple architecture can be proposed.

Parallel architectures are needed to achieve the high performance requirements. A problem is the programming of these parallel architectures—the mapping of the application to the particular architecture. In fact, many problems that originally had limited throughput because of intensive serial computations are now found to suffer from lack of communication bandwidth on some parallel computers. That is, the problem of communication between processors will limit the execution time to process data. If an application is to be mapped onto a general purpose parallel processor (if one exists), then one often has to settle with an architecture that is not matched to the particular application. The implementation becomes unbalanced. Consequently, the utilization of the processors will be inefficient. Special purpose architectures can be designed to match the particular range of tasks by balancing computation load with input-output.

There are many parallel solutions to simple low-level image processing tasks. Most of the previous work has concentrated on solving an isolated simple problem as opposed to complete scenarios that make use of a wide range of tasks. There are solutions for many low-level tasks: filtering, matrix-vector operations, Fourier transforms, and eigenvalue computation. The parallel solution to these algorithms is the mapping of the algorithm onto a specific parallel architecture. Often the algorithm is modified to be able to do this (algorithm synthesis). These methods are briefly discussed in the following sections.

1.2.1 Mapping of Algorithms onto Architectures

Algorithms are mapped onto architectures. A mapping determines which processors perform what action at a specific time. Different architectures will be more suitable for some tasks than others. The proper type of mapping can very often be elusive, because many of the mappings are *ad hoc* and the designer will often try to force a particular architecture when another is more appropriate. (Somewhat like trying to fit a square peg into a round hole.) Sometimes the designer is constrained by hardware limitations. Some mapping methods are index transformation, algorithm modification, and partitioning.

Index Transformation

Usually, the most repetitive parts of a sequential program are those that are within the indexed loops (e.g. DO loops in FORTRAN). Much of the parallelism within a program can be found in these loops. For example, index transformation is a method of "spreading-out" the indices over the processors (spatially) that are usually spread-out over time (temporally). Pipeline and systolic architectures (see Subsection 1.2.4) take advantage of this method when properly programmed. The only constraint to consider

for the simultaneous execution of statements are the data dependencies between them. A detailed discussion of this technique can be found in [13].

Algorithm Modification

Very often, the algorithm itself can be modified in order to expose some of its parallelism. Often, the designer of a sequential algorithm does not concern himself with this. Currently, most compilers for parallel machines do not perform well in detecting all the parallelism in a program. Most of the rules used to detect parallelism are *ad hoc;* the compiler will search for iterative loops and such in order to exploit the parallelism there. To exploit parallelism even further, the programmer must modify it by hand to expose more of the parallelism. Examples of these modifications can be found in [14].

Partitioning

Partitioning is a technique to map a problem into an architecture where the size of the problem is larger than the size of the architecture. Details of this technique can be found in [15] and [16].

1.2.2 Architecture Design

There are many schools of thought in architecture design. Many will take existing hardware and try to modify it to suit their needs. Others will design around certain fixed topological configurations (e.g. mesh connected, cosmic cube, and pipeline). Modular approaches to design are superior for the following reasons:

1. Complicated tasks can be very "unwieldy" when attempted all at once. A modular approach (sometimes called "divide and conquer") provides easier design subtasks. Also, if the project is large enough for many designers, a modular approach is the only feasible way.

2. A good modular design is easily testable, verifiable, and debugged.
3. A good modular design can be reconfigured.
4. A good modular design can be easily modeled, simulated, and analyzed.
5. A good modular design can be easily tuned.

The drawbacks are that a modular design can often decrease performance. Bottlenecks can occur where interface bandwidth does not meet the needs of the communicating modules. Hopefully, the interfaces can be matched to prevent any bottlenecks. It is felt that the benefits outweigh the drawbacks and thus suggest a stronger case for modular design. A modular design does not imply that the finished product must look like a bunch of boxes with skinny lines between them. In fact, the modular approach is often used as a prototype for a fully integrated system. The design is modular, not necessarily the result.

This modular approach can be used when designing an architecture for a large task. A case study of this approach has been presented by Barad and Moldovan [17]. The Karhunen-Loéve transform is broken down into smaller segments (or modules). Each of these modules is far easier to design. In fact, many of the modules used were already existing designs. The only extra consideration is the interfaces between the modules.

An object-oriented methodology is used for modeling the SCOOP pyramid. An object-oriented design is inherently modular and provides for total encapsulation of the objects. This will be fully discussed in Chapter 3.

1.2.3 Data Structures

The data structure of images (iconic structures) are very regular and large. They are large planar arrays representing intensity, temperature, gradients, or other features. Trying to devise data structures for the higher levels that

suitably model the data is quite difficult. Higher level data structures are dynamic, adaptive, and unpredictable as compared to the well structured, pixel-based data structures for the lower levels. These structures must be compatible to the extent that the processing between levels of abstraction will allow for easy conversion in either direction.

The overall set of data structures in a vision system should be hierarchical [8]. Pyramid architectures have been shown to have properties that lends itself to natural vision [18]. This type of hierarchical model provides a medium in which to have the proper set of data structures and processing relationships.

1.2.4 Parallel Architectures

Computer architectures can be classified into several groups. Classically, these groups are [9]

SISD:	single instruction stream—single data stream
SIMD:	single instruction stream—multiple data stream
MISD:	multiple instruction stream—single data stream
MIMD:	multiple instruction stream—multiple data stream

Many new architectures exhibit properties of different categories and are placed in a new category of their own (e.g. systolic, data flow, and MSIMD).

SISD

These architectures are the conventional single processor architectures commonly in use. These are sometimes labeled as "Von Neumann architectures" and use the stored-program concept. Essentially, a single processor will process data and will pass the data back and forth between it and memory. This type of architecture is very easy to control, but allows only sequential processing.

SIMD

The term SIMD refers to a class of architectures that contain many processors, each performing identical functions at the same time. The SIMD machines have received much attention in the field of image processing owing to the nature of many of the low level algorithms. Typically, these architectures are composed of a collection of processing units, and interconnection network, and a controller. Each processing unit contains a processor and some local memory. The interconnection network provides for interprocessor communication. The controller broadcasts the instructions to the processing units, providing the single instruction stream. Examples of SIMD architectures include many processor pipeline and mesh connected architectures.

MISD

These architectures generally do not exist. The entry was only included to complete the possible combinations of single/multiple instruction and data streams.

MIMD

This category of architectures includes a wide range of multiple processor systems that do not have the restriction of identical operations nor synchronous operation. They are typically interconnected through a common bus, a fixed network, a switched network (such as a crossbar), or through shared memories.

Systolic Arrays and Wavefront Processors

Systolic arrays are very closely related to SIMD architectures. They consist of a regularly connected array of identical processors, each performing identical tasks. They operate in a synchronous manner: taking in data

from neighboring processors, operating on the data, and then passing the data to neighboring processors during each time unit [19].

Wavefront arrays are similar to systolic arrays, except that they are asynchronous. Their computation can be described in wavefronts [20]. Both of these architectures offer efficient low level processing of data using VLSI technology.

Data Flow

Data flow architectures are asynchronous, multiple processor systems that operate in a data driven or demand driven fashion [21, 22]. Data flow architectures are the first serious attempt at a general purpose parallel architectures. On the other hand, the overhead and control of data flow systems are quite high. Also, a special purpose architecture can always be made more efficient than a data flow architecture if one can accept the limited capability of the special purpose machine.

1.2.5 Hybrid and Hierarchical Systems

Hybrid and hierarchical systems have architectures that use additional hardware that allows for some special communication between the processors. For example, a mesh connected architecture with only local interconnections can be augmented with a global bus in order to allow global broadcasting. Brief descriptions of some hierarchical systems are presented in the following subsections.

2-D MCN with Broadcasting

A two-dimensional mesh-connected network with row and column broadcasting has been proposed by Prasanna Kumar and Raghavendra [23]. This is a hybrid architecture that combines the capabilities of local communication (via 4 nearest neighbors) and global communication (via individual row and column buses). This particular architecture exhibits

many advantages over a mesh with one global bus as it allows for much parallel non-local communication, often without the problem of bus contention. Many operations can be executed quite effectively (and some optimally) [23].

Hierarchical Mesh Architecture

An improvement upon the previously discussed 2-D MCN with row and column buses is the HMESH architecture [24]. It is a two-dimensional mesh-connected network of processors with a hierarchy of broadcast buses along the rows and columns. The hierarchy of broadcast buses allow for global communications in $O(\log N)$ time. This is comparable to global communication time in a pyramid architecture. This architecture has been shown to provide good performance for many geometric algorithms [25].

Digital Optical Architectures

A digital optical architecture consists of a collection of optical processors and an optical interconnection network. Optical interconnection networks are discussed in Subsection 1.2.6. Optical architectures offer many potential advantages over conventional electronic architectures [26]. The design of such architectures may overcome some of the limitations of electronic architectures [27, 28]. Some of these advantages are

1. Light signals can pass through each other without interfering, while electronic signals may interfere with each other in close proximity at high data rates.
2. Optical architectures are inherently parallel.
3. Optical interconnections are inherently three dimensional in nature.

Thus, some types of optical interconnection networks may be more cost effective, physically compact, and have higher performance than electronic networks.

Pyramid Architecture

A pyramid architecture is a series of interconnecting two dimensional mesh-connected arrays of decreasing sizes. These architectures can employ a multiresolution model for the data. The pyramid architecture is the primary architecture of interest in this research. Chapter 2 is devoted to the pyramid.

1.2.6 Data Routing via Interconnection Networks

The goal of a parallel architecture is to achieve a speed up that is proportional to the number of processors in the system (in practice, the speed up is often much less). "Speed up" is defined as

$$S = \frac{\text{number of steps for sequential algorithm}}{\text{number of steps for parallel algorithm}}. \tag{1.1}$$

In order to achieve a high utilization of the processors, they must often communicate with each other. Examination of the data dependencies specify the required communication. "Utilization" is defined as

$$U = \frac{\text{number of processors busy}}{\text{number of processors}}. \tag{1.2}$$

Utilization of the architecture can change with time. High utilization is desired to show that the architecture is being used in a cost effective manner.

In practice, the communication costs of many parallel algorithms becomes the dominant factor in the design. If a crossbar network is used the

hardware costs grow as the square of the number of processors grows. This cannot be acceptable when the number of processors is large, or when the planar design of a VLSI circuit would make the layout too complex.

Permutations

Many networks can be described by the set of permutations that it can perform. Some algorithms require perfect shuffle, exchange, or other permutations. Often, certain permutations can be achieved by several elementary permutations that the architecture performs in one step. Thus, some data exchanges can be performed by a multiple stage (or multiple pass) network.

Broadcasting

Broadcasting can be a convenient means of achieving global communications throughout an architecture. Considering the number of steps that some interconnection networks will require (especially when an architecture is not matched to the particular tasks), broadcasting becomes more attractive; however, bus contention and hardware cost must be considered.

Fixed Connections

Many architectures incorporate fixed connections. This implies a fixed set of data exchange paths that can be used. The advantage of this is the low hardware cost and the lower cost of control of the network. In fact, the only control needed is to work out the protocol between processors that communicate.

Optical Interconnection Networks

Interconnection networks created with optical hardware offer possible advantages over conventional electrical interconnections. Electronic

hardware is, for the most part, a two-dimensional medium. Electronic components are etched onto a small number of layers of material to form chips, which in turn are mounted on circuit boards. Many parallel architectures have a three-dimensional topology. Mapping a three-dimensional architecture onto a two-dimensional surface can create problems in interconnecting the components. Optics system are inherently three-dimensional, and light beams can cross each other in a linear medium without interfering.

Figure 1.2 shows a simple optical interconnection network. This network is a 4×4 crossbar. Although the complexity of an N×N crossbar network grows by $O(N^2)$, the parallel nature of optics may make large crossbar networks feasible.

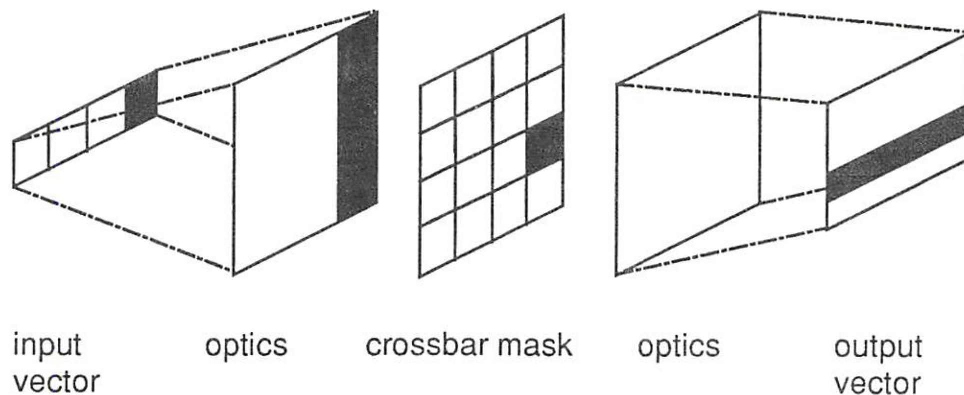

Figure 1.2. Optical Interconnection Network

The crossbar mask is a hologram and is used to route the optical signals. The use of such an optical interconnection network makes the layout and configuration of three-dimensional architectures much easier.

The current state of optical hardware lags behind that of electronic hardware. The switching (or reconfiguration) time of optical networks are longer than equivalent electronic networks, although the bandwidths are much higher [26]. Advances in optical hardware will increase the

likelihood that optical networks will used more often in future architectures.

1.2.7 Performance Evaluation

Parallel computers are useful in that they achieve a speedup over that of sequential computers (see Equation 1.1). If a parallel computer has **N** processors, the maximum speedup achievable is **N**. In practice, this speedup is rarely achieved. Some reasons for this inefficiency are: idle processors, setup time, memory conflicts, communication costs, and inefficient algorithms.

Simulation

The simulation of a parallel architecture on a single processor, general purpose computer is quite complex. The simulation is often done at various levels of abstraction so that verification can take place at each these levels. Tanimoto suggests that the design of the simulation can take place at various levels [29]

1. machine level - emulate each cycle of each processor, or
2. intermediate (modular) level - the individual simulated processing tasks are routines to be called for a more complex simulation.

Fritsch proposes similar levels of simulation [30]. Simulations can take place at even lower levels, such as the register or gate levels.

The simulation at the machine level is important to analyze the mapping of the algorithm to the architecture. Very often, unforeseen problems arise that are not anticipated in the theoretical analysis of the algorithm. In particular, a result of this research (discussed in Chapter 5) is that the setup times for many algorithms are not negligible, and in fact often dominate the total processing time for a task. The intermediate level of simulation is

useful in speeding up such a simulation. It is important that all intermediate tasks be verified at the machine level before being used at a higher level.

For example, the algorithm is simulated at a modular level to verify that the mapping is correct. Then, simulation and verification on each module is performed. Finally, a simulation of the entire system is performed by simulating the modules and their interfaces.

The methods used in the simulation will have to take advantage of the modular nature of the design. Thus, a prototype of the architecture must be built that will properly simulate the hardware to the level of interest. The modular design requires that each processor be individually constructed and integrated into the architecture and that it properly executes the necessary instructions for each algorithm.

Benchmarks

A benchmark is a program that is run on a computer system to test that system's performance. Benchmarks can be useful in evaluating a system's performance for a class of problems—providing that the benchmark program is representative of that class of problems.

Benchmarks are useful if they take into account the following:

1. the size of the problem,
2. setup times,
3. overhead related to communication management,
4. overhead related to loading data, both input and other data (e.g. kernels),
5. overhead related to loading processor instructions, and
6. overhead related to output.

If the benchmarks do not take these into account, then the benchmarks can be very deceiving of the system's real performance. There still might be some assumptions made in the benchmarks, but the times should still reflect expected performance.

Duff discusses the pitfalls of performing benchmarks on image processors in [31]. The conventional approach of ranking systems solely on their timing performance must be avoided when considering parallel architectures. Other factors are important, such as cost, reliability, resolution, precision, flexibility, and programmability. The book, edited by Uhr et al. [32], contains many papers with benchmarks on various image processing architectures.

Often a class of problems is best represented by a realistic scenario on real data. The benchmarks might look good for a collection of tasks, but if the final goal is to perform these tasks together, then the benchmarks might not represent the overhead needed to manage the transition from one task to the other. That is, achieving good performance in a set of tasks done separately does not insure good performance for these tasks done together. This is discussed further in Section 1.3. For this reason, a complete scenario is included in the benchmarks for the SCOOP pyramid.

1.2.8 Considerations

Current parallel implementations (and many proposed solutions) do not effectively meet the requirements for a full range of vision tasks. For example, many pipeline architectures are totally unsuited for data dependent processing as this would cause the pipeline to be flushed and restarted. Also, special purpose architectures suffer from their inherent inflexibility. Their specialized usage limits the scope of problems that can be solved efficiently. For some, this limitation is tolerable. For others, this limitation can be very costly. There are trade-offs that must be considered

between the performance of specialized architectures versus the flexibility of general purpose architectures.

Many complex algorithms are compute-bound. "Compute-bound" means that the computer takes longer to process the data than its takes to input and output the data. More precisely, an algorithm is compute-bound if and only if the number of computing operations is larger than the total number of input and output data elements [9]. Otherwise, it is considered I/O-bound. When an algorithm is unbalanced (i.e. heavily bounded by computation or I/O), the type of bounding is the limiting factor in the performance of the execution. For instance, a VLSI chip has a limited number of pins. Only a limited number of these pins can be utilized for I/O. Once the implementation is considered to be I/O bound, then improving the calculation steps within the chip does not improve the overall timing as the chip cannot input or output data fast enough to keep up with faster calculations. The hardware inside the chip will have to wait for the I/O operations. Thus, finding a fast parallel implementation for the calculations of an operation is not enough to improve overall performance if the implementation becomes I/O-bound.

Two more important considerations are whether special purpose architectures can be reconfigured and easily programmed so that they become flexible and thus efficient for a larger class of problems. These considerations suggest a modular approach to the design of the architecture. The modules are very specialized for performance, but can be integrated as a whole system that exhibits flexibility. But, in order to pursue the modular approach, the nature of the tasks to perform must be examined.

Another consideration is whether the approach to the vision tasks are top-down or bottom-up. A top-down approach can be goal oriented and directs the lower level processes. A bottom-up approach will perform specified

tasks during the analysis and infer consequences from the results. A comprehensive system will use both approaches [33].

1.3 Motivation

There are problems when attempting to integrate different levels of vision tasks on the same architecture. Many architectures are suited for a particular class of tasks. As an example, a pipeline of processors handles data in a stream format. This data must be very regular in structure. Also, the pipeline cannot make decisions that are dependent on the data without risking the interruption of the smooth flow of data through the pipeline. Thus, while a pipeline is well suited for low level vision tasks, it does not allow for more complex data structures and more complex algorithms needed for higher level vision tasks.

More massively parallel two-dimensional mesh-connected arrays are well suited for low level local operations, but their SIMD nature does not efficiently support more complex algorithms. MIMD mesh-connected arrays still suffer from lack of global communication and global control.

Also, many architectures created for symbolic operations (e.g. Symbolics 3600) are not well suited for the massive amounts of pixel-based data as are other massively parallel architectures. Even though the individual processors of these architectures might be quite powerful, these architectures are often too small to handle a data set of images efficiently.

Figure 1.3 emphasizes another difficulty in attempting to integrate various tasks into a system. Suppose there tasks A and B are to be processed. Further, suppose that two separate architectures are used—each one is designed to efficiently execute task A and task B, respectively. If the communication between processes A and B (i.e. between subarchitectures A and B) takes longer than the processing of tasks A and B, then the overall process is now I/O-bound. That is, an inter-architecture bottleneck

is formed as data is *pushed* from one architecture to the other. The effects of this bottleneck become even more severe when we consider that the overall process might be an iterative process between tasks A and B.

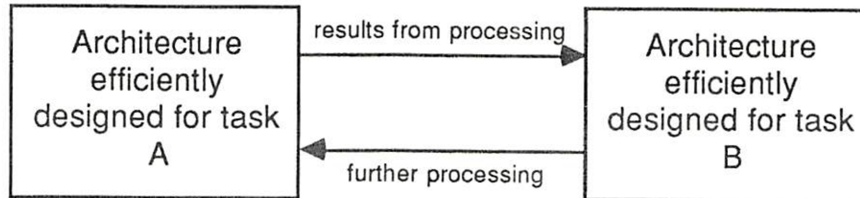

Figure 1.3. Separate architectures for separate tasks

A prime motivation of this work is the desire to integrate all levels of the vision process onto the same architecture to avoid such a bottleneck that would result in re-evaluation of the data. If higher-level analysis showed that further lower-level processing is needed (goal directed processing), then switching back and forth between separate specialized architectures results in a bottleneck. Therefore, it becomes important to be able to efficiently perform different tasks at different levels of abstraction on a *unified* architecture that would allow a smooth transition between these levels.

Another motivation is to develop a methodology to construct prototypes of proposed architectures. It is often difficult to build the full-sized architecture to be studied and often a scaled-down hardware prototype is not a suitable representation of the final architecture. Software prototypes have several advantages over scaled-down hardware prototypes

1. The time needed to design, build, and debug the prototype is considerably less is software than in hardware.
2. The software prototype can better assist the software developer of algorithms for the prototype since the software prototype can represent an architecture of a larger size.
3. Software prototypes are much cheaper to build.

4. Software prototypes can be easily modified to reflect architectures of other topologies, or even architectures composed of radically different types of hardware (e.g. optical).

Of course, the major drawback of the software prototype is that the actual time to execute an algorithm is much longer than the hardware prototype. Several microseconds of processing in hardware may require minutes of simulation. Thus, the software prototype is ideal for designing algorithms, analyzing the architecture, and performing benchmarks, but extremely poor for processing data in a production environment.

1.4 Research Contributions

The following is a list of contributions resulting from this research:

1. The pyramid architecture is shown by simulation to be an efficient and effective architecture in which to implement a wide range of computer vision tasks. The pyramid can adapt to the different levels of abstraction of the data that correspond to the different levels of vision tasks. It meets the requirements for a vision architecture [34] and has the hierarchical structure necessary for recognition cone algorithms [8]. The following algorithms are mapped and benchmarked on the SCOOP pyramid:
   a. convolution,
   b. morphological operations (used in item f),
   c. Nevatia-Babu edge detector,
   d. computing histograms of arbitrary shaped regions from multispectral data (used in item e),
   e. Ohlander-Price-Reddy segmentation,
   f. a complete scenario to find bridges in LANDSAT scenes.

2. An object-oriented methodology is presented to model any architecture by creating objects (instances of abstract class descriptions) for each component of the architecture. This provides a flexible environment to study alternative architectures.
3. The SCOOP pyramid is a working software prototype of a pyramid architecture. It serves as a testbed in which to develop and benchmark different algorithms. The SCOOP pyramid is currently the seven levels—two larger than existing hardware prototypes (see Section 3.1.4). The design of SCOOP allows modeling of architectures of arbitrary size (see Section 5.1.3).
4. The overhead to set up certain tasks on parallel architectures is often the most time consuming process (see Section 5.7.5).

Each of these items are fully explored in the thesis and are summarized again in Chapter 6. The following chapters present the pyramid architecture, an object-oriented methodology for constructing prototypes of parallel architectures, a selection of algorithms that are implemented on this prototype, and timing results. Finally, the implications of all this work and suggestions for future work are discussed in the final chapter.

Chapter 2 The Pyramid

The pyramid architecture is the primary architecture of interest in this research. This chapter will present the pyramid in detail, summarize past and current use of the pyramid, and discuss the motivation behind the use of the architecture as a single unified architecture for a wide range of vision tasks. Later chapters present the actual prototype, algorithms, and results.

2.1 The Structure

A pyramid (or cone) architecture is a hierarchical layer of mesh connected arrays [35, 36, 37]. Each level decreases in size until reaching an apex at the top. The reduction of size of the levels is by a constant factor—usually four. The inter-level interconnections are such that each processor communicates with other processors on neighboring levels. Typically, each processor has one parent processor and four child processors.

Figure 2.1 is a partial view of the top three levels of the pyramid. The intra-level mesh connections are not shown in order to avoid cluttering up the drawing. Each level is mesh connected to its nearest four neighbors on the same level. The bottom level of the pyramid is the same size as the image produced by the sensor. Suppose a camera produces digitized images that are 512×512 pixels. These images are loaded directly into the pyramid's bottom level, one pixel per processor. If the camera is constructed properly, the image loaded into the bottom level of the pyramid at the start of each time frame. Standard video frame rates are 30 frames/sec. Thus, the bottom level of the pyramid will have a new image loaded into the bottom level 30 times each second.

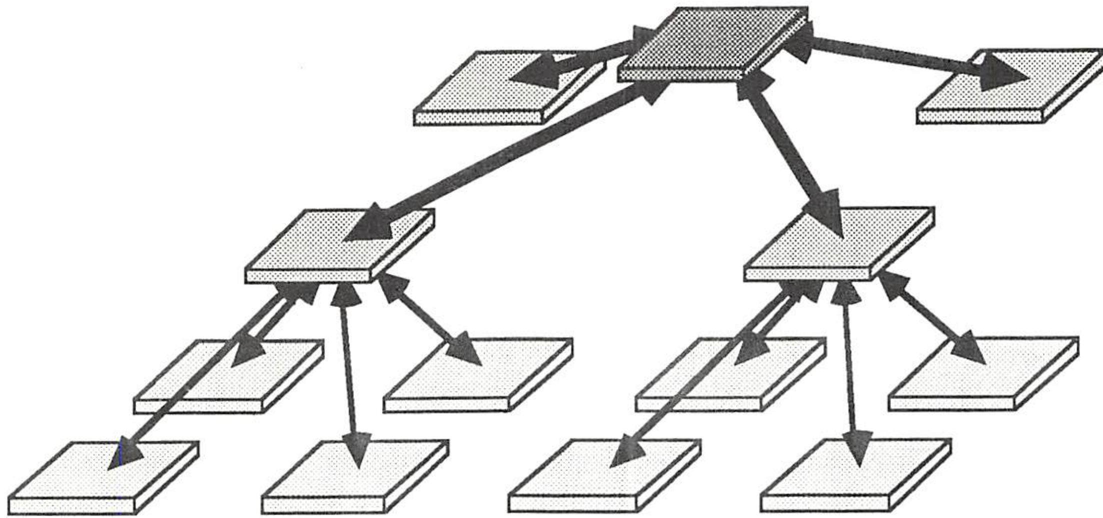

Figure 2.1. Pyramid Architecture

A processor's location is given by a 3-tuple (x, y, z) representing the (x,y) location on level z. The levels are numbered such that the top level is level 1. The interconnections between the levels can be mathematically expressed in the following recurrence relations. The top level processor has no parent processors. Equations (2.1)—(2.3) give the parents' coordinates

$$x_{parent} = \left\lfloor \frac{x+1}{2} \right\rfloor \tag{2.1}$$

$$y_{parent} = \left\lfloor \frac{y+1}{2} \right\rfloor \tag{2.2}$$

$$z_{parent} = z - 1 \tag{2.3}$$

in terms of the coordinates of the child x, y, and z, where '$\lfloor \; \rfloor$' denotes greatest integer less than the argument. Equations (2.4)—(2.11) describe the coordinates of a processors' neighbors. The directions north, south, east, and west refer to the relative directions of the processors as if viewed on a map. If one looks "down" on any level, then the intra-level, four connected neighbors of a processor will be its northern, southern, eastern, and western neighbors. Actual directions are not really important, the

names are used as they are more familiar than constant referral to a processors relative indices. The z coordinate is not defined in the first four sets of intra-level relations as the processors are on the same level. Thus the relations

$$x_{north} = x - 1 \tag{2.4}$$

$$y_{north} = y \tag{2.5}$$

$$x_{south} = x + 1 \tag{2.6}$$

$$y_{south} = y \tag{2.7}$$

$$x_{east} = x \tag{2.8}$$

$$y_{east} = y + 1 \tag{2.9}$$

$$x_{west} = x \tag{2.10}$$

$$y_{west} = y - 1 \tag{2.11}$$

are defined. Equations (2.12)—(2.23) describe the coordinates of a processors' children, as summarized by

$$x_{NW-child} = (x \times 2) - 1 \tag{2.12}$$

$$y_{NW-child} = (y \times 2) - 1 \tag{2.13}$$

$$z_{NW-child} = z + 1 \tag{2.14}$$

$$x_{NE-child} = (x \times 2) - 1 \tag{2.15}$$

$$y_{NE-child} = y \times 2 \tag{2.16}$$

$$z_{NE-child} = z + 1 \tag{2.17}$$

$$x_{SW-child} = x \times 2 \tag{2.18}$$

$$y_{SW-child} = (y \times 2) - 1 \tag{2.19}$$

$$z_{SW-child} = z + 1 \tag{2.20}$$

$$x_{SE-child} = x \times 2 \quad (2.21)$$

$$y_{SE-child} = y \times 2 \quad (2.22)$$

$$z_{SE-child} = z + 1. \quad (2.23)$$

Note that the bottom level processors of the pyramid do not have children. A processor can also determine which child it is, relative to its parent, by using the following rules:

if my row is even, then I am a southern child,
if my row is odd, then I am a northern child,
if my column is even, then I am an eastern child, and
if my column is odd, then I am a western child.

