CUCS-139-84

Image Understanding Algorithms on Fine-Grained Tree-Structured SIMD Machines

Hussein A. H. Ibrahim

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
1984

ABSTRACT

Image Understanding Algorithms on
Fine-Grained Tree-Structured SIMD Machines

Hussein A. H. Ibrahim

An important goal for researchers in computer vision is the construction vision systems that interpret image data in real time. Such systems typically require a large amount of computation for processing raw image data at the lowest level, and for sophisticated decision making at the highest level. Recent advances in VLSI circuitry have led to several proposals for parallel architectures for computer vision systems. In this thesis, we demonstrate that fine-grained tree-structured SIMD machines, which have favorable characteristics for efficient VLSI implementation, can be used for the rapid execution of a wide range of image understanding tasks. We also identify the limitations of these architectures and propose methods to ameliorate these difficulties. The NON-VON supercomputer, currently being constructed at Columbia University, is an example of such an architecture.

The major contribution of this thesis is the development and analysis of several parallel image understanding algorithms for the class of architectures under consideration. The algorithms developed in this research have been selected to span different levels of computer vision tasks. They include image correlation, histogramming, connected component labeling, the computation of geometric properties, set operations, the Hough transform method for detecting object boundaries, and the correspondence problem in moving light display applications. The algorithms incorporate novel approaches to reduce the effects of communication bottleneck usually associated with tree architectures.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I wish to express my sincerest thanks to my research advisors, David Shaw and John Kender, for their constant guidance, support and encouragement throughout my graduate studies at Columbia University; their influence on this work and on me has been tremendous. I consider myself fortunate indeed to have had the privilege and pleasure to have worked with two such distinguished and dedicated individuals. Salvatore Stolfo has also contributed to the contents of the research reported in this thesis with his support and thoughtful counseling on several occasions.

Special thanks are due to Cyrus Levinthal of Columbia University and Robert Hummel of New York University for generously agreeing to serve as outside committee members, and for their careful reading of this dissertation.

Among the other individuals who have made contributions of various sorts to the completion of this dissertation are Theodore Bashkow, Mary Forry, Bruce Hillyer, Richard Korf, Jim Kurose, Gerald Maguire, and Stephen Unger.

But beyond all else, I owe much to my wife Nihal Nounou, not only for her continuous support and encouragement, but also for her help in proof-reading this thesis and for her constructive criticism of its contents.

This research was supported in part by the Defense Advanced Research Projects Agency, under contract N00039-84-C-0165, and by the New York State Center for Advanced Technology in Computers and Information Systems at Columbia University.

To my parents

Chapter 1 Introduction

The field of image understanding, also referred to as computer vision or image analysis, has developed quickly during the last decade, with growing applications in various fields. Industrial production, medicine, space exploration, robotics, and the discovery of natural resources are but a few examples of such areas. An important goal for researchers in this field is to construct computer-based vision systems that receive an image or a sequence of images from a sensory device and output an interpretation of this input in real time. Input images with reasonable resolution contain large quantities of data, and conventional von Neumann machines require an excessive amount of time to sequentially process the fetched data. Image understanding applications, however, usually involve computations that can be performed simultaneously on many or all of the image elements. Consequently, parallel computers are highly desirable for fast execution of image understanding tasks.

A computer implementation of a complete vision system not only requires the performance of many computations on large structured arrays of raw image data at the lowest level, but also sophisticated decision making at the highest level. With recent advances in very large scale integrated (VLSI) circuitry, it is feasible now to embed a number of processing and memory elements within a single chip in a cost-effective manner. This has led to a surge in research aimed at developing new computer organizations that meet the large computational and decision requirements of image analysis tasks by exploiting the new technology. Various kinds of special parallel machines for computer vision have been proposed and some have been implemented; examples are described in [Duff 76], [Krus 76], [Dyer 81], [Kush 83], [Pott 83], and [Reev 84].

The organization of some of the proposed machines is based on a very large number of very small ***processing elements*** (PE's). Throughout this thesis, we will refer to such machines as ***fine-grained*** or ***highly parallel*** machines. In such machines, different schemes are used to interconnect the PE's. For example, the PE's can be connected together in the form of a two-dimensional mesh, or they can be placed at the nodes of a binary tree. If all the PE's simultaneously execute the same instruction on their own data, the machine is said to be executing in ***single instruction stream, multiple data stream*** (SIMD) mode [Flyn 72]. On the other hand, if the PE's execute different instruction streams concurrently on different data streams, then the machine is said to be executing in ***multiple instruction stream, multiple data stream*** (MIMD) mode.

In this thesis, we investigate how fine-grained tree-structured SIMD computer architectures, which have favorable characteristics for efficient VLSI implementation, can be used for the rapid execution of a wide range of vision tasks. We also discuss certain limitations of these architectures as vision machines and propose methods to ameliorate these difficulties The NON-VON supercomputer, currently being constructed at Columbia University[1], is a representative example of this class of architectures

Several parallel image understanding algorithms, spanning different levels of vision algorithms, have been developed and implemented on a functional simulator of the NON-VON machine. Some of these algorithms have also been tested on a machine instruction-level simulator of NON-VON The design, implementation and time analysis of these algorithms are discussed in this thesis, as are issues related to image representation and image I/O NON-VON's performance for the developed algorithms is also compared with that of other highly parallel vision machines.

In the rest of this chapter, we first discuss the nature of image understanding tasks and the manner in which they affect the design of the proposed architectures In Section 1 2, we outline the central issues related to the construction of such highly parallel machines, and describe the motivations for this research In Section 1 3, we state the major contributions of this work and outline the organization of subsequent chapters

[1]The first prototype is expected to be completed by March 1985.

1.1. Levels of Computer Vision Tasks

Computer vision tasks may be divided into three characteristic classes *low-level vision*, ***intermediate-level vision***, and ***high-level vision***. Low-level image processing deals with the raw image data received from the sensory devices and usually produces an output of the same size as the input. Low-level vision processing is sometimes referred to in the literature as ***signal level processing***, since input image data may be viewed as a signal to be processed. Examples of low-level vision tasks include image restoration, noise removal, gathering certain statistics about the image, image enhancement, and simple feature extraction such as edge detection [Ball 82]. Since such tasks usually involve the execution of the same sequence of instructions repeatedly on all of the image data, they are well suited for fast execution on machines of SIMD architecture. Most special hardware systems proposed for image understanding tasks are of this type.

Intermediate-level vision tasks are usually concerned with aggregating image features obtained from low-level vision tasks and transforming the image into some symbolic representation, such as labeled graphs of relations between image features. Examples of such tasks include the Hough transform method for detecting object boundaries described by parametric curves, and interpreting the shape of three-dimensional objects from two-dimensional images. Intermediate-level vision tasks may be viewed as the interface between low-level processing and processing of the symbolic image representations on the high-level. There are some open research questions regarding tasks on this level. Examples of such problems include the choice of the best sets of features to be extracted by low-level procedures for the task at hand, and the manner in which they are represented for use in high-level vision tasks.

High-level vision tasks accept a symbolic representation of an image and classify image features and segments into known classes. They also match these symbolic representations to known symbolic structures for the final interpretation of the image. Techniques used in these tasks are similar to those used in the fields of artificial intelligence and pattern recognition. High-level vision tasks usually involve multiple operations that can be executed independently. For example, the same image segment can be analyzed using different techniques. Architectures suited for these kinds of tasks are usually of the MIMD class.

In Chapter 2, we overview some of the architectures that have been proposed to implement these tasks, and discuss the advantages and disadvantages of these different architectures.

1.2. Motivations For This Research

There are several motivations for this research. First, tree-structured machines have favorable characteristics for efficient VLSI implementation, such as area-efficient layout, simple interconnection scheme, and a bounded number of I/O ports per chip. Thus, tree machines are easy to construct and expand. The reader is referred to [Ibra 83] for a detailed discussion of this aspect of tree machines.

Second, NON-VON's hardware, with its support for the fast global broadcast of instructions and data to all PE's, is well-suited for the rapid execution of a wide range of vision tasks, especially low-level SIMD vision tasks. A detailed discussion of this aspect of NON-VON's architecture is presented in Chapter 2.

Third, NON-VON has some special hardware features that have been designed to support large scale data processing, and which have proven useful in vision tasks to ameliorate some of the problems related to the communication bottleneck generally associated with tree architectures. We will describe these features in Chapter 3.

The fourth motivation is based on the fine granularity of the NON-VON PE's, which is well suited to image analysis tasks involving large amounts of data.

Furthermore, the hierarchical nature of the NON-VON tree allows the efficient implementation of existing hierarchical and multi-resolution algorithms for image analysis. Algorithms based on the aggregation of values computed at a number of image points can be executed very quickly by virtue of the hierarchical nature of the machine.

NON-VON's architecture also supports the concurrent manipulation of massive amounts of symbolic data, which is useful in high-level vision tasks. Relational image databases can be handled very efficiently on the NON-VON machine. The implementation of such systems is discussed in [Shaw 82]. Rapid execution of expert systems on tree machines is also discussed in [Stol 82]. The efficient use of image databases and of expert vision systems for high-level vision, are interesting research questions that are, however, beyond the scope of this thesis.

1.3. Contributions of This Research and Organization of Subsequent Chapters

The major contribution of this thesis is the development and analysis of parallel algorithms for several image understanding tasks on highly parallel tree-structured SIMD machines. The image analysis applications considered in this thesis have been selected to span different levels of computer vision applications. These algorithms incorporate novel approaches to exploit the machine's tree organization and to reduce the effects of communication bottleneck usually associated with tree architectures.

Issues that affect the design and time analysis of these algorithms are also addressed in this dissertation. Image representation in tree machines is one such issue. We describe how hierarchical data structures can be modified to represent images in the NON-VON tree. Fast image I/O is an important factor for efficient implementation of vision algorithms. In this thesis, we propose different methods to perform I/O efficiently in tree machines.

More specifically, we have developed and analyzed parallel algorithms for fast image correlation, and for quasi-parallel connected component labeling. A fast, distributed, space-efficient algorithm has been developed to implement the Hough transform method for detecting object boundaries. We have also developed a parallel algorithm that quickly enumerates possible solutions for the correspondence problem in moving light display applications. Other fast algorithms have been developed, including image histogramming, set operations, and the computation of the geometric properties of objects.

NON-VON's performance for different image algorithms is analyzed and compared with that of other highly parallel image understanding architectures. Two simulators have been used to simulate the image algorithms. A functional simulator has been implemented on a VAX

11/750 augmented with a Grinnell image processor, and using the programming language C We have used this simulator to test all of the algorithms described in this thesis. A Lisp-based machine instruction-level simulator that has been developed for the NON-VON machine is used to execute some of the image algorithms Based on simulation results, NON-VON's performance is compared with that of other highly parallel architectures for image analysis systems, and many algorithms are shown to execute faster on NON-VON than on other highly parallel machines. We have also identified the limitations of tree machines in the rapid execution of certain image analysis tasks, and have proposed special modifications to the NON-VON architecture for the rapid execution of these tasks

In what follows, we outline the organization of the remaining chapters In the following chapter, a number of special parallel architectures for image understanding are reviewed, with an emphasis on their basic architectural features and the vision applications for which they are best suited The NON-VON architecture is described in Chapter 3, and is compared with other proposed hierarchical architectures for vision A parallel programming language, based on PASCAL, is also described in Chapter 3 This language, referred to as N-PASCAL, is used to describe the developed algorithms throughout the thesis

In Chapter 4, we introduce certain hierarchical data structures for image processing, and demonstrate how they can be used to represent images in the NON-VON tree. We also discuss in Chapter 4 the initialization procedures for the NON-VON tree, along with various issues related to image I/O.

Four groups of algorithms are presented in this thesis:

1. Signal level processing algorithms.
2. Geometric algorithms.
3. Aggregation algorithms.
4. High-level algorithms.

Examples of these groups are presented in Chapters 5 through 8, respectively. The first two groups represent low-level vision tasks, while the third and fourth groups represent intermediate- and high-level vision tasks respectively. Time analysis results are presented for each algorithm, and NON-VON's performance is compared with that of other architectures. Simulation results, obtained by implementing the image understanding algorithms on the functional simulator and on the NON-VON instruction-level simulator, are also presented and analyzed. Chapter 9 includes the conclusion of this thesis and outline possible directions for further research.

Chapter 2
Parallel Image Processing Architectures: An Overview

In this chapter, we describe several of the parallel architectures that have been proposed for computer vision, with an emphasis on the match between their underlying architectural features and various image analysis tasks Advantages and disadvantages of the surveyed machines as vision machines are also discussed These architectures may be classified into four categories based on the scheme used to interconnect the processing elements

1 Mesh-connected architectures

2 Pipelined architectures.

3 Multiprocessor Architectures

4 Hierarchical architectures.

We will focus on the last of these architectural families, and will show in more detail the motivation behind it

2.1. Mesh-Connected Architectures

Cellular logic arrays, proposed by Unger [Unge 58], [Unge 59] for use as parallel image processors, form the basis for many later architectural proposals in this category. In cellular logic arrays, also referred to as two-dimensional arrays or parallel array processors, an image is divided into a regular two-dimensional array of *cells*, with a PE assigned to each cell. Physically adjacent PE's can communicate with each other, and each PE has some local storage and some hardware to manipulate its contents. The PE's execute in SIMD mode with instructions broadcast by the host computer. Figure 2-1 shows the organization of a two-dimensional cellular array.

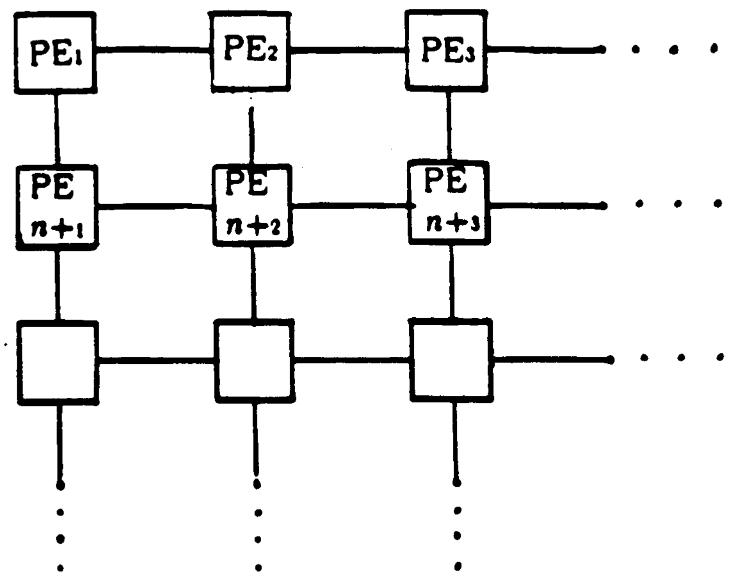


Figure 2-1: A Two-Dimensional Cellular Array (Adapted from [Rose 83])

Loading and unloading of images are usually performed alongside the perimeter of the array. All the data paths within a single PE are typically one bit wide, for this reason, such machines are also referred to as ***binary array processors*** [Reev 84]. With recent advances in VLSI, machines containing as many as 16K one-bit PE's (organized in a 128 x 128 array) have been constructed.

The fundamental advantage of this family of architectures is that it maps the physical adjacency of image elements directly into hardware, thus making access to neighborhood information very rapid. Many low-level image operations, such as image filtering and local image feature detection, can be executed very rapidly in parallel on this architecture. Operations involving the gathering of statistics about the whole image are not as fast as local operations. They execute in a time proportional to the array diameter (that is, to the square root of the number of PE's).

VLSI implementation of such machines involves designing chips with a number of PE's interconnected together in the form of a rectangular grid. The PE's on the perimeter of the grid communicate with other chips through I/O ports. With VLSI device dimensions scaling down, an increasing number of PE's can be embedded on one chip. However, the number of pins required for inter-chip communication increases in proportion to the square root of the number of PE's per chip. Thus, the number of PE's to be embedded on one chip is limited by the number of pins allowed by the technology. One way of dealing with this problem involves time-multiplexing the use of I/O ports between several PE's on the perimeter of the chip [Weem 84]. This, however, reduces the speed of inter-chip communication.

Examples of operational machines in this architectural family include CLIP4 [Duff 76] which is a 96 × 96 PE LSI machine, the MPP [Pott 83] with an array of 128 × 128 PE's, and the ICL DAP [Mark 80], containing an array of 64 × 64 PE's. A further discussion of cellular logic arrays can be found in [Rose 83] and [Reev 84].

2.2. Pipelined Architectures

Other parallel machines proposed for image understanding make use of pipelining as a way of introducing parallelism into the system.

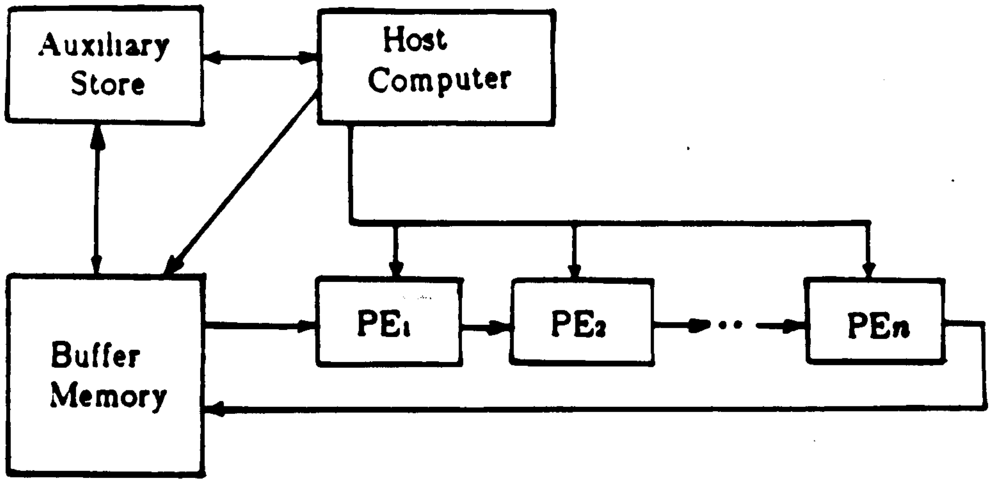


Figure 2-2: Organization of Pipelined Architectures (Adapted from [Reev 84])

Figure 2-2 depicts the basic organization of this family of architectures.

Image data is passed to the first stage of the pipeline from the scanning device, or from a buffer memory. The function of each stage is specified by the host computer through the instruction bus

Machines of this type are most efficient in real-time low-level image processing applications where the image data source is connected to the machine directly, and generates data at the speed of a simple pipeline step. Such architectures can be fully utilized when the image processing tasks have a number of steps equal to the number of the pipeline stages. The architecture has its limitations, though, and when dealing with more than one image at a time, or when performing operations such as geometric corrections, due to the limited interconnection scheme of the pipeline

An example of this family of architectures is the Cytocomputer [Ster 83], which is used in biomedical image processing. The Cytocomputer has 80 binary stages in one pipeline and 25 grey level stages in a second pipeline, each stage operates on a three by three window in the image. Binary stages are capable of implementing all possible logical operations on the nine elements by means of a look-up table memory in each stage. This makes such operations extremely fast in these stages. The gray level stages can perform 8-bit arithmetic operations on their window operands

2.3. Multiprocessor Architectures

Members of this category of architectures make use of a high-bandwidth interconnection network for communication between an independent set of PE's

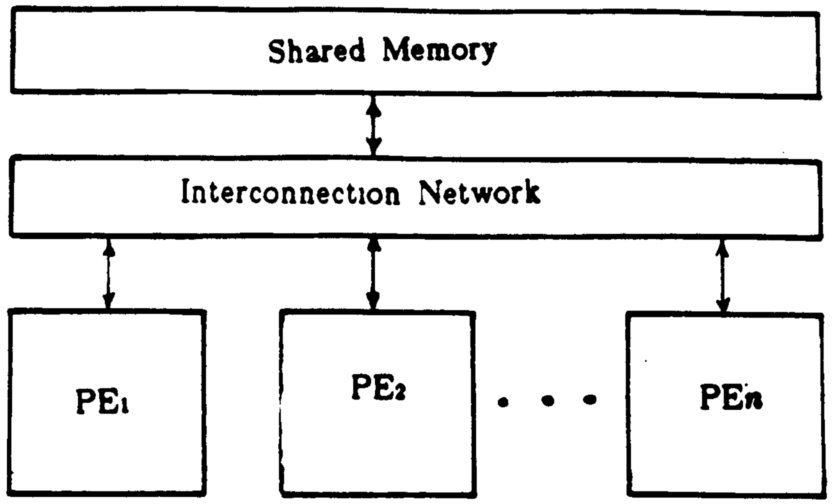


Figure 2-3: An MIMD Architecture (Adapted from [Reev 84])

The processors in such architectures can typically execute different programs (MIMD mode), or the same program (SIMD mode) at any point in time Communication among the PE's is affected by sending messages through the interconnection network or through a shared memory, as shown in Figure 2 3

Parallel machines of this type are mostly efficient in executing high-level image understanding tasks, in which the image is no longer represented as a large array of data, but rather in the form of a symbolic description of objects For example, different processors may be assigned different algorithms for the analysis of the same image object Because of the complexity of the interconnection network required to connect the PE's, machines of this type can not embody more than few thousand processors The distribution of tasks between independent processors and the synchronization of different PE's present added complications

Examples of this class of architectures include PASM [Sieg 81] and ZMOB [Kush 82] The proposed architecture of PASM comprises 1024 PE's interconnected together by means of a permutation network The machine also contains a number of control units that enable the machine to execute as an independent set of SIMD machines ZMOB, on the other hand, consists of 256 identical PE's, connected to each other and to a VAX 11/780 host computer by a high-speed bus The PE's communicate with each other and with the host machine by means of messages transferred through the bus The reader is referred to [Kush 82] and [Sieg 81] for a description of the implementation of some image tasks on these two machines

2.4. Hierarchical Architectures

A fourth architectural approach is suggested by the vision systems of humans and higher animals The human visual system processes pictorial information through a series of layers, each containing a large set of parallel receptors and processors Input information to the visual system is received by a large parallel set of sensory receptors, the rods and cones, in the retina. The retina behaves in many ways like a mesh of PE's. This raw information is then transformed into gradients, and contours are enhanced by two other parallel sets of processor layers in the retina [Uhr 80]. The transformed information is then carried by the optic nerve to the higher portions of the visual system. Information reaching the higher portions is compressed in size by a factor of about 100, and it is then processed and transformed by several layers of parallel processors.

These observations have generated proposals for hierarchical architectures for image understanding systems. Such architectures are often referred to in the literature as ***hierarchical***, ***cone***, or ***pyramid*** machines. In hierarchical architectures, processing takes place in a series of levels, as shown in Figure 2.4. At the lowest level is the raw pictorial information input to the system by a sensory device. A set of transformations is first performed on this input; their output is then either stored on the same level or passed to the next level in the hierarchy. This process continues for several layers in the hierarchy. Data may also flow top-down in the hierarchy of layers, or laterally within any layer.

Hierarchical architectures are well suited to the fast execution of multi-resolution feature (color, texture, edge, etc.) algorithms. In addition,

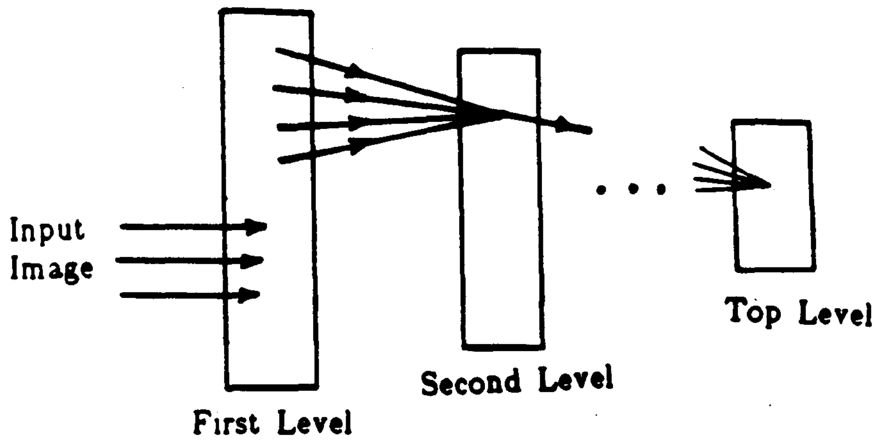


Figure 2-4: A Hierarchically Organized Architecture (Adapted from [Uhr 84])

many algorithms that use hierarchical data structures, such as quadtrees, [Klin 76], can be implemented efficiently on hierarchical machines Global feature information, such as bit counting, can be rapidly accumulated at the top of the hierarchical structure

A number of special hierarchical parallel machines have been proposed for image processing tasks ([Hans 78], [Uhr 78], [Dyer 81], [Tani 83a]) The ***pyramid machine*** proposed by Dyer [Dyer 81] is a representative example of this class of architectures The organization of a pyramid machine is shown in Figure 2 5 PE's in each layer are organized as two dimensional arrays, with each PE capable of communicating with its immediately adjacent PE's Each PE also communicates with four PE's in the layer below it, and with one PE in the layer above it Pyramid

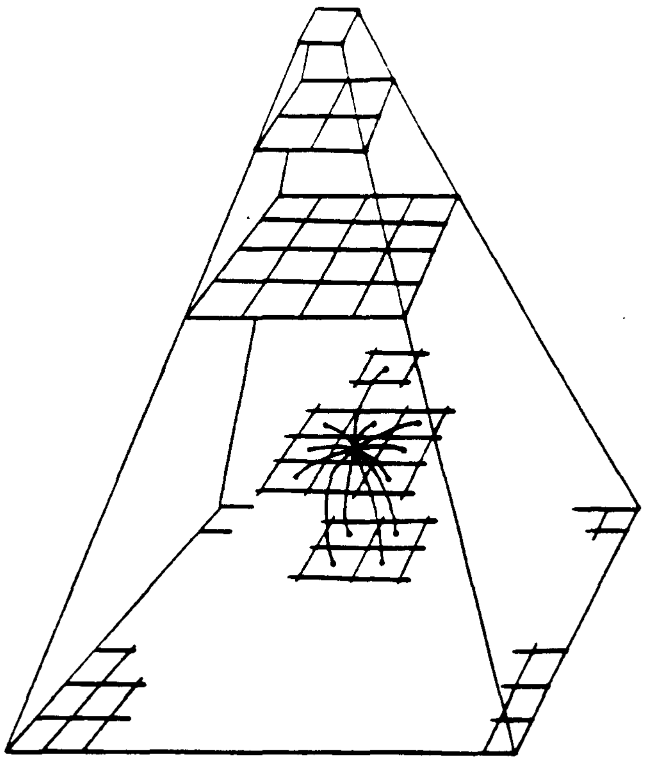


Figure 2-5: Organization of the Pyramid Machine (From [Tani 83a])

machines are difficult to build because of the complexity of their wiring, and only a few projects for building a 16 × 16 base pyramid machines are under way ([Tani 83a] and [Scha 84])

The NON-VON supercomputer [Shaw 82] is another example of a hierarchical machine Its architecture includes a large number of small PE's that form the nodes of a complete binary tree NON-VON has been designed to support the massively parallel manipulation of data records stored in its PE's This aspect of the NON-VON machine makes it attractive for vision applications that involve a large amount of data. Furthermore, hierarchical data structures proposed for image analysis, including multi-resolution pyramids and quadtrees, can be effectively used to represent images on NON-VON Also, the binary image tree data structure proposed by Knowlton [Know 80] as a variant of quadtrees, can be mapped directly onto the NON-VON machine to represent binary images in a manner to be described in Chapter 4

The present version of the NON-VON architecture differs from other proposed pyramid machines in that it does not implement in hardware the mesh connections at each level Thus, local operations execute faster on mesh-connected and pyramid machines Careful design of the algorithms can speed up these operations considerably, as will be described later The architecture of NON-VON is described in the following chapter, and the differences between its architecture and other proposed pyramid machines are presented

Chapter 3 The NON-VON Supercomputer Architecture

The name NON-VON is used to describe a family of massively parallel tree-structured machines intended to support large scale data manipulation [Shaw 84] The architectures of all the NON-VON family members include a tree-structured primary processing subsystem (PPS) based on custom VLSI circuits, along with a secondary processing subsystem (SPS) based on a bank of intelligent disk drives. Figure 3-1 shows the top-level organization of the NON-VON architecture.

The PPS is configured as a binary tree of small processing elements (SPE's) Each SPE comprises a small RAM (up to 256 bytes), a modest amount of processing logic, and an I/O switch that supports various modes of communication within the tree, as will be described in Section 3.2.

The SPS is based on a number of rotating storage devices. Associated with each disk head in the SPS is a separate sense amplifier and a small amount of logic capable of dynamically examining the data passing

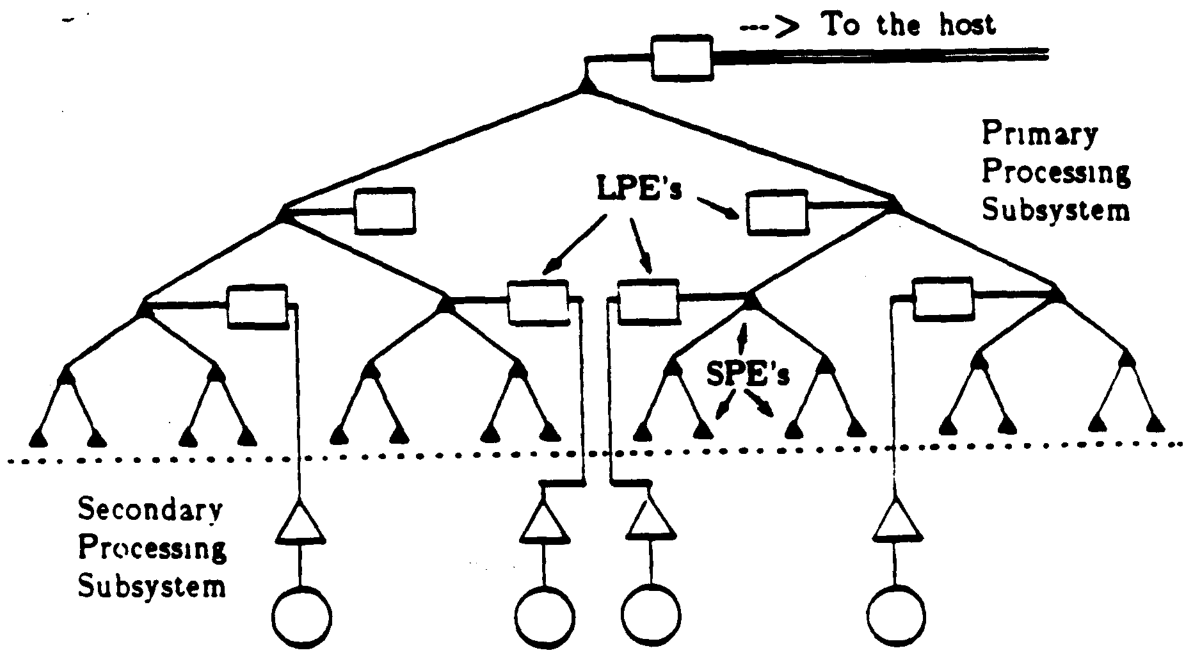


Figure 3-1: Top Level Organization of NON-VON (From [Hill 83])

beneath it [Shaw 82]. This organization supports parallel transfer of data between the PPS and SPS, which is necessary to keep I/O from becoming a bottleneck.

NON-VON 1 and NON-VON 3, the first two members of the NON-VON family, include a single special control processor (CP) at the root of the tree. The CP is responsible for coordinating different activities within the PPS. It is capable of broadcasting instructions to be executed simultaneously by all active PE's. Thus, NON-VON 1 and NON-VON 3 function for the most part as SIMD machines, with all SPE's simultaneously executing the same instruction. (The single exception involves transfers between the SPS and the PPS, which will not be discussed in this dissertation.) We will call the algorithms that use this mode of execution ***SIMD algorithms***.

The first member of the NON-VON family, NON-VON 1, contains chips with only one PE, and is being constructed primarily to evaluate certain electrical, timing, and layout area characteristics. The chip has already been tested and has been proven functional. A modified version of the chip with eight PE's has been designed for use in NON-VON 3. The modified chip, partial prototype of which has recently been fabricated, has less area per PE, and the instruction set has been made more powerful by generalizing register-to-register data transfers and adding more arithmetic processing power.

The design of the NON-VON 3 PE is briefly described in the following section. All algorithms developed in this thesis are based on the NON-VON 3 architecture and instruction set. Appendix A contains a listing of all such instructions. It is expected that the time required to execute a NON-VON 3 instruction in all PE's in a tree of 15 levels (32K PE's) is approximately 250 nsec. We will use this number throughout this thesis to compute the execution time for the developed algorithms.

The emerging design for NON-VON 4 [Shaw84a] would include a number of *large processing elements* (LPE's) connected to all nodes above a certain tree level. Each LPE would include an off-the-shelf 32-bit microprocessor, a reasonable amount of memory (between 256K bytes and one megabyte), and some special hardware to interface with the rest of the machine. A high-bandwidth multi-stage interconnection network would be used to interconnect the set of LPE's. The LPE's would be capable of executing their own programs, or of functioning as CP's for the subtrees they root. Thus, NON-VON 4 would have the capability of executing in MIMD and "*multiple-SIMD*" (MSIMD) modes. The LPE network should significantly improve the bandwidth of communication involving the top of the tree.

In the following sections, we describe the design of the SPE in the NON-VON 3 machine and the various communication modes supported by both the NON-VON 1 and NON-VON 3 machines. We also introduce the N-PASCAL programming language, which will be used to describe the vision algorithms developed on the NON-VON machine.

3.1. The Small Processing Element Design

Figure 3-2 depicts the main functional blocks of the NON-VON 3 PE. They are the eight-bit ***arithmetic logical unit*** (ALU); an array of five byte registers, called A8, B8, C8, IO8, and IMAR; an array of five one-bit registers, called A1, B1, C1, IO1, and EN1; a 64 word × 9-bit *random access memory* (RAM), and two special combinational networks, called the I/O switch and the RESOLVE circuit [Shaw 84b]. A PE executes the instructions broadcast by the CP as long as its ***enable bit***, EN1, is set. If the enable bit is reset, the PE is disabled and only an ENABLE instruction will activate it again.

Two internal buses, called the A bus and the IO bus, run through the data path. Both are capable of transferring either one- or eight-bit data, depending on the instruction being executed. The A bus is used to transfer data between the registers, the RAM, and the I/O switch. The IO bus is required to support inter-PE communication, as will be seen in the following section. It connects the dual-port registers IO and A, the I/O switch, and the ALU.

The ALU comprises an eight-bit comparator that compares the contents of the A8 register with one of the other eight-bit registers, and sets A1 and B1 to indicate the result. If A8 is compared with B8, for example A1 is set iff A8 is equal to B8, and B1 is set iff A8 is less than B8. Eight-bit addition, subtraction, and logical operations are also supported by the ALU. In the case of addition and subtraction, C1 is used to hold the carry output. One-bit logical operations are also supported by a special one-bit logical function unit. The RAM allows access to one 8- or 1-bit location per instruction cycle, and the IMAR register is used to store the memory address used in RAM operations.

