HYPERCUBE MACHINE IMPLEMENTATION OF LOW-LEVEL VISION ALGORITHMS

A Thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

 CHOON KEE LIM
August, 1988

Thesis
M 1988
Lim
## CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>1</th>
<th>Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER</td>
<td>2</td>
<td>The Hypercube Parallel Processor</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>Hypercube Configuration</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td>Topology Mapping on the Hypercube</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2.3</td>
<td>AMETEK System 14 Architecture</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>Programming the AMETEK System 14</td>
<td>18</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>3</td>
<td>Development and Implementation of Low-Level Vision Algorithms</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>3.1</td>
<td>Problem Decomposition</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>3.2</td>
<td>Topology Selection</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>3.3</td>
<td>Hypercube Machine Implementation of the Vision Algorithms</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>3.3.1</td>
<td>Vision Algorithm Descriptions</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>3.3.1(a)</td>
<td>Edge Detection</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>3.3.1(b)</td>
<td>Histogramming and Thresholding</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>3.3.1(c)</td>
<td>Thinning</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>3.3.1(d)</td>
<td>Noise Cleaning</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>3.3.2</td>
<td>Ring Topology Implementation</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>3.3.3</td>
<td>Mesh Topology Implementation</td>
<td>41</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>4</td>
<td>Experimental Results</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>4.1</td>
<td>Results of Ring Network Implementation</td>
<td>49</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Results of Mesh Network Implementation</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>CHAPTER</td>
<td>5</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>Program Compilation on the System</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>Program Execution on the System</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>Special Purpose Timer Routine</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>Communication Calls in User Topology Interfaces</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>APPENDIX E</td>
<td>Sequential and Concurrent Program Listings</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

The limited computational power and low speed of the conventional von Neumann sequential computers have always imposed serious problems on computer vision and image processing tasks. These tasks frequently require processing of immense array data within a very short time interval, typically in the range of a few seconds or less per image. This challenge remains unsolved even after the appearance of the powerful uniprocessor supercomputers. Although these machines are able to provide the required computation power, they are still unable to do the processing jobs cost-effectively. However, with the advent of commercial parallel processors in recent years, most of the computationally intensive image processing problems can be solved both cost- and time-effectively. This is due to the introduction of parallelism by these systems which increases significantly their processing power at a high speed. Unlike uniprocessor supercomputers, most of the parallel machines are built around multiple low-cost microprocessors which make supercomputing possible at a reasonable cost.

The fundamental principle of parallel processing is to couple a collection of processors in order to attain a much larger computing power which would not be possible with a single processor acting alone. In contrast to the traditional serial computers which execute instructions in a
serial manner, multiple set of instructions and data can be distributed and executed simultaneously within the parallel systems. This form of parallelism is often referred to as MIMD (multiple instruction stream–multiple data stream) form. Whereas the more restricted SIMD (single instruction stream–multiple data stream) parallelism is achieved by having all processors working concurrently on the same instructions and on different data. Naturally, the increase in processing speed in both cases is expected to be of a factor $N$ with the use of $N$ processors. However, this ideal factor can hardly be achieved due to various limiting factors such as the interprocessor communication delay, the degree of parallelism inherent in the corresponding task, the number of processors utilized, etc. Therefore, it is of prime importance that many of these parallel processing restrictions be addressed when designing parallel algorithms. In particular, the performance of a parallel algorithm is mainly dependent on the interprocessor communication.

Interprocessor communication in a parallel machine is usually achieved by an interconnection network which is a complex connection of switches and links [1]. These switches and links are employed to permit data communication between processors in a concurrent system. The performance of an interconnection network is commonly evaluated based on two criteria, namely the diameter of the network and the degree of a node [1]. The diameter is defined as the shortest distance or path between the two most remote processors, whereas the degree is the maximum number of links connected to each node. An efficient network would, of course, be one with both low diameter and low degree. This implies that the number of intermediate processors that need to be traversed during communication is
small and that the cost of the interconnection network is reduced with small number of links joined to each node.

Most interconnection networks are constructed based on the complete graph concept [2]. A complete graph is a network which links every node directly to all the other nodes. The complete graph representation of the conventional serial computer is simply a single node, which is the simplest network structure. Figure 1.1 and Figure 1.2 show the complete graph for both the 8-node ring and the 4-node square grid, respectively. Clearly, a complete graph would be the ideal interconnection structure for a parallel processor since communication between any pair of processors is always possible. However, the number of channels, and therefore the cost, drastically increases by an addition of a node to the system if the total number of nodes is large in the system. For example, an N-node system would require N-1 channels to connect each node to every other node in the network. Thus, a total of N(N-1) channels or \( \frac{N(N-1)}{2} \) bidirectional channels are needed in order to construct a complete graph network. An optimum solution to this problem is to select an interconnection network that closely resembles a complete graph and requires a small number of interconnections.

Besides the interconnection network, parallel processors are often characterized by their memory organizations. The memory organization can be classified as either tightly coupled shared-memory type or loosely coupled distributed-memory type. In a shared-memory system, all processors can directly access the shared memory by either reading from or writing to it. The main difficulty encountered in this case is the memory
Figure 1.1 8-node complete ring.

Figure 1.2 4-node complete mesh.
contention problem which occurs when several processors attempt to access the shared memory at the same time or when a processor attempts to access the shared memory while a previous access is still in progress. This leads to the conflict-free data access requirement for the parallel algorithms. In a local or distributed-memory system, a local memory is attached to each processor and interactions between processors are achieved via point-to-point message passings over the interconnection network. Intercommunication is the main degrading factor for this kind of system organization. High performance by parallel processing can, therefore, be attained only if the number of interprocessor communications is minimized.

In order to achieve high throughput with concurrent processors, parallel algorithms must be designed in such a way that the host-to-processor and processor-to-processor communications are kept to a minimum. This is due to the fact that if communication overhead is not minimized, it may even exceed the actual computation time. Therefore, proper scheduling of the required message or data transfer between processors is essential to reduce the communication overhead and to improve the overall system performance.

Most low-level vision algorithms require only repetitive local two-dimensional window convolutions on every pixel of the image. These local operations involve only pixels which are in the neighborhood of the one being processed. This iterative characteristic and the data locality feature make them particularly well-suited to the parallel processor implementation. In such implementation, each processor can be assigned a part or portion of the whole image to perform the local operations independently from other
processors except possibly at the subimage boundaries. Interactions between processors are, therefore, limited to exchanging only boundary pixel values. Hence, if the communication aspects are properly handled, significant reduction in processing time can be achieved.

Considerable amount of research effort has been devoted to studying the properties of the hypercube network and the array processor implementation of vision algorithms. For example, Kushner and Rosenfeld [3] have given the general implementation of various classes of image processing tasks such as point and local operations, transform, and statistics computations on string (or ring), array, and hypercube topologies. A general performance evaluation has also been provided for these topologies in performing various image operations on an N×N image. Because of its similar array structure, the array network is said to be very efficient for local operation. However, the authors concluded that hypercube structure is the best topology in comparison to the other two for image processing tasks if the image can be stored in a hypercube form.

Mudge and Abdel-Rahman [4] have developed a general model for hypercube machines to show the execution of vision algorithms on these systems. A thick-film (TF) circuit inspection task is particularly implemented to evaluate the hypercube machine performance. Using an NCUBE hypercube system with 1024 processors connected in a regular hypercube array topology, the authors reported that the 1 second processing time for inspecting 10 Mbytes of TF data can be achieved. However, the result was obtained by ignoring certain communication overhead and I/O time effect. Based on their study, they concluded that hypercube machines have great potential for low- and intermediate-level vision algorithms.
Fang, Li, and Ni [5] have presented several parallel algorithms for image template matching on an SIMD array processor with a hypercube interconnection network. They reported that communication time among the processors can be greatly reduced with the efficient use of hypercube architecture. Because substantial computation time is necessary for squared-error pattern clustering due to its iterative nature, Fang and Li [6] have also proposed a parallel algorithm for squared-error pattern clustering on a hypercube SIMD computer. They found that the hypercube SIMD architecture significantly reduces the processing time in comparison to the traditional SIMD computers.

A tightly coupled MIMD Homogeneous Multiprocessor with a local area H-network was proposed and used by Ramanamurthy et al. [7] to simulate three low-level vision algorithms. The authors reported an almost linear speedup with varying number of processors in the smoothing and edge detection algorithm simulations. A decrease in speedup was also reported for histogram generation. The limitation of the proposed architecture is that it has a resemblance of a string network which does not allow an easy mapping of other network topologies onto it. Therefore, higher level image processing tasks are possible only by making use of its tightly coupled memory feature. However, the potential for higher level vision tasks are great for loosely coupled hypercube machines since it can mimic many different network structures. By mapping it as a ring, the same tasks performed by the Homogeneous Multiprocessor can similarly be done.

In this regard, several low-level vision algorithms are implemented on a 16-node hypercube parallel processor (AMETEK S-14) by exploiting its network-embedding feature. This includes edge detection with Sobel
operator [8], histogramming, a one-pass parallel binary image thinning [9], and noise cleaning. The primary objective here is to parallelize these basic vision algorithms with the use of hypercube concurrent processor so that the image to processor-topology mapping problem is studied, and the actual speedup factor of parallel implementation over the sequential programming can be determined.

The remainder of this thesis is organized as follows. Chapter 2 is dedicated to a detail description of the hypercube machines with the emphasis on the AMETEK System 14. Also, various topologies that can be mapped onto the hypercube are discussed. Programming aspects of the System 14 is also included. Chapter 3 presents the development and implementation of the various low-level vision algorithms. The influence of internode communication on algorithm performance due to different topology mapping is particularly emphasized. Experimental setup and results are described in chapter 4. Finally, chapter 5 provides the conclusions and the discussion of further research topics.
CHAPTER 2

THE HYPERCUBE PARALLEL PROCESSOR

The design and analysis of interconnection network among processors have been the subject of extensive research (see, for example, [10-14]). Of the many network topologies explored up-to-date, hypercube has emerged as one of the most widely used architectural configuration. The wide acceptance of the hypercube structure can be attributed to several of its attractive features, some of which are listed below:

1. Many other network topologies such as tree, array (mesh), and ring can be mapped onto it.

2. No contention problem will be encountered due to its locally distributed memory feature.

3. Its diameter and degree are low and both of these parameters increase only logarithmically as the number of processors (N) increases.

4. It can be easily expanded to incorporate a large number of processors.

Based on the concept of the prototype 64-node "Cosmic Cube" developed by Seitz [15], there are now many commercially available hypercube parallel processors with variable number of nodes, ranging from a few to thousands of processors. Among them are the AMETEK System 14
Mainly because of its high degree of flexibility to embed many other network structures, hypercube is well-suited for solving a wide range of image processing problems. The general hypercube architecture and the various network topologies that can be mapped onto it are discussed in this chapter. A detailed description of the AMETEK hypercube machine is then presented. Finally, the discussion of various programming aspects of the AMETEK parallel processor is followed.