For example, a processor at (5, 4, 6) is in the 6th level of the architecture. Its parent processor is at location (3, 2, 5) and it is a northeast child relative to that parent. The location of its other neighbors can be similarly computed.

2.2 Pyramid Data Structure

A pyramid data structure is a type of generalized image structure. The data structure is constructed from an image at different levels of resolution. For example, an image that is originally N×N is represented at N/2×N/2, N/4×N/4, N/8×N/8, and so on (see Figure 2.2). This is often done by averaging the four pixel values together and the result is the value that represents the parent processor's value. The purpose is to allow for processing at a chosen level of reduced resolution to increase performance as processing a smaller image takes less time. Many "coarse-to-fine" image-resolution algorithms have been developed that are much more efficient than conventional high-resolution image algorithms. In matching algorithms for example, the idea is to take a "quick look" at lower levels of resolution so that the algorithm can "home-in" on the object to match at the higher resolution [1]. Baugher and Rosenfeld use this technique to find

boundaries of large objects [38]. Burt discusses many coarse-to-fine search strategies [18]. Klinger discusses multiresolution processing with regard to actual parallel hardware [39]. Cibulskis and Dyer discuss a technique whereby segmentation is pursued using evidence gathered at many levels of resolution [40]. Crowley defines a multiple-resolution representation for shape of regions [41]. Preston uses multiresolution techniques for the processing of microscopic images [42]. Shneier makes use of a pyramid data structure for encoding of image features [43].

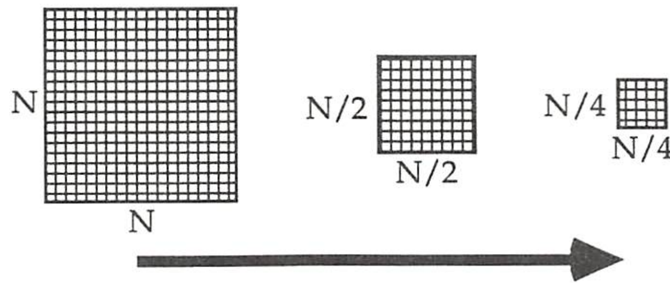

Figure 2.2. A Pyramid Data Structure

Not all data structures are formed by simple averaging of the neighborhood. Gaussian and Laplacian pyramid data structures have also been studied for multiresolution research. As defined by Burt in [18], a Gaussian pyramid is a series of images G_l and is formed by taking a weighted average $G_l(x,y)$ of the neighborhood using the relation

$$G_l(x,y) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n)\, G_{l+1}(2x+m, 2y+n) \tag{2.24}$$

where level $l \in [1,N]$ for an N level pyramid and w(m,n) is a separable, normalized, and symmetric kernel. The series of images starts with the original image G_N.

The inverse of the previous relation is

$$G_{l,0} = G_l \tag{2.25}$$

$$G_{l,k}(i,j) = 4 \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n)\, G_{l,k=l}\left(\frac{i+m}{2}, \frac{j+n}{2}\right) \tag{2.26}$$

where $G_{l,k}$ is the image obtained by applying Eq. (2.26) to G_l k times. Note that only the terms for which (i+m)/2 and (j+n)/2 are integers contribute to the sum in Eq. (2.26). G_{ll} is the same size as the original image.

A Laplacian data structure L_l represents a bandpass filter of the image and is defined in terms of the Gaussian structure by

$$L_l = G_l - G_{l-1,l} \tag{2.27}$$

where G_l and $G_{l-1,l}$ are defined by the previous equations and $L_N = G_N$.

In the absence of noise, the original image can be completely recovered from the Laplacian using

$$G_0 = \sum_{l=1}^{N} L_{ll}. \tag{2.28}$$

where L_{lk} is obtained by expanding L_l k times using relations similar to Eqs. (2.25) and (2.26), substituting L for G.

Clark and Lawrence makes use of zero crossings in a Laplacian pyramid to find boundaries and even propose a systolic architecture for implementation [44].

2.3 Processing and Recognition Cones

The processing cone model of image processing provides for hierarchical control of images of varying resolution. Figure 2.3 shows the three modes

of processing in the cone: reduction, iteration, and projection. Tanimoto [45], Hanson and Riseman [6], and Nazif and Levine [46] have used the processing cone model.

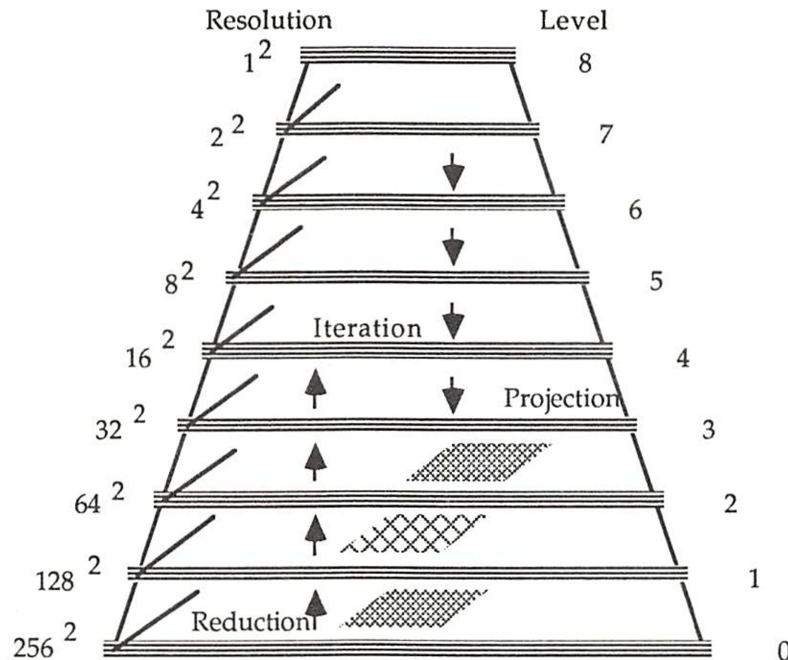

Figure 2.3. Processing Cone Model

In the processing cone model of computation, the pyramid structure is used only for low level image processing tasks. All processing is still "image-in, image-out."

Recognition cones are the next step in evolution of processing cones. Recognition cone perceptual systems (described in [47], [8], and [48]) are a collection of modular, massively parallel procedures to examine global aspects of an image. The processing flow in a recognition cone suggests a pyramid architecture for efficient execution. Uhr describes the processing flow as [8]:

1. Information is input into a large 2-dimensional retinal array.
2. One or more processes are applied either in parallel or serially.

3. Whenever desired, the output of a process is merge into the parent array.
4. Whenever desired, parents send information to children.
5. Whenever desired, processors send information to one another.
6. These kind of processes continue to be executed, at every level of the pyramid.

The SCOOP pyramid provides an excellent environment in which to imbed recognition cone processes. The SCOOP pyramid is flexible, reconfigurable, and programmable. It can be the prototype architecture for the investigation of recognition cone concepts.

In this dissertation, the SCOOP pyramid is configured as a conventional pyramid architecture. That is, the architecture with a constant factor of reduction between levels (e.g. a factor of 4), four child processors, and one parent processor. Other pyramid structures such as linked pyramids, one with a non-constant factor of reduction, or different number of child processors for each processor will perform differently for each of the tasks, but will exhibit the same properties of the hierarchical structure [8].

Most pyramid architectures operate in an SIMD mode. The SCOOP pyramid operates in a multiple SIMD mode (MSIMD). That is, each level of the pyramid is an SIMD mesh connected array, but the levels of the pyramid operate in an MIMD fashion for inter-level communication. Each level is homogeneous (i.e. all processors are identical). However, the higher levels of the pyramid have more powerful processors.

2.4 Hierarchical Processing and Multiresolution Processing

Much of the work done with pyramids, both the architecture and the data structure, has been multiresolution processing. Multiresolution processing allows for the processing of the image data at various levels of

resolution. Often an algorithm can produce intermediate results quickly when done at a reduced (coarse) resolution version of the original image. The results from this provides guidance when processing at the original (fine) resolution.

Hierarchical processing includes a wide range of tasks that are well integrated together. This includes feedback, iterative processing, and higher level tasks to control lower level ones. For example, Levine proposed to use a knowledge-base to control lower level image processing [4]. Hanson and Riseman's work [6, 12] makes use of higher level tasks to control lower level tasks. In this case, a processing cone paradigm is used for the iconic processing. Features are extracted from the processing cone. An image specific model (STM - Short Term Memory) is formed and processed with a general knowledge model (LTM - Long Term Memory).

Currently, researchers at the University of Massachusetts are designing a hierarchical architecture known as the Image Understanding Architecture (IUA) [34]. The architecture consists of 3 levels of two-dimensional mesh-connected arrays of processors. The top level consists of 8×8 array of LISP processors. The next level consists of 64×64 array of synchronous-MIMD 16-bit processors. The bottom level consists of 512×512 SIMD array of 1-bit processors. This bottom level of the IUA is also known as the CAAPP (Content Addressable Array Parallel Processor) and is documented in [3]. Being a hierarchy of 2-D meshes, the IUA resembles a "truncated pyramid." The researchers at the University of Massachusetts are currently building a hardware prototype of the IUA scaled down by a factor of 64:1. Weems et al. list the requirements for vision architectures as [34]:

1. the ability to process both pixel and symbolic data,
2. a fast processing rate for huge amounts of sensory and intermediate level data,

3. the ability to transform an image into a set of meaningful symbols that describe it,
4. the ability to select particular subsets of the data for varying types of processing,
5. feedback mechanisms that allow focusing of attention and data-directed processing, without having to dump the image to some "host" for external evaluation,
6. multiple levels of representation and stages of processing are essential and require very different types of processing elements, and
7. fine grained and high speed communication and control is required both among the processes at each level and between the different processing levels.

The SCOOP pyramid architecture satisfied these requirements. It processes both pixel and symbolic data (requirement #1). The processing is quite fast (requirement #2) and a symbolic representation of iconic data can be derived (requirement #3). Various algorithms can process only a subset of the total data set at any one time (requirement #4) and the high-low and low-high direction of processing provides for feedback without the use of another architecture (requirement #5). The pyramid processes data at varying levels of abstraction (requirement #6) and there exists fine grain, high speed communication and control between the levels (requirement #7). All of these attributes are shown in the chapters to follow.

The pyramid architecture is not claimed to be an optimal architecture for vision processing, but it does meet the previously listed requirements for a vision architecture. A pyramid is a specific topology for hierarchical architectures and other hierarchical architecture can conceivably provide for better performance. It still remains to be seen which architecture provides the best performance. Until the IUA prototype is constructed, the exact performance of the architecture processing real data is unknown.

The SCOOP pyramid is a working prototype with some performance results and processed images. The differences and tradeoffs between the various stages in the design of architectures and hardware/software prototypes are presented in the next chapter.

Chapter 3 The Prototype

The method used to model and construct a working prototype of the pyramid architecture is described in this chapter. The complexity of such a massively parallel architecture requires a suitable environment to develop a prototype. A pyramid with a bottom level of 512×512 processors requires well over 300,000 processors to simulate. An object oriented modeling approach is chosen using the Smalltalk-80[1] system on a Sun 3/110 workstation. The Smalltalk system provides an interactive environment to describe, construct, and program the prototype in a very efficient manner. This Smalltalk system (known as PS[2]) was developed by ParcPlace Systems [49].

3.1 Building a Prototype

A dictionary definition of the word prototype [50] is "An original type, form, or instance that serves as a model on which later stages are based or judged." The reason people build prototypes is that they are simpler, cheaper, and more flexible than the actual system that they model, and thus provide a disposable basis for exploring aspects of a proposed engineering effort.

In this research, building an actual pyramid architecture of significant size was infeasible. This limitation does not prohibit the study of vision algorithms for this architecture. Software prototypes have many advantages over hardware prototypes: they are less expensive, more flexible, and quicker to develop. The advantage of a hardware prototype is performance. The hardware prototype is generally a scaled-down version of the actual system it models. Nevertheless, the cost of designing, wiring, and troubleshooting boards is still significant.

---

[1]Smalltalk-80 is a trademark of Xerox and will hereafter be referred to as Smalltalk.
[2]PS is a trademark of Xerox.

The software prototype offers great flexibility. The structure or the size of the architecture can be changed with a small effort (see Chapter 6). The methods of building prototypes used in this work can assist future work in architecture studies. Prototypes of other architectures, including optical architectures, can be built and studied. This will be discussed in future chapters and especially in Chapter 6.

The difference between a software prototype and a simulation must be pointed out. A simulation represents the data structures and processing of those structures. A software prototype is an abstract description of the actual architecture. One constructs the software prototype to have a virtual architecture in which to program, analyze, and run simulations upon.

Architectures can be analyzed by many "levels of reality," such as

1. a theoretical analysis of the architecture,
2. a software simulation,
3. a software prototype,
4. a scaled-down hardware prototype, and
5. the actual architecture.

These different levels represent the various stages that a designer can use for analysis of various computer systems. Usually, they are performed in the order listed. Analyzing and describing architectures at each of these various levels have their own advantages and disadvantages. These tradeoffs are now discussed in the following subsections.

3.1.1 Theoretical Analysis Method

A theoretical analysis of an architecture consists of a mental abstraction or mathematical description of the architecture. A researcher describes the topology of the processors and how they interconnected and thus forms an analytical model for the architecture. Equations (2.1) through (2.23)

describe the pyramid interconnection scheme. This method has the advantages that it is simple to perform and very inexpensive. It is really the first step in designing or analyzing any architecture. The drawbacks to such an approach is that there are many pitfalls and stumbling blocks that are not considered in such an analysis. Also, detailed programming of an architecture from just a mental (and even mathematical description) is extremely difficult.

3.1.2 Software Simulation

Most real world situations are difficult to analyze using an analytical model. In fact, an accurate analytical model is often difficult to derive. Once the theoretical analysis is performed, a simulation of such an architecture is often the next step. In this case, a computer program evaluates a model numerically over a time period [51]. Often commercially available simulation packages can be used. This method has the advantage that actual timing results of processing "real data" are produced.

A simulation does not have to be functional. That is, there is not necessarily a functional equivalent for each object in the architecture that is represented in the simulation. A simulation can just be a timing model that is driven to produce statistics about algorithmic performance. For example, many algorithms on a 2-D mesh-connected array can be simulated using a spreadsheet program. Certainly, the spreadsheet does not represent a functional representation of each of the processors, their communications paths, communication protocols, local memory usage, instruction set, and so on.

3.1.3 Software Prototype

A software prototype is an abstract representation of the actual architecture. Software prototypes have functional equivalents to the architecture that

they model. Individual processors and ports are constructed and interconnected. These are actual objects (in the software sense) and each represent an actual component in the final architecture. A prototype is then used as a "testbed" in which to run the simulations that were described in the previous subsection. The advantage of using a prototype as a testbed for simulations is that all overhead processing is forced to be considered. Also, the individual processors are actually *programmed*, which facilitate the design of algorithms as well as the programming of the actual hardware. If the actual architecture which the prototype models actually exists, then the prototype is said to emulate that architecture.

3.1.4 Hardware Prototype

The next step is to build an actual hardware prototype. Currently, there are only 2 known hardware prototypes of pyramid architectures: the University of Washington pyramid (5 levels) [37] and the George Mason University pyramid (5 levels) [36]. This step is necessary in the actual construction of complicated hardware. A designer needs to test all the hardware: power supplies, faulty components, bad interconnections. Getting rid of all the bugs in the hardware often takes longer than the actual construction of the hardware. Also, this is often limited to a scaled-down version of the proposed hardware. The two existing hardware prototypes of a pyramid architecture have only 5 levels. This maps the bottom level into only a 16×16 image.

In general, both types of prototypes (hardware and software) can be used as a testbed for performing simulations.

3.1.5 The Actual Architecture

Finally, the product of all the design, analysis, simulation, and prototype work is the actual full-sized architecture. There is no substitute for this level of reality. Even so, each of the previous steps in the design of the

architecture allows the designer to reflect on the design at earlier and less costly stages.

3.2 Object Oriented Programming

The method chosen to model and construct a prototype of this architecture is object oriented. In an object oriented approach, one describes objects and the methods that they perform. Some objects represent the specific pieces of the architecture and the methods describe the operations they perform.

A non-object oriented approach uses specific data structures to represent the items within the architecture. The data structure must change each time the characteristics of modeled items are changed. The code that represents the functions of this item must be updated with each and every change. There is much redundant code involved for each of the different types of objects.

3.2.1 What is Object Oriented Programming?

Object oriented programming has its own terminology and is radically different from conventional procedural languages. A programmer must get used to sending "messages" to "objects" instead of performing procedures on data structures. The idea of objects sending messages to one another is used directly in the modeling of the architecture.

Some terminology used in object oriented programming is listed as follows:

Object — An object is a component of the system. It is a totally encapsulated set of data that can receive messages, perform methods associated with these messages, and send messages to other objects. All objects belong to a class.

Message	When a message is sent to an object, the object (known as the receiver of the message) will select and perform the method that corresponds to the message.
Method	A method is a sequence of instructions (i.e. messages sent to objects) to perform a task. The result of the method is an object and is returned in response to the message.
Receiver	The receiver of a message is the object that the message is sent to.
Class	A class describes the characteristics of the objects that are members of the class. Those objects are known as "instances" of the class. Objects "know" which methods correspond to particular messages because of the description of the class they belong to.
Subclass	A subclass is itself a class. It inherits all methods and variables of the class of which it is a subclass.
Superclass	Likewise, if class **A** is a subclass of class **B**, then class **B** is the superclass of class **A**. A class, its subclasses, and its superclass make up its class hierarchy.
Instance Variables	These are variables (whose values are themselves objects) that are private to an object. All objects of a class have the same set of instance variables, but the values they take on are private to each object.
Class Variables	These are variables that are shared by all instances of the class. That is, they are global to all members of the class.
Inheritance	An object that is a member of a class inherits all variables and methods of its superclass, and the superclass's superclass, and so on. A class can override a method that it inherits. That is, a class can change the method that corresponds to a message that an object might receive. Multiple inheritance is not considered here so all classes have only one superclass.

In this dissertation, different fonts are used to distinguish between the different parts of the object oriented description. Bold type is used for class names (e.g. class **Pyramid**) and a non-proportional, typewriter font for the names of messages (e.g. `performConvolution`). Also, global names are always capitalized and local names start with lower case letters. Therefore, class names always start with capital letters. The names of messages always start with lower case letters. Note that descriptive names are encouraged. Names with multiple words are appended together with the first letter of each word capitalized (with the possible exception of the very first letter). These are the conventions used in the Smalltalk language and community.

Programming in an object oriented language involves describing classes. That is, describing the instance and class variables and defining the methods that perform the tasks associated with the messages. This is done in Smalltalk using an interactive environment suited for object oriented programming [52].

A very simple example illustrates these language features. Suppose one wants to create a program that will perform operations and compute features on different geometric figures. The first step is to create a class structure. Figure 3.1 shows a simple class structure.

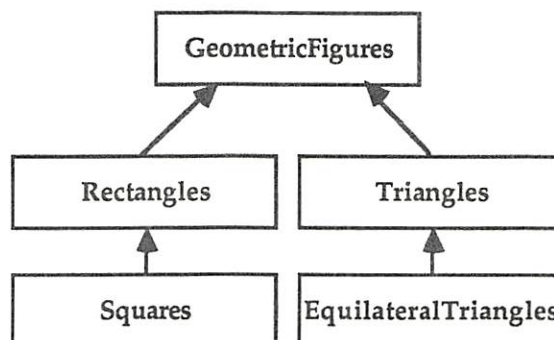

Figure 3.1. Class structure of **GeometricFigures**

3.2.2 Message Passing

One starts by creating a class structure, beginning with the class **GeometricFigures**. Instance variables are used to describe the location of the vertices of the figure. Suppose that one wants to compute the area of a geometric figure. Further, suppose that this method is called `area` and it returns an object which is a number whose value is the area of the receiver of the message `area`. The statement is

```
theArea ← aFigure area.
```

This statement sends the message `area` to aFigure (an instance of **GeometricFigures**) and assigns the area (an instance of **Number**) to the variable theArea. An example of how to accumulate the sum of the areas of a collection of figures is

```
totalArea ← 0.
1 to: numberOfFigures do: [:figureNumber |
    totalArea ← totalArea + (theFigures at: figureNumber) area].
```

These statements work in the following way. First, totalArea is initialized to zero (an instance of **Number**). The variable numberOfFigures refers to the total number of figures in the array theFigures. The variable theFigures is an instance of **Array** containing an ordered collection of **GeometricFigures** (arrays don't have to contain only numbers). This second statements executes the block (the statements between the square brackets) numberOfFigures times. This is similar to a DO LOOP in FORTRAN. The variable figureNumber is increased upon each iteration of the control loop. It is the index into the array theFigures. The area is then computed and added with the totalArea. Note that parentheses can control the order in which messages are sent. Object oriented programming must always be thought of as objects sending messages to one another. Even the addition in the last statement should be thought of

as the message "+" being sent to a number (totalArea) with the argument of a number (equal to the area of the figure being indexed by figureNumber) with the final sum being referenced by totalArea. Figure 3.2 shows an abstract view of this process. The object (known as anObject) sends the message `area` to aGeometricFigure (an instance of class **GeometricFigures**). The object aGeometricFigure will perform the method associated with the message `area` and return the result aNumber, which is an instance of class **Number**. The black arrow refers to the sending of a message and the gray arrow refers to the instantiation (i.e. creation) of a new object.

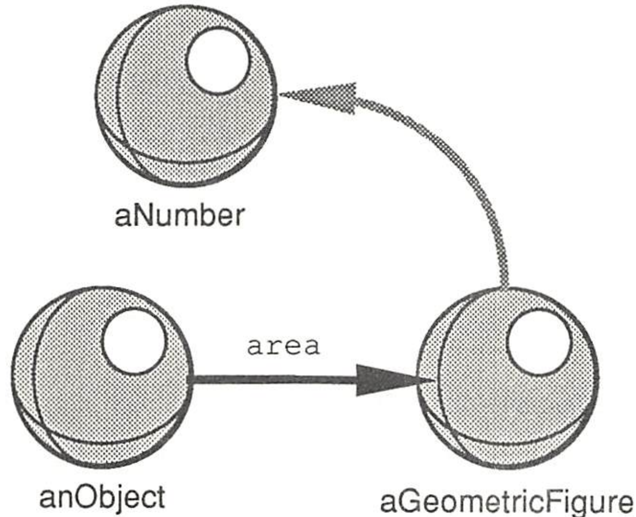

Figure 3.2. Abstract representation of objects passing messages

3.2.3 Overriding Inherited Methods

A fairly complex method is needed to perform the steps necessary to compute the area of an arbitrary geometric figure. If some of the types of figures to be encountered are known, subclasses of **GeometricFigures** (e.g. **Rectangles** and **Triangles**) that describe certain types of geometric figures can be created. A three sided geometric figure will be an instance of class **Triangles** as well as an instance of the **GeometricFigures**. Similarly for the class **Rectangles**. These subclasses will inherit all the instance

variables (i.e. vertices) of their superclass and the method `area`. There are simple methods to determine the area of rectangles and triangles so one can override the method `area` with a newer method to take advantage of this knowledge. One can create further subclasses of these to take advantage of knowledge that edges are of equal length, thus creating classes **Squares** and **EqualateralTriangles**. Therefore, one overrides the method `area` to implement an even simpler algorithm for finding the area. In general, it is easiest to describe a class by: finding an existing class very close to the new class, make the new class a subclass of the existing one (thus inheriting from the existing class), and then overriding or adding new methods to the new class.

3.2.4 Advantages of Object Oriented Programming

One of the biggest advantages of object oriented programming is the reusability of code that comes from inheritance. When subclasses are created, the methods are inherited from their superclass. Presumably, one does not override all the methods of the superclass. Therefore, much of the code is reused. Also, when a method is changed for a class, all the subclasses inherit the new method. That is, the change is propagated throughout the class structure.

Another advantage to modeling with object oriented programming is the encapsulation property of an object [53]. Think of the receiver of a message as the producer of a service and the sender of the message as the consumer of a service. The consumer of the service knows what services the producer will offer, but knows nothing about how the service will be performed. The methods and data that the producer uses are totally encapsulated (i.e. hidden) within the producer and thus protected. In the previous example, it is the particular geometric figure that determines how to compute the area, not the source requesting the area. This provides a complete and protected modular system.

3.3 Protocols for Multiple Processes and Simulations

Modeling and constructing prototypes of a parallel computer architecture requires the maintenance of multiple processes. These processes must be aware of each other and their cooperation must be explicitly defined. These processes (each one allocated to a processor) are dependent upon one another in that data dependencies exist between them. Often a calculation requires the results from another process and must wait until this result is available. André et al. discuss two types of cooperating processes: centralized and distributed [54]. The distributed implementation makes use of queues (ports) between the processes that perform the necessary handshaking. This localized handshaking avoids the practical hardware problem of clock skew while increasing the complexity of the communication ports. The distributed implementation adapts much easier to the different levels of processing requirements (and different levels of processor power) within a collection of processes (the architecture). For this reason, the distributed implementation of process management was chosen.

The communication of these processes require protocols for mutual exclusion of the data. Raynal discusses the mutual exclusion problem using both the centralized and distributed frameworks [55]. Two solutions are presented: one based on state variables and another based on message communication. The message communication methods were chosen since they match the object oriented paradigm of Smalltalk precisely.

A substantial amount of work has been done by the developers of Smalltalk to provide assistance with simulations of systems. Provided are such classes as **Process**, **Semaphore**, **Delay**, and **SharedQueue** for describing the workings of multiple independent processes. Also, classes that are provided for simulation support are **Simulation**, **SimulationObject**, and **EventMonitor**. These are all described in detail in [56]. The protocols of

these classes that are used in the creation of this prototype are described in Subsections 3.3.1 through 3.3.7. This is not intended to completely reproduce that information which is already available in the reference; it only provides a short synopsis of each class.

3.3.1 Class Process

A process is an object that represents a sequence of actions that are described by Smalltalk expressions. These actions can be carried out independently of other processes. There are messages to fork, suspend, resume, and terminate these processes. In this prototype, instances of class **Process** enumerates the sequence of events that a processor will undergo. (Note that the class **Process** is completely different from the class **Processors** that are described later.)

3.3.2 Class Semaphore

The class **Semaphore** is used to send signals between independent processes. Together with class **SharedQueue** (to be discussed later), the necessary handshaking between the processors of the architecture takes place.

3.3.3 Class Delay

A delay allows a process to suspend itself for a specific period of time. When the time is up, a semaphore is sent to the process to cause it to resume.

3.3.4 Class SharedQueue

The class **SharedQueue** is an important class in the prototype. This is how the processors within the architecture pass data and instructions between one another. The shared queue provides the necessary handshaking to synchronize the communication between the asynchronous, independent

processors. There will be two shared queues between every two processors that are interconnected. Each shared queue is unidirectional, so two are needed to provide bidirectional capability.

Note that this architecture requires no global clock. Many other architectures make use of a global clock to provide processor synchronization. The global clock (and thus problems such as clock skew) are eliminated by the use of these queues.

To save space in the simulation, class **SharedQueue** has been replaced by class **UnidirectionalPort**. This class is a stripped down version of class **SharedQueue** and provides only that which is necessary for this work. For instance, the length of each queue is only one (a port with no buffer) and thus space was saved by not providing a queue of arbitrary length. Class **UnidirectionalPort** will be discussed later in detail.

3.3.5 Class Simulation

Class **Simulation** controls the set of objects that make up the simulation. This class maintains the simulation clock, the queue of events that are waiting to be processed, and maintains references to all the objects in the simulation. Class **Simulation** understands messages: to initialize the simulation, define a schedule whereby the objects enter and leave the simulation, schedule and coordinate resources, and to finish and clean up the simulation. This type of discrete-event simulation is known (in [51]) as a "next-event time advance simulation" as opposed to a "fixed-increment time advance simulation." A queue of events is a collection of events that are ordered by the time that they should be executed. In a fixed-increment time advance method, the clock is advanced by a fixed amount and then all events that are scheduled to be executed by that time are then run. In the next-event time advance method, the queue of events is a collection of events that are scheduled to execute. Each event has a tag that associates

itself with a time on the simulation clock. The events are ordered by this time. When all events for time t_i are executed, the clock is then set to time t_j, where t_j is the lowest value of all time tags in the event queue but still greater than t_i.