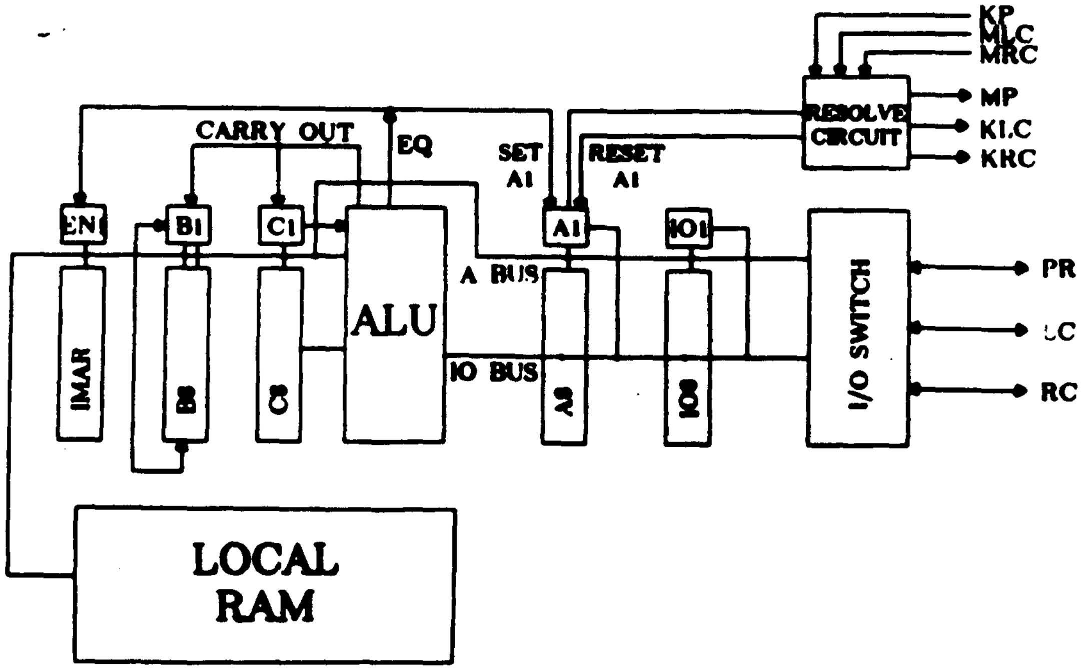


Figure 3-2: NON-VON 3 Block Diagram of the Small Processing Element (From [Shaw 84b])

3.2. Communication in NON-VON

Inter-PE communication in NON-VON is supported by the I/O switch, which is a matrix of pass transistors that routes data between the two internal buses and the I/O ports. The NON-VON I/O switch supports the following three modes of communication.

1 *Global bus communication*, supporting both broadcast by the CP to all PE's in the PPS as required for SIMD execution, and data transfers from a single selected PE to the CP. No concurrency is achieved when data is transferred from one PE to another through

the CP using the global communication instructions An instruction called RESOLVE can be used to disable all but a single PE chosen from a specified set of PE's. This is an example of a hardware ***multiple match resolution*** scheme, in the terminology of the literature of associative processors. The CP, upon executing a RESOLVE instruction, is able to determine whether executing the instruction has resulted in any PE being enabled. The REPORT instruction transfers data from the single chosen PE to the CP using global bus communication.

2. ***Tree communication***, supporting data transfers among PE's that are physically adjacent within the PPS tree. Instructions support data transfers to the parent (P), left child (LC), and right child (RC) PE's. Full concurrency is achieved in this mode, since all nodes can communicate with their physical tree neighbors in parallel.

3. ***Linear communication***, in which the whole tree is reconfigured to act as a linear array of PE's This mode of communication supports data transfers to the left neighbor (LN) or right neighbor (RN) PE's in the linear array Linear communication is useful for applications that require a predefined total ordering of data. Figure 3-3 shows how the linear logical sequence is mapped onto the tree-structured physical topology of the PPS by inorder enumeration [Knut 73]. The paths needed to transfer data concurrently between linear neighbors in the tree concurrently are shown in Figure 3-3 Two phases are required to complete the linear communication cycle. Note that every other element in the inorder sequence is a leaf node In the first phase, data is transferred along the arrows originating from the leaf PE's, while in the second phase, data passes along the black arrows terminating at the leaf PE's

The original NON-VON architecture which was not intended to efficiently support computer vision applications, differs from other proposed highly parallel hierarchical image understanding architectures (for example, [Tani 83]) in that it does not employ any extra physical links to perform mesh neighbor communication This has certain advantages from a hardware point of view, as it results in a fixed

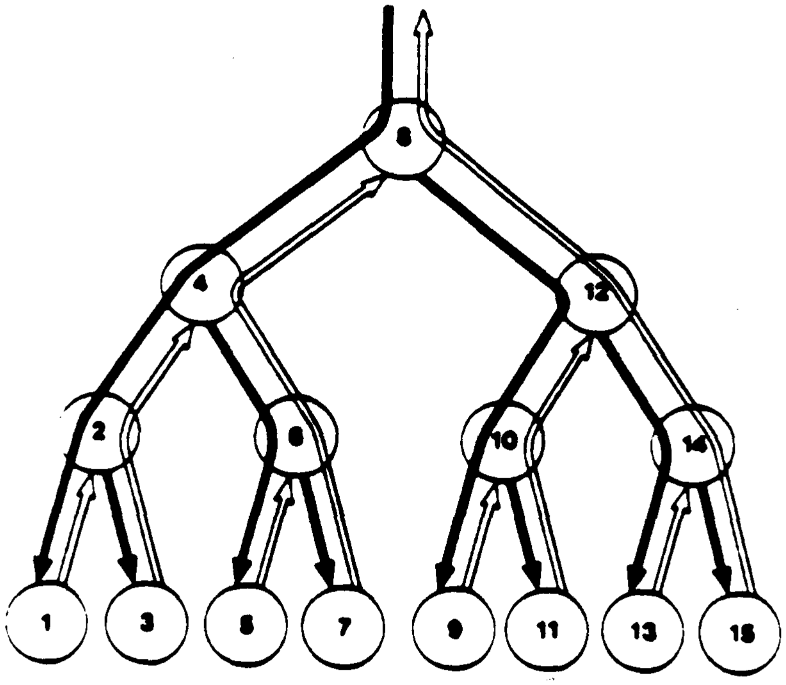


Figure 3-3: Inorder Embedding of the Linear Array (From [Shaw 82])

number of pins per integrated circuit chip, independent of the number of PE's on that chip This makes it possible to increase the size of the tree as chip dimensions scale down by simply embedding more PE's on the chip Increasing the machine size involves only removing the old PE chips and plugging in the new ones On the other hand, the lack of mesh connections slows many local operations in which the output value at an image point depends on its own image value and that of neighbor points

A vision-oriented variant of the NON-VON 3 machine that includes mesh connections to supplement the current tree-structured architecture is now in the early stages of design Alternative algorithms exploiting these hardware modifications will be discussed later in this thesis.

NON-VON's other special hardware features, including its ability to be configured logically as a linear array, its fast global instruction broadcast and its hardware multiple match resolution scheme, have proven useful in the vision algorithms we have developed to overcome some of the problems related to the communication bottleneck generally associated with tree architectures.

3.3. N-PASCAL : An Overview

In this section we introduce a PASCAL-based parallel language called N-PASCAL, which will be used to describe the NON-VON vision algorithms presented in this thesis. This language is closely related to a PASCAL-based parallel language, NV-PASCAL, that has been designed to be used on SIMD architectures [Baco 82]. The principal idea behind the design of N-PASCAL has been to create features that make full use of the machine's parallel capabilities while retaining all of the high-level constructs of PASCAL.

N-PASCAL is based on standard PASCAL as described in [Jens 74]. One new data type and two extra constructs have been added to standard PASCAL. The new data type is referred to as the ***vector—var*** (for vector variable), and the two new constructs are the ***parallel assignment*** and ***WHERE*** statements. In addition, built-in functions allow the programmer to explicitly make use of the NON-VON tree communication instructions We now briefly describe the N-PASCAL constructs that have been used to describe image understanding algorithms presented in this thesis

The new data type ***vector—var*** is used to express the parallelism in the language Vector variables reside in the PPS and are associatively addressed, whereas standard PASCAL variables reside in the CP and are sequentially addressed In the following sections, variables that are defined to be of type vector_var are referred to as ***vector variables***, while ***scalar variables*** refer to those that are stored in the CP In the NV-PASCAL procedures described in this thesis, we will use italics to refer to scalar variables and capital letters to refer to vector variables. Small bold letters will be used to refer to the reserved keywords of the language

There are two types of statements in N-PASCAL: sequential and parallel The sequential statements are those of standard PASCAL, while the parallel statements are those that operate on vector variables The assignment statement can be either sequential or parallel. The sequential assignment statement is the assignment statement encountered in standard PASCAL The parallel assignment statement is the one that refers to a variable that is defined as a vector variable. The parallel assignment statement is executed concurrently in all active PE's in the machine For example, upon execution of the following segment of an NV-PASCAL program

```
vector_var
 COUNTER  integer;

begin
 COUNTER := 0;
```

the vector variable **COUNTER** stored in all **PE's** is initialized to zero

The **WHERE** statement is a form of parallel conditional statement that operates only on vector variables. The form of the **WHERE** statement is as follows

```
WHERE <conditional expression>
    DO <statement>
  [ ELSEWHERE <statement> ] ;
```

It is used to first select only those **PE's** with vector variables that satisfy the boolean expression. The statement following the **DO** is then executed in only those **PE's**. If the optional **ELSEWHERE** clause is included, the statement following the **ELSEWHERE** keyword is executed in the subset of the **PE's** that failed to satisfy the original conditional expression. An example of the **WHERE** statement follows:

```
vector_var
 COUNTER , VALUE  integer;

begin
 where COUNTER > 50 do VALUE := 100
   elsewhere VALUE := 0;
```

The vector variable **COUNTER** is tested in all **PE's** and in those **PE's** whose **COUNTER** value exceeds 50, the vector variable **VALUE** is set to

100 In all PE's whose COUNTER value is less than or equal to 50, the variable VALUE is set equal to zero. An important point to remember is that the WHERE statement in general executes ***both*** the statement following the DO ***and*** the statement following the ELSEWHERE (the exception being the case in which all or none of the PE's satisfy the condition).

Built-in functions based on the NON-VON instruction set are employed to implement operations that use the tree communication modes of the machine, which are described in Section 3.2. Function names that start with 'N_' correspond to NON-VON machine instructions, and their parameters correspond to the arguments of these instructions.

Chapter 4
Image Representation

In this chapter, we examine certain data structures for representing images on parallel tree-structured machines. This subject is of prime importance, as it greatly affects the design of image algorithms. Methods used for image input and output are also affected by the choice of data structure. The choice of a data structure for a set of problems can even influence the design of a machine architecture for efficiently solving those problems. For example, mesh-connected architectures map into hardware the two-dimensional array data structure used most commonly to represent images on sequential machines. Similarly, the hierarchical nature of the NON-VON architecture affects the choice of data structures used to represent images on it.

An overview of data structures used to represent images on sequential machines is presented in Section 4.1, with an emphasis on hierarchical data structures. We then demonstrate how two of these hierarchical data structures can be modified to represent images on binary trees. A procedure for initializing the NON-VON tree is presented in Section 4.2. Algorithms and issues related to the loading and unloading of images in tree machines are also discussed in that section We then describe procedures used to build the data structures employed in the algorithms described in this thesis

4.1. Image Data Structures: An Overview

Storing raw pictorial data requires a large amount of memory About 512 x 512 bytes are needed, for example, to store a single black and white television frame Two-dimensional arrays are commonly used for storing images, where every element in the array represents a corresponding area in the image space. These small areas are referred to as *pixels* Pixels can take different shapes, producing different ***tessellations*** Most commonly they are squares, but they can also be rectangles, triangles, or hexagonals The value of the array elements can represent the intensity of the image at the corresponding pixels or other values such as the spectral components of color pixels Mesh-connected architectures use this data structure to represent images, with each PE being assigned a pixel, or a block of pixels in the case where there are fewer PE's than there are pixels in the image Other data structures used to represent images include chain codes, graphs, and relational databases See [Tani 80b] for further discussion of these data structures.

Hierarchical data structures can be mapped naturally onto tree machines They are often used in image understanding tasks because they allow many algorithms to be expressed in forms suitable for divide-

and-conquer techniques. They also support certain techniques for data compaction and image transmission [Know 80]. Hierarchical data structures include ***multi-resolution pyramids***, ***regular decompositions***, and *quadtrees* [Tani 80b].

In what follows, we present two of these hierarchical data structures, namely multi-resolution pyramids and a modified version of quadtrees called binary image trees, and show how they are used to represent images on NON-VON.

4.1.1. Multi-Resolution Pyramids on NON-VON

A multi-resolution pyramid can be defined as a sequence $\{I(L), I(L-1), \ldots I(0)\}$ of images, each represented as a two dimensional array, where $I(L)$ is the original image, and $I(m-1)$ is a version of $I(m)$ at half the resolution. (This is the same definition Tanimoto used in [Tani 80b].) The term "image resolution" refers to the number of pixels used to describe the image. For example, if $I(m)$ represents a version of the image with resolution 64 × 64, then $I(m-1)$ represents the same image at resolution 32 × 32. Figure 4-1 shows an image and its multi-resolution pyramid representation. The pyramid provides reduced resolution versions of the image. If more than one operation is to be performed on the image, then each operation should use only the resolution required for this operation. The extra amount of memory required to store the pyramid representation is 1/3 that of the amount of memory used to

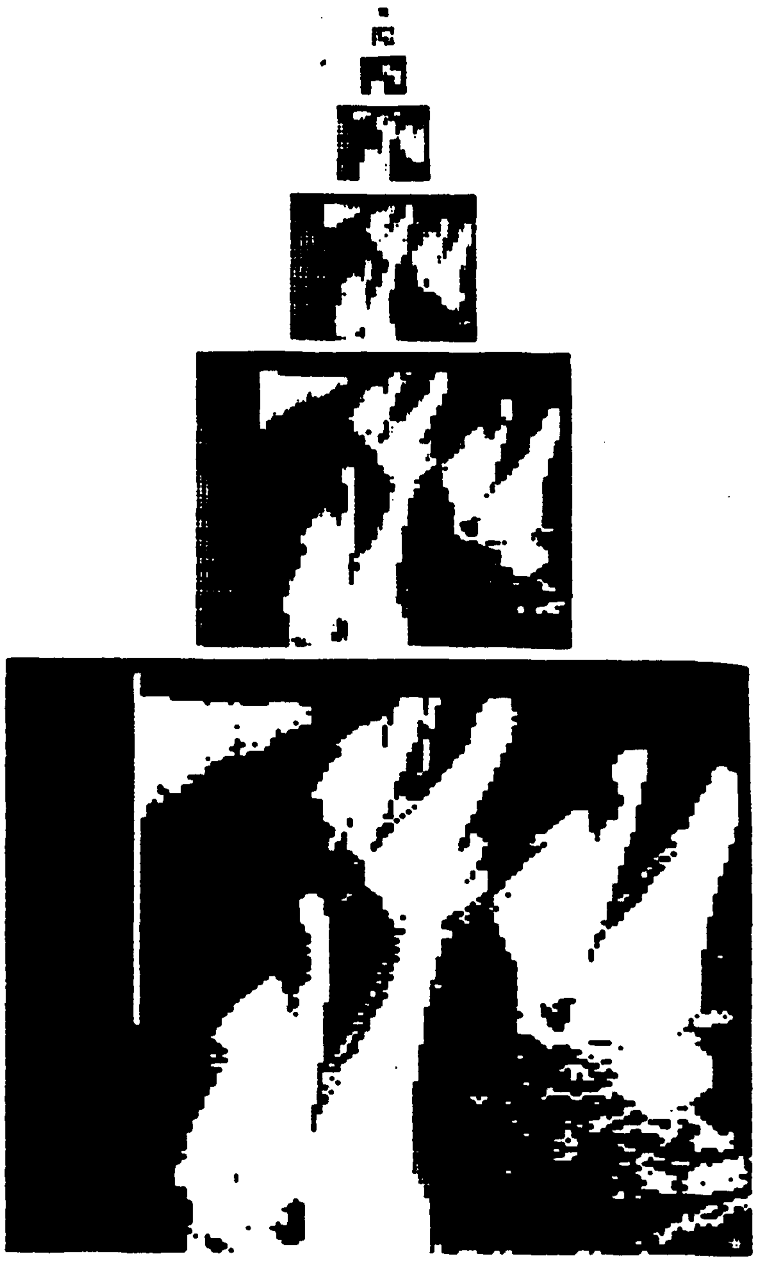


Figure 4-1: An Image and its Multi-Resolution Pyramid Representation (From [Tani 75])

store the original image

Multi-resolution pyramids can also be defined in terms of trees rather than arrays. In this case, they are referred to as ***pyramid trees***. A multi-resolution pyramid is defined in terms of *quartic (4-ary) trees* as follows. The leaf nodes represent the pixels of the image at its highest resolution (the base of the pyramid), and the nodes of an internal level of the tree represent the pixels of a reduced resolution version of the image. Thus, going from one level in the tree to the level above it results in an image with one-fourth the resolution. Note that the four child nodes of a parent node represent a 2×2 region in the image.

An image at a specific level can be computed from the image at the level below it in the pyramid tree in different ways. Typically, a parent node is set equal to the average value of its four children. This averaging process can be viewed as a kind of low-pass filtering of the image followed by resampling. Hence, images with lower resolution retain the gross features of the image. For binary images, the averaging process is defined to result in the binary value 1 only if three or more of the children have the binary value 1, and to result in 0 otherwise.

In the NON-VON tree, the leaf level is used to store the original image, whereas the internal levels are used to represent the image at coarser resolutions. Since NON-VON is a binary tree, the resolution reduction from one level in the tree to the level above it is only a factor of two, and hence two NON-VON levels are used to effect the same reduction as one level in the multi-resolution pyramid. We use this image representation whenever we deal with gray-scale images. In

Section 4.2, we show in detail how this can be done

4.1.2. Binary Image Trees on NON-VON

Binary image trees are a variant of quadtrees, which were proposed by Knowlton [Know 80] as an encoding scheme for compactly transmitting gray-scale and binary images. Quadtree data structures are similar in many aspects to multi-resolution pyramids. A good way to visualize the quadtree is by assuming that the image is a square whose dimensions are a power of 2. The quadtree data structure is built by subdividing the whole image into four square quadrants with dimensions that are half that of the image. This process is repeated recursively for each quadrant n times, until the single pixel level is reached, as shown in Figure 4-2. The root of the quadtree corresponds to the whole image, the leaves correspond to the single pixels, and the nodes of the tree correspond to quadrants of the square represented by their parent node. If the four children of a node share the same value, they are all deleted and the father's value represents them all.

Quadtrees are used mainly to encode binary images, and the nodes in a quadtree are interpreted differently from the nodes in a multi-resolution pyramid tree. In the case of binary images, nodes of the quadtree can take one of three values. If the node's children are all black, then the node is black. If they are all white, then the node is white. The node will take the value gray if its children do not have the same value, or if

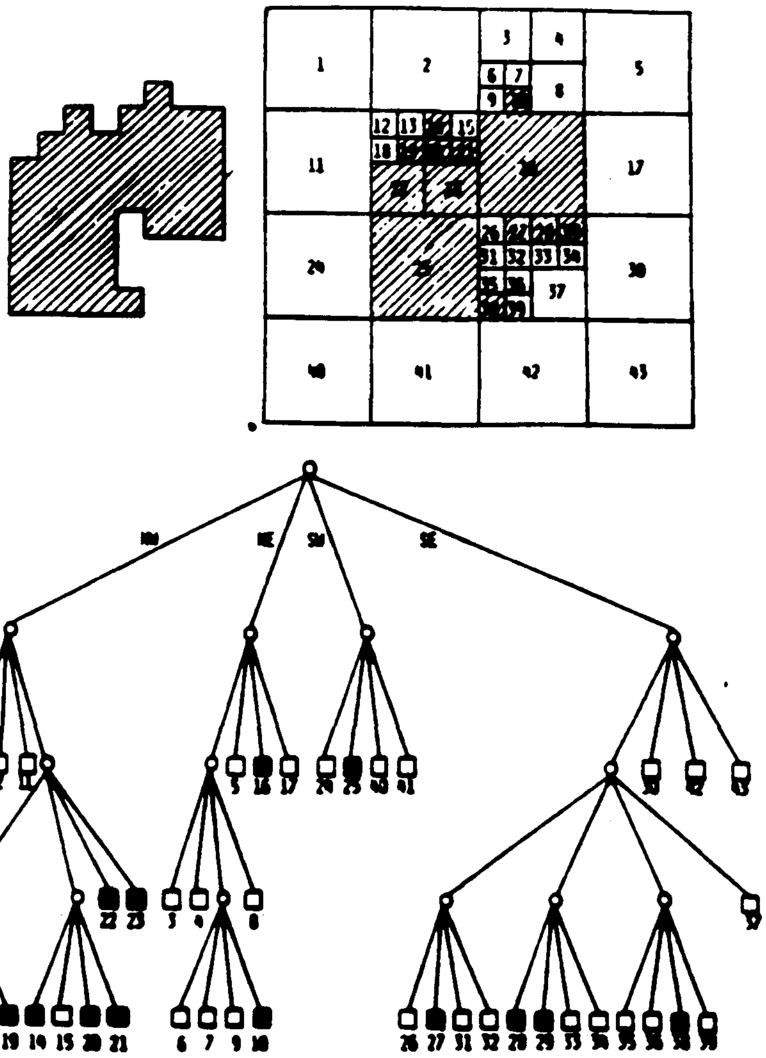


Figure 4-2: A Picture and its Quadtree (From [Same 82])

they all have the value gray All subtrees rooted with either a white or a black node may thus be omitted, significantly reducing the amount of memory required to store the picture on a sequential machine

Binary image trees are a variant of quadtrees in which the whole image is subdivided into two halves. This process of subdividing into two halves is repeated recursively until the single pixel level is reached Figure 4-3 illustrates the binary image tree data structure Note that the shape of the subdivisions changes from level to level in the binary image tree Specifically, it is either a square or a rectangle with the width equal to twice the length. We will refer to these subdivisions as rectangles throughout the rest of this thesis.

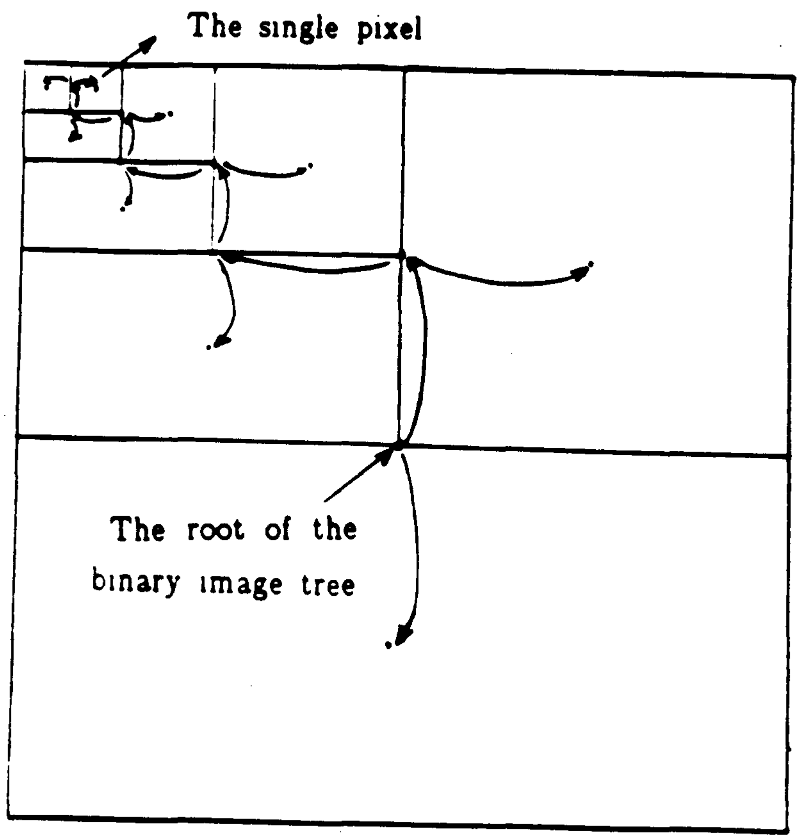


Figure 4-3: Binary Image Tree Rectangle Arrangement

The shape of the rectangles at any level can be determined by testing to see if the level number is odd or even. Going from one level to the next level down the tree increases the resolution by only a factor of two, while in quadtrees the resolution is increased by a factor of four.

Binary image trees are mapped naturally onto binary tree machines. On NON-VON, the leaf processors are used to store image information at the single pixel level, while non-leaf PE's correspond to rectangles of size larger than the pixel size. A record associated with each PE is used to store information about the location, size, contents, and adjacency relation of the part of the image it represents. A flag associated with each PE indicates whether the rectangle represented by that PE is part of the binary image tree. An algorithm for building the binary image tree and storing the rectangle information is described later in this chapter. We will use binary image trees whenever we are dealing with binary images.

4.2. Initialization and Image Loading

Image processing algorithms on NON-VON use information that is stored initially in each PE. Each PE corresponds to a rectangle in the original image. On the leaf PE level, each rectangle corresponds to a single pixel in the two-dimensional array that represents the original image. The location of each rectangle is indicated by specifying the coordinates of its upper left-most corner pixel. The horizontal direction is referred to as the x-direction, while the vertical direction is referred to as the y-direction. The origin of the coordinate system (0,0) is the upper left-most pixel in the image, and the values of the coordinates increase to the right in the x-direction, and down in the y-direction, as shown in

Figure 4-3

In addition to the x-address and y-address, each PE stores the width (x-side), and the height (y-side) of the rectangle it represents. For a 256 x 256 image, four bytes are needed to store the location and size information. The root level is labeled the **0-*th*** level, while the leaf level is the ***n-th*** level. Other information is also stored in each PE, and will be described later.

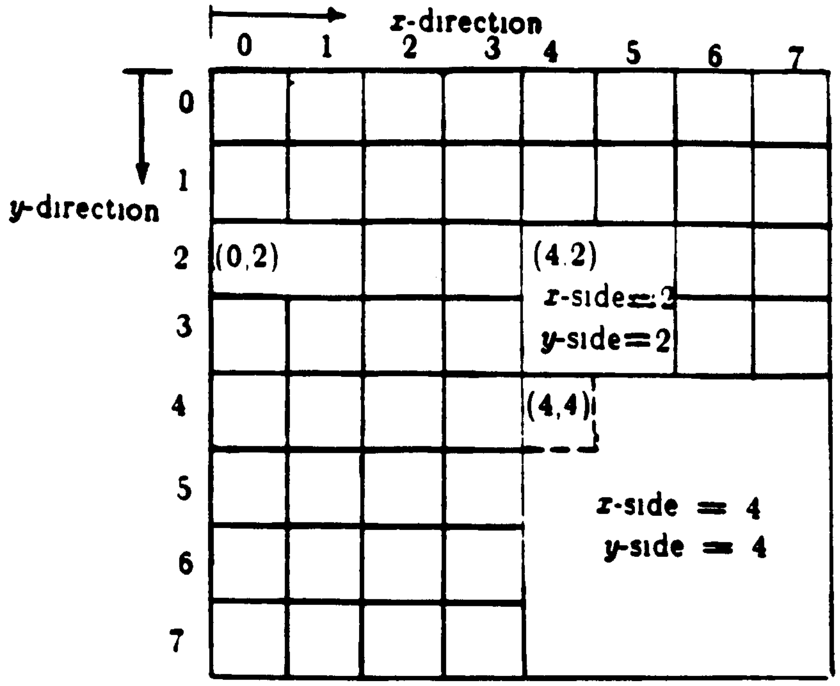


Figure 4-4: Coordinate System for Binary Image Trees

The N-PASCAL algorithm for initializing the NON-VON tree follows:

/* The following variables are defined in the main driver of all vision manner. This has led to a surge in research aimed at developing new computer organizations that meet the large computational and decision requirements of image analysis tasks by exploiting the new technology. Various kinds of special parallel machines for computer vision have been proposed and some have been implemented; examples are described in [Duff 76], [Krus 76], [Dyer 81], [Kush 83], [Pott 83], and [Reev 84].

The organization of some of the proposed machines is based on a very large number of very small ***processing elements*** (PE's). Throughout this thesis, we will refer to such machines as ***fine-grained*** or ***highly parallel*** machines. In such machines, different schemes are used to interconnect the PE's. For example, the PE's can be connected together in the form of a two-dimensional mesh, or they can be placed at the nodes of a binary tree. If all the PE's simultaneously execute the same instruction on their own data, the machine is said to be executing in ***single instruction stream, multiple data stream*** (SIMD) mode [Flyn 72]. On the other hand, if the PE's execute different instruction streams concurrently on different data streams, then the machine is said to be executing in ***multiple instruction stream, multiple data stream*** (MIMD) mode.

In this thesis, we investigate how fine-grained tree-structured SIMD computer architectures, which have favorable characteristics for efficient VLSI implementation, can be used for the rapid execution of a wide range of vision tasks. We also discuss certain limitations of these procedures described in this thesis. Some of these variables are used in the initialization procedure. The rest are used in other procedure definitions. */

```
program vision_algorithms();
vector_var
    XSIDE, YSIDE, XADD, YADD: integer;
    GRAY_VALUE, TREE, COMP_LABEL: integer;
    FQUAD: char;
    BINARY: boolean;
```

/* The following procedure marks leaf PE's by setting the variable LEAF equal to 1 in all leaf PE's and 0 elsewhere. */

```
Procedure mark_leaf(var  LEAF: boolean);
vector_var
    TEMP: boolean;
begin
```

/* 1 Initialize TEMP to the value 0 in all PE's. On executing a "receive from left child" instruction (RECV1 LC), all leaf PE's receive a logical 1. All other PE's receive whatever is sent by their left children. This procedure thus serves to mark all leaf PE's. The N_RECV1 is a primitive function that corresponds to the NON-VON instruction RECV1. On executing this function each PE receives the value of its left child boolean variable TEMP into its own variable LEAF. */

```
  TEMP  = false,
  N_RECV1(LC, TEMP, LEAF),
end,

Procedure tree_init(no_levels INTEGER),
var
   lev_count, x, y  integer;
vector_var
   XADD1, YADD1  integer,
    LEAF  N  boolean,
begin
```

/* 1 The initialization algorithm starts by initializing the size information It starts at the leaf level by storing the value 1 in each PE width and length variables (XSIDE, YSIDE) The function mark_leaf(L) sets the boolean variable L to 1 only in leaf PE's The boolean variable N is used to mark the level up the tree next to the current level. It is set equal to 0 only in current level PE's, and to 1 elsewhere. A level counter *lev_count* is initialized to 0. */

```
lev_count = 0,
x = 1, y = 1,
N = true,
mark_leaf(LEAF),
where LEAF = true do
  begin
    XSIDE = 1,
    YSIDE = 1,
    N     = false,
  end,
```

/* 2 Enable PE's on the next level up the tree. N_RECV1(LC, N, N) sets the variable N equal to 0 only in PE's whose children have the N variable set to 0 The size variables are then computed on that level If the level number is equal to *no_levels*, then the size initialization is complete and the algorithm starts address initialization, otherwise the size initialization continues */

```
while lev_count < no_levels do
  begin
    N_RECV1(LC, N, N);
    if lev_count mod 2 = 1 then
      x = x * 2
     else y = y * 2,
    where N = false do
      begin
        XSIDE = x,
        YSIDE = y;
      end,
    lev_count = lev_count + 1,
  end.
```

/* 3 At this point only the root PE has its N variable set to 0 Set the variables XADD and YADD in the parent PE equal to 0. */

```
  where N = false do
    begin
      XADD  = 0;
      YADD  = 0;
    end;
```

/* 4. For address information, left children always have the address of their parents. This step computes the right child address of each enabled parent and stores the computed values in the variables XADD1 and YADD1. Left children are then enabled. Read the address of their parent and store it as their own address. The same is repeated for right children, except that they read their address from the variables XADD1, and YADD1. */

```
  while lev_count > 0 do
    begin
      if lev_count mod 2 = 0 then
        begin
          XADD1  = XADD;
          YADD1  = YADD + YSIDE div 2;
        end
       else
         begin
           XADD1  = XADD + XSIDE div 2;
           YADD1  = YADD;
         end,

      N_SEND8(LC, XADD, XADD),
      N_SEND8(LC, YADD, YADD),
      N_SEND8(RC, XADD1, XADD),
      N_SEND8(RC, YADD1, YADD),

      lev_count  = lev_count - 1,
    end;
end.
```

The initialization algorithm takes time proportional to the number of levels in the tree (19 levels in the case of a 512 X 512 image). The

NON-VON 3 code for this procedure is presented in Appendix B. It takes 18 μsec to initialize one level at 4 Mhz (68 NON-VON 3 instructions). Initializing a tree with 15 levels thus requires 0.27 msec.

4.2.1. Loading the Image

In tree machines, loading and unloading the tree through the root can be a bottleneck for algorithms with extensive I/O operations. To overcome this, a real NON-VON system would load and unload image data in parallel through I/O devices connected to all PE's at some intermediate level in the PPS tree. Loading an image point through the root only involves first broadcasting its *x*- and *y*-coordinates and enabling the PE with the same values for *x* and *y* on the leaf level. The image point value is then broadcast and stored in the enabled PE. The N-PASCAL procedure for this loading procedure follows:

```
Procedure tree_load1(x_side, y_side: integer);
var
   i, j: integer;
begin

  /* 1. The function read_file(file-name) returns the next integer value in
the file "file-name". The procedure arguments are the lengths of the
image sides. */

  for i := 0 to x_side-1 do
     for j := 0 to y_side-1 do
        where ((XADD = i) and (YADD = j))
           do GRAY_VALUE := read_file(image);
end;
```

The NON-VON 3 code for this loading procedure is included in Appendix B. Seven NON-VON 3 instructions, requiring about 2.0 μsec of execution time at 4 Mhz, are used to load one image point. Loading an 128 x 128 image through the root thus takes about 32 msec using this procedure.

Instead of broadcasting the data byte by byte and isolating a single destination leaf PE at a time, blocks of image data can be broadcast and stored in the PE's in an intermediate level. These PE's then load the blocks of data in parallel into the leaf PE's in their subtrees. Next we describe the N-PASCAL procedure to perform this operation along with an analysis of the time required for its execution.

```
Procedure tree_load2(x_side, y-side: integer);
var
   i, j, k, n1: integer;
vector_var
   LEAF, N: boolean;
   X1, Y1, TEMP: integer;
begin

 /* 1. The leaf PE's are marked, and so are the PE's on the
intermediate level. We assume a block size of 16 bytes */

  N := false;
  mark_leaf(LEAF);
  where ((XSIDE = 4) and (YSIDE = 4)) do N := true;

 /* 2. Loop to load the blocks in intermediate level PE's starting at
RAM location 16. */
```

```
i = 0;
while  i < x_side-1  do
  begin
    j = 0;
    while j < y_side-1 do
      begin
        where ((XADD = i) and (YADD = j) and (N = true))
          do for k = 1 to 16
            do N_BROADCAST8(read_file(image), RAM(15+k));
        j  = j + 4;
      end;
    i  = i + 4;
  end;
```

/* 3 Now load the blocks in parallel into the leaf PE's. The addresses of the PE's relative to the address of the root of the subtree are stored in X1 and Y1. */

```
X1  = XADD mod 4;
Y1  = YADD mod 4;
n1  = 16;
```

/* Loop until the first block element reaches the leaf PE's. */

```
for k = 1 to 4 do
  begin
    where N = true do N_READRAM8(n1, TEMP);
    N_RECV8(P, TEMP, TEMP);
    n1  = n1 + 1;
  end;
```

/* Now loop to load the elements */

```
for i = 0 to 3 do
  for j = 0 to 3 do
    begin
      where ((i = X1) and (j = Y1)) do
         GRAY_VALUE  = TEMP;
      if n1 < 32 then
        where N = true do N_READRAM8(n1, TEMP);
      N_RECV8(P, TEMP, TEMP);
      n1  = n1 + 1;
```

```
        end;
    end;
```

In the above algorithm, Step 2 is similar to the first loading procedure. It takes about 2 μsec to load the first byte, but the next 15 bytes in the block are loaded at the rate of 4 bytes per μsec. Thus, it takes about 6 μsec to load a block of 16 bytes, and about 6.14 msec to load a 128 × 128 gray-scale image. Step 3 requires about 10 NON-VON 3 instructions to load one byte of the block into a leaf PE. Thus, about 0.04 msec is needed to perform this step, which is very small compared with the time required to execute Step 3.

It should be noted that increasing the block size reduces the time required to perform the first step of the second loading procedure. For example, a block size of 32 requires 10 μsec to be loaded into the intermediate level, and hence 5.12 msec are required to load a 128 × 128 gray-scale image. Increasing the block size also increases the time required to load the blocks in the subtrees. Using the numbers cited above, the time to load an image of size A using block size S can be computed by the following expression

$$\textbf{\textit{loading time}} = (A/S)(2 + S/4) + 2.5S \quad \mu\text{sec}.$$

The value of S that minimizes this expression is equal to $0.9A^{1/2}$. Thus, a block size of about 115 bytes results in the minimum loading time for an image of size 128 × 128. If the size of the available memory in each

PE is less than this number, then using as much memory as we can for this loading procedure results in the minimum loading time The NON-VON 3 code for this procedure is presented in Appendix B

The time required to load images could be reduced significantly through parallel loading of the subtrees rooted by the PE's at some intermediate level If the I/O devices are connected to the intermediate level containing 64 PE's, then the above described procedures would be executed in parallel in the 64 subtrees, and the time of execution would be reduced approximately by a factor of 64. If more than one image are to be stored in the tree, then the address and size information stored in each PE will be common to these images Other image information has to be duplicated for each loaded image

Table 4 1 provides a summary of the execution times for different NON-VON I/O procedures and for some existing parallel image processing machines

4.2.2. Building the Binary Image Tree

In this subsection, we describe the procedure for constructing the binary image tree representation of a binary image stored in the leaf PE's The vector character variable FQUAD is used in each PE to indicate the type of rectangle held by this PE The value 'B' refers to black rectangles, 'W' to white ones, and 'G' to gray rectangles The value 'N' indicates

Table 4-1: Image I/O Execution Time for Some Parallel Machines

The Parallel Machine	Instruction Rate	I/O time (msec) (128 × 128)
ICL DAP	4 Mhz	4.096
Goodyear aerospace MPP	10 Mhz	0.102
NON-VON 3 --		
a. Loading through the root only	4 Mhz	5.120
b. Loading through I/O device connected to the 64 PE level.	4 Mhz	0.080

that the rectangle is white or black but has been merged with a similar rectangle to form a larger one. The vector integer variable TREE corresponds to the number of black pixels in the rectangle represented by the PE. It takes the value 0 when the rectangle is white, and is equal to the area of the rectangle in the case of black rectangles. We assume that the binary image is stored in the NON-VON tree in the vector variable BINARY. The N-PASCAL algorithm for building the binary image tree follows:

```
Procedure build_binimg(no_levels INTEGER),
  label 2;
var
  cur_lev integer;
vector_var
  GV1, GV2 integer,
  FQ1, FQ2 char;
  LEAF, N boolean,
begin
```

/* 1 Enable all PE's on the leaf level. Set N equal to 0 only in the current level. */

```
N = true,
cur_lev = 1,
mark_leaf(LEAF),
where LEAF = true do
  begin
    where BINARY = true do TREE = 1
      elsewhere TREE = 0;
    N = false,
  end,
```

/* 2 Mark PE's on the level just above cur_lev. Let all the enabled PE's read the values of TREE and FQUAD in their children. The value of TREE in the enabled PE's will be set equal to the sum of the two TREE variables in their children. FQUAD will be set to 'G' if FQUAD in the two children are different, or if one of them is 'G'. If the two variables FQUAD in the two children are both either 'W' or 'B', then the parent FQUAD will be set to the mutual value and the FQUAD in the two children will be set to 'N'. */

```
2
N_RECV1(LC, N, N),
where N = false do
  begin
    N_RECV8(LC, TREE, GV1);
    N_RECV8(LC, FQUAD, FQ1);
    N_RECV8(RC, TREE, GV2),
    N_RECV8(RC, FQUAD, FQ2),
    TREE = GV1 + GV2 ,
    if (((FQ1 = 'B') and (FQ2 = 'B')) or
            ((FQ1 = 'W)' and (FQ2 = 'W')))
     then FQUAD = FQ1
     else FQUAD = 'G' ,
  end,

N_RECV8(P, FQ1, FQUAD),
if (FQ1 <> 'G') then FQUAD = 'N',
```

/* 3 If the root is reached, stop, otherwise, enable all PE's above and

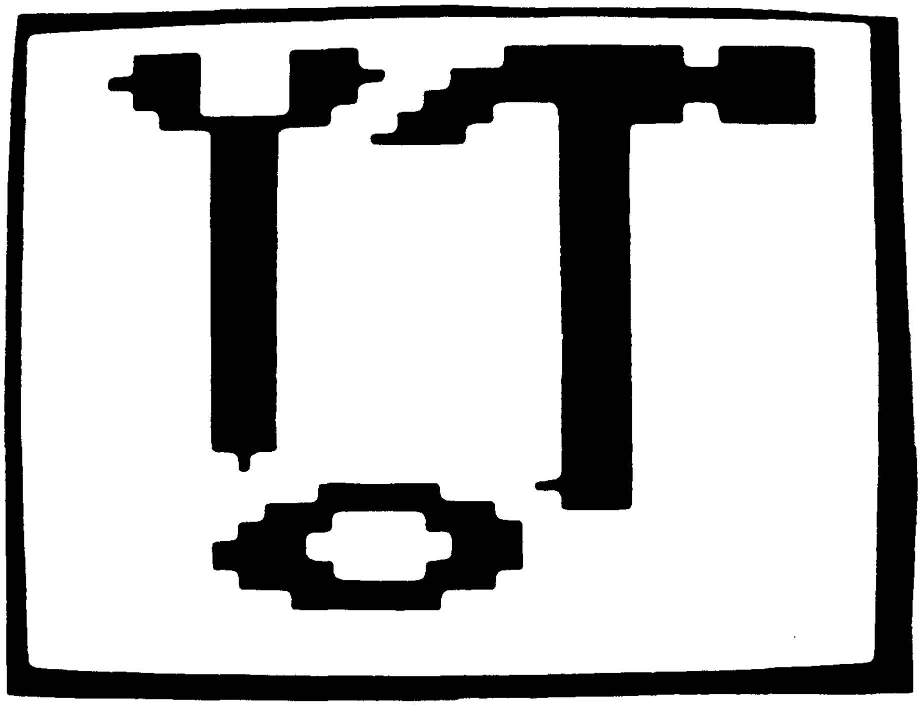
```
go to step two. */
  if cur_lev <> no_levels then
    begin
      cur_lev := cur_lev + 1;
      goto 2;
    end;
end;
```

After the above algorithm is executed, the root PE has its TREE variable set equal to the number of black pixels in the whole image, and in general, each PE's TREE variable will be equal to the number of black pixels in the tree rooted by that PE. Steps 2 and 3 are repeated a number of times equal to the number of levels in the tree. Thus, the algorithm takes time proportional to the height of the binary tree. The NON-VON 3 code is included in Appendix B. It requires about 50 NON-VON 3 instructions per level (12.5 μsec); for a tree with 15 levels (corresponding to a 128 $\times$ 128 original image), the execution time for this procedure is thus about 0.175 msec. Figure 4-5 shows a binary image and the binary image tree representation of it as output by the functional simulator.

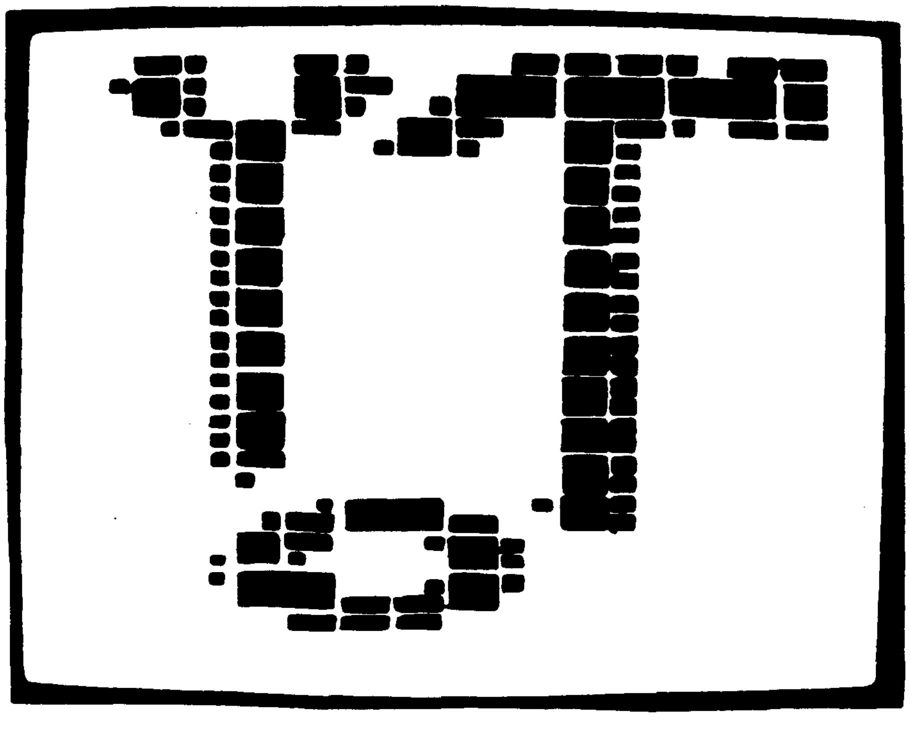
4.2.3. Building the Multi-Resolution Pyramid

The multi-resolution pyramid representation of a gray-scale image can be built using a procedure similar to the one used to build the binary image tree. The variable GRAY_VALUE in each PE is set equal to the average of the values of GRAY_VALUE in its two children. This step is repeated a number of times equal to the height of the tree. The

Figure 4-5: A Binary Image and its Binary Image Tree Representation



(a) The Input Binary Image



(b) The Binary Image Tree

roundoff errors due to the averaging process from one level to the next up the tree can accumulate, resulting in large errors in the computed average values. To solve this problem, the averaging should take place only after all the sums have been computed in all levels. Then each PE divides the sum by the rectangle size it corresponds to. Because the rectangle sizes are powers of two, the division is equivalent to logical shift operations. If we start with gray level values that are 8 bits long, the sum at the root of 15 levels tree could be 23 bits long. The add operations should be therefore 24 bits long (three bytes). The N-PASCAL algorithm to build the multi-resolution pyramid follows:

```
Procedure build_multi_resol(no_levels: INTEGER);
  label 2;
var
  cur_lev: integer;
vector_var
  GV1, GV2: integer;
  LEAF, N: boolean;
begin

 /* 1. Enable PE's on the leaf level. Set N equal to 0 only in the
current level */

  N := true;
  cur_lev := 1;
  mark_leaf(LEAF);
  where LEAF = true do
    begin
      GV1 := GRAY_VALUE;
      N := false;
    end;

 /* 2. Mark PE's on the next level above cur_lev. Let all the enabled
PE's read the values of GV1 in their children. The value of GV1 in
each enabled PE's will be set equal to the sum of the variables GV1 in
```

```
its two children */

  2
  N_RECV1(LC, N, N);
  where N = false do
    begin
      N_RECV8(LC, GV1, GV1);
      N_RECV8(RC, GV2, GV1);
      GV1 = GV1 + GV2;
    end;

 /* 3 If the root is reached, stop, otherwise enable all PE's in the next
level above and go to step two */

  if cur_lev <> no_levels then
     begin
       cur_lev = cur_lev + 1;
       goto 2;
     end;

 /* 4 Compute the average value in each PE */

  GRAY_VALUE = GV1 div (XSIDE * YSIDE);

end;
```

The time analysis of this procedure is similar to that of the algorithm for constructing the binary image tree. The algorithm executes in time proportional to the height of the tree. Each step consists of approximately 30 NON-VON 3 instructions (approximately 0.120 msec to build the multi-resolution pyramid in a 15 level tree). The multi-resolution pyramid representation of binary images can be computed using a similar procedure, with the exception that the computed value in Step

4 is set equal to 1 iff the sum is larger than half the rectangle size, otherwise the computed value is set equal to 0.

Chapter 5
Low-Level Image Processing Algorithms

In this chapter, we describe the implementation on NON-VON of some low-level image understanding algorithms (also referred to as signal level image processing algorithms). In low-level image processing, the input is typically the original image input by some sensory device or the output from some other low-level operations. The output is usually of the same size as the input. Some examples include image restoration, image enhancement, and noise removal. Other low-level operations extract from the input image such physical characteristics as color, surface orientation, range, velocity, and edges. The output images in this case are called *intrinsic images* [Ball 82]. The low-level operations described in this chapter are the gray level image histogram computation, image segmentation by thresholding, and image correlation. The selected tasks are representative of a large class of low-level image understanding algorithms. For our time analysis, we assume that the image has already been loaded in the NON-VON leaf PE's, as described in the previous chapter.

5.1. Image Histogramming

A gray level histogram of a gray-scale image is a function that gives the frequency of occurrence in the image of each possible gray level. The gray level at each image point is quantized from 0 to m (typically m is equal to 255). The value of the histogram at a specific gray level p is the number of image points with gray level value equal to p. The histogram of an image can be useful in many ways. It can be used to select a threshold value (or values) for segmenting an image into a foreground-background image, or it can be used to guide the filtering of an image [Ball 82]. Other applications include image enhancement and image encoding [Palv 82]. Sometime, it is desirable to compute the gray level histogram, not for each possible gray value, but for non-overlapping ranges of gray values. In the latter case, the range of gray level values is usually divided into equal intervals called the ***histogram bins***. The interval range is referred to as the ***bin width***. The histogram value for a certain histogram bin is the number of pixels in the image with gray level intensity within the bin range.

We now describe a simple algorithm to compute a histogram of a gray-scale image stored in the leaf PE's of the NON-VON tree. We assume that the image has n pixels ($n^{1/2}$ on a side), and that the whole image can fit in the leaf PE's of the NON-VON tree. Also, we assume that the histogram to be computed contains b bins. For each histogram bin, the algorithm first marks all leaf PE's corresponding to image pixels with gray level value falling in this bin range. A vector variable HIST_SUM is then set equal to 1 in the marked leaf PE's and 0 elsewhere. Note that the marking operation is performed concurrently in all leaf PE's, and therefore requires a fixed number of instructions for execution. The marked PE's are then counted using the tree connections. The counting operation consists of h steps, where h is the height of the tree. In each counting step, each parent node in the tree sets the value of its vector variable HIST_SUM equal to the sum of the same vector variable in its two children. Thus, the counting operation executes in a time proportional to the NON-VON tree height (logarithmic in the number of PE's). This simple algorithm thus executes in time proportional to the product of NON-VON tree height and the number of histogram bins ($O(b \log n)$). The computed histogram values can be stored in the CP, or can be stored in the NON-VON tree for further processing. It should be noted that during the counting operations only PE's on a certain level are performing a useful work at any specific time. Actually, the marking and counting steps described above can be interleaved, resulting in a more efficient algorithm.

In general terms, the new algorithm involves the repetition, a number of times equal the number of bins in the histogram, of a sequence of a marking operation followed by a counting operation step. Then the count operation steps are repeated a number of times equal to the number of tree levels. Before a formal description of this SIMD-pipelined algorithm and its time analysis are given, we describe the variables used in our algorithm. The vector variable GRAY_VALUE in leaf PE's is used to hold the gray level intensities of the original image. The vector variable HIST_SUM in all PE's is used to store the partial sums resulting from counting the marked PE's for a specific bin in the histogram. The vector variables HIST_VAL and BIN_VAL are used to store the values of the histogram and the corresponding bin values in the NON-VON tree. The scalar variable ***num_s*** stores the number of computed histogram values which have been reported to the CP. The scalar variable ***count*** keeps track of how many histogram bin values have been broadcast (the number of mark operations performed), while the scalar variable ***value*** contains the minimum value in the bin range to be broadcast. The scalar variables ***nbins*** and ***bwid*** denote the number of histogram bins to be computed and the bin-width, respectively. The variable ***bwid*** is computed by dividing the histogram range by the number of bins. The N-PASCAL procedure follows:

```
/* The following procedure adds to the integer variable Z in each PE
the value of the two variables X and Y in its two children. */

Procedure add_chtp(var  X, Y, Z: Integer);
vector_var
    TEMP Integer;
begin

/* 1. read the value of left child variable X into TEMP.  Add this
```

```
value to the value of the vector variable Z, and store the sum in Z. */

  N_RECV8(LC, X, TEMP);
  N_ADD(Z, TEMP, Z);

 /* 2. Repeat step 1 for the right child and Y instead of X. */

  N_RECV8(RC, Y, TEMP);
  N_ADD(Z, TEMP, Z);
end;


Procedure gray_level_histogram(h: integer);
    label 2, 4, 7;
var
    value, count, num_s, nbins, bwid: integer;
vector_var
    HIST_VAL, BIN_VAL: integer;
    HIST_SUM: integer;
    LEAF: boolean;
begin

 /* 1. Initialize the scalar variables value, count, and num_s in the
CP, and the vector variables BIN_VAL, and HIST_VAL in all PE's. */

  value := 0; count := 0; num_s := 0;
  BIN_VAL := -1;
  HIST_VAL := -1;

 /* 2. Set the vector variable HIST_SUM equal to zero in all leaf PE's.
The procedure mark_leaf sets the vector variable LEAF equal to 1 only
in leaf PE's. */

  2 mark_leaf(LEAF);
  where LEAF = true do HIST_SUM := 0;

 /* 3. This is the marking step. Enable only the leaf PE's with gray
```

level value falling within the current bin range. Set the vector variables HIST_SUM equal to 1 in the enabled PE's. */

```
where ((LEAF = true) and (value <= GRAY_VALUE)
          and (GRAY_VALUE < value+bwid))
  do  HIST_SUM := 1;
```

/* 4. This is a counting operation step. It is performed by setting the variables HIST_SUM in each PE equal to the sum of the HIST_SUM variables in its two children. The function add_chtp(*x*,*y*,*z*) adds the two variables *x* , *y* in LC and RC respectively and stores the sum in the variable *z* of their parent node. */

```
4: add_chtp(HIST_SUM, HIST_SUM, HIST_SUM);
```

/* 5 If all the marking operations have been performed (*nbins* of them) and the number of counting steps is larger than the tree height, then skip the next step which computes the new bin range. */

```
if ((count = nbins) and (count >= h))
  then goto 7,
```

/* 6 Increment *count* by one, and compute the new bin range by incrementing *value* by *bwid*. If the number of count steps is less than both the tree height and the number of bins, then perform another marking operation. If the first histogram value has not arrived to the root, then perform another count step; otherwise read a histogram value from the tree root. */

```
count = count + 1,
value = value + bwid;
if ((count < h) and (count < nbins))
  then goto 2;
if ((count < h) and (count >= nbins) )
  then goto 4;
```

/* 7 Read a histogram value. this is performed by reading the value HIST_SUM in the root of the NON-VON tree. Store the reported value back in the NON-VON tree. Select a PE (HIST_VAL equal to -1) to store the histogram value. */

```
  7: N_RECV8(LC, HIST_SUM, N_GG8);
  where HIST_VAL = -1 do N_A1 := true
   elsewhere N_A1 := false;
  N_RESOLVE();
  where N_A1 = true do
    N_BROADCAST8(N_GG8 , HIST_VAL);
```

/* 8. Increment the number of stored histogram values (***num_s***) by one. If all histogram values have already been stored, then stop. If not, then check to see whether to perform another counting step, or a marking operation. */

```
  num_s := num_s + 1;
  if num_s <> nbins then
    begin
      if count = nbins then goto 2
        else goto 4;
    end;
end;
```

Steps 2, 3 (constituting the match operation), 5, and 6 are executed a number of times equal to ***b***, while step 4 (the counting operation step) is executed ***h+b*** times, where ***h*** is the height of the tree. Therefore the time required to execute the histogramming algorithm is of $O(b+h)$ (or in terms of the image size ***n***, $O(b+\log n)$). Appendix B contains the NON-VON 3 code for the above algorithm. For a 128 x 128 NON-VON 3 machine and a histogram with 64 bins, the algorithm executes in approximately one msec. By way of comparison, 120 msec is required on a 128 x 128 MPP machine [Pott 83], and 17.5 msec on a 32 x 32 DAP machine [Mark 80]. The numbers given above are for

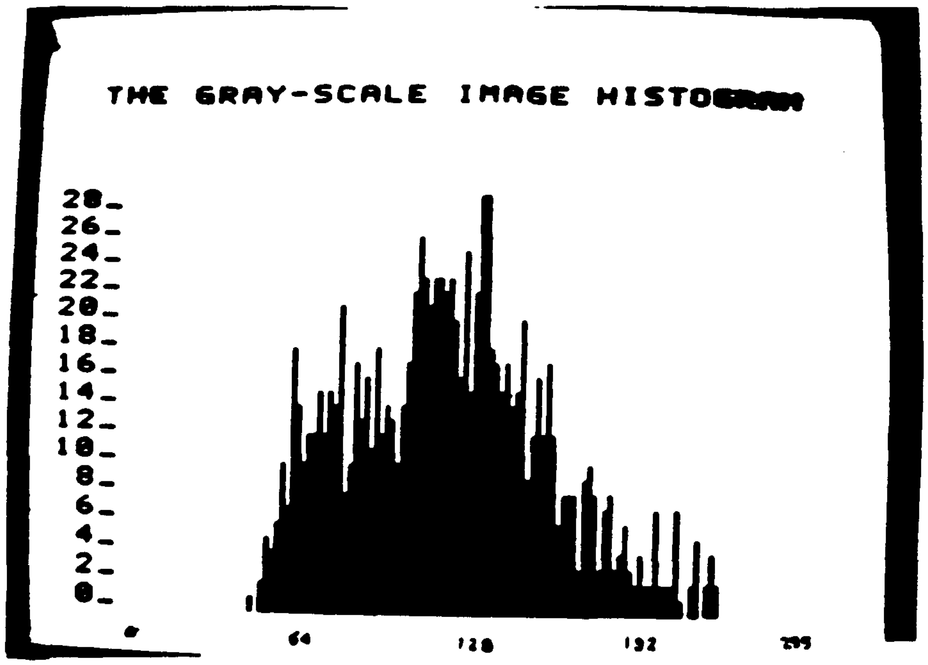


Figure 5-1: The Gray-Scale Image Histogram

gray-scale images of the same size as the machine size. If the image is larger than the NON-VON tree, then each leaf PE would hold more than one image point. If each leaf PE holds k image points, then the time needed to execute the histogram increases approximately by a factor of k. Figure 5-1 shows a 128-bin histogram of a 32 × 32 gray-scale image, as computed by the functional simulator using the algorithm described in this section.

The histogram algorithm described in this section can be easily adapted to compute different variants of the image histogram. For example, computing the histogram of a subimage involves changing Step 2 to enable the subset of leaf PE's in the subimage instead of all the leaf PE's. A ***cumulative histogram*** of a gray-scale image is a function that gives for each gray level p the number of pixels that have gray level values less than or equal to the value p. To compute a cumulative histogram, we create a new scalar variable in the CP, initialize it to zero, and add to it the histogram values as they are being reported to the CP. The accumulated values are then stored in the NON-VON tree. A ***normalized histogram*** can be computed from the accumulated histogram by dividing its values by the number of pixels in the image.

5.2. Thresholding

Thresholding is one technique that is used for ***image segmentation*** in image understanding applications. Image segmentation is concerned with identifying areas of the image that are homogeneous with respect to one or more characteristic. Examples of such characteristics include intensity, continuity, and range. One approach to image segmentation separates image "objects" from the "background". The resulting image is referred to as an object-background image. For a gray-scale image, for example, this technique picks a threshold value from the image histogram and uses that value to divide the set of image points into object points background points [Ball 82]. The object points are those pixels which have a gray level value exceeding the threshold value, all other pixels are background pixels. There are many techniques for selecting a threshold value [Cast 79]. The choice of a certain technique depends on the nature of the image under consideration. Assume, for example, that the objects pixels are predominantly dark, while the background pixels are light. The histogram of such an image might have two peaks , corresponding to the dark and light regions.

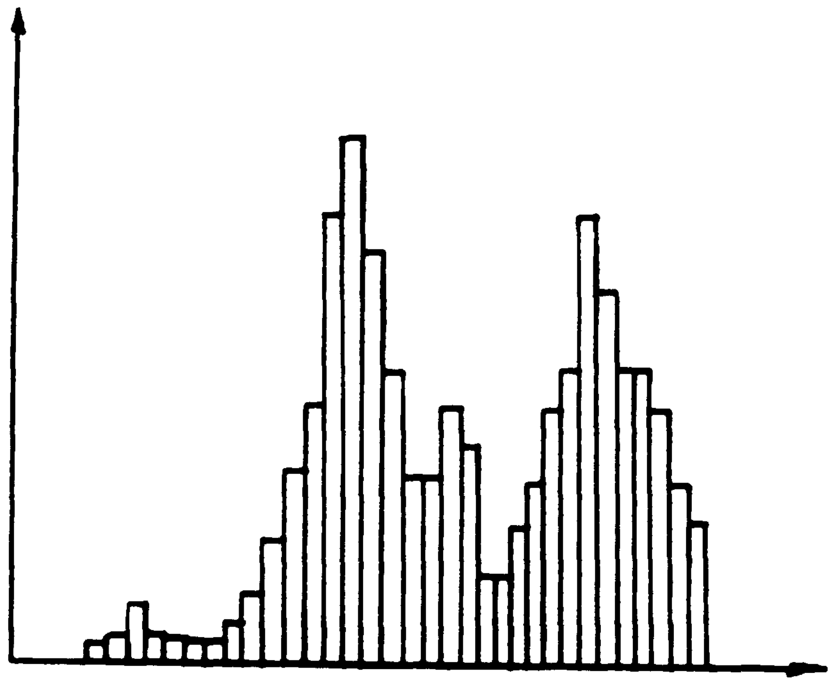


Figure 5-2: A Bimodal Histogram

Such a histogram (Figure 5-2) is called a ***bimodal histogram***. One way to pick a threshold value is to search the histogram and find a minimum separating the two peaks.

The N-PASCAL algorithm for segmenting the image into objects and background based on a single threshold value follows:

```
Procedure seg_by_thresholding(thr: integer);
begin
 /* 1. The threshold value thr is compared with the variable
 GRAY_VALUE, which holds the gray level value. If GRAY_VALUE is
 larger than or equal to thr, then the globally defined local one-bit
 variable BINARY is set to 1 and the point is an object point; otherwise
 the point is a background point and BINARY is set to 0. */
   where GRAY_VALUE >= thr do BINARY := true
     elsewhere BINARY := false;
end;
```

The algorithm executes a fixed number of instructions, independent of the number of pixels in the image. The NON-VON 3 code for this algorithm is provided in Appendix B (6 NON-VON 3 instructions). The time required to execute the algorithm is 1.5 μsec. Figure 5-3 shows the binary image resulting from thresholding the gray-scale image whose histogram is shown in Figure 5-1 using the gray intensity value of 80.

Image segmentation based on a single threshold value is useful only in simple situations [Ball 82]. For example, a common problem with the single threshold method occurs when the image has a background of varying gray levels. A spatially varying threshold can be used to segment the image in such a case [Ball 82]. In this method, the image is divided into subimages and a threshold is computed for each subimage based on the histogram of this subimage as described earlier. These subimages typically correspond to separate subtrees. The entire image is then segmented by segmenting each subimage using its own computed

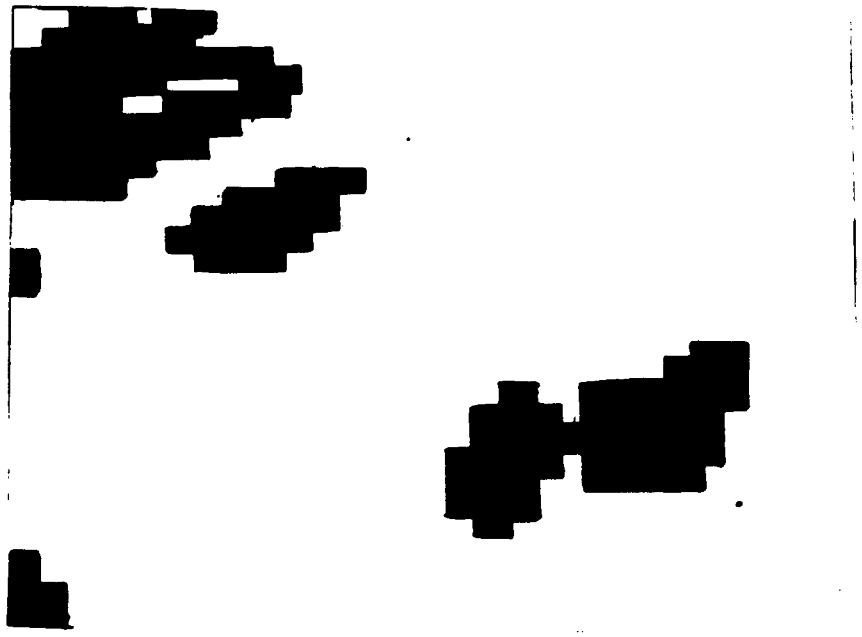


Figure 5-3: The Binary Image After Thresholding

threshold value. The threshold value computed for a subimage is stored in the root of the subtree corresponding to this subimage. Thresholding can then be performed by broadcasting the separate threshold values simultaneously from the roots of all subtrees to all the PE's in their respective subtrees. Step 2 of the thresholding algorithm described in this section is then executed. This approach can be thought of as a MSIMD approach with each subtree representing an image for which thresholding is applied. Other segmentation methods based on thresholding, such as hierarchical refinement (recursive region splitting), can be performed in a similar manner [Ball 82].

5.3. Image Correlation

Correlation techniques are widely used in many image understanding tasks, including simple filtering to detect a particular feature in an image, edge detection, image registration, motion and stereo analysis, and object detection by template matching [Ball 82]. Image correlation involves determining the position at which a relatively small ***template*** image best matches the input image. The ***correlation function*** reflects how well the image data match the template image for each possible template location. Image correlation is a representative of a wider class of image operations known as ***local operations*** (also referred to as ***window-based operations***). In local operations, the output value at a specific point is a function of the image values at this point and at a number of points in its immediate neighborhood. Techniques and algorithms developed in this section to compute image correlation are applicable to many other local operations.

In what follows, we present several numerical measures of the correlation function. Let us assume an image array X and a template array Y, with x and y representing the elements of X and Y respectively. One correlation measure is the ***Euclidean distance***, d, defined for each possible relative location of the input image and the template as follows:

$$d^2 = \sum (x - y)^2 \tag{5.1}$$

The value of d is zero for an exact match. There are other correlation measures which are variations of this basic measure. One of these measures is the covariance of the template with a portion of the input area, which is defined as follows [Sieg 81a]:

$$S_{xy} = \sum xy - (\sum x \sum y)/A \tag{5.2}$$

where A is the area of the template. Large positive covariance values indicate similarity between the image and the template, while large negative values indicate similarity between a positive and a negative image. Values near zero indicate no similarity. Another correlation measure is a normalized version of S_{xy} and is defined as [Sieg 81a]:

$$R_{xy} = S_{xy} \; / \; (S_{xx} \cdot S_{yy})^{1/2} \tag{5.3}$$

One way to visualize the computation of the correlation function is to imagine a template scanning the image at all possible offsets, computing the correlation at each offset, and storing these correlation values for later computations. On a sequential machine, the time required to execute such a function is $O(nm)$, where n is the number of pixels in the image and m is the number of pixels in the template.

A basic operation that is performed repeatedly in the parallel image correlation algorithms described in this section is the ***image shift*** operation In this operation, the whole image stored at the leaf PE's is shifted one or more positions in the right, left, up, or down direction. The image shift operation involves the transfer of all the image elements and is thus communication-intensive, accounting for a high percentage of the local operation execution time on the present version of NON-VON 3. In the next subsection, we describe two algorithms to perform this operation for both gray-scale and binary images. In the following subsection, we present the algorithms for image correlation.

5.3.1. Image Shift Algorithms

The algorithm for shifting a binary image involves reporting the size and location information of the black rectangles, one by one, to the CP using the RESOLVE instruction. For each reported rectangle, the new location of the rectangle is computed using the reported location and the horizontal and vertical shifting required. The new location information and the rectangle size are then broadcast to all the PE's in the tree. All leaf PE's corresponding to pixels falling within the rectangle boundary set their BINARY1 variable equal to 1. The boolean variable BINARY1 is initialized to the value 0. The binary image tree representation of the shifted image can be then computed as described in Chapter 4. The algorithm just described does not perform "image wraparound". As a result, parts of the original image, determined by the amount of shift, no longer exist in the shifted image. The algorithm can be modified to perform image wraparound as follows. If the reported rectangle in its new position contains a portion which is outside the boundary of the image, then this portion wraps around. To perform this wraparound, the rectangle in its new location is shifted horizontally by a distance equal to the width of the rectangle, then vertically by a distance equal to the rectangle length. Finally, it is shifted vertically and horizontally a distance equal to its length and width, respectively. The direction of the shift is opposite to the original shift direction. For example, if the original shift is in the east and south directions, then the wraparound shifts are performed in the west and north directions. If the portion of the rectangle to be wrapped around totally exists on the east or west side of the image, then only a horizontal shift is needed. On the other hand, if this portion exists only on the north or south sides of the image, then only a vertical shift is needed.

We now describe a non-wraparound N-PASCAL algorithm to shift an image *i* places in the horizontal direction and *j* places in the vertical direction. Positive values of *i* and *j* indicate image shifts in the right and down directions, respectively while negative values indicate shifting the image in the left and up directions.

```
Procedure bimage_shift(i, j: integer ; k:char);
    label 2, 4;
var
    i1, j1, k1: integer;
    x, y, l, w: integer;
vector_var
    BINARY1, REPORTED: boolean;
```

```
begin

/* 1. Initialize the vector variables BINARY1 and REPORTED.
REPORTED is set equal to 1 only for rectangles to be shifted. The
character scalar variable k specifies the type of rectangles to be used in
the shift operation. */

  if k = 'B' then k1 := 1
   else k1 := 0;
  if k1 = 1 then BINARY1 := false
    else BINARY1 := true;
  REPORTED := false;
  where FQUAD = k do REPORTED := true;

/* 2. Select a PE corresponding to a rectangle in the binary image
tree representation that has not yet been reported. Report its size and
address information to the CP. If there are no PE's satisfying this
condition, the shift operation is done. */

  2:
  where REPORTED = false do N_A1 := true
   elsewhere N_A1 := false;

/* The function N_RESOLVE selects a single PE among the enabled
PE's. It returns 0 if there are no enabled PE's. */

  if N_RESOLVE() = 0 then goto 4;
  where N_A1 = true do
    begin
      N_REPORT8(XADD, x);
      N_REPORT8(YADD, y);
      N_REPORT8(XSIDE, w);
      N_REPORT8(YSIDE, l);
      y := y + j;
      x := x + i;
      REPORTED := true;
    end;

/* 3. Broadcast the new location information, and set the vector
```

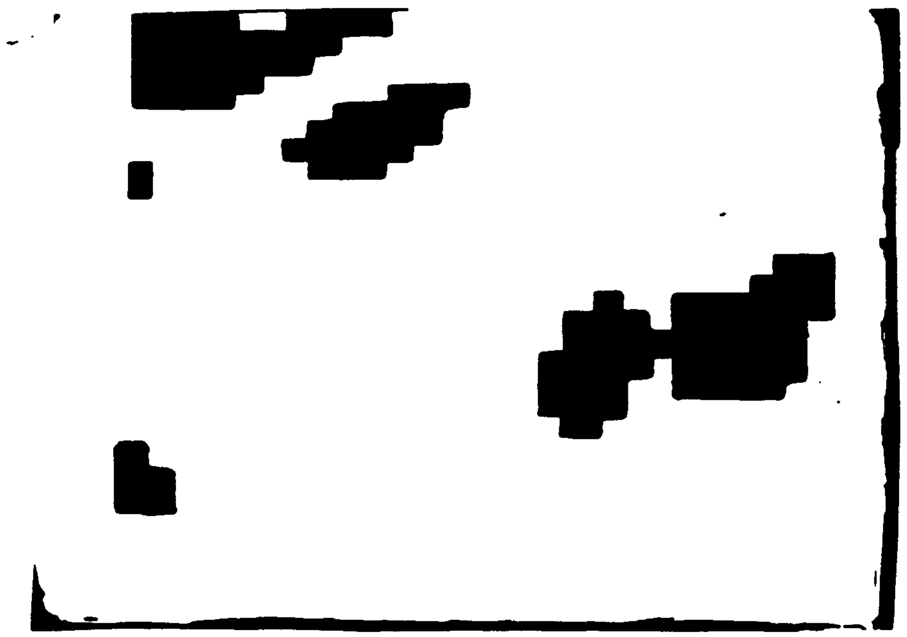
```
variable BINARY equal to 1 only in those leaf PE's corresponding to
pixels falling within the boundary of the rectangle. */

  where (XADD >= $x$) and (XADD < $x+w$)
          and (YADD >= $y$) and (YADD < $y+l$) do
    if $k1$ = 0 then BINARY1 := false
      else BINARY1 := true;
  goto 2;

4.;
end;
```

The algorithm executes in time proportional to the number of black rectangles in the binary image tree representation of the image. Typically, this number is of $O(d)$, where d is the diameter of the image [Dyer 82a], as will be discussed in Chapter 6. Thus, the time required to execute this operation is typically $O(n^{1/2})$, where n is the image size. Alternatively, the white rectangles can be used in place of the black rectangles. The only change involves initializing the BINARY1 variable to 1 in Step 1, and setting it to 0 in step 3. If the number of white and black rectangles is known in advance, this decision may be made in such a way as to minimize number of rectangles to be processed. Note that the distance to be shifted in both the the horizontal and vertical positions does not affect the execution time of the algorithm. The NON-VON 3 code (Appendix B) executes about 40 instructions per reported rectangle. Thus, shifting a 128 $\times$ 128 binary image containing 500 rectangles requires about 5 msec. Figure 5-4 depicts the results of shifting the binary image of Figure 5-3 three pixels in the right directions and 5 pixels in the up direction. Two cases are shown in the figure, the

Figure 5-4: Binary Image Shifting



(a) Without Wraparound

(b) With Wraparound

first one, Figure 5-4-a, is shifting without wraparound and the second case, Figure 5-4-b, shows the wraparound effect.

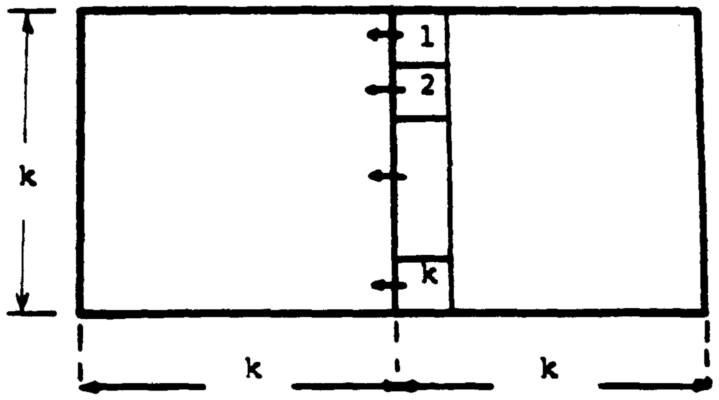
Next, we describe the algorithm to perform gray-scale image shifting. For the sake of simplicity, we consider the case of shifting the gray-scale image one position in the left direction. Slightly modified versions of this algorithm may be used to shift the gray-scale image in other directions.

Recall that gray-scale images are stored in the leaf PE's, and that the leaf PE's of a subtree in the NON-VON tree correspond to a block of the stored image. Figure 5-5 shows two adjacent $k \times k$ blocks of the image and the NON-VON tree representation of these two subimages. Shifting the image one position in the left direction involves transferring k image values from subtree number 1 to subtree number 2 through the common root of the two subtrees (PE3). This operation can be performed in parallel for all subimages of size $k \times k$, using the PE's at the level corresponding to rectangles of size $2k \times k$.

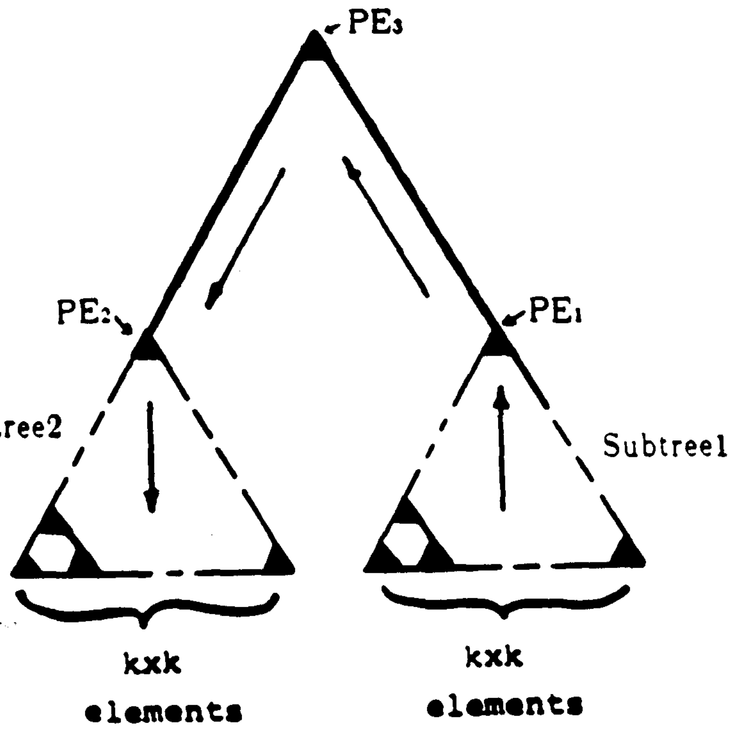
The procedure to transform the k elements sends the elements to be shifted up the tree one by one in a pipelined fashion. After a number of steps equal to the height of the subtree ($2 \log k$), the first element reaches the root of the source subtree, PE1. This element is then transferred to the root of the destination subtree, PE2, through the common root of the two subtrees, PE3. At this point, the algorithm

Figure 5-5: A $2k \times k$ Subimage and its NON-VON Tree Representation

(a)



(b)



starts sending the elements as they arrive to PE1 to PE2 through PE3. Each time an element is transferred to PE2, the algorithm moves the elements that have arrived to PE2 one level down the destination subtree. Thus, in time proportional to (k + log k), image elements on the boundary of $k \times k$ subimages are shifted one position left. Shifting the whole image includes repeating this operation for

$k = 1, 2, 4, \ldots, (n^{1/2})/2$

where n is the image size. The time required to shift the whole image is proportional to the sum of 1, 2, 4, ..., $(n^{1/2})/2$, which is equal to $n^{1/2} - 1$. Thus, the time required to shift the whole image one position left is of $O(n^{1/2})$.

The N-PASCAL algorithm to perform the shifting of k elements on the western boundary of a $k \times k$ subimage to the neighboring $k \times k$ subimage follows:

```
Procedure subimage_left_shift(k, h: integer);
var
    i, j: integer;
vector_var
    REL_X, REL_Y, SHIFT_VALUE: integer;
    SH_LC, SH_RC, SH_P, GRAY2_VALUE: integer;
    LEAF: boolean;

 /* The following procedure enables in each subtree the PE
corresponding to element number i among the elements to be shifted and
reads its gray-scale value into the vector variable SHIFT_VALUE.
SHIFT_VALUE is set equal to 0 in all other leaf PE's. */
```

```
Procedure pick_element(n: integer);
begin
  where LEAF = true do
    begin
      SHIFT_VALUE := 0;
      where (REL_X = 0) and (REL_Y = n) do
        SHIFT_VALUE := GRAY_VALUE;
    end;
end;
```

/* The following procedure sends the elements to be shifted one level up the tree */

```
Procedure move_up;
begin
  where LEAF = false do
    begin
      N_RECV8(LC, SHIFT_VALUE, SH_LC);
      N_RECV8(RC, SHIFT_VALUE, SH_RC);
      N_OR8(SHIFT_VALUE, SH_LC, SH_RC);
    end;

end.
```

/* The following procedure enables a single PE in each subtree corresponding to the destination element *i* and assigns the value of SH_P to its GRAY2_VALUE vector variable. */

```
Procedure assign_element(n,m: integer);
begin
  where (LEAF = true) and (REL_X = m)
            and (REL_Y = n)
    do GRAY2_VALUE := SH_P;
end;
```

/* The following procedure sends the elements of the vector variable SH_P one level down the tree */

```
Procedure move_down;
begin
  N_RECV8(P, SH_P, SH_P);
  end;

 /* The following procedure assigns to the variable SH_P in the root of
subtree 2 the value of the variable SHIFT_VALUE in the root of the
subtree 1 */

  procedure move_around;
  begin
    N_RECV8(RC, SHIFT_VALUE, SH_RC);
    N_SEND8(LC, SH_RC, SH_LC);
    SH_P = SH_LC;
  end;

 /* This is the main procedure: */

begin

 /* 1. Compute the address of each image point relative to the k x k
block in which it exists, and mark leaf PE's. */

  REL_X = XADD mod k;
  REL_Y = YADD mod k;
  mark_leaf(LEAF);

 /* Now start calling the various procedures to move the boundary
elements between the two blocks. Note that h is the number of the
level where the roots of the subtrees exist. */

  for i = 1 to (k + 2 * h) do
    begin
      if i <= k then pick_element(i - 1);
      if i <= k+h then move_up;
      if i >= h then move_around;
      if i > h then move_down;
      if i >= 2 * h then assign_element(i - 2 * h, k);
    end;

end;
```

The NON-VON 3 code for this procedure is included in Appendix B. To shift the whole gray-scale image one position, this procedure is called with values of k ranging from 1 to $n^{1/2}/2$. (The cases of k equal to 1 and 2 can be actually programmed differently as they consist only of few tree communication steps.) For each element shifted, 48 instructions are executed, requiring 12 μsec. The time required to shift the whole binary image is proportional to image side length. For a 128 $\times$ 128 gray-scale image, 1.6 msec is needed to shift the whole image one pixel to the left. Shifting the gray-scale image more than one pixel is performed by executing this algorithm a number of times equal to the number of shifts required.

Adding one-bit mesh connections to the leaf PE's of the NON-VON machine should reduce significantly the time required for gray-scale image shifting. Such connections allow single-pixel shifts in 2 μsec for gray-scale images shifting, and 250 nsec for binary images. Unlike the algorithm described above to shift binary images, however, this approach would require additional time for shifts of more than one pixel.

5.3.2. Image Correlation Algorithms

In this subsection, we describe two algorithms to perform the cross correlation operation on the NON-VON machine. The cross correlation value at a certain position in the image is defined as

$$cross\_correlation = \sum_{i=1}^{m} x_i y_i \qquad (5.4)$$

where y_i are the template elements, x_i are the image elements covered by the template elements, and m is the number of template elements. The first algorithm is a direct parallel implementation of the standard sequential machine algorithm. The second one uses the tree structure of NON-VON to reduce the number of image shifts required to compute the correlation, thus reducing the time required to execute these operations.

The first algorithm starts by initializing the variable CORR_VAL, which stores the correlation function value in each leaf PE, to zero. Each leaf PE then computes the correlation function term corresponding to its own pixel value and adds the resulting value to the variable CORR_VAL. To compute the rest of the correlation function terms, each leaf PE reads the value of image points in its neighbor PE's using the shift operation, as described in the previous section. For each value read, a term in the correlation function is computed by each of the leaf PE's and its value is added to the vector variable CORR_VAL. Consequently, the algorithm consists of a repeated sequence of image shift and compute steps. This sequence is repeated a number of times depending on the template size. For example, if we have a 3 x 3 template, the sequence is repeated eight times (template size - 1).

We now describe the first image correlation algorithm in N-PASCAL. For simplicity, we assume that the template size is 3 x 3 and that the correlation function is the sum of products of image elements and template elements. The correlation function value is stored in the image element under the center pixel in the template.

The N-PASCAL algorithm follows:

```
Procedure image_corr1;
var
    i: integer;
    temp: array[0..8] of integer;
vector_var
    G_VAL1, CORR_VAL: integer;
    LEAF: boolean;
begin

 /* 1. Initialize the vector variable CORR_VAL and compute the first
term of the correlation function.  */

  CORR_VAL := 0;
  CORR_VAL := CORR_VAL + temp[0] * GRAY_VALUE ;

 /* 2.  Compute the rest of the correlation terms.  The function
shift_l(X, Y) shifts left the image represented by the vector variable X,
and stores the result image in the vector variable Y. Shift_u, shift_d,
and shift_r are defined similarly.  */

  shift_l(GRAY_VALUE, G_VAL1);
  CORR_VAL := CORR_VAL + temp[1] * G_VAL1 ;

  shift_d(G_VAL1, G_VAL1);
  CORR_VAL := CORR_VAL + temp[2] * G_VAL1 ;

  shift_l(G_VAL1, G_VAL1);
  CORR_VAL := CORR_VAL + temp[3] * G_VAL1 ;

  shift_r(G_VAL1, G_VAL1);
  CORR_VAL := CORR_VAL + temp[4] * G_VAL1 ;
  shift_r(GRAY_VALUE, G_VAL1);
  CORR_VAL := CORR_VAL + temp[5] * G_VAL1 ;

  shift_u(G_VAL1, G_VAL1);
  CORR_VAL := CORR_VAL + temp[6] * G_VAL1 ;

  shift_l(G_VAL1, G_VAL1);
  CORR_VAL := CORR_VAL + temp[7] * G_VAL1 ;

  shift_l(G_VAL1, G_VAL1);
  CORR_VAL := CORR_VAL + temp[8] * G_VAL1 ;
end;
```

Note that the order in which the image shifts are performed depends on where the value of the operation is to be stored. The time required to execute the function is $O(m(s+c))$, where m is the template size, c is the time required to compute a term in the correlation function, and s is the execution time of the image shift operation. On the present version of NON-VON, the image shift time is $O(n^{1/2})$ for typical images, where n is the image size. Thus, the time required can be expressed in terms of image size as $O(m(n^{1/2}+c))$. In the N-PASCAL procedure described above, the correlation function term is computed by multiplying two 8-bit integers, and then performing a 32-bit integer add (for a 15-level tree). This operation takes about 30 μsec on the present version of NONVON 3. The image shift operation for a 128 $\times$ 128 image takes about 1.6 msec, on the present version of NON-VON, as stated earlier. Therefore the computation time is very small compared to the shift time. For a 3 $\times$ 3 template, the correlation function as defined earlier executes in 12.8 msec. On a mesh-connected machine, the image shift operation is performed in constant time, and the computation of image correlation thus requires only $O(mc)$ time. For templates of different sizes, the same procedure can be adapted to compute the correlation function value. Note that the time required to execute the procedure on the present version of NON-VON (which lacks mesh connections) is dominated by the image shift time.

The second approach treats the whole image as a number of subimages stored in the leaf PE's of NON-VON subtrees. Each subimage contains all local image information required to compute the correlation for a subset of its points. The image correlation for these points is computed for all subimages in parallel. Computing the correlation function for image points on the boundary of subimages requires image information from neighboring subimages. This is the only case in which the image shifts are required. Figure 5-6 depicts a subimage of size 4 × 4 and a template of size 3 × 3. There are 4 points in this subimage (points 1 to 4) at which the correlation function can be computed using only the subimage values.

The computation of the correlation function for these four points is performed by storing the template for each position in the leaf PE's, such that PE's that are not covered by the template have template value zero. This is equivalent to scanning the template all over the subimage, and at

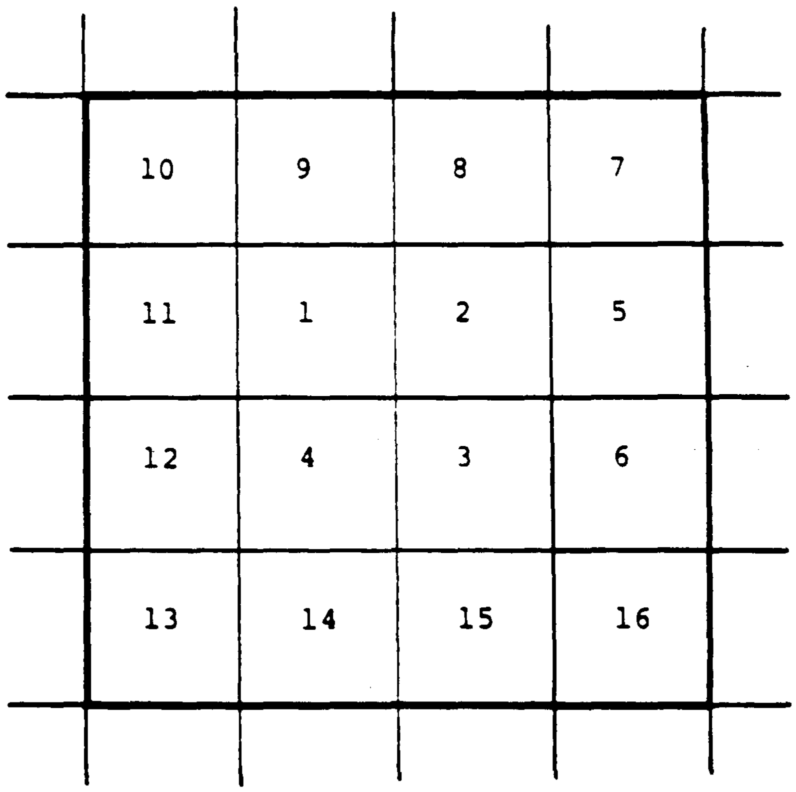


Figure 5-6: Image Correlation Template in a 4 × 4 Subimage

each of the positions where all the subimage elements covered by the template exist, the correlation function is computed. For a 3 × 3 template, four template values must be stored in each PE. This is performed for all PE's in time proportional to the template size. The correlation function is then computed in the leaf PE's, and the results are compiled using the tree connections in each subtree root. Shifting the whole image one position to the left enables us to compute the correlation function for points 5 and 6. Shifting the resulting image one position down makes it possible to compute the function at points 7 and 8. Similarly four shifts enable us to compute the function at points 9 through 16. Note that only six image shifts are required in this approach instead of eight shifts in the standard algorithm described earlier in this section. The number of shifts required is given by the following equation:

$$number\ of\ shifts = (s - k^2)/k \tag{5.5}$$

where s is the subimage size, and $k \times k$ is the number of points at which the correlation function can be computed using the subimage information. The value of k depends on both the template size and the subimage size. For example, if the template size is 5 × 5 and the subimage size is 8 × 8, then k is 4 and the number of shifts required is 12 instead of the 24 required by the first approach.

We now present the N-PASCAL algorithm to compute the image correlation, assuming subimages of size 4 × 4 and a template of size 3 × 3.

```
Procedure image_corr2;
var
    i: integer;
    temp: array[1..4,1..16] of integer;
vector_var
    G_VAL1, G_VAL2: integer;
    CORR_VAL: integer;
    TEMP1: array[1..4] of integer;
    LEAF: boolean;
```

```
Procedure comp_corr(i: integer);
var
    j: integer;
vector_var
    CORR_L, CORR_R, X, Y, NO: integer;
begin

/* 1. Compute the correlation terms at the leaf level. */

  X := Y * TEMP1[i];

/* 2. Add the terms together using the tree connections. */

  for j := 1 to 4 do
    begin
      N_RECV8(LC, X, CORR_L);
      N_RECV8(RC, X, CORR_R);
      X := CORR_L + CORR_R;
    end;

/* 3. Send the result back to the leaf PE's. */

  for j := 1 to 4 do
    N_RECV8(P, X, X);

 /* 4. Enable only PE's in position i in the subimages and store the
correlation function value in the variable CORR_VAL */

  where NO = i do CORR_VAL := X;
end;


begin

 /* 1. Compute the correlation function for points at positions 1
through 4. The function comp_corr(i) computes the correlation function
at position i. */

  for i := 1 to 4 do
   comp_corr(i);
  /* 2  Compute the rest of the correlation terms  */

   shift_l(GRAY_VALUE, G_VAL1);
   comp_corr(5), comp_corr(6),

   shift_d(G_VAL1, G_VAL2);
   comp_corr(7); comp_corr(8);

   shift_u(G_VAL1, G_VAL2);
   comp_corr(15); comp_corr(16);

   shift_r(GRAY_VALUE, G_VAL2);
   comp_corr(9), comp_corr(10);

   shift_u(G_VAL2, G_VAL1),
   comp_corr(11); comp_corr(12);

   shift_u(G_VAL2, G_VAL1);
   comp_corr(13), comp_corr(14);
end,
```

The above N-PASCAL algorithm executes in time proportional to ($j(s + k \log a)$), where j is the number of shifts, s is the shift time, k is the number of correlation function values computed after each shift, and a is the area of the subimage This time is again dominated by the time required to perform the image shifting In the case of 3 × 3 templates, this time is approximately 9 6 msec

The performance figures presented above clearly indicate that if tree communication is used to shift the whole image in any direction, then this time dominates the execution time of local operations on the present version of NON-VON Adding the capability of fast image shifting to

NON-VON speeds up these operations considerably There are several ways to add this capability The simplest and most direct approach is to incorporate a single-bit mesh connections at the leaf level, as is in fact currently planned. A second possible solution would be to add the mesh connections at an intermediate level, reducing the complexity of the machine wiring. For example, if the mesh connections are incorporated at the second level above the leaves, the required number of wires is reduced by half. This, however, will increase the time required for image shifting.

We are also considering a slight modification to the current PE design, by adding a second special rotating register, that will result in byte multiplication being performed in about 3.0 μsec instead of 30 μsec on the present version. The fast multiplication will reduce significantly the execution time in case of adding mesh connections. Based on these two proposed modifications to the NON-VON hardware, we have computed the expected execution time for the image correlation operations. Table 5-1 summarizes these projections

Table 5-1: Execution Time for Some Low-Level Operations on Parallel Machines

	ICL DAP	MPP	NON-VON 3 (without mesh connections)	NON-VON 3 (with mesh connections)
Speed (MIPS)	5.0000	10.0000	4.0000	4.0000
Execution time a. Histogram (msec) (128 X 128 image, 256 bins)	70.0000	120.0000	4.0000	4.0000
b. Thresholding (μsec)	--	2.5000	1.5000	1.5000
c. Binary image shifting (msec)	--	0.0001	5.0000	0.00025
d. Gray-scale image shifting (msec)	--	0.0008	1.6000	0.0020
e. Cross correlation (3 X 3) msec	--	0.2000	9.6000	0.0500
(7 X 7)	--	1.0000	27.0000	0.2500

Chapter 6
Geometric Algorithms

Geometric operations usually accept binary images as their input and produce a symbolic description of the geometric properties of input image objects as their output. The output of these operations usually results from combining image data that is found in distant parts of the image. Consequently, these operations can be viewed as global operations performed on images. Examples of these operations include identifying separate objects in the image and computing different geometric descriptions of these objects. In this chapter, we describe algorithms for labeling image objects, and for computing some of their geometric properties, such as area, perimeter, genus, centroid, moments, and compactness. Algorithms to perform set operations on binary images are also described. We assume throughout this chapter that the binary image is already stored in the tree, and that the binary image tree representation is constructed as described in Chapter 4.

6.1. Connected Component Labeling

Connected component labeling is a basic operation in image analysis that identifies the disjoint regions of a binary image. The connected component labeling algorithm assigns unique labels to disjoint connected regions of a binary image as illustrated in Figure 6-1. The disjoint regions identified by the algorithm may be then analyzed separately using one label at a time.

We assume in this chapter that the image objects have been separated from their background using some segmentation procedure. Thresholding based on image histogramming, as described in the previous chapter, is an example of such a segmentation procedure. In describing the NON-VON connected component algorithm, we assume that only the foreground components (the black areas) are to be labeled; but the same procedure can also be applied to background components (the white areas).

6.1.1. The Connected Component Labeling Algorithm

There are several algorithms for performing the labeling operation on a sequential machine depending on the data structure used to represent the image. If for example, a two dimensional array is used, the classical sequential algorithm scans the binary image from left to right and top to bottom [Ball 82]. For each foreground pixel (pixel with value '1'), its left and top neighbors are examined. If they both have no labels assigned to

(a) The input binary image

```
00000000000000000000000000000000
00111101111111000001110001110000
00111111111111110011110001111000
00111111111111111001110001110000
00111101111100111100111111100000
00000000011100011110001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000001110000000
00000000011100000000000100000000
00000000011110001111100000000000
00000000011100111111111000000000
00000000000001111000111100000000
00000000000001110000011110000000
00000000000001111000111110000000
00000000000000111111111100000000
00000000000000001111110000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

(b) The labeled image

```
00000000000000000000000000000000
00777707777777000008880008880000
00777777777777770088880008888000
00777777777777777008880008880000
00777707777700777700888888800000
00000000077700077770008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000008880000000
00000000077700000000000800000000
00000000077770002222200000000000
00000000077700222222222000000000
00000000000002222000222200000000
00000000000002220000022220000000
00000000000002222000222220000000
00000000000000222222222200000000
00000000000000002222220000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Figure 6-1: Connected Component Labeling of a Binary Image

them, then the pixel is assigned a new label. If only one of them has a label, or both have the same label, then the examined pixel is assigned this label. If they have different labels, then the pixel is assigned the smallest of them, and an entry is created in an equivalence table taking note of the equivalence between the two labels. So, after one scan of the whole image, each pixel is assigned a label, and equivalence relations between labels are known. A second pass is thus required to reassign a unique label to pixels in the same equivalence class. This algorithm takes time proportional to the size of the image. For a 512 × 512 image (256K pixels), this algorithm executes in about 300 seconds on a VAX 11/750 [Lumi 83]. Other variations of the algorithm and their execution times for different image sizes are described in [Lumi 83].

A second algorithm on sequential machines that uses the quadtree data structure is described in [Same 81c]. The average execution time of this algorithm is proportional to the sum of the black and white squares in the quadtree data structure representing the binary image. A parallel algorithm that computes the connected components on an $n^{1/2} \times n^{1/2}$ mesh-connected parallel SIMD computer in $O(n^{1/2})$, is described in [Nass 80].

The NON-VON algorithm described in this thesis scans the rectangles of the binary image tree representation of the image. The rectangles are scanned in terms of their size, rather than in terms of their location. The

RESOLVE instruction is used to achieve that, and the reason for that is to eliminate the need for a second pass, as will be clear soon. For each rectangle, all neighbor rectangles (rectangles having a common boundary) are labeled with the same label. If any of the neighboring rectangles has already been assigned a label, then this equivalence case is noted. Equivalence cases are limited by two at each step, and they are treated after each scanning step.

The algorithm, as implemented on NON-VON, starts by assigning the label zero to all black rectangles of the binary image tree. The RESOLVE instruction is then used to report to the CP the black rectangles of the binary image one by one, in order of their sizes. This can be done simply by starting at the root level and enabling only PE's holding black rectangles at that level, and then reporting them to the CP in an order that depends on how the RESOLVE instruction is implemented. This order is not important to our algorithm, since all rectangles on a specific level have the same size. After all the black rectangles on level i level have been reported, we enable the PE's with black rectangles in the next level $i+1$ down the tree and repeat the reporting procedure. The algorithm terminates when all black rectangles in the leaf level have been reported.

For each reported rectangle, the CP assigns a new label if it has not already been assigned a label. The CP broadcasts instructions to mark and label all adjacent rectangles in different directions with the same label of the reported rectangle. If during the adjacency test, any adjacent rectangle has already been labeled, then this adjacent rectangle and all rectangles having the same label value will be assigned the label of the reported rectangle. Another black rectangle is picked as described above and the labeling procedure is repeated. This procedure guarantees that all black rectangles are labeled, since all black rectangles are reported to the CP and are assigned a label if they have not already have a label. One pass is enough because at each step, all rectangles that have been labeled and are adjacent to each other have the same label. During the execution of this algorithm, information about the common boundaries between rectangles will be stored locally at each node to be used later for computing some geometrical properties of different components.

The algorithm is described more rigorously below using N-PASCAL. The vector variable COMP_LABEL is used to store the region label to which the rectangle belongs. The vector boolean variables TE, TN, TW, and TS are used to indicate the existence of a common boundary between a rectangle and its neighbors in the east, north, west and south directions respectively. The algorithm sets the value of these variables, such that if two rectangles share a common boundary, then this information is stored only in the smaller of the two rectangles. Another vector variable, REPORTED, is used by a rectangle to mark itself as reported. The scalar variable ***newlabel*** is used to store a new unassigned label. While the scalar variable ***comlabel*** is used to store the label assigned to adjacent rectangles in adjacency testing respectively. The scalar variable ***curlev*** will be used to refer to the current level from which the algorithm picks black rectangles.

In what follows, we describe the N-PASCAL algorithm to label the connected components of a binary image:

```
Procedure connected_comp(no_levels: INTEGER);
    label 3, 6, 8;
var
    newlabel, comlabel, curlev: integer;
    x, y, xs, ys, l: integer;
vector_var
    TEMP, CUR_LEV integer;
    TE, TN, TW, TS, TO_BE_LAB boolean;
    ANY_A1, REPORTED, EQUIV boolean;
begin
```

/* 1 Initialize the scalar variables ***newlabel*** and ***curlev*** to 0. Set the variable CUR_LEV equal to the level number in all PE's. The procedure to perform this is very simple and is implicitly included in the code for initializing the NON-VON tree. We will not describe this procedure here. */

```
  newlabel = 0; curlev = 0,
  TW = false, TE = false,
  TN = false; TS = false,
  REPORTED = false;
  set_level_number(CUR_LEV),
```

/* 2 Enable all PE's corresponding to black rectangles (FQUAD='B'). Set the vector variable COMP_LABEL (dclared in the main procedure) equal to 0, and the variable REPORTED to 1 */

```
where FQUAD = 'B' do
  begin
    COMP_LABEL := 0;
    REPORTED := true;
  end;
```

/* 3. Enable all PE's corresponding to black rectangles that have not been reported yet at the current level. If none is enabled and the current level is the leaf level, then stop; else if none is enabled and the current level is not the leaf level then repeat this step for the next level. From the enabled PE's, select and enable only one PE. */

```
3:
where (CURLEV = curlev) and (REPORTED = true)
  do N_A1 := true
 elsewhere N_A1 := false;
ANY_A1 := N_RESOLVE();
if (ANY_A1 = false) and (curlev = no_levels) then
  goto 8;
if (ANY_A1 = true) and (curlev <> no_levels) then
  begin
    curlev := curlev + 1;
    goto 3;

  end;
```

/* 4. Report the address, size, and label information of the enabled PE to the CP. Mark the enabled PE as being reported. If the rectangle associated with the PE has not been labeled before (COMP_LABEL = 0), then assign to it a new label. Set *comlabel* equal to the label of the reported rectangle. */

```
where N_A1 = true do
  begin
    N_REPORT8(XADD, x);
    N_REPORT8(YADD, y);
    N_REPORT8(XSIDE, xs);
    N_REPORT8(YSIDE, ys);
    N_REPORT8(COMP_LABEL, l);
    REPORTED := false;
  end;
```

```
if l = 0 then
  begin
    newlabel := newlabel + 1;
    l := newlabel;
  end;
comlabel := l;
COMP_LABEL := l;
```

/* 5. Test for adjacency in the four directions one at a time. This is done by broadcasting for each direction the range in which the location of the adjacent rectangles should lie. This range is computed using the reported rectangle size and location information. Only rectangles in this range will be enabled. If any of them has a label other than zero, then its value is reported to the CP. Only two rectangles at most can have their labels equal a value other than zero, as will be proven later. All adjacent rectangles labels are set to **comlabel**. During check for adjacency, information regarding adjacency are stored in PE's. */

```
TO_BE_LAB := false; EQUIV := false;
where REPORTED = true do
  begin
    where (XADD = x + xs) and (YADD < y + ys)
                          and (YADD >= y) do
      begin
        TW := true;
        TO_BE_LAB := true
      end;

    where (YADD = y + ys) and (XADD < x + xs)
                          and (XADD >= x) do
      begin
        TN := true;
        TO_BE_LAB := true;
      end;

    where (YADD + YSIDE = y)  and (XADD < x + xs)
                              and (XADD >= x) do
      begin
        TS := true;
        TO_BE_LAB := true;
      end;

    where (XADD + XSIDE = x)  and (YADD < y + ys)
```

```
                                   and (YADD >= y) do
          begin
            TE  = true;
            TO_BE_LAB  = true;
          end;
      end;

    where  TO_BE_LAB = true  do
      begin
        if COMP_LABEL = 0 do
          begin
            EQUIV  = true;
            TEMP  =  COMP_LABEL;
          end;
        COMP_LABEL  = comlabel;
      end;
```

/* 6. This step takes care of the equivalence cases. For each adjacent rectangle with COMP_LABEL not equal to zero, broadcast the value of its COMP_LABEL. Set COMP_LABEL in all PE's having the same label value equal to *comlabel* */

```
    6
    where EQUIV = true  do N_A1  = true
      elsewhere N_A1  = false;
    if  N_RESOLVE() = 0 then  goto 3
     else where N_A1 = true  do
       begin
         EQUIV  = false;
         N_REPORT8(TEMP, l);
         where (COMP_LABEL = l) and (FQUAD = 'B')
           do COMP_LABEL  = comlabel;
         goto 6;
       end;
    8 ;
  end;
```

In Step 5, a crucial part of the algorithm's efficiency is due to the fact that at most two of the adjacent rectangles can have labels other than zero. To prove this, we assume that rectangle 3 has been reported to the

CP and that without loss of generality we are looking for rectangles adjacent to it along its eastern boundary, as shown in Figure 6-2. Assume also that rectangle 2 is adjacent to rectangle 3 in the east direction as shown in Figure 6-2-a. If rectangle 2 has been labeled before, then it must be adjacent to a rectangle 1 of size greater than or equal to rectangle 3. This is true because rectangles are reported to the CP in order of their size. Rectangle 2 can share a common boundary with rectangle 1 in the east, north, or south direction along the boundaries of the shaded area shown in Figure 6-2. From the way we build the binary image tree, we know that if rectangle 1 is to the east of rectangle 3 and is larger than or equal to it, then the distance separating them is greater than or equal to the width of rectangle 3 (L3). Thus, we conclude that if rectangle 2 is adjacent to both 1 and 3, and rectangle 2 is smaller than or equal to rectangle 3, then its width (L2) is equal to L3. There is only one rectangle that can satisfy this condition, as shown in Figure 6-2-b, where its two unique positions are shown. In addition to rectangle 2 in the previous case, we can have only a second rectangle 2 that could have been labeled before because it is adjacent to a larger or equal size rectangle in either the north or south direction as shown in Figure 6-2-b. Figure 6-2-c shows the third possible case, where we have two rectangles (not necessarily of the same size) that have been labeled before, and which are adjacent to rectangle 3, and to larger or equal size rectangles in the north and south direction respectively.

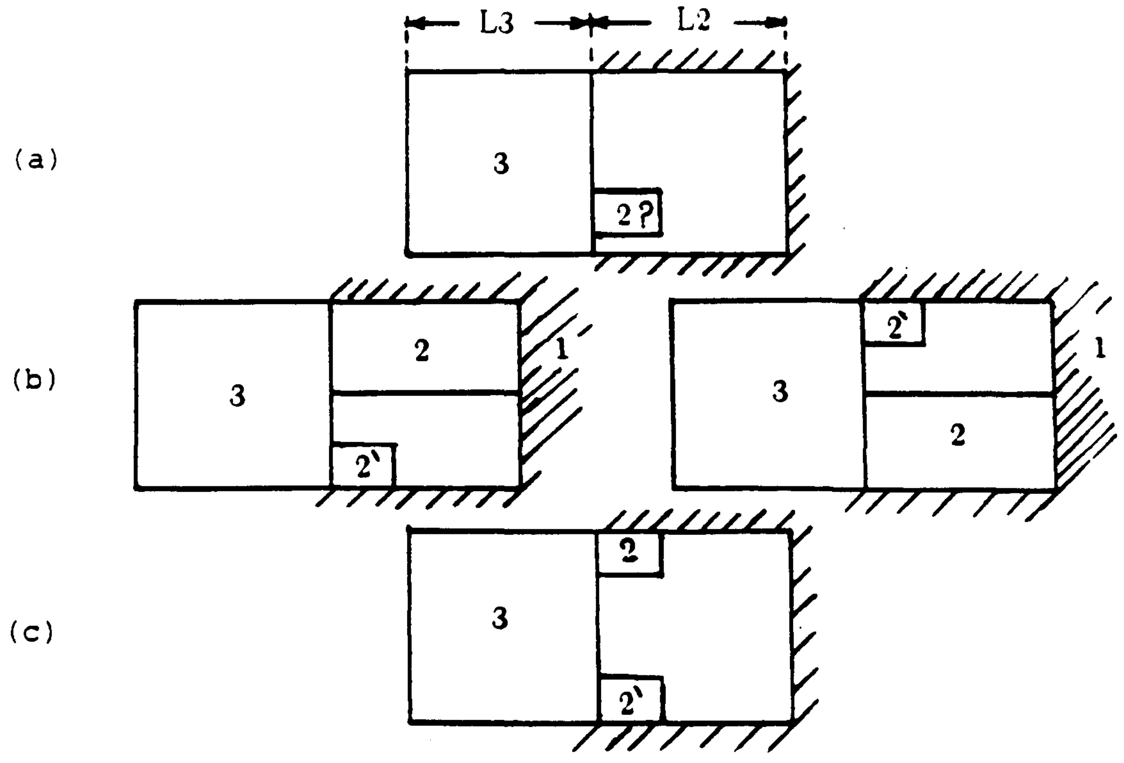


Figure 6-2: Cases of a Rectangle Adjacent to Two Rectangles

A similar proof is valid for adjacent rectangles in the other directions.

Steps 3 through 7 are repeated a number of times equal to the number of black nodes in the image. Each step consists of at most a fixed number of NON-VON instructions. Thus, the time the algorithm takes is proportional to the number of rectangles in the binary tree $O(B)$. If prior information about the adjacency for single pixels is known (for example, during the broadcasting of the image), then those rectangles with adjacencies only in one direction do not have to be reported to the CP once they are labeled.

The information obtained about the common boundaries between rectangles can be used not only to compute component properties in time proportional to the height of the NON-VON tree, but also to mark all boundary pixels with respect to a black region. This is a simple procedure, and it will not be described in this thesis. It executes in a time proportional to the number of black rectangles in the binary image tree. Marking boundary pixels can be used in other algorithms, such as determining adjacency relationships between components.

Note that in the labeling procedure, only binary image tree rectangles belonging to the connected component are being labeled. If all the image pixels contained within the connected components are also to be labeled, then a simple procedure that executes in time proportional to the height of the NON-VON tree may be used to perform this operation. The basic step in this procedure is to let each PE reads the label of its parent node, and only if this PE corresponds to a non-gray rectangle that is not part of the binary image tree (FQUAD is equal to 'N'), then the read value is stored in its LABEL variable. This step is repeated a number of times equal to the tree height, after which all leaf PE's will have their LABEL variables set equal to the label value of the connected component to which they belong. The N-PASCAL for this simple procedure follows

```
Procedure spread_label(no_levels INTEGER);
var
```

```
   i  integer;
vector_var
   TEMP  integer;
begin
  for i := 2 to no_levels do
    begin
      N_RECV8(P, COMP_LABEL, TEMP);
      where (FQUAD = 'N') do COMP_LABEL := TEMP;
    end;
end;
```

The NON-VON 3 code for this procedure contains 4 instruction per iteration. Thus for a 15-level tree the procedure executes in approximately 16 μsec.

6.1.2. Connected Component Labeling Simulation

The algorithm described in the previous section has been simulated on both the functional and instruction-level simulators. Figure 6-3 shows (a) a 32 × 32 binary image (the same as that of Figure 5-3), that was input to the functional simulator and (b) the labeled foreground components. The simulator has also been used to label background objects as shown in part (c) of Figure 6-3. The binary image representation of this image contains 64 black rectangles and 88 white rectangles. It took about three seconds of actual (sequential) CPU time for the simulator to label all black components.

The NON-VON 3 code for the algorithm executes using at most about 200 NON-VON instructions, per iteration, in case of the existence of two equivalence cases in all directions.

With a NON-VON 3 instruction cycle of 250 nsec, the algorithm execution time is approximately .05B msec, where B is the number of black components. For an $n^{1/2} \times n^{1/2}$ binary image the average number of black rectangles in the binary image tree is $O(n^{1/2})$ [Dyer 82]. Thus the average case running time for the algorithm is $O(n^{1/2})$. The average running time of the algorithm on NON-VON for a 512 × 512 image with 1000 black rectangles is about 50 msec. (We can always in a time proportional to the height of the tree compute the number of black rectangles in the NON-VON tree, and use that number to estimate the running time of the algorithm.) The simulator was also used to compute some components properties based on the information produced by the connected component algorithm, as will be described in the next section.

6.2. Computing Connected Component Properties

In this section, algorithms that compute various shape properties of binary images are presented. This quantitative description of image shape properties is used to classify objects in the image, and is usually fed to high level vision procedures that interpret the image. Area, perimeter, moments, centroid, compactness, eccentricity, and image genus are the shape properties discussed in this section. Algorithms for computing the complement, intersection, and union of binary images are also described. We assume throughout this section that the image objects

```
00000000000000000000000000000000
00011011000000000000000000000000
00111110000000000000000000000000
11111111110000000000000000000000
11111100011000000000000000000000
11110011110000000000000000000000
11110011110000000000000000000000
11111111000000000000000000000000
11111110000000000000000000000000
11111000000000000000000000000000
11110000001110000000000000000000
00000000111100000000000000000000
00000001111100000000000000000000
00000011111000000000000000000000
10000001110000000000000000000000
10000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000001100000
00000000000000000000000011100000
00000000000000000010011111100000
00000000000000000111011111000000
00000000000000000111111111000000
00000000000000001111011111000000
00000000000000001110011110000000
00000000000000001110000000000000
00000000000000000100000000000000
00000000000000000000000000000000
10000000000000000000000000000000
11000000000000000000000000000000
11000000000000000000000000000000
```

(a)

```
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 9 9 0 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 9 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 9 9 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 9 9 0 0 0 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 0 0 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 0 0 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9 9 9 9 0 0 0 0 0 0 7 7 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 7 7 7 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 7 7 7 7 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 7 7 7 7 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 7 7 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 4 4 4 4 4 4 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 4 4 4 4 4 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 0 4 4 4 4 4 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 0 4 4 4 4 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(b)

```
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 9 9 5 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 9 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 9 9 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 9 9 6 6 6 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 3 3 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 3 3 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 9 9 9 9 5 5 5 5 5 5 7 7 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 7 7 7 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 7 7 7 7 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 7 7 7 7 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
10 5 5 5 5 5 5 7 7 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
10 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 4 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 4 4 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 5 5 4 4 4 4 4 4 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 4 4 5 4 4 4 4 4 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 4 4 4 5 4 4 4 4 4 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 4 4 5 5 4 4 4 4 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 2 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 2 2 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 2 2 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

(c)

Figure 6-3: Some Simulation Results

have been labeled by the connected component algorithm as described in the first section of this chapter, and that the vector variable LABEL contains the value of the label associated with each rectangle.

Before describing the algorithms, a common function that will be called by these algorithms is presented. The function, add_tree(X), is used to compute the sum of the values of the vector variable X found in all PE's in the NON-VON tree. The N-PASCAL procedure for evaluating the add_tree function follows:

```
Function add_tree(var X: integer):integer;
   label 2;
var
   count, sum: integer;
vector_var
   TEMP: integer;
   LEAF, TEMP1: boolean;
begin
```

/* 1. Initialize *count*. We assume that the number of tree levels *h* is initialized by the calling procedure. Enable only those PE's in the level above the leaf level. */

```
  count := 1;
  mark_leaf(LEAF);
  LEAF := not LEAF ;
  N_RECV1(LC, TEMP1, LEAF);
  LEAF := not TEMP1;
```

/* 2. Using the tree connections, every enabled PE reads the contents of the variable X in its two children, adds them to the value of its own variable X, and places the result in its variable X. */

```
  2:
  where LEAF = true do
```

```
    begin
      N_RECV8(LC, TEMP, X);
      X = X + TEMP;
      N_RECV8(RC, TEMP, X);
      X = X + TEMP;
    end;
```

/* 3. If the enabled level is the root level, then stop. Otherwise, enable all the PE's in the level above the currently enabled level. Goto step 2. */

```
  count = count + 1;
  if count <> h then
    begin
      LEAF = not LEAF ;
      N_RECV1(LC, TEMP1, LEAF);
      LEAF = not TEMP1;
      goto 2;
    end;
  add_tree = read_root(X);
end;
```

Steps 2 and 3 are executed h times, where h is the number of levels in the tree, after which the sum will be reported to the CP. Thus, the time required to compute the function add_tree(X) is $O(h)$. The NON-VON 3 code for this procedure is included in Appendix B. The time required to execute this procedure assuming, a 32-bit add operation, is 10 μsec per level. For a 128 x 128 image, the time required is thus 0.15 msec.

6.2.1. Area

The area of a region (connected component) is defined as the total number of black pixels in this region. The area of a connected component can be found by setting the vector integer variable **AREA** in all PE's containing rectangles belonging to this component equal to the area of the rectangle, and setting the same variable equal to zero in all other PE's. Then the function add_tree(**AREA**) is called, and the value of the function is the area being sought. The same procedure is repeated for other regions of interest in the image. An N-PASCAL procedure for implementing this simple algorithm follows:

```
Procedure conn_area(conn_label: integer);
var
   area_value: integer;
vector_var
    AREA: integer;
begin

  /* 1. Enable all PE's, and set the local variable AREA equal to zero in
all of them. */

  AREA := 0;

  /* 2. Enable only those PE's with their label equal to the label of the
connected component for which the area is to be computed. In all
enabled PE's, multiply the width by length and place the result in
AREA. Note that multiplication is performed through a series of shift
operations because the x-side and y-side values are all powers of 2. Once
this is done, enable all PE's. XSIDE and YSIDE are the globally defined
local variables initialized by the initialization procedure. */

  where COMP_LABEL = conn_label do
```

```
    AREA = XSIDE * YSIDE;

  /* 3. Call the function add_tree(AREA). When the function is
executed, the area will be reported to the CP.*/

  area_value = add_tree(AREA);
end;
```

The algorithm executes a fixed number of NON-VON instructions to compute Steps 1 and 2, independent of the size of the image. The multiplication in Step 2 is computed by a series of shift operations, since the values of XSIDE and YSIDE are powers of two. Step 3 takes time proportional to the height, h, of the tree. Thus, the algorithm executes in $O(h)$ time. The execution time of this algorithm is dominated by the time required to compute the function add_tree, which is approximately 0.15 msec for a 15-level tree, as shown earlier in this section.

6.2.2. Perimeter

The computation of object perimeter is a basic operation in image processing. The perimeter of a binary image object, represented by binary image trees, is defined as the total length of object black rectangle sides that are adjacent to white rectangles. The algorithm makes use of the information stored in each PE in the course of executing the connected component algorithm about the common boundaries of rectangles. The algorithm computes the perimeter of a region by adding the perimeter of all rectangles in this region, and then subtracting from this sum twice the sum of the lengths of all common boundaries The algorithm proceeds as follows:

```
Procedure conn_perimeter(conn_label: integer);
var
    perimeter: integer;
vector_var
    PER, COM: integer;
begin

 /* 1 Enable all PE's and initialize the two variables which store the
lengths of the perimeter and the common boundary of each rectangle. */

  PER = 0; COM = 0;

 /* 2 Enable only PE's that belong to the region for which the
perimeter is to be computed In all enabled PE's, set the variable PER
equal to the perimeter of the rectangle held by the PE. Compute the
total length of the common boundaries, and store it in the variable
COM */

  where COMP_LABEL = conn_label do
    PER = 2 * (XSIDE + YSIDE)
   elsewhere PER = 0,
  where COMP_LABEL = conn_label do
    begin
      where (TN = true) do COM = COM + XSIDE,
      where (TS = true) do COM = COM + XSIDE,
      where (TE = true) do COM = COM + YSIDE,
      where (TW = true) do COM = COM + YSIDE,
    end;

 /* 3 Enable all PE's and compute the perimeter. */

  perimeter = add_tree(PER) - 2 * add_tree(COM);
end.
```

Steps 1 and 2 execute 10 NON-VON 3 instructions independent of the image or tree size. Step 3 executes in time proportional to the running time of the function add_tree. Thus, the algorithm executes in $O(h)$ time. The execution time of the NON-VON 3 code for this procedures is dominated by the time to execute the function add_tree, which is equal to 0.3 msec.

6.2.3. Moments

There are many shape descriptors that can be derived from image moments. The set of moments of a bounded discrete function $f(x,y)$ of two variables x,y is defined by

$$M_{ij} = \sum x^i y^j f(x,y) \tag{6.1}$$

The parameter $i + j$ is called the ***order of the moment***, where i and j take on all nonnegative integer values. There is an infinite set of moments for every function. This infinite set is unique for every function, and is sufficient to specify the function completely [Cast 79].

For a binary image, $f(x,y)$ takes the value 1 inside the objects and 0 elsewhere. This function reflects the shape of the object, and it has a unique set of moments. Notice that the zero-order moment corresponds to the area of the object. The two first order moments, M_{10} and M_{01}, divided by the area of the object (M_{00}) correspond to the coordinates of the center of gravity (centroid). If an object moment is divided by the object area, the resulting value is size-invariant.

The *central moments* μ_{ij} of an object are defined by the following equation:

$$\mu_{ij} = \sum (x_0-x)^i(y_0-y)^j f(x,y) \tag{6.2}$$

where x_0 and y_0 are the center of gravity. The central moments are position-invariant. If the second central moment μ_{11} computed relative to the coordinate axes X',Y' is equal to zero, then these axes are called the *principal axes*. Moments computed relative to these axes are rotation-invariant. We can conclude from past definitions that object area-normalized moments computed relative to the principal axes can be used to describe uniquely the shape of an object, independent of its size, translation, or rotation. The set of such moments necessary to describe uniquely an object is object-dependent.

To compute the moments of an object in a binary image, the vector integer variable MOMENT is initialized to zero in all PE's. PE's associated with pixels belonging to this object are then enabled. The moment of each of these pixels is computed in the enabled PE's using the address information stored at each PE (x-address, y-address), The computed moment is stored in the vector integer variable MOMENT. The function add_tree(MOMENT) is next called to compute the object moment. To compute central moments, first the coordinates of center of gravity are broadcast to all PE's. Each enabled PE computes the central moment of the pixel held by this PE. The function add_tree is then used to compute the object central moment. In what follows, we describe an algorithm that computes the central moment μ_{11} of a binary image, which is defined by:

$$\mu_{11} = \sum (x-x_0)(y-y_0)f(x,y) \tag{6.3}$$

```
Procedure moment_11(conn_label, x0, y0: integer);
var
   moment_value: integer;
vector_var
    MOMENT, X0, Y0: integer;
    LEAF: boolean;
begin

 /* 1. Enable all PE's, and set the local variable MOMENT equal to
zero in all of them. */

  MOMENT = 0;

 /* 2. Enable only those leaf PE's whose label is equal to the label of
the connected component for which the area is to be computed.
Broadcast the values of the center of gravity (x0, y0). In all enabled
PE's, compute the central moment for the rectangle associated with this
PE. */

  N_BROADCAST8(x0, X0);
  N_BROADCAST8(y0, Y0);
  mark_leaf(LEAF);
  where (COMP_LABEL = conn_label) and (LEAF = true) do
    MOMENT = ((XADD - X0) * (YADD - Y0));

 /* 3. Call the function add_tree(MOMENT). When the function is
executed, the moment will be reported to the CP. */
```

```
  moment_value = add_tree(MOMENT);
end;
```

Steps 1 and 2 execute a fixed number of NON-VON instructions, while the execution time of Step 3 is proportional to the height of the tree. Thus, the time required to compute image moments is proportional to the tree height ($O(h)$). As in the case of area and perimeter computation, the time required to compute the function add_tree dominates the time required to execute this procedure (approximately 0.15 msec to compute a moment value).

6.2.4. Centroid

The coordinates of the centroid of an object are defined as the first order moments of the object divided by its area. To compute the two first order moments two values are computed in each rectangle belonging to the region we are interested in. The first, YMOM, is the result of multiplying the rectangle area by the sum of its x-address and half its x-side. The second, XMOM, is the product of multiplying the rectangle area by the sum of its y-address and half its y-side. The centroid is a pair consisting of add_tree(XMOM) divided by the area of the region, and add_tree(YMOM) divided by the area of the region. The area can be computed as described earlier in this section. Note that multiplication can again be performed using a series of simple shifts and additions, which takes a constant time. Thus, the time required to compute the centroid is proportional to the height of the tree, and is approximately 0.30 msec on NON-VON 3.

6.2.5. Compactness

The *compactness* (or *circularity*) of an object is one measure of the complexity of the shape of its boundary. The most commonly used measure of compactness is perimeter^2/area, which is a dimensionless quantity that is minimized by a circular area. Perimeter and area are computed as described before in time proportional to the height of the tree. Thus this measure of compactness is computed in time proportional to the height of the tree. There are other measures of compactness; the reader is referred to [Cast 79] for more details.

6.2.6. Eccentricity

Eccentricity is another characteristic of objects. Also referred to as *rectangularity*, it measures the elongation of an object. There are several measures of eccentricity. One of them is the ratio A/B where A is the maximum chord of the object, and B is the maximum chord perpendicular to it. Another possible measure is the ratio of the ***principal axes of inertia***. The principle axes of inertia for an object are the two orthogonal axes that pass through the center of gravity, such that one of the values of the two moments μ_{20} and μ_{02} computed relative to them is maximum, and the other value is minimum. One formula that approximates this ratio [Ball 82] is

$$E=\{(\mu_{20} - \mu_{02})^{1/2} + 4\mu_{11}\}/\text{area} \tag{6 4}$$

Again the time required to compute E is proportional to the time required to compute the moments in the above equation, which is $O(h)$. This time is approximately equal to 0.45 msec on NON-VON 3.

6.2.7. Euler Number

The *Euler number* (*genus*) of an image is a topological property that describes the connectedness of a region. The Euler number of an image is defined as the number of connected components minus the number of "holes" in the image. If there is only one connected component under consideration, then one minus the Euler number gives the number of holes in this connected component.

For a binary image, the Euler number may be computed from the expression

$$E = V - E + F \tag{6 5}$$

where V is the number of 1's in the image, E is the number of horizontally or vertically adjacent pairs of 1's, and F is the number of 2 x 2 blocks of 1's [Dyer 80b]. Similarly, Dyer [Dyer 80b] has shown that the Euler number for a binary image represented as a quadtree (the proof is similar for a binary image tree) can be computed from the expression

$$E = B - A + S \tag{6.6}$$

where B is the number of black rectangles, A is the number of adjacent pairs of black rectangles, and S is the number of triples or quadruples of black rectangles that surround a point. Figure 6-4-a shows how three rectangles can surround a node, while Figure 6-4-b show four rectangles surrounding a point.

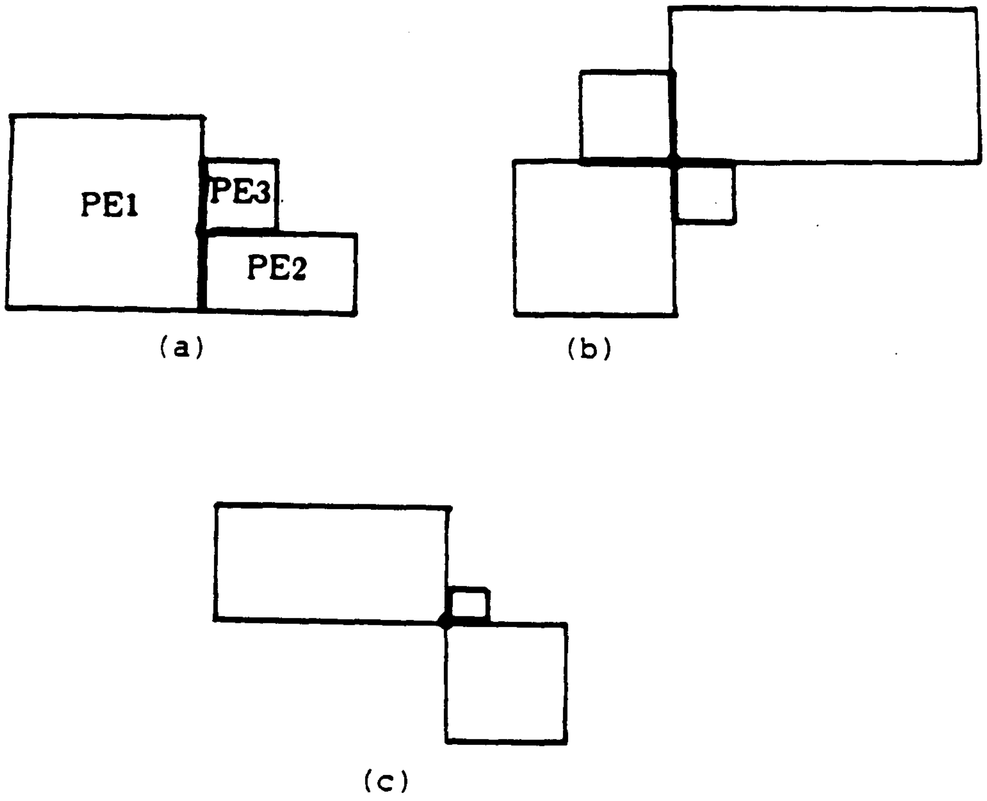


Figure 6-4: Possible Configurations of Three or Four Nodes That Intersect a Point

In computing the Euler number, the number of black rectangles B can be computed by setting a variable TEMP equal to one in all PE's associated with black rectangles that are part of the component under consideration, and zero in all other PE's. The function add_tree(TEMP) is then called to compute the value of B. Counting the number of adjacent rectangles is performed in a similar way using the common boundary information variables (TE, TN, TW, and TS). As described in Section 6.1, if there are two adjacent rectangles, then only one of them sets the appropriate common boundary variable equal to 1. The variable TEMP is set equal to the number of common boundary variables that are equal to one in each PE representing a rectangle of the component. Next, the function add_tree(TEMP) is called to compute A.

To compute S, all the points surrounded by three or four black rectangles are examined, one at a time. These points can be located using the common boundary information stored at each PE as follows. By inspecting Figure 6-4-a, if there are three rectangles surrounding a point, then there are three common boundary variables stored in the three PE's representing these rectangles. Due to the way these variables are set in the labeling procedure, the first PE of the three to report its rectangle (depending on their sizes) has none of these common boundary variables set equal to 1. Consequently, one of the other two PE's has two of these common boundary variables set (PE 3 in Figure 6-4-a). Note also that these two common boundaries intersect at one of the rectangle corners. Following a similar argument, it is easy to show that in the case of four rectangles surrounding a point, one or two of the PE's have two common boundary variables set (Figure 6-4-b). It is possible, however, to have three rectangles intersecting at a point without surrounding it, as shown in Figure 6-4-c. To compute *S*, all PE's with two common boundary variables set equal to 1, and which correspond to a corner in the rectangle, are examined. The corner point is counted only if it is surrounded on all sides by black pixels. If there is another PE with two common boundary variables that intersect at the same point, then it is flagged and not counted.

For example, if a rectangle has a common boundary along its north and west boundaries (that is, if both TW and are TN are set), the point at the north western corner of this rectangle is surrounded by two rectangles as shown in Figure 6-5.

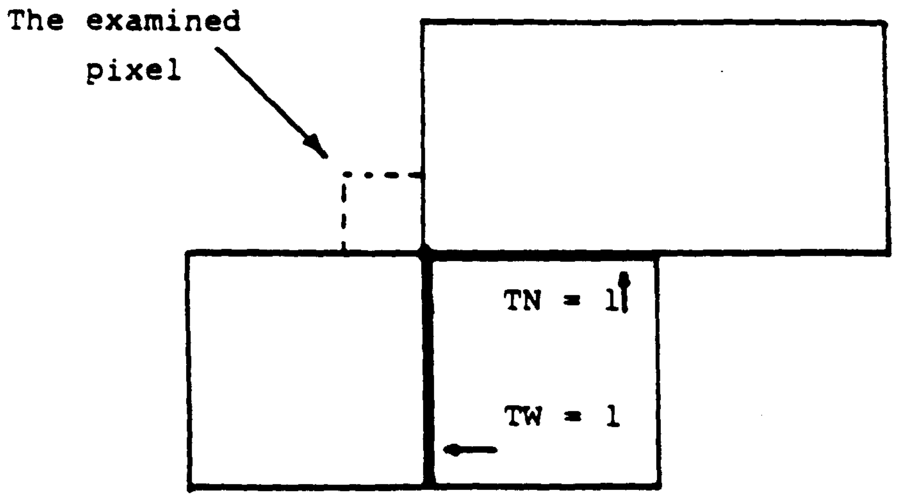


Figure 6-5: Testing for *S* Points

This corner point is examined to check whether there are other rectangles surrounding it. The check is performed by examining the neighboring image point in the north western direction. If this point is black, then we increment the value of S. The corner of the rectangle containing this point is flagged so that it will not be examined later by the algorithm.

The N-PASCAL algorithm follows:

```
var
   x, y, xs, ys: integer;
Procedure conn_euler(conn_label: integer);
var
   b, a, s, euler_no: integer;
vector_var
   TEMP: integer;
   NW, NE, SW, SE, LEAF: boolean;
begin

 /* 1. Compute the value of B in the Euler number formula by setting
 the variable TEMP equal to 1 in all PE's associated with rectangles of
 the component, and then counting the number of 1's. */

  TEMP := 0;
  where COMP_LABEL = conn_label do TEMP := 1;
  b := add_tree(TEMP);

 /* 2. Compute the value of A in the Euler number equation as in Step
 1. This time, set the variable TEMP equal to the number of common
 boundary variables equal to 1 in the component PE's. */

  TEMP := 0;
  where COMP_LABEL = conn_label do
    begin
      if TE = true then TEMP := TEMP + 1;
      if TN = true then TEMP := TEMP + 1;
      if TW = true then TEMP := TEMP + 1;
      if TS = true then TEMP := TEMP + 1;
    end;
  a := add_tree(TEMP);
```

/* 3. Set the variables corresponding to rectangle corner pixels surrounded by triples or quadruples of rectangles. Mark all leaf PE's corresponding to black pixels.*/

```
  NW := false; NE := false;
  SW := false; SE := false;
  TEMP:=0;
  where (TN = true) and (TW = true)
           and (COMP_LABEL = conn_label)
    do NW := true;
  where (TN = true) and (TE = true)
           and (COMP_LABEL = conn_label)
    do NE := true;
  where (TW = true) and (TS = true)
           and (COMP_LABEL = conn_label)
    do SW := true;
  where (TE = true) and (TS = true)
           and (COMP_LABEL = conn_label)
    do SE := true;
  mark_leaf(LEAF);
  LEAF := LEAF and BINARY;
```

/* 4 For each corner, report the size and address information of the rectangle containing it. The function check_euler(D) checks for any PE with the boolean variable D set equal to 1 if there exists a black pixel which surrounds the corner from the only left direction. If none exists, then a zero value is returned, otherwise a single PE with D equal to 1 is enabled, and its address and size information are reported. Also, this step checks to determine whether a black pixel exists diagonally across the corner being examined. */

```
  TEMP := 0;
  while check_euler(NW) <> 0 do
    begin
      where (XADD + XSIDE = x) and (YADD + YSIDE = y)
        do SE := false;
      where (XADD = x - 1) and (YADD = y - 1)
               and (LEAF = true)  do TEMP := 1;
      end;

    while check_euler(NE) <> 0 do
      begin
        where (XADD = $x + xs$) and (YADD + YSIDE = $y$)
          do SW := false;
        where (XADD = $x + xs$) and (YADD = $y$ - 1)
                   and (LEAF = true)  do TEMP := 1;
      end;

    while check_euler(SW) <> 0 do
      begin
        where (XADD + XSIDE = $x$) and (YADD = $y + ys$)
          do SE := false;
        where (XADD = $x$ - 1) and (YADD = $y + ys$)
                   and (LEAF = true)  do TEMP := 1;
      end;

    while check_euler(SE) <> 0 do
      begin
        where (XADD = $x + xs$) and (YADD = $y + ys$)
          do SE := false;
        where (XADD = $x + xs$) and (YADD = $y$ - $ys$)
                   and (LEAF = true)  do TEMP := 1;
      end;

    $s$ := add_tree(TEMP);

  /* 5 Compute the Euler number for the connected component. */

    euler_no := $b$ - $a$ + $s$;
  end
```

/* The following procedure checks to see whether there are any PE's with the boolean variable D equal to 1 If so, one such PE is enabled and the address and size information associated with the rectangle it represents are reported to the CP If there are no PE's with D equal to 1, then the function returns 0 */

```
function check_euler(var D: boolean) integer;
label 1;
```

```
begin
  where D = true do N_A1 := true
   elsewhere N_A1 := false;
  if N_RESOLVE(N_A1) = 0 then
    check_euler := 0
   else
     begin
       where N_A1 = true do
         begin
           N_REPORT8(XADD, x);
           N_REPORT8(YADD, y);
           N_REPORT8(XSIDE, xs);
           N_REPORT8(YSIDE, ys);
           D := false;
           check_euler := 1;
         end;
     end;
end;
```

Steps 1 and 2 of this algorithm execute in time proportional to the height of the tree. Step 4 executes in time proportional to the number of points of intersection of three or four rectangles. This number is always less than the number of black rectangles. Thus, in the worst case, the algorithm executes in $O(h+b)$, where h is the height of the tree and b is the number of black rectangles. Simulation results for some simple binary images show that the value of S on the average is approximately equal to half the number of black rectangles.

6.2.8. Connected Component Properties Simulation

The algorithms described in the previous section have been tested using the functional simulator. Six geometric properties have been computed for each component in the binary image. The output of the simulator resulting from computing these properties of the connected components of Figure 6-3 follows.

```
Computing some geometric properties for the labeled foreground objects:


label[0]= 9 , area[0]=  67 , perimeter[0]=  62
x-center[0]=  4.32 , y-center[0]= 24.10 , No. of holes[0]= 1
compactness[0]= 57.37 , elongation[0]=  8.30

label[1]= 2 , area[1]=   5 , perimeter[1]=  10
x-center[1]=  0.90 , y-center[1]= -0.50 , No. of holes[1]= 0
compactness[1]= 20.00 , elongation[1]= -0.48

label[2]= 4 , area[2]=  49 , perimeter[2]=  48
x-center[2]= 21.66 , y-center[2]=  7.07 , No. of holes[2]= 0
compactness[2]= 47.02 , elongation[2]= 13.78

label[3]= 7 , area[3]=  20 , perimeter[3]=  24
x-center[3]=  9.50 , y-center[3]= 18.05 , No. of holes[3]= 0
compactness[3]= 28.80 , elongation[3]=  5.19

label[4]=10 , area[4]=   2 , perimeter[4]=   6
x-center[4]=  0.50 , y-center[4]= 16.00 , No. of holes[4]= 0
compactness[4]= 18.00 , elongation[4]=  0.00


Computing some geometric properties for the labeled background objects:


label[0]=5 , area[0]= 874 , perimeter[0]= 232
x-center[0]= 16.93 , y-center[0]=  7.67 , No. of holes[0]= 3
compactness[0]= 61.58 , elongation[0]= 46.27
```

```
label[1]=3 , area[1]=    4 , perimeter[1]=   8
x-center[1]=  5.00 , y-center[1]= 25.00 , No. of holes[1]= 0
compactness[1]= 16.00 , elongation[1]=  0.00

label[2]=6 , area[2]=    3 , perimeter[2]=   8
x-center[2]=  7.50 , y-center[2]= 26.50 , No. of holes[2]= 0
compactness[2]= 21.33 , elongation[2]=  0.47
```

Note that the perimeter of components containing holes includes the perimeter of these holes. The existence of holes also increases the compactness measure of objects, since it increases the length of the object perimeter. Performance comparison with other highly parallel machines is not possible, since performance data for geometric algorithms on these machines are not available as of the time of writing this thesis. It is expected that NON-VON's performance for computing the geometric properties is an order of magnitude better than the mesh-connected machines because of the hierarchical nature of NON-VON.

6.3. Set Operations

Set operations on images involve the computation of a new image based on one or more existing images. For example, locating common objects between two images in the same relative position involves intersecting the two images point by point and determining which points are common to both of them. In the following subsections, we will present NON-VON algorithms for computing the complement, intersection, and union of binary images. Other set operations can be expressed as a combination of these three set operations. We assume that the binary images have already been loaded in the leaf PE's of the NON-VON tree, and that the binary image representation has been built. In this representation, the character vector variable **FQUAD** contains the type of the rectangle associated with the PE, and the vector integer variable **TREE** indicates how many black pixels exist in this rectangle. Also, the vector boolean variable **BINARY** stores the value of the pixel in a binary image.

6.3.1. Complement

Computing the complement of a binary image involves changing the black pixels into white and the white pixels into black. The most obvious manner in which the complement operation involves computing the complement of all pixels of the original binary image stored in the leaf PE's. This operation is executed concurrently in all leaf PE's, and involves reading the pixel value and then complementing it. If the binary image tree representation of the complement is required for subsequent processing stages, the algorithm for building the binary image tree representation described in Chapter 4 can be executed. This algorithm executes in $O(h)$ time. A more efficient method to build the binary image tree representation of the complement image uses the binary image tree representation of the original image. This method can compute the binary image tree representation of the complement in constant time. The algorithm for complementing an image and constructing its binary image tree follows.

```
Procedure set_comp;
vector_var
    TREE1: integer;
    FQUAD1: char;
begin

 /* 1. The FQUAD variable is checked. If it is equal to 'B', then it is
changed to 'W', and vice versa. If the value is 'G' it remains as it is.
FQUAD1 and TREE1 are the image complement variables. FQUAD and
TREE are globally defined. */

  if FQUAD = 'B' then FQUAD1 := 'W'
   else if FQUAD = 'W' then FQUAD1 := 'B'
    else if FQUAD = 'G' then FQUAD1 := 'G'
     else FQUAD1 := 'N';

 /* 2. The variable TREE (denoting the number of black pixels in the
rectangle) is set equal to the size of the rectangle minus its old value. */

  TREE1 := XSIDE * YSIDE - TREE;

end;
```

Steps 1 and 2 take a fixed number of NON-VON instructions (about 32 NON-VON 3 instructions) regardless of the NON-VON tree size, so the algorithm will be executed in a constant time (8 μsec on NON-VON 3).

6.3.2. Intersection

The intersection of two binary images involves a pixel-wise logical conjunction of the two images. We assume that the two binary images are stored in the NON-VON tree. $BINARY_1$, $TREE_1$ and $FQUAD_1$ represent the first image, while $BINARY_2$, $TREE_2$ and $FQUAD_2$ represent the second image. The two images in their finest resolution (the pixel level) are stored in the leaf PE's. The intersection algorithm follows

```
Procedure set_int;
vector_var
    FQUAD3: char;
    LEAF, BINARY3: boolean;
begin
 /* 1. Enable all leaf PE's. Compare the two variables BINARY1 and
 BINARY2, and if they are both equal to 1, set the variable BINARY3
 equal to 1; otherwise set BINARY3 equal to 0. This comparison reduces
 to logical conjunction. BINARY1 and BINARY2 are globally defined. */

  mark_leaf(LEAF);
  where LEAF = true do BINARY3 := BINARY1 and BINARY2;

 /* 2. In the case of binary image tree representation is needed, set the
 variable FQUAD3 to either the value 'B' or the value 'W', depending on
 the value of BINARY3. Call the procedure build_bit() to build the
 binary image tree representation for the result image. */

  where BINARY3 = true do FQUAD3 := 'B'
    elsewhere FQUAD3 := 'W';
  build_binimg(h);
end.
```

Execution of the intersection algorithm on the two original images takes a fixed number of instructions, independent of image size. Building the binary image tree takes time proportional to the height of the tree, as described in Chapter 4.

Image intersection can also be defined for gray-scale images. Gray-scale intersection is performed by comparing each pixel in the first image with the corresponding pixel in the second image. If the intensities are equal, the corresponding pixel in the result image is set to the common value, otherwise it is set to zero

6.3.3. Union

Computation of the union of two binary images involves the pixel-wise disjunction of the two input images The algorithm for the union of two binary images is analogous to the one for intersection. The difference is that the logical operation performed in Step 1 of the algorithm is the disjunction of the two pixels instead of the conjunction. The N-PASCAL procedure to perform the union of two binary images follows:

```
Procedure set_union;
vector_var
    FQUAD3: char;
    LEAF, BINARY3: boolean;
begin

 /* 1 Enable all leaf PE's. Compare the two variables BINARY1 and
BINARY2, and if one of them is equal to 1, set the variable BINARY3
equal to 1, otherwise set BINARY3 equal to 0. This comparison reduces
to logical disjunction. BINARY1 and BINARY2 are globally defined. */

  mark_leaf(LEAF);
  where LEAF = true do BINARY3 := BINARY1 or BINARY2;

 /* 2. To build the binary image tree representation, set the variable
FQUAD3 to either the value 'B' or the value 'W', depending on the
value of BINARY3. Call the procedure build_bit() to build the binary
image tree representation for the result image. */

  where BINARY3 = true do FQUAD3 := 'B'
    elsewhere FQUAD3 := 'W';
  build_binimg(h);
```

end,

The time analysis for the union algorithm is therefore similar to that of the intersection algorithm.

Chapter 7
The Hough Transform

The Hough transform method is used frequently in image understanding tasks for among other uses to detect the shape of object boundaries described by parametric curves. This method is based on the duality between points on a curve and the parameters of that curve. In his initial work, Hough [Houg 62] described a method for detecting straight lines in an image using the slope-intercept parameterization of the line. According to this parameterization, the line equation is expressed as:

$$y = mx + c \tag{7.1}$$

Suppose that we have a set of image points $\{(x_1,y_1), \ldots, (x_n,y_n)\}$ that have a likelihood of being on linear boundaries. In this paper, we refer to these points as ***boundary points***. The Hough transform method organizes the boundary points into a set of straight lines as follows. Consider a boundary point (x_i,y_i) in the image plane. The parameters of all lines passing through this point must satisfy the equation:

$$y_i = mx_i + c$$

This equation corresponds to a straight line in the ***m-c*** space (***the parameter space***). Thus, the set of boundary points in the image plane corresponds to a set of lines in the m-c plane. If two boundary points are on a line AB in the image plane with parameters m_1 and c_1, then the two lines corresponding to these two points in the m-c plane intersect at the point (m_1,c_1). In fact, all boundary points in the image plane on the same line AB map to lines in the m-c plane that intersect at the point (m_1,c_1). Thus, the problem of finding the set of lines in the image plane is reduced to that of finding common points of intersection of lines in the parameter space. A better parameterization of a straight line is suggested by Duda [Duda 72], in which the parameters θ and ρ are used, where θ is the angle of the line normal and ρ is the algebraic distance from the origin. The advantage of this parameterization is that the values of θ and ρ are bounded, while in the case of m-c parameterization the values are not bounded. The Hough transform can be extended to detect other curves of analytical parameters [Ball 75], or to detect general curve shapes using edge orientation at the image points and a reference point [Ball 81]. A memory efficient implementation of the Hough transform on sequential machines is described in [Brow 84]. A parallel algorithm based on the Hough transform for detecting a general curve with specific orientation has been developed by Merlin et al [Merl 75].

The implementation of the Hough transform for detecting straight lines on a sequential machine involves a quantization of the parameter plane into a quadruled grid. The grid size is determined by the acceptable errors in the parameter values, and the quantization is confined to a specific region of the parameter plane determined by the range of parameter values A two-dimensional array (***the accumulator array***) is then used to represent the parameter plane grid, where each array entry corresponds to a grid cell. For each boundary point, the algorithm on a sequential machine increments the counts in all accumulator array entries that correspond to grid cells along the straight line in the parameter plane. After this step, grid cells corresponding to the accumulator array entries where the count exceeds a certain threshold value are selected as the set of parameters for the image straight lines being sought The increment of accumulator array counts can be thought of as a process of "voting" by the boundary points for the parameter values of possible curves passing through these points The time required to execute this algorithm on a sequential machine is proportional to the size s of the grid. plus the number m of boundary points times the number of votes v of each point ($O(s+mv)$) Memory space required is proportional to the size of the grid

In what follows, we describe two parallel algorithms to implement the Hough transform on NON-VON The first one is a direct parallel implementation of the standard sequential algorithm. The disadvantages of this approach are presented, and we describe a second approach that solves these problems We assume that the boundary points have been detected by some other procedures and that the PE's holding them are marked using a special flag. Without loss of generality, we also assume that the curves being sought are straight lines whose equations are expressed using the slope and intercept parameters.

7.1. The Hough Transform Algorithm - A Direct Approach

In the sequential machine implementation of the Hough transform, each boundary point casts its votes in the accumulator array by incrementing all the entries corresponding to grid cells along the parameter space curve associated with this point. This process is repeated for all image boundary points. Next, accumulator array entries whose count exceeds a specified threshold value are selected. We now describe how this algorithm is implemented on NON-VON.

Each NON-VON PE is associated with a grid cell in the parameter space. The procedure to perform this is very simple, and it executes in time proportional to the tree height. The first step is to enumerate the NON-VON tree PE's using the inorder enumeration described in [Knut 74]. (Figure 3-3 illustrates such an ordering.) The number assigned to each PE is stored in the vector integer variable **ADDR**. If the **parameter** space is *m* by *c*, then the address of the grid cell held by each **PE** is the pair (M, C) resulting from computing the remainder and the quotient of dividing **ADDR** by *m*. The **N-PASCAL** procedure to perform this follows:

Procedure ***para_space(no_levels*** : **integer),**

```
var
  i: integer;
vector_var
  M, C, ADDR: integer;
  LC, RC, ROOT: boolean;
begin
```

/* 1. Enumerate the PE's using the inorder enumeration. We assume that left and right children are marked and that the marking result is stored in the LC and RC variables, respectively. Also ROOT is assumed to be true only in the root of the tree. *x_side* is the length of the image side. */

```
 ADDR := (x_side * x_side) div 2;
 for i := 2 to no_levels do
 where ROOT = false do
   begin
     N_RECV8(P, ADDR, ADDR);
     where LC = true do
       ADDR := ADDR div 2;
     where RC = true do
       ADDR := ADDR + (ADDR div 2);
   end;
```

/* 2. M, and C are now computed. */

```
  M := ADDR mod m;
  C := ADDR div m;

end;
```

This procedures executes in time proportional to the tree height. Note also that dividing by 2 in this procedure is equivalent to shifting the binary representation of the number one position to the left.

A vector integer variable COUNT is initialized to zero in all PE's before starting the algorithm. The coordinates of boundary points are then reported to the CP one point at a time using the RESOLVE instruction. The coordinates of each reported point are then broadcast to all PE's and all those PE's holding a grid cell across the curve in the parameter space corresponding to the reported point increment the vector variable COUNT by one. This step is performed by substituting the broadcast values in the parameter space curve equation and if it satisfies the equation then COUNT is incremented. Each PE whose COUNT variable exceeds the threshold value is marked, and the value of the grid cell associated with it is reported to the CP using the RESOLVE and REPORT instructions A vector character variable HT is used to flag those boundary points that have not voted yet. The NON-VON PASCAL algorithm that describes the procedure follows .

```
Procedure hough1(thresh integer),
    label 2, 4, 5,
var
  x. y, m, c: integer,
vector_var
  COUNT. X. Y integer.
  PARAMETER boolean,
begin

 /* 1 Initialize the vector variable COUNT in all PE's. The other
vector variables M, C, and HT are assumed to be defined and initialized
by the calling procedure */
```

```
COUNT := 0;
PARAMETER := false;
```

/* 2. Enable all PE's in which the boundary points have not been reported yet. Report the coordinates of a single boundary point using the RESOLVE instruction, and mark this point as reported. If none is enabled then all the boundary points have been reported. In this case start computing the parameter values using the threshold value. */

```
2:
where HT = true do N_A1 := true
 elsewhere N_A1 := false;
if N_RESOLVE(N_A1) = 0 then
     goto 4;
where N_A1 = true do
 begin
  HT := false;
  N_REPORT8(XADD, x);
  N_REPORT8(YADD, y);
 end;
```

/* 3. Enable all PE's holding the grid cells. Broadcast the reported image point value. Substitute this value in the equation of the parameter space curve. Increment COUNT in all PE's in which the equation is satisfied. Now loop to select another boundary point. */

```
X := x;
Y := y;
if Y = (M * X + C) then
         COUNT := COUNT + 1;
goto 2;
```

/* 4. Broadcast the threshold value *thresh*. Mark all PE's in which the count exceeds the threshold value. Alternatively, the user can be prompted to input the threshold value. */

```
4: where COUNT > thresh do
     PARAMETER := true;
```

/* 5. Report the grid cell values held by these enabled PE's one by one using the RESOLVE instruction. */

```
 5
 where PARAMETER = true do
     N_A1 := true
   elsewhere N_A1 := false;
 if N_RESOLVE(N_A1) <> 0 then
   begin
     where N_A1 = true do
       begin
         PARAMETER := false;
         N_REPORT8(M, m);
         N_REPORT8(C, c);
         goto 5;
       end;
   end;
end;
```

Steps 2 through 4 are executed a number of times equal to the number of boundary points *b*. Step 5 executes a number of times equal to the number of curves found, which is less than *b*. Thus, the algorithm takes time proportional to the number of image boundary points (O(*b*)). The NON-VON 3 code for this procedure [Ibra 84c] executes about 200 instructions for Steps 2 through 4 (50 μsec at 4 Mhz). Of these 200 NON-VON 3 instructions, approximately 160 instructions implement the evaluation of the straight line equation. (This number can be reduced significantly by implementing the hardware modification proposed in Chapter 5.) Step 5 executes about 12 NON-VON 3 instructions for each set of parameter values found. Thus, if the image contains 1000 boundary points, the execution time of the algorithm is approximately 53 msec

The number of PE's required by this approach is equal to the number of grid points. If the the grid size is larger than the machine size by a factor of k, then the parameter space is divided into k parts. The above procedure is then executed for each of these parts. The time required to execute the algorithm in this case is O(kb).

One disadvantage of this approach is that it requires a NON-VON machine of size comparable to the grid size, despite the fact that many of the PE's will never get their COUNT incremented. Note also that each time a boundary point is broadcast the curve equation has to be evaluated in each PE, which is a time consuming operation as clear from the numbers cited earlier. The second approach we describe below solves these problems. It uses a number of PE's equal to the number of votes cast by the boundary points rather than the grid size, and the curve equation is evaluated only once.

7.2. The Hough Transform Algorithm - A MSIMD Approach

In our improved approach, the NON-VON tree is treated as if it were an independent set of subtrees, and each boundary point casts its votes one by one in one of these subtrees. This voting process is performed concurrently in all the subtrees. Thus, in time proportional to the number of votes cast by each boundary point, all votes are cast and stored throughout the tree. The problem of finding the parameter values which exceed the threshold value is equivalent to that of finding the local peaks of a two-dimensional histogram in the *m-c* example. Because of the way the votes are cast in this second approach, we refer to this algorithm as a multiple-SIMD (MSIMD) algorithm.

The size of these subtrees is determined by the number of votes cast by each boundary point. For example, if each boundary point casts 60 votes, then the subtree size required is at least 60 (A subtree of six levels will suffice). Boundary points are stored in the roots of these subtrees. This can be performed by more than one method. The simplest one is to report the boundary points to the CP one by one using the RESOLVE instruction, and then to broadcast them to be stored in the roots of the subtrees

The PE's in these subtrees are enumerated in such a way that each PE in a subtree is assigned a unique address (stored in the vector variable ADDRESS) in the range [0, *max_num_votes*], where *max_num_votes* is the value of the maximum number of votes casted by each point. The enumeration is performed in such a way that all PE's in the same relative position within these subtrees have the same address. This enumeration procedure is similar to the enumeration procedure described in the previous section, except that the number assigned to each PE is the remainder of the computed address divided by the subtree size. The remainder operation actually can be performed by extracting the right least significant number of bits from the address computed by the enumeration procedure. Time required by this procedure is proportional to the height of the subtree. We now describe the algorithm for storing the votes in the NON-VON tree.

We assume that the boundary points reside in the roots of the subtrees, with the PE's being enumerated as described earlier. We also assume that the parameter space is a two-dimensional one. The vector integer variables X and Y are used to store the value of the boundary points, while the vector variables M1 and M2 are used to store the parameter values voted for by the boundary points. A scalar variable ***g_m1*** stores the value of parameter M1 to be broadcast, and the scalar constant ***delta_m1*** is the increment used to change the value of ***g_m1***. The scalar constant ***h_subtree*** is the height of the subtree. The N-PASCAL voting procedure follows:

```
Procedure hough2;
    label 3,
var
    i, j, g_m1: integer;
vector_var
    M1, M2, X, Y: integer,
begin

 /* 1. Initialize the scalar variables. The scalar variables h_subtree,
 delta_m1, max_num_votes are initialized by the calling procedure. */

  i = 0,
  g_m1 = 0,
```

2 Enable all PE's that are not the root of some voting subtree. The
ble SUBTREE_ROOT is assumed to be set by the calling
dure. Set X and Y in each child equal to X and Y in its parent
at this step ***h_subtree*** times. */

```
re SUBTREE_ROOT = false do
gin
 _RECV8(P, XADD, X);
 _RECV8(P, YADD, Y);
or j = 1 to h_subtree-1 do
  begin
    N_RECV8(P, X, X);
    N_RECV8(P, Y, Y);
  end;
l;
```

 Enable the PE's with ADDRESS equal to *i* in voting subtrees. In
e enabled PE's store the new value of the parameter M1. */

```
ere ADDRESS = i do
l1 = g_m1;
```

 Increment ***g_m1*** by ***delta_m1***, and increment ***i*** by 1. If all the
 of M1 have been stored in the voting PE's, then proceed to
ute the value of M2 in those PE's, otherwise repeat step 3. */

```
= i + 1;
m1 = g_m1 + delta_m1;
i < max_num_votes then
    goto 3;
```

 Enable all PE's. Using the values of X, Y, and M1, compute M2
he curve equation. */

```
 = compute_m2(X , Y , M1);
```

Step 2 is executed a number of times equal to the subtree height (log v), where v is equal to the number of votes cast by each point. Steps 3 and 4 are executed a number of times equal to v. Thus, the procedure to store the votes in the subtree takes time of O(v). Note that step 5, the evaluation of the curve equation, is executed only one time. If the evaluation of the curve equation results in more than one M2 value for each value of M1, then each PE stores more than one parameter set values. This case depends on the parameter space curve, and should result in a slightly modified version of the algorithm to compute the local peaks of the parameter histogram described later. If substituting the known values in the curve equation results in a non-solvable equation in the parameter being sought, then one possible way to overcome this problem is by keeping a table of the parameter values and corresponding function values in the CP. The CP broadcasts these pairs of parameter and function values for all PE's and only PE's holding similar function values (maybe within a small range) set the value of their parameter variable equal to the broadcast parameter value. This process takes time proportional to the length of the table, but it is executed only one time in this second approach to Hough transform method.

The NON-VON 3 code for this procedure executes approximately (10v + 160) NON-VON 3 instructions. For v equal to 100, the time required to cast the votes in the tree is thus about 0.3 msec. If there are more votes than the NON-VON tree size, each PE stores more than one vote.

In this case, if each PE stores k values, then the time required to execute the above procedure is $O(kv)$, where k is the ratio between the total number of votes and the NON-VON tree size.

Next, we describe how to find the parameter values that have votes exceeding the threshold value. These values occur at the the local peaks of the two dimensional histogram of the votes for M1 and M2. We assume in the following discussion that there are few of these local peaks. This is a realistic assumption, as the number of curves being sought is usually small.

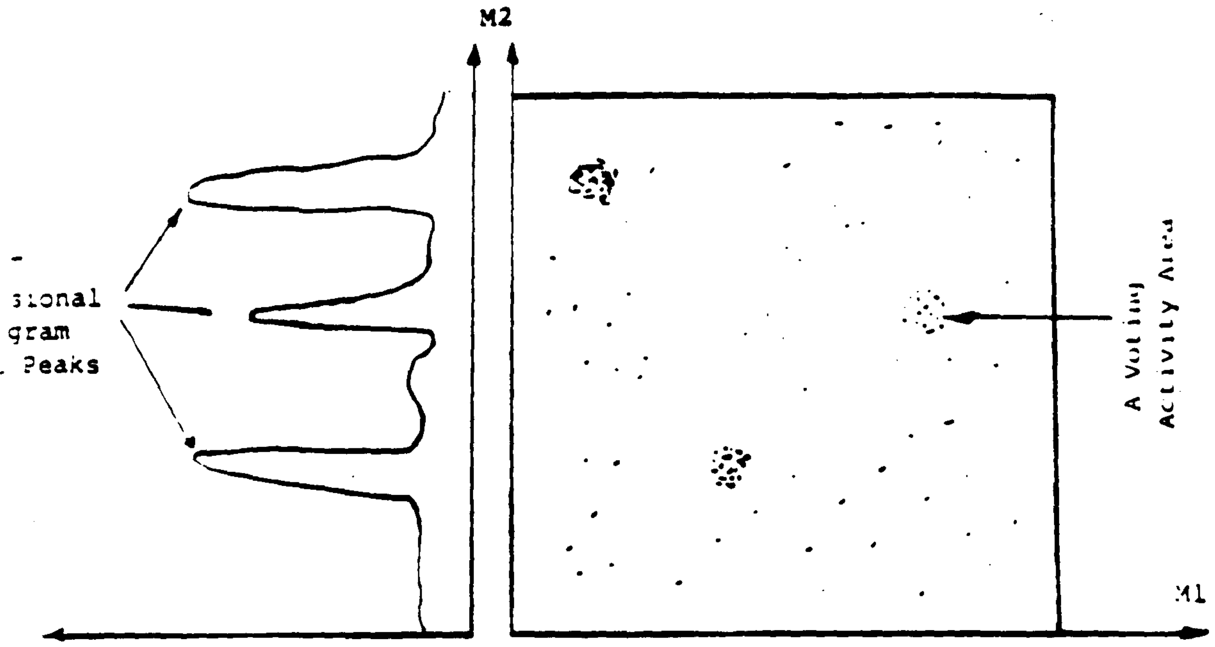


Figure 7-1: The Two-Dimensional Histogram of Parameter Values

Figure 7-1 shows such a histogram. In this example, there are a few areas of voting activity (local peaks). A direct approach to the identification of these local peaks involves dividing the two dimensional histogram space into grid cells. For each grid cell, all PE's with M1 and M2 values falling within this grid cell are then marked and counted. The time required to execute this simple procedure is O(sh), where s is the grid size and h is the NON-VON tree height. Counts that exceed the threshold value are the parameter values being sought. A large percentage of the time in this procedure is spent counting votes in grid cells corresponding to areas that contain few votes.

A different approach, in which areas of non voting activity are not considered in locating the local peaks of the two-dimensional histogram, is described now. The procedure first computes a one-dimensional histogram of the parameter M2, as shown in Figure 7-1. (A pipelined-SIMD algorithm to compute the one-dimensional histogram is described in Chapter 5.) A small number of local peaks corresponding to regions of the two-dimensional histogram where most of the votes occur, appear in the one-dimensional histogram. Only votes in those regions are then marked. A second one-dimensional histogram of the parameter M1 is then computed for the marked votes only. The local peaks of this histogram are the values of M1, for which there are local peaks in the two-dimensional histogram. The values of M1 and M2 for which exist local peaks of the two one-dimensional histograms mark the regions of activity in the two-dimensional histogram. These regions of active voting are then checked for exact vote counts. Round off errors in computing the parameter values can result in peaks that are relatively flat. For this reason, a small window around the regions of voting activity should also be checked when counting the exact votes. This second approach executes in time of O($m1$ + $m2$ + h), where $m1$ and $m2$ are the number of bins in the two one-dimensional histograms. The computation of a 64 bins one-dimensional histogram requires about one msec. The algorithm for locating the local peaks in the two-dimensional histogram of the parameter values as described earlier executes in about 5 msec. The total execution time of the second approach is thus about 5.3 msec, which is considerably less than the time required by the first approach (50 msec for 1000 boundary points).

The algorithms described here can be extended using slight modifications to deal with parameter spaces of higher dimensions. For example, in the first approach if we have an n-dimensional parameter space, then each PE will correspond to a n-dimensional grid cell in this space. In the second approach, the subtree size will correspond to that of (n-1)-dimensional area of the parameter space, and each PE will store parameter values that represent cells in this sub-parameter space. A second approach to extend the Hough transform to parameter spaces of higher dimensions involves applying the current algorithms to two-dimensional cross sections of the multi-dimensional parameter space.

7.3. Simulation Results

The two algorithms described in this chapter have been tested using the functional simulator Boundary points representing straight lines in a 32 x 32 binary image, as shown in Figure 7-2, have been input to the simulator. The parameter space grid is a 32 x 64 grid, with *m* taking the values -15 to 16 and *c* assuming the values -10 to 53. Appendix C includes the Hough transform simulation as performed on the functional simulator. Nine lines, each consisting of five or more points, have been found using the first approach. The two-dimensional accumulator array of these lines are shown in Figure 7-3-a.

In the second approach, 16 votes are cast in each subtree with *m* varying from -7 to 8 Figure 7-3-b depicts the two-dimensional histogram of the votes stored in the tree Appendix C contains the values of the two one-dimensional histograms computed for the stored votes. The second approach has computed the same set of straight lines found by the first approach.

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000011111111100000000000000
00000000100000001000000000000000
00000001000000010000000000000000
00000010000000100000000000000000
00000100000001000000000000000000
00000111111111000000000000000000
00000100000001000000000000000000
00000010000000100000000000000000
00000001000000010000000000000000
00000000111100111000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000001000000100000000000000
00000000000000000000000000000000
00000000100000010000000000000000
00000010000000000000000000000000
00000010000001000000000000000000
00000100000000000000000000000000
00001000000100000000000000000000
00001000000000000000000000000000
00100000010000000000000000000000
00010000000000000000000000000000
00000000100000000000000000000000
00100000000000000000000000000000
00000000000000000000000000000000
01000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Figure 7-2: The Input Boundary Points

Figure 7-8: Some Hough Transform Simulation Results

```
0000000000000000000000000000100000000000001001000000000000000010
0000000000000000000000000001000000000000010010000000000000010000
00000C0000000000000000000010000000000000100100000000000010000000
0000000000000000000000000100000000000001001000000000001000000000
0000000000000000000000001000000000000100100000000010000000000001
0000000000000000000000010000000000010010000000001000000000011000
0000000000000000000000100000000001001000000001000000000110000000
0000000000000000000001000000000100100000001000000000110000000100
0000000000000000000010000000010010000001000000001100000010000000
0000000000000000000100000001001000010000000011000000100000210011
0000000000000000001000000100100010000001100000100001110001110102
0000000000000000010000010010010000011000010010110000112111110200
0000000000000000100001001010001100011001100010122202031022011211
0000000000000001000100110001101100111000102222131212021112224111
0000000000000010010020021011110110011424221426021212121100000000
0000000000000101112121212020072228222290000000000000000000000000
0000000006001223621217232323250000000000000000000000000000000000
2111114232282221211121222100000000000000000000000000000000000000
2021132133112202021100000000000000000000000000000000000000000000
0221130212111110000000000C00000000000000000000000000000000000000
1211020131000000000000000000000000000000000000000000000000000000
0012101100000000000000000000000000000000000000000000000000000000
0100101000000000000000000000000000000000000000000000000000000000
0010010000000000000000000000000000000000000000000000000000000000
1000100000000000000000000000000000000000000000000000000000000000
0001000000000000000000000000000000000000000000000000000000000000
0010000000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

(a) The 32 X 64 Accumulator Array

```
0000000000000000000010000000010010000001000000001100000010000000
0000000000000000000100000001001000010000000011000000100000210011
0000000000000000001000000100100010000001100000100001110001110102
0000000000000000010000010010010000011000010010110000112111110200
0000000000000000100001001010001100011001100010122202031022011211
0000000000000001000100110001101100111000102222131212021112224111
0000000000000010010020021011110110011424221426021212121100000000
0000000000000101112121212020072228222290000000000000000000000000
0000000006001223621217232323250000000000000000000000000000000000
2111114232282221211121222100000000000000000000000000000000000000
2021132133112202021100000000000000000000000000000000000000000000
0221130212111110000000000000000000000000000000000000000000000000
1211020131000000000000000000000000000000000000000000000000000000
0012101100000000000000000000000000000000000000000000000000000000
0100101000000000000000000000000000000000000000000000000000000000
0010010000000000000000000000000000000000000000000000000000000000
```

(b) The 16 X 64 Two-Dimensional Histogram

Chapter 8
Moving Light Displays

In this chapter, we describe a NON-VON algorithm that implements the tracking step in systems that interpret the motion of jointed objects from a sequence of binary images representing points lying on the moving objects. The term ***moving light display*** (MLD) is used to refer to this kind of image motion [Rash 80]. An MLD system uses only information about the position and velocity of its points for the perception of motion, and a sequence of such binary images (frames) are required for the interpretation of the object motion. The objects in these frames are represented by a relatively small number of points (typically less than one hundred). Rashid [Rash 80] has implemented a system, which he calls Lights, that interprets simple MLD's. The input to this system is a set of coordinate pairs corresponding to the points of the MLD. In this chapter, we present an algorithm that implements the "tracking" step in Rashid's algorithm.

The tracking problem is concerned with determining the correspondence of points from one frame to the next. The only information known is the position of frame points depicting parts in relative motion, and the average velocity of these points based on previous frame information A fundamental assumption is that the velocity of MLD points vary "smoothly" from one frame to the next This assumption can be used to predict the position of the MLD points in the next frame The tracking algorithm computes the correspondence that minimizes the sum of differences between the expected position of each point and the actual position of the corresponding point in the next frame. Assume that the first frame contains m points and the next frame contains n points (Note that m and n may differ, since different points may be occluded in the two frames.) One approach is to try all the possible matches between the two frames. There are $O(m^n)$ such possible matches; this approach is thus prohibitively time-consuming. Rashid has proposed a second approach based on the observation that the actual corresponding point is usually one of the ones which are found relatively near the predicted position in the case of images produced by real physical objects.

We describe in the following section a NON-VON implementation of the tracking step, which is based on this heuristic.

8.1. The Tracking Algorithm

The NON-VON algorithm starts by computing a good approximate solution based on the heuristic mentioned earlier. This is performed by calculating for each point in the first frame, the point closest to its predicted position among the points that have not been selected yet in the second frame. (This approach to compute the initial solution is basically a greedy algorithm, where the best local match among the available ones is selected.)

This solution is then stored as an m-vector in the root PE of the NON-VON tree. If the number of the points in the first frame is larger than the number of the second frame points (some points in the second frame are occluded), then one approach to handle the inequality is to mark the correspondences of the extra points in the first frame with a special character and they are not considered in computing the sum of differences. However, for the sake of simplicity and to demonstrate how the actual computation is being performed, we assume in this chapter that the number of points in both frames is equal.

The points of the first frame are ordered such that points near to each other in the image are also near to each other in the ordered set. This ordering is important to our algorithm, as will be explained later in this section. It should be noted however that in general such a perfect ordering is not possible, since we are ordering points in a two-dimensional space into a one-dimensional vector that is supposed to keep the spatial relationships of the points in the two-dimensional space. One possible way to solve this problem is by constructing an array of m lists, with each list corresponding to a point in the first frame and containing the nearest k elements to this point. In the case that there are two points that are near each other in the two-dimensional frame, as indicated by the array of lists, but they are far from each other in the initial vector, then an additional initial solution is computed by moving one of the two points in the initial vector and inserting it near the other point Computing these extra initial solutions involves searching the k-element adjacency lists If an MLD point is found in the adjacency list of another point that is far from the first point in the initial solution vector, then an additional initial solution, considering these two points is computed as described earlier The procedure described in the next section is then applied for each of the computed initial solutions. To demonstrate the basic principals of the algorithm, we assume that the frame points can be ordered, without the need to create more than one initial vector, as is often the case of MLD's representing physical objects in non degenerate positions

The basic idea of the algorithm is to quickly enumerate possible solutions of the correspondence problem and store these solutions in the leaf PE's The set of possible solutions contains permutations of the initial solution, such that only points near to each other in the first frame are permuted. The sum of differences for these solutions can then be computed in parallel in all leaf PE's. The tree connections are then used to compute the matching that minimizes the computed sums (requiring $O(h)$ time) The match corresponding to the selected minimum value is the solution being sought We now show how to compute the permutations of the initial solution. For the sake of simplicity, we assume that only permutations of clusters of initial solution points are to be performed. Each cluster consists of three consecutive points in the initial solution vector, with clusters overlapping. We describe now how possible solutions containing all permutations of the first three elements can be computed.

The initial solution in the root PE is passed to its two children. The left child keeps the parent's solution, while the right child performs a permutation on this solution by swapping the first two correspondences, as shown in Figure 8-1

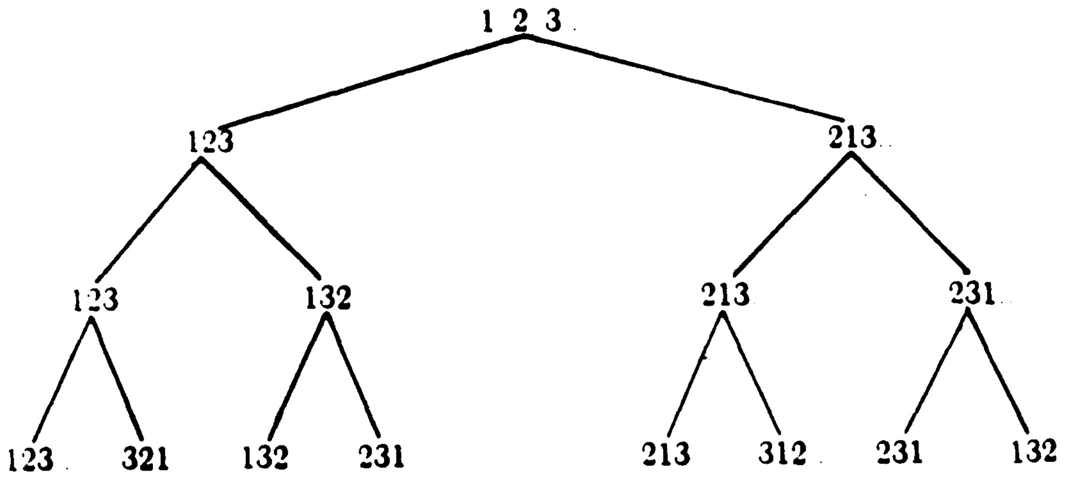


Figure 8-1: Permutations of the Initial Solution First Three Elements

Again the solution is passed to the next level PE's. The right children swap the second and third elements in their solution. When the solution is passed to the following level, right children swap the first and third elements. At this point solutions containing all possible permutations of the first three elements are found at the fourth level down the tree. Note that at the fourth level we have two solutions that are duplicated. This will not affect the procedure, but it will result in less efficient use of the PE's. This, however, can be avoided by replacing the duplicate solutions with other permutations of elements that are more than two elements apart in the solution vector. The same procedure is now repeated for the element three through five. This process continues until the leaf level is reached. At this point we have $O(2^h)$ possible solutions stored in the leaf PE's of the NON-VON tree.

The set of permutations of the initial solution may be computed in other ways. One possible method is to compute for each of the predicted points of the first frame their best match in the second frame. (This can be performed in time proportional to the number of points in the second frame.) If a point in the second frame has been selected as the best match for more than a single first frame point, then these points are reported to the CP. Permutations of this set of points are computed as part of the possible solutions to be stored in the leaf PE's.

We argue that the selected solution using this algorithm is near the optimal match, if not equal to it. The rationale for this is two-fold. First, the initial solution is presumably a good approximation of the desired solution, based on the heuristic assumption used in computing it Second, as much as possible the permutations on the initial solution are performed between points that are near to each other in the image frame That in turn insures that if there is a conflict resulting from two points in the first frame selecting the same correspondence in the second frame, then alternatives including both selections will be among the set of possible solutions.

The N-PASCAL procedure describing this algorithm for frames containing at most 16 points follows:

```
Procedure mld2;
var
  i, j, k, delta, dist_sqr integer;
  sol, x, y, x2, y2, xp, yp array[1..16] of integer;
vector_var
  X1 Y1 U V X2, Y2 integer,
  XP YP LEVEL_NO integer,
  NUM, DIST, TEMP integer,
  S array[1 16] of integer,
  RIGHTC ROOT, F1, F2, N boolean,

 /* The following function finds the minimum value of the vector
variable passed to it as an argument It looks for the minimum value
among the vector variable stored in enabled leaf PE's  */

function min_leaf(var MIN integer) integer,
var
  j integer,
vector_var
  TEMP integer,
begin

  for j = 2 to no_levels do
   begin
```

```
      N_RECV8(LC, MIN, MIN),
      N_RECV8(RC, MIN, TEMP);
      if TEMP < MIN then MIN = TEMP
    end;
   N_RECV8(LC, MIN, MIN),

  /* At this point the minimum value is stored in the CP register GG8
*/

   min_leaf = N_GG8;
end

Procedure swap_sol(i, j integer);
vector_var
   TEMP integer;
begin

   where RIGHTC = true do
     begin
       TEMP = S[j],
       S[j] = S[i];
       S[i] = TEMP;
     end,
end,


begin


  /* 1 Mark the right children, mark the root PE, and store the level
number of each PE in the vector variable LEVEL_NO  The first frame
points (X1,Y1) are assumed to be stored in the NON-VON leaf PE's,
with the corresponding velocity components U and V  F1 is set to 1 in
these PE's  The same is true for the second frame points (X2,Y2)  The
time lapse between two frames is stored in the variable delta  */


   XP = 0;
   YP = 0;
   mark_rc(RIGHTC),
   mark_root(ROOT),
   set_level_number(LEVEL_NO),


  /* 2 Compute the initial solution (points nearest to the predicted
```

solution). The points of the first frame are stored in the NON-VON tree, one per PE. X1 and Y1 are the coordinates of each of these points. XP and YP are the computed predicted values for the location of ~~each~~ point in the next frame. NUM holds the number of the point in the frame. The second frame is assumed to contain *n* points, while the first frame has *m* points. */

```
where F1 = true do
  begin
    XP := X + U * delta;
    YP := Y + V * delta;
   end;
N  = F2;
i = 1;
while (i <= m) do
  begin
    where (NUM = i) and (F1 = true) do
      begin
        N_REPORT8(XP, xp[i]);
        N_REPORT8(YP, yp[i]);
      end;
    DIST = 0,
    where N = true do
      DIST = (xp[i] - X2) * (xp[i] - X2)
      + (yp[i] - Y2) * (yp[i] - Y2),
    dist_sqr = min_leaf(DIST);
    where (N = true) and (DIST = dist_sqr) do
      begin
        N = false,
        N_REPORT8(NUM, sol[i]);
      end,
    i = i + 1;
  end;
```

/* 3 Store the initial correspondence in the root PE */

```
where ROOT = true do
  for j = 1 to m do S[j] = sol[j],
```

/* 4 Perform the permutations on the initial solution and store these possible solutions in the leaf PE's We will not deal with the duplicate

```
cases in this procedure */

    j := 2; i := 2;
    while j <= no_levels do
      begin
        where LEVEL_NO = j do
          begin
            for k := 0 to m do
              N_RECV8(P, S[k], S[k]);
            swap_sol(i, (i-1));
          end;
        j := j + 1;
        where LEVEL_NO = j do
          begin
            for k := 0 to m do
              N_RECV8(P, S[k], S[k]);
            swap_sol(i, (i+1));
          end;
        j := j + 1;
        where LEVEL_NO = j do
          begin
            for k := 0 to m do
              N_RECV8(P, S[k], S[k]);
            swap_sol((i-1), (i+1));
          end;
        j := j + 1;
        i := i - 2;
      end,

 /* 5 Compute the sum of differences for each possible solution. */

  DIST := 0;
  for j := 1 to m do
    begin
      for k := 1 to n do
        where S[j] = k do
          begin
            XP := x2[k];
            YP := y2[k];
          end,
      DIST := DIST + (XP - xp[j]) * (XP - xp[j])
            + (YP - yp[j]) * (YP - yp[j]);
    end,
/* 6. find the value of the minimum sum of differences.  */

dist_sqr = min_leaf(DIST);