2.1 Hypercube Configuration

In its general form, hypercube is an n-dimensional binary cube which is also referred to as Boolean n-cube or simply n-cube. Here n denotes the dimension or the order of the hypercube which is given by \( \log_2 N \), where N is the total number of processors. The dimension of a hypercube can also be determined by the total number of channels connected to each node. For example, a three-dimensional hypercube has exactly three channels which link each node to three other neighbors. An n-dimensional hypercube consists of \( N = 2^n \) nodes or processors that are arranged in a cubic pattern, with each node being placed at a particular corner of the cube. Each processor in this architecture has a corresponding binary identification number within the range 0 to \( 2^n-1 \). An illustration of the hypercube with different dimensions is given in Figure 2.1. An n-dimensional hypercube can also be constructed from two (n-1)-dimensional hypercubes with some additional connections. This can be achieved by directly linking nodes on two separate subcubes which differ only in their most significant bits (MSB). (see Figure 2.1)
<table>
<thead>
<tr>
<th>DIMENSION</th>
<th>TOTAL NUMBER OF PROCESSORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 2.1 Hypercube topologies.
Every node in an n-dimensional hypercube has exactly n neighboring nodes, where neighbors are defined as nodes that differ in exactly one bit in their corresponding binary representations. This differing bit position is also identified as number of the channel which is used to enable communication between two connected nodes. For example, node 010 (2) and node 011 (3) are linked together by channel 0 since they differ in the least significant bit position. Each pair of neighboring nodes are always linked with a specific channel to provide direct communication. Communications between nonadjacent nodes are possible only by routing the required message through the intermediate nodes in a store-and-forward manner.

2.2 Topology Mapping On The Hypercube

One of the key issues in determining the overall parallel processing performance and the speedup factor is concerned with how well the decomposed algorithms can be mapped onto the actual parallel processor hardware configuration. A perfect match is, of course, most desired. However, if the communication structure of a parallel algorithm is different from the network structure, then the basic strategy is to match the two different structures in order to minimize the communication complexity. This goal can be achieved if those processors which require frequent interactions can be placed in close proximity. Therefore, proper topology mapping becomes an important factor for improving the algorithm performance.

Ring and the nearest neighbor are two well known and commonly used topologies which can be mapped onto the hypercube. In the ring topology, all processors are organized sequentially into a ring fashion and labeled from 0
through \( n-1 \). In this topology, nodes 0 and \( n-1 \) are always considered to be physically connected. This condition is referred to as wrap-around or periodic boundary condition. User program has to be written specifically to break the connection between node 0 and node \( n-1 \) if non-periodic conditions are desired. By doing so, however, the ring has in fact been converted to a string network.

Every processor in the ring is capable of directly communicating with its left and right neighbors, except possibly the nodes 0 and \( n-1 \) if non-periodic boundary conditions are set. Figure 2.2 illustrates the general configuration of an \( n \)-node ring. The actual mappings of a 4-node and 8-node ring onto the hypercube are shown in Figure 2.3. It is clear that by breaking certain connections, several different topologies can be configured from the hypercube.

The main disadvantage of the ring network is that only a small number of processors can be employed in order to maintain the data flow efficiency unless the problem exhibits linear structure. Communication overhead would increase if information is needed from those nodes other than the neighboring nodes. In fact, it takes at least \( \frac{N}{2} \) steps in order to route the message from any source node to its corresponding most distant destination node in this topology. For example, eight steps are needed to transmit information from node 8 to node 0 in a 16-node ring.

In the nearest neighbor topology, processors are organized in an \( n \)-dimensional nearest neighbor fashion and only processors that are adjacent to each other can have direct communications. For example, only neighboring processors in the south, north, east, and west directions are allowed to
Figure 2.2  Ring topology.

Figure 2.3  Actual ring mapping on the hypercube
(a) 4-node ring.
(b) 8-node ring.
interact with each other in the case of a two-dimensional mesh. As for the three-dimensional nearest neighbor network, two additional nodes on top and on the bottom are also considered as neighbors for each node. End nodes in each dimension are assumed to be not connected to each other. This means that no default periodic boundary conditions are present in this topology. In contrary to the ring topology which always provides the periodic boundary conditions, any periodic boundary conditions have to be defined by the user program so as to ensure the wrap-around between boundary processors in the corresponding dimension. Such periodic boundary conditions can be set in x, y, or z direction, or as a combination of these three directions.

Figure 2.4 shows the two-dimensional nearest neighbor topology. In this topology, processors are arranged as a square grid or rectangular array. This connection is also referred to as a mesh. Processors are always numbered first in the x, and then in the y direction. Similarly, the processor number assignment is made first in the x, then in the y, and finally in the z direction for a three-dimensional mesh shown in Figure 2.5. Each processor is capable of communicating with its various neighbors in the +x, -x, +y, -y, +z, and -z directions. For example, for a two-dimensional mesh with periodic boundary condition in both the x and y directions, the neighbors of node 0 are node 1, 3, 4, and 12. This type of processor arrangement can similarly be extended to the n-dimensional architecture.

Mapping of tree structure can also be performed on the hypercube but would not be discussed here. For a detailed discussion on the subject of embedding the tree network into hypercube, the interested readers are referred to [19]. Also, mapping of the ring and mesh networks into
Figure 2.4 Two-dimensional mesh topology.

Figure 2.5 Three-dimensional mesh topology.
hyper-cube is examined from the graph theory point of view in [20].

2.3 AMETEK System 14 Architecture

AMETEK System 14 is a loosely coupled message-passing parallel processor with a hypercube architectural configuration. The system can incorporate a minimum of 16 \(2^4\) nodes (processors) and up to a maximum of 256 \(2^8\) nodes. It is also capable of delivering a peak performance of 15 Mflops (millions of floating-point operations per second) if the total possible number of nodes is employed [21]. Although nodes are physically connected as a hypercube, other topologies such as ring, tree, and array (mesh) can also be mapped onto it. Also, in order to handle user interface and to coordinate any host-to-node communications, a controlling host computer is required to operate in conjunction with the System 14. Most importantly, however, the host is needed to ensure the overall concurrent program integrity [16].

Each node in the System 14 is based on an 8-Mhz Intel iAPX286 application processor which functions as the CPU and an 8-MHz Intel iAPX287 numeric coprocessor which provides the floating-point operation. In addition, a 10-MHz Intel iAPX186 communication processor is present in each node to manage communications with other nodes and to provide I/O interface with other external devices such as the timers, LED, etc. A one-Mbyte local memory is also attached to each node. The availability of this large amount of local memory makes it possible for the entire application program to be loaded into it.

In the System 14, every node is connected to its immediate neighbors via serial bidirectional communication channels. Communications between
the nodes are achieved via the message-passing mechanism. Any communication between the host and nodes is always through node 0 over the Direct Memory Access (DMA) channels. Node 0 is also called corner node.

2.4 Programming The AMETEK System 14

Because interprocessor communications have to be explicitly designed, parallel programming generally poses much more difficulty than conventional sequential programming. Unlike sequential programming, programmer needs to be familiar with details of the underlying machine architecture in order to construct an efficient parallel program. However, the System 14 comes with a set of software tools which are designed to help developing and debugging the concurrent program easily. Also, communication calls between processors are greatly simplified with the XOS operating system. The following is a brief description of the software provided.

There are two sets of routines provided by the System 14 to choose from when developing concurrent program. They include the XOS and the user topology interface routines. Besides determining a suitable topology to use, the programmer also needs to decide which of these routines are to be selected based on the problem complexity. The XOS routines are more difficult to use but they provide the flexibility of changing the topology within the concurrent program. Also, since the floating-point formats are incompatible between the VAX (the host computer) and the System 14, the floating-point format conversion is required for host-to-node communications if the XOS routines are used. However, although restricting the concurrent program to only one topology, the user topology interfaces
are much easier to use and they perform the floating-point format conversions automatically. The type of routines employed in a given program is indicated by the name of the parent and child program main functions. For the XOS routines, the main function names in both the child and parent programs must be given as \texttt{x\_main}, whereas \texttt{user\_main} are required for the user topology interface routines. Typically, user topology interfaces are used if the topology will not be changed throughout the concurrent program.

Because of the distributed memory structure of the system 14, any remote message required from other nodes should be passed over the interconnection network from the source to the destination node in a synchronized form. However, this communication synchronization aspect is taken care of by a set of communication calls and that no effort is required on the part of the programmer. With these communication macros, details of the actual communications are hidden and the programmer needs only to correctly specify the source and destination node identification numbers to successfully set up the communication paths.

Generally, after the concurrent program is created, it is then compiled, first in the host compiler to prepare it for running in the Simulator (simulated multiprocessor or Sim mode) and later using the Lattice C compiler for running in the System 14 hardware (Node) mode. Running the concurrent program in the Simulator allows one to debug and correct it before running it on the Node mode. Only then the program is downloaded for actual concurrent processing. This procedure ensures a smoother execution in the hardware mode.
To ease the conversion process of running alternately between the Sim mode and the Node mode, a developmental software called ADE (AMETEK Development Environment) is supported by the System 14. Using the ADE, only a simple "tonode" or "tosim" command is needed to switch from the sim to node mode, or vice versa.

The concurrent program compilation and execution procedures are provided in Appendix A and Appendix B, respectively. Timer routine for the parallel program time measurement is given in Appendix C. A summary (extracted from [15]) of the host-to-node and node-to-node communication calls using the user topology interfaces is presented in Appendix D. A more detailed description of all the programming aspects can be found in [16]. Finally, both the sequential and concurrent program listings are given in Appendix E.
CHAPTER 3

DEVELOPMENT AND IMPLEMENTATION OF LOW-LEVEL VISION ALGORITHMS

Low-level vision algorithms usually require massive amount of computation. However, these computations often involve only point or local operations. For point operations, the output gray level at a pixel depends solely on the input gray level at the same pixel [22]. Image thresholding is one such operation. As for local operations, the output gray level at a pixel depends only on the input gray levels in a neighborhood of that pixel [22]. Edge detection and thinning are considered local operations. Global information is not needed in either point or local neighborhood operations. Therefore, this data locality feature makes the low-level vision algorithms particularly well-suited for parallel processor implementation.

The detailed description and implementation of some low-level vision algorithms are presented in this chapter. These algorithms are edge detection with Sobel operator, histogram generation, thresholding, thinning operation by using a parallel one-pass thinning algorithm, and noise cleaning. Several main issues such as the efficient image decomposition, selection of a suitable topology, and proper internode communications are discussed. However, main emphasis will be focused on the communication aspect.

3.1 Problem Decomposition
For proper topology mapping, a given image is first partitioned into several subimages which are then distributed among the individual processors. Parallelism is achieved by processing each subimage independently in a parallel operation. This is also referred to as the divide-and-conquer approach. Upon the completion of operations, partial results from each processor element are then combined. For an MXM or M² square image, the basic decomposition is simply \( \frac{M^2}{N} \) and this amount of image data points is then downloaded equally to each processor. Because of the well-balanced and symmetrical array structure of the image, this primitive decomposition method is trivial. However, the host-to-node downloading process still depends heavily on the network topology employed.