3.3.6 Class SimulationObject

Class **SimulationObject** represents the components of a simulation. Consider a simulation of traffic. **Automobiles, Trucks,** and **Buses** are all road vehicles and can all be subclasses of **SimulationObject**. The reason they should be described by different classes is that automobiles, trucks, and buses all interact with each other in different ways. Autos tend to move more quickly than the other classes, buses tend to make many stops, and trucks will stop for lengthy times to load or unload something. Other classes can then enter into the simulation such as **RailRoad**, **FuneralProcession** (a collection of **Automobiles**), and so on.

Class **SimulationObject** understands messages for: initialization, simulation control, and task language. These objects report the fact that they are entering the simulation to the ActiveSimulation (an instance of **Simulation**) and release themselves upon exiting the simulation.

3.3.7 Class EventMonitor

EventMonitor is a subclass of **SimulationObject**. This class adds messages and overrides some inherited ones to help document what is going on within a simulation. Basically, this class helps provide a log file of events that take place during the simulation.

3.4 SCOOP Class Protocols

The class protocol structure for the prototype is divided into two class hierarchies. The first class hierarchy is the class of processors that make up the architecture. All processors within the architecture are instances of one

of the classes within this class structure. Figure 3.3 shows the class structure of all the processors.

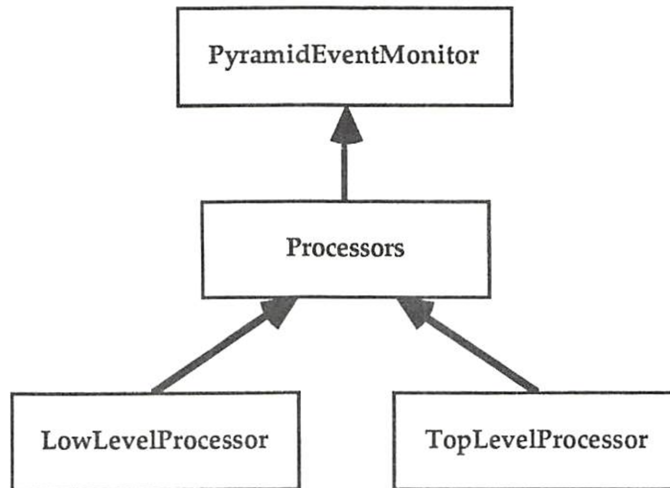

Figure 3.3. Class hierarchy of processors

The class **PyramidEventMonitor** is the abstract class which describes the attributes of objects within a simulation that monitor and report on events. It itself is a subclass of **EventMonitor**, which is a subclass of **SimulationObject**. All of the processors within the architecture are instances of **Processors** (or its subclasses). The subclasses **LowLevelProcessor** (for processors at the bottom level) and **TopLevelProcessor** accurately reflect the different characteristics and capabilities of the different types of processors that make up the SCOOP pyramid.

Figure 3.4 is a class hierarchy of some of the simulation classes. The class **Pyramid** is a subclass of **Simulation**. The class **Pyramid** refers to the architecture itself, as well as the simulation to be performed.

All the subclasses of **Pyramid** are the actual simulations. They inherit all the variables and methods of class **Pyramid** so that they only describe the different situations for each simulation. Note that the actual algorithms to perform are described in the processor classes, not in the simulation classes

since it is the processors themselves that must know what steps to perform.

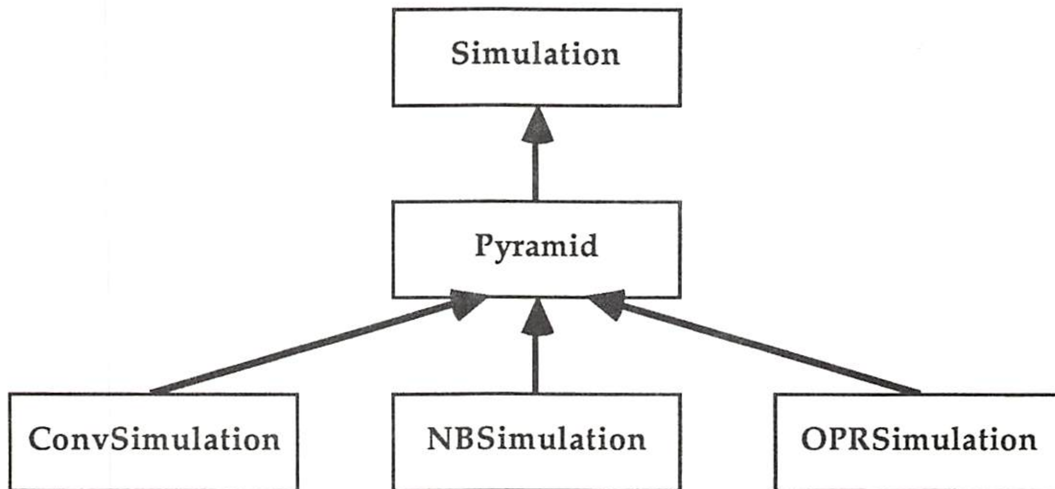

Figure 3.4. Class hierarchy of simulations

In the following descriptions, a specific format is used to document the class. For example, the protocol for class **GeometricFigures** might look like

GeometricFigures instance protocol

---

feature extraction

```
area
```

Computes and returns the area of the receiver.

GeometricFigures class protocol

---

instance creation

```
new: aList
```

Return an instance of **GeometricFigures** with vertices at aList.

The class protocols describe methods used to create instances of the class. The lines "feature extraction" and "instance creation" are category names.

The following class protocol descriptions are not complete. Only the more important messages are included in order make this list more readable. Also, subclasses inherit methods from their superclasses and these methods will not be listed again unless they are overridden by newer methods.

3.4.1 Class UnidirectionalPort

This class is a stripped down version of class **SharedQueue**. The protocol for this class is

UnidirectionalPort instance protocol

---

initialize-release

`release`

Releases and empties the port.

accessing

`next`

Returns the object waiting in the port. Sender of message will be suspended until something is returned.

`nextPut: aValue`

Send aValue through the port to the processor at the other end.

UnidirectionalPort class protocol

---

instance creation

`new`

Return a new instance of UnidirectionalPort.

3.4.2 Class PyramidEventMonitor

The class **PyramidEventMonitor** describes all objects within the simulation that report and monitor events. It is a subclass of **EventMonitor**. The following is the protocol for class **PyramidEventMonitor**

PyramidEventMonitor instance protocol

scheduling

`startUp`

Initialize instance variables. Inform the simulation that the receiver is entering it, enter a log entry into logfile, and then initiate the receiver's tasks.

task language

`holdFor: aTimeDelay reason: aString`

Record entry into log file stating aString as reason that receiver will delay for aTimeDelay.

printing

`printOn: aStream`

Print label onto aStream which identifies receiver.

3.4.3 Class Processors

Class **Processors** is a subclass of **PyramidEventMonitor**. All processors within the architecture are instances of this class. This following description documents the messages used by the processors to configure themselves and perform operations for the simulation. Some of the messages refer to algorithms that will be discussed in the next chapter.

simulation steps

`performConvolution`

Perform the steps necessary to complete a convolution.

`performNBSimulation`

Perform the steps necessary to complete the Nevatia-Babu edge detector.

`performOPRIteration`

Perform the steps necessary for one single Ohlander-Price-Reddy iteration.

`performOPRSimulation`

Perform the iterations necessary to complete the Ohlander-Price-Reddy segmentation process.

simulation control

`tasks`

Determine the type of simulation and continue on the proper route.

control tasks

`broadcastTasks`

Broadcast instructions down throughout the architecture starting from the processor at the top.

histogram tasks

`accumulateBinCounts`

Remain idle while each **LowLevelProcessor** accumulate bin counts of gray levels.

`sendHistogramsUp`

Receive the subhistograms from children, combine them to generate larger subhistogram, and send this to parent.

analysis tasks

`analyzeHistograms`

Remain idle while **TopLevelProcessor** analyze histograms.

threshold tasks

`broadcastThreshold`

Receive threshold rule from parent and send this to the children.

merging tasks

`mergeRegions`

Remain idle while each **LowLevelProcessor** merges regions.

hardware setup

`connectPort: direction to: port1 from: port2`

Connect receiver to the proper ports to communicate with neighbor in the specified direction.

`initializeProcessorsPorts`

Initialize the processors ports before setup.

data transfer

`getFromEast`

Obtain packet from eastern neighbor through the shared queue connecting us. Returns nil if there is no neighbor.

`getFromNorth`

Same, but from northern neighbor.

... ...

`putToEast: aValue`

Put aValue to my eastern neighbor through the shared queue connecting us. Do nothing if there is no neighbor.

`putToNorth: aValue`

Same, but for northern neighbor.

... ...

inquiries

`getProcessorID`

Returns a string describing the processor ID (i.e. row, column, and level).

`whoIsMyNorthEastChild`

Returns the coordinates of my child to the northeast.

... ...

`whoIsMyParent`

Returns the coordinates of my parent processor.

Processors class protocol

---

instance creation

`new: x andY: y andL: level`

Returns an instance of **Processors** with coordinates (x, y, level).

class initialization

`initialize`

Initialize class variables of **Processors**.

`initNumberOfLevels: height`

Initialize the height of the pyramid. That is, all **Processors** know the size of the architecture.

3.4.4 Class LowLevelProcessor

Class **LowLevelProcessor** is a subclass of **Processors**. This subclass further refines the description of the processors to that which is particular to the

processors at the bottom level of the pyramid. Some of the messages refer to algorithms that will be discussed in the next chapter.

LowLevelProcessor instance protocol

control tasks

`broadcastTasks`

Receive instruction from parent and execute it.

histogram tasks

`accumulateBinCounts`

Accumulate bin counts of gray levels of columns keeping track of regions separately.

`sendSubHistogramsUp`

Send the subhistograms to the parent. Do this for each region in a pipeline fashion.

threshold tasks

`broadcastThreshold`

Receive threshold rule from parent. Execute the threshold rule.

merging tasks

`mergeRegions`

Merge regions to remove isolated/small regions.

accessing

`pixelValue`

Return current pixel value associated with the processor.

`regionLabel`

Return the region label associated with the processor.

local computations

`find3Neighborhood`

Communicate with neighbors to determine the processor's 3x3 neighborhood.

`find5Neighborhood`

Communicate with neighbors to determine the processor's 5x5 neighborhood.

`performConvolution: sizeOfWindow`

Perform the convolution steps for this processor with the KernelWindow.

`performNBComparison`

Perform the comparison steps for the Nevatia-Babu Edge Detection.

`performNBConvolution`

Perform the convolution steps for the Nevatia-Babu Edge Detection.

`performNBLineDescription`

Perform the line description steps for the Nevatia-Babu Edge Detection.

data manipulations

`atWindow: theRow andY: theColumn`

Return the value in the neighborhood window at location (theRow, the Column).

`atWindow: theRow andY: theColumn put: aValue`

Put aValue into the neighborhood window location (theRow, theColumn).

`initPixelValue: aValue`

Initialize this processor's pixel value to aValue.

`initPixelValue: aValue inDataBamd: bandNumber`

Initialize this processor's pixel value to aValue for data band bandNumber.

simulation steps

`performOPRIteration`

Perform the steps necessary for a single Ohlander-Price-Reddy iteration.

LowLevelProcessor class protocol

---

instance creation

`new: x andY: y andL: level`

Return an instance of **LowLevelProcessor** at location (x, y, level).

class initialization

`initialize`

Initialize the class variables for all instances of class **LowLevelProcessor.**

`initImageSize: imageSize`

Initialize the size of the bottom level (i.e. the image).

`initKernel`

Initialize the kernel for convolutions.

`initNBKernels`

Initialize a full set of NB kernels.

`initNeighborhoodSize: size`

Initialize the size of the neighborhood window to size.

3.4.5 Class TopLevelProcessor

The class **TopLevelProcessor** describes the processor at the top of the pyramid. It is a subclass of **Processors**. The power of the processors increase as the size of the level decreases. The top level processor is the most powerful one and can control the rest of the architecture by broadcasting instructions. In general, the more complex operations are

reserved for the top few levels of the pyramid. Some of the messages refer to algorithms that will be discussed in the next chapter.

TopLevelProcessor instance protocol

control tasks

`broadcastTasks`

Send an instruction to the children.

histogram tasks

`sendSubHistogramsUp`

Receive the subhistograms from the children, combine these to form a complete histogram, and save them for further processing.

analysis tasks

`analyzeHistogram: aHistogram for: aRegion forDataBand: bandNumber`

Analyze the histogram associated with region aRegion in data band bandNumber.

`analyzeHistograms`

Analyze the complete set of histograms.

`chooseThreshold: theIntervals forRegion: region forDataBand: aBand`

A score is computed for each interval set (of a region). This score is 1000*(peakHeight-higherShoulderHeight)/peakHeight. Choose the interval (of all the histograms of spectral components for that region) that has the highest score. Return the threshold rule.

`comparePeakAreas: theIntervals inHistogram: aHistogram`

Peak area is now compared with an absolute threshold and with a relative threshold, representing a percentage of the total histogram area. Only peaks larger than these thresholds are retained.

`condenseIntervals: theIntervals`

Shift intervals to left to fill up slots left empty by intervals that merged into other intervals.

`finalScreeningTest: theIntervals`

A final screening is performed to reduce the interval set to a specific number of intervals. This is done by repeatedly merging regions with low peak-to-shoulder ratios until only the correct number of valley remain.

`globalHistogramTest: theIntervals`

The second highest peak and those peaks whose height is less than a percentage of it are merged. The lowest interior valley is then found, and any interval whose right shoulder is more than a certain number of times that valley height is merged with its right neighbor.

`mergeInterval: intervalNumber for: theIntervals`

Merge interval into neighbor with higher shoulder.

`peakToShoulderRatioTest: theIntervals`

Merge any intervals into dominant neighboring interval if the peak-to-shoulder ratio does not exceed a certain parameter.

`screenIntervals: theIntervals`

Scan across the intervals in order to merge some of the intervals into others.

`smoothHistogram: aHistogram`

Smooth aHistogram with a Gaussian mask.

threshold tasks

`broadcastThreshold`

Send the threshold rule to the children.

simulation steps

`performOPRSimulation`

In between iterations, dump out results into interim files for later examination.

TopLevelProcessor class protocol

instance creation

`new`

Create the top banana!

class initialization

`initialize`

Initialize all the class variables for **TopLevelProcessor**.

3.4.6 Class Pyramid

Class **Pyramid** is a subclass of class **Simulation**. It not only represents the simulation, but the architecture itself. It has references to all the processors and ports (shared queues) in the architecture and can instruct the processors at the bottom level to read in and write out data.

Pyramid instance protocol

accessing

`bottomLevelImage`

Returns a two dimensional array of pixel values from the bottom level of the architecture.

`bottomLevelImageAsArray`

Returns a one dimensional array of pixel value in lexicographical order to write out to file.

`imageRegionsAsArray`

Returns a one dimensional array of region labels to write out to file.

`imageSize`

Returns the size of the image (i.e. the size of the bottom level).

`numberOfLevels`

Returns the number of levels.

`numberOfLevels: height`

Set the number of levels equal to height.

initialization

`defineResources`

Schedule the entrance of all processors into the simulation.

simulation control

`deactivate`

Release the simulation from memory.

`finishUp`

Send a message to all processors for them to release themselves from the simulation.

file access

`dumpImageIntoFile: selectedFileName`

Open a file named selectedFileName and write out the image data to it.

`dumpRegionLabelsIntoFile: selectedFileName`

Same as previous, but for region labels instead of image data.

`loadDataBandIntoPyramid: selectedFileName forDataBand: bandNumber`

Open a file named selectedFileName and load this into pyramid as data band bandNumber.

`loadImageIntoPyramid: selectedFileName`

Open a file named selectedFileName and read the image data into the processors at the bottom level.

Pyramid class protocol

instance creation

`createPyramid: numberOfLevels`

Create a pyramid architecture with numberOfLevels levels.

monitor simulation

`monitorSpaceUsage`

A background process that monitors the consumption of objects and memory.

3.4.7 Class ConvSimulation

Class **ConvSimulation** is a subclass of **Pyramid**. It is the simplest of simulations performed on the prototype.

ConvSimulation instance protocol

accessing

`imageSize`

Return the size of the image (i.e. the bottom level).

initialization

`defineResources`

Define the arrival schedule of processors into the simulation.

simulation control

`finishUp`

Allow the processors to release themselves.

ConvSimulation class protocol

examples

`doExample`

Perform a simulation of the convolution and store results.

instance creation

`new`

Return an instance of ConvSimulation with all class variables initialized.

3.4.8 Class NBSimulation

Class **NBSimulation** is a subclass of **Pyramid**. It is a simulation of the Nevatia-Babu edge detector.

NBSimulation instance protocol

accessing

`imageSize`

Returns the size of the image (i.e. the bottom level).

initialization

`defineResources`

Define the arrival schedule of processors into the simulation.

simulation control

`finishUp`

Allow the processors to release themselves.

NBSimulation class protocol

examples

`doExample`

Perform a simulation of the Nevatia-Babu algorithm and store results.

instance creation

`new`

Return an instance of NBSimulation with all class variables initialized.

3.4.9 Class OPRSimulation

Class **OPRSimulation** is a subclass of **Pyramid**. It is a simulation of the Ohlander-Price-Reddy segmentation algorithm.

OPRSimulation class protocol

examples

`doExample`

Perform a simulation of the Ohlander-Price-Reddy algorithm and store the results.

3.5 The Methods

The complete code of the prototype is not included in this thesis due to its bulk.[3] The methods describe precisely the individual steps for the construction of the prototype and all the steps for the processors and ports for processing tasks to be described.

[3]The complete source to the SCOOP pyramid are available by contacting the author at Tulane University, New Orleans, LA 70118-5674.

The following chapters will focus on the algorithms that were simulated on the prototype, the results of these simulations, and general discussions about this implications of the results.

Chapter 4 The Algorithms

This chapter describes the algorithms implemented on the SCOOP pyramid. It is important to choose a wide range of vision algorithms dealing with data at different levels of abstraction. The intent here is not to actually improve the quality of the algorithms, but to find efficient parallel implementations, to map them onto the architecture, and to compare their performance to the execution of other architectures. The selected algorithms cover a wide range of vision tasks:

1. low level image processing: convolution, Nevatia-Babu edge detection,
2. image analysis: Ohlander-Price-Reddy segmentation (Phoenix version),
3. high level symbolic computation: rule chaining, and
4. a complete scenario that includes a wide range of tasks from low level to high level.

Each of the algorithms are discussed in the following sections. A brief sketch of the pyramid implementation is also given in these sections. Later chapters provide the details of the implementation of these algorithms.

4.1 Low Level Image Processing

The low level tasks are usually local (i.e. they deal with small neighborhoods) and involve simple computations done repetitively throughout the image. Also, the output is usually a data structure of similar size. These tasks can be viewed as "image in, image out" algorithms. A non-exhaustive list of categories of low level image processing tasks include [57]:

1. Image coding
2. Image enhancement
3. Image restoration
4. Image transformations

All of these categories (excluding transformations) are typically local operations. These operations can be viewed as a local neighborhood function Φ (of size $(2n+1)\times(2n+1)$) on the neighborhood of the pixel at location (x,y) as given by

$$\mathbf{image}_{out}(x, y) = \underset{i,j=-n}{\overset{n}{\Phi}} (\mathbf{image}_{in}(x+i,y+j)) \tag{4.1}$$

where the output pixel is a function of pixel values in its local neighborhood. Many low level algorithms are performed by multiple iterations of the process as described by Eq. (4.1).

Because the data to be processed is usually represented by pixel values, the data structures are typically large, regular arrays of these values. If the picture is digitized to 512×512 resolution, then there are 262,144 pixel values. If the image is a multispectral image (e.g. red, green, and blue color components), the there will be 786,432 values represented by a three dimensional data structure. If there are a sequence of these color images arriving at video rates of 30 frames/second, then there will be 23,592,960 values in just one second of image data represented by a four dimensional data structure. The massive quantity of the data should be kept in mind at all times when considering processing requirements.

4.1.1 Convolution

Convolution of an image is a relatively simple yet frequently used algorithm. It is used for image smoothing, edge enhancement, and many other filtering operations. It can also form the basis of more complex

algorithms such as the Nevatia-Babu Edge Detector (discussed in the next section). For this reason, it was chosen as the starting place for the simulation work. Convolution of a k×k kernel with an image is specified by

$$\mathbf{image}(x,y) = \sum_{i=1}^{k}\sum_{j=1}^{k} \mathbf{image}(x-i,y-j) \times \mathbf{kernel}(i,j) \qquad (4.2).$$

The contents of the kernel determine the type of filtering operation to perform. For example, simple edge enhancement can be performed by a gradient operator. The gradient is measured by the following [58]

$$s(x,y) = \sqrt{\Delta_1^2+\Delta_2^2} \qquad (4.3)$$

$$\phi(x,y) = \tan^{-1}\left(\frac{\Delta_2}{\Delta_1}\right) \qquad (4.4)$$

is the magnitude and direction at location (x,y), and where

$$\Delta_1 = f(x+n,y) - f(x,y) \qquad (4.5)$$

$$\Delta_2 = f(x,y+n) - f(x,y) \qquad (4.6).$$

The value of n (known as the "span") is a small value (typically 1). The function f(x,y) is the image function. Figure 4.1 shows the 2×2 Roberts gradient kernels for a span of one [1]

$$\Delta_1 = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \qquad \Delta_2 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Figure 4.1. Roberts gradient kernels

Figure 4.2 shows two other directional kernels used to measure gradients in the 0° and 90° directions

$$\Delta_1 = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad \Delta_2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 4.2. Directional edge kernels

Again, looking at Eq. (4.2), it becomes immediately obvious that the number of operations involved in determining the output value at any one point is proportional to the size of the kernel. A serial computer must perform this convolution for each of the N^2 pixels in the image (assuming an $N \times N$ image). Thus, the serial execution of a convolution of an $N \times N$ image with a $k \times k$ kernel is $O(N^2k^2)$ steps. In other words, the time is dominated by the size of the image. It is desirable to perform the convolution using a method that is independent of the size of the image. The output value at any location (x,y) in the image is a function only of the kernel and the values of its neighboring pixels. The size of the neighborhood is equal to the size of the kernel.

If each processor at the bottom level of the pyramid must determine the output value for its location, then each processor must communicate its value to its neighbors in order to determine the entire neighborhood. Then, each processor convolves this neighborhood with the kernel (as given in Eq. (4.2)). For a 3×3 kernel, the following steps are taken to determine the neighbors of a pixel:

1. Pass the pixel value to the Northern, Southern, Eastern, and Western neighbors through the 4 ports.
2. Receive the pixel values from the Northern, Southern, Eastern, and Western neighbors. These are the pixel values passed from execution of step 1 by the neighbors.
3. Pass the value received from the North to the East and West.
4. Receive the pixel values from the East and West. These are the pixel values from the Northeast and Northwest neighbors, respectively.

5. Pass the value received from the South to the East and West.
6. Receive the pixel values from the East and West. These are the pixel values from the Southeast and Southwest neighbors, respectively.

Figure 4.3 shows the sequence of how the knowledge of the neighborhood is acquired. A filled in square indicates knowledge of the pixel value.

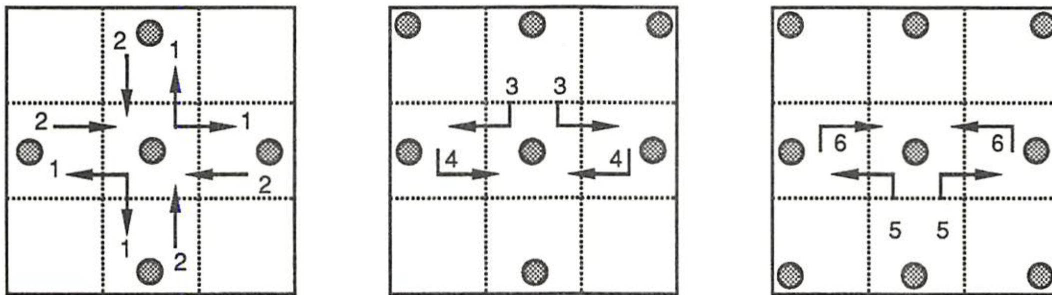

Figure 4.3. Knowledge of pixel's 3×3 neighborhood is acquired

After these six steps, each processor has all the pixel values for the entire 3×3 neighborhood. A very similar algorithm is used to find the 5×5 neighborhood (as done in Section 4.2) for a 5×5 convolution. The steps needed for a 5×5 convolution are just an extension of the 3×3 case:

1-6. Find the 3×3 neighborhood (as previously described).

7. Put the Southwest value to the North, the Northeast value to the South, the Northwest value to the East, and the Southeast value to the West.
8. Get the NorthNorthEast value from the North, the SouthSouthWest value from the South, the EastSouthEast value from the East, and the WestNorthWest value from the West.
9. Put the Southeast value to the North, the Northwest value to the South, the Southwest value to the East, and the Northeast value to the West.

10. Get the NorthNorthWest value from the North, the SouthSouthEast value from the South, the EastNorthEast value from the East, and the WestSouthWest value from the West.
11. Put the South value to the North, the North value to the South, the West value to the East, and the East value to the West.
12. Get the OuterNorth value from the North, the OuterSouth value from the South, the OuterEast value from the East, and the OuterWest value from the West.
13. Put the WestSouthWest value to the North, the EastNorthEast value to the South, the NorthNorthWest value to the East, and the SouthSouthEast value to the West.
14. Get the OuterNorthEast value from the North, the OuterSouthWest value from the South, the OuterSouthEast value from the East, and the OuterNorthWest value from the West.

Figure 4.4 shows the sequence of how the knowledge of the neighborhood is acquired (after the 3×3 neighborhood is acquired). Again, a filled in square indicates knowledge of the pixel value.

The remaining convolution formula can take place in just k^2 steps (where k is the size of the kernel). This is a very efficient algorithm and it is performed in time that is independent of the size of the image. This is a substantial speed-up of the $O(N^2k^2)$ steps needed for the serial algorithm.

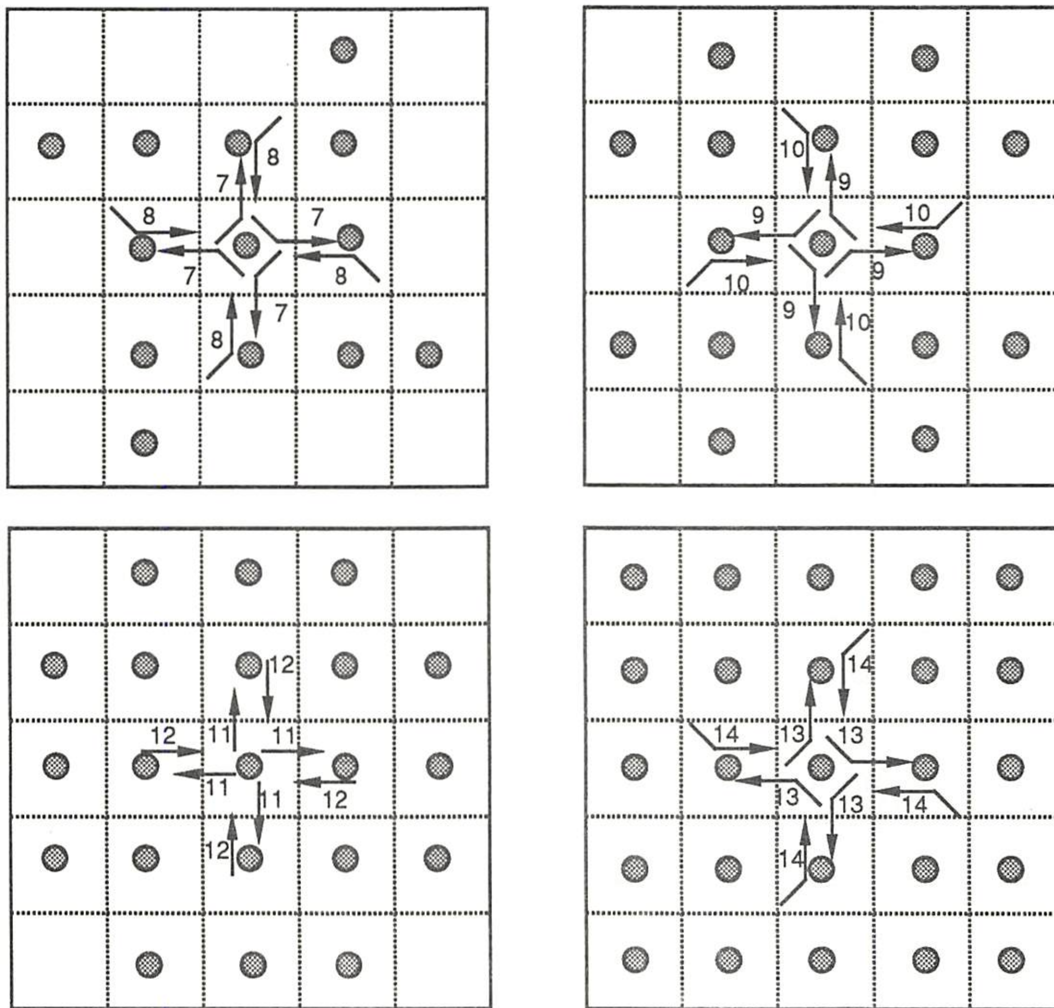

Figure 4.4. Knowledge of pixel's 5×5 neighborhood is acquired

4.1.2 Morphological Operations

Many cellular operations such as erosion, dilation, shrinking, and isolated pixel removal are classified as morphological operations. These techniques were developed formally by Matheron [59] and Serra [60].