/* 7. At this point the CP contains the value of the minimum sum of
position differences between the predicted positions and their
correspondence. Enable only the PE holding the solution with this
minimum value, and report the solution to the CP. */

  where DIST = dist_sqr do
      for j = 0 to m do
         N_REPORT8(S[j], sol[j]);
end;
```

Steps 2 and 4 of the algorithm execute in time proportional to the product of the number of points in the first frame and the tree height ($O(mh)$). On the other hand, Step 5 of the algorithm takes time proportional to the product of the number of points of the two frames ($O(mn)$). Thus, the algorithm executes in $O(\mathrm{Max}(h, n)m)$ time.

The functional simulator session for testing the algorithm described in this chapter is included in Appendix C. The first frame points have been used instead of their predicted positions. They have been ordered by starting at an arbitrary point and finding the nearest point to it. This greedy algorithm is continued until all points have been ordered. Frames with up to six points have been tested on a tree of 10 levels, and good solutions have been computed using the described algorithm.

The occlusion of some points in MLD frames, and the inability to perfectly order two-dimensional points in a one-dimensional vector are two problems that can cause the algorithm to compute solutions that are far from the optimal solution. However, it should be remembered that this algorithm has been intended to demonstrate the feasibility of solving such problems on tree machines, and not necessarily to show efficient ways of solving these problems.

Chapter 9
Conclusion and Directions for Further Research

In this research, we have addressed the problem of how fine-grained tree-structured SIMD machines can be used for high-speed execution of a wide range of image understanding tasks. Parallel algorithms have been developed for several image understanding tasks on the NON-VON machine, a highly parallel tree-structured SIMD machine. The image analysis applications considered in this thesis were selected to span different levels of computer vision tasks.

More specifically, we have developed and analyzed parallel algorithms for fast image correlation and quasi-parallel connected component labeling. A fast distributed algorithm that uses the NON-VON PE's space efficiently has been developed to implement the Hough transform method for detecting object boundaries. We have also developed a parallel algorithm that quickly enumerates possible solutions for the correspondence problem in the moving light display application. Other fast algorithms have also been developed, including image histogramming, set operations, and the computation of the geometric properties of objects.

The developed algorithms incorporate novel approaches to exploit the tree organization of the machine, and to reduce the effects of communication bottleneck usually associated with tree architectures. One technique we have used to this end involves partitioning the problem into a number of smaller problems that fit within a set of independent subtrees, in which communication is performed locally. Communication between these subproblems is less than would be required if the problem were not distributed among the subtrees. One example of this approach is the second image correlation algorithm described in Chapter 5. Another technique uses NON-VON's special hardware features to perform the communication for certain problems, as in the connected component labeling algorithm.

Issues related to the representation of images in tree machines have been addressed in this research. We have demonstrated how hierarchical data structures can be modified to represent images in the NON-VON tree. Fast image I/O is another important operation that affects the efficient implementation of vision algorithms. In this thesis, we have described different methods to perform such I/O efficiently in tree machines.

NON-VON's performance for different image algorithms has been analyzed and compared with that of other highly parallel image understanding architectures. Two simulators have been used to simulate the image analysis algorithms. A functional simulator has been implemented using the C programming language on a VAX 11/750 augmented with a Grinnell image processor. We have used this simulator to validate all of the algorithms described in this thesis. A LISP-based machine instruction-level simulator that has been developed for the NON-VON machine has been used to execute some of the image algorithms, and to provide accurate measures of the execution time of the machine-coded versions of our algorithms. Based on this comparison, NON-VON's execution time for several algorithms has been shown to be considerably less than that of other highly parallel vision architectures. We have identified the limitations of tree machines in the execution of certain image analysis tasks, and have proposed particular modifications to the NON-VON hardware for the rapid execution of these tasks.

This work can be extended in several possible directions. One possible avenue of further research would involve the investigation of other parallel algorithms for low- and intermediate-level vision application. A second involves the implementation of symbolic high-level vision tasks on the present version of the machine. In this regard, it is worth noting that the NON-VON architecture supports the efficient execution of operations arising in relational database management and expert systems applications. The relevance of algorithms in these two areas to high-level vision applications would suggest such effort might prove fruitful.

Another interesting research problem involves the manner in which currently proposed architectural additions to the machine (NON-VON 4) might be used to expand the set of vision tasks, which may be executed at very high speed. The MIMD, SIMD, and MSIMD capabilities of the proposed architecture and the availability of fine- and medium-grained PE's should prove useful in implementing systems that perform well on all levels of computer vision tasks.

References

[Anto 82] Antonisse, H. J.
Image Segmentation in Pyramids.
Computer Graphics and Image Processing 19:367-383, 1982.

[Arce 73] Arcelli, C., and Levialdi, S.
On Blob Reconstruction.
Computer Graphics and Image Processing 2:22-38, 1973.

[Baco 82] Bacon, D., Ibrahim, H., Newman, R., Piol, A., and Sharma, S.
The NON-VON PASCAL.
Technical Report, Computer Science Department, Columbia University, May, 1982.

[Ball 81] Ballard, D. H.
Generalizing the Hough Transform to Detect Arbitrary Shapes.
Pattern Recognition 13(2):111-122, 1981.

[Ball 82] Ballard, D. H., and Brown, C. M.
Computer Vision.
Prentice Hall, 1982.

[Brow 79] Browning, S.
Computations on a Tree of Processors.
In *Proceedings of The First Caltech Conference on VLSI.* January, 1979.

[Brow 84] Brown, C. M.
Peak Finding with Limited Hierarchical Memory.
In *Proceedings of the 7th. International Conference on Pattern Recognition, Montreal.* 1984.

[Burt 80] Burt, P J
Tree and Pyramid Structures for Coding Hexagonally Sampled Binary Images.
Computer Graphics and Image Processing 14:271-280, 1980

[Cast 79] Castleman, K. R
Digital Image Processing.
Prentice Hall, 1979

[Dubi 81] Dubitzki, Tsvi, Wu, A. Y., and Rosenfeld, A.
Parallel Region Property Computation By Active Quadtree Networks.
IEEE Transactions on Pattern Analysis and Machine Intelligence 3(6), November, 1981.

[Duda 72] Duda, R O, Hart, P E.
Use of the Hough Transformation To Detect Lines and Curves in Pictures.
Communications of the ACM 15(1), January, 1972.

[Duff 76] Duff, M. J. B
A Large Scale Integrated Circuit Array Parallel Processor.
In *Proceedings of the IEE Conference on Pattern Recognition and Image Processing*, pages 728-733. 1976.

[Dyer 80a] Dyer, C R.
Computing the Euler Number of an Image from its Quadtree.
Computer Graphics and Image Processing 13:270-276, 1980.

[Dyer 80b] Dyer, C. R., Rosenfeld, A., and Samet, H.
Region Representation Boundary Codes from Quadtrees.
Communications of the ACM 23(3), March, 1980.

[Dyer 81] Dyer, C R.
A VLSI Pyramid Machine for Hierarchical Parallel Image Processing.
In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, pages 381-386. 1981.

[Dyer 82a]	Dyer, C. R. The Space Efficiency of Quadtrees. *Computer Graphics and Image Processing* 19 335-348, 1982.
[Dyer 82b]	Dyer, C. R. Pyramid Algorithms and Machines. In *Multicomputers and Image Processing: Algorithms and Programs, Preston, K. Jr., and Uhr, L., eds.* Academic Press, 1982.
[Flyn 72]	Flynn, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* 21(9), September, 1972.
[Garg 82]	Gargantini, I. An Effective Way to Represent Quadtrees. *Communications of the ACM* 25(12), December, 1982.
[Gibs 82]	Gibson, L., and Lucas, D. Vectorization of Raster Images Using Hierarchical Methods. *Computer Graphics and Image Processing* 20 82-89, 1982.
[Gros 83]	Grosky, W. I., and Jain, R. Optimal Quadtrees for Image Segmenting. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 5(1), January, 1983.
[Hans 78a]	Hanson, A. R., and Riseman, E. M. *Computer Vision Systems.* Academic Press, 1978.
[Hans 78b]	Hanson, A. R., and Riseman, E. M. Segmentation of Natural Scenes. In *Computer Vision Systems, Hanson, A. R., and Riseman, E. M., eds.* Academic Press, 1978.
[Hans 80]	Hanson, A. R., and Riseman, E. M. Processing Cones: A Computational Structure for Image Analysis. In *Structured Computer Vision, Tanimoto, S., and Klinger, A. eds.* Academic Press, 1980.

[Hill 81] Hillis, W D
The Connection Machine
Technical Memo, M I T Artificial Intelligence Lab, September, 1981

[Hill 83] Hillyer, B K, Shaw, D E, and Nigam, A
NON-VON's Performance on Certain Database Benchmarks
Technical Report, Computer Science Department, Columbia University, November, 1983

[Houg 62] Hough, P V C
Methods and Means to Recognize Complex Patterns
US. Patent 3069654, 1962

[Hunt 79a] Hunter, G M, and Steiglitz, K
Linear Transformation of Pictures Represented by Quadtress
Computer Graphics and Image Processing 10:289-296, 1979

[Hunt 79b] Hunter, G m, and Steiglitz, K
Operations on Images Using Quadtrees
IEEE Transactions on Pattern Analysis and Machine Intelligence 1(2), April, 1979

[Ibra 83] Ibrahim, H A H
Tree Machines: Architecture and Algorithms
Technical Report, Computer Science Department, Columbia University, June, 1983.

[Ibra 84a] Ibrahim, H A H
The Connected Component Algorithm on the NON-VON Supercomputer
In *Proceedings of the IEEE Computer Society Workshop on Computer Vision: Representation and Control*, pages 37-45 1984

[Ibra 84b] Ibrahim, H A H, Kender, J R, and Shaw, D E
The Hough Transform Method on Fine-Grained Tree-Structured SIMD Machines
In *Proceedings of DARPA Image Understanding workshop, Lee S. Baumann, Ed.*. Science Applications Inc, October, 1984

[Ibra 84c] Ibrahim, H. A. H.
Some Image Understanding Algorithms on Fine-Grained Tree-Structured SIMD Machines.
In *Proceedings of the Workshop on Algorithm-Guided Parallel Architectures for Automatic Target Recognition.* 1984.

[Jens 74] Jensen, K., and Wirth, N.
PASCAL User Manual and Report.
Springer-Verlag, 1974.

[Klet 80] Klete, R.
Parallel Operations on Binary Images.
Computer Graphics and Image Processing 14:145-158, 1980.

[Klin 76] Klinger, A., and Dyer, C. R.
Experiments on Picture Representation Using Regular Decomposition.
Computer Graphics and Image Processing 5:68-105, 1976.

[Know 80] Knowlton.
Progressive Transmission of Gray-Scale and Binary Pictures by Simple, Efficient, and Lossless Encoding Schemes.
Proceedings of the IEEE 68(7), July, 1980.

[Knut 73] Knuth, D. E.
The Art of Computer Programming.
Addison Wesley, 1973.

[Kruse 76] Kruse, B.
The PICAP Picture Processing Laboratory.
In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, pages 875-881. 1976.

[Kung 80a] Kung, H. T.
Special Purpose Devices for Signal Processing: An Opportunity in VLSI.
Technical Report, Computer Science Department, Carnegie Mellon University, July, 1980.

[Kung 80b] Kung, H. T., and Song, S. W.
A Systolic Array Chip for the Convolution Operator in Image Processing.
Technical Report, Computer Science Department, Carnegie Mellon University, February, 1980.

[Kung 81] Kung, H. T., and Song, S. W.
A Systolic 2-D Convolution Chip.
Technical Report, Computer Science Department, Carnegie Mellon University, March, 1981.

[Kush 82] Kushner, T., Wu, A. U., and Rosenfeld, A.
Image Processing on ZMOB.
IEEE Transactions on Computers 31(10), October, 1982.

[Levi 80] Levine, M. D.
Region Analysis Using a Pyramid Data Structure.
In *Structured Computer Vision, Tanimoto, S., and Klinger, A., eds.*. Academic Press, 1980.

[Lumi 83] Lumia, R., Shapiro, L., and Zuniga, O.
A New Connected Components Algorithm for Virtual Memory Computers.
Computer Graphics and Image Processing 22:287-300, 1983.

[Mark 80] Marks, P.
Low Level Vision Using an Array Processor.
Computer Graphics and Image Processing 14:281-292, 1980.

[Mead 79] Mead, C and Conway, L.
Introduction to VLSI Systems.
Addison Wesley, 1979.

[Merl 75] Merlin, P. M., and Farber, D. J.
A Parallel Mechanism for Detecting Curves in Pictures.
IEEE Transactions on Computers 24(1), January, 1975.

[Ming 81] Ming, Li, Grosky, W. I., and Jain, R.
Normalized Quadtrees with Respect to Translations.
Computer Graphics and Image Processing 20:72-81, 1981.

[Nass 80] Nassimi, D., and Sahni, S.
Finding Connected Components and Connected Ones on A Mesh-Connected Parallel Computer.
SIAM J Computing 9:744-757, 1980.

[Pavl 82] Pavlidis, T.
Algorithms for Graphics and Image Processing.
Computer Science Press, 1982.

[Pott 83] Potter, J. L.
Image Processing on the Massively Parallel Processor.
IEEE Computer Magazine 16(1), January, 1983.

[Rao 76] Rao, C. V., Prasada, B., and Sarma, K. R.
A Parallel Shrinking Algorithm for Binary Patterns.
Computer Graphics and Image Processing 5:265-270, 1976.

[Rash 80a] Rashid, R.F.
Towards a System for the Interpretation of Moving Light Displays.
IEEE Transactions on Pattern Analysis and Machine Intelligence 2(6), November, 1980.

[Rash 80b] Rashid, R.F.
Lights: A Study in Motion.
PhD thesis, University of Rochester, April, 1980.

[Reev 84] Reeves, A. P.
Parallel Computer Architectures for Image Processing.
Computer Graphics and Image Processing 25:68-88, 1984.

[Rose 76] Rosenfeld, A.
Digital Picture Analysis.
Springer-Verlag, 1976.

[Rose 83] Rosenfeld, A.
Parallel Image Processing Using Cellular Arrays.
IEEE Computer Magazine 16(1), January, 1983.

[Same 80] Samet, H.
Region Representation: Quadtrees from Boundary Codes.
Communications of the ACM 23(3), March, 1980.

[Same 81a] Samet, H.
Computing Perimeters of Regions in Images Represented by Quadtrees.
IEEE Transactions on Pattern Analysis and Machine Intelligence 3(6), November, 1981.

[Same 81b] Samet, H.
An Algorithm For Converting Rasters to Quadtrees.
IEEE Transactions on Pattern Analysis and Machine Intelligence 3(1), January, 1981.

[Same 81c] Samet, H.
Connected Component Labeling Using Quadtrees.
Journal of the ACM 28(3), July, 1981.

[Same 82a] Samet, H.
Quadtrees and Medial Axis Transforms.
In *Proceedings of the IEEE Parallel Processing*, pages 184-187. 1982.

[Same 82b] Samet, H.
Neighbor Finding Techniques for Images Represented by Quadtrees.
Computer Graphics and Image Processing 18:37-57, 1982.

[Same 82c] Samet, H.
Distance Transform for Images Represented by Quadtrees.
IEEE Transactions on Pattern Analysis and Machine Intelligence 4(3), May, 1982.

[Same 82d] Samet, H.
A Top-Down Quadtree Traversal Algorithm.
Technical Memo, University of Maryland, Computer Science Department, december, 1982.

[Scha 84] Schaefer. D. H., Wilcox, G. C., and Harris, V.J.
A Pyramid of MPP Processing Elements.
Technical Report, Department of Electrical and Computer Engineering, George Mason University, 1984.

[Schw 80] Schwartz, J. T.
Ultracomputers.
ACM Transactions on Programming Languages and Systems 2, 1980.

[Sequ 79] Sequin, C. H.
Single Chip Computers, The New VLSI Building Blocks.
In *Proceedings of the First Caltech Conference on VLSI.* January, 1979.

[Shaw 82] Shaw, D E.
The NON-VON Supercomputer.
Technical Report, Computer Science Department, Columbia University, August, 1982.

[Shaw 83] Shaw, D E., and Hillyer, B K.
Allocation and Manipulation of Records in the NON-VON Supercomputer.
Technical Report, Computer Science Department, Columbia University, January, 1983.

[Shaw 84a] Shaw, D E.
SIMD and MSIMD Variants of the NON-VON Supercomputer.
In *Proceedings of the COMPCON Spring '84*. February, 1984.

[Shaw 84b] Shaw, D E., and Sabety T M.
An Eight-Processor Chip for a Massively Parallel Machine.
Technical Report, Computer Science Department, Columbia University, July, 1984.

[Shne 81a] Shneier, M.
Path-Length Distance for Quadtrees.
Information Sciences 23:49-67, 1981.

[Shne 81b] Shneier, M.
Calculations of Geometric Properties Using Quadtrees.
Computer Graphics and Image Processing 16:296-302, 1981.

[Sieg 79] Siegel, H J.
A Model of SIMD Machines and a Comparison of Various Interconnection Networks.
IEEE Transactions on Computers 28(12), December, 1979.

[Sieg 81a] Siegel, L J., Siegel, H J., and Feather, A E.
Parallel Image Correlation.
In *Proceedings of the IEEE Parallel Processing*, pages 190-198. 1981.

[Sieg 81b] Siegel, H. J., Siegel, L. J, Kemmerer, F. C., Mueller, P. T., Smalley, H. E., and Smith, S. D.
PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition.
IEEE Transactions on Computers 30(12), December, 1981.

[Siro 76] Siromoney, G., and Siromoney, R.
Hexagonal Arrays and Rectangular Blocks.
Computer Graphics and Image Processing 5:353-381, 1976.

[Sloa 79] Sloan, K. R., Tanimoto, S. L.
Progressive Refinement of Raster Images.
IEEE Transactions on Computers 28(11), November, 1979.

[Snyd 83] Snyder, W. E., and Cowart, A.
An Iterative Approach to Region Growing Using Associative Memories.
IEEE Transactions on Pattern Analysis and Machine Intelligence 5(3), May, 1983.

[Stam 75] Stamopoulos, C. D.
Parallel Image Processing.
IEEE Transactions on Computers 24(4), April, 1975.

[Ster 83] Sternberg, S. R.
Biomedical Image Processing.
IEEE Computer Magazine 16(1), January, 1983.

[Stol 82] Stolfo, S. J., Shaw, D. E.
DADO: A Tree-structured Machine Architecture for Production Systems.
In *Proceedings of the 2nd National Conference on Artificial Intelligence.* August, 1982.

[Tani 75] Tanimoto, S., and Pavlidis, T.
A Hierarchical Data Structure for Picture Processing.
Computer Graphics and Image Processing 4:104-119, 1975.

[Tani 78] Tanimoto, S.
Regular Hierarchical Image and Processing Structures in Machine Vision.
In *Computer Vision Systems, Hanson, A. R., and Riseman, E. M., eds.* Academic Press, 1978.

[Tani 80a] Tanimoto, S., and Klinger, A.
Structured Computer Vision.
Academic Press, 1980.

[Tani 80b] Tanimoto, S.
Image Data Structures.
In *Structured Computer Vision, Tanimoto, S., and Klinger, A., eds.*. Academic Press, 1980.

[Tani 82] Tanimoto, S. L.
Sorting, Histogramming, and Other Statistical Operations on a Pyramid Machine.
Technical Report, Computer Science Department, University of Washington, August 982.

[Tani 83a] Tanimoto, S. L.
A Pyramidal Approach to Parallel Processing.
Technical Report, Computer Science Department, University of Washington, January, 1983.

[Tani 83b] Tanimoto, S. L.
Algorithms for Median Filtering of Images on a Pyramid Machine.
Technical Report, Computer Science Department, University of Washington, January, 1983.

[Uhr 78] Uhr, L.
Recognition Cones, and Some Test Results
In *Computer Vision Systems, Hanson, A. R., and Riseman, E. M., eds.*. Academic Press, 1978.

[Uhr 80] Uhr, L.
Psychological Motivation and Underlying Concepts.
In *Structured Computer Vision, Tanimoto, S., and Klinger, A., eds.*. Academic Press, 1980

[Unge 58] Unger, S.H.
A Computer Oriented towards Spatial Problems.
In *Proceedings of the IRE*, pages 1744. 1958.

[Unge 59] Unger, S.H.
Pattern Detection and Recognition.
In *Proceedings of the IRE*, pages 1737-1752. 1959.

[Weem 84] Weems, C., Foster, C., Levinthal, S., Riseman, E., Hanson, A., and Lawton, E.
Content Addressable Parallel Array Processor
In *Proceedings of the Workshop on Algorithm-Guided Parallel Architectures for Automatic Target Recognition*. 1984.

[Zuck 76] Zucker, S. W.
Region Growing: Childhood and Adolescence
Computer Graphics and Image Processing 5 382-390, 1976.

Appendix A
The NON-VON 3 Instruction Set

The semantics of each NON-VON 3 instruction are described below along with the set of permissable operands, where appropriate[2].

INSTRUCTION	SEMANTICS
MOV8 <byte reg 1> <byte reg 2>	<byte reg 2> <- <byte reg 1>
<byte reg> = {A8, B8, C8, MAR, IO8}	
MOV1 <bit reg 1> <bit reg 2>	<bit reg 2> <- <bit reg 1>
<bit reg> = {A1, B1, C1, EN1, IO1}	

The MOV8 and MOV1 instructions transfer are used to transfer data between bit and byte registers within the PE.

READRAM8 <byte reg>	<byte reg> <- RAM8 (IMAR)
WRITERAM8 <byte reg>	RAM8 (MAR) <- <byte reg>
<byte reg> = {A8, B8, C8 or IO8}	
READRAM1 <bit reg>	<bit reg> <- RAM1 (IMAR)
WRITERAM1 <bit reg>	RAM1 (MAR) <- <bit reg>
<bit reg> = {A1, B1, C1 or IO1}	
INCREMENT	MAR <- MAR + 1

[2]The material presented in this appendix is from the paper "An Eight-Processor Chip for a Massively Parallel Machine", Technical Report by David Elliot Shaw and Theodore M. Sabety

The READRAM and WRITERAM instructions are used to transfer data between a register and the RAM location whose address is stored in the incrementing memory address register", IMAR. The INCREMENT instruction adds one to the address stored in the IMAR.

```
ADD  <byte reg>                  C8 <- (<byte reg> + A8 + C1);
                                 C1 <- carry

SUB  <byte reg>                  C8 <- (A8 - <byte reg> - C1);
                                 C1 <- borrow

COMPARE  <byte reg>              if <byte reg> = A8 then A1 <- 1
                                    else A1 <- 0;
                                 if <byte reg> > A8 then B1 <- 1
                                    else B1 <- 0

<byte reg> = {B8, IO8, MAR, or RAM}
```

The ADD, SUB and COMPARE instructions may be used to perform arithmetic and comparison operations on two 8-bit operands. The carry bit must be cleared before these instructions are initiated. The results of a COMPARE are stored in the A1 and B1 flags.

```
ROTRB                            Rotate B right 1 bit

ROTLB                            Rotate B left 1 bit
```

The B8 and B1 registers contain logic enabling them to function together as a 9-bit circular shift register.

```
LOGICAL8  <operation>            C8 <- (A8 <operation> B8)

LOGICAL1  <operation>            C1 <- (A1 <operation> B1)
```

where <operation> is a four-bit code specifying one of the sixteen possible boolean functions of two variables. LOGICAL8 applies the specified operation in a bitwise fashion to all eight bits of its operands.

Special cases of the LOGICAL1 instruction include SET, CLEAR, NEGATE, AND, OR, XOR, EQU, NAND, NOR and NOP LOGICAL1 may be used to combine the results of a COMPARE instruction to test all six possible arithmetic relational predicates (EQ, NE, GT, LT, GE and LE) on two 8-bit operands.

```
SEND8     <PE>    <byte reg>              IO8 (<PE>) <- <byte reg>
SEND1     <PE>    <bit reg>               IO1 (<PE>) <- <bit reg>
RECV8     <PE>    <byte reg>              <byte reg> <- IO8 (<PE>)
RECV1     <PE>    <bit reg>               <bit reg>  <- IO1 (<PE>)
<byte reg> = {A8, B8, C8, MAR}
<bit reg> = {A1, B1, C1, EN1}
<PE> = {LC, RC, LN, RN} for SEND instructions
       {LC, RC, LN, RN, PR} for RECV instructions
```

The SEND and RECV instructions are used to transfer data in parallel not only between PE's that are physically adjacent within the PPS, but also between two PE's that are adjacent in a particular linear sequence defined by an inorder traversal of the nodes of the PPS tree In both cases, data is transferred between some register in the PE in which the instruction is executed and the IO register within some neighboring PE It is always possible to RECV data from a PE, regardless of whether it is enabled, but an attempt to SEND data to a disabled PE will not result in a transfer of data

```
BROADCAST8  <byte reg>  <byte>        <byte reg> <- <byte>
BROADCAST1  <bit reg>  <bit>          <bit  reg> <- <bit)
REPORT8     <byte reg>                logical reg. CG8 (in CP) <- <byte reg>
REPORT1     <bit reg>                 logical reg. CG1 (in CP) <- <bit reg>
<byte reg> = {A8, B8, C8, MAR, IO8}
```

```
<bit reg> = {A1, B1, C1, EN1, IO1}
```

The BROADCAST instructions are used to transfer a single data value from the control processor into a specified destination register within all enabled PE's simultaneously. The REPORT instructions, on the other hand, are defined only when exactly one PE is enabled, and result in the transfer of data from the specified register within that PE to a particular logical register" within the control processor, which is called GG.

```
RESOLVE                                 A1 <- 0 in all PE's except
                                           "first" PE where A1 = 1;
                                        if no PE has A1 = 1 then
                                           logical register R1 (in CP) <- 0
                                           else R1 <- 1
```

After execution of a RESOLVE, the A1 register is reset to zero in all PE's except the one that occurs first in inorder traversal order. The RESOLVE instruction is frequently used in conjunction with REPORT to read data into the control processor from each one of a set of PE's in turn

```
ENABLE                                  EN1 <- 1 in all PE's, including
                                          those previously disabled
```

ENABLE is the only instruction that is executed by all PE's, whether or not they are already enabled. It is used to set all of the the EN1 registers to 1, thus awakening" all PE's in the PPS after some subset have been disabled.

```
STRINGBROADCAST <length> <string>       The semantics of these three
                                          operations are described below
STRINGREPORT <length>
```

STRINGCOMPARE <length> <string>

The three string operations use the auto-increment capability of the MAR to perform highly efficient loading, unloading, and matching operations on successive locations in RAM. The STRINGBROADCAST instruction transfers a common string from the control processor into the local RAM's of all enabled PE's, starting at the location specified by their respective MAR's. STRINGREPORT functions in a similar manner, but is used to transfer a string into the control processor from a single enabled PE. STRINGCOMPARE compares a string broadcast by the control processor in parallel against those stored in all enabled PE's. At the end of the STRINGCOMPARE instruction, only those PE's containing a matching string are left enabled.

Appendix B NON-VON 3 Code for Selected Algorithms

B.1. NON-VON Tree Initialization

```
(comment "This program initializes the NON-VON tree for vision
algorithms. The program will store at each node in the tree 4 values in
memory locations 0-3 representing the x-side, y-side, x-address, and y-
address of each rectangle in the binary image tree. The globally defined
variable no-levels is the number of levels in the tree.")

(fluid '(no-levels x y i))
(setq no-levels 7)

(de init-vision ()
(prog ()

% Store 1 in the variables x-side and y-side in all PE's.

 (N-ENABLE)
 (N-BROADCAST8 MAR 0)
 (N-BROADCAST8 A8 1)
 (N-WRITERAM8 A8)
 (N-INCREMENT MAR)
 (N-WRITERAM8 A8)

(comment "Set IO1 equal to 0 in leaf PE's, and equal to 1 in all other
PE's. Then enable only leaf PE's.")

 (N-CLEAR1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 RC)
 (N-SET1)
 (N-MOV1 C1 IO1)
```

```
(N-MOV1 A1 EN1)
(N-CLEAR1)
(N-MOV1 C1 IO1)
```

(comment "The following is a CP code that initializes x, and y global variables, and start the loop for computing in each PE the length and width of the rectangle it represents.")

```
(setq x 1)
(setq y 1)
(do ((i 1 (1+ i)))
    ((= i no-levels))

(N-ENABLE)
```

% Enable only PE's on the next level up the tree.

```
(N-RECV1 A1 RC)
(N-MOV1 A1 IO1)
(N-NEGATEA1)
(N-MOV1'C1 EN1)
```

(comment "This is a CP code that computes the rectangle dimensions on the current level."),

```
(cond ((eq i (times (quotient i 2) 2)) (setq y (times y 2)))
      (t (setq x (times x 2))))
```

% Store the x-side and y-side values in the enabled PE's.

```
(N-BROADCAST8 MAR 0)
(N-BROADCAST8 A8 x)
(N-WRITERAM8 A8)
(N-INCREMENT MAR)
(N-BROADCAST8 A8 y)
(N-WRITERAM8 A8)
) % END OF THE LOOP
```

(comment "At this point only the root PE is enabled with x=y=2**(n0-levels/2), and iol set equal to 0 in the root PE, and to 1 in all other PE's Store 0,0 in locations 2,3 in the root PE.")

```
(N-BROADCAST8 A8 0)
```

```
 (N-BROADCAST8 MAR 2)
 (N-WRITERAM8 A8)
 (N-INCREMENT MAR)
 (N-WRITERAM8 A8)
```

(comment "The following is a CP code to initialize the and start the loop for storing addresses.")

```
 (do ((i 1 (1+ i)))
    ((= i no-levels))
```

% Read x-side into the B8 register.

```
 (N-BROADCAST8 MAR 2)
 (N-READRAM8 B8)
```

(comment "This is a CP code to check if the current level number is odd or even.")

```
  (cond ((eq i (times (quotient i 2) 2))
        (setq x (quotient x 2))
```

(comment "Compute the x-address and y-address of the right-child in addresses 14,15.")

```
 (N-BROADCAST8 A8 x)
 (N-CLEAR1)
 (N-ADD B8)
 (N-BROADCAST8 MAR 14)
 (N-WRITERAM8 C8)
 (N-BROADCAST8 MAR 3)
 (N-READRAM8 A8)
 (N-BROADCAST8 MAR 15)
 (N-WRITERAM8 A8)
            )
```

% The else part.

```
 (t (setq y (quotient y 2))

 (N-BROADCAST8 MAR 14)
 (N-WRITERAM8 B8)
 (N-BROADCAST8 A8 y)
```

```
(N-BROADCAST8 MAR 3)
(N-READRAM8 B8)
(N-CLEAR1)
(N-ADD B8)
(N-BROADCAST8 MAR 15)
(N-WRITERAM8 C8)
    ))

% Enable PE's on the next level down the tree

(N-ENABLE)
(N-MOV1 IO1 B1)
(N-BROADCAST8 MAR 2)
(N-READRAM8 A8)
(N-SEND8 A8 LC)
(N-MOV8 IO8 B8)

(N-INCREMENT MAR)
(N-READRAM8 A8)
(N-SEND8 A8 LC)
(N-MOV8 IO8 C8)

(N-BROADCAST8 MAR 15)
(N-READRAM8 A8)
(N-SEND8 A8 RC)
(N-MOV8 IO8 C8)

(N-BROADCAST8 MAR 14)
(N-READRAM8 A8)
(N-SEND8 A8 RC)

% Enable only RC's.

(N-SET1)
(N-MOV1 C1 A1)
(N-CLEAR1)
(N-MOV1 C1 IO1)
(N-SEND1 A1 RC)
(N-MOV1 IO1 EN1)

(N-MOV8 IO8 B8)

% Enable next level down the tree
```

```
 (N-ENABLE)
 (N-MOV1 B1 IO1)
 (N-RECV1 A1 P)
 (N-MOV1 A1 IO1)
 (N-NEGATEA1)
 (N-MOV1 C1 EN1)

 (N-BROADCAST8 MAR 2)
 (N-WRITERAM8 B8)
 (N-INCREMENT MAR)
 (N-WRITERAM8 C8)
)))
```

B.2. Image I/O

```
% global variables declaration

(fluid '(i j))

(de load-img1 (x-side y-side)
(prog ()

 (N-ENABLE)

% B8 and C8 are used to hold the x-address and y-address respectively.

 (N-BROADCAST8 MAR 2)
 (N-READRAM8 B8)
 (N-INCREMENT MAR)
 (N-READRAM8 C8)

% Set MAR equal to 14 in all PE's

 (N-BROADCAST8 MAR 14)

% Loop to send bytes one bye one.

(do ((i 0 (add1 i)))
    ((= i x-side))
(comment "Store 1 in IO1 only in PE's with XADD equal to i, and set
MAR equal to 0")

 (N-ENABLE)
 (N-BROADCAST8 A8 i)
 (N-COMPARE B8)
 (N-MOV1 A1 IO1)
 (N-BROADCAST8 MAR 0)

  (do ((j 0 (add1 j)))
     ((= j y-side))

% Enable PE's in row i.

 (N-MOV1 IO1 EN1)
 (N-MOV8 MAR A8)
 (N-COMPARE C8)
 (N-MOV1 A1 EN1)
 (N-BROADCAST8 IO8 j)
 (N-ENABLE)
 (N-INCREMENT MAR)

))

% Enable only leaf PE's

 (mark_leaf)

% Store the gray level value.

 (N-BROADCAST8 MAR 4)
 (N-WRITERAM8 IO8)
))

(de load-img2 (bside imside)
( prog ()
 (N-ENABLE)

% 'bside'  is the length of the block side.
% 'imside' is the image side size.

% Set C1 equal to 1 only in the intermediate level PE's.
```

```
 (N-CLEAR1)
 (N-CLEAR8)
 (N-MOV8 C8 MAR)
 (N-READRAM8 A8)            % A8 <-- XSIDE
 (N-BROADCAST8 B8 bside)
 (N-COMPARE B8)
 (N-MOV8 A1 EN1)
 (N-INCREMENT MAR)
 (N-READRAM8 A8)
 (N-COMPARE B8)
 (N-MOV8 A1 C1)
 (N-ENABLE)

% B8 and C8 are used to hold the x-address and y-addresses.

 (N-BROADCAST8 MAR 2)
 (N-READRAM8 B8)
 (N-INCREMENT MAR)
 (N-READRAM8 C8)

% Loop to, send blocks of bytes.

 (setq n (quotient imside bside))
 (do ((i 0 (add1 i)))
         ((= i n))
     (do ((j 0 (add1 j)))
             ((= j n))
             (N-ENABLE)
             (N-MOV1 C1 EN1)
             (N-BROADCAST8 A8 (times i 4))
             (N-COMPARE B8)
             (N-MOV1 A1 EN1)
             (N-BROADCAST8 A8 (times j 4))
             (N-COMPARE C8)
             (N-MOV1 A1 EN1)
             (N-BROADCAST8 MAR 15)

% Store the block of 16 pixels

        (N-STRING-BROADCAST8 '(A B C D E F G H I
                               J K L M N O P))

))
```

```
(comment "Loading of blocks finished.  Now in parallel load the leaf
PE's.  Relative address are stored in B8 and C8.")

 (N-ENABLE)
 (N-MOV8 C8 IO8)
 (N-BROADCAST8 A8 3)
 (N-AND8)
 (N-MOV8 IO8 B8)
 (N-MOV8 C8 IO8)
 (N-AND8)
 (N-MOV8 IO8 B8)

% Store 15 in intermediate level PE's MAR.

 (N-BROADCAST8 MAR 15)

  (do ((j 0 (add1 j)))
    ((= j 4))

 (N-MOV1 C1 EN1)
 (N-INCREMENT MAR)
 (N-READRAM8 IO8)
 (N-ENABLE)
 (N-RECV8 A8 P)
 (N-MOV8 A8 IO8)
  )

 (N-ENABLE)
 (N-MOV1 C1 B1)
 (N-CLEAR1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 RC)
 (N-MOV1 A1 IO1)
 (N-MOV1 B1 C1)
 (N-MOV1 IO1 EN1)
 (N-BROADCAST8 MAR 4)


  (do ((j 0 (add1 j)))
    ((= j bside))
  (do ((i 0 (add1 i)))
    ((= i bside))
  (N-ENABLE)
  (N-MOV1 IO1 EN1)
  (N-BROADCAST8 A8 i)
  (N-COMPARE B8)
  (N-MOV1 A1 EN1)
  (N-BROADCAST8 A8 j)
  (N-COMPARE C8)
  (N-MOV1 A1 EN1)
  (N-WRITERAM8 IO8)

  (N-ENABLE)
  (N-MOV1 C1 EN1)
  (N-INCREMENT MAR)
  (N-READRAM8 IO8)
  (N-ENABLE)
  (N-RECV8 A8 P)
  (N-MOV8 A8 IO8)
))
))

(comment "The following function enables leaf PE's only,
 and set A1 equal to 1 only in leaf PE's.")

(de mark_leaf()
 (N-ENABLE)
 (N-CLEAR1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 RC)
 (N-MOV1 A1 EN1)
)

(comment "The following function prints the contents of one of RAM
location in leaf PE's.")

(de show-img (x-side y-side ram k)

 (N-ENABLE)

% IO8 holds the value to be printed

 (N-BROADCAST8 MAR ram)
 (cond ((= k 1) (N-READRAM8 IO8))
        (t (N-CLEAR8)
           (N-MOV8 C8 B8)
           (N-READRAM1 B1)
           (N-ROTLB)
           (N-MOV8 B8 IO8)
        )
 )


 (N-BROADCAST8 MAR 2)
 (N-READRAM8 B8)
 (N-INCREMENT MAR)
 (N-READRAM8 C8)

% Loop to report bytes one bye one.

(do ((j 0 (add1 j)))
    ((= j y-side))

 (terpri)
 (princ "column number ")
 (princ j)
 (princ "  ")

% Store 1 in IO1 only in leaf PE's with YADD equal to j

 (N-ENABLE)
 (N-CLEAR1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 LC)
 (N-MOV1 A1 IO1)
 (N-BROADCAST8 A8 j)
 (N-COMPARE C8)
 (N-MOV1 IO1 B1)
 (N-AND1)
 (N-MOV1 C1 IO1)
 (N-BROADCAST8 MAR 0)

  (do ((i 0 (add1 i)))
    ((= i x-side))

 (N-MOV1 IO1 EN1) % Enable PE's in row i
 (N-MOV8 MAR A8)
 (N-COMPARE B8)
 (N-MOV1 A1 EN1)
 (N-REPORT8 IO8)
 (princ (N-GET-GG8))
 (princ " ")
 (N-ENABLE)
 (N-INCREMENT MAR)
))
)
```

B.3. Binary Image Tree Building

```
(comment "This function building the binary image tree representation of
a binary image stored in the RAM1 location 4.")


(de build-binimg ()
(prog ()
 (N-ENABLE)
 (N-BROADCAST8 A8 71)   % 71 is the code for 'G'
 (N-BROADCAST8 MAR 5)   % RAM8 location 5 is FQUAD.
 (N-WRITERAM8 A8)       % Store 'G' in all PE's FQUAD
 (N-CLEAR1)
 (N-MOV1 C1 IO1)        % IO1 <-- 0

(comment "Current level is 0  Store 1 in RAM1 location 14 (X1) and 0
in RAM1 location 15 (Y1) only in current level PE's.")

 (N-RECV1 A1 RC)
 (N-BROADCAST8 MAR 14)
 (N-WRITERAM1 A1)
 (N-MOV1 A1 EN1)        % Enable leaf PE's only.
 (N-BROADCAST8 MAR 4)   % Read BINARY value into.
 (N-READRAM1 B1)        % B1
 (N-MOV1 B1 C1)
 (N-BROADCAST8 A8 87)
 (N-INCREMENT MAR)
 (N-WRITERAM8 A8)       % RAM8 5 <-- 'W'
 (N-CLEAR8)             % Store TREE in RAM8 6
 (N-MOV8 C8 B8)
 (N-ROTLB)
 (N-INCREMENT MAR)
 (N-WRITERAM8 B8)       % RAM8 6 <-- TREE.
 (N-MOV1 C1 EN1)         % Enable leaf black pixels
 (N-BROADCAST8 A8 66)  % Store 'B' in those PE's
 (N-BROADCAST8 MAR 5)
 (N-WRITERAM8 A8)

 (N-ENABLE)
 (N-BROADCAST8 MAR 14)
 (N-READRAM1 A1)         % A1  is 1 only in leaf PE's
 (N-NEGATEA1)            % C1  is 0  "    "    "    "
 (N-INCREMENT MAR)
 (N-WRITERAM1 C1)         % RAM1 15 is 0 "    "    "    "

% Loop to build the binary image tree.

(do ((i 1 (add1 i)))
    ((= i no-levels))

% Read the value of TREE into IO8.

 (N-ENABLE)
 (N-BROADCAST8 MAR 6)
 (N-READRAM8 IO8)

% Enable next level up the tree
 (N-BROADCAST8 MAR 15)
 (N-READRAM1 IO1)
 (N-RECV1 A1 RC)
 (N-WRITERAM1 A1)
 (N-NEGATEA1)
 (N-MOV1 C1 EN1)

% Receive gray values from  your two children.

  (N-RECV8 B8 LC)
  (N-RECV8 A8 RC)
  (N-CLEAR1)              % Add the two gray values.
  (N-ADD B8)
  (N-BROADCAST8 MAR 6)  % and store the resulting
  (N-WRITERAM8 C8)        % TREE in RAM8 6.

% Store the value of FQUAD in IO8 in previous level.

  (N-ENABLE)
  (N-BROADCAST8 MAR 5)
  (N-READRAM8 IO8)

% Enable current level again.

  (N-BROADCAST8 MAR 15)
  (N-READRAM1 A1)
  (N-NEGATEA1)
  (N-MOV1 C1 EN1)
  (N-RECV8 B8 LC)
  (N-RECV8 A8 RC)

% Compare the FQUAD in the two children.

  (N-COMPARE B8)
  (N-MOV1 A1 IO1)        % Enable only PE's where the two
  (N-MOV1 A1 EN1)        % FQUAD's are the same.
  (N-BROADCAST8 MAR 5)
  (N-WRITERAM8 A8)       % FQUAD in the result PE is set.
  (N-BROADCAST8 A8 71) % A8 <-- 'G'
  (N-COMPARE B8)
  (N-NEGATEA1)
  (N-MOV1 C1 IO1)        % IO1 is 1 where the two FQUAD
                         % are equal but not Gray.
  (N-ENABLE)
  (N-BROADCAST8 MAR 14)
  (N-READRAM1 A1)        % Enable only PE's on previous level.
  (N-MOV1 A1 EN1)

  (N-RECV1 EN1 P)        % Enable PE's with FQUAD to change to 'N'
  (N-BROADCAST8 A8 78) % A8 <-- 'N'
  (N-BROADCAST8 MAR 5)
  (N-WRITERAM8 A8)
  (N-ENABLE)
  (N-BROADCAST8 MAR 15)% Set RAM8 14 equal to 1 only
  (N-READRAM1 A1)        % in current level
  (N-NEGATEA1)
  (N-BROADCAST8 MAR 14)
  (N-WRITERAM1 C1)
 ))
)
```

B.4. Gray-Scale Image Histogram and Thresholding

```
(comment "This program computes the histogram of a gray-scale image
stored in the leaf PE's PE's of the NON-VON tree  The gray level level
value is stored in memory location 4 at each of these PE's ")


(fluid '(x y count nums bwid nbins no-levels level-1))
(setq no-levels 7)

(de image-histo ()
(prog ()

 (N-ENABLE)

(comment "Enable only leaf PE's and set IO1 equal to 1 in leaf PE's
and equal to 0 elsewhere ")

 (N-CLEAR1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 RC)
 (N-MOV1 A1 IO1)

(comment "Store the address 4 in MAR, and initialize global variables
(step 1) ")

 (N-BROADCAST8 MAR 4)
 (setq x 0)            % x is the min. value in bin range
 (setq y 15)           % y is the max. value in bin range
 (setq count 0)        % count is how many match operations.
 (setq nums 0)         % number of reported histogram values.
 (setq nbins 16)       % number of bins.
 (setq bwid 16)        % bin width.
 (setq level-1 6)      % no of levels above leaf level.

% Marking steps (steps 2,3)

 step2
 (N-MOV1 IO1 EN1)
 (N-READRAM8 A8)
 (N-BROADCAST8 IO8 0)
 (N-BROADCAST8 B8 x)
 (N-COMPARE B8)        % A1 <-- 1 if gray value = x.
 (N-NEGATEB1)          % B1 <-- 1 if gray < x
 (N-MOV1 C1 EN1)       % Enable only leaf PE's with gray >= x
 (N-BROADCAST8 B8 y)
 (N-COMPARE B8)        % A1 <-- 1 if gray value = y
 (N-OR1)             % B1 <-- 1 if gray < y
 (N-MOV1 C1 EN1)       % Enable PE's with gray <= y
 (N-BROADCAST8 IO8 1)

% The counting step (step 4).

 step4
 (N-ENABLE)
 (N-RECV8 B8 LC)
 (N-RECV8 A8 RC)
 (N-CLEAR1)
 (N-ADD B8)
 (N-MOV8 C8 IO8)

% Steps 5,6

 (setq count (plus count 1))
 (cond ((and (= count nbins)(ge count level-1))(go step7))
  (t (setq x (plus x bwid))(setq y (plus y bwid))
     (cond ((and (< count level-1)(< count nbins))(go step2))
     )
     (cond ((and (< count level-1)(ge count nbins))(go step4))
     )
  ))

(comment "Steps 7,8 For simplicity the histogram value is not stored
back in the tree  Enable root only and report the value in its IO8.")

 step7
 (N-ENABLE)
 (N-MOV1 IO1 B1)
 (N-put-gg1 1)     % gg1 <-- 1
 (N-CLEAR1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 P)    % A1 is 1 only in the root PE.
 (N-MOV1 A1 EN1)   % A1 is 1 only in the root PE.
 (N-REPORT8 IO8)
 (setq nums (plus nums 1))
 (princ "The value of bin histogram number ")
 (princ nums)
 (princ " is equal to ")
 (princ GG8)
 (terpri)

 (N-ENABLE)
 (N-MOV1 B1 IO1)

 (cond ((= nums nbins)(go end))
      (t (cond ((< count nbins)(go step2))
              (t (go step4)))))
end
))

(de image-thresh (thresh)
(prog ()
 (N-ENABLE)
 (N-BROADCAST8 MAR 4)
 (N-READRAM8 B8)           % Read the gray value in B8.
 (N-BROADCAST8 A8 thresh) % Broadcast the threshold value.
 (N-COMPARE B8)            % Compare (B1 <-- 1 if gray > thresh)
 (N-WRITERAM1 B1)          % Write value in RAM 1-bit location 4.
))

(comment "This function creates a random gray-scale image in the leaf
PE's.")

(de random-gray-image (rand-val1 rand-val2)
( prog ()

% Enable root only

 (N-ENABLE)
 (N-BROADCAST8 A8 rand-val2)
 (N-BROADCAST8 MAR 14)
 (N-WRITERAM8 A8)

 (N-PUT-GG1 0)              % gg1 <-- 0
 (N-SET1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 P)             % A1 is 0 only in the root PE
 (N-NEGATEA1)                    % A1 is 1 only in the root PE.
 (N-MOV1 C1 IO1)                 % IO1 is 1 only in the root PE.
 (N-MOV1 C1 EN1)                 % Enable the root PE only.
 (N-BROADCAST8 B8 rand-val1)
 (N-BROADCAST8 MAR 4)
 (N-WRITERAM8 B8)


 (do ((i 1 (1+ i)))
     ((= i no-levels))

 (N-ADD B8)
 (N-MOV8 C8 IO8)
 (N-ROTRB)
 (N-ROTRB)
 (N-ADD B8)
 (N-MOV8 IO8 A8)

% Enable PE's on the next level.

 (N-ENABLE)
 (N-SEND8 C8 LC)
 (N-SEND8 A8 RC)
 (N-MOV8 IO8 B8)

% Enable PE's on the next level down the tree.

 (N-RECV1 A1 P)
 (N-MOV1 A1 IO1)
 (N-MOV1 A1 EN1)
 (N-BROADCAST8 MAR 14)
 (N-READRAM8 A8)
 (N-BROADCAST8 MAR 4)
 (N-WRITERAM8 B8)

 )
))
```

B.5. Binary Image Shifting

```
 % Global variables declaration

(fluid '(xadd yadd w l k1 no-levels))
(setq no-levels 7)

(de binimg-shift(i j k)

% i,j shift distances, k=0 use white rectangles,
% k=1 use black ones.

(prog ()

  (N-ENABLE)

  (cond ((= k 0)(setq k1 87))
       (t (setq k1 66)))

(comment "The B8 and IO8 registers are used to store XADD and
YADD, IO1 for the shifted image, and C1 for REPORTED.")

  (cond ((= k 0)(N-SET1))
       (t (N-CLEAR1)))

  (N-MOV1 C1 IO1)        % IO1 <-- 0 or 1
  (N-BROADCAST8 MAR 5)
  (N-READRAM8 B8)         % read FQUAD into B8
  (N-BROADCAST8 A8 k1)  % A8 <-- 'B' or 'W'
  (N-COMPARE B8)          % A1 is 1 only in
                        % rectangles to be shifted.
  (N-CLEAR8)             % C8 <-- 0
  (N-MOV8 C8 MAR)
  (N-READRAM8 A8)         % A8 <-- XSIDE
  (N-INCREMENT MAR)
  (N-READRAM8 C8)         % C8 <-- YSIDE

% Read XADD, YADD into B8, IO8

  (N-INCREMENT MAR)
  (N-READRAM8 B8)         % B8 <-- XADD
  (N-INCREMENT MAR)
  (N-READRAM8 IO8)        % IO8 <-- YADD
```

```
(N-BROADCAST8 MAR 14) % MAR <-- 14
(N-WRITERAM1 A1)      % RAM1 14 <-- REPORTED
(N-WRITERAM8 A8)      % RAM8 14 <-- XSIDE


repeat
(N-READRAM1 A1)
(N-RESOLVE)

(cond ((= (N-GET-R1) 0)(go fini))
      (t (N-MOV1 A1 EN1)
        (N-CLEAR1)                % REPORTED <-- 'Y'
        (N-WRITERAM1 C1)
        (N-REPORT8 B8)             %report XADD

        (setq xadd (N-GET-GG8))

        (N-REPORT8 IO8)            %report YADD

        (setq yadd (N-GET-GG8))

        (N-READRAM8 A8)
        (N-REPORT8 A8)             %report XSIDE

        (setq w (N-GET-GG8))

        (N-REPORT8 C8)             %report YSIDE

        (setq l (N-GET-GG8))
        (setq xadd (plus xadd i))
        (setq yadd (plus yadd j))

        (N-ENABLE)                % Enable all PE's
        (N-BROADCAST8 A8 xadd)
        (N-COMPARE B8)
        (N-OR1)
        (N-MOV1 C1 EN1)

        (setq xadd (plus xadd w))

        (N-BROADCAST8 A8 xadd)
        (N-COMPARE B8)
        (N-NOR1)
```

```
                (N-MOV1 C1 EN1)
                (N-BROADCAST8 A8 yadd)
                (N-COMPARE IO8)
                (N-OR1)
                (N-MOV1 C1 EN1)

                (setq yadd (plus yadd 1))

                (N-BROADCAST8 A8 yadd)
                (N-COMPARE IO8)
                (N-NOR1)
                (N-MOV1 C1 EN1)
                (N-MOV1 IO1 A1)
                (N-NEGATEA1)
                (N-MOV1 C1 IO1)
                (N-ENABLE)

                (go repeat)
             )
   )
  fini

  % This is the case of non wraparound and
  % white rectangles

    (cond ((= k 0)

      % Enable area to be set to 0's

           (cond ((> i 0)(setq xadd 0)(setq w i))
                 (t (setq w 8)(setq xadd (minus 8 i))))

                (N-ENABLE)              % Enable all PE's
                (N-BROADCAST8 A8 xadd)
                (N-COMPARE B8)
                (N-OR1)
                (N-MOV1 C1 EN1)
                (N-BROADCAST8 A8 w)
                (N-COMPARE B8)
                (N-NOR1)
                (N-MOV1 C1 EN1)
                (N-CLEAR1)
                (N-MOV1 C1 IO1)
```

```
            (N-ENABLE)

          (cond ((> j 0)(setq yadd 0)(setq l i))
                (t (setq l 8)(setq yadd (minus 8 j))))

            (N-BROADCAST8 A8 yadd)
            (N-COMPARE IO8)
            (N-OR1)
            (N-MOV1 C1 EN1)
            (N-BROADCAST8 A8 l)
            (N-COMPARE IO8)
            (N-NOR1)
            (N-MOV1 C1 EN1)
            (N-CLEAR1)
            (N-MOV1 C1 IO1)
        ))
  (N-ENABLE)
  (N-BROADCAST8 MAR 20)
  (N-WRITERAM1 IO1)
))
```

B.6. Gray-Scale Image Shifting

```
(de pick_element (n)
 (N-ENABLE)
 (N-MOV1 IO1 E1)         % mark leaf PE's
 (N-CLEAR8)
 (N-MOV8 C8 IO8)
 (N-MOV8 C8 A8)
 (N-BROADCAST8 MAR 14) % read relative x_address
 (N-READRAM8 B8)
 (N-COMPARE B8)
 (N-MOV1 A1 EN1)
 (N-INCREMENT MAR)
 (N-BROADCAST8 A8 n)
 (N-READRAM8 B8)
 (N-COMPARE B8)
 (N-MOV1 A1 EN1)
 (N-BROADCAST8 MAR 4)  % read gray_value
 (N-READRAM8 IO8)
)
```

```
(de move_up()
 (N-ENABLE)
 (N-MOV1 IO1 A1)
 (N-NEGATEA1)
 (N-MOV1 A1 EN1) % mark non-leaf PE's
 (N-RECV8 A8 LC)
 (N-RECV8 B8 RC)
 (N-OR8)
 (N-MOV8 C8 IO8)
)

(de assign_element (n m)
 (N-ENABLE)
 (N-MOV1 IO1 E1)          % Enable leaf PE's
 (N-BROADCAST8 MAR 14) % read relative x address
 (N-BROADCAST8 A8 m)
 (N-COMPARE B8)
 (N-MOV1 A1 EN1)
 (N-INCREMENT MAR)
 (N-BROADCAST8 A8 n)
 (N-READRAM8 B8)
 (N-COMPARE B8)
 (N-MOV1 A1 EN1)
 (N-BROADCAST8 MAR 12) % read SH_P
 (N-READRAM8 A8)
 (N-INCREMENT MAR)
 (N-WRITERAM8 A8)         % store it in GRAY2_VALUE
)

(de move_down ()
 (N-ENABLE)
 (N-MOV8 IO8 B8)
 (N-BROADCAST8 MAR 12) % read SH_P
 (N-READRAM8 IO8)
 (N-RECV8 A8 P)
 (N-WRITERAM8 A8)
 (N-MOV8 B8 IO8)
)

(de move_around ()
 (N-ENABLE)
 (N-RECV8 A8 RC)
 (N-MOV8 IO8 B8)
```

```
  (N-SEND8 A8 LC)
  (N-BROADCAST8 MAR 12)
  (N-WRITERAM8 IO8)
  (N-MOV8 B8 IO8)
 )

 (de subimage_left_shift (k h)
  (N-ENABLE)
  (N-CLEAR1)
  (N-MOV1 C1 IO1)
  (N-RECV1 A1 RC)
  (N-MOV1 A1 IO1) % set LEAF

  (N-BROADCAST8 MAR 2)
  (N-READRAM8 B8)
  (N-BROADCAST8 A8 (minus k 1))
  (N-AND8)
  (N-MOV8 C8 IO8)
  (N-INCREMENT MAR)
  (N-READRAM8 B8)
  (N-AND8)
  (N-BROADCAST8 MAR 14)
  (N-WRITERAM8 IO8)
  (N-INCREMENT MAR)
  (N-WRITERAM8 C8)
 % The main program loop starts at this point

 )
```

B.7. Connected Component Labeling

```
(comment "This function labels the black or white rectangles of a binary
image  The label is stored in RAM8 location 7 while the common
boundary information are stored in RAM8 location 8  RAM8 location 9
is the level number ")


% global variables declaration

(fluid '(x y xs ys l k1 xxs yys newlabel no-levl))
(fluid '(comlabel curlev no-levels tw ts te tn))

(setq no-levels 7)
```

```
(de conn-comp(k)

% k = 1 --> label foreground components.
% k = 0 --> label background components.

(prog ()
 (N-ENABLE)

(comment "Step 1: The following function stores in RAM8 location 9 the
tree level number."

 (number-levels no-levels)

% Initialize global variables

 (cond ((= k 0)(setq k1 87))
       (t (setq k1 66)))
 (setq newlabel 0)
 (setq curlev 0)
 (setq no-levl 6)
 (setq tw 1)
 (setq tn 4)
 (setq te 16)
 (setq ts 64)

% Initialize the common boundary variable.

 (N-BROADCAST8 MAR 8)
 (N-CLEAR8)
 (N-WRITERAM8 C8)

% Initialize the REPORTED variable (IO1).

 (N-CLEAR1)
 (N-MOV1 C1 IO1)

(comment "Step 2 IO1 is set only in rectangles of type k1   RAM1
location 5 is set in only PE's with rectangles to be labeled ")

 (N-BROADCAST8 MAR 5)
 (N-READRAM8 B8)
 (N-BROADCAST8 A8 k1)
 (N-COMPARE B8)
 (N-MOV1 A1 IO1)
 (N-WRITERAM1 A1)

% Set LABEL equal to 0 in rectangles to be labeled.

 (N-MOV1 A1 EN1)
 (N-BROADCAST8 MAR 7)
 (N-CLEAR8)
 (N-WRITERAM8 C8)
```

(comment "Step 3 in the labeling algorithm. Report the rectangles one by one in order of their sizes.")

```
 step3 (N-ENABLE)
 (N-BROADCAST8 A8 curlev)
 (N-BROADCAST8 MAR 9)
 (N-READRAM8 B8)
 (N-COMPARE B8)
 (N-MOV1 IO1 B1)
 (N-AND1)
 (N-MOV1 C1 A1)
 (N-RESOLVE)

 (cond ((= (N-GET-R1) 0)
        (cond ((= curlev no-levl)(go fini))
              (t (setq curlev (plus curlev 1))(go step3))))
```

(comment "Step 4. Report the information of the selected rectangle, and mark it as reported.")

```
        (t (N-MOV1 A1 EN1)
        ))
           (N-CLEAR1)
           (N-MOV1 C1 IO1)  % REPORTED = 'Y'
           (N-CLEAR8)
           (N-MOV8 C8 MAR)
           (N-READRAM8 A8)
           (N-REPORT8 A8)   % Report XSIDE

           (setq xs (N-GET-GG8))

           (N-INCREMENT MAR)
           (N-READRAM8 A8)
        (N-REPORT8 A8)   % Report YSIDE

        (setq ys (N-GET-GG8))

        (N-INCREMENT MAR)
        (N-READRAM8 A8)
        (N-REPORT8 A8)   % Report XADD

        (setq x (N-GET-GG8))

        (N-INCREMENT MAR)
        (N-READRAM8 A8)
        (N-REPORT8 A8)   % Report YADD

        (setq y (N-GET-GG8))

        (N-BROADCAST8 MAR 7)
        (N-READRAM8 A8)
        (N-REPORT8 A8)   % Report LABEL

        (setq l (N-GET-GG8))
        (cond ((= l 0)(setq newlabel (plus newlabel 1))
              (setq l newlabel))
        )
        (setq comlabel l)

        (N-BROADCAST8 A8 l)
        (N-WRITERAM8 A8)

% Step 5 Test for adjacency in 4 directions

       (N-ENABLE)

% Store IO1 (REPORTED) in RAM1 location 9

       (N-BROADCAST8 MAR 9)
       (N-WRITERAM1 IO1)
       (N-CLEAR1)
       (N-MOV1 C1 IO1)
       (N-BROADCAST8 MAR 2)
       (N-READRAM8 B8)   % B8 <-- XADD
       (N-INCREMENT MAR)
       (N-READRAM8 IO8)  % IO8 <-- YADD
```

```
        (setq yys (plus y ys))
        (setq xxs (plus x xs))

        (N-BROADCAST8 MAR 9)
        (N-READRAM1 A1)
        (N-MOV1 A1 EN1)

% Check in the east direction.

        (N-BROADCAST8 A8 xxs)
        (N-COMPARE B8)
        (N-MOV1 A1 EN1)
        (N-BROADCAST8 A8 yys)
        (N-COMPARE IO8) % B is set if yys < YADD
        (N-NOR1)
        (N-MOV1 C1 EN1)
        (N-BROADCAST8 A8 y)
        (N-COMPARE IO8) % B1 is set if y < YADD
        (N-OR1)
        (N-MOV1 C1 EN1)
        (N-SET1)
        (N-MOV1 C1 IO1)
        (com-boundary tw)

        (N-ENABLE)
        (N-BROADCAST8 MAR 9)
        (N-READRAM1 A1)
        (N-MOV1 A1 EN1)

% Check in the south direction.

        (N-BROADCAST8 A8 yys)
        (N-COMPARE IO8)
        (N-MOV1 A1 EN1)
        (N-BROADCAST8 A8 xxs)
        (N-COMPARE B8) % B is set if yys < YADD
        (N-NOR1)
        (N-MOV1 C1 EN1)
        (N-BROADCAST8 A8 x)
        (N-COMPARE B8) % B1 is set if y < YADD
        (N-OR1)
        (N-MOV1 C1 EN1)
        (N-SET1)
        (N-MOV1 C1 IO1)
        (com-boundary tn)

        (N-ENABLE)
        (N-BROADCAST8 MAR 9)
        (N-READRAM1 A1)
        (N-MOV1 A1 EN1)

% Check in the west direction.
        (N-BROADCAST8 MAR 0)
        (N-READRAM8 A8)
        (N-CLEAR1)
        (N-ADD B8)
        (N-MOV8 B8 A8)
        (N-MOV8 C8 B8)
        (N-MOV8 A8 C8)
        (N-BROADCAST8 A8 x)
        (N-COMPARE B8)
        (N-MOV8 C8 B8)
        (N-MOV1 A1 EN1)
        (N-BROADCAST8 A8 yys)
        (N-COMPARE IO8) % B is set if yys < YADD
        (N-NOR1)
        (N-MOV1 C1 EN1)
        (N-BROADCAST8 A8 y)
        (N-COMPARE IO8) % B1 is set if y < YADD
        (N-OR1)
        (N-MOV1 C1 EN1)
        (N-SET1)
        (N-MOV1 C1 IO1)
        (com-boundary te)

        (N-ENABLE)
        (N-BROADCAST8 MAR 9)
        (N-READRAM1 A1)
        (N-MOV1 A1 EN1)

% Check in the north direction

        (N-BROADCAST MAR 1)
        (N-READRAM8 A8)
        (N-CLEAR1)
```

```
          (setq yys (plus y ys))
          (setq xxs (plus x xs))

          (N-BROADCAST8 MAR 9)
          (N-READRAM1 A1)
          (N-MOV1 A1 EN1)

% Check in the east direction.

          (N-BROADCAST8 A8 xxs)
          (N-COMPARE B8)
          (N-MOV1 A1 EN1)
          (N-BROADCAST8 A8 yys)
          (N-COMPARE IO8) % B is set if yys < YADD
          (N-NOR1)
          (N-MOV1 C1 EN1)
          (N-BROADCAST8 A8 y)
          (N-COMPARE IO8) % B1 is set if y < YADD
          (N-OR1)
          (N-MOV1 C1 EN1)
          (N-SET1)
          (N-MOV1 C1 IO1)
          (com-boundary tw)

          (N-ENABLE)
          (N-BROADCAST8 MAR 9)
          (N-READRAM1 A1)
          (N-MOV1 A1 EN1)

% Check in the south direction.

          (N-BROADCAST8 A8 yys)
          (N-COMPARE IO8)
          (N-MOV1 A1 EN1)
          (N-BROADCAST8 A8 xxs)
          (N-COMPARE B8) % B is set if yys < YADD
          (N-NOR1)
          (N-MOV1 C1 EN1)
          (N-BROADCAST8 A8 x)
          (N-COMPARE B8) % B1 is set if y < YADD
          (N-OR1)
          (N-MOV1 C1 EN1)
```

```
        (N-ADD IO8)
        (N-BROADCAST8 A8 y)
        (N-MOV8 C8 IO8)
        (N-COMPARE IO8)
        (N-MOV1 A1 EN1)
        (N-BROADCAST8 A8 xxs)
        (N-COMPARE B8) % B is set if yys < YADD
        (N-NOR1)
        (N-MOV1 C1 EN1)
        (N-BROADCAST8 A8 x)
        (N-COMPARE B8) % B1 is set if y < YADD
        (N-OR1)
        (N-MOV1 C1 EN1)
        (N-SET1)
        (N-MOV1 C1 IO1)
        (com-boundary ts)

% Step 5-b . Mark equivalence labels.

        (N-ENABLE)
        (N-BROADCAST8 MAR 7)
        (N-CLEAR1)
        (N-WRITERAM1 C1)
        (N-MOV1 IO1 EN1) % Only rectangles to be labeled
                         % are enabled.
        (N-CLEAR8)
        (N-MOV8 C8 A8)
        (N-COMPARE RAM8)
        (N-NEGATEA1)
        (N-MOV1 C1 EN1)
        (N-WRITERAM1 C1) % RAM1 7 <-- EQUIV
        (N-READRAM8 C8)  % C8 <-- LABEL

    % Set the label in all blocks

        (N-ENABLE)
        (N-MOV1 IO1 EN1)
        (N-BROADCAST8 A8 comlabel)
        (N-WRITERAM8 A8)

% Step 6

        (N-ENABLE)
```

```
            (N-CLEAR1)
            (N-MOV1 C1 IO1)
            (N-READRAM8 IO8)
      step6 (N-ENABLE)
            (N-BROADCAST8 MAR 7)
            (N-READRAM1 A1)
            (N-RESOLVE)
            (cond ((= (N-GET-R1) 0)
                   (N-ENABLE)
                   (N-MOV1 IO1 EN1)
                   (N-BROADCAST8 A8 comlabel)
                   (N-WRITERAM8 A8)
                   (N-ENABLE)
                   (N-BROADCAST8 MAR 9)
                   (N-READRAM1 IO1)
                   (go step3)
                  )
                  (t (N-MOV1 A1 EN1)
                     (N-CLEAR1)
                     (N-WRITERAM1 C1)
                     (N-REPORT8 C8)
                     (setq l (N-GET-GG8))
                     (N-ENABLE)
                     (N-BROADCAST8 MAR 5)
                     (N-READRAM1 A1)
                     (N-MOV1 A1 EN1)
                     (N-BROADCAST8 A8 l)
                     (N-COMPARE IO8)
                     (N-MOV1 A1 EN1)
                     (N-MOV1 A1 IO1)
                     (go step6)
                 )
            )
fini (N-ENABLE)
     (terpri)
     (princ "          THE END")
     (terpri)
     (princ "          -------")
))

(de com-boundary (l)
 (N-BROADCAST8 MAR 8)
 (N-READRAM8 A8)
 (N-MOV8 B8 MAR)
 (N-BROADCAST8  B8 i)
 (N-OR8)
 (N-MOV8 MAR B8)
 (N-BROADCAST8 MAR 8)
 (N-WRITERAM8 C8)
)

(comment "This routine stores the level number in RAM8 location 9.")

(de  number-levels (n)
 (N-ENABLE)
 (N-BROADCAST8 MAR 9)
 (n-put-ggl 1)

% Enable the root only.

 (N-CLEAR1)
 (N-MOV1 C1 IO1)
 (N-RECV1 A1 P)
 (n-put-ggl 0)

 (do ((i 0 (add1 i)))
     ((= i n))

     (N-MOV1 A1 EN1)
     (N-BROADCAST8 A8 i)
     (N-WRITERAM8 A8)
     (N-ENABLE)
     (N-MOV1 A1 IO1)
     (N-RECV1 A1 P)
))
```

Appendix C
Some Functional Simulator Results

C.1. Hough Transform

```
------------------------------------------------
Input subroutine number?  (0 to 11) <0>  6

Input boundary points file name  <>  pic32_4

Do you want the first or second method used (1 or 2):  (1 to 2) <1>

The first HT method
--------------------
Do you want to print the accumulator array on the screen?  <no>

How many points constitute a line?  (3 to 32) <5>

The solution
-------------
 count=  6, par1=  1, par2= -1
 count=  8, par1=  2, par2=  1
 count=  6, par1=  1, par2=  6
 count=  7, par1=  1, par2= 11
 count=  7, par1=  0, par2= 19
 count=  5, par1=  1, par2= 19
 count=  8, par1=  0, par2= 23
 count=  9, par1=  0, par2= 28
 count=  6, par1= -1, par2= 35
```

```
---------------------------------------------
Input subroutine number?   (0 to 11) <0>  6

Input boundary points file name  <>  pic32_4

Do you want the first or second method used (1 or 2): (1 to 2) <1> 2

The second HT method:
--------------------

Do you want to print the 2-dimensional histogram on the screen?  <no>

The first histogram values (par2):
 5, 6, 8, 6, 5,10, 8, 7,10,14, 4,10, 6, 8, 6, 8,10, 8, 6, 8, 7,11,
 7,10, 7, 4, 7, 6, 5,15, 4, 5, 6, 8, 3, 6, 6, 7,12, 5, 5, 4, 3, 7,
 6, 9, 3, 8, 5, 6, 2, 8, 3, 9, 5, 3, 5, 6, 6, 6, 5, 6, 3, 5.

Input the threshold value for the first histogram:  (1 to 200) <5>  7
 -8,-5,-4,-2,-1, 1, 3, 5, 6, 7, 9,11,13,19,23,28,35,37,41,43.

The second histogram values (par1):
 3, 2, 2, 5, 9,12,18,31,36,31,21,10, 7, 2, 1, 2.

Input the threshold value for the second histogram:  (1 to 200) <10>

Possible values of par1 are:
 -2,-1, 0, 1, 2, 3

How many points constitute a line?  (3 to 32) <5>

The solution
-------------
 par1= -1, par2= 35
 par1=  0, par2= 19
 par1=  0, par2= 23
 par1=  0, par2= 28
 par1=  1, par2= -1
 par1=  1, par2=  6
 par1=  1, par2= 11
 par1=  1, par2= 19
 par1=  2, par2=  1
```

C.2. Moving Light Displays

```
% vision

How many levels in the tree?  (2 to 12) <10>
 Do you intend to use the Grinnell ?  <yes>  n

 Subroutines Menu :
 ----------------
 0 : load an image
 1 : Build the Binary Image Tree
     [or the multi-resolution pyramid]
 2 : Label forground/background objects
 3 : Geometric properties of forground/background objects
 4 : Computing a gray image histogram
 5 : The gray image thresholding/Enhancement
 6   Executing the Hough Transform
 7   Display a multi-resolution pyramid tree level
 8 : Image shifting
 9   Image correlation
 10: Moving light displays
 11   Quit
 ------------------------------------------
Input subroutine number?  (0 to 11) <0>  10

Input first frame points file name  <>  frame1

Input first frame points file name  <>  frame1

Do you want to print the first frame points on the screen?  <no>  y

 The input first frame points
 -----------------------------------

 ( 4, 1)
 ( 3, 3)
 ( 1, 3)
 ( 4, 5)
 ( 5, 3)
 ( 7, 3)
```

```
Input the following frame points file name  <>  frame2

Do you want to print the second frame points on the screen?  <no>  y

 The input second frame points
 ------------------------------
 ( 8, 6)
 ( 5, 8)
 ( 7, 8)
 ( 9, 8)
 (11, 8)
 ( 8,10)

Do you want to print the initial solution on the screen?  <no>  y

 The initial solution
 ----------------------
 0, 1, 2, 3, 5, 4,

Do you want to print the final solution on the screen?  <no>  y

 The final solution
 ------------------
 0, 2, 1, 5, 3, 4,

 ----------------------------------------

Input subroutine number?  (0 to 11) <0>  10

Input first frame points file name  <>  plane1
```

```
Do you want to print the first frame points on the screen?  <no>  y

 The input first frame points
 ----------------------------------------

 ( 1, 5)
 ( 1, 7)
 ( 2, 9)
 ( 5, 7)
 ( 7, 5)
 ( 9, 7)

Input the following frame points file name  <>  plane2


Do you want to print the second frame points on the screen?  <no>  y

 The input second frame points
 ------------------------------------------


 ( 2, 6)
 ( 8, 6)
 ( 2, 8)
 ( 6, 8)
 (10, 8)
 ( 3,10)


Do you want to print the initial solution on the screen?  <no>  y

 The initial solution
 --------------------
 0, 2, 5, 3, 1, 4,
Do you want to print the final solution on the screen?  <no>  y

 The final solution
 ------------------
 0, 2, 5, 3, 1, 4,
Is there another frame?  <no>  y

Input the following frame points file name  <>  planel


Do you want to print the second frame points on the screen?  <no>  y

 The input second frame points
 ----------------------------

 ( 1, 5)
 ( 7, 5)
 ( 1, 7)
 ( 5, 7)
 ( 9, 7)
 ( 2, 9)

Do you want to print the initial solution on the screen?  <no>  y

 The initial solution
 --------------------
 0, 5, 2, 3, 1, 4,

Do you want to print the final solution on the screen?  <no>  y

 The final solution
 ------------------
 0, 2, 5, 3, 1, 4,

Is there another frame?  <no>

----------------------------------------------
Input subroutine number?   (0 to 11) <0>  11
% ^D
```