

5-31-1991

Parallel implementation of spatial domain image processing algorithms on the mesh connected computer simulator

Adury Devi Prasad
New Jersey Institute of Technology

Follow this and additional works at: <https://digitalcommons.njit.edu/theses>



Part of the [Electrical and Electronics Commons](#)

Recommended Citation

Prasad, Adury Devi, "Parallel implementation of spatial domain image processing algorithms on the mesh connected computer simulator" (1991). *Theses*. 2599.
<https://digitalcommons.njit.edu/theses/2599>

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be “used for any purpose other than private study, scholarship, or research.” If a user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of “fair use” that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select “Pages from: first page # to: last page #” on the print dialog screen

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

2/ Parallel Implementation of Spatial Domain Image
Processing Algorithms on the Mesh Connected
Computer Simulator

by

1/ Adury Devi Prasad
//

*Thesis submitted to the faculty of the Graduate School of
the New Jersey Institute of Technology in partial
fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering*

1991

Approval Sheet

Title: Parallel Implementation of Spatial Domain
Image Processing Algorithms on the Mesh
Connected Computer Simulator

Name of Candidate: Adury Devi Prasad
Master of Science in Electrical Engineering, 1991

Thesis & Abstract Approved
by the Examining Committee:

Dr. John Carpinelli, Advisor	Date
Assistant Professor	
Department of Electrical and Computer Engineering	

Dr. Sotirios Ziavras	Date
Assistant Professor	
Department of Electrical and Computer Engineering	

Dr. Anthony Robbi	Date
Associate Professor	
Department of Electrical and Computer Engineering	

New Jersey Institute of Technology, Newark, New Jersey.

VITA

Name: Adury Devi Prasad

Permanent Address:

Degree and Date to be Conferred: Master of Science in
Electrical Engineering, 1991.

Date of Birth:

Place of Birth:

Secondary Education: Nrupatunga High School, 1981.
Hyderabad, INDIA, 1981.

Collegiate Institutions Attended	Dates	Degree	Year
Nrupatunga Junior College	06/81-06/83.	Intermediate	1983.
K.S.R.M. College of Engineering	07/83-06/87.	B.Tech	1987.
New Jersey Institute of Technology	01/89-05/91.	M.S E.E.	1991

Major: Electrical Engineering

ABSTRACT

Title: Parallel Implementation of Spatial Domain Image Processing
Algorithms on the Mesh Connected Computer Simulator

By: Adury Devi Prasad

Thesis Directed by: Dr. John Carpinelli, Department of Electrical and
Computer Engineering.

The performance of conventional computers in image processing applications is known to be inadequate due to the enormous computational requirements. Almost all image processing applications can be implemented in parallel. This thesis is the parallel implementation of neighborhood averaging and edge detection by the Sobel and Laplacian operators on the Mesh Connected Computer Simulator (MCCS), a package developed at NJIT to study the behavior of SIMD machines. Expressions for speedup, utilization and efficiency have been derived. Suggestions for further improvements that can be incorporated in MCCS and its instruction set have been made.

ACKNOWLEDGEMENTS

I would like to express sincere gratitude to my advisor Dr. John Carpinelli, Department of Electrical and Computer Engineering, for his expert guidance and valuable advice. I would like to express my sincere thanks to Mr. Norman J. Van Houten, Director, Department of Health and Environmental Safety, NJIT, for his cooperation in letting me use the office premises during all times. Finally I would like to express my sincere gratitude to my parents for their constant moral support and encouragement.

Contents

1	INTRODUCTION	1
1.1	Background	1
1.2	The Relevance of Parallel Processing in DIP	2
1.3	Motivation and Objectives	4
1.4	Outline	5
2	ESSENTIAL FEATURES OF MCCS AND THE ARCHI- TECTURE	6
2.1	Introduction	6
2.1.1	Basic Operations and Structure of MCCS	6
2.1.2	Simple Array Processor Language	13
2.1.3	Instruction Set and Control Symbols	13
2.1.4	The Control Symbol Set	15
2.2	SIMD Array Processors	17
3	ALGORITHMS	22
3.1	Introduction	22
3.1.1	Spatial Domain Methods	23
3.2	Image Smoothing and Neighborhood Averaging	23
3.2.1	Neighborhood Averaging	24

3.2.2	Algorithm for Neighborhood Averaging	25
3.3	Edge Detection by Sobel Operators and Laplacian	29
3.3.1	Edge Detection	31
3.3.2	Algorithm	35
4	PERFORMANCE EVALUATION OF THE MESH	43
4.1	Common Measures of Performance Evaluation	43
4.2	Common Parameters	44
4.3	Calculations for Neighborhood Averaging Algorithm	46
4.3.1	SPEEDUP	46
4.3.2	UTILIZATION	47
4.3.3	EFFICIENCY	48
4.4	Calculations for the Edge Detection Algorithm	48
4.4.1	SPEEDUP	48
4.4.2	UTILIZATION	49
4.4.3	EFFICIENCY	49
4.5	Discussion	49
5	CONCLUSIONS AND FUTURE DEVELOPMENTS	51
5.1	Conclusions	51
5.2	Discussion	51
A	SAL Program for Neighborhood Averaging	53
B	Neighborhood Averaging on One Pixel Using a Single Pro- cessor	59
C	SAL Program for Edge Detection	62

D Edge Detection on One Pixel Using A Single Processor	65
Bibliography	66

List of Figures

2.1	Main menu of M CCS	8
2.2	The help menu of M CCS	9
2.3	The configuration menu of M CCS	9
2.4	The execution menu of M CCS	10
2.5	The exit menu of M CCS	10
2.6	Block diagram I of M CCS	11
2.7	Block diagram II of M CCS	12
2.8	A typical SIMD mesh	20
2.9	A typical processing element	21
3.1	A square subimage centered at \mathbf{x}_5	24
3.2	Mask for neighborhood averaging	25
3.3	A 6x6 mesh as specified in *.cfg file in M CCS	26
3.4	Original image for neighborhood averaging	29
3.5	Processed image after neighborhood averaging	30
3.6	Mask for computing G_x	33
3.7	Mask for computing G_y	33
3.8	Mask for computing Laplacian (L)	34
3.9	Original image for edge detection	40
3.10	Gradient image in x -direction G_x	40

3.11 Gradient image in y -direction G_y	41
3.12 Gradient image G	41
3.13 Laplacian image L	42

Chapter 1

INTRODUCTION

1.1 Background

The advent of VLSI technology has greatly influenced the field of computer architecture. It contributed to increased device speed and reliability, reductions in hardware size and cost and better performance. The task of achieving high performance is not only dependent upon using faster and more reliable hardware devices but also on major improvements in computer architecture and processing techniques.

The main principle on which all advanced computer architectures are centered is the concept of parallel processing. Hwang and Briggs[1] define parallel processing as an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. One of the main requirements for achieving parallel processing is the development of more capable and cost-effective computer systems.

One of the fields in which the concept of parallel processing has a wide range of applications is Digital Image Processing (DIP). As a result ex-

tensive research is being done to achieve increased speed from both the hardware and software. DIP is required for improvement of the pictorial information for human interpretation and for better machine perception.

1.2 The Relevance of Parallel Processing in DIP

A digital image is a two-dimensional light intensity function $f(x, y)$ where x and y denote the spatial coordinates and the value f at any given point (x, y) denotes the intensity or the brightness level at that point. Every element of the array is called the pixel or pel.

Let us assume that a continuous image $f(x, y)$ is arranged in the form of an $N \times N$ array where each element is a discrete quantity obtained from equally spaced samples over the entire image as in equation 1.1.

$$f(x, y) = \begin{bmatrix} f(0, 0) & f(0, 1) & \cdots & f(0, N-1) \\ f(1, 0) & f(1, 1) & \cdots & f(1, N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(N-1, 0) & f(N-1, 1) & \cdots & f(N-1, N-1) \end{bmatrix} \quad (1.1)$$

If N is the number of pixels and G is the number of brightness or grey levels per pixel then the number, b , of bits required to store a digital image is equal to:

$$b = N \times N \times m \quad (1.2)$$

where $m = \log_2 k$ (k is the total number of grey levels) is the number of bits required to represent a grey level. In general, for an image with 512x512

display points and 256 grey levels (8 bits), 262,144 bytes are required to store the image. This fact is of enormous significance since television images are generated at the rate of 30 interlaced frames per second; it implies the need for systems that can tackle nearly 60 million operations per second.

Under these circumstances, researchers in this field are striving to achieve high performance from their systems. Fortunately it has been found that almost all low level and intermediate level image processing tasks can be decomposed into a set of sub-tasks distributed over the entire image. Such decompositions permit simultaneous processing of different regions of the image, i.e., parallel processing, where each sub-image is served by a simple processor.

All image processing problems can be broadly categorized into the following three classes: low level vision, intermediate level vision and high level vision. Low level image processing operations are identical over all pixels in the image. Each processor basically has to execute the same sequence of instructions over the subimages that are assigned to it. Single Instruction Multiple Data (SIMD) machines are suitable for such tasks as such machines are optimized for parallel neighborhood operations. The data access for each processor is usually restricted to its nearest neighbors.