Gerritsen and Verbeek [11] describe a method of using table look-up procedures to perform binary morphological operations. This is done, for each pixel, by ordering the bits of the neighborhood into a 9-bit index into the look-up table. The 9-bit index into the table implies that the table is 512 bytes long. There is substantial overhead to load these tables (see

Section 5.7.5). There is a specific look-up table for each morphological operation. The ordering of the neighbors within the 9-bit word starts from the center pixel, then to the southeastern neighbor, circling clockwise around the neighborhood to the eastern pixel. Another way to look at it is that the binary values are multiplied by their corresponding coefficients and summed together (see Figure 4.5). The 3×3 coefficients are

$$\begin{bmatrix} 8 & 4 & 2 \\ 16 & 256 & 1 \\ 32 & 64 & 128 \end{bmatrix}.$$

Figure 4.5. Ordering of 3×3 Morphological Coefficients.

The method for performing these operations in the pyramid are very similar to the convolution operation:

1. The 3×3 neighborhood is found using the steps outlined in Section 4.1.1.
2. The index to the morphological table is found by ordering the binary pixels into a 9-bit word whose place in the 9-bit word can be associated with the ordering as given by Figure 4.5. This 9-bit word is an index into a 512 byte table.
3. The center value is replaced by the value given in the table.

The timings for the different architectures are the same order of magnitude for the 3×3 convolution. Morphological operations are basically look-up table responses to a 3×3 binary neighborhood. Many of these can be performed using multiple iterations to extract certain features. For instance, the TM scenario (in Chapter 5) uses 50 iteration of a topologically-preserving shrinking look-up table in order to realize the medial-axis transform of the water. Results of the morphological operations are presented as part of the two scenarios in Chapter 5.

4.1.3 Nevatia-Babu edge detector

The Nevatia-Babu edge detector [61] is a convolution based form of edge detection. Six convolutions are performed with the image and six 5×5 directional edge kernels and their outputs are compared at neighboring pixels to determine whether or not each pixel is an edge pixel. The steps at each pixel location are [62]:

1. Perform a convolution with six 5×5 directional edge kernels (see Figure 4.6). Save each of the outputs.
2. The direction with the greatest magnitude is the edge direction and its magnitude is its edge magnitude.
3. A pixel is an edge pixel if
   a. its edge magnitude is greater than the edge magnitudes of the neighboring pixels in the direction normal to the edge direction of this pixel (the diagonals approximate a normal direction to a 30° edge),
   b. the edge directions of the two neighboring pixels are within 30° of the center pixel, and
   c. the edge magnitude is greater than a specified threshold.
4. If parts 3a and 3b are true, then the two neighboring pixels are disqualified from being edge pixels.

Figure 4.6 are the edge kernels for the convolutions. They are edge masks in six directions: 0°, 30°, 60°, 90°, 120°, and 150°.

$$\begin{bmatrix} -100 & -100 & 0 & 100 & 100 \\ -100 & -100 & 0 & 100 & 100 \\ -100 & -100 & 0 & 100 & 100 \\ -100 & -100 & 0 & 100 & 100 \\ -100 & -100 & 0 & 100 & 100 \end{bmatrix} \quad \begin{bmatrix} -100 & 32 & 100 & 100 & 100 \\ -100 & -78 & 92 & 100 & 100 \\ -100 & -100 & 0 & 100 & 100 \\ -100 & -100 & -92 & 78 & 100 \\ -100 & -100 & -100 & -32 & 100 \end{bmatrix}$$

$$\begin{bmatrix} 100 & 100 & 100 & 100 & 100 \\ -32 & 78 & 100 & 100 & 100 \\ -100 & -92 & 0 & 92 & 100 \\ -100 & -100 & -100 & -78 & 32 \\ -100 & -100 & -100 & -100 & -100 \end{bmatrix} \quad \begin{bmatrix} 100 & 100 & 100 & 100 & 100 \\ 100 & 100 & 100 & 100 & 100 \\ 0 & 0 & 0 & 0 & 0 \\ -100 & -100 & -100 & -100 & -100 \\ -100 & -100 & -100 & -100 & -100 \end{bmatrix}$$

$$\begin{bmatrix} 100 & 100 & 100 & 100 & 100 \\ 100 & 100 & 100 & 78 & -32 \\ 100 & 92 & 0 & -92 & -100 \\ 32 & -78 & -100 & -100 & -100 \\ -100 & -100 & -100 & -100 & -100 \end{bmatrix} \quad \begin{bmatrix} 100 & 100 & 100 & 32 & -100 \\ 100 & 100 & 92 & -78 & -100 \\ 100 & 100 & 0 & -100 & -100 \\ 100 & 78 & -92 & -100 & -100 \\ 100 & -32 & -100 & -100 & -100 \end{bmatrix}$$

Figure 4.6. Convolution kernels for Nevatia-Babu edge detector

This algorithm is implemented in the pyramid in a similar fashion to the convolution. There are some additional savings. First of all, all six convolutions take place after only one communication sequence. That is, each bottom level processor determines its 5×5 neighborhood only once. Then all six convolutions are calculated. This saves the communication steps for five of the convolutions. After the edge magnitudes are determined for the six directions, each processor then finds the edge magnitudes and directions for its 3×3 neighborhood. Each processor then determines if it is an edge pixel and sets an edge flag accordingly.

4.2 Image Segmentation

Winston [63] defines image segmentation as, "the problem of dividing an image into coherent regions corresponding to object faces." Another way of phrasing it is that image segmentation as the process of separating an image into spatially coherent regions. The spatial coherency of the final segmentation provides for fully connected regions without any unidentified holes. There is no actual metric for spatial coherency, as it is somewhat subjective.

Haralick and Shapiro present a detailed survey of many image segmentation techniques [64]. They classify these techniques into several categories:

1. measurement space guided spatial clustering
2. single linkage region growing schemes
3. hybrid linkage region growing scheme
4. centroid linkage region growing schemes
5. spatial clustering schemes
6. split-and-merge schemes

Ballard and Brown present a simpler categorization of segmentation techniques [1]. The two basic approaches to segmentation are edge based and region based. Both of these approaches are complementary to each other. Edge based segmentation methods rely upon finding boundaries between objects by searching for the edges. Region based methods group pixels into regions that represent individual objects (or parts of objects).

Each of these approaches makes use of many features from the image. Intensity values is a single pixel characteristic. Color or multispectral features are intensity values from individual pixels of multiple spectral bands. Texture features make use of intensity values of neighboring pixels and the statistical or structural properties that can be inferred from these. Hanson and Riseman incorporate these different methods into the VISIONS system [6, 12]. This system makes use of the processing cone (see Chapter 2) model to extract boundary and region features to form an image specific model containing regions, segments, and vertices. This model is then compared to the world model for matching. A similar approach is taken by Levine at around the same time [4]. Later work by Levine incorporates pyramid data structures into the extraction and analysis of image features [65].

4.2.1 Color Segmentation

Many segmentation schemes make use of measurements taken from different spectral bands. This may improve segmentation performance where regions might have similar intensities, but different hue or saturation. Typically, these could be the color spectral components for Red, Green, and Blue (RGB). Early work in multispectral segmentation was performed by Ohlander, Price, and Reddy [66]. They make use of histogram features from multiple data bands in a split and merge technique that is examined in detail in Section 4.2.2.

Multiple data bands contain more information than a simple monochrome band and thus should provide for better segmentation results. The different color bands provide for another dimension within the feature space. The data bands of the images need not include solely intensities from the visible spectra, but may also include infrared and ultraviolet bands. This discussion is restricted to only visible data bands.

Data from the three color bands (RGB) can be transformed into HSI coordinates (Hue, Saturation, and Intensity) by [66]

$$H = \arccos \frac{(R - G) + (R - B)}{2\sqrt{(R - G)(R - G) + (R - B)(G - B)}} \tag{4.7}$$

$$S = m\left(1 - 3\,\frac{\min(R, G, B)}{R + G + B}\right) \tag{4.8}$$

$$I = \frac{(R + G + B)}{3} \tag{4.9}$$

The parameter m is the maximum desired saturation value.

Data from the three color bands (RGB) can also be transformed into YIQ coordinates[1] (perceptual brightness, In-phase, and Quadrature) by [66]

---

[1] The I in the HSI system should not be confused with the I in the YIQ system.

$$Y = 0.509\ R + 1.000\ G + 0.194\ B \tag{4.10}$$

$$I = 1.000\ R - 0.460\ G - 0.540\ B + M \tag{4.11}$$

$$Q = 0.403\ R - 1.000\ G + 0.597\ B + M \tag{4.12}$$

The parameter M is the highest possible intensity value (255 for 8-bit images).

The image data can be transformed before segmentation to provide more data bands to enhance the results of the segmentation.

4.2.2 Ohlander-Price-Reddy Segmentation

The Ohlander-Price-Reddy Segmentation Algorithm uses features from histograms of image data bands to split regions. This technique is intended to be used with images containing multiple data bands. The data bands represents features from different spectral components.

Initially, the entire image is considered to be one uniform region. A histogram of each data band is taken and analyzed to determine a suitable threshold. The criteria for selecting a suitable threshold is listed later. Once a threshold is determined, the region is then split in accordance with this threshold. Merging is then performed to remove regions that are too small (typically isolated pixels). Each region is again processed individually: histogram of data bands, threshold determination, and merging. This iterative process continues until regions are no longer split; that is, a suitable threshold is not found for the region.

The threshold determination is performed by selecting the strongest peaks in the histogram. Pixels with values outside of this peak are split from the

region. The criteria for selecting the best peaks[2] is as follows (in order of precedence) [66]:

1. An intensity peak in 0-60 or 200-255 ranges (i.e. close to end of histogram),
2. Both minima < 10% highest value, max/min ratio > 4, another peak exists with max/min ratio > 2,
3. Both minima < 25% of peak value, max/min ratio > 4, another peak with max/min ratio > 2,
4. Max/min ratio > 2, another peak with max/min ratio > 2, if maxima are within 10% then both are acceptable (bimodal distribution),
5. Saturation only: minima in 0-200 (lowest 20%), max/min ratio > 2, specified minima must separate peak with max/min ratio > 1.2,
6. Minima < 10% of highest value and 10% of all points must be outside the peak, or
7. Minima < 70% of highest value and max/min ratio > 1.7.

Timings of the serial execution for the OPR algorithm indicate that the histogram computation generally takes more than 52% of the time needed [66]. In fact, as the size of the image grows, the number of data bands increases, or the complexity of the image increases, the computation time for the histograms increases faster than the times for the other operations. Thus, the percentage of time needed for histogram computation also increases. The threshold analysis does not get more complicated as the more complex image still contains the same number of bits per pixel (and thus, the same number of bins for the histogram). So, it is reasonable to assume that the histogram computation for the serial execution of the OPR algorithm is by far the dominant task in the entire

[2] The maximum and minimum values of the peaks are used as features to calculate threshold. The max/min ratio is the ratio of the peak's maximum to the smaller of the peak's minima.

process. With this in mind, a parallel method to substantially reduce this burden is presented in Section 4.2.4.

4.2.3 Phoenix Segmentation

The Phoenix Segmentation Algorithm is part of the SLICE segmentation scheme developed at SRI [67]. It is similar to the Ohlander-Price-Reddy Segmentation Algorithm in that it uses a recursive histogram-threshold-merge technique. The difference is in the method used to determine the threshold. The OPR algorithm determines the strongest intervals by concentrating on the peaks in the histogram. The Phoenix algorithm concentrates on the valleys. The method for selecting the threshold is [67]:

1. The histogram is first smoothed and then broken down into intervals. An interval is from valley, to peak, back to valley.
2. The histogram then goes though a series of interval merging tests (parameters can be fine tuned)
   a. An interval is retained if the ratio of peak height to the higher of the two shoulders, is greater than 1.6.
   b. An interval is retained if its peak area is greater than 30. Also, its area relative to the area of the histogram must be greater than 0.02.
   c. The second highest peak, and those peaks whose height is less than 20% of it are merged.
   d. Any interval whose right shoulder is more than 10 pixel counts greater than the lowest interior valley, is merged with its right neighbor.
   e. Merge intervals with low peak-to-shoulder ratios until only 2 valleys remain.

3. A score is now computed for each interval. The score is the maximum over all intervals of the function

$$1000 \left(\frac{\text{peak height} - \text{higher shoulder}}{\text{peak height}} \right)$$

4. The data band receiving the highest score is used to compute the threshold rule.

Pixels outside of the interval selected for the threshold rule are split away from the region to form another region. This method is claimed to give higher quality segmentation results because of the difference in histogram analysis [67]. Therefore, it is actually the Phoenix method that is used in this research (although it is still referred to as the OPR Segmentation Algorithm).

4.2.4 Histogram of Arbitrary Shaped Regions

Unlike the features used by many vision algorithms, the histogram is a global feature. It is dependent on information from the entire region (as opposed to a small, local neighborhood). The histogram is a bin count of the values (typically intensity values) at each pixel in the region of interest. The histogram of a partially segmented, multispectral scene $H(l, r, b)$ is a function of gray level, region label, and spectral data band.

A region based segmentation method, such as the Ohlander-Price-Reddy (OPR) algorithm (described in Section 4.2.2), requires that histograms be accumulated from a collection of regions that have arbitrary size and shape. The regions may have irregular shapes and may have holes (i.e. contain other regions). Therefore, there is no single processor in the pyramid architecture that can (with certitude) represent the "top" of any region without including parts of other regions within its domain. This means that the method cannot simply have the bottom level processors communicate their values upward and accumulate the histograms for regions separately.

The OPR Segmentation scheme is used as the vehicle in which the histogram algorithm for the pyramid is presented, although this algorithm is not limited to this segmentation scheme. Results from the OPR Segmentation work indicate that the histogram computation on color images takes approximately 52% of the computation time [66] (the more data bands, the larger the data bands, or the more complex the image, the higher this percentage will be). In contrast, the histogram analysis to determine thresholds takes less than 2% of the computation time; thus, the accumulation of histograms is by far the most computationally intensive part of the method. The number of steps required on a serial processor is $O(N^2)$ for each histogram (and there is one histogram for every spectral component of every region). This section presents a method using a pyramid architecture is given that reduces this to $O(N)$ for each histogram.

This method is based on a method used by Kushner, Wu, and Rosenfeld [68] to take histograms on the MPP. This method has been modified to handle arbitrary shaped regions and many spectral components. Here, the pyramid's hierarchical structure is useful to pipeline multiple histograms towards the top processor for analysis in choosing a threshold. The hierarchical structure of the pyramid reduces the number of steps needed to broadcast the threshold to $O(\log N)$ from the $O(N)$ needed for the MPP.

The method to perform the accumulation of the bin counts in the lowest level of the pyramid is as follows. Initially, each processor in the bottom level holds the gray level and region label of the pixel that it represents in the image. Each processor is responsible for accumulating the bin counts (for *all* regions in *all* spectral components) for the gray level value that corresponds to the row number of the processor (Figure 4.7). This means that it will keep track of Rp separate bin counts (where R is the number of regions and p is the number of spectral components). To simplify the discussion, a monochrome image is considered. Thus a packet is a pair of

numbers representing one pixel value and label. For p spectral components, it is p pixel values and a label.

Therefore, all the instances of class **LowLevelProcessor** hold a packet that contains the gray level and region label of the pixel that it represents. It then decides if the gray level of the packet is equal to its row number. If it is, it then increments the bin count for the region that corresponds to the packet's region label. If not, nothing happens.

All instances of class **LowLevelProcessor** then pass these packets (containing the gray level and region label) to its northern neighbor (and receive a packet from the south). The choice of direction for passing is arbitrary. All processors look to see if the gray level corresponds to its row number and if it does, it will increment the bin count for that region. Again, they continue to pass these packets on to their neighbors. After N steps, each processor has accumulated bin counts for all the packets south of it. This entire process is then repeated in the opposite direction. This process is repeated in the opposite direction because any given processor has not yet "seen" the packets from processors that are north of it. If the bottom level of the pyramid has a wrap-around feature (as on the MPP), then this is not necessary. After 2N steps, each processor holds the bin count (for all regions) for the gray level that corresponds to its row (for the entire column).

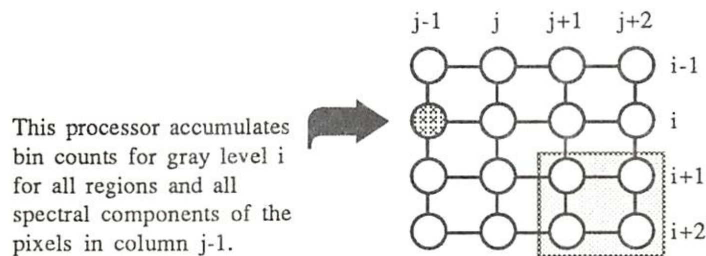

Figure 4.7. Accumulating bin counts in the bottom level processors

The other processors in the pyramid are idle during the accumulation of the bin counts. Of course, the distributed histograms residing throughout the entire bottom level of the architecture are very difficult to analyze. These must be sent upward towards the top level. Also, the information from the different columns is combined as the information is passed to the higher levels. When a higher level processor receives a set of subhistograms from its four children (consider the 4 processors in the shaded region of Figure 4.7), it combines data from the child at (i+1,j+1) with the child at (i+1,j+2) and the child at (i+2,j+1) with the child at (i+2,j+2) by adding the corresponding bin counts together. Then the two subhistograms are appended together to form a new, larger subhistogram. The new subhistogram has a length of two (gray levels i+1 and i+2) and covers two columns (j+1 and j+2). The method used to do this is similar for all processors (the bottom level does not receive anything from below and the top level does not pass anything on). This "combine and append" operation continues up the pyramid with the subhistograms growing at each interlevel communication step.

The histograms for each region and spectral component are sent in a pipeline manner up the architecture towards the top. That is, the histogram for all spectral components are sent up the pyramid (one following the other in pipeline fashion) for the first region. Then all spectral components are sent for the next region, and so on. The histograms are now accumulated at the top processor of the pyramid. The analysis is done in one processor (i.e. serially) so that the determination of a threshold (by comparing the histograms) happens efficiently. After the determination of a threshold, it is broadcasted downward.

The final merging process is performed by smoothing the split regions with a smoothing kernel (e.g. Gaussian mask). The convolution algorithm on a pyramid is documented in Section 4.1.1.

The only assumption made for this method of histogram accumulation is that the total number of gray levels possible in a pixel does not exceed the total number of rows in the image. If there are 8 bits/pixel (in each spectral component), then there are 256 possible gray levels. Therefore, the bottom level of the pyramid must be at least 256×256 (i.e. 9 levels). This is not really much of a restriction; as in Kushner, Wu, and Rosenfeld's method, larger histograms can be partitioned by requiring processors to keep track of multiple gray levels. That is, 9 bit images (512 gray levels) could be handled by each processors keeping track of gray levels i and 256+i (if it is in row i). In fact, this is the method used in the SCOOP pyramid.

4.3 High Level Symbolic Computation

Symbolic processing for vision often takes the form of semantic network processing or knowledge based processing [1]. Recent work has been done in the area of semantic network processing on a small two-dimensional mesh-connected network of processors [69]. This mesh is the same size as the third level of the SCOOP pyramid (i.e. 4×4), so adapting this technique to the pyramid is straightforward. Duda et al. describe a method whereby semantic networks can be represented in a rule-based inference system [70]. One of the main intents of this research is a goal based vision system, so the tasks of forward and backward chaining of rules is studied as part of the overall system.

4.3.1 Rule Chaining within a Production System

A Production System is composed of a set of rules (productions), a database (context) to keep track of the current state of knowledge, and an inference engine (interpreter). The symbolic information (i.e. rules and context) make up the form of knowledge representation for this level of abstraction of computer vision work.

A rule contains two parts: an antecedent and a consequence. The antecedent (or condition) implies the consequence (or the resulting action). The format of a rule can be expressed as

antecedent $\Rightarrow$ consequence.

The antecedent is a Boolean value or a logical combination of Boolean values. These values are determined from the given situation, the result of some action, or other Boolean variables yet to be determined. These values represent the current state of knowledge (or context). The consequence can be a Boolean variable to be used as an antecedent of another rule, a final goal, or it can initiate some action. The consequence is inferred from an antecedent that is asserted. The following is an example set of rules that could be used to describe the detection of bridges in an aerial scene.

Across(LINE, WATER) $\Rightarrow$ BRIDGES

EdgeDetect(DataBand5) $\Rightarrow$ LINE

RegionGrow(DataBand6) $\Rightarrow$ WATER

The interpretation of the rules can follow two directions: forward and backward. This process is called "chaining." The forward chaining of rules involves the logical inference of outcomes given a set of premises. Backward chaining of rules starts with a goal and determines what conditions need to be true in order to prove that goal. In this example, we can use the backward chaining process to form a goal tree of the antecedents that need to be proved in order to resolve the goal BRIDGES. Figure 4.8 is such a goal tree.

Notice that this goal tree is balanced. That is, the number of rules to be fired to infer the left half is equal to the number of rules to be fired to infer

the right half. The circles represent actions to be performed and the rounded rectangles represent antecedents or consequences.

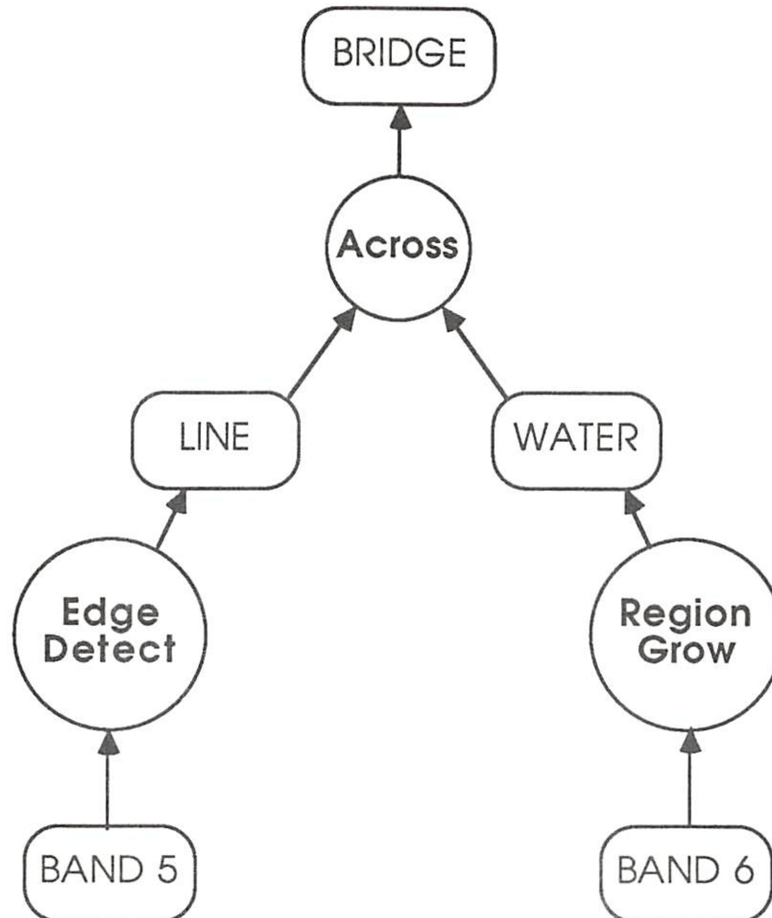

Figure 4.8. Goal Tree of Sample Rules

4.3.2 Parallelism During Chaining Process

The are two types of parallelism that can be utilized in a rule-based system. The inference engine (responsible for performing the forward and backward chaining of rules) can take advantage of **AND** parallelism and **OR** parallelism.

and parallelism When an antecedent is a combination of two (or more) partial antecedents and they are logically joined together via an AND statement, then each of these partial antecedents can be proved

concurrently. Another aspect is that if just one partial antecedent results in a *false* value, then the proof of the other antecedents is unnecessary as the logical AND of these will always result in a false value.

or parallelism When an antecedent is a combination of two (or more) partial antecedents and they are logically joined together via an OR statement, then each of these partial antecedents can be proved concurrently. Another aspect is that if just one partial antecedent results in a *true* value, then the proof of the other antecedents is unnecessary as the logical OR of these will always result in a true value.

Each of the antecedents in the rule base of a vision system is often an "action rule."

action rule A rule that is composed of antecedents that require some lower level processing.

These two types of parallelism can be exploited to reduce the effort needed to prove (or disprove) an antecedent. The rule base constructed for the SCOOP system is an action-oriented set of rules. Action-oriented means that many of the antecedents are not just simple logical values, but logical values that are determined as the result of some complex action that is performed at a lower level (i.e. an action rule). Specifically, these actions often make use of the entire pyramid (or critical parts of it) and therefore prohibit the other actions from taking place. These actions must be performed serially, even though there is a parallel implementation for it. In Figure 4.8, the circles represent actions: Edge Detect might represent a Nevatia-Babu edge algorithm (see Section 4.1.3), Region Grow might represent some clustering algorithm, and Across represents the overlaying of both results. Both the edge detection and the region growing use most of the processors in the architecture during their execution. The

overlaying cannot start until the results from the edge detection and region growing are both finished. Therefore, the actions must occur in sequential order and neither AND nor OR parallelism can be exploited to the fullest extent. This is not to say that AND and OR parallelism cannot be exploited at all. If the region growing is performed first and there are no water regions detected, then the edge detection need not be performed and the system reports that there are no bridges. That is, the unnecessary steps can be "short-circuited" (i.e. bypassed). The overlaying operation is basically a logical AND of the two processed images and the definition of AND parallelism states that if one partial antecedent is false, then the entire antecedent is false.

Conjecture: The action-oriented nature of a rule base in a vision system prohibits the concurrent proving of each antecedent of a rule on a single architecture (where that architecture is monopolized by each action). Therefore, the logical combination of two (or more) antecedents (each containing actions) must be performed sequentially.

This conjecture is related to the so called "frame problem" as described in Minsky's paper [71]

> *"The new, more successful symbolic theories use hypothesis formation and confirmation methods that seem, on the surface at least, more inherently serial. It is hard to solve any very complicated problem without giving essentially full attention, at different times, to different subproblems. Fortunately, however, beyond the brute idea of doing many things in parallel, one can imagine a more serial process that deals with large, complex, symbolic structures as units! This opens a new theoretical "niche" for performing a rapid selection of large substructures; in this niche our theory hopes to find the secret of speed, both in vision and in ordinary thinking."*

By forming a goal tree, a hypothesis is created that is to be proven by proving several subproblems. Making use of AND parallelism and OR parallelism, the execution of some of these subproblems can be short-circuited. However, the entire architecture (or at least its critical parts) does focus its attention on each subproblem during the firing of each action rule.

Therefore, the inference mechanism is not distributed amongst the processors because of the action-oriented nature of the rule base. Even though this might have resulted in quicker search times through the rule base, the savings would not have been worth the overhead needed to distribute and manage the rule base. As presented in Chapter 5, the overhead associated with parallel algorithms is often substantial. The concurrent execution of the actions within the rules prohibit this inference mechanism from taking advantage of its major source of parallelism. However, AND and OR parallelism is still used to trim rules (i.e. short-circuit) to prevent the unnecessary execution of some actions as explained earlier.