Depending upon the topology employed, a subimage can either be in the form of an \( \frac{M}{N} \)XM strip for the ring topology or an \( \frac{M}{\sqrt{N}} \times \frac{M}{\sqrt{N}} \) square matrix form for the nearest neighbor topology as shown in Figures 3.1 and 3.2, respectively. For ring topology, downloading a strip form subimage can be carried out with a single download macro command. However, difficulty arises when downloading a square form subimage in the nearest neighbor topology. In this case, to download the subimages to the corresponding nodes successfully with a single macro command, the image has to be rearranged in a strip form as shown in Figure 3.1. An alternative to this approach is that a subimage can be sent one row at a time until the whole subimage is accumulated into the local memory of the corresponding node.

For algorithms that involve a great deal of internode communications, it is sometimes feasible to download any information needed by the nodes prior to the processing. This results in limited number of internode
Figure 3.1  Image partitioning in ring topology.

Figure 3.2  Image partitioning in mesh topology.
communications during the data manipulation. In this way, the communication time is limited to the host-to-node data transfer time. For instance, local image operations can be performed with no internode communication if the image can be downloaded in an overlap fashion so that the boundary pixels of subimages are stored in the processor memories prior to the operations. Since the host-to-node communication time is much greater than that of the node-to-node communication, this scheme will be useful only if timing improvement can be achieved. However, not all node-to-node communications can always be replaced with host-to-node communications to have a reduction in communication time. The disadvantage of such downloading process is that it is suitable for the local image operations which require only one iteration [3]. For an iterative process which needs to be performed several times, the required information in the remaining iterations would again need to be downloaded from the host. Since this might involve uploading the processing results of the previous iterations from the children, this scheme will not be effective in such cases.

Load balance is another key factor for performance improvement. Each processor should be assigned equal amount of processing task with approximately the same computational complexity. Efficient processing would be hampered if all other processors have finished their processing jobs but have been forced to wait for some “heavy loaded” processors to finish their jobs. However, this presents no real problem for low-level vision algorithms since similar local operations have to be performed by each processor on every pixel of a subimage. This means that the load is well balanced for each child except possibly for some nodes performing a
few more operations on the border pixels.

3.2 Topology Selection

Matching parallel algorithm with the interconnection network structure is often a difficult task. Before selecting any particular topology to solve a given problem, it is important to know the data flow requirement between processors for performing specified tasks on the input data. Different topology used leads to different algorithm performance. If the topology employed well matches the underlying data structure, efficiency of the concurrent program can be significantly improved. This improvement is a result of the reduction of interprocessor communications.

Because the whole image is subdivided among the processors, internode communication is inevitable in image processing tasks by the array processors. For local image operations, intercommunication occurs whenever operations involve some boundary data that are stored in different processors. Operations can be carried out only if all the needed information directed from the various processors has been received. Therefore, topology that would permit fast data transfer among processors is the most suitable network for local neighborhood operations.

3.3 Hypercube Machine Implementation of the Vision Algorithms

Parallel processor implementation of the various vision algorithms are divided into four stages as shown in the block diagram of Figure 3.3. Output image of each stage is used as an input image for another cycle of computation in the next stage. Edge points of the input image are first detected using the Sobel edge detector. Histogram of the edge-detected
Figure 3.3 Parallel image processing block diagram.
image is then generated to convert the gray-level image into binary form by means of thresholding. With the one-pass parallel thinning algorithm, the resultant binary image is then processed to obtain the image skeleton. Finally, as a post-processing stage, any remaining isolated noisy points in the thinned image are filtered with a pair of noise cleaning templates.

The low-level vision algorithms mentioned above are implemented using the ring and two-dimensional nearest neighbor mesh topology. Figures 3.4 through 3.6 give the flowcharts of the overall parallel implementations of these algorithms with these topologies. Although the parent’s task remains unchanged in both the ring and mesh implementations, the child programs differ mainly in the way the internode communications are set up and histogram is accumulated into the corner node. A common child program or instruction stream is first downloaded to all nodes from the parent. The entire MxM image is subsequently partitioned into N, with each consisting of \( \frac{M^2}{N} \) pixels, distinct subimages. Each subimage block is then downloaded or assigned to the available processors according to the mapping. Image processing tasks are performed concurrently in each node upon the receiving of the subimage.

3.3.1 Vision Algorithms Description

Object detection and recognition play a significant role in many computer vision tasks such as automated visual inspection, workpiece acquisition, robotic assembly, etc. In these tasks, several image processing operators such as edge detection, histogramming, thresholding, thinning, and filtering are applied to an input image in order to extract the necessary object features. Therefore, a brief description of these vision algorithms is
Figure 3.4 Parent program flowchart.
Figure 3.5 Child program flowchart in ring topology.
Figure 3.6 Child program flowchart in mesh topology.
Figure 3.6 (continued) Child program flowchart in mesh topology.
given in the following subsections. Their ring and mesh implementations are also described in detail.

3.3.1(a) Edge Detection

One of the fundamental operations in computer vision is to detect edges of an object in order to identify its shape. An edge exists at image points where the image gray levels exhibit sharp discontinuities. This large variation in gray levels can be detected with some gradient operators, which normally are given in the form of 3x3 pixel masks. Edge points are determined as points where the local maxima occur as a result of the first derivative of the gradient, which gives the rate of change of the gray level values.

Although many type of edge operators have been proposed, the Sobel operator is by far the most popular edge detector. This operator consists of two 3x3 windows as shown in Figure 3.7. Mask $G_x$ is the horizontal edge detector which is used to locate edges in the x direction and mask $G_y$ is the vertical edge detector that is used to detect edges in the y direction. The gradient magnitude in the horizontal and vertical directions are calculated as follows:

$$G_x = (P1 - P6) + 2*(P2 - P7) + (P3 - P8)$$

$$G_y = (P1 - P3) + 2*(P4 - P5) + (P6 - P8)$$

(1.1)

(1.2)

The magnitude of the gradient at a given pixel $P$ can then be calculated as

$$G = \sqrt{(G_x)^2 + (G_y)^2}$$

(1.3)

For the ease of computation, this form is often approximated by
Figure 3.7 Sobel edge detector.

Figure 3.8 Edge detection.
As shown in Figure 3.8, the edge strength or gradient of the pixel $P$ can be easily computed by simply convolving the local image with the two masks. After the two window convolutions are performed on each pixel of the entire image, a resultant gradient image can then be obtained. This gradient or edge-detected image is subject to further processing in the following stage.

3.3.1(b) Histogramming and Thresholding

A histogram is the frequency distribution of gray levels in an image. It is often used for thresholding purpose to convert a gray-level image into a binary image which consists of only object boundary points. If the gray levels of an image are in the range from 0 through $p$, then the histogram value of gray level $p$ is given as $h(p)$, where $h(p)$ denotes the number of pixels corresponding to gray level $p$ in that image.

In its sequential implementation, histogram generation is also considered as point operation. However, since different parts of the image reside distinctively in a number of processors, parallel implementation would require global information in order to sum up the partial histograms generated by each of the $N$ processors. Therefore, it can no longer be considered as point operation. In this case, it is classified as statistics computation [3]. In parallel processing environment, proper coordination among processors is particularly important in order to correctly condense the histogram into the node 0 within a reasonable period of time.

Gradient image $G$ resulted from the edge detection process has to be

$$G = |G_x| + |G_y|$$
converted into binary or bilevel form $B$ in order to carry out the thinning operations with the employed parallel one-pass thinning algorithm. This conversion can simply be done by comparing each of the gradient image pixel with some predetermined threshold values. This process is defined as

$$\begin{align*}
B(i,j) = \begin{cases} 
1 & \text{if } T_1 < G(i,j) < T_2 \\
0 & \text{otherwise}
\end{cases} \quad (1.5)
\end{align*}$$

Here, a pixel whose gradient lies between the threshold values $T_1$ and $T_2$ is considered as an edge point and is assigned a value of one. A zero pixel represents a nonedge point and is considered as a background or object point. Noise introduced in the detection of image boundaries can be cleaned up considerably by the thresholding process. The resultant binary image contains less salt-and-pepper type noise and still provides the necessary edge information.

### 3.3.1(c) Thinning

Edges detected after applying the edge operators are often few pixels wide and can be thinned to a skeleton of unit thickness. Thinning is a common computer image processing operation. It can be defined as the process of eliminating layers of edge pixels from the many pixel wide object for generating skeletal pixels of the object [22]. It is desired that the resultant skeleton image after thinning still gives a well-described configuration of the object's shape in the approximate connected form. Connectivity of the skeleton is especially important in order not to distort the actual object's shape.
Data compression is, for the most part, the main purpose of thinning operation. This skeletonizing operation is often implemented by some thinning algorithms. Most existing thinning algorithms carry out the thinning process mainly in an iterative fashion. The thick layers of pixels from the object are removed iteratively until only one pixel-wide skeleton is obtained. The thinning algorithm used here is a parallel one-pass thinning algorithm [9], which requires only a one-pass or a single-cycle of parallel operations per iteration. These thinning operators are realized with the use of eight thinning templates and two restoring templates as shown in Figure 3.9(a)-(b). The eight thinning templates are designed to simultaneously eliminate boundary pixels in all the eight-connected (i.e., the north, east, west, south, northeast, northwest, southeast, and southwest) directions of a binary edge point. Two restoring templates are included to prevent a possible occurrence of any breakage in or disappearance of horizontal and vertical two-pixel wide strips.

3.3.1(d) Noise Cleaning

As a post-processing phase, any remaining isolated noisy pixels in the thinned image are filtered with a pair of noise cleaning templates as shown in Figure 3.10. It should be noted that because of the 3x3 window size of the templates, only completely isolated noisy image points can be eliminated. Any remaining two or more pixels wide isolated noisy segments would not be eliminated unless the template size is increased.

3.3.2 Ring Topology Implementation

After receiving the corresponding subimage from the host, all window operations are then performed locally in each node. Instead of waiting for
Figure 3.9 One-pass thinning algorithm templates.
(a) $3 \times 3$ thinning templates.
(b) $1 \times 4$ and $4 \times 1$ restoring templates.
Figure 3.10 Noise cleaning templates.

Figure 3.11 Interprocessor communications for ring topology (where PE refers to the processing element).
Figure 3.10 Noise cleaning templates.

Figure 3.11 Interprocessor communications for ring topology (where PE refers to the processing element).
the required boundary data to be sent by the neighbors, all necessary data items are gathered before engaging in a mask operation in a particular processing stage. Therefore, all required border strips are first copied from the neighbors prior to the processing. For the ring implementation, this involves only interactions of neighbors on the left and on the right of a specific node, which makes the interprocessor communications straightforward.