Intermediate level vision is subsequent processing done on the image after low level vision. Once the manipulation on the pixels is done in low level vision other information like the area of the image and the connectivity of the edges are determined in this step. In most cases SIMD architectures

are suitable for intermediate level vision problems. Hypercubes and other such architectures are also being extensively used for these problems.

High level vision operations involve regions of the image rather than a small neighborhood and require the application of different operations over different regions. Multiple Instruction Multiple Data (MIMD) machines are better suited for such operations.

This thesis is the simulation of an SIMD machine for low level vision for image processing applications using the Mesh Connected Computer Simulator (MCCS), a package for simulating mesh connected multiprocessor systems.

1.3 Motivation and Objectives

The motivation for this thesis is the SIMD machine for low level vision that is being designed by the department of Electrical and Computer Engineering, Indian Institute of Science, Bangalore, India. It is a special purpose architecture to implement relaxation labelling algorithms for image processing. Digital image processing is a challenging field because in the final analysis, one is often trying to “understand” and “improve” a digital image which is just an array of numbers.

The objectives of this thesis are to implement spatial domain image processing algorithms in parallel on MCCS and to suggest improvements that can be incorporated into the package thereby enhancing its utility to the users.

1.4 Outline

The rest of the thesis is organized as follows. Chapter 2 highlights the essential features of the architecture being simulated and that of MCCS, the package being used. The algorithms for various operations performed on the machine and sections of the code written are discussed in Chapter 3. The performance evaluation of the architecture based upon the results obtained is discussed in Chapter 4. Significant results and further improvements that can be incorporated are highlighted in Chapter 5.

Chapter 2

ESSENTIAL FEATURES OF MCCS AND THE ARCHITECTURE

2.1 Introduction

The Mesh Connected Computer Simulator, MCCS, is a PC based software package coded in TURBO C. MCCS is compiled to a command file, which can be run directly from a floppy or hard disk in the DOS environment. The user constructs the mesh by specifying the number of rows and columns and the size of local memory for each processing element. The programs, coded in Simple Array-Processor Language (SAL), can be executed in one of the following three modes: continuous, single step or continuous with break points. The package allows the user to examine the contents of registers and local memory during or after program execution [3].

2.1.1 Basic Operations and Structure of MCCS

The simulator is a menu driven system and is divided into six substructures as follows:

- The cover page which gives information about the authors on the screen.
- The main diagram, which displays the four basic function entries, F1, F2, F3 and F4, which select the help menu, configuration menu, execution menu and exit to DOS, as shown in figure 2.1.
- The Help screen can be accessed by pressing F1 in the main menu. All the pages can be accessed by pressing the PgUp and PgDn keys. The Help screen is shown in figure 2.2.
- The Configuration screen can be accessed by pressing the F2 key in the main menu. This screen is used to specify the configuration of the mesh, which is user defined. This screen is shown in Figure 2.3.
- The Execution screen can be accessed by pressing the F3 key in the main menu. This menu helps the user to execute programs in one of the three modes specified previously. The screen is shown in figure 2.4.
- The Exit screen can be accessed by pressing the F4 key in the main menu. This allows the user to quit M CCS and return to DOS. This screen is shown in figure 2.5.

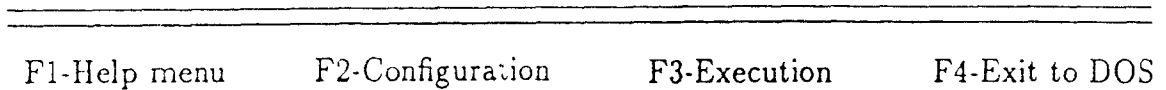


Figure 2.1: Main menu of MCCA

The block diagram of MCCA is shown in figures 2.6 and 2.7 which display all the modules [3].

The maximum size of the mesh that can be simulated using MCCA is 8x8. The maximum size of the local memory for each processing element is 4 Kbytes.

Background:

The Mesh Connected Computer Simulator is a simulator that can run Simple Array processor Language coded programs and at the same time show the array processor operation and data flow to the user.

Operation:

The sequence to run MCCS is to first define the array processor configuration. That means use F2 to define the PE array, assign local memory to each processor element (PE) and save them.

```
*****
* F1 - Exit Help Menu *   * PgDn - Next Page *
*****
```

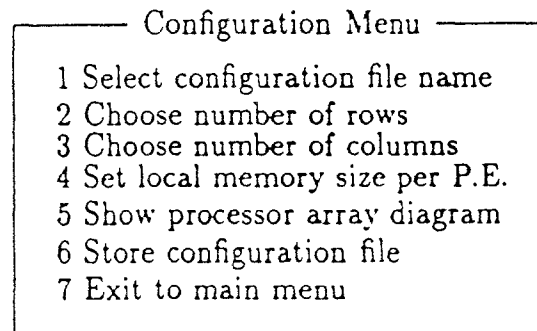
F1-Help menu

F2-Configuration

F3-Execution

F4-Exit to DOS

Figure 2.2: The help menu of MCCS



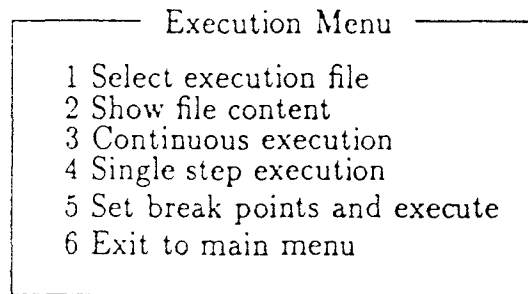
F1-Help menu

F2-Configuration

F3-Execution

F4-Exit to DOS

Figure 2.3: The configuration menu of MCCS



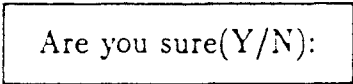
F1-Help menu

F2-Configuration

F3-Execution

F4-Exit to DOS

Figure 2.4: The execution menu of M CCS



A rectangular dialog box containing the text "Are you sure(Y/N):".