4.4 Sequence of tasks

In order to complete a wide range of tasks on the prototype, a complete scenario that include tasks that range from low level to high level algorithms must be performed. The scenario that is presented here includes algorithms that are as simple as taking a threshold and are all incorporated within an expert system shell that includes an inference engine for both forward and backward chaining as well as reasoning with uncertainty. This scenario starts with a high level abstract goal: given an aerial scene taken from a LANDSAT satellite, find all the bridges within the scene. This scenario uses this goal and a set of rules as a starting point. The inference engine will then form a goal tree and explicitly prepare for a sequence of lower level algorithms to execute.

Zucker describes the use of production systems with feedback [72]. Multiple matches can result in an ambiguity as to which rules to fire. Often, this can be resolved using feedback of rules. For instance, the sample rule-base of the previous section laid out a method of finding bridges in an aerial scene. Many edge detection algorithms produce many more edges than would be reasonable for bridges (even after overlaying these with regions identified as water). Further rules can be used to eliminate bridge candidates by requiring that these lines go *across* the water instead of *along* it. The medial-axis transform of the water can be used as a criteria of what it means to go across the water. Thus, if the first set of rules results in an unreasonable number of bridges, a further set of rules can be used to further qualify these candidates as actual bridges. If the production system includes confidence measures or incorporates fuzzy logic, then the feedback and execution of these additional rules can increase the confidence measures attached to the resulting bridges [73]. Therefore, one can infer that these edges (that went across the water and not along it) are bridges with a much higher confidence than those that are eliminated.

The next chapter presents the results of these simulations. Processed images are shown, together with timing results, memory usage, and analytical comparisons to some other architectures.

Chapter 5 Results

The results of the simulations on the SCOOP pyramid are discussed and presented in this chapter. First, the construction of the prototype is presented. SCOOP memory usage is then examined. Comparison of object table usage[1] and memory usage[2] for different simulations are shown. Then, the overhead due to algorithm setup is discussed. Following this is a presentation of timing results for each of the different algorithms implemented: convolution, Nevatia-Babu edge detection, Ohlander-Price-Reddy segmentation, and a complete scenario that includes a wide range of tasks. Timing results from the prototype are compared with expected results from the SCOOP pyramid and other architectures. Tables comparing the performance of several alternative architectures are shown. These table values for the various architectures were determined using analytical methods (or from named references). The table values for the pyramid were verified by SCOOP simulations.

5.1 The Prototype

This section presents the details of the construction and monitoring of the SCOOP pyramid. Subsection 5.1.1 discusses the construction of the prototype that is done by creating instances of the classes discussed in Sections 3.3 and 3.4. Subsection 5.1.2 presents the methods used to monitor the object table usage and memory usage within the SCOOP pyramid. This discussion is necessary to understand how some of the results of later sections of this chapter where determined. Subsection 5.1.3 discusses the limitations of SCOOP. Subsection 5.1.4 discusses the overhead required to setup the architecture for the task. This overhead cannot be ignored if the

---

[1]Object table usage is often known as "oops usage" in the Smalltalk community.
[2]Memory usage is often known as "core usage" in the Smalltalk community.

architecture is to perform a wide range of tasks that is driven by the knowledge-base (as are the scenarios of Section 5.7).

5.1.1 Prototype Construction

The construction of the SCOOP pyramid involves the creation of thousands of Smalltalk objects. There are objects to represent each of the processors, the ports between the processors, and objects to represent the architecture as a whole. There are also objects to represent the different values and parameters during the simulation: pixel values, results from operations, Boolean flags, and so on. The process of building the pyramid begins with the sending of the message `createPyramid:` to the driving simulation class (a subclass of **Pyramid** class).

For example, an instance of class **Convolution** is created for the simulation of a convolution of the image by a kernel. The instance of the simulation sends a message to the classes **Processor, LowLevelProcessor, TopLevelProcessor**, and **UnidirectionalPort** to create instances of each of these. These objects are then interconnected in a fashion that represents the topology of the architecture. The pyramid is created by connecting each level of processors together and then interconnecting the levels. An interconnection is created by assigning a common instance of class **UnidirectionalPort** to the processor's instance variables that represent the correct port connection to its neighbor. For example, the processor–(5, 5, 5)[3] is connected to its northern neighboring processor–(4, 5, 5) by assigning an instance of **UnidirectionalPort** to processor–(5, 5, 5) instance variable toNorth and also assigns the very same **UnidirectionalPort** to processor–(4, 5, 5) instance variable fromSouth. This is continued until the entire architecture is constructed and interconnected. To repeat, the prototype is built by creating an instance of each element of the architecture and then by interconnecting those elements to form the topology of that architecture.

---

[3]Processor–(x,y,l) refers to the processor at location x,y on level l.

A pyramid architecture can be built to various sizes. The number of processors within the architecture is

$$\text{Proc}_L = \sum_{i=1}^{L} 2^{2(i-1)} \tag{5.1}$$

for a pyramid with L levels and the recurrence equation

$$\text{Ports}_L = \text{Ports}_{L-1} + 10 \times 2^{2(L-1)} - 8 \times 2^{L-1} \tag{5.2}$$

computes the number of ports needed (with $\text{Ports}_2 = 16$). Table 5.1 shows the number of processors and ports needed for several different sizes.

# of levels	# of Processors	# of Ports
3	21	144
4	85	720
5	341	3152
6	1365	13136
7	5461	53564
8	21845	216400
9	87381	869712
10	349525	3487056

Table 5.1. Size of pyramid for different levels.

5.1.2 Monitoring of Prototype During Execution

The abstract representation of processors and ports by instances of their classes in the Smalltalk system uses a certain amount of memory depending on the total number of objects (referred to as oops) used and the memory (referred to as core) used for each of those objects. Smalltalk keeps track of the objects created by keeping an entry for each object in an object table. The amount of memory used for each object in the object table depends upon the complexity of that object. The properties of an object are determined by the protocol of the class and the private instance variables for that object. Therefore, a more complex object is expected to not only have more instance variables, but also each of those instance variables (each one of them objects themselves) are more complex and will take

more memory. The memory usage increases with the creation of new objects.

The creation of the architecture creates quite a demand on the object table and memory usage of the Smalltalk system. A monitoring process is used during the simulation to keep track of the memory and object usage during the construction of the architecture and the startup of the simulation. This monitoring process is started by sending the message `monitorSpaceUsage:` to the simulation. The Smalltalk code for `monitorSpaceUsage:` and its explanation follow.

```
monitorSpaceUsage: aFileStream
"This method will fork a process (at higher priority) to monitor stack
space and oops space usage. The higher priority is needed so that a
simulation will not prohibit this from running."
| aBlock |
aBlock ← [1 to: 10000 do:
    [:i |
    aFileStream nextPutAll: i printString; nextPut: Character tab;
    nextPutAll: Smalltalk coreLeft printString; nextPut: Character tab;
    nextPutAll: Smalltalk oopsLeft printString; nextPut: cr.
    (Delay forSeconds: 10) wait]].
MonitorProcess ← aBlock newProcess.
MonitorProcess priority: (Processor activePriority + 1).
MonitorProcess resume.
```

The MonitorProcess runs in the background during the simulation. A continuous loop prints the coreLeft (memory left for objects) and oopsLeft (space left in object table) to a file. This process is given a higher priority than all the processes running under the **Processors** since it must be able to perform these measurements without the simulation preventing this monitor from running.

Figure 5.1 represents object table usage as a function of real-clock time (not simulated time) for a 7 level pyramid simulation.

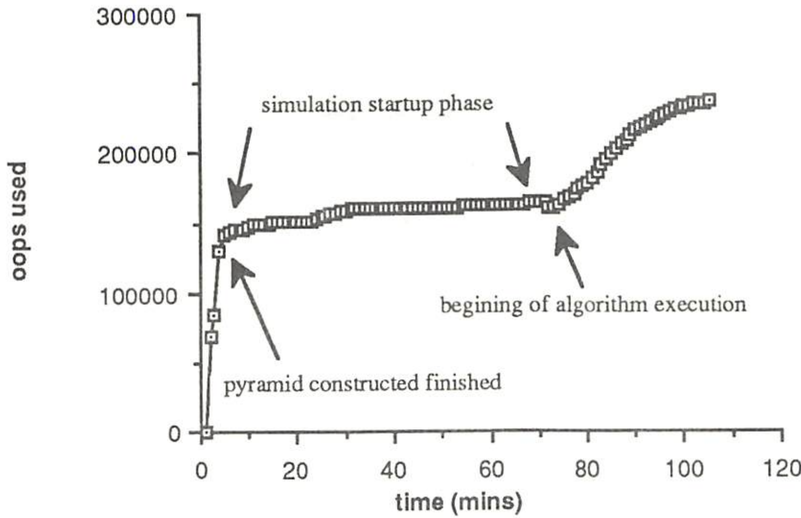

Figure 5.1. Object table usage in 7 level SCOOP pyramid

In addition to object table usage, a prototype requires memory for each of the objects themselves. Figure 5.2 represents memory usage as a function of time for a 7 level pyramid simulation.[4]

[4] It is interesting to note that a reduction in the number of levels to 6 substantially reduces the oops usage, but increases the core usage due to the larger arrays needed for histogram accumulation (see Figure 5.12).

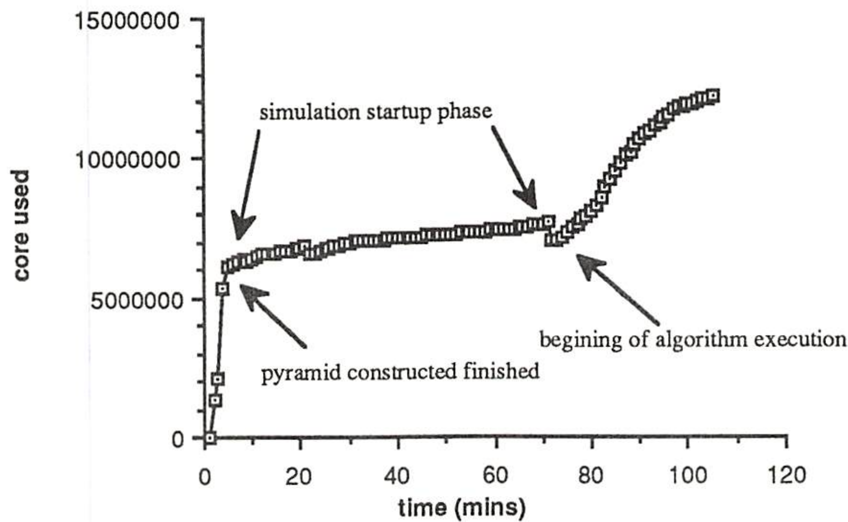

Figure 5.2. Memory usage for a 7 level SCOOP pyramid

Although both Figures 5.1 and 5.2 were measured during the start of an OPR simulation (to be discussed in detail in Section 5.4), the plots for the other simulations are identical up to the actual begining of the algorithm execution. In fact, the tasks (and kernels or tables) are not loaded until the actual begining of the algorithm execution. This is part of the setup overhead considered in Section 5.1.4. The plots only extend slightly past the construction of the SCOOP pyramid and do not present object table or memory usage during the algorithm execution.

The shape of Figures 5.1 and 5.2 are similar for other size pyramids—they would differ only in actual object table usage and memory usage due to the to number of instances of classes **Processors** and **UnidirectionalPort** (see Table 5.1). Both classes **TopLevelProcessor** and **LowLevelProcessor** are subclasses of **Processors**. Instantiating class **Processors** (or one of the subclasses) takes the same number of entries in the object table and uses the approximately the same amount of memory for each one. This is shown later in Figures 5.3 and 5.4.

The pyramid is constructed (i.e. all processors and ports in place) by 4.5 minutes (Figures 5.1 and 5.2). This is the most intensive use of objects during the preparation for the simulation. In fact, it is the most intensive use of objects throughout the entire simulation. The consumption of object table space increases rapidly and almost linearly during the pyramid construction stage. This happens during the execution of the method `createPyramid:` when sent to the class **Pyramid**. The 5,461 processors and 53,564 ports consume almost 150,000 objects during the construction. These results are expected because a large number of similar objects are created during the construction phase.

The simulation startup phase is the period of time that all processors (instances of a subclass of **EventMonitor**) report to the instance of **Pyramid** (a subclass of **Simulation**) and prepare to collect simulation statistics. This is a much longer period of time and does not consume objects quite as fast. What does happen is that memory for each of the memory intensive objects is consumed very rapidly. In particular, all objects that are instances of **Array** (or a subclass) are very memory intensive.

During the simulation startup phase, part of the executable Smalltalk image is constantly swapped in and out of memory. Also, garbage collection and object table compaction is invoked, thus slowing the simulation from this point on. The swapping process is used in the Unix system to manage an executable process that is larger than the actual amount of memory in the physical machine. These simulations were performed on a Sun 3/110 and much of the swapping was performed through a server over an Ethernet network, thus exacerbating this overhead. The garbage collection process tries to recover unreferenced objects in order to free space for further creation of new objects. Object table compaction marks all entries in the object table that no longer exist to be removed from the table. The active entries are then made contiguous within the table, thus freeing space for new objects at the end of the table.

However, most of the objects are not relinquished until the end of the simulation; thus the garbage collection and object table compaction (automatically invoked in Smalltalk) is often time consuming and not too productive.

Memory usage increases as the object usage increases. This is expected because the simulation is creating many instances of the same classes. A simulation with many more diverse classes of objects might have had the memory usage increase faster as certain types of objects were created. This did not occur in the pyramid as the three classes of processors are all closely related and all ports are the same. The next two figures confirm this. In both Figure 5.3 and 5.4, object table consumption and memory consumption are plotted against each other. The plots rise linearly during the construction of the architecture and increase linearly (although at a higher rate) during the simulation itself. The construction of the architecture entails the instantiation of the necessary processors and ports and the interconnection of these. The memory usage after the construction is due to the instantiation of local storage within each of the processors during each of the steps. Garbage collection is often invoked during this process (not shown in these figures) to retrieve the memory used and free up object table space from unreferenced objects.

The plots for both five and seven levels of the pyramid are similar in that they have two linear portions, with the architecture construction phase followed by the simulation startup phase. These plots show that the relation between object table usage and memory usage are independent of the size of the pyramid. This is expected because the objects that are created (all subclasses of **Processors** or **UnidirectionalPort**) are similar in complexity regardless of the pyramid size. The plot for the seven level pyramid covers a much longer period of time, and thus has many more data points than were collected using the method `monitorSpaceUsage:`.

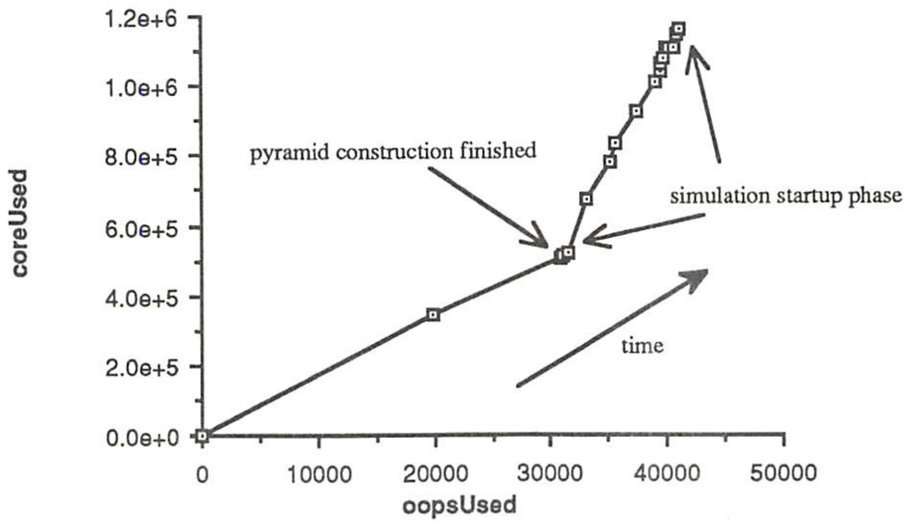

Figure 5.3. Memory usage for 5 level SCOOP pyramid.

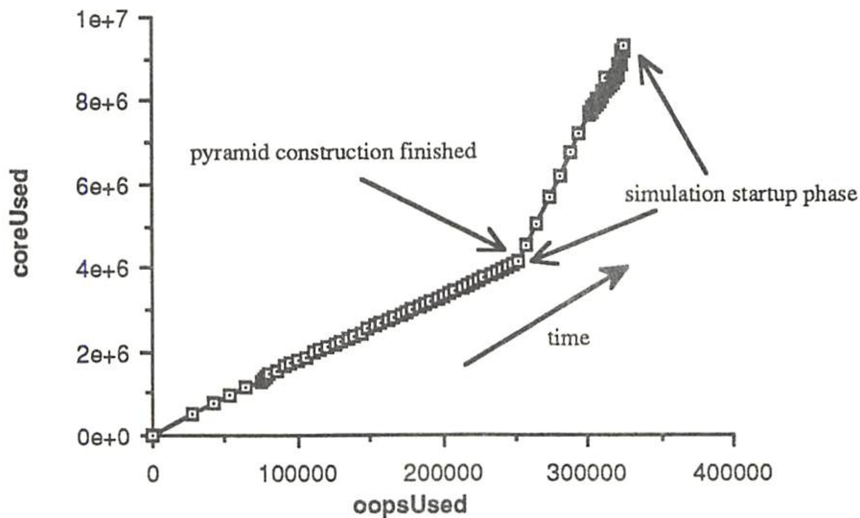

Figure 5.4. Memory usage for 7 level SCOOP pyramid.

5.1.3 Limitations of SCOOP

Additional memory is always needed for overhead of the interpreter, arrays, and other large data structures. The current version of the virtual machine interpreter (v. 1.1) for PS Smalltalk has a limit that the total

executable size of a Smalltalk image not exceed 16 MBytes [49].[5] This prevented the simulation of larger pyramid structures. Future versions should not have this limit and will allow for modeling of architectures of *arbitrary size*. The SCOOP pyramid does not need to be changed in order to model pyramids of larger sizes. The method `createPyramid:` requires the number of levels to be given as an argument and there is no limit imposed by the design of SCOOP. Sections 5.2 through 5.4 present actual processed images by the SCOOP pyramid at either 6 levels (Ohlander-Price-Reddy segmentation) or 7 levels (convolution and Nevatia-Babu edge algorithm). These pyramids correspond to bottom levels of 32×32 and 64×64, respectively. In order to reproduce these images for this thesis, the images had to be enlarged to a size of 256×256 by replicating pixels. This produces a "blocky effect" that is unavoidable without altering (e.g. smoothing or interpolating) the data. This is an artifact of the printing and reproduction process that is not inherent in the SCOOP pyramid.

The scenario results in Section 5.7 shows processed images of 512×512 resolution. It was possible to process images of this size because the processing had to be performed offline. The HUMBLE Expert System Shell (see Section 5.5) version was compatible with Apple's Smalltalk-80 (version 0.3) for the Macintosh. The images were processed offline, in accordance with the instructions of HUMBLE, on a Sun 3/110.

The processes of garbage collection and object table compaction reduce the simulation performance as the processes of the simulation are halted during these phases. Garbage collection and compaction are invoked automatically when needed. There is significant swapping to disk during these phases that also reduces the simulation performance. The swapping is reduced by adding more physical memory to the machine. The workstation used for these simulations relies upon a disk server over a

[5]This appears to be an outgrowth of the era of 16-bit machines at Xerox PARC, where Smalltalk was developed.

network. The swapping process (when needed) is much faster if the workstation has its own disk. Therefore, both physical memory and fast local data storage increase the simulation performance.

5.1.4 Algorithm Setup Considerations

Another point not previously mentioned (and often not thought about in predicting architecture performance) is that many vision algorithms require some sort of setup before actual execution. For example, a convolution requires that the values for the kernel must be sent to the processors. Thus, the implementation of the algorithm incurs setup overhead. Setup overhead is defined as the amount of processing time spent upon preparation for the particular algorithm. In image processing, this usually entails the distribution of kernel values or tables to be used during the algorithm.

An analogy to the setup overhead is that of a magic trick using a deck of cards. Many mechanical card tricks require that the deck of cards be setup before hand (not in the presence of the audience) in order to work as smoothly as it seems to when performed.

Equation 5.3 defines the setup overhead as a ratio of the computation to the overhead time. The ratio γ is defined as

$$\gamma = \frac{t_c}{t_o} \tag{5.3}$$

where t_c is the computation time and t_o is the overhead time. Time t_o includes communication overhead as well as setup overhead. Previous research by Sunwoo et al. has shown that some overhead is dependent upon the size of the data set [74].

5.2 Convolution

The actual simulation is performed by creating an instance of the class **ConvSimulation**. This is a subclass of class **Pyramid** and it inherits all the needed methods to construct the actual prototype. This instance of class **ConvSimulation** constructs the actual pyramid, performs the setup for the convolution, loads the image, and then convolves the image with the kernel. During the simulation, each processor reports to a log file each step it performs at each time during the simulation. The log file represents a report of the history of the simulation.

5.2.1 Algorithm Performance

The steps for the convolution are explained in Chapter 4. The convolution is performed totally within the bottom level of the pyramid. This makes the performance the same as two-dimensional mesh-connected network of processors. Table 5.2 presents a summary of the performance for the convolution algorithm on several architectures. Utilization is defined in Section 1.2. The parameters within the table assume an N×N image to be convolved with a k×k kernel. The systolic design is one consisting of k^3 systolic processors and is described in detail in [75]. The effects of filling and flushing pipeline are not considered for the systolic design.

	# of steps[6]	utilization
serial	N^2k^2	1.0
systolic	N^2k^{-1}	1.0
2-D MCN	k^2	1.0
pyramid	k^2	0.75

Table 5.2. Performance for convolution

While the performance of the pyramid is as strong as the 2-D MCN (and stronger than the others), the utilization of the architecture is lower than

[6]For this chapter, all values given are assumed to be order of magnitude, unless otherwise stated.

the 2-D MCN. The implementation of the convolution on the pyramid is identical to the 2-D MCN, thus the processors on levels other than the bottom level seem somewhat superfluous for this algorithm. However, the pyramid does allow for continuous processing of other algorithms while the convolutions are taking place. Consider a sequence of images (e.g. motion detection) where frames appear on the bottom level at a steady rate. A convolution takes place in $O(k^2)$ steps. The convolved image is sent up to the next level where the result is reduced in resolution by a factor of 2. The bottom level is now ready for the next frame in the image sequence. Meanwhile, levels 1 through L-1 (for an L level pyramid) now act as its own L-1 level pyramid to perform further processing on the convolved frame. If the rate of processing within this upper part of the architecture is fast enough to keep up with the rate of acquisition of the image frames in the bottom level, then bottom level will represent a 2-D MCN of processors that is fully integrated into an L-1 level pyramid for efficient data transfer. The advantage here is the full integration of the 2-D MCN to other parts of the architecture for further processing.

5.2.2 Setup Overhead

The broadcasting ability of the pyramid sets up each kernel in $O(k^2 \log N)$ time. The 2-D MCN does not have this broadcasting ability and requires $O(k^2 N)$ time to setup the kernel. This is not insignificant and is in fact *larger than the time to perform the convolution itself* (although still much faster than the serial machine with its setup time). The setup need not be performed for each convolution if repeated convolutions with the same kernel are to be performed. The scenario described in Subsection 5.2.1 about a sequence of images is such an example. The setup time (only performed once) is then insignificant compared with the processing time needed for the convolutions. However, this is a special case and a designer must consider the setup overhead in general. Table 5.3 shows the setup

times and total times for the convolution (the totals assume one setup and one convolution).

	# of steps for setup	total # of steps
serial	k^2	$(N^2+1)k^2$
systolic	k^3	$N^2k^{-1} + k^3$
2-D MCN	k^2N	$(k^2+1)N$
pyramid	$k^2 \log N$	$(k^2+1) \log N$

Table 5.3. Performance for convolution

The time to fill up the pipeline (systolic case) can be ignored since it is small compared with the execution time. The same is especially true in the serial case. However, the very efficient mesh and pyramid algorithms are now so efficient that their own setup times become the dominant factor.

5.2.3 Results

As an example of this procedure, Figure 5.5 is a gray level image before convolution. Figures 5.6 and 5.7 are the same image convolved with a smoothing kernel and an edge enhancement kernel, respectively. These images (64×64) are convolved in a 7 level pyramid (64×64 bottom level) in partitions. There are no edge effects because a row of **LowLevelProcessors** is added to the bottom level along each side of the image (66×66 bottom level). These extra rows of **LowLevelProcessors** contain the pixel values from the neighboring partitions. The time to process the image (ignoring partitioning) is completely independent upon the size of the image as well as the content of the data itself. The times are only a function of kernel size. These images were convolved with 3x3 kernels. The setup times for loading the convolution kernels are a function of the size of the kernel and the number of levels in the image.

Figure 5.5. Gray level image before convolution.

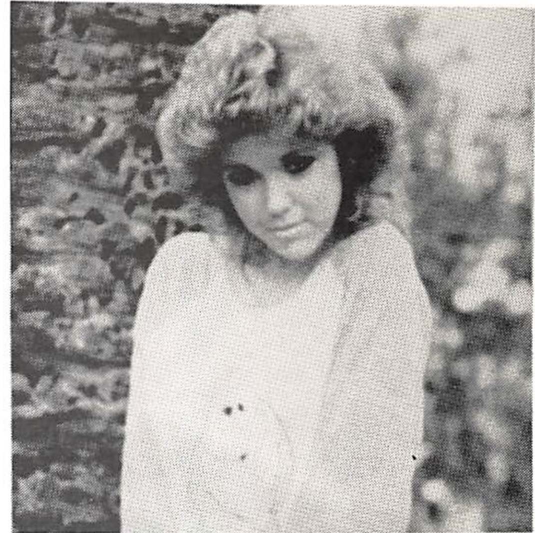

Figure 5.6. Image after convolution with smoothing kernel.

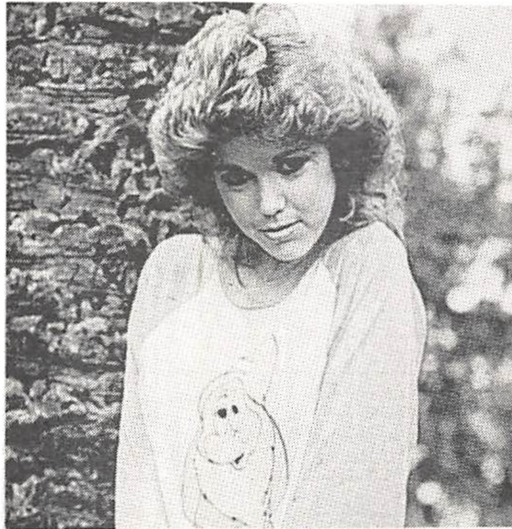

Figure 5.7. Image after convolution with edge enhancement kernel.

Figures 5.8 and 5.9 are the smoothing and edge enhancement kernels, respectively.

$$\begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.2 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}$$

Figure 5.8. Smoothing kernel.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 5.9. Edge enhancement kernel.

5.3 Nevatia-Babu Edge Algorithm

This algorithm is an interesting example of the combining several simple operations in order to realize a more complex process. This operation starts with six 5×5 convolutions and the results of the convolutions are compared to determine whether or not each pixel is to be labeled as an edge pixel. Section 4.1.3 contains a more detailed description of the algorithm.

As with the previous simulation, this simulation is started by creating an instance of class **NBSimulation**. It is a subclass of **Pyramid**. It inherits all the methods required to construct the prototype. A log file of this simulation is created with the reports of each processor during each step.

5.3.1 Algorithm Performance

Table 5.4 shows the results for the Nevatia-Babu edge algorithm. The times for the convolution of the 6 kernels are combined with the time for the comparison steps. In this table, N is the size of the image (i.e. N×N), k_1 is the size of the convolution kernels, and k_2 is the size of the comparison neighborhood.

	# of steps	utilization
serial	$N^2(k_1^2 + k_2^2)$	1.0
systolic	$N^2 (k_1^{-1} + k_2^{-1})$	1.0
2-D MCN	$k_1^2 + k_2^2$	1.0
pyramid	$k_1^2 + k_2^2$	0.75

Table 5.4. Performance for Nevatia-Babu Edge Detection

5.3.2 Setup Overhead

As with the convolution, the results in Table 5.4 indicate that both the 2-D MCN and the pyramid perform the best. Again, the pyramid does not achieve complete utilization of the architecture because the task uses only the processors on the bottom level. However, the pyramid does have a lower setup time for convolution kernels (see Table 5.3). This setup time is even more dominant in this algorithm as the convolution can be performed in an even more efficient manner than before (for both mesh and pyramid). After the neighborhood is determined for the first of the six convolutions, the second through sixth convolution can take place with no interprocessor communication at all. The entire neighborhood is already determined after the first convolution.