The amount of data items that need to be transferred from the neighbors is dependent on the window size used by the algorithm. As presented in the previous section, all algorithms implemented here employ only 3x3 mask except the 1X4 and 4X1 restoring templates of the thinning process. For 3x3 edge template computations, one row of boundary image elements has to be obtained from both the left and the right neighbors. Although one row of data items is routed from the left neighbor, two rows of border pixels have to be delivered by the right neighbor in order to implement the restoring templates. This intercommunication strategy for the ring implementation is illustrated in Figure 3.11.

In the case of a 512x512 image and a 16-node hypercube processor, each node first receives the corresponding 32x512 image portion downloaded by the host. By declaring the subimage array in the child program to be of size 34x512, the received subimage is then placed in the center of this larger array, which leaves free the first and the last rows of the array. This scheme facilitates storing of the boundary image pixels coming from both the left and the right neighbors. For the 3x3 window computations, the interprocessor communications begin by having all processors, except the node 0, sending their first image rows (rows1) to
their corresponding destination nodes on the left. Similarly, each node also receives a complementary row of data items from the right neighbor. This row of data is then appended to the end (row 33) of the subimage array. Since the node 0 is excluded from this operation, its corresponding left neighbor, which is the node 15, does not receive any data from it. This is also true for the case when the last rows (rows 32) of the subimages are transmitted to the right neighbors. In such a case, the node 0 does not receive any data from the node 15. At nodes other than node 0, the data rows obtained from the left neighbors are stored into the top rows (rows 0) of the subimage arrays. After all these communications, each subimage has been expanded to a size of 34x512 with the inclusion of the received boundary data rows. The edge detection task can then be successfully performed on each pixel of the original 32x512 subimages. This interprocessor communication technique is used in other stages when boundary image pixels have to be acquired from the neighbors.

Since the intercommunications involve only the immediate left and right neighbors, it is realized that all other local window convolution tasks with varying mask size can be easily implemented under the ring topology. However, since processor intercommunications are also dependent on the amount of data being transferred, longer communication time is expected for transferring larger subimage blocks. With the ring network, histogram generation requires processing time in the magnitude order of N since all partial results have to be added up in the node 0. A slight improvement can be achieved by breaking the ring into two halves and accumulating the partial results from the left and right neighbors at the same time. This method, however, still requires \( \frac{N}{2} \) processing steps. All partial histograms
generated in the non-corner nodes are shifted from each node toward the node 0 and are accumulated to form the total histogram.

### 3.3.3 Mesh Topology Implementation

As in ring implementation, all necessary boundary image pixels are initially gathered from the neighboring nodes before beginning any operation so that the entire subimage can be processed without data transmission interruption. After receiving all the required boundary data items from the neighbors, each node then begins its processing tasks independently until boundary pixels from the adjacent nodes are needed for the next processing stage. In the two-dimensional mesh topology, data items which reside at all the eight-connected neighbors of a particular processor will be needed when all the frame pixels of a subimage have to be processed. For processing an input image of 512x512, each node is assigned a 128x128 subimage after image partitioning. All points within a subimage frame are successfully processed without any data transfer. As can be seen from Figure 3.12(a), all boundary points located at the eight-connected neighbors are needed in order to process the four corner pixels. This, therefore, requires boundary data points to be transmitted from all eight neighbors. To cope with the substantial data movement, a careful design strategy is desired to ensure smooth and efficient data flow. The communication scheme utilized is depicted in Figure 3.12(b) and can be carried out in only four steps. As an example, the communication issue for the 3x3 mask operations is explained below. This scheme can be easily extended to an operation (e.g., thinning process) which utilizes different window size.

First, a subimage array of size 130x130 is created in each node. The
Figure 3.12 (a) Illustration of corner pixel mask operation
(b) Interprocessor communications for mesh topology
(where PE refers to the processing element).
received 128\times 128 subimage is then centered within this array, leaving free the first and the last rows and columns (i.e., rows 0 and 129, and columns 0 and 129). Initially, each node sends its first row (row 1) to its corresponding north neighbor. The same node also receives the data row coming from its south neighbor and append it to its original subimage array as row 129. With the same approach, each node then sends the last row (row 128) to its south neighbor. In this case, the received data row is added to the top row (row 0) of the initial subimage at the destination node. Clearly, the two communication steps described above extend the subimage size from 128\times 128 to 130\times 128. However, in order to successfully process all pixels of a subimage, data from the east and west neighbors as well as from the diagonal neighbors must be received. This presents some difficulty since the diagonal neighbors are two steps away from the corresponding center node. Notice, however, that after the two data row transmissions, the required diagonal pixels have in fact been transferred to the four-connected neighbors of a node that requests the data. They are, therefore, only a step away from the node and can be sent directly. However, to smooth the data transmission process, the first and last column of image points are copied into two separate array buffers, with buffer 1 containing the first column of data and buffer 2 containing the last column of data. As the third step, array buffer 1 in each node is transmitted to the corresponding west neighbor. This column of data is then appended to the last column of the 130\times 128 image at the receiving node. By this time, the subimage has been expanded to the size of 130\times 129. Finally, buffer 2 of each node is also sent to the east neighbor and added to the subimage as its first column. This further expands the subimage size to 130\times 130 and enables all original subimage pixels to be processed with no further difficulty.
Apart from the partial histograms generated in nodes 0, 4, 8, and 12 under the nearest neighbor mesh topology, all histogram results obtained in other nodes are shifted to their west neighbors. They are then summed up separately in the nodes 0, 4, 8, and 12 in three iterations. Since only the node 0 can directly communicate with the host, the final histogram has to be generated in the corner node. Hence, all remaining partial histograms are shifted from north to south toward the node 0. Partial histograms that reach node 0 are accumulated. After these steps are completed, the node 0 contains the final sum. Figure 3.13 shows this step by step histogram generation procedure.
Figure 3.13 Histogram generation in nearest neighbor topology.
CHAPTER 4

EXPERIMENTAL RESULTS

Parallel implementation of the low-level vision algorithms in this research has been conducted on the AMETEK System 14 16-node hypercube system as described in Chapter 2. The concurrent processor is hosted by the VAX 11/750 minicomputer that runs under the UNIX 4.3 BSD operating system. Image acquisition, digitization, and display are performed through the VICOM image processor, which is interfaced with an RCA camera and controlled by the VAX computer. The overall experimental environment is shown in the block diagram of Figure 4.1.

Two concurrent programs which consist of parent and child programs were written for each topology. These programs were developed using the user topology interface routines. All aspects of the processor coordinations and communications were programmed. Each of the parallel programs was designed based on a particular network structure and coded in the C programming language with the aid of the ADE software tools.

Initially, gray-scale images of several complex shaped objects were acquired through the VICOM image processor and stored into the host computer. They are 512x512 images with the pixel gray levels ranging from 0 through 255. Pixel elements which lied on the image frame were treated as part of the background and were not processed. For edge detection and noise cleaning algorithms, the rows 0 and 511 and the columns 0 and 511
Figure 4.1 Block diagram of experimental environment.
were considered to be as part of the image frame. For thinning algorithm, the row 510 and the column 510 were also considered as part of the image frame.

A total of three images have been processed in both sequential and parallel processors to determine the achievable speedup factors. Here the speedup factor is defined as

$$S = \frac{T_S}{T_p}$$

(4.1)

where $T_S$ is the sequential program execution time and $T_p$ is the parallel program execution time. The sequential program execution time was measured with respect to a single node of the System14. Timing measurement on both the sequential and parallel programs were made by a general purpose timer routine given in Appendix C. All I/O file reading and writing time was not included into the time measurement. For the sequential program, only actual computation time, starting from the beginning of the edge detection stage to the end of the noise cleaning stage, was measured. The time measurement for the parallel program presented here was divided into two parts; the communication and computation time.

Two case studies were then conducted accordingly. In one case, the node-to-host upload, host-to-node download, and the interprocessor communication time was included into the overall communication time. In the second case, these upload and download time measurements were completely ignored. With these two separate measurements, the effect of host-to-node communication on algorithm performance was determined.
4.1 Results of Ring Network Implementation

The overall parallel program which implements the edge detection, histogramming, thresholding, thinning, and noise cleaning algorithms was first executed in the System 14. The timer was initialized to zero just before the input image was downloaded to the children and terminated after the histogram and output filtered image were uploaded to the host. In this way, the program execution time was measured as the time needed to process the image through the several stages as illustrated in Figure 4.1. With the same timer routine, the sequential program execution time for a single node was then measured. To ensure accuracy of the time measurement, each program was executed five times and the processing time was taken as the average time of these five executions. Next, the downloading and uploading communication time was excluded from the time measurement. The overall timing is measured as the computation and node-to-node communication time.

For a detailed analysis, each algorithm was then processed individually in both sequential and parallel manner. Similarly, the impact of the communication complexity was treated in two separate cases as explained before. All timing results and speedup factors obtained, along with those mentioned above, are tabulated in Tables 1 and 2, respectively. As can be seen from the two tables, speedup factor of in the order of 10 can be obtained for all algorithms except for the histogramming process. This is primarily because of the fact that the communication over computation ratio is higher due to accumulating all partial histograms into the node 0.

Also as noted from the tables, the communication time far exceeds
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Sequential</th>
<th>Parallel (I/O)</th>
<th>Parallel (no I/O)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Ring</td>
<td>Mesh</td>
</tr>
<tr>
<td>Overall</td>
<td>43.911568</td>
<td>33.253053</td>
<td>33.083547</td>
</tr>
<tr>
<td>Edge Detection</td>
<td>18.319696</td>
<td>30.881406</td>
<td>30.823004</td>
</tr>
<tr>
<td>Histogramming</td>
<td>3.915280</td>
<td>14.705346</td>
<td>14.758472</td>
</tr>
<tr>
<td>Thresholding</td>
<td>4.672832</td>
<td>29.851842</td>
<td>29.861399</td>
</tr>
<tr>
<td>Thinning</td>
<td>6.938560</td>
<td>30.002389</td>
<td>29.956726</td>
</tr>
<tr>
<td>Noise Cleaning</td>
<td>11.669216</td>
<td>30.368146</td>
<td>30.506656</td>
</tr>
</tbody>
</table>

**Table 1** Processing time summary (all values are given in seconds).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ring</td>
</tr>
<tr>
<td>Overall</td>
<td>11.558751</td>
</tr>
<tr>
<td>Edge Detection</td>
<td>12.834833</td>
</tr>
<tr>
<td>Histogramming</td>
<td>9.740567</td>
</tr>
<tr>
<td>Thresholding</td>
<td>11.747336</td>
</tr>
<tr>
<td>Thinning</td>
<td>12.654101</td>
</tr>
<tr>
<td>Noise Cleaning</td>
<td>12.763257</td>
</tr>
</tbody>
</table>

**Table 2** Speedup summary.
the actual processing time if the communications between node and host are taken into account. This is due to the low I/O bandwidth between the VAX and System 14, which allows only a maximum of 64 Kbytes of data transfer in a single communication.

4.2 Results of Nearest Neighbor Network Implementation

The time measurement was carried out in exactly the same manner as in the case of ring topology. In one case, the host and nodes communication time was included to the real processing time, whereas this communication time was excluded in another case. Also, the overall algorithm was broken down into distinct individual cases for detailed analysis. The obtained results are also listed in Table 1 and Table 2 for comparison.

As shown in Tables 1 and 2, speedup factor in the order of 10 is also attainable for all algorithms with the use of nearest neighbor mesh topology except for the histogram generation. In fact, all results obtained in this case are close to those generated using the ring topology. This is because the amount of data transfer is the same for both topologies and that all boundary pixels can be obtained directly from the immediate neighbors. Therefore, besides differing in the number of communication steps needed to accumulate the histogram and the communication scheme used, no significant difference exists between the two topologies. With careful algorithm design for efficient interprocessor communications, it is rather hard to determine which topology is significantly advantageous over the other in terms of simple window convolutions. However, since the square image can be easily projected onto the same array structure, the mesh topology is, in this sense, considered more suitable for image processing.
Although no significant difference is indicated by the speedup factors achieved using the two topologies, it is clear that ring topology has virtually no potential for higher level vision algorithms due to its high network diameter.

It should also be noted that although each parallel program is executed five times in order to ensure correct time measurement, the timer routine used here has the disadvantage that its result is dependent upon the workload present on the host computer at the time of measurement [24]. This results in poor accuracy for short time measurements. This presents some problems for time measurement here since all algorithms implemented in this research require only processing time of only few seconds or less, precision of the results is hard to verify. However, almost linear speedup has been obtained for many experimental trials.

The resultant images after the edge detection, thresholding, thinning, and noise cleaning operations on the various input images are presented in Figure 4.2(a)-(e) through Figure 4.4(a)-(e). However, due to the poor quality of the filtered PCB image, this result is not included. Histograms generated for each image are also illustrated in Figures 4.5 through 4.7.
Figure 4.2 (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.2 (continued) (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.2 (continued) (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.3 (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.3 (continued) (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.3 (continued) (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.4 (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.4 (continued) (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.4 (continued) (a) Original image (b) Edge-detected image (c) Thresholded image (d) Thinned image (e) Filtered image.
Figure 4.5 Scissor histogram.

Figure 4.6 Plier histogram.
Figure 4.7 PCB histogram.
CHAPTER 5

CONCLUSIONS AND DISCUSSION

Significant reduction in image execution time can be gained by employing parallel processing techniques. This is particularly true for low-level vision algorithms as a consequence of their inherent parallelism, problem regularity, and data locality.

The idea of parallel image processing with hypercube machine has been described in this research. Parallel implementation of four typical low-level vision algorithms has been derived. These algorithms include point operations, local operations, and statistics computations. Detailed experiments have also been conducted to study the influence of interprocessor communications on vision algorithm performance. The objective here is to determine the possible speedup by parallelizing these algorithms and mapping them onto the ring and the mesh network topologies of a concurrent processor. Due to the partitioning and image uploading and downloading difficulties encountered with the use of user topology interfaces, the low-level vision processing tasks are not implemented with the actual hypercube interconnection network. However, this problem can be solved by making use of the topology reconfigurable feature of the XOS routines. In this case, image I/O requirement can be performed with ring topology and later the processing job be carried out using hypercube topology.
For low-level vision algorithms, parallelism is exploited by decomposing an image into 16 subimages and distributing the subimages to the 16 processors of the concurrent machine. Each processor then works independently on its assigned image portion unless boundary image pixels are needed from the neighboring nodes. Interprocessor communications are achieved by routing the required data from the source node to the destination node in a store-and-forward manner. A considerable improvement in speedup factor in the order of 10 in comparison to the serial programming has been obtained by executing various low-level vision algorithms concurrently on several 512x512 gray level images. The result is obtained by reducing the number of interprocessor communications with careful algorithm design and by excluding the host-to-node and I/O communications. Although no significant difference in speedup is observed with the use of the ring or the mesh topology, it is realized that the mesh topology is more suitable for processing higher level vision problems.

Only edge detection, histogram calculation, thresholding, thinning, and noise cleaning algorithms have been implemented in this research. However, other low-level vision algorithms such as smoothing, median filtering, template matching, etc. can also be implemented with the same proposed approach on the hypercube machine. A further development of this research will emphasize implementing algorithms which involve global operations and object motion.
References


Program Compilation on the System 14

In order to ease the program compilation process, it is suggested that a makefile be created. A makefile is simply a file which specifies program dependencies. It keeps track of the changes of all program modules so that recompilation will be performed only for program modules which have been changed. This eliminates the unnecessary waste of time for recompiling all programs when in fact not all modules have been changed. A sample makefile used in the ADE is given below.

CFLAGS = -Uring
#
# For nearest neighbor topology CFLAGS = -Unn
# and for hypercube topology CFLAGS = -Uhcube
#

vision: visionp.exe visionc.exe

visionp.exe : visionp.o
    ldp -Uring -o visionp.exe visionp.o

visionc.exe : visionc.obj
    ldc -Uring -o visionc.exe visionc.obj

.SUFFIXES : .obj
.c.o : ;ccp -c $(CFLAGS) $<
.c.obj : ;ccc -c $(CFLAGS) $<

After the makefile is created, a simple "make filename" will begin
the program compilation process. The filename is the actual filename for the program. For the above makefile, "make vision" should be issued instead. A more general makefile which incorporates many program modules can be created with the same suggested approach.
APPENDIX B

Program Execution on the System 14

The concurrent program can be run on either the Simulator mode or the actual hardware mode. Running the concurrent program on the Simulator allows one to debug the program with the single-process (amdbx) as well as the multi-process (mpdbx) debuggers. Any error occurs can be corrected during this simulation process so as to ensure a smooth run on the System 14.

B.1 Program execution on the Simulator

Steps which are needed to execute the concurrent program on the Simulator are presented below.

1. tosim (if not already in the Sim mode).

2. sim -n N visionp.exe, where N is the number of processors employed and visionp.exe is the executable parent program (in machine code).

3. sim -n N -d P visionp.exe, where P is the processor number whose process is to be debugged. In this case, the parent process number is -1 and the child process numbers are similar to the childnum ranging from 0 through n-1. This command is used for single process debugging. For direct execution without debugging, the command given in step 2 should be used
B.2 Program execution on the System 14 hardware

Steps which are needed to run the concurrent program in the Node mode are given below.

1. tonode (if not already in node mode).

2. nreset -r s14. This command resets all the nodes in the concurrent processor. The -r option is to request reset with retry so as to make sure that all nodes have been reset correctly.

3. visionp.exe s14. This is the concurrent program execution command. There are two options which can be specified to obtain additional information during downloading process (-v) and to cause the system to use the serial port for node communications.
APPENDIX C

Special Purpose Timer Routine

The following are the parent and child timer routines which can be used for time measurement of the parallel algorithms. These routines can be used to measure the elapsed time (CPU or real) between certain events. However, because the timing results obtained are dependent on the host computer workload at the time of measurement, their accuracies for short time measurement are poor.

/*
 * (C) Copyright September 1986. AMETEK Inc./Computer Research Division
 * David S. Lim
 *
 * General Purpose timer routines
 */
#include <sys/time.h>
#include <sys/resource.h>

typedef int void;

static long zero_sec, zero_usec;
static int timerflag;

/*
 * zero_timer( flag ) - This routine zeroes the timer. The flag indicates
 * whether you want to time by real time, or cpu time.
 * flag = 0 means by real elapsed time, otherwise by cpu time.
 */
void
zero_timer ( flag )
int flag;
{
    struct timeval tp;
struct timezone tzp;
struct rusage ru;

timerflag = flag;
if (timerflag == 0) {
    gettimeofday(&tp, &tzp);
    zero_sec = tp.tv_sec;
    zero_usec = tp.tv_usec;
} else {
    getrusage(RUSAGE_SELF, &ru);
    zero_sec = ru.ru_utime.tv_sec + ru.ru_stime.tv_sec;
    zero_usec = ru.ru_utime.tv_usec + ru.ru_stime.tv_usec;
}

/*
 * read_timer() - reads the elapsed time since the timer was zeroed.
 * This routine returns the elapsed time in seconds.
 */

void read_timer(time)
double  *time;
{
    struct timeval tp;
    struct timezone tzp;
    struct rusage ru;
    long cur_sec, cur_usec;
    long diff;

    if (timerflag == 0) {
        gettimeofday(&tp, &tzp);
        cur_sec = tp.tv_sec;
        cur_usec = tp.tv_usec;
    } else {
        getrusage(RUSAGE_SELF, &ru);
        cur_sec = ru.ru_utime.tv_sec + ru.ru_stime.tv_sec;
        cur_usec = ru.ru_utime.tv_usec + ru.ru_stime.tv_usec;
    }

diff = cur_usec - zero_usec;
if (cur_usec < zero_usec) {
    diff += 1000000;
    cur_sec --;
    cur_usec += 1000000;
}


```c
/*
 * (C) Copyright September 1986. AMETEK Inc./Computer Research Division
 *    David S. Lim
 *
 * General Purpose timer routines for the nodes
 */
#define CLOCKRATE 2.5    /* clock rate in Mhz */
typedef int void;    /* for Lattice 1.24 this statement is req'd */
                    /* remove this line for Lattice 3.1 */

/*
 * zero_timer() - This routine zeroes the timer.
 */
void
zero_timer()
{
    long t_zst ();
    t_zst ();
}

/* read_timer() - reads the elapsed time since the timer was zeroed.
   
   This routine returns the elapsed time in seconds.
 */
void
read_timer ( timer )
double   *time;
{
    long t_rd ();
    long count;

    count = t_rd ();
    *time = count * 1.0e-6 / CLOCKRATE;
}
APPENDIX D

Communication Calls in User Topology Interfaces

The summary of the host-to-node and node-to-node communication calls using user topology interface routines [16] is presented as follows:

## D.1 Host-to-Node Communication Calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sndx to rcvx</td>
<td>The host sends the same scalar variable to all the nodes.</td>
</tr>
<tr>
<td>sndax to rcvax</td>
<td>The host sends the same array variable to all the nodes.</td>
</tr>
<tr>
<td>c_sndx to c_rcvx</td>
<td>The host sends a scalar variable to the corner node.</td>
</tr>
<tr>
<td>c_rcvx to c_sndx</td>
<td>The host receives a scalar variable from the corner node.</td>
</tr>
<tr>
<td>c_sndax to c_rcvax</td>
<td>The host sends an array variable to the corner node.</td>
</tr>
<tr>
<td>c_rcvax to c_sndax</td>
<td>The host receives an array variable to the corner node.</td>
</tr>
</tbody>
</table>
variable from the corner node.

The host downloads a single term from an array to each of the nodes.

The host downloads a fixed number of terms from an array to each of the nodes.

The host downloads a variable number of terms from an array to each of the nodes.

The host uploads a scalar variable from each of the nodes.

The host uploads a fixed-size array from each of the nodes.

The host uploads a variable-size array from each of the nodes.

### D.2 Node-to-Node Communication Calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sndx</td>
<td>rcvx</td>
</tr>
</tbody>
</table>

A node passes a scalar variable to a neighbor.
A node passes an array variable to a neighbor.

ALL NODES

shftx

The nodes shift the first term of an array in a specified direction.

shftax

The nodes shift a specified number of terms of an array in a specified direction.

skpx

The nodes shift the first term of an array in a specified direction to a node a specified number of neighbors away.

skpax

The nodes shift a specified number of terms of an array in a specified direction to a node a specified number of neighbors away.

brdcstx

A node broadcasts a scalar variable to all the other nodes.
<table>
<thead>
<tr>
<th>brdcstax</th>
<th>A node broadcasts an array variable to all other nodes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>globalx</td>
<td>Combines scalar data from all nodes using a specified function.</td>
</tr>
<tr>
<td>globalax</td>
<td>Combines array data from all nodes using a specified function.</td>
</tr>
</tbody>
</table>
Sequential and Concurrent Program Listings

The order of the program listings (starting from next page) is as follows:

1. Sequential program.
2. Concurrent program in ring topology.
3. Concurrent program in nearest neighbor topology.
/*
 * Program: Sequential program sequent.c
 * Purpose: Sequential implementation of edge detection, thresholding,
 * histogramming, thinning, and noise cleaning algorithms.
 * Written by: LIM CHOON KEE June 1988
 */

#include <stdio.h>
#include <math.h>
#include "timerp.c"

/* Preprocessor definition for thinning */

#define O(m,n) (bin[m+1][n+1] == 100)
#define E(m,n) (bin[m+1][n+2] == 100)
#define W(m,n) (bin[m+1][n] == 100)
#define S(m,n) (bin[m+2][n+1] == 100)
#define N(m,n) (bin[m][n+1] == 100)
#define SE(m,n) (bin[m+2][n+2] == 100)
#define SW(m,n) (bin[m+2][n] == 100)
#define NE(m,n) (bin[m][n+2] == 100)
#define NW(m,n) (bin[m][n] == 100)
#define T1(m,n) (bin[m+1][n+3] == 0)
#define T2(m,n) (bin[m+3][n+1] == 0)
#define retain(m,n) (thin[m+1][n+1] = 100)
#define remove(m,n) (thin[m+1][n+1] = 0)

/*
 * Global array declarations
 *
 * image : original image array
 * pro    : edge_detected image array
 * bin    : thresholded binary image array
 * thin   : thinned image array
 * clean  : noise-clean image array
 */
static unsigned char image[512][512], pro[512][512];
static unsigned char bin[512][512], thin[512][512];
static unsigned char clean[512][512];

main()
{

  /*
  * Variable declarations
  *
  * sec : time variable
  * fname : input file name array
  * i, j : indices
  * hist : histogram array
  * thrsh1, thrsh2 : upper and lower threshold values
  * fp1...fp6 : various file pointers
  */

  double sec;
  char fname[30];
  register int i, j;
  int hist[256], thrsh1, thrsh2, temp;

  /*
  * Query for input image file and threshold values
  */

  printf("Enter input file name:");
  scanf("%s", fname);
  printf("Enter threshold value 1:");
  scanf("%d", &thrsh1);
  printf("Enter threshold value 2:");
  scanf("%d", &thrsh2);

  /*
  * Open input file for reading into image array
  */

  if ((fp1 = fopen(fname, "r")) == NULL) {
    fprintf(stderr, "Error opening %s for input\n", fname);
    exit(1);
  }
fread(image,512*512,1,fp1);
fclose(fp1); /*
 * Timer initialization
 */
zero_timer(0);

/*
 * Zero the edge, thinned, and filtered image frames
 */
for (j = 0; j < 512; ++j) {
    pro[0][j] = pro[511][j] = 0;
    pro[j][0] = pro[j][511] = 0;
    thin[0][j] = thin[510][j] = thin[511][j] = 0;
    thin[j][0] = thin[j][510] = thin[j][511] = 0;
    clean[0][j] = clean[510][j] = clean[511][j] = 0;
    clean[j][0] = clean[j][510] = clean[j][511] = 0;
}

/*
 * Edge detection using Sobel operator
 */
Sobel();

/*
 * Histogram initialization
 */
for (i = 0; i < 256; ++i)
    hist[i] = 0;

/*
 * Histogram generation & binary image
 * generation with thresholding
 */
for (i = 0; i < 512; ++i)
    for (j = 0; j < 512; ++j) {
hist[pro[i][j]] = ++hist[pro[i][j]];
bin[i][j] = (pro[i][j] > thrsh1 && pro[i][j] < thrsh2) ? 100 : 0;

/*
 * Thinning operation
 */

for (i = 0; i < 509; ++i)
  for (j = 0; j < 509; ++j)
    Thinning(i,j);

/*
 * Noise Cleaning
 */

Clean();

/*
 * Terminate timer
 */

read_timer(&sec);

printf("\nTotal Elapsed Time: %f secs\n", sec);

/*
 * Open output image files for processed images writing
 */

fp2 = fopen("histo.seq","w");

for (i = 0; i < 256; ++i)
  fprintf(fp2,"%3d%8d\n",i,hist[i]);

fp3 = fopen("output.img","w");
fwrite(pro,512*512,1,fp3);

fp4 = fopen("binary.img","w");
fwrite(bin,512*512,1,fp4);

fp5 = fopen("thin.img","w");
fwrite(thin,512*512,1,fp5);
fp6 = fopen("clean.img", "w");
fwrite(clean, 512*512, 1, fp6);
fclose(fp2);
fclose(fp3);
fclose(fp4);
fclose(fp5);
fclose(fp6);

/*
* Sobel operator subroutine
*/
Sobel()
{
    int m, n, Gx, Gy;

    /*
    * m, n : indices
    * Gx, Gy : horizontal and vertical gradients
    */
    for (m = 0; m < 510; ++m)
    for (n = 0; n < 510; ++n) {
       Gx = image[m][n] + 2*image[m][n+1] + image[m][n+2] -
           image[m+2][n] - 2*image[m+2][n+1] - image[m+2][n+2];
       Gy = image[m][n] + 2*image[m+1][n] + image[m+2][n] -
           image[m][n+2] - 2*image[m+1][n+2] - image[m+2][n+2];
       pro[m+1][n+1] = abs(Gx) + abs(Gy);
    }

    /*
    * Thinning subroutine
    */
    Thinning(m, n)
    int m, n;
/* 
 * m, n: indices
 */
{
 if (O(m,n)) {
   if (E(m,n)) {
     if (W(m,n)) {
       if (S(m,n)) {
         if (N(m,n))
           retain(m,n);
         else {
           if (NE(m,n))
             retain(m,n);
           else {
             if (NW(m,n))
               retain(m,n);
             else {
               if (T2(m,n))
                 retain(m,n);
               else
                 remove(m,n);
             }
           }
         }
       }
     }
   }
 }
 else {
   if (N(m,n)) {
     if (SE(m,n))
       retain(m,n);
     else {
       if (SW(m,n))
         retain(m,n);
       else
         remove(m,n);
     }
   }
 else
   retain(m,n);
 }
 else {
}
if (S(m,n)) {
  if (N(m,n)) {
    if (NW(m,n))
      retain(m,n);
    else {
      if (SW(m,n))
        retain(m,n);
      else {
        if (T1(m,n))
          retain(m,n);
        else
          remove(m,n);
      }
    }
  }
}
else {
  if (NW(m,n))
    retain(m,n);
  else {
    if (T2(m,n))
      retain(m,n);
    else {
      if (T1(m,n))
        retain(m,n);
      else
        remove(m,n);
    }
  }
}
else {
  if (N(m,n)) {
    if (SW(m,n))
      retain(m,n);
    else {
      if (T1(m,n))
        retain(m,n);
      else
        remove(m,n);
    }
  }
}
else
  retain(m,n);
else {
  if (W(m,n)) {
    if (S(m,n)) {
      if (N(m,n)) {
        if (NE(m,n))
          retain(m,n);
        else {
          if (SE(m,n))
            retain(m,n);
          else
            remove(m,n);
        }
      }
    }
  }
  else {
    if (NE(m,n))
      retain(m,n);
    else {
      if (T2(m,n))
        retain(m,n);
      else
        remove(m,n);
    }
  }
}
else {
  if (N(m,n)) {
    if (SE(m,n))
      retain(m,n);
    else
      remove(m,n);
  }
  else
    retain(m,n);
}
else
  retain(m,n);
remove(m,n);
}

/**< *
 * Noise Cleaning Subroutine
 */

Clean()
{
    int m, n, sum;

    /**< *
     * m, n : indices
     * sum : summing variable
     */

    for (m = 0; m < 510; ++m)
        for (n = 0; n < 510; ++n) {
            sum = thin[m][n] + thin[m][n+1] + thin[m][n+2] +
                 thin[m+1][n] + thin[m+1][n+2] + thin[m+2][n] +
                 thin[m+2][n+1] + thin[m+2][n+2];

            if (sum == 800 && thin[m+1][n+1] == 0)
                clean[m+1][n+1] = 100;
            else if (sum == 0 && thin[m+1][n+1] == 100)
                clean[m+1][n+1] = 0;
            else
                clean[m+1][n+1] = thin[m+1][n+1];
        }
}
/*
 * Program : Parent program tringp.c (Ring Topology)
 * 
 * Written by : LIM CHOON KEE  June 1988
 * */

#include <stdio.h>
#include "ringp.h"

/*
 * Global array declarations
 * 
 * image : original image array
 * clean : filtered image array
 */

static unsigned char image[512][512];
static unsigned char clean[512][512];

user_main()
{

/*
 * Variable declarations
 * 
 * sec : time variable
 * fname : input file array
 * i, j : indices
 * hist : histogram array
 * thrsh1, thrsh2 : threshold values
 * fp1, fp2, fp3 : various file pointers
 */

double sec;
char fname[30];
register int i, j;
int hist[256], thrsh1, thrsh2;
FILE *fp1, *fp2, *fp3;

/*
 * Ring topology initialization
 */
ring(SAME,"tringc.exe");

/*
 * Query for input file and threshold values
 */

printf("Enter input file name :\n");
scanf("%s", fname);
printf("Enter threshold value 1 :\n");
scanf("%d", &thrsh1);
printf("Enter threshold value 2 :\n");
scanf("%d", &thrsh2);

/*
 * Open input file for reading into image array
 */

if ((fp1 = fopen(fname,"r")) == NULL) {  
  fprintf(stderr,"Error opening %s for input\n",fname);
  exit(1);
}

fread(image,512*512,1,fp1);
fclose(fp1);

/*
 * Timer initialization
 */

zero_timer(0);

/*
 * Download subimages
 */

dnldac(image,32*512);

/*
 * Receive accumulated histogram from corner node
 */

c_rcvai(hist,256);
/*
* Broadcast threshold values based on the return histogram
*/

sndi(thrsh1,ALL);
sndi(thrsh2,ALL);

/*
* Receive filtered image from children
*/

upldac(clean,32*512);

/*
* Terminate timer
*/

read_timer(&sec);

printf("Total Elapsed Time is: %lf secs\n", sec);

/*
* Write returned histogram and processed image to files
*/

if ((fp2 = fopen("ringhisto","w")) == NULL) {
    fprintf(stderr,"Error opening ringhisto for output\n");
    exit(1);
}

for (i = 0; i < 256; ++i)
    fprintf(fp2,"%3d%8d\n", i, hist[i]);

if ((fp2 = fopen("ringclean","w")) == NULL) {
    fprintf(stderr,"Error opening ringclean for output\n");
    exit(1);
}

fclose(fp2);
fclose(fp3);
/*
* Program : Child program tringc.c (Ring Topology)
* Purpose : Parallel hypercube machine implementation of Sobel edge
detection, histogram generation, thresholding, thinning,
and noise cleaning algorithms with ring topology
*
* Written by : LIM CHOON KEE  June 1988
*
*/

#include "ringc.h"

/*************************************************************************
 * Preprocessor definition for thinning operation
*************************************************************************/

#define O(m,n) (bin[m+1][n+1] == 100)
#define E(m,n) (bin[m+1][n+2] == 100)
#define W(m,n) (bin[m+1][n] == 100)
#define S(m,n) (bin[m+2][n+1] == 100)
#define N(m,n) (bin[m][n+1] == 100)
#define SE(m,n) (bin[m+2][n+2] == 100)
#define SW(m,n) (bin[m+2][n] == 100)
#define NE(m,n) (bin[m][n+2] == 100)
#define NW(m,n) (bin[m][n] == 100)
#define T1(m,n) (bin[m+1][n+3] == 0)
#define T2(m,n) (bin[m+3][n+1] == 0)
#define retain(m,n) (thin[m+1][n+1] = 100)
#define remove(m,n) (thin[m+1][n+1] = 0)

/*************************************************************************
 * Global array declarations
*************************************************************************/

* pro : subimage
* bin : thresholded subimage array
* thin : thinned subimage array
*************************************************************************/

static char pro[32][512], bin[35][512], thin[34][512];

user_main()
{

/*     * Variables declaration     *     */
char sub[34][512];
register int i, j, count;
int Gx, Gy, sum, thrsh1, thrsh2;
int hist[256], histl[256], histr[256];

/*     * Each node receives its corresponding 32x512 subimage     */
dnddac(&sub[1][0],32*512);

/*     * Boundary info communications for edge detection     */
if (childnum != 15) {
    sndac(&sub[32][0],512,RIGHT);
    rcvac(&sub[33][0],512,RIGHT);
}
if (childnum != 0) {
    rcvac(sub,512,LEFT);
    sndac(&sub[1][0],512,LEFT);
}

/*     * Sobel edge detection     */
for (i = 0; i < 32; ++i)
    for (j = 0; j < 510; ++j)
{ 
    Gx = (sub[i][j] - sub[i+2][j]) + 2*(sub[i][j+1] - sub[i+2][j+1])
    + (sub[i][j+2] - sub[i+2][j+2]);

    Gy = (sub[i][j] - sub[i][j+2]) + 2*(sub[i+1][j] - sub[i+1][j+2])
    + (sub[i+2][j] - sub[i+2][j+2]);

    pro[i][j+1] = abs(Gx) + abs(Gy);
}

/* 
* Convert the frame to background 
*/

for (i = 0; i < 32; ++i)
    pro[i][0] = pro[i][511] = 0;

/* 
* Histogram initialization 
*/

for (j = 0; j < 256; ++j)
    hist[j] = 0;

/* 
* Histogram generation 
*/

for (i = 0; i < 32; ++i)
    for (j = 0; j < 512; ++j)
        hist[pro[i][j]] = ++hist[pro[i][j]];

/* 
* Summation of partial histogram from neighbors 
*/

for (count = 0; count < 8; ++count) {
    if (childnum == 0) {
        rcvai(hist1,256,LEFT);
        rcvai(histr,256,RIGHT);
    }

    if (count < 7) {
        for (i = 0; i < 256; ++i)
hist[i] = hist[i] + histl[i] + histr[i];
}

else {
    for (i = 0; i < 256; ++i)
        hist[i] = hist[i] + histl[i];
}

if (childnum < 8 && childnum != 0) {
    sndai(hist,256,LEFT);
    rcvai(hist,256,RIGHT);
}

if (childnum > 8) {
    sndai(hist,256,RIGHT);
    rcvai(hist,256,LEFT);
}

if (childnum == 8) {
    sndai(hist,256,RIGHT);
    sndai(hist,256,LEFT);
}
}

/*
 * Send histogram to host from corner node
 */
c_sndai(hist,256);

/*
 * Receive threshold values from parent
 */
rcvi(thrsh1,PAR);
rcvi(thrsh2,PAR);

/*
 * Thresholding
 */
for (i = 0; i < 32; ++i)
for(j = 0; j < 512; ++j)
bin[i+1][j] = ((pro[i][j]) >= thrsh1 && pro[i][j] <= thrsh2) ? 100 : 0;

/*
 * Boundary info communications for thinning
 */

if (childnum != 15) {
    sndac(&bin[32][0],512,RIGHT);
    rcvac(&bin[33][0],2*512,RIGHT);
}

if (childnum != 0) {
    rcvac(bin,512,LEFT);
    sndac(&bin[1][0],2*512,LEFT);
}

/*
 * Thinning operations
 */

for (i = 0; i < 32; ++i)
    for (j = 0; j < 509; ++j)
        Thinning(i,j);

/*
 * Converting the frame into background
 */

for (i = 0; i < 32; ++i)
    thin[i+1][0] = thin[i+1][510] = thin[i+1][511] = 0;

/*
 * Boundary info communications for noise cleaning
 */

if (childnum != 15) {
    sndac(&thin[32][0],512,RIGHT);
    rcvac(&thin[33][0],512,RIGHT);
}

if (childnum != 0) {
    rcvac(thin,512,LEFT);
sndac(&thin[1][0],512,LEFT);
}

/*
 * Noise cleaning
 */

for (i = 0; i < 32; ++i) {
    for (j = 0; j < 510; ++j) {
        sum = thin[i][j] + thin[i][j+1] + thin[i][j+2] + thin[i+1][j] +
                thin[i+1][j+2] + thin[i+2][j] + thin[i+2][j+1] + thin[i+2][j+2];

        if (sum == 800 && thin[i+1][j+1] == 0)
            sub[i][j+1] = 100;
        else if (sum == 0 && thin[i+1][j+1] == 100)
            sub[i][j+1] = 0;
        else
            sub[i][j+1] = thin[i+1][j+1];
    }

    sub[i][0] = sub[i][511] = 0;
}

/*
 * Upload filtered image to host
 */

upldac(sub,32*512);
}

/*
 * Thinning subroutine
 */

Thinning(m,n)

int m, n;
{
    if (O(m,n)) {
        if (E(m,n)) {
            if (W(m,n)) {
                if (S(m,n)) {
                    if (N(m,n))
                        )
retain(m,n);
else {
    if (NE(m,n))
        retain(m,n);
    else {
        if (NW(m,n))
            retain(m,n);
        else {
            if (T2(m,n))
                retain(m,n);
            else
                remove(m,n);
        }
    }
}
}
else {
    if (N(m,n)) {
        if (SE(m,n))
            retain(m,n);
        else {
            if (SW(m,n))
                retain(m,n);
            else
                remove(m,n);
        }
    }
    else
    retain(m,n);
}
}
else {
    if (S(m,n)) {
        if (N(m,n)) {
            if (NW(m,n))
                retain(m,n);
            else {
                if (SW(m,n))
                    retain(m,n);
                else {
                    if (T1(m,n))
                        retain(m,n);
                    else
                        remove(m,n);
                }
            }
        }
    }
}

remove(m,n);
}
}
else {
  if (NW(m,n))
    retain(m,n);
  else {
    if (T2(m,n))
      retain(m,n);
    else {
      if (T1(m,n))
        retain(m,n);
      else
        remove(m,n);
    }
  }
}
else {
  if (N(m,n)) {
    if (SW(m,n))
      retain(m,n);
    else {
      if (T1(m,n))
        retain(m,n);
      else
        remove(m,n);
    }
  }
  else
  retain(m,n);
}
}
else {
  if (W(m,n)) {
    if (S(m,n)) {
      if (N(m,n)) {
        if (NE(m,n))
          retain(m,n);
        else {
          if (SE(m,n))
            
        }
ret


/* 
* Program : Parent program mes2p.c (Nearest Neighbor Topology) 
* 
* Written by : LIM CHOON KEE June 1988 
*/ 

#include <stdio.h> 
#include "hnp.h" 

/* 
* Global array declarations 
*/ 
static unsigned char image[512][512], pro[512][512]; 
static unsigned char clean[512][512], buf[4][512]; 

user_main() 
{ 

/* 
* Variable declarations 
*/ 

double sec; 
char fname[20]; 
register int i; 
int hist[256], thrsh1, thrsh2; 
FILE *fp1, *fp2, *fp3; 

/* 
* Nearest neighbor topology initialization 
*/ 

nn(SAME,4,1,"mes2c.exe"); 

/* 
* Query for input file and threshold values 
*/ 

printf("Enter input file name :"); 
scanf("%s", fname); 
printf("Enter threshold value1 :"); 
scanf("%d", &thrsh1);
printf("Enter threshold value2 :");
scanf("%d", &thrsh2);

/*
 * Open input file for reading into array image
 */
if ((fp1 = fopen(fname,"r")) == NULL) {
    fprintf(stderr,"Error opening %s for input\n", fname);
    exit(1);
}

fread(image, 512*512, 1, fp1);
fclose(fp1);

/*
 * Image rearrangement for download purpose
 */
array_to_strip(image, pro);

/*
 * Timer initialization
 */
zero_timer(0);

/*
 * Download image
 */
dnldac(pro, 128*128);
/*
 * Receive histogram from corner node
 */
c_rcvai(hist, 256);
/*
 * Broadcast threshold values to children
 */
sndi(thrsh1, ALL);
sindi(thrsh2,ALL);

/*
 * Receive filtered image from children
 */

upldac(clean,128*128);

/*
 * Terminate timer
 */

read_timer(&sec);

printf("\nTotal Elapsed Time is: %f secs\n",sec);

/*
 * Write histogram to file
 */

    if (((fp2 = fopen("meshhisto","w")) == NULL) {
        printf(stderr,"Error opening meshhisto for input\n");
        exit(1);
    }

/*
 * Rearrange filtered image and write to file
 */

    strip_to_array(clean,image);

    if (((fp3 = fopen("meshclean","w")) == NULL) {
        printf(stderr,"Error opening meshclean for input\n");
        exit(1);
    }

    fwrite(image,512*512,1fp3);
    fclose(fp3);
}

/*
 * Image rearrangement subroutine
 *
 * This subroutine is used to convert the image which
* consists of sixteen of the 32*512 subimages into
* image which is formed by sixteen 128*128 subimages
*/

strip_to_array(in, out)
unsigned char in[[512], out[[512]];
{
    register int i, j, k, l, m, n, ix, iy, i1, i2;
    int count, const;

    i1 = count = 0;
    const = 384;

    for(l = 0; l < 32; ++l) {
        for(k = 0; k < 4; ++k) {
            ix = iy = 0;
            for(i = i1; i < 512; i += 32) {
                for(j = k*128; j < (k*128+128); ++j) {
                    buf[ix][iy] = in[i][j];
                    ++iy;
                }
                ++count;
            }
            if(count != 0 && (count%4)==0) {
                ++ix;
                iy = 0;
            }
        }
    }

    i2 = const;
    for(m = 0; m < 4; ++m) {
        for(n = 0; n < 512; ++n) {
            out[i2][n] = buf[m][n];
            i2 -= 128;
        }
        ++const;
    }
    ++i1;
}

/*
* Image rearrangement subroutine
*
* This subroutine is used to convert the image which
* consists of sixteen of the 128*128 subimages into
* image which is formed by sixteen 32*512 subimages
*/

array_to_strip(in,out)
unsigned char in[512][512], out[512][512];
{
    register int i, j, k, l, ix, jx, i1, j1;

    i1 = 384;
    ix = jx = j1 = 0;

    for(l=0;l<4;++l) {
        for(k=0;k<4;++k) {
            for(i=i1;i<i1+128;++i) {
                for(j=j1;j<j1+128;++j) {
                    out[ix][jx]=in[i][j];
                    ++jx;
                    if(jx==512){
                        ++ix;
                        jx=0;
                    }
                }
            }
            j1=j1-128;
        }
        i1=i1-128;
        j1 += 128;
    }
    i1 -= 128;
    j1=0;
}
/*
* Program : Child program mes2c.c (Nearest Neighbor Topology)
*
* Purpose : Parallel hypercube machine implementation of Sobel edge
detection, histogram generation, thresholding, thinning,
* and noise cleaning algorithms with mesh topology
*
* Written by : LIM CHOON KEE  June 1988
*
*/

#include "nnc.h"

/*
* Preprocessor definition for thinning
*/

#define O(m,n) (bin[m+1][n+1] == 100)
define E(m,n) (bin[m+1][n+2] == 100)
define W(m,n) (bin[m+1][n] == 100)
define S(m,n) (bin[m+2][n+1] == 100)
define N(m,n) (bin[m][n+1] == 100)
define SE(m,n) (bin[m+2][n+2] == 100)
define SW(m,n) (bin[m+2][n] == 100)
define NE(m,n) (bin[m][n+2] == 100)
define NW(m,n) (bin[m][n] == 100)
define T1(m,n) (bin[m+1][n+3] == 0)
define T2(m,n) (bin[m+3][n+1] == 0)
define retain(m,n) (thin[m+1][n+1] = 100)
define remove(m,n) (thin[m+1][n+1] = 0)

/*
* Global array declarations
*/

static char pro[128][128], thin[130][130];
static char bin[131][131], sbuf[131], dbuf[131];
static char bufs[2][131], bufd[2][131];

user_main()
{
/*
* Variables declaration
*/

char sub[130][130];
register int ix, i, j, m, n;
int Gx, Gy, thrsh1, thrsh2, sum;
int hist[256], temhist[256];

/*@ */

*/
* Receive threshold values from parent
*/

rcvi(thrsh1,PAR);
rcvi(thrsh2,PAR);

/*@ */

* Receive subimage and transfer into larger array
*/

dndac(pro,128*128);

for(i=0;i<128;++i)
  for(j=0;j<128;++j)
    sub[i+1][j+1] = pro[i][j];

/*@ */

* Interprocessor communications for edge detection
*/

sndac(&sub[1][1],128,NORTH);
rcvac(&sub[129][1],128,SOUTH);

sndac(&sub[128][1],128,SOUTH);
rcvac(&sub[0][1],128,NORTH);

if (childnum % 4 != 3) {
  for (i = 0; i < 130; ++i)
    sbuf[i] = sub[i][128];
}

sndac(sbuf,130,EAST);
rcvac(dbuf,130,WEST);
for (i = 0; i < 130; ++i)
    sub[i][0] = dbuf[i];

    if (childnum % 4 != 0) {
        for (i = 0; i < 130; ++i)
            sbuf[i] = sub[i][1];
    }

sndac(sbuf,130,WEST);
rcvac(dbuf,130,EAST);

    for (i = 0; i < 130; ++i)
        sub[i][129] = dbuf[i];

/*
   * Sobel edge detection
   */

    for (i = 0; i < 128; ++i)
        for (j = 0; j < 128; ++j)
        {
            Gx = (sub[i][j] - sub[i+2][j]) + 2*(sub[i][j+1] - sub[i+2][j+1])
                + (sub[i][j+2] - sub[i+2][j+2]);

            Gy = (sub[i][j] - sub[i][j+2]) + 2*(sub[i+1][j] - sub[i+1][j+2])
                + (sub[i+2][j] - sub[i+2][j+2]);

            pro[i][j] = abs(Gx) + abs(Gy);
        }

/*
   * Histogram initialization
   */

    for (i = 0; i < 256; ++i)
        hist[i] = 0;

/*
   * Histogram generation & thresholding
   */

    for (i = 0; i < 128; ++i)
        for (j = 0; j < 128; ++j) {

for (i = 0; i < 3; ++i) {
    sndai(hist, 256, WEST);
    rcvai(temhist, 256, EAST);
    if (childnum % 4 == 0) {
        for (j = 0; j < 256; ++j)
            hist[j] = hist[j] + temhist[j];
    }
}

for (i = 0; i < 3; ++i) {
    if (childnum % 4 == 0) {
        sndai(hist, 256, SOUTH);
        rcvai(temhist, 256, NORTH);
    }
    if (childnum == 0) {
        for (j = 0; j < 256; ++j)
            hist[j] = hist[j] + temhist[j];
    }
}

/*
 * Intercommunications for thinning
 */
sndac(&bin[1][1], 128, NORTH);
rcvvac(&bin[129][1], 128, SOUTH);
sndac(&bin[2][1], 128, NORTH);
rcvvac(&bin[130][1], 128, SOUTH);
sndac(&bin[128][1], 128, NORTH);
if (childnum % 4 != 3) {
    for (i = 0; i < 131; ++i)
        sbuf[i] = bin[i][128];
}

sndac(sbuf,131,EAST);
rcvac(dbuf,131,WEST);

for (i = 0; i < 131; ++i)
    bin[i][0] = dbuf[i];

if (childnum % 4 == 0) {
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 131; ++j)
            buf[i][j] = bin[j][i+1];
}

sndac(bufs,2*131,WEST);
rcvac(bufd,2*131,EAST);

ix = 0;
for (j = 129; j < 131; ++j) {
    for (i = 0; i < 131; ++i)
        bin[i][j] = bufs[ix][i];
    ++ix;
}

for (i = 0; i < 128; ++i)
    for (j = 0; j < 128; ++j) {
        Thinning(i,j);
        pro[i][j] = thin[i+1][j+1];
    }

/*
 * Intercommunications for noise cleaning
 */

sndac(&thin[1][1],128,NORTH);
rcvac(&thin[129][1],128,SOUTH);
sndac(&thin[128][1], 128, SOUTH);
rcvac(&thin[0][1], 128, NORTH);

if (childnum % 4 != 3) {
    for (i = 0; i < 130; ++i)
        sbuf[i] = thin[i][128];
}

sndac(sbuf, 130, EAST);
rvcac(dbuf, 130, WEST);

for (i = 0; i < 130; ++i)
    thin[i][0] = dbuf[i];

if (childnum % 4 != 0) {
    for (i = 0; i < 130; ++i)
        sbuf[i] = thin[i][1];
}

sndac(sbuf, 130, WEST);
rvcac(dbuf, 130, EAST);

for (i = 0; i < 130; ++i)
    thin[i][129] = dbuf[i];

/*
 * Noise cleaning
 */

for (i = 0; i < 128; ++i)
    for (j = 0; j < 128; ++j) {
        sum = thin[i][j] + thin[i][j + 1] + thin[i][j + 2] + thin[i + 1][j] + thin[i + 1][j + 2] + thin[i + 2][j] + thin[i + 2][j + 1] + thin[i + 2][j + 2];

        if (sum == 800 && thin[i + 1][j + 1] == 0)
            pro[i][j] = 100;
        else if (sum == 0 && thin[i + 1][j + 1] == 100)
            pro[i][j] = 0;
        else
            pro[i][j] = thin[i + 1][j + 1];
    }

/*
* Upload filtered image to parent
*/

upldac(pro,128*128);
}

/*
 * Thinning subroutine
*/

Thinning(m,n)

int m, n;
{
  if (O(m,n)) {
    if (E(m,n)) {
      if (W(m,n)) {
        if (S(m,n)) {
          if (N(m,n))
            retain(m,n);
          else {
            if (NE(m,n))
              retain(m,n);
            else {
              if (NW(m,n))
                retain(m,n);
              else {
                if (T2(m,n))
                  retain(m,n);
                else
                  remove(m,n);
              }
            }
          }
        }
      }
    }
  }
  else {
    if (N(m,n)) {
      if (SE(m,n))
        retain(m,n);
      else {
        if (SW(m,n))
          retain(m,n);
        else
remove(m,n);
}
}
else
  retain(m,n);
}
}
else {
  if (S(m,n)) {
    if (N(m,n)) {
      if (NW(m,n))
        retain(m,n);
      else {
        if (SW(m,n))
          retain(m,n);
        else {
          if (T1(m,n))
            retain(m,n);
          else
            remove(m,n);
        }
      }
    }
  }
  else {
    if (NW(m,n))
      retain(m,n);
    else {
      if (T2(m,n))
        retain(m,n);
      else {
        if (T1(m,n))
          retain(m,n);
        else
          remove(m,n);
      }
    }
  }
  else {
    if (N(m,n)) {
      if (SW(m,n))
        retain(m,n);
      else {
        
      }
    }
  }
}
if (T1(m,n))
    retain(m,n);
else
    remove(m,n);
}
else
    retain(m,n);
}
}

else {
    if (W(m,n)) {
        if (S(m,n)) {
            if (N(m,n)) {
                if (NE(m,n))
                    retain(m,n);
                else {
                    if (SE(m,n))
                        retain(m,n);
                    else
                        remove(m,n);
                }
            }
        }
    }
    else {
        if (NE(m,n))
            retain(m,n);
        else {
            if (T2(m,n))
                retain(m,n);
            else
                remove(m,n);
        }
    }
}
else {
    if (N(m,n)) {
        if (SE(m,n))
            retain(m,n);
        else
            remove(m,n);
    }
}
else
retain(m,n);
}
}
else
retain(m,n);
}
}
else
remove(m,n);