F1-Help menu

F2-Configuration

F3-Execution

F4-Exit to DOS

Figure 2.5: The exit menu of M CCS

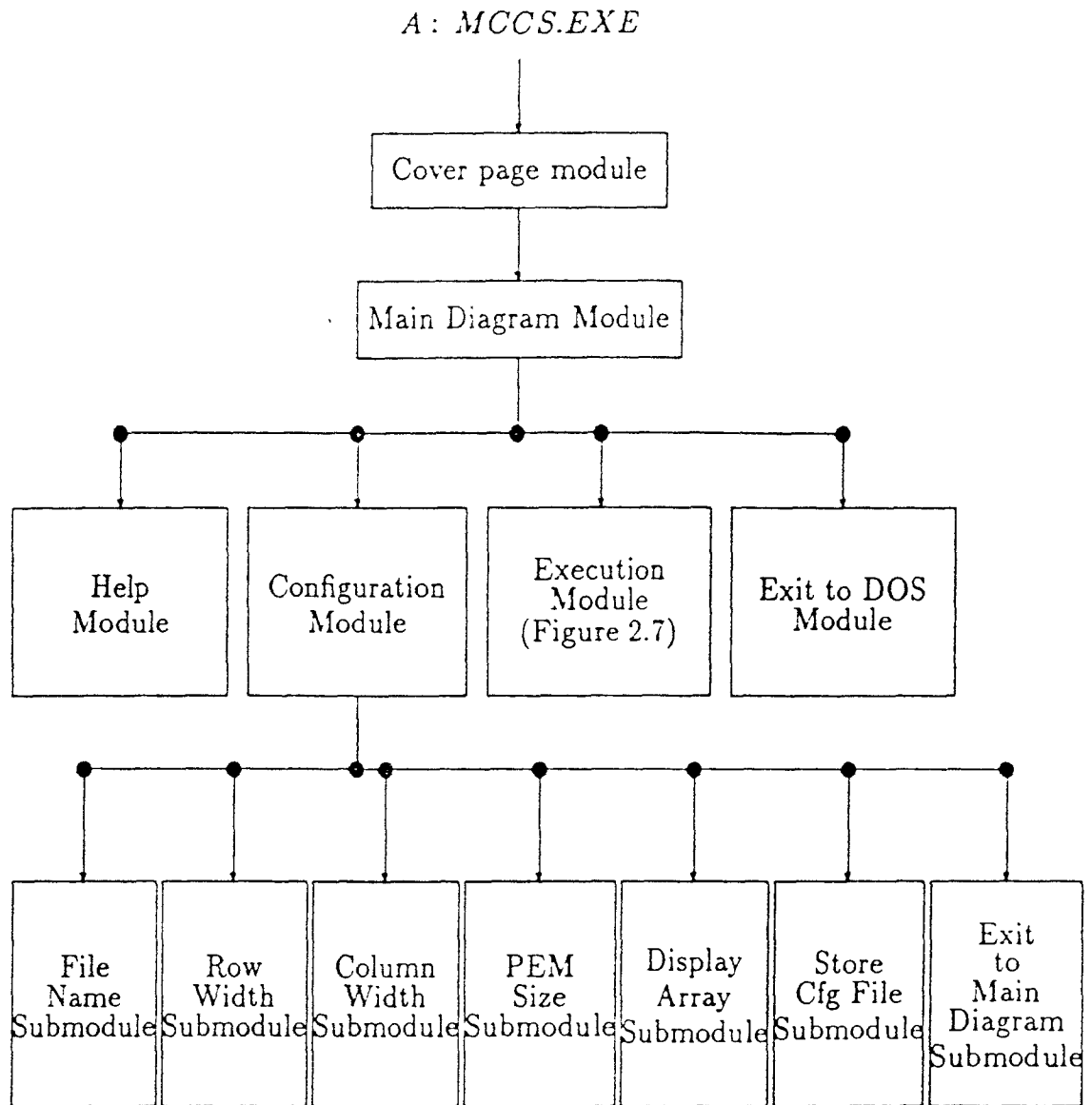


Figure 2.6: Block diagram I of MCCS

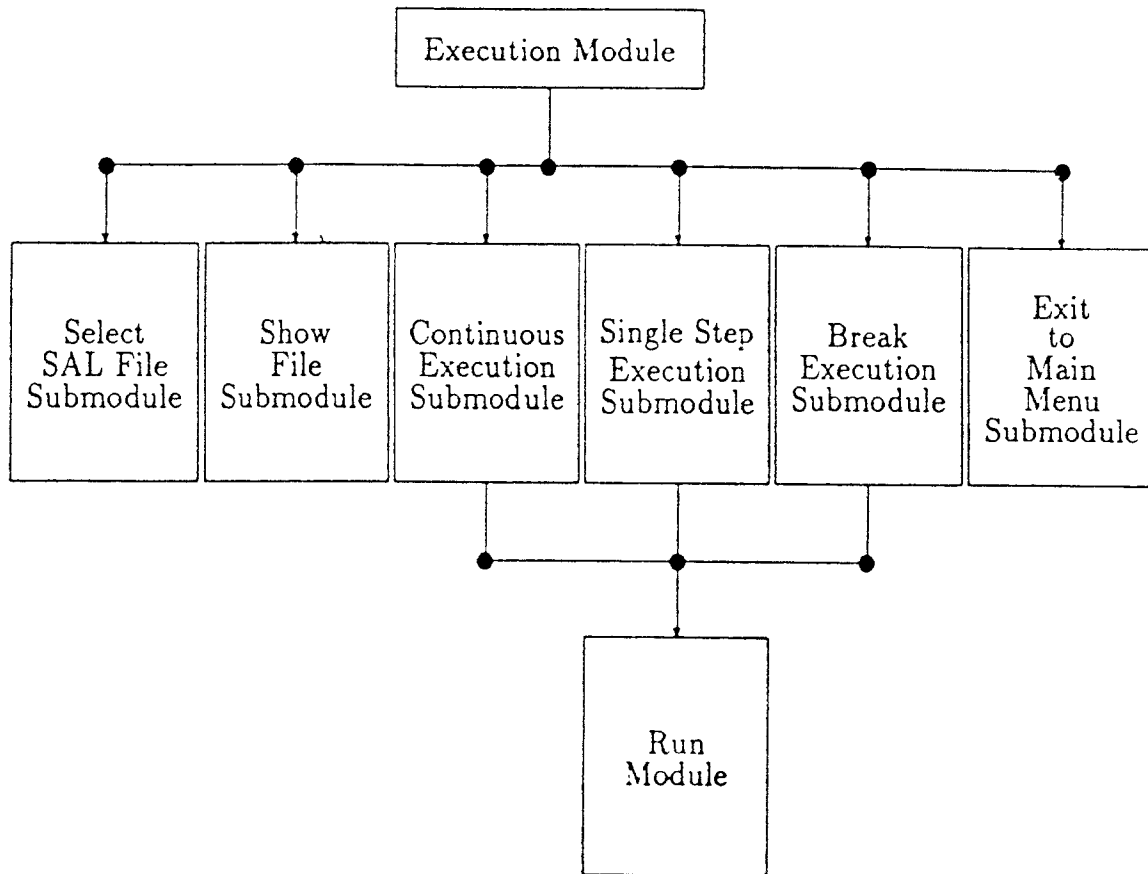


Figure 2.7: Block diagram II of M CCS

2.1.2 Simple Array Processor Language

The programs executed on MCCS are coded using Simple Array Processor Language.

2.1.3 Instruction Set and Control Symbols

The instructions which deal with PE operation are as follows:

- Rotate Down (rtd), Rotate Up (rtu), Rotate Left (rtl) and Rotate Right (rtr). These instructions are useful in routing data from register R (RGR) of one PE to another.
- The Load (lod xxx) and Store (sto xxx) instructions are used to load and store data between register A (RGA) of each PE and location xxx, which ranges from 0 to 127.
- The Add (add), Subtract (sub), Multiply (mul) and Divide (div) instructions are used to carry out basic arithmetic operations. For all these instructions the two operands must be loaded into RGA and register B (RGB) before execution. The result is always stored in RGA.
- Exclusive or (xor), Or (orb) and And (and) instructions are logical operations where the operands are in RGA and RGB and the result

is stored in RGB.

- Move to register B (mbi), move from register B (mbo), move to register R (mri), move from register R (mri), move to register X (mxi) and move from register X (mxo) instructions are used to move between the respective registers and register A.
- Jump zero to entry xx (jpz xx) and jump no zero to entry xx (jnz xx) instructions are used for branching operations where register C is used as a counter. The value xx ranges from 0 to 31.
- Increment (inc) and decrement (dec) instructions are used to manipulate the counter register C.
- The Mask (msk) instruction is used to make a PE active or inactive as required in the program. A 0 denotes a masked PE while a 1 denotes an active PE.

The control unit (CU) instruction set dealing with branch control, data broadcasting and masking is described as follows:

- Set xxx (set xxx) instruction is used to set the content of RGC to the value xxx, which ranges from 0 to 127.

- The jump (`jmp`) instruction is used to branch to location `xx`, which ranges from 0 to 31.
- The broadcast instruction (`bcd xxx`) is used to broadcast the value `xxx` to the register B of all active PEs.
- The broadcast instruction (`cbcd`) sends the value in register RGCA to register B of all active PEs.
- The control load (`clod ij xxx`) and control store (`csto ij xxx`) instructions are used to load and store the contents of RGCA from and to location `xxx` of the `ij`th local memory (PEM) where `i` and `j` are the numbers of the row and column and `xxx` ranges from 0 to 127.
- The (`ent xx`) is not an instruction but is used to indicate the entry point `xx` which ranges from 0 to 31 during `jnz`, `jpz` and `jmp` instructions to transfer control under the conditions designated by the user.

2.1.4 The Control Symbol Set

Every SAL program consists of the following control symbols. They must be appropriately placed, otherwise programs will be aborted. They are:

- # : To denote the start and end of the program.
- \$: To denote start and end of memory allocation algorithms.
- & xxx: Used for loop control where xxx denotes the number of the loop. It is used in \$ and !, the memory allocation and display algorithms, respectively.
- % : To denote start and end of function algorithms.
- !xxx: ! is the display control symbol and xxx is the number of displays. This command is optional. !0 is used to skip this command.

Any SAL program is structured as follows:

- # : start of program
- \$: Start of the memory allocation algorithms (MAA)
- MAA body
- \$: End of the memory allocation algorithms

- % : Start of function computation algorithms (FCA)
- FCA body
- % : End of function computation algorithms
- !xx : Start of display function algorithms (DFA)
- DFA body
- # : denotes the end of program.

2.2 SIMD Array Processors

A synchronous array of parallel processors is called an array processor, which consists of multiple processing elements (PEs) supervised by a single CU. These machines are best suited for single instruction multiple data streams. The typical applications are:

- matrix algebra
- matrix eigenvalue calculations
- linear programming
- general circulation and weather modeling

- beam forming and convolution
- image processing and pattern recognition
- real time scene analysis

A typical mesh is shown in figure 2.8. Each PE (i, j) is connected to four of its neighbors denoted by $(i, j - 1)$, $(i, j + 1)$, $(i - 1, j)$ and $(i + 1, j)$. The PEs are connected in an end around fashion wherein the PEs on the leftmost column are connected to PEs in the previous rows in the rightmost column and the PEs in the topmost row are connected to the PEs in the bottom row. Each PE has its own local memory whose maximum size is 4 Kbytes and can be accessed by sequential or indexed addressing. In the case of indexed addressing the register X is set to the value of the offset. A typical processing element is shown in figure 2.9. Each processing element consists of an arithmetic and logic unit (ALU) to perform arithmetic and logic operations. Each PE also has four registers as follows:

- Register A, called the accumulator for mathematical and logical operations with register B. The results of mathematical operations are integer type for MCCS version 1.00. Another function of RGA is to load data from and store data to local memory (PEM).
- Register R, called the routing register, is used to route data from RGR of one PE to RGR of another PE. Masked PEs can not transmit data

stored in their RGR to other PEs, but can receive data from another PE.

The control unit (CU) consists of the following registers:

- The CU counter register (RGCC) is an unsigned register for counting data from 0 to 65534. It will automatically start from 0 when over 65534.
- The CU mask register (RGCM), which contains M by N bits where M and N are defined by the number of rows and columns in the array processor. A 0 represents that the corresponding PE is masked and a 1 implies that the PE is active.

All the registers except RGCM are 16 bit registers.