5.3.3 Results

Figure 5.11 is the image that results from processing Figure 5.10 using the Nevatia-Babu algorithm.

Figure 5.10. Gray level image before Nevatia-Babu algorithm

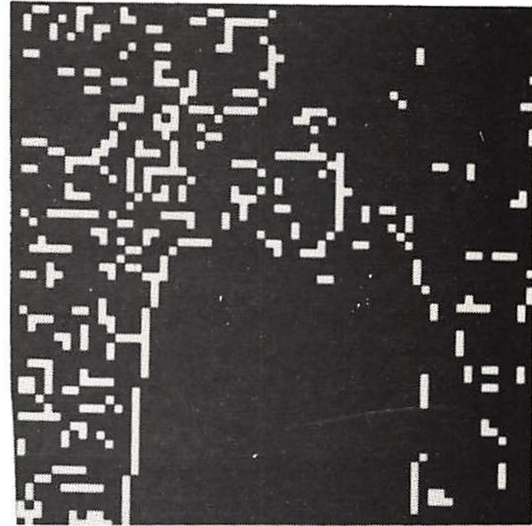

Figure 5.11. Processed with Nevatia-Babu algorithm

5.4 Segmentation

The segmentation simulation is quite complex compared to the previous simulations. In this case, all the processors within the architecture are expected to be more active during the algorithm. The previous simulations were performed on the bottom level (except during setup). An instance of class **OPRSimulation** is created for this simulation. As with the other simulations, it is also a subclass of **Pyramid** and inherits all the needed methods to construct the prototype. This simulation can be adjusted to handle segmentations with varying image size, number of data bands, and number of bits per pixel. Again, a log file is written as a history of each segmentation.

The results of the simulations of the segmentation algorithms are inherently more interesting in that they are typically dependent on the data. Very complex images will take longer to segment than simpler ones. This usually implies that more iterations are needed to converge on the final results.

Segmentation is also interesting in that it facilitates the transition between different levels of data abstraction. One starts out by segmenting an image (pixel-based data) and form regions that can be represented by symbolic information. This symbolic information often includes

1. shape description,
2. area of region,
3. label of region,
4. surface orientation,
5. illumination/shading information.

The important step in this transition is to define the regions correctly. With poor region definition, the symbolic descriptions derived will be misleading at best.

The segmentation process studied is the class of split-and-merge (or divide-and-conquer) techniques. In particular, a modified version of the Ohlander-Price-Reddy Segmentation process (known as the Phoenix Segmentation process) is simulated and results are presented.

5.4.1 Simultaneous Histogram Computation of Arbitrary Shaped Regions

A requirement of the split-and-merge segmentation algorithms is to accumulate histograms of the regions. The entire image starts as one uniform region. A histogram is accumulated and the degree to which the histogram is bimodal is determined. This is used to determine a threshold if one is acceptable. If the histogram is not bimodal enough, then a threshold is not chosen and the region is not split. If a threshold is chosen, then the region is split into two separate regions in accordance with this threshold.

Tanimoto (in [76]) describes a method of computing the histogram on a pyramid. The image size is N×N pixels and contains G gray levels. In

sequence, the values from 0 to G-1 are used as a threshold for the bottom level processors. Each step requires that the processor compare whether or not its pixel value is less than the threshold. If so, it contributes to the bin count for that gray level.[7] This process is continued in pipeline manner up the architecture towards the apex. This process for computing the histogram requires G iterations of the counting process. The counting process takes **O**(log N), so the entire histogram takes **O**(G log N) steps. This procedure is quite efficient, but is too general to be used to compute the histogram of arbitrary regions.

Levialdi (in [77]) describes a method whereby the histogram of the region is not actually computed, but bimodality is detected (or more precisely, estimated) at various levels of resolution. Each parent processor finds the partition (of the children) so as to minimize the variance about the mean. By comparing means, bimodality can estimated. At some level, a processor will decide that the region is bimodal and then the threshold will be broadcast downwards. This method is quite analogous to human visual system. However, there are practical programming problems in the operation of implementing this. First, it is not quite precisely an implementation of the algorithm in question. The Phoenix version of the OPR Segmentation process searches for dominant intervals in the histogram, not for valleys. This produces improved segmentations results [67]. Secondly, this assumes an image model of a single object and background. Complex images of many objects (particularly overlapping and occluding) will present programming problems for coordinating the independent bimodal searches.

Section 4.2.4 presents a detailed description of the method developed to compute the histograms of arbitrary shaped regions simultaneously. The method is to accumulate the bin counts simultaneously for all regions and

[7]Notice that this is an accumulative histogram and that the actual count for that bin can be obtained by subtracting values previous bin counts.

all spectral components. The histograms are now distributed among the bottom level processors. The second phase is to send the subhistograms upwards in a pipeline, combined-and-append fashion until they reach the top processor (see Section 4.2.4). This takes O(N + RpG) steps.

5.4.2 Ohlander-Price-Reddy Segmentation (Phoenix Version)

The following results were obtained in the SCOOP pyramid by instantiating the class **OPRSimulation**. The simulation makes use of the entire pyramid and utilization plots are presented.

For the following times, N is the size of the bottom level, p is the number of data bands, R is the number of regions, G is the number of gray levels, and k is the size of the neighborhood used in merging. The following times are for 1 iteration.

The communication for accumulating the bin counts takes O(N) steps. The time to send the histograms up takes O(RpG) steps. The histogram analysis requires O(RpG) steps (serially). Broadcasting the threshold rule takes O(log N) steps. Table 5.5 lists the times needed (order of magnitude) for the different subtasks of a single iteration of the OPR segmentation algorithm using serial, 2-D mesh-connected network, and pyramid architectures.

	histogram[8]	threshold	analysis	merge
serial	RpN^2	N^2	RpG	k^2N^2
2-D MCN	RpN	N	RpG	k^2
pyramid	N+RpG	log N	RpG	k^2

Table 5.5. Performance for OPR iteration

There is quite a speed up for certain tasks in the pyramid (more noticeable as N becomes large). One takes advantage of using a powerful processor as the top processor in the SCOOP pyramid. When the histogram is loaded

[8]The 2-D MCN histogram computation leaves a distributed histogram along the first column of processors. An additional N steps are required to have them "funnel-off" to a single processor for analysis.

into this processor for analysis, it can analyze the histogram at a very fast speed so that the resulting threshold will be obtained very quickly. The histogram is still the dominant task in the OPR algorithm and the SCOOP pyramid manages to speed it up quite a bit. Note that all the results are for just 1 OPR iteration. The number of iterations is dependent on the data, and the effects of the speed-up is compounded if there are many iterations (i.e. as the complexity of the data increases).

Figure 5.12 is a plot of number of bins each processor is responsible for versus different sizes of pyramids and different dynamic ranges. The taller the pyramid, the larger the bottom level. Thus, the lower level processors for the taller pyramids have less bin counts to keep track of.

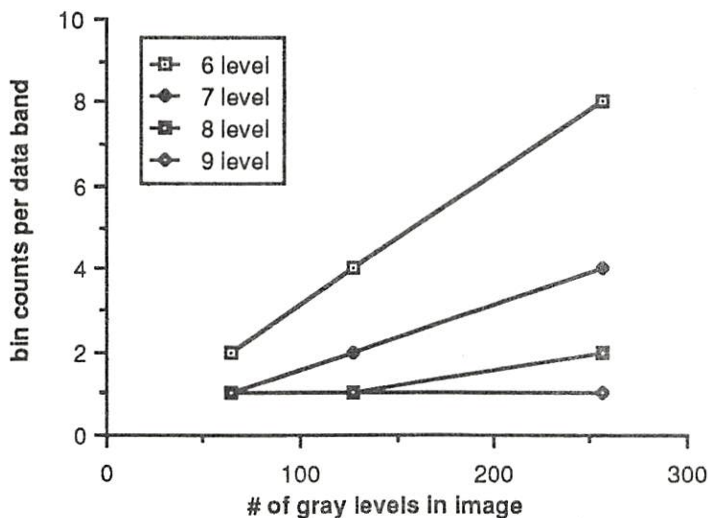

Figure 5.12. Number of bins each processor is responsible for

Figure 5.13 is a plot of the utilization of the processors in the pyramid as a function of time. The accumulation of histogram bin counts is the dominant process. The bottom level (~75% of the architecture) works alone on this process. The histograms (for each region and spectral band) are sent up the pyramid to the top processor. The histogram analysis is done in the top processor while the other processors remain idle (this

accounts for the near zero utilization during this process). The broadcasting of the threshold is then performed in logN steps. Finally, the bottom level performs the merging.

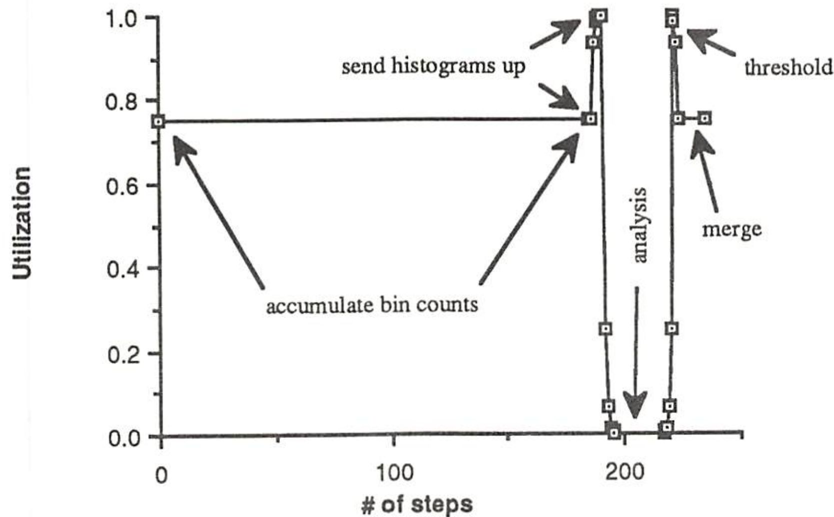

Figure 5.13. Utilization of the pyramid during each iteration of the OPR algorithm

Figure 5.14 is a plot of time needed (as a percentage of each iteration) for each of the processes for 3 data bands at 6 bits/pixel in each band.

Figure 5.15 is a plot of time needed (as a percentage of each iteration) for each of the processes for 1 data band at 8 bits/pixel in each band.

Notice that the task of accumulating the bin counts is the dominant task. The histogram accumulation is always the dominant task in the Ohlander-Price-Reddy Segmentation [66]. The pyramid has a good ability to broadcast and speeds up the threshold task by quite a bit. The local communication of the bottom level mesh provides for rapid merging process. The analysis is done serially. The accumulating of bin counts pushes the values of the image around the bottom level in the same way a systolic array operates upon the data. This accounts for the smaller amount of speed-up in this task.

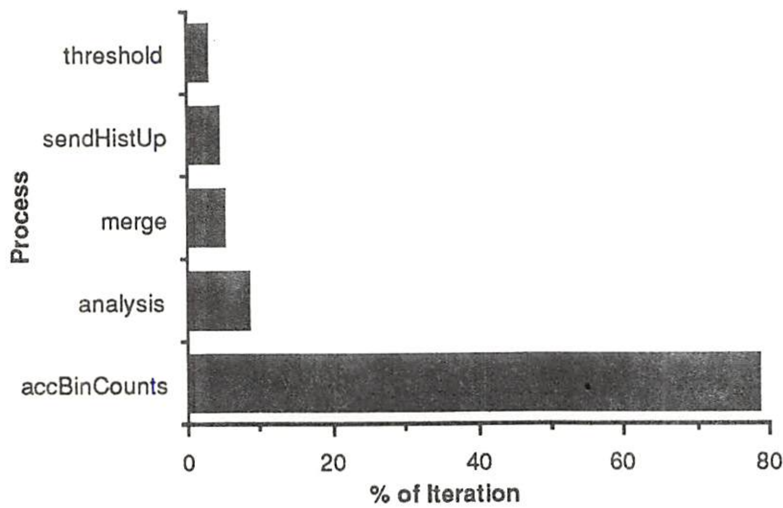

Figure 5.14. Processing time for each step of OPR iteration—3 data bands, 6 bits/pixel

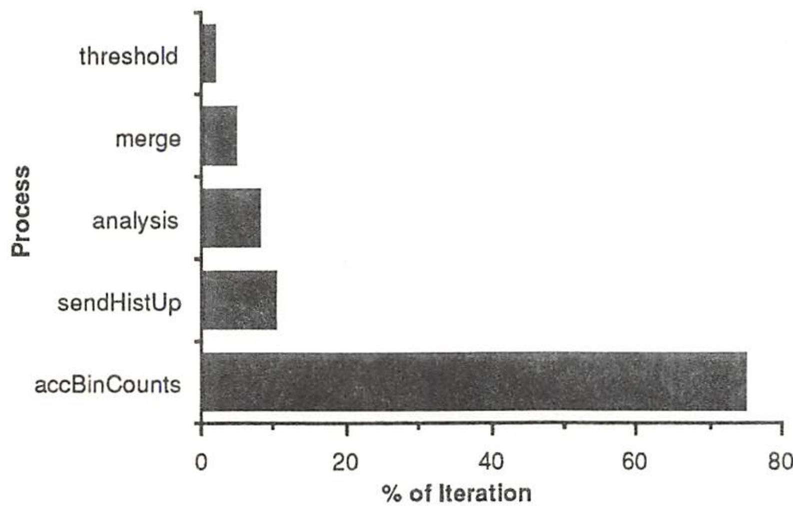

Figure 5.15. Processing time for each step of OPR iteration—1 data band, 8 bits/pixel

SCOOP simulations were done for data sets of varying size, bits per pixel, and number of data bands. Pictorial results of a typical segmentation are now presented.

5.4.3 Results

Figures 5.16, 5.17, and 5.18 are the original RGB data bands processed. In this simulation, the pyramid could only complete the algorithm for a 6-level pyramid because of the restricted memory available, as explained in Subsection 5.1.3. These data bands are 32×32 and have 6 bits of dynamic range. Even with this degradation of the input data, the segmentation results shown in Figures 5.19, 5.20, and 5.21 are quite good. These results overlay an outline of the region boundaries on top of the data bands.

Figure 5.16. Red data band.

Figure 5.17. Green data band.

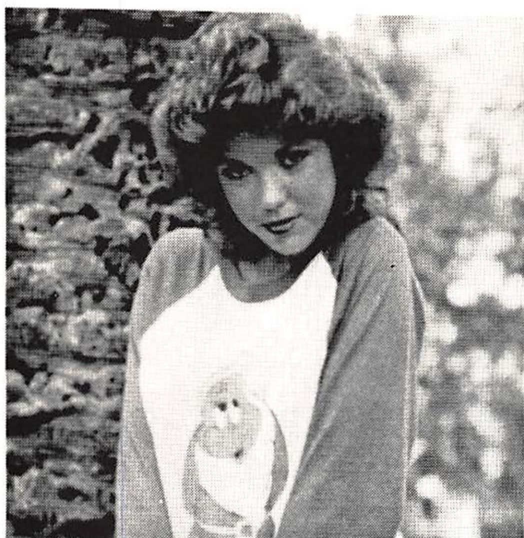

Figure 5.18. Blue data band.

Figure 5.19. Region boundaries overlaid with red data band.

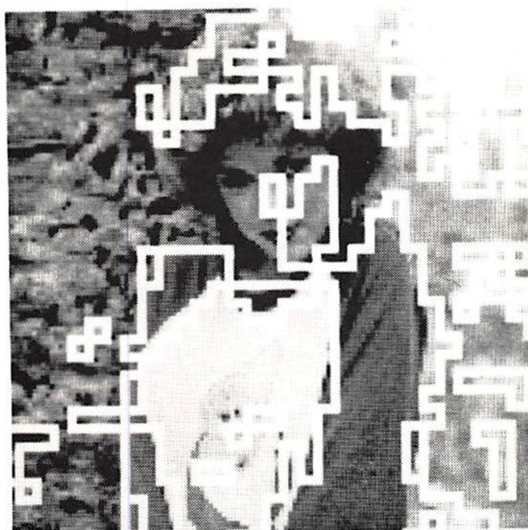

Figure 5.20. Region boundaries overlaid with green data band.

Figure 5.21. Region boundaries overlaid with blue data band.

5.4.4 Space Tradeoffs

A curious phenomena occurs when segmenting images of different sizes. An image of smaller resolution should take less time and space in the

SCOOP simulation. It is true that it took less time by the expected values predicted in Table 5.5. However, the object table usage and memory table usage did not decrease proportionally (see Figures 5.3 and 5.4). A five level pyramid is only 1/16th the size of a seven level pyramid.[9] However, the amount of object table usage and memory usage for a seven level pyramid is less than 16 times that of a five level pyramid. This is due to the relative increase in bin count responsibility with increasing bits per pixel, as shown in Figure 5.12, for pyramids with a smaller number of levels. When taking this into account, the object table usage and memory usage turn out as expected.

5.5 The Expert System Shell

Ishikawa and Tokoro present the idea of Distributed Knowledge Object Modeling in their Orient84/K language [78]. The Orient84/K language is an object-oriented language (based on Smalltalk) for knowledge representation. It was not possible to use Orient84/K in this research as it does not allow for execution of standard Smalltalk programs underneath it and therefore does not integrate well with the Smalltalk environment. This led to the use of an alternative expert system shell to drive the simulations.

In order to keep all the processing within the Smalltalk system, an object-oriented inference mechanism was needed. The HUMBLE[10] expert system shell [79] is used to construct the knowledge-base (rules) for the vision system. HUMBLE is written in Smalltalk and can send messages to Smalltalk objects outside of the HUMBLE shell.

---

[9]A pyramid asymptotically increases in size by a factor of four as each level is added to the bottom. Thus, adding two levels will increase the relative size by a factor of 16.
[10]HUMBLE is a trademark of Xerox Corp and is a product of Xerox Special Information Systems.

The format of HUMBLE rules are not Smalltalk, but are compiled into Smalltalk code within the HUMBLE shell. The basic structure of the rules are

```
RuleName
      "comments - should explain purpose of rule"
      statements.
```

Any statements surrounded by a pair of double angle brackets ('<<' and '>>') are interpreted directly as Smalltalk code. Thus the HUMBLE system will allow for firing of rules to trigger other Smalltalk tasks (such as previously simulated algorithms). The rules surrounded by curly brackets and preceded by an at sign ('{@...}') are rules to be chained in the forward direction. Therefore, both the forward and the backward inference mechanism were used.

A HUMBLE knowledge-base is constructed by building Entities [79]. An Entity is a formal representation of something about which rules can be written. An Entity is much like an instance of a class of entities known as an Entity Type. Entities contain parameters that are manipulated by the rules of the system.

In this research, the only version available for the HUMBLE shell was for the Macintosh computer. The instructions from the inference engine are used to control off-line processing since it was not possible to complete the scenario (Section 5.6) with the limited memory of the Macintosh. The Smalltalk messages resulting from the formation of the goal tree (by HUMBLE) directed the off-line processing (on the Sun 3/110 workstation). The timing results presented later takes the known parallel execution times of the off-line processing into account.

5.6 A Scenario: Finding Bridges in Aerial Scenes

In this scenario, the goal is to find bridges in a multispectral LANDSAT image. A set of rules is constructed to represent the knowledge used in

processing LANDSAT data. As much *a priori* knowledge is used to take advantage of experience from previous processing tasks. The following are the rules of the knowledge base that fired in this scenario[11]:

```
findBridgeCandidates
        "Rule will check for lines going across water - these are probably bridges."
        ifNoneOf:   Region
        have: [labelOfObject = 'water']
        then: [@{findWater}].
        "If no water exists now, then the image has no water."
        ifNoneOf:   Region
        have: [labelOfObject = 'water']
        then: [<<Transcript cr ; show: 'No water regions in
              image.'.>>].
        "Now that we have water, we can now find and overlay edges."
        {@findEdges}.
        {@overlayEdgesWithRegion}.

findEdges
        "Determine the instrument type and perform correct"
        "sequence of tasks."
        if: (dataSet = 'MSS')
        then: [{@findEdgesMSS}]
        else: [
              if: (dataSet = 'TM')
              then: [{@findEdgesTM}]
              else: [<<Simulation report: 'unknown instrument'].
              ].

findEdgesMSS
        "Find edges in a multispectral scanner (MSS) data set."
        "Signal to architecture to perform edge detection"
        <<Simulation active findEdges: #band5>>.
        "Now threshold edges using 5 most significant bits."
        <<Simulation active threshold: 32>>.
        "Get rid of isolated pixels."
        <<Simulation active morph: #isolRm>>.

findEdgesTM
        "Find edges in a Thematic Mapper (TM) data set."
        "Signal to architecture to perform detection"
        <<Simulation active findEdges: #band5>>.
        "Now threshold edges using 5 most significant bits."
        <<Simulation active threshold: 32>>.
        "Get rid of isolated pixels."
        <<Simulation active morph: #isolRm>>.
```

[11]The rules are expressed in a variant of Backus-Normal form (BNF). This is the syntax used in the Humble system. Many other rules exist, such as finding region size, but do not fire in this example.

```
findWater
        "This rule will find water regions in image."
        if: (dataSet = 'MSS')
        then: [{@findWaterMSS}]
        else: [
                if: (dataSet = 'TM')
                then: [{@findWaterTM}]
                else: [<<Simulation report: 'unknown instrument'].
                ].

findWaterMSS
        "Find water regions in MSS data set."
        <<Simulation active threshold: #band6 at: 50>>.
        <<Simulation active inversion #band6>>.
        labelOfObject = 'water'.

findWaterTM
        "Find water regions in TM data set."
        <<Simulation active threshold: #band4 at: 32>>.
        <<Simulation active inversion: #band4>>.
        labelOfObject = 'water'.
        "Now skeletonize to find medial axis of water."
        <<Simulation active morph: #skeleton times: 50>>.
        labelOfObject = 'skeleton'.

overlayEdgesWithRegions
        "Now overlay water and edges."
        if: (dataSet = 'MSS')
        then: [
                <<Simulation active overlay: #band5 with: #band6>>.
                <<Simulation active morph: #erode times: 2>>.
                <<Simulation active morph: #isolRm>>.
                labelOfObject = 'bridge' withCertainty: 0.7
                ].
        "TM bridges have higher confidence since they are"
        "overlaid with medial axis of water."
        if: (dataSet = 'TM')
        then: [
        <<Simulation active overlay: #band4 with: #band5>>.
        labelOfObject = 'bridge' withCertainty: 0.8
                ].
```

In plain English, these rules essentially follow these heuristics:

1. Bridge candidates are lines that run across water.
2. One finds water by passing the proper spectral band through a threshold.
3. One finds lines using an edge detector.

4. Lines found using rule 3 overlaid with water found using rule 2 are bridge candidates. Bridge candidates are bridges with a moderate confidence level.
5. One can increase the confidence that the bridge candidates are bridges by eliminating those bridges that are not approximately normal to the axis of the water. That is, bridges go *across* water and not along water. Therefore, the bridge candidates should intersect the medial axis of the water to improve the confidence that the candidates are bridges.

All of these rules will be explained as they fire with the output images shown in Subsections 5.7.1 and 5.7.3.

5.7 Scenario Results

As explained in Section 5.1.3, the limitations forced the actual low level image processing tasks to be executed offline. Within the Smalltalk environment, the HUMBLE shell is used to create a goal tree that determined the processing steps necessary at each point in time. Prior simulations determined the time necessary for these offline operations and these times are incorporated into the timing results.

Two scenarios are described in the following subsections. The first one is from a data set of the Washington, D.C. area using the Multispectral Scanner instrument. The second one is from a data set of the Baltimore area using the Thematic Mapper instrument. These are both LANDSAT instruments. These scenarios are meant to show the flexibility of the system—that it adapts to different sets of data and perform the corresponding operations as described by the rule set.

5.7.1 MSS Scenario

Figures 5.23 and 5.24 are bands 5 and 6 of the MSS scene, respectively. Figure 5.25 is the result of performing a convolution of band 5 with the Laplacian mask as shown in Figure 5.22 after a threshold of 32 (5 most significant bits). In this image, the outline of the water becomes noticeable and the some of the lines stretching across the water are the bridges.

$$\begin{bmatrix} -1.0 & -1.0 & -1.0 \\ -1.0 & 8.0 & -1.0 \\ -1.0 & -1.0 & -1.0 \end{bmatrix}$$

Figure 5.22. 3×3 Laplacian Kernel

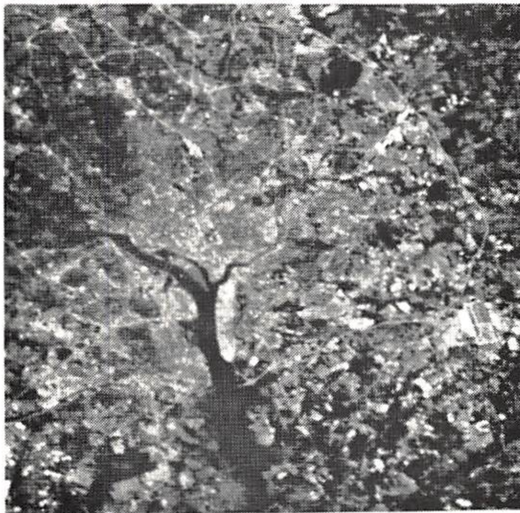

Figure 5.23. MSS data band #5.

Figure 5.24. MSS data band #6.

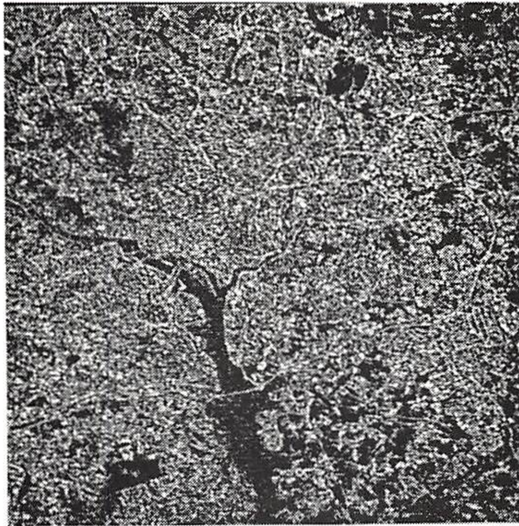

Figure 5.25. Band 5, edge detection, threshold 32.

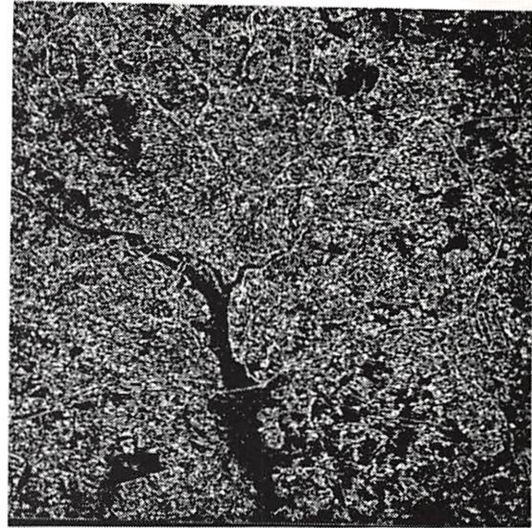

Figure 5.26. Band 5, edge detection, threshold 32, isolated pixel removal.

Figure 5.26 is the result of one pass of the isolated pixel removal operation performed upon Figure 5.25. This removes much of the edge effects on the water without removing any bridge pixels.

Next band 6 is used to determine the water regions. Experience has shown that using threshold of 50 is a good value to find water in band 6 of the MSS. Figure 5.27 is band 6 after a threshold of 50.

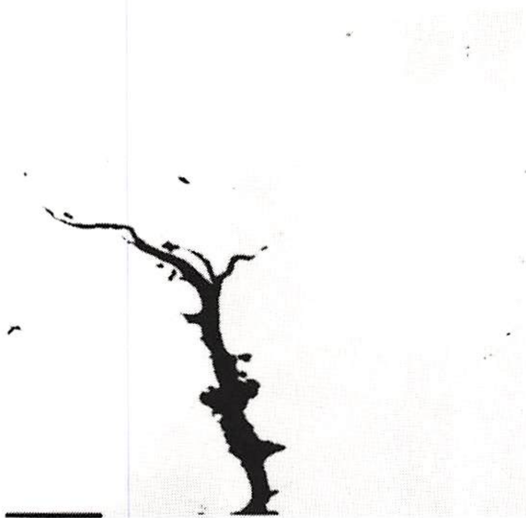

Figure 5.27. Band 6, threshold 50.

Figure 5.28. Invert pixels of previous.

The results have to be overlaid. This operation is basically an AND operation. So, each pixel of the image must be inverted so that the water has a value of one and the non-water regions are zero. This results in Figure 5.28.

The two images are overlaid and Figure 5.29 is the result (Figure 5.30 is an inverted and magnified version of Figure 5.29). Notice that the only errors are those small isolated edges over the water. Erosion will be used in the next steps to get rid of these without severely affecting the strong lines.

Figure 5.29. Overlay of processed bands 5 & 6.

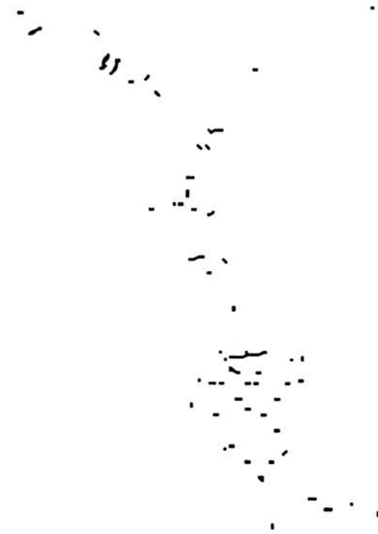

Figure 5.30. Inverted and magnified image of Fig. 5.29

Figure 5.31. Isolated pixels removed from Fig. 5.29

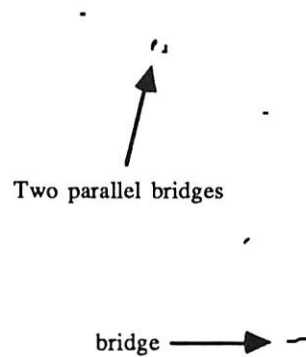

Figure 5.32. Inverted and magnified image of Fig. 5.31

Isolated pixels are removed after erosion and six bridge candidates (4 real and 2 false) are found in Figure 5.31. The resulting image is inverted for clarity, annotated and shown in Figure 5.32.

This processing found three bridges and resulted in also three errors (false alarms). Further rules can be used to filter out such errors (the TM scenario in Section 5.7.3 uses such a rule). A rule stating that a bridge must go *across* the water and not *along* it increases the confidence measure associated with each qualifying bridge. This is used in the next scenario.

5.7.2 MSS Timing Data

The processing times are reflected in the following plots. Figure 5.33 shows the utilization of the architecture as a function of time. This plot includes overhead associated with loading convolution kernels and morphological operation tables during the execution of the actual process. This overhead cannot be ignored unless a sufficient amount of local memory is allocated in each processor in the bottom level.

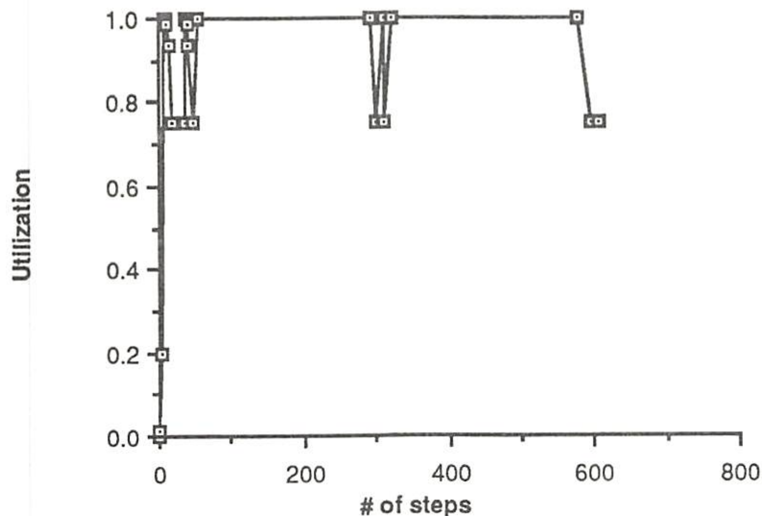

Figure 5.33. MSS utilization (includes overhead).

If the overhead is excluded from the timing analysis, it must be assumed that each processor on the bottom level has enough local memory to keep copies of the most often used kernels and morphological operation tables. The morphological tables require 512 bytes each (see Section 4.1.2). The

convolution kernels require k^2 bytes for each k×k kernel. Figure 5.34 is a plot of utilization for the scenario without the need to load kernels and tables.

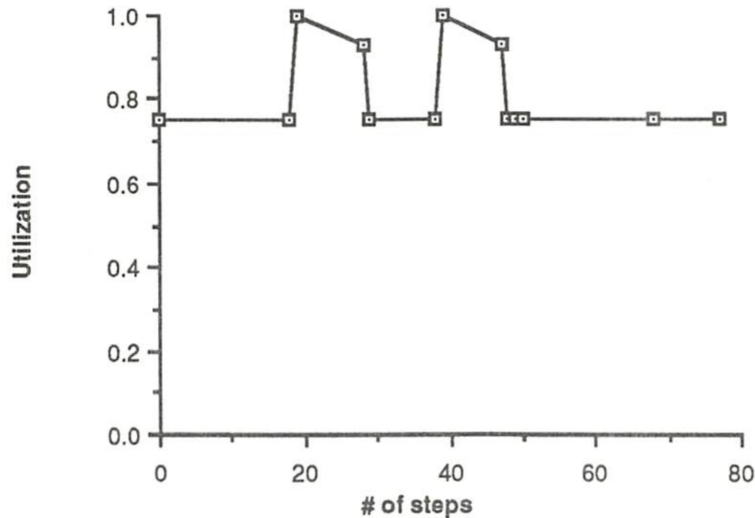

Figure 5.34. MSS utilization (without overhead).

The processing time overhead for the convolution, threshold, and morphological operations are shown in Figure 5.52 in Section 5.7.4. Figures 5.33 and 5.34 show that the SCOOP pyramid allows the designer to evaluate design parameters (in this case, local memory) before building any actual hardware. This tradeoff will be shown again in the Thematic Mapper scenario results (Section 5.7.4).

5.7.3 TM Scenario

The following scenario is has the same goal as the previous MSS scenario, but uses the data from the TM (Thematic Mapper) LANDSAT instrument. Figures 5.35 and 5.36 are bands 4 and 5 of the TM data set.

Figure 5.35. TM band #4.

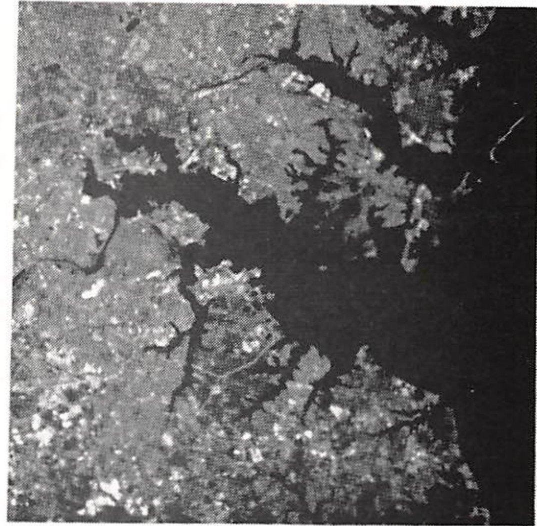

Figure 5.36. TM band #5.

Figure 5.37 is band 4 after a threshold of 32. This extracts the water regions. Figure 5.38 is the erosion of figure 5.37 after inversion (white pixels are eroded). This operation erodes some of the shoreline between the water and land that shows up better in TM data than in MSS data.

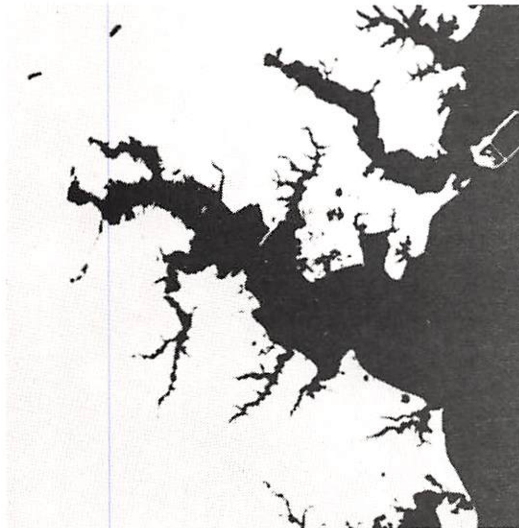

Figure 5.37. Band 4, threshold 32.

Figure 5.38. Band 4, threshold 32, inverse, erosion.

The medial axis of the water is now obtained by performing 50 iterations of skeletonization (a morphological operation). Figure 5.39 is the image after just 10 iterations. Figure 5.40 is after 50 iterations. Apparently, the wide bay requires more iterations, but bridges are typically not located over large bodies of water.

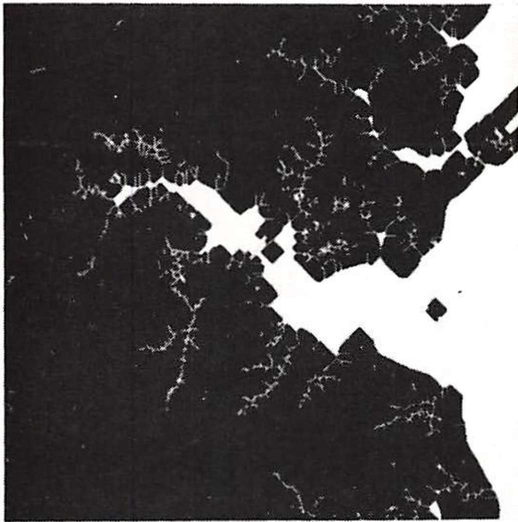

Figure 5.39. After 10 skeletonizations

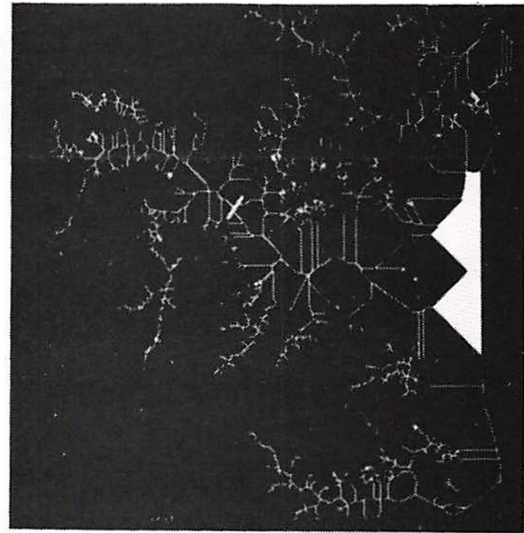

Figure 5.40. After 50 skeletonizations

Figure 5.41 is the result of the convolution of band 5 with the Laplacian edge kernel (see Figure 5.22). The edge detected image is then then passed through a threshold of 32 to get rid of weak edges. The result is in Figure 5.42.

Figure 5.41. Band 5, edge detection

Figure 5.42. Band 5, edge detection, threshold 32

The isolated pixels are removed from Figure 5.42 (another morphological operation) and the result is shown in Figure 5.43. Figures 5.43 and 5.38 are then overlaid in Figure 5.44.

Figure 5.43. After isolated pixel removal

Figure 5.44. Overlay of Fig. 5.38 and Fig. 5.43

Figure 5.45. Inverted and magnified image of Fig. 5.44

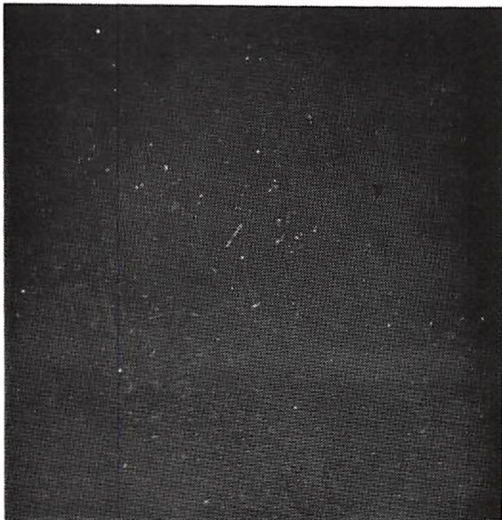

Figure 5.46. Overlay using medial-axis transform

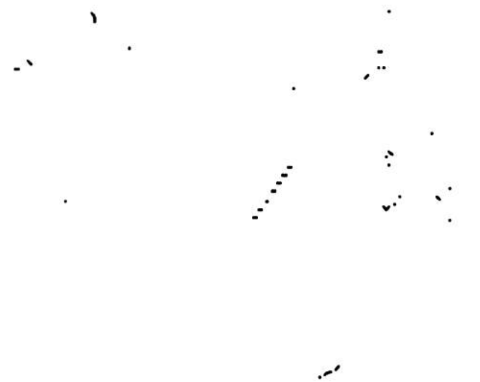

Figure 5.47. Inverted and magnified image of Fig. 5.46

There is the strong presence of the one bridge, but there are many false alarms. These are now bridge candidates. The false ones are now eliminated using the results from the medial axis transform. Figure 5.46 is

the result of overlaying Figures 5.43 and 5.40. Many of the bridge candidates that did not go *across* the water (edge effects from shoreline) are eliminated in this step. Figure 5.48 is the result from overlaying the same two images, but only after erosion and skeletonization is performed upon Figure 5.43 (to remove some of the smallest bridge candidates). This produced the best results. The bridge is still visible and there are only 4 false alarms. If the length of these bridges are used as a discriminator, then only the bridge is found with no false alarms.

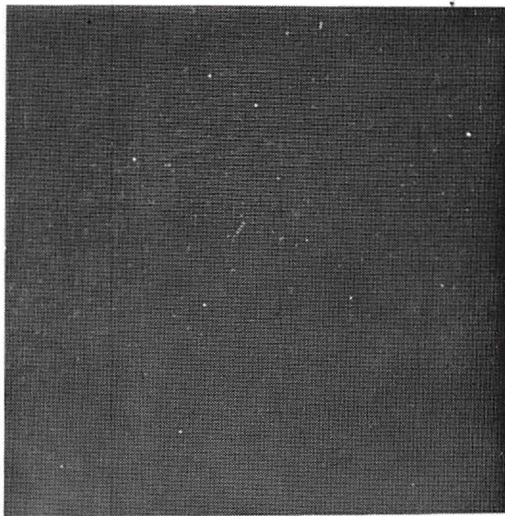

Figure 5.48. Overlay after erosion and skeletonization

Figure 5.49. Inverted and magnified image of Fig. 5.48

5.7.4 TM Timing Data

Figure 5.50 is a plot of the utilization for the Thematic Mapper case including overhead (as explained in Section 5.7.2). There is full utilization during the overhead stages (broadcasting kernels and morphological tables). The utilization drops to approximately 75% during the actual operations that only the bottom level participates in (e.g. convolution and morphological operations).

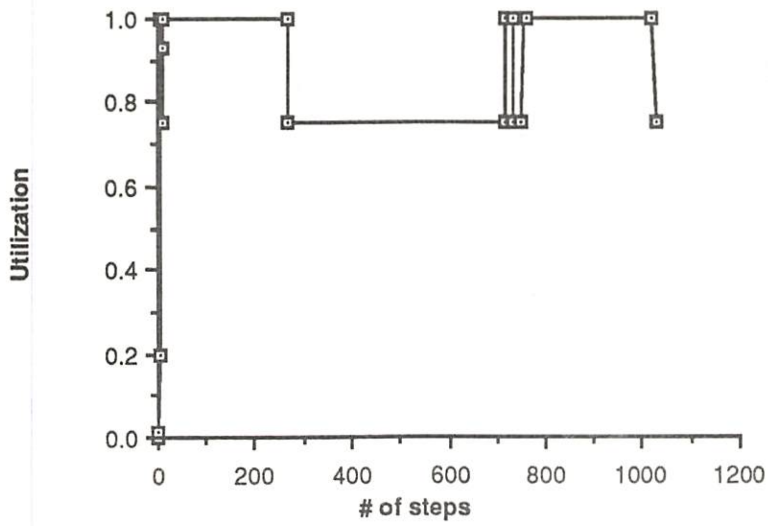

Figure 5.50. TM utilization with overhead.

Figure 5.51 is a plot of utilization for the TM case using the assumption of sufficient local memory to avoid dynamic loading of tables and kernels. The time need is much less and the full utilization only happens during broadcasting of instructions or thresholds.

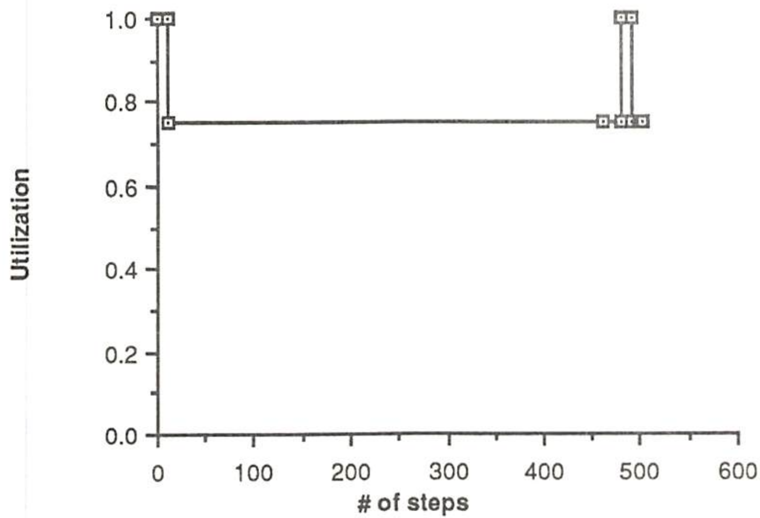

Figure 5.51. TM utilization without overhead.

As with the MSS scenario (Subsection 5.7.2), the SCOOP pyramid allows the architecture designer to evaluate the performance with different amounts of local memory. Thus, the software prototype enables the fine-tuning of the architecture before hardware is built.

5.7.5 Processing Overhead

The overhead for the major processing tasks is shown in Figure 5.52. The setup time (or overhead) is often more than the time to actually perform the task. This is especially true in the case of morphological operations (see Subsection 4.1.2). However, repeated iterations of morphological operations do not require retransmission of this table. In the case of processing the image from the Thematic Mapper, a skeletonization was performed that required 50 iterations to produce the medial-axis of the water.

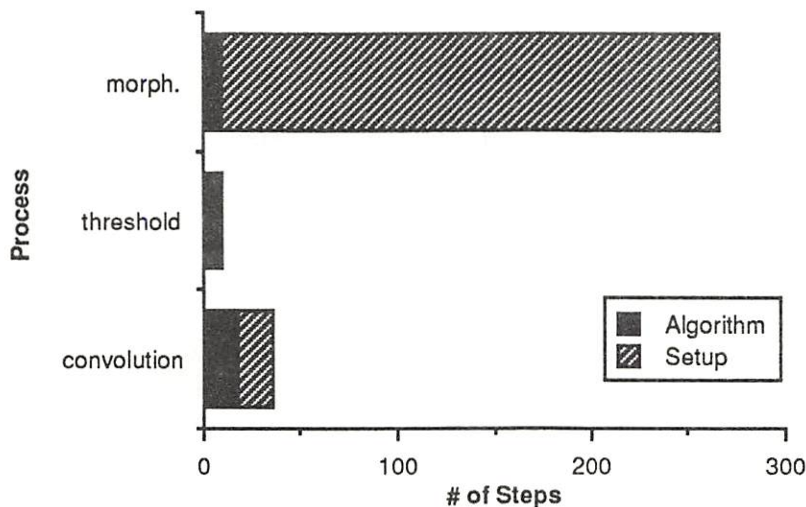

Figure 5.52. Processing times shown with associated overhead during scenario

5.8 Comparison to Other Architectures

In summary, the presented algorithms were performed at least as fast as the other architectures and often much faster (refer back to Tables 5.2, 5.3, 5.4, and 5.5). The 2-D MCN often performed as well as the pyramid for local neighborhood operations because the pyramid was, in fact, operating as a mesh. However, the pyramid has additional flexibility for use in hierarchical processing as shown in the Phoenix Segmentation and LANDSAT scenarios.

The following chapter will discuss the implications of these results and will also offer thoughts on the use of this methodology of constructing prototypes (used in the SCOOP pyramid) to aid researchers in analyzing other architectures.

Chapter 6 Discussions

This chapter reviews the results of the Chapter 5. Important points that result from this research are discussed. Many of these points are qualitative in nature, but offer substantial assistance to those interested in constructing prototypes of other architectures for analysis.

6.1 Methodology of Constructing Prototypes

Perhaps the most interesting characteristic of this research is the method used to build a working prototype of the architecture. The method used to model the architecture is object oriented. The Smalltalk-80 development system is used to create a working prototype of the architecture. The prototype is then used as a testbed for simulations of a wide range of computer vision algorithms.

Obviously, building the actual hardware with the correct number of processors interconnected properly would be even more favorable than just constructing a prototype. However, this is often not a practical solution. In particular, it would take many years to design, fabricate, assemble, wire-wrap, and debug all the necessary hardware. The cost of constructing this can be enormous and the debugging effort is often lengthy and frustrating.

Of course, someone interested in the architecture could then build a scaled-down hardware prototype of the architecture. For instance, Weems et al. propose a hierarchical architecture for image understanding: the Image Understanding Architecture (IUA) [34]. This architecture consists of three interconnected levels of 2-D mesh connected arrays. The top level consists of 8×8 LISP 32-bit processors (MIMD), the second level consists of 64×64 array of 16-bit processors (synchronous MIMD), and the bottom level consists of a 512×512 array of 1-bit (serial) ALUs. This hierarchy is a

"truncated pyramid" which the high, medium, and low levels of vision tasks are mapped onto the top, middle, and bottom levels of the architecture, respectively. The researchers at the University of Massachusetts are presently building a scale prototype of this architecture at a 64:1 ratio [80]. That is, the top level is represented by a single processor, the middle level will include 64 processors, and the bottom level will contain 4096 cells (in a 64×64 array for the image level). This is scheduled for completion in early 1988.

Building the hardware prototype is an important step in the construction of the actual architecture. However, a software prototype is quite advantageous in this case. First of all, the prototype could be built in a manner of weeks instead of many laborious months. In fact, the current pyramid prototype constructed in this research can be modified to connect in the manner that this IUA prescribes by only modifying the method `createPyramid:` and by modifying the classes **Processors**, **TopLevelProcessor**, and **LowLevelProcessor** to conform to the characteristics of their processors. Of course, the software prototype runs slowly in comparison to actual hardware. What can take seconds on a hardware prototype can often take many minutes (even several hours) on a software prototype. However, the software prototype can help their analysis and especially programming of the architecture quite quickly. First of all, the IUA and the pyramid studied in this research differ in only several ways: the particular topology (although it is still quite similar to a pyramid) and the actual specified processors that are proposed. The IUA, like the SCOOP pyramid, is planned to be an architecture that processes vision algorithms on many levels of abstraction. Which is the better one to pursue? Is there another variation of either of these architectures that will perform better than both? These questions can best be answered by building software prototypes of each and running them against each other. Committing to building even a scaled-down prototype of the actual

hardware will leave little room for change if the researchers decide that a variant might be in order.

The necessity to actually build a prototype of an architecture cannot be overstated. Theoretical analysis of algorithms is quite useful, but it is unable to predict the unforeseen pitfalls of actually programming the architecture. A case in point is the overhead used to setup many algorithms (see Chapter 5).

6.1.1 Aid in Hardware Implementation

The advantages of this methodology, from a hardware designers point of view, are that SCOOP

1. aids in the programming of architecture, both global control and individual processors, and
2. allows designers to examine potential modifications to the architecture without performing actual hardware modifications.

The first point will be further explored in Section 6.1.4. The second point allows, rather encourages, the designer to "fine tune" the design by making incremental changes to the prototype. Certain tradeoffs can be studied to determine the proper design parameters. Some tradeoffs include

1. more local memory per processor versus cost,
2. central controller versus distributed control,
3. local handshaking versus global clock,
4. wider bandwidth communication channels versus cost,
5. changes in instruction set for some/all processors, and
6. changes in interconnection network properties.

To summarize, the enormous flexibility in the software prototype allows a designer to change the design parameters of the proposed architecture to

improve the final design. The software prototype also acts as a testbed for simulations to predict the performance of the architecture.

6.1.2 Modelling Other Topologies

As discussed previously at the beginning of this chapter, the SCOOP pyramid can be reconfigured to model architectures of other topologies.

To begin with, the standard two-dimensional mesh-connected array has already been modeled because each level of the pyramid is a mesh. Many of the low level image processing algorithms in the pyramid primarily make use of only the bottom level. The convolution and morphological operations performed in the SCOOP pyramid are essentially the same models for the SCOOP mesh. The only difference is the methods for setting up the architecture for the tasks (i.e. broadcasting the kernels or morphological tables).

The method `createPyramid:` is the method that is responsible for creating processors and ports and connecting them together to reflect the pyramid topology. By substituting the method `createHypercube:` for `createPyramid:`, a SCOOP hypercube can now be modeled. The argument to the message `createHypercube:` is the number of dimensions for the cube. Also, some minor changes to the class **Processors** would be necessary to reflect the fact that a processor in a pyramid communicates with 9 other processors and that a processor in a cube communicates with N processors (for an N-dimensional Hypercube).

The class structure of the processors for the architecture must change also if other topologies are modeled. The SCOOP pyramid is not homogeneous. Different levels have different processors (although within each level all the processors are the same). In the hypercube, all the processors should be the same. Figures 6.1 and 6.2 is a partial protocol diagram that describes the mapping the class hierarchy of processors for the pyramid into the

hypercube. Only some of the instance variables and methods related to interprocessor communication within the structure is shown. The class structures are represented by rounded rectangles. Within each rounded rectangle are one or more of the following: class name (bold type), more rounded rectangles (subclasses), comments (italic type), a partial list of instance variables (follows comments), a partial list of methods (enclosed in a box), and possibly a partial list of class variables.

The class hierarchy starts with class **SimulationObjects**, which is the superclass of **EventMonitor**. The subclasses **PyramidEventMonitor** and **CubeEventMonitor** differ only in name as they perform similar functions; they are both abstract classes that alter the reporting functions of class **EventMonitor**. The SCOOP pyramid then has a class **Processors** with subclasses that represent processors with special properties: **LowLevelProcessor** and **TopLevelProcessor**. The SCOOP hypercube has homogeneous processors, so no subclasses of **Processors** are necessary.

The instance variables that point to the 9 neighbors for each processor in the pyramid are replaced by the instance variable toNeighbors and fromNeighbors for the cube. Both of these are instances of class **Array** whose length is the number of neighbors for each processor (i.e. the dimension of the hypercube). The index into either of these arrays points to the only neighbor in that hyperplane. All methods related to communicating with the neighbors must be altered to take this new structure into account.

Class structure for the pyramid processors

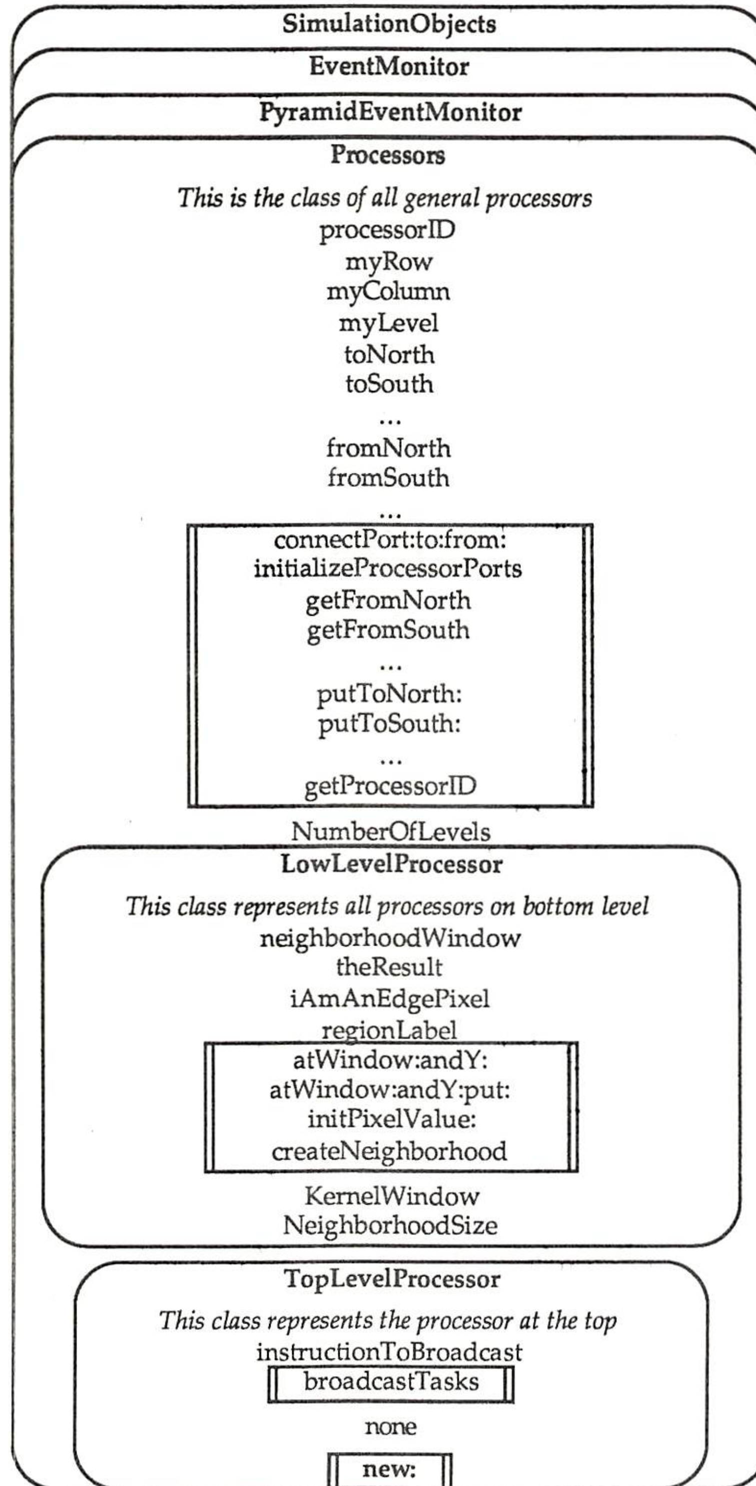

Figure 6.1. Processor class hierarchy for SCOOP pyramid

Class structure for the cube processors

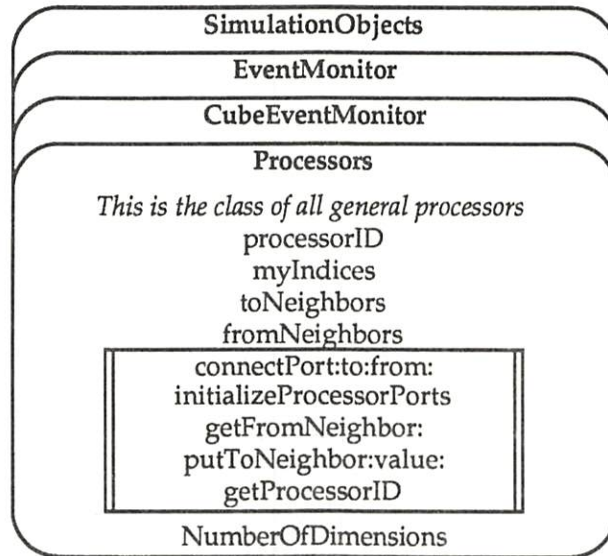

Figure 6.2. Processor class hierarchy for SCOOP hypercube.

Of course, the mapping of the algorithms are different to take into account the vastly differing topologies. However, the transition to the SCOOP hypercube is straightforward and thus provides for a software prototype to use as a testbed for hypercube algorithms.

Some researchers (e.g. Uhr in [8]) propose that pyramid architectures be augmented with extra internal or external connections. Extra internal connections (or higher bandwidth connections at upper levels) can be easily modeled by the SCOOP pyramid. The augmented pyramid needs only change the method `createPyramid:` and add the appropriate instance variables within the class **Processors** to handle the extra connections. Higher bandwidth connections are handled by altering the communication properties of class **UnidirectionalPort**. External connections between the pyramid and other processors (e.g. a host) can be modeled either by modelling both architectures in the SCOOP environment, or by externally integrating the SCOOP pyramid to the host via the external operating system links that PS provides [49].

6.1.3 Optical Architectures

An interesting ability of software prototypes is that they are flexible enough to model architectures implemented using different technologies as well as other topologies. An interesting area of research of computer architectures uses optical hardware as a substitute for electronic hardware. In fact, optical hardware should not be thought of only as a substitute for electronic hardware. The design of optical architectures should not be limited by the constraints of electronic hardware. Rather, the design should take advantage of the characteristics of optics [27].

Giles and Jenkins [27] discuss complexity issues of generalized architectures and the relation to optical architectures. Optical architectures offer significant advantages over electrical architectures for several reasons

1. Three dimensional nature of light - signals with intersecting light paths do not interfere with each other during free space propagation. Photons are non-interactive, while electrons exhibit mutual interaction. This may make optics more immune from interference and crosstalk.
2. Lower space complexity. Space complexity is a measure of either VLSI surface area or hologram surface area of interconnections implemented [27].
3. Higher fan-out is achievable using optics.
4. Lower likelihood of global clock skew.
5. Freedom from capacitative loading providing for higher bandwidths than with electronics [81].

The methodology of constructing prototypes in this research involves creating abstract descriptions of the components of the physical machine. The SCOOP pyramid described in the previous chapters is completely electrical. Abstract classes are used to describe the architectures. Similarly,

abstract classes can be constructed that describe optical phenomena. These new classes must be capable of describing the limitations of optical hardware while allowing for some characteristics that extend beyond that of electrical hardware.

The first step might be to consider a collection of electrical processors that are connected via an optical interconnection network in the topology of a pyramid. The interconnection network may be an optical crossbar network constructed using spatial light modulators (SLM) [81]. This will involve the description of a new model of communication between the processors.

Using a global clock for synchronous communications in a massive electronic architecture with electronic interconnections could lead to serious differential timing delays. Optical interconnection networks with reduced clock skew as compared to electronics can be envisioned, as shown in Figure 6.3.

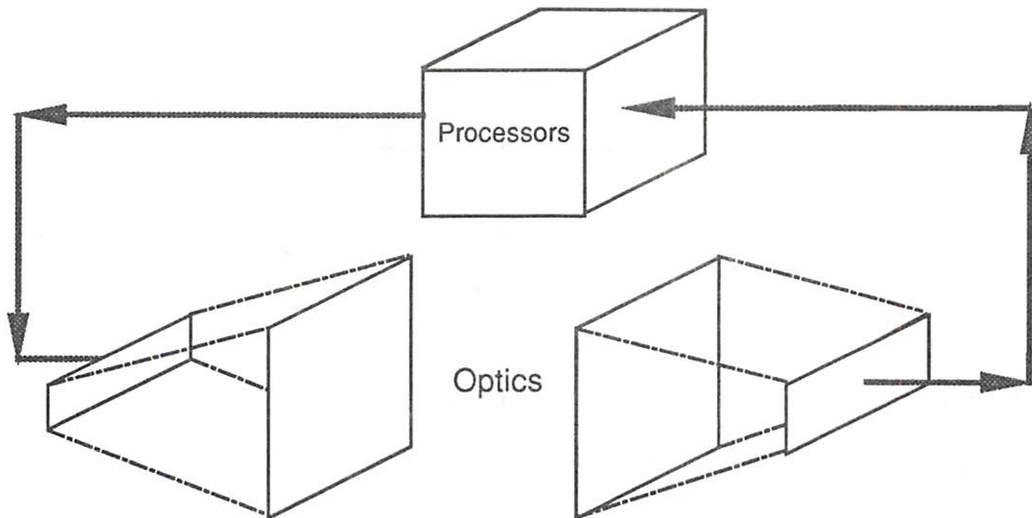

Figure 6.3. Optical Interconnection Network

The processor ports in Figure 6.3 are now thought of as part of the interconnection network. All processors "hand-off" the values to the ports asynchronously, but within the proper time of interprocessor communication. This allows for skew between processors, but the

interconnection network is now responsible for the synchronization of the architecture between communication steps. To model this, a new type of **UnidirectionalPort** needs to be constructed. This will be called **OpticalPort**. This port needs to have a class variable GlobalClock visible to all instances of **OpticalPort**. There is no handshaking during the communication, each port just waits for the clock signal. The methods, instance variables, and class variables will differ between **OpticalPort** and **UnidirectionalPort**. However, the protocols are still identical due to the encapsulation property of Smalltalk-80. Because the protocols are identical, the class **OpticalPort** can be easily substituted for **UnidirectionalPort** in the prototype.

At present, optical architectures have limited processing element (PE) complexity due to technology limitations. Thus, while arbitrarily complex optical interconnections could be built, they may exceed the capabilities of the processing elements.

The number of simultaneous inputs that can be accepted is limited by the complexity of the PEs. This is usually independent of N (size of architecture), for large N. For a pyramid, the degree of connectivity is fixed (at 9 for the type of pyramid studied in this research). This is independent of N, so simultaneous input reception is a reasonable assumption. Some other topologies (e.g. hypercube and fully connected) have degrees of connectivity that grow as a function of N. Thus, simultaneous input reception is *not* a reasonable assumption.

Similarly, multiple copies of outputs can be sent simultaneously. However, multiple messages sent simultaneously also require increased PE complexity.

All of these require changes to the model of communication and computation within the prototype. Thus, to model all of these situations, it is useful to describe the most general port in a class called **Port**. All other

ports with various restrictions and differing characteristics are subclasses of **Port**.

Another change in the model of interprocessor communication is the use of a shared memory instead of an interconnection network. In fact, different models of computation can be explored using optical networks. Massively parallel electronic architectures often communicate via a message passing scheme as a shared memory model is difficult to implement. However, a shared memory model of communication between processors may be easier to build with optics than with electronics due to the greater fan-in capability [27].

6.1.4 Aid in Programming Architectures

Various estimates indicates that as much as 80% of computer development costs comes from software development [81]. Increasing the cost of the complexity of the hardware can be justified if it decreases the cost of software development.

Boehm's study of software development costs [82] shows that software costs are increasing every year while hardware costs are decreasing. The article also lists methods for increasing software productivity:

1. efficient development environments,
2. eliminate steps that can be automated (e.g. documentation and quality assurance),
3. eliminate rework,
4. rapid prototyping, and
5. reusability of software.

The methodology behind the SCOOP model makes use of some of these productivity aids. The Smalltalk-80 system is a well integrated development environment for constructing prototypes quickly. The

abstract descriptions of the elements of the architecture are done in class descriptions so that the software is reused extensively. The characteristics of the *entire* architecture can be changed by altering the class descriptions that globally effect the entire model.

Using SCOOP to model architectures will assist in the programming of those architectures. The abstract representations of the processors in SCOOP are objects in Smalltalk. Each object has local memory and can receive messages. Those messages can reflect the instruction set, or modules of code to execute certain tasks. The design of the methods associated with those messages are modular and lead to efficient and effective integration of the software.

If a goal of architecture development is to reduce overall costs, then the ability to fine-tune or even totally reconfigure the architecture design might be justified if it can reduce the overall software development costs. Hardware prototypes do not exhibit the flexibility and reconfigurability of software prototypes. The object oriented paradigm used to create the SCOOP architecture allows for such flexibility. In fact, the SCOOP project can be thought of as a "template" for arbitrary architectures. The class descriptions of **Processors** (and its possible subclasses), communication structures (e.g. **UnidirectionalPort**) and their interconnections are the parameters that mold SCOOP into a prototype of the desired architecture.

6.2 Comparison of Pyramid to Other Architectures for Vision

Regardless of the method of modeling the architecture, a motivation of this research is to determine if the pyramid architecture can be used as a unified architecture for a wide range of computer vision tasks. The results in Chapter 5 show that the pyramid performs well in comparison with some other popular architectures. The hierarchical nature of the pyramid provides effective top-down control and smooth bottom-up

communication of results. The many levels, differing in size and power, provide the flexibility needed to handle data at various levels of abstraction. The simulations performed (using the SCOOP pyramid as a testbed) meets the requirements for vision architectures as listed in Section 2.4.

6.3 Conclusion and Future Work

In summary, a working software prototype of a pyramid architecture has been constructed. An object-oriented methodology is used for constructing the model of the architecture. The SCOOP pyramid is then used as a testbed to perform simulations of a wide range of computer vision tasks. The motivation behind this is to explore the potential of the pyramid architecture as a single, unified architecture to perform a wide range of vision tasks.

This simulation of the complete scenario (described in Section 4.4 and results presented in Section 5.5) is important to the overall evaluation of an architecture for a wide range of vision tasks. The results indicate that the performance of the architecture in a whole scenario cannot be inferred from the combination of the performance of the parts. This statement is quite similar to the definition of synergy as put forth by Fuller [83]

Synergy Synergy means behavior of whole systems unpredicted by the behavior of their parts taken separately.

Both the usefulness of the pyramid as an architecture for vision work and the methodology of prototype construction are exhibited in this thesis. The conclusion is that the hierarchical nature of the pyramid makes it efficient and effective architecture for computer vision tasks.

The SCOOP model makes an effective template to prototype other computer architectures. These SCOOP models can serve as a set of testbeds for performing benchmarks on different algorithms.

Future work in this area should pursue the enhancement of the SCOOP model so that the characteristics of the actual hardware it models can be used to *directly* assist in the building or manufacturing of such an architecture. A formal method of mapping a software prototype to the hardware would go a long way in automating such a process.

Another topic of future work is to examine the relationship between action-oriented rule-bases and non-action-oriented ones. The firing of rules (both forward and backward) is a highly parallel task. It would be useful to distinguish between those rule-bases from which parallelism can be extracted from them in firing rules (and thus performing actions). Possibly, the modeling of a rule-base through Distributed Knowledge Object Modeling (DKOM) in ORIENT84/K can take advantage of much of the parallelism inherit in the rule-base while forming a sequence of actions to be performed [78]. Many of these actions might be pruned in the process if one antecedent proved to be false (AND parallelism).

Bibliography

[1] Ballard, D.H. and Brown, C.M., *Computer Vision*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1982.

[2] Reeves, A.P., "Parallel Computer Architectures for Image Processing," *Computer Vision, Graphics, and Image Processing*, pp. 68—88, 1984.

[3] Weems, C., Lawton, D., Levitan, S., Riseman, E., Hanson, A., and Callahan, M., "Iconic and Symbolic Processing Using a Content Addressable Array Parallel Processor," *Conference on Computer Vision and Pattern Recognition*, pp. 598—607, IEEE Computer Society, IEEE Computer Society Press, San Francisco, June, 1985.

[4] Levine, M.D., *A Knowledge-Based Computer Vision System*, in Computer Vision Systems, Academic Press, Inc., Hanson and Riseman (eds.), pp. 335—352, San Francisco, 1978.

[5] Ballard, D.H., Brown, C.M., and Feldman, J.A., *An Approach to Knowledge-Directed Image Analysis*, in Computer Vision Systems, Academic Press, Inc., Hanson and Riseman (eds.), pp. 271—282, San Francisco, 1978.

[6] Hanson, A.R. and Riseman, E.M., *Segmentation of Natural Scenes*, in Computer Vision Systems, Academic Press, Inc., Hanson and Riseman (eds.), pp. 129—163, San Francisco, 1978.

[7] Marr, D., *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*, W.H. Freeman and Company, San Francisco, 1982.

[8] Uhr, L., *Highly Parallel, Hierarchical, Recognition Cone Perceptual Structures*, in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 249—292, San Diego, 1987.

[9] Hwang, K. and Briggs, F., *Computer Architectures and Parallel Processing*, McGraw-Hill, Inc., San Francisco, 1984.

[10] Preston, K. and Uhr, L.M. (eds.), Multicomputers and Image Processing: Algorithms and Programs, Academic Press, Inc., San Diego, 1982.

[11] Gerritsen, F.A. and Verbeek, P.W., "Implementation of Cellular-Logic Operators Using 3*3 Convolution and Table Lookup Hardware," *Computer Vision, Graphics, and Image Processing,* pp. 115—123, 1984.

[12] Hanson, A.R. and Riseman, E.M., *VISIONS: A Computer System for Interpreting Scenes,* in Computer Vision Systems, Academic Press, Inc., Hanson and Riseman (eds.), pp. 303—333, San Francisco, 1978.

[13] Moldovan, D.I., "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Transactions on Computers,* pp. 1121—1126, November, 1982.

[14] Fortes, J.A.B. and Moldovan, D.I., "Parallelism Detection and Algorithm Transformation Techniques Useful for VLSI Architecture Design," University of Southern California, no. PPP 83-1, Los Angeles, 1983.

[15] Navarro, J.J., Llaberia, J.M., and Valero, M., "Partitioning: An Essential Step in Mapping Algorithms Into Systolic Array Processors," *Computer,* no. 7, pp. 77—89, July, 1987.

[16] Moldovan, D.I. and Fortes, J.A.B., "Partitioning of Algorithms for Fixed Size VLSI Architectures," University of Southern California, no. PPP 83-5, Los Angeles, 1983.

[17] Barad, H. and Moldovan, D.I., "A Systems Approach to Mapping a Karhunen-Loéve Transform into a Systolic Array," *Proc. of the International Conference on Parallel Processing,* Degroot (ed.), pp. 48—55, PSU, IEEE, and ACM, IEEE Computer Society Press, August, 1985.

[18] Burt, P.J., *The Pyramid as a Structure for Efficient Computation,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 2, pp. 6—35, New York, 1984.

[19] Fortes, J.A.B. and Wah, B.W., "Sysolic Arrays: A Survery of Seven Projects," *Computer,* no. 7, pp. 91—103, (with contributions from researchers of the 7 projects), July, 1987.

[20] Kung, S.Y., Lo, S.C., Jean, S.N., and Hwang, J.N., "Wavefront Array Processors—Concept to Implementation," *Computer,* no. 7, pp. 18—33, July, 1987.

[21] Dennis, J.B., "Data Flow Supercomputers," *Computer,* no. 11, pp. 48—56, November, 1980.

[22] Arvind and Gostelow, K.P., "The U-Interpreter," *Computer,* no. 2, pp. 42—49, February, 1982.

[23] Prasanna Kumar, V.K. and Raghavendra, C.S., "Image Processing on an Enhanced Mesh Connected Computer," *Workshop on Computer Architectures for Pattern Analysis and Image Database Management,* pp. 243—247, The Computer Society of the IEEE, IEEE Computer Society Press, November, 1985.

[24] Raghavendra, C.S., "HMESH: A VLSI Architecture for Parallel Processing," *Proc. of Conference on Algorithms and Hardware for Parallel Processing,* September, 1986.

[25] Chalasani, S.B. and Raghavendra, C.S., "Geometric Algorithms on HMESH Architecture," *Workshop on Computer Architectures for PAMI,* pp. 169—175, The Computer Society of the IEEE, IEEE Computer Society Press, Seattle, October, 1987.

[26] Sawchuk, A.A., Jenkins, B.K., Raghavendra, C.S., and Varma, A., "Optical Crossbar Networks,"*Computer,* no. 6, pp. 50—60, June, 1987.

[27] Giles, C.L. and Jenkins, B.K., "Complexity Implications of Optical Parallel Computing," *Twentieth Annual Asilomar Conf. on Signals, Systems, and Computers,* Pacific Grove, CA, November, 1986.

[28] Jenkins, B.K. and Giles, C.L., "Superposition and Digital Optical Computing," July, 1987, Submitted to Optics Letters.

[29] Tanimoto, S.L., *Paradigms for Pyramid Machine Algorithms,* in Pyramidal Systems for Computer Vision, Springer-Verlag, Cantoni and Levialdi (eds.), pp. 173—194, New York, 1986.

[30] Fritsch, G., *General Purpose Pyramidal Architectures,* in Pyramidal Systems for Computer Vision, Springer-Verlag, Cantoni and Levialdi (eds.), pp. 41—58, New York, 1986.

[31] Duff, M.J.B., *How Not to Benchmark Image Processors,* in Evaluation of Multicomputers for Image Processing, Academic Press, Inc., Uhr, Preston, Levialdi, and Duff (eds.), pp. 3—12, San Diego, 1986.

[32] Uhr, L., Preston, K. Jr., Levialdi, S., and Duff, M.J.B. (eds.), *Evaluation of Multicomputers for Image Processing,* Academic Press, Inc., San Diego, 1986.

[33] Uhr, L. and Schmitt, L., *The Several Steps from Icon to Symbol using Structured Cone/Pyramids,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 6, pp. 86—100, New York, 1984.

[34] Levitan, S.P., Weems, C.C., Hanson, A.R., and Riseman, E.M., *The UMass Image Understanding Architecture,* in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 215—248, San Diego, 1987.

[35] Cantoni, V. and Levialdi, S., *PAPIA: A Case History,* in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 3—13, San Diego, 1987.

[36] Schaefer, D.H., Ho, P., Boyd, J., and Vallejos, C., *The GAM Pyramid,* in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 15—42, San Diego, 1987.

[37] Tanimoto, S.L., Ligocki, T.J., and Ling, R., *A Prototype Pyramid Machine for Hierarchical Cellular Logic,* in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 43—83, San Diego, 1987.

[38] Baugher, E. and Rosenfeld, A., "Boundary Localization in an Image Pyramid," *Pattern Recognition,* no. 5, pp. 373—395, 1986.

[39] Klinger, A., *Multiresolution Processing,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 5, pp. 77—85, New York, 1984.

[40] Cibulskis, J. and Dyer, C.R., *Node Linking Strategies in Pyramids for Image Segmentation,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 8, pp. 109—120, New York, 1984.

[41] Crowley, J.L., *A Multiresolution Representation for Shape,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 12, pp. 169—189, New York, 1984.

[42] Preston, K. Jr., *Multiresolution Microscopy,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 21, pp. 356—364, New York, 1984.

[43] Shneier, M., *Multiresolution Feature Encodings,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 13, pp. 190—199, New York, 1984.

[44] Clark, J.J. and Lawrence, P.D., *A Hierarchical Image Analysis System Based Upon Oriented Zero Crossings of Bandpassed Images,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 11, pp. 148—168, New York, 1984.

[45] Tanimoto, S.L., *Regular Hierarchical Image and Processing Structures in Machine Vision,* in Computer Vision Systems, Academic Press, Inc., Hanson and Riseman (eds.), pp. 165—174, San Francisco, 1978.

[46] Nazif, A.M. and Levine, M.D., "Low Level Image Segmentation: An Expert System," *IEEE Trans. on Pattern Analysis and Machine Intelligence,* no. 5, pp. 555—577, September, 1984.

[47] Uhr, L., *"Recognition Cones," and Some Test Results; The Imminent Arrival of Well-Structured Parallel-Serial Computers; Positions, and Positions on Positions,* in Computer Vision Systems, Academic Press, Inc., Hanson and Riseman (eds.), pp. 363—377, San Francisco, 1978.

[48] Honavar, V. and Uhr, L., "Recognition Cones: A Neuronal Architecture for Perception and Learning," Univ. of Wisconsin, Computer Sciences Dept., no. 717, Madison, September, 1987.

[49] Pope, S.T., Goldberg, A., Krasner, G., and Bay, D., *The Smalltalk-80™ Programming System: Reference Guide and Release Notes*, ParcPlace Systems, Palo Alto, CA, revision 2.2c, 1987, DE Version - Release 1.

[50] *American Heritage Dictionary,* Houghton Mifflin Company, Boston, Second College Edition, 1985.

[51] Law, A.M. and Kelton, W.D., *Simulation Modeling and Analysis,* McGraw-Hill Book Company, McGraw-Hill series in industrial engineering and management science, San Francisco, 1982.

[52] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, Menlo Park, CA, 1984.

[53] Cox, B.J., *Object Oriented Programming: An Evolutionary Approach,* Addison-Wesley Publishing Co., Menlo Park, CA, second reprint, August, 1986.

[54] André, F., Herman, D., and Verjus, J.P., *Synchronization of Parallel Programs,* The MIT Press, Scientific Computation Series, Cambridge, Mass., 1985.

[55] Raynal, M., *Algorithms for Mutual Exclusion*, The MIT Press, Scientific Computation Series, Cambridge, MA, 1986.

[56] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Weseley, Menlo Park, CA, 1983.

[57] Pratt, W., *Digital Image Processing*, John Wiley & Sons, New York, 1978.

[58] Roberts, L.G., *Machine Perception of Three-Dimensional Solids*, in Optical and Electro-Optical Information Processing, The MIT Press, J.P. Tippett et al (eds.), Cambridge, MA, 1965.

[59] Matheron, G., *Random Sets and Integral Geometry*, Wiley, New York, 1975.

[60] Serra, J.P., *Image Analysis and Mathematical Morphology*, Academic Press, New York, 1982.

[61] Nevatia, R., *Machine Perception*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1982.

[62] Nevatia, R. and Babu, K.R., "Linear Feature Extraction and Description," *Computer Graphics and Image Processing*, pp. 257—269, 1980.

[63] Winston, P.H., *Artificial Intelligence*, Addison-Weseley Publishing Company, Inc., Menlo Park, CA, Second Edition, July, 1984.

[64] Haralick, R.M. and Shapiro, L.G., "Image Segmentation Techniques," *Computer Vision, Graphics, and Image Processing*, pp. 100—132, 1985.

[65] Levine, M.D., *Region Analysis with a Pyramid Data Structure*, in Structured Computer Vision: Machine Perception Through Hierarchical Computation Structures, Tanimoto and Klinger (eds.), pp. 57—100, 1980.

[66] Ohlander, R., Price, K. and Reddy, D. R., "Picture Segmentation using a Recursive Region Splitting Method," *Computer Graphics and Image Processing*, no. 3, pp. 313—333, 1978.

[67] Laws, K., "Goal-Directed Textured-Image Segmentation," S.R.I. International, Semiannual Technical Report, no. SRI Proj. 5355, Menlo Park, CA, September, 1984.

[68] Kushner, T., Wu, A. and Rosenfeld, A., "Image Processing on MPP: 1," *Pattern Recognition*, no. 3, pp. 121—130, 1982.

[69] Dixit, V. and Moldovan, D.I., "Discrete Relaxation on SNAP," *Conf. on Artificial Intelligence Applications,* pp. 637—644, IEEE, 1984.

[70] Duda, R., Hart, P., Nilsson, N., and Sutherland, G., *Semantic Network Representations in Rule-Based Inference Systems,* in Pattern-Directed Inference Systems, Academic Press, Inc., Waterman and Hayes-Roth (eds.), pp. 203—221, San Diego, 1978.

[71] Minsky, M., *A Framework for Representing Knowledge,* in The Psychology of Computer Vision, McGraw-Hill Book Company, Winston (ed.), Ch. 6, pp. 211—277, San Francisco, McGraw-Hill Computer Science Series, 1975.

[72] Zucker, S.W., *Production Systems with Feedback,* in Pattern-Directed Inference Systems, Academic Press, Inc., Waterman and Hayes-Roth (eds.), pp. 539—555, San Diego, 1978.

[73] Negoita, C.V., *Expert Systems and Fuzzy Systems,* The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1985.

[74] Sunwoo, M.H., Baroody, B.S., and Aggarwal, J.K., "A Parallel Algorithm for Region Labeling," *Workshop on Computer Architectures for PAMI,* pp. 27—34, The Computer Society of the IEEE, IEEE Computer Society Press, Seattle, October, 1987.

[75] Kung, H.T. and Song, S.W., *A Systolic 2-D Convolution Chip,* in Multicomputers and Image Processing: Algorithms and Programs, Academic Press, Inc., Preston and Uhr (eds.), pp. 373—384, San Diego, 1982.

[76] Tanimoto, S.L., *Sorting, Histogramming, and Other Statistical Operations on a Pyramid Machine,* in Multiresolution Image Processing and Analysis, Springer-Verlag, Rosenfeld (ed.), Ch. 10, pp. 136—147, New York, 1984.

[77] Levialdi, S., "Issues on Parallel Algorithms for Image Processing," in The Characteristics of Parallel Algorithms, Jamieson, Gannon, and Douglass (eds.), The MIT Press, Cambridge, Mass., pp. 191—208, Ch. 7, 1987.

[78] Ishikawa, Y. and Tokoro, M., *Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation,* in Object-Oriented Concurrent Programming, The MIT Press, Yonezawa and Tokoro (eds.), pp. 159—198, Cambridge, Mass., 1987.

[79] *HUMBLE™ V1.1 Reference Manual,* Xerox Special Information Systems, Pasadena, CA, June, 1986.

[80] Weems, C., Levitan, S., Hanson, A., and Riseman, E., "The Image Understanding Architecture," *Proceedings: Image Understanding Workshop,* pp. 483—496, Los Angeles, 1987.

[81] McAulay, A.D., "Spatial Light Modulator Interconnected Computers," *Computer,* no. 10, pp. 45—57, October, 1987.

[82] Boehm, B.W., "Improving Software Productivity," *Computer,* no. 9, pp. 43—57, September, 1987.

[83] Fuller, R.B., *Synergetics,* MacMillan Publishing Co., Inc., New York, 1982 paperback edition, 1975.