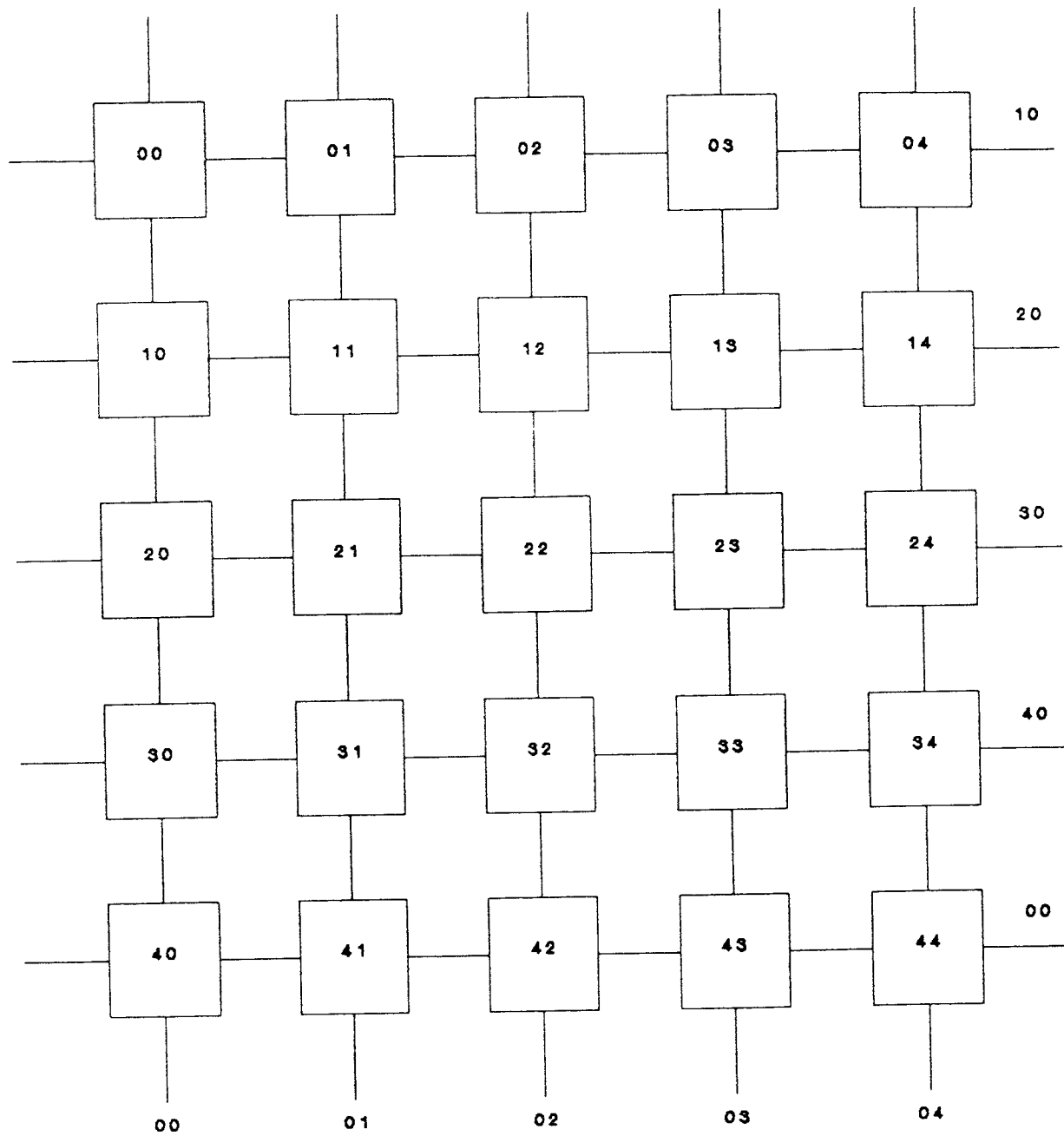


Figure 2.8: A typical SIMD mesh

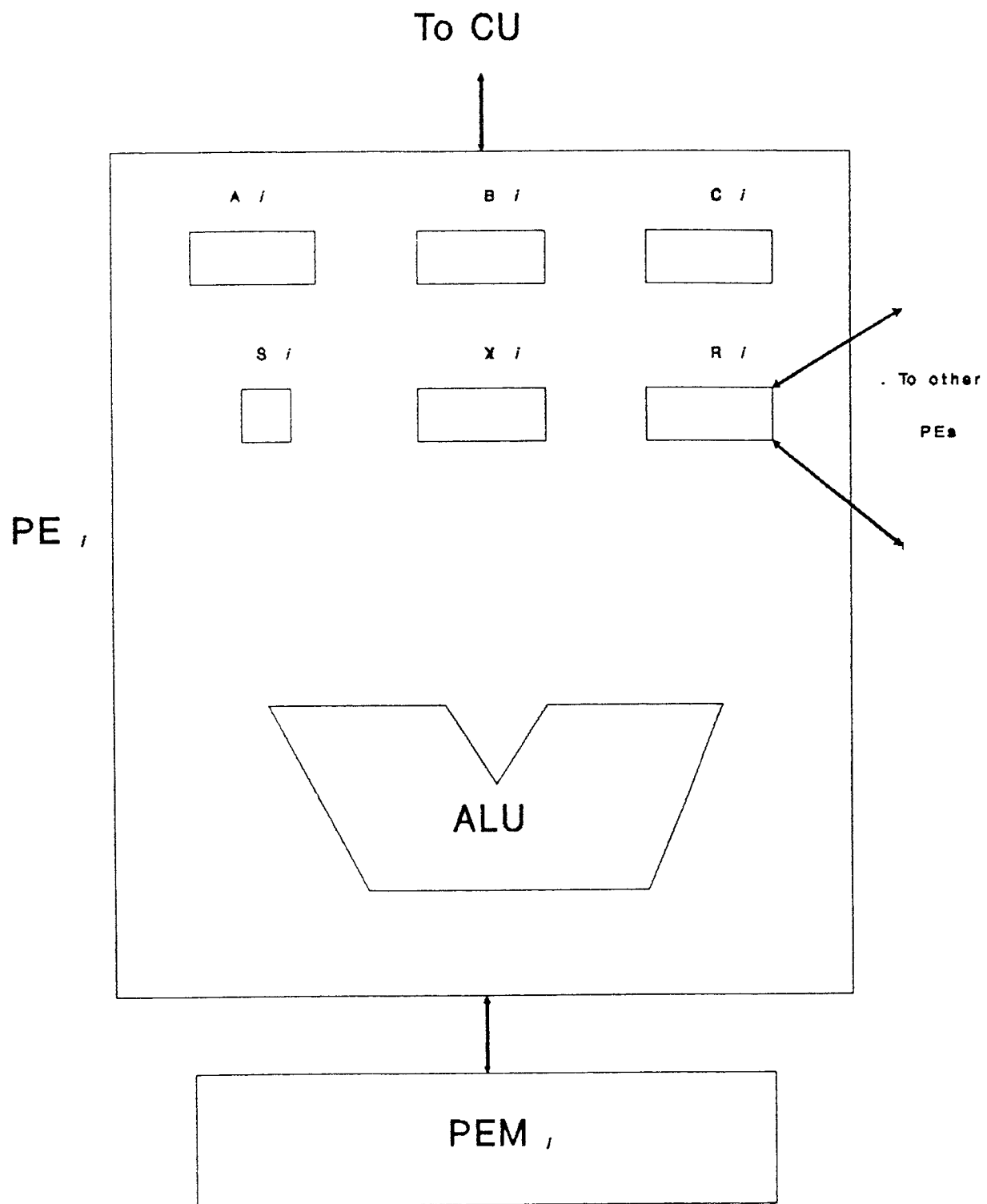


Figure 2.9: A typical processing element

Chapter 3

ALGORITHMS

3.1 Introduction

The first problem that has been parallelized using MCCS is ‘Neighborhood Averaging’. It is a popular image enhancement technique under the category of image smoothing. The principal objective of any image enhancement technique is to process a given image to remove noise so that the result is more suitable than the original image for a specific application. The word ‘specific’ is significant as most of the techniques in this category are problem dependent. These enhancement techniques can be categorized into two categories. They are:

1. Frequency Domain Techniques
2. Spatial Domain Techniques

The frequency domain techniques involve the manipulation of the Fourier Transform of the image whereas the spatial domain techniques involve the

image plane itself, i.e., direct manipulation of the pixels of an image.

3.1.1 Spatial Domain Methods

The term ‘spatial domain’ is defined as the aggregate of the pixels that constitute an image. The methods in this domain are procedures that operate directly on these pixels. The general representation of image processing functions in this domain is:

$$g(x, y) = T[f(x, y)] \quad (3.1)$$

where $g(x, y)$ is the processed image and $f(x, y)$ is the original image. T is the operator on f , defined over some neighborhood of the pixel (x, y) . The basic approach in these methods is to define a neighborhood about (x, y) , for example a square or a rectangular subimage area centered at (x, y) as shown in figure 3.1. Then the center is moved from pixel to pixel over the entire image and the operator T is applied at each pixel. Though the subimage area can be of any geometrical shape the square is chosen because of the ease of implementation.

3.2 Image Smoothing and Neighborhood Averaging

The primary aim of smoothing operations is to “reduce/eliminate spurious effects present in a digital image due to a poor sampling system or transmission channel” [2].

X1	X2	X3
X4	X5	X6
X7	X8	X9

Figure 3.1: A square subimage centered at x_5

3.2.1 Neighborhood Averaging

This problem is a straightforward spatial domain technique for smoothing. Given an $N \times N$ digital image denoted by $f(x, y)$, the aim is to generate a processed image $g(x, y)$ whose grey level intensity at every point (x, y) is obtained by averaging the intensities of the pixels contained in a user defined neighborhood of (x, y) , given by the equation:

$$x_5 = (1/9)[x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9] \quad (3.2)$$

The mask to implement this is shown in figure 3.2.

1	1	1
1	1	1
1	1	1

Figure 3.2: Mask for neighborhood averaging

3.2.2 Algorithm for Neighborhood Averaging

The assumptions made are that the subimage size is 4x4 and the image size is 16x16. Hence 16 iterations are required to process the entire image. The mesh size is chosen to be 6x6, $\sqrt{N+2} \times \sqrt{N+2}$, to eliminate interprocessor communication during the processing of a subimage, while accessing the pixels in the window. Here the window implemented is a square of size 3x3. The mesh is shown in figure 3.3.

Algorithm

1. Load the subimages, i.e., the 4x4 subimages along with the neighboring rows and columns of that subimage starting from location 10 of each PE.

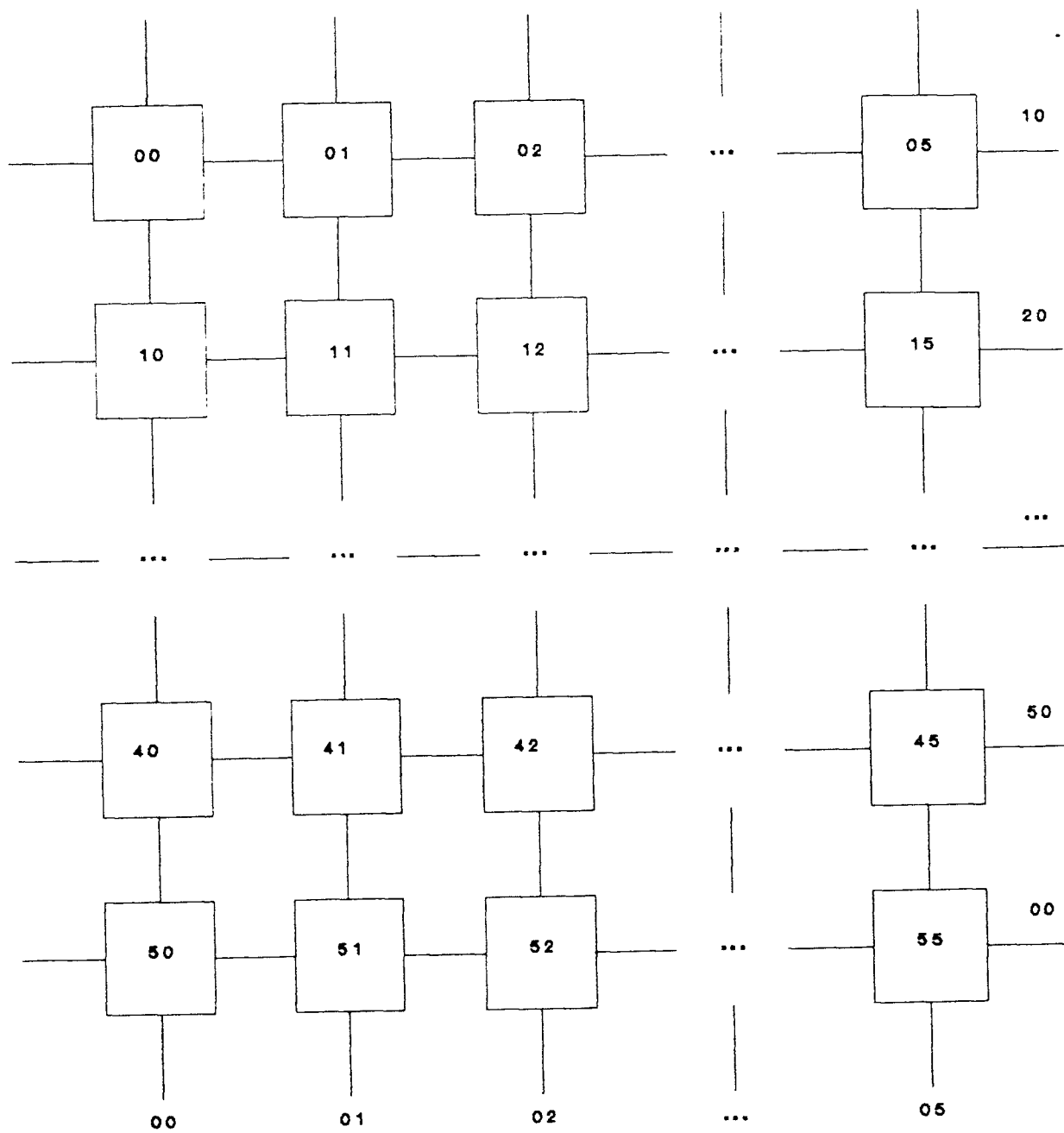


Figure 3.3: A 6x6 mesh as specified in *.cfg file in MCCS

NOTE: Since the pixels on the edge, i.e., row1, column1, row16 and column16 are to be left unchanged, the subimages involving pixels will be of size 5x5 and will be padded with '0's to make them of size 6x6 suitable to be loaded into the mesh. Subimage1 is at location 10 of each PE, subimage2 is at location 11 and so on up to subimage16 at location 25 of each PE.

2. Load register A with subimage1 at location 10 of each PE and store it at location 0 of each PE.
3. If the subimage involves pixels on the edge then using the Rotate Right (rtr), Rotate Left (rtl), Rotate Down (rtd) and Rotate Up (rtu) reposition the subimage so that it is stored in location 0 of PEs 2,1 to 2,5, 3,1 to 3,5, 4,1 to 4,5 and 5,1 to 5,5 of the mesh shown in figure 3.3. This step can be avoided if the padding with 0s is done carefully while the subimages are loaded into the mesh. The 0s must be added to either the leftmost or the rightmost column and either at the top row or the bottom row, depending on the subimage.
4. Using the Rotate Right and Rotate Left instructions, access the neighbor values from PEs $(i, j - 1)$ and $(i, j + 1)$ for all the PEs (i, j) . Store them at locations 1 and 2 respectively. If the subimage contains pixels from the edge of the original image then the PEs containing them will be masked and will be active only when passing the pixel value it holds to the neighboring PEs.

5. Using the Rotate Up and Rotate Down instructions three times each, the six other neighbor values of the pixel (x, y) are accessed from PEs $(i - 1, j)$ and $(i + 1, j)$ for each PE (i, j) and are stored in locations 3 to 8. In this step PEs containing the pixels from the edge are masked.
6. If the subimage being processed contains pixels from the edge then set all the PEs containing those pixels to the active state and all the others to the inactive state (mask). Store location 0 of the active PEs to the corresponding location where the processed subimages are being stored (26 to 41).
7. Mask all the PEs active in step 6; unmask all the others. Add the contents of locations 0 to 8 for all the active PEs.
8. Broadcast 9 to all the active PEs in step 7.
9. Divide the sum in register A by 9 which is in register B for all the active PEs in step 8.
10. Store the result in the corresponding location for all the active PEs in step 9.

56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	
56	79	92	44	83	66	23	108	82	66	43	78	92	126	43	56
56	43	68	72	49	82	104	88	62	96	72	48	54	72	98	56
56	79	42	86	79	64	82	25	72	128	100	70	76	84	92	56
56	102	76	44	68	92	38	76	42	86	82	80	45	62	88	56
56	72	82	98	28	42	56	70	60	72	176	82	98	76	80	56
56	88	90	62	48	162	70	98	78	80	44	69	54	88	102	56
56	78	68	54	32	76	82	90	66	52	74	90	92	102	20	56
56	43	126	92	78	43	66	82	108	23	66	83	44	92	79	56
56	98	72	54	48	72	96	62	88	104	82	49	72	68	43	56
56	92	84	76	70	100	128	72	25	82	64	79	86	42	79	56
56	20	102	92	66	52	74	90	90	82	76	32	54	68	78	56
56	88	62	45	80	82	86	42	76	38	92	68	44	76	102	56
56	72	82	98	28	42	56	70	60	72	176	82	98	76	80	56
56	102	88	54	69	44	80	78	98	70	162	48	62	90	88	56
56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56

Figure 3.4: Original image for neighborhood averaging

11. Activate all PEs.

12. Repeat steps 1 to 11 for the remaining 15 subimages.

The code for this algorithm is written in Simple Array Processor Language (SAL) and is included in Appendix A. The original image and the processed image are shown in figures 3.4 and 3.5. The performance evaluation of the mesh is discussed in chapter 4.

3.3 Edge Detection by Sobel Operators and Laplacian

Gradient and Laplacian operators are part of image segmentation. Image segmentation is the process that subdivides an image into its constituent

56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56
56	62	62	64	62	63	71	70	74	65	63	61	70	72	68	56
56	63	67	68	69	70	71	71	80	80	77	70	77	81	75	56
56	64	68	64	70	73	72	65	75	82	84	69	65	74	73	56
56	69	75	67	66	61	60	57	70	90	97	89	42	45	39	56
56	75	79	66	71	67	78	65	73	80	85	81	41	45	42	56
56	71	76	62	66	66	82	74	74	78	82	86	64	42	37	56
56	73	77	72	71	73	85	82	75	65	64	68	44	46	33	56
56	72	76	69	61	65	74	77	75	73	69	72	76	68	63	56
56	75	81	77	70	77	80	76	71	71	70	69	68	67	63	56
56	70	76	73	70	78	82	76	77	77	72	66	61	65	60	56
56	68	73	75	73	82	80	75	66	69	68	66	61	69	68	56
56	66	73	72	65	62	66	71	68	84	79	80	66	75	72	56
56	73	76	65	60	63	64	71	67	93	89	92	71	79	75	56
56	69	73	63	55	54	59	67	68	89	86	88	69	73	68	56
56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56

Figure 3.5: Processed image after neighborhood averaging

parts or objects. Segmentation is of enormous interest because it helps in the extraction of entities from the image for subsequent processing, such as description and recognition. Algorithms in this class are usually based on one of the following two properties of grey levels: discontinuity and similarity. In the first category, we partition an image based on abrupt changes in grey level. This helps in the detection of isolated points, lines and edges in an image. In the second category the approach is usually based upon thresholding, region splitting, region growing and merging. The methods used to identify these properties are dependent on spatial masks which differ from one application to another.

3.3.1 Edge Detection

Edge detection is the most commonly used method to detect discontinuities in grey level. This is so because isolated points and thin lines are not frequent occurrences in most real life images.

An edge is defined as the boundary between two regions with relatively distinct grey level values. In order to identify edges based on discontinuities in grey level it is assumed that the regions in question are sufficiently homogeneous so that transition between two regions can be determined. If this is not the case then other methods are used to determine edges [2]. Basically, most edge detection algorithms are dependent on the computation of a local derivative operator. The first derivative at any point in an image can be obtained by using the magnitude of the gradient at that point, while the second derivative can be obtained from the Laplacian at that point. Expressed mathematically, the gradient G of an image $f(x, y)$ at location (x, y) is defined by the two dimensional vector given by the equation:

$$G[f(x, y)] = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \quad (3.3)$$

The gradient vector G can be decomposed into two components G_x and G_y . G_x is the component of the gradient vector in the x direction and G_y is the component of the gradient vector in the y direction. For edge detection we need the magnitude (ABS) of this vector, which is given by the equation:

$$ABS(G[f(x, y)]) = \sqrt{G_x^2 + G_y^2} \quad (3.4)$$

It can be approximated by the following equation:

$$ABS(G[f(x, y)]) = ABS(G_x) + ABS(G_y) \quad (3.5)$$

because it is easier to implement when dedicated hardware is being used. If we choose a 3x3 window as the neighborhood size, the masks to compute G_x and G_y are shown in figures 3.6 and 3.7, respectively. The neighbors closest to the center pixel of the mask are given a weight of two because this results in additional smoothing [2]. Referring to figure 3.1 the components G_x and G_y of the gradient vector are given by the following equations:

$$G_x = (x_7 + 2 \times x_8 + x_9) - (x_1 + 2 \times x_2 + x_3) \quad (3.6)$$

$$G_y = (x_3 + 2 \times x_6 + x_9) - (x_1 + 2 \times x_4 + x_7) \quad (3.7)$$

Larger windows can be implemented along the same lines, but 3x3 windows are preferred because of their modest hardware requirements and increased computational speed [2]. The two masks shown in figures 3.6 and 3.7 are commonly referred to as Sobel Operators.

The Laplacian is commonly referred to as the second derivative operator given by the equation:

$$L[f(x, y)] = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} \\ \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \quad (3.8)$$

-1	-2	-1
0	0	0
1	2	1

Figure 3.6: Mask for computing G_z

-1	0	1
-2	0	2
-1	0	1

Figure 3.7: Mask for computing G_y

0	1	0
1	-4	1
0	1	0

Figure 3.8: Mask for computing Laplacian (L)

The digital Laplacian at a point (x, y) with grey level x_5 as shown in figure 3.1 is given by the equation:

$$ABS(L[f(x, y)]) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \quad (3.9)$$

$$= x_2 + x_4 + x_6 + x_8 - 4 \times x_5 \quad (3.10)$$

This can be implemented by the mask shown figure 3.8.

The Laplacian is sensitive to noise and hence is seldom used by itself to detect an edge. It is merely used to determine whether the pixel is on the light or dark side of the edge.

3.3.2 Algorithm

1. Load the 4x4 subimage and the neighboring rows and columns of that subimage, starting from location 10 of each PE.

Note: Since pixels on the edge i.e. row1, row16, column1 and column16 are to be left unchanged, the subimages involving these rows and columns will be 5x5 and so are padded with 0s to make the subimage of size 6x6 suitable to be loaded into the mesh.

2. Load subimage1 at location 10 and store it at location 0.
3. If the subimage involves pixels from row1, column1, row16 and column16 then using the Rotate Right (rtr), Rotate Left (rtl), Rotate Down (rtd) and rotate Up (rtu) instructions reposition the subimage such that the area of interest is stored in location 0 of PEs 2,1 to 2,5, 3,1 to 3,5, 4,1 to 4,5, and 5,1 to 5,5. This step can be avoided if the padding with 0s in step 1 is done carefully depending upon the subimage as explained in the previous algorithm.
4. Using the Rotate Right instruction first and the Rotate Left instruction next access the neighbors on the right and left of each pixel (x,y) in the subimage and store them at locations 1 and 2 of each PE, respectively. If the subimages involve the pixels on the edge of the image then the PEs containing those pixels are masked.

Note: The order in which the neighbors are accessed is important in this algorithm because, for the calculation of gradient, some pixel intensities have to be added and some pixel intensities subtracted. If we follow the same order for all subimages then we can write the code in the form of a loop provided the instruction set is powerful enough to permit it.

5. Using the Rotate Down instruction three times first and the Rotate Up instruction three times next access neighbors from locations $(i-1, j)$, $(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j)$, $(i+1, j-1)$ and $(i+1, j+1)$ for every location (i, j) and store them in locations 3 to 8 of each PE, respectively. They are accessed in the same order as mentioned above.
6. If the subimage involves pixels on the edge of the image store them at the corresponding location for the processed subimage and mask those PEs.
7. Load the contents of location 3 and broadcast 2 to all active PEs. Multiply the contents of register A by 2 and move the product to register B.
8. Add the contents of locations 4 and 5 to the product in register B and store the result in the corresponding location.

9. Repeat steps 6, 7 and 8 on the contents of locations 6, 7 and 8 of active PEs in step 8 and move the result to register B.
10. Load the partial result earlier stored and subtract it from the contents of register B for the same set of active PEs in step 9 and store the result at the corresponding location. The result is the gradient image in the x direction.
11. Load the contents of location 1 for the same set of active PEs in step 10. Broadcast 2 to all active PEs. Multiply the contents of register A by 2 and move the result to register B.
12. Add the contents of locations 4 and 7 to the product and store the partial result at the corresponding location for G_y , the gradient image in the y direction.
13. Repeat steps 11 and 12 for locations 2, 5 and 8 in the same order for the same set of active PEs and move the result to register B.
14. Repeat step 10 and store the result in the corresponding location for G_y .
15. Mask all the PEs for which G_x is positive. Load from the location containing G_x for the remaining PEs. Broadcast -1 to all active PEs

and multiply it with the contents of register A. Store the result in the same location from where it was loaded.

16. Repeat step 15 for G_y with the appropriate mask settings.
17. Unmask all the PEs inactive in step 15. Load G_x and move it to register B. Load G_y and add it to the contents of register B. Store the result in the corresponding location for the gradient image.
18. Load the contents of location 1 for all the active PEs in step 17. Add the contents of locations 2, 3 and 6, moving the partial sum to register B before the contents from each location is loaded.
19. Store the result in the corresponding location for the Laplacian.
20. Load the contents of location 0 for the active PEs in step 19. Broadcast -4 to all active PEs and multiply it with the contents of register A. Store the product in register B.
21. Load the result earlier stored from the location of the Laplacian and subtract it from the contents of register B.

22. Store the result in the corresponding location for the Laplacian.

23. Activate all the PEs. Load the next subimage.

24. Repeat steps 1 to 22 until the whole image is processed.

The code for this algorithm is written in Simple Array Processor Language (SAL) and is included in appendix C. The original and processed images are shown figures 3.9, 3.10, 3.11, 3.12 and 3.13.

72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
72	72	200	200	200	200	200	200	200	200	200	200	200	200	200	200	72	72
72	72	200	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
72	72	200	72	0	72	72	72	72	72	72	72	72	72	72	72	72	72
72	72	200	72	72	0	72	72	72	72	72	72	72	72	72	72	72	72
72	72	200	72	72	72	0	72	72	72	72	72	72	72	72	72	72	72
72	72	200	72	72	72	72	0	72	72	72	72	72	72	72	72	72	72
72	72	200	72	72	72	72	72	0	72	72	72	72	72	72	72	72	72
72	72	200	72	72	72	72	72	72	0	72	72	72	72	72	72	72	72
72	72	200	72	72	72	72	72	72	72	0	72	72	72	72	72	72	72
72	72	200	72	72	72	72	72	72	72	72	0	72	72	72	72	72	72
72	72	200	72	72	72	72	72	72	72	72	72	0	72	72	72	72	72
72	72	200	72	72	72	72	72	72	72	72	72	72	0	72	72	72	72
72	72	200	200	200	200	200	200	200	200	200	200	200	200	200	200	72	72
72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72

Figure 3.9: Original image for edge detection

72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
72	-128	-384	-512	-512	-512	-512	-512	-512	-512	-512	-512	-512	-384	-128	72	72	72
72	-128	-256	-128	0	0	0	0	0	0	0	0	0	-384	-128	72	72	72
72	0	128	384	656	584	512	512	512	512	584	656	456	200	200	72	72	72
72	0	0	0	72	144	72	0	0	72	144	-128	0	0	0	72	72	72
72	0	0	-72	-144	0	144	72	72	144	72	0	-72	0	0	72	72	72
72	0	0	0	-72	-144	0	216	216	0	-144	128	0	0	0	72	72	72
72	0	0	0	0	-72	-72	-72	144	144	-72	0	0	0	0	72	72	72
72	0	0	0	0	72	72	-216	-144	72	72	-128	0	0	0	72	72	72
72	0	0	0	72	144	0	-216	-216	0	144	72	0	0	0	72	72	72
72	0	0	72	144	0	-144	0	-72	-144	0	272	72	0	0	72	72	72
72	0	0	0	-72	-144	-72	0	0	-72	-144	-72	0	0	0	72	72	72
72	0	-128	-456	-656	-584	-512	-512	-512	-512	-512	-512	-384	-128	0	72	72	72
72	128	256	128	0	0	0	0	0	0	0	0	128	256	128	72	72	72
72	128	384	512	512	512	512	512	512	512	512	512	512	384	128	72	72	72
72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72

Figure 3.10: Gradient image in x -direction G_x

72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
72	0	128	128	128	128	128	128	128	128	128	128	128	128	128	0	72	72
72	128	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-301	-256	128	72	72
72	128	-256	256	56	128	128	128	128	128	128	56	436	-301	128	72	72	72
72	128	-256	56	288	-144	0	0	0	0	-72	0	56	-256	128	72	72	72
72	128	-256	128	-144	288	-144	0	0	-144	288	128	128	-256	128	72	72	72
72	128	-256	128	0	-144	288	-144	-144	288	-144	0	128	-256	128	72	72	72
72	128	-256	128	0	0	-144	144	144	-144	0	0	128	-256	128	72	72	72
72	128	-256	128	0	0	-144	144	144	-144	0	0	128	-256	128	72	72	72
72	128	-256	128	0	-144	288	-72	-144	288	-144	128	128	-256	128	72	72	72
72	128	-256	128	-144	288	-144	0	0	-144	288	-144	128	-256	128	72	72	72
72	120	-256	56	288	-144	0	0	0	0	-144	288	-144	128	-256	128	72	72
72	128	-256	256	56	128	128	128	128	128	128	128	256	-256	128	72	72	72
72	128	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	128	72	72	72
72	0	128	128	128	1228	128	128	128	128	128	128	128	128	128	0	72	72
72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72

Figure 3.13: Laplacian image L

Chapter 4

PERFORMANCE EVALUATION OF THE MESH

4.1 Common Measures of Performance Evaluation

SPEEDUP: Speedup (S_n) is the ratio of the time taken by a single processor to that taken by n processors given by the equation:

$$S_n = T_1/T_n \quad (4.1)$$

where S_n is the speedup, T_1 is the time taken by a single processor to execute a program, and T_n is the time taken by n processors to execute the same program.

UTILIZATION: Utilization (U) is the ratio of the total number of busy processor cycles to the total number of available processor cycles. If a program takes K instruction cycles to execute using N processors, then the total number of processor cycles available is the product of K and N . Not all processors will be active during the execution of the entire program. If

B is the number of busy processor cycles for all processing then utilization is given by the equation:

$$U = B/(K \times N) \quad (4.2)$$

EFFICIENCY: Efficiency of the mesh is the utilization of available resources (PEs)[1]. Efficiency is given by the equation:

$$E = S_n/n \quad (4.3)$$

where E is the efficiency, S_n is the speedup with n processors and n is the total number of processors in the mesh.

4.2 Common Parameters

The mesh size for both algorithms is 6×6 since the subimage size is 4×4 . The original image size is 16×16 . To calculate speedup in both the cases, the time taken by a single processor to execute the same algorithm should be determined. If the image size is $N \times N$ then there are N^2 pixels in the image. Since it has been assumed that the pixels on the edge are left unchanged only two instructions are required to process them, i.e. one to load them into their respective PEs and the other to store them at the corresponding location in the local memory of each PE. For any given $N \times N$ image there are $(4N - 4)$ such pixels. Then the remaining pixels are given by the equation:

$$\text{Remaining pixels} = N \times N - (4N - 4) = (N - 2)^2 \quad (4.4)$$

Therefore the time taken for a single processor to execute any algorithm is given by the equation:

$$(4N - 4) \times 2 + (N - 2)^2 \times L \quad (4.5)$$

where L is the number of instructions required to implement the algorithm on a single pixel using one processor on MCCS.

The second step in determining the speedup is to calculate the number of instructions required by n processors to implement the same algorithm. One need not write the whole program to determine this. Irrespective of the image size there will be three distinct types of subimages on which the algorithm is implemented. They are: subimages with one row and one column of pixels from the edge of the original image, subimages with one row or one column of pixels from the edge and subimages with no pixels from the edge. Therefore, given an $N \times N$ image, one can determine the number of subimages in each of the three categories mentioned above. The code is written for each type of subimage and the number of instructions required to process them is determined. For the total number of instructions required to process the entire image, we obtain the sum of the products of the number of instructions required to process each distinct subimage with the number of such subimages over the entire $N \times N$ image. Therefore, if A, B and C are the number of subimages of each type and P, Q and R are the number of instructions required, respectively, to process each type of subimage, the total number of instructions required by n processors (T_n) is given by the equation:

$$T_n = A \times P + B \times Q + C \times R \quad (4.6)$$

The algorithms have been implemented on 16×16 images. In this particular case:

- $A = 4 = \#$ of subimages with one row and one column from the edge of the original image. This will be so irrespective of the image size.
- $B = 8 = \#$ of subimages with one row or one column from the edge of the original image.
- $C = 4 = \#$ of subimages with no pixels from the edge of the original image. The values of B and C change as image size varies.

The actual values of the three parameters discussed above are determined for the two algorithms and general formulae for the three parameters for any given $N \times N$ image are derived.

4.3 Calculations for Neighborhood Averaging Algorithm

4.3.1 SPEEDUP

It has been determined that 28 instructions are required to implement the algorithm on a single pixel on MCCS. Refer to appendix B. Therefore equa-

tion 4.5 is rewritten as:

$$T_1 = (4N - 4) \times 2 + (N - 2)^2 \times 28 \quad (4.7)$$

A total of 5608 instructions are required to implement the neighborhood averaging algorithm on a 16x16 image using a single processor.

It has been determined that 72, 69 and 69 instructions are required to implement the neighborhood averaging algorithm on each distinct type of subimage mentioned above. Therefore the total number of instructions required to implement the algorithm on a 6x6 mesh is equal to:

$$4 \times 72 + 8 \times 69 + 4 \times 69 = 1120 \quad (4.8)$$

Therefore the speedup achieved is:

$$S_{36} = T_1/T_{36} = 5608/1120 = 5.00 \quad (4.9)$$

4.3.2 UTILIZATION

1120 instructions are required to implement the neighborhood averaging algorithm on a 6x6 mesh. Therefore the number of available processor cycles is equal to the product 1120 and 36, i.e., the total number of processors in the mesh. The number of busy processor cycles as determined from appendix A is equal to 19640 cycles. Therefore the utilization in the case of neighborhood averaging algorithm is the ratio of the available processor cycles and busy processor cycles and is equal to:

$$U = 19640/(1120 \times 36) = 0.48 = 48\% \quad (4.10)$$

4.3.3 EFFICIENCY

Efficiency is the ratio of the speedup and the number of processors in the mesh. The speedup in the case of neighborhood averaging algorithm is 5.00 and the number of processors is 36. Therefore efficiency is:

$$E = S_{36}/36 = 5.00/36 = 0.138 = 13.8\% \quad (4.11)$$

4.4 Calculations for the Edge Detection Algorithm

4.4.1 SPEEDUP

It has been determined that 69 instructions are required (refer to appendix D) to determine the gradient of a pixel in the x direction (G_x), the gradient in the y direction (G_y), the gradient (sum of the absolute values of G_x and G_y) and the Laplacian (L) of that pixel. Therefore the time taken by one processor to implement the edge detection algorithm is given by the equation:

$$T_1 = (4N - 4) \times 2 + (N - 2)^2 \times 69 \quad (4.12)$$

Therefore a total of 13644 instructions are required to implement the edge detection algorithm using a single processor on MCCS. It has been determined that 122, 120 and 122 instructions are required to implement the edge detection algorithm on a 6x6 mesh. Therefore the total number of instructions required to implement the edge detection algorithm using 36 processors is equal to:

$$4 \times 122 + 8 \times 120 + 4 \times 122 = 1936 \quad (4.13)$$

Therefore speedup achieved using 36 processors in the case of the edge detection algorithm is equal to:

$$S_{36} = T_1/T_{36} = 13644/1936 = 7.04 \quad (4.14)$$

4.4.2 UTILIZATION

1936 instructions are required to implement the edge detection algorithm using 36 processors. Therefore the number of available processor cycles is the product of 1936 and 36 which is equal to 69696. The number of busy processor cycles as determined from appendix C is equal to 28464. Therefore the utilization in the case of the edge detection algorithm is equal to:

$$U = 28464/69696 = 0.408 = 40.8\% \quad (4.15)$$

4.4.3 EFFICIENCY

Efficiency is the ratio of the speedup and the number of processors in the mesh. Therefore efficiency in the case of the edge detection algorithm is equal to:

$$E = S_{36}/36 = 7.04/36 = 19.5\% \quad (4.16)$$

4.5 Discussion

The common aspect of both the algorithms is the accessing of eight neighboring pixels in a 3x3 window around each individual pixel except those on

the edge of the image.

In the case of the neighborhood averaging algorithm the deviation in the intensity of the pixel as compared to its neighbors in a 3x3 window around it is compensated by its neighbors. If the intensity of the pixel is greater than its neighbors then its value gets reduced by averaging and the opposite takes place if the intensity of the pixel is lower than that of its neighbors. This has a smoothing effect on the image.

The edge detection algorithm belongs to the category of image segmentation. The main objective is to identify the sections of the image with grey level discontinuities. This is accomplished by the gradient operators. Once these discontinuities are determined one has to decide whether the pixel belongs to the brighter or darker portion of the image. This is accomplished by the Laplacian operator. Points where L is positive belong to the darker portion of the image and points where L is negative belong to the brighter portion of the image.

Chapter 5

CONCLUSIONS AND FUTURE DEVELOPMENTS

5.1 Conclusions

Two different classes of spatial domain algorithms have been implemented on MCCS. The first algorithm, which was implemented in parallel, was the neighborhood averaging algorithm, a popular image enhancement technique under the category of image smoothing. The speedup achieved was 5.00, the utilization achieved was 48% and the efficiency of the mesh was 13.8%. The second algorithm implemented was the edge detection algorithm using Sobel operators and Laplacian. This algorithm falls in the category of image segmentation. The speedup achieved was 7.04, the utilization achieved was 40.8% and the efficiency of the mesh was 19.5%.

5.2 Discussion

The reasons for such low values of speedup and efficiency can partially be attributed to the inadequate instruction set of the MCCS. In addition to that, twenty of the thirty six processors in the mesh are required for communication purposes only. Hence they are inactive for most of the pro-

gram. Each access of a neighbor pixel takes five instructions since all load and store operations are to be done through register A in MCCS. Rough calculations show that if the neighbor pixel access can be accomplished in two instructions the speedup almost doubles. One way to access data from neighbor PEs faster is by modification of the load and store instructions of MCCS. Instead of a default register A the user specifies the register. One other important instruction required which not only helps in speeding up these algorithms but also facilitates the implementation of new algorithms in this class is the COMPARE instruction. With this instruction, conditional statements can be implemented and conditional mask settings can be accomplished.

One other shortcoming of MCCS is that it operates only on integer data. If suitable modifications can be made and operations on real data can also be performed on MCCS, the algorithms such as histogram modification and histogram specification can also be implemented on MCCS. The real challenge is the implementation of frequency domain algorithms in parallel. In order to accomplish this major modifications have to be made to the instruction set of MCCS to facilitate manipulations on floating point data. Work is being done at NJIT on another package called Euclid which accepts only floating point data. Hence it may be more suitable for frequency domain algorithms. Since both the packages are complementary, efforts can be made to integrate them into one powerful package.

Appendix A

SAL Program for Neighborhood Averaging

This program is written in SAL to implement a neighborhood averaging algorithm. Each of the sixteen 6x6 subimages are loaded into the local memory of each PE starting from location 10 through 25. The corresponding processed subimages are stored from location 26 through 41. The code and comments are written for one subimage. The number of busy processor cycles are determined as follows:

For each new masking scheme in the program the number of active processors and the number of instructions in that scheme are multiplied to get utilized processor cycles.

... Begin of SAL program.

\$ Begin of memory allocation.

[A] = 6x6 ... subimage A is of size 6x6.

0 0 0 0 0 0

0 56 56 56 56 56

0 56 79 92 44 83

0 56 43 68 72 49

0 56 79 42 86 79

0 56 102 76 44 68

&6

m00 10 = A[1,1]6

m10 10 = A[2,1]6

m20 10 = A[3,1]6

m30 10 = A[4,1]6

m40 10 = A[5,1]6

m50 10 = A[6,1]6

\$ End of memory allocation.

The execution of this part of the algorithm loads subimage A into location 10 of each PE. Similarly all the other 15 subimages are declared and stored from location 11 through 25.

% Begin function body.

msk 111111 111111 111111 111111 111111 111111. This instruction enables all the PEs in the mesh. A 0 in any position indicates that the corresponding PE is masked.

lod 10 This instruction loads subimage A stored in location 10 into register A.

sto 0 Subimage A is stored in location 0.

msk 000000 011110 011110 011110 011110 011110. Change the mask settings.

lod 0 Load location 0 into register A.

mro Move the contents of register A to register R.

rtr Rotate right the contents of register R.

mri move the new contents of register R to Register A.

sto 1 Store the contents of register A in location 1.

The PEs in the rightmost column of the mesh are masked whenever the neighbor pixel is being accessed from the right. For this masking scheme there are five instructions. Therefore the available processor cycles is equal to 180. But only 20 of the 36 processors are active. Therefore the number of busy or utilized cycles is equal to 100.

msk 000000 001111 001111 001111 001111 001111.

lod 0

mro

rtl Rotate left the contents of register R.

mri

sto 2

The PEs in the leftmost column of the mesh are masked whenever the neighbor pixel from the left is being accessed.

msk 000000 001110 001110 001110 001110 000000.

lod 0

mro

rtd Route the contents of register R down.

mri

sto 3

lod 1

mro

rtd

mri

sto 4

lod 2

mro

rtu

mri

sto 5

The execution of this part of the program causes the pixels in top row of the 3x3 window, with each active PE as the reference point to be accessed. The PEs in the bottommost row of the mesh are masked whenever data is being routed down.

msk 000000 001110 001110 001110 001110 001110 000000.

lod 0

mro

rtu Route up the contents of register R.

mri

sto 6

lod 1

mro

rtu

mri

sto 7

lod 2

mro

rtu

mri

sto 8

The execution of this part of the program causes the pixels in the bottom

row of the 3x3 window with every active PE as the center of the window to be accessed. The code written so far is common to both algorithms.

```
msk 000000 011110 010000 010000 010000 010000.
```

```
lod 0
```

```
sto 26
```

If the subimage has any pixels from the edge of the original image then all other PEs except those with such pixels are masked. Then the contents of location 0 of all active PEs are stored in the corresponding location for the processed subimage. These two instructions are not required for those subimages that do not contain any pixels from the edge of the original image.

```
msk 000000 000000 001110 001110 001110 000000
```

```
lod 1
```

```
mbo Move the contents of register A to register B.
```

```
lod 2
```

```
add Add the contents of register A to the contents of register B.
```

```
mbo move the partial sum to register B.
```

```
lod 3
```

```
add
```

```
mbo
```

```
lod 4
```

```
add
```

```
mbo
```

```
lod 5
```

```
add
```

```
mbo
```

lod 6

add

mbo

lod 7

add

mbo

lod 8

add

mbo

lod 0

add

bcd 9 Broadcast 9 to all active PEs.

div Divide the contents of register A by 9.

sto 26 Store the result (average) in location 26.

This segment is repeated for all the 16 subimages with the corresponding mask settings.

Appendix B

Neighborhood Averaging on One Pixel Using a Single Processor

... Begin of program.

\$ begin memory allocation.

[A] = 9x1

56

56

56

56

72

86

92

78

89

9

m00 0 = A[1,1]


```

m00 1 = A[2,1]
m00 2 = A[3,1]
m00 3 = A[4,1]
m00 4 = A[5,1]
m00 5 = A[6,1]
m00 6 = A[7,1]
m00 7 = A[8,1]
m00 8 = A[9,1]

```

\$ end of memory allocation.

Execution of this part of the program loads a pixel and its neighbors in a 3x3 window around it starting from location 0 to location 8 of the local memory of the PE.

%.... begin of function body.

lod 0 Load the contents of location 0 into register A.

mbo Move the contents of register A to register B.

lod 1 Load the contents of location 1 into register A.

add Add the contents of register A to that of register B.

mbo Move the partial sum to register B.

lod 2

add

mbo

lod 3

add

mbo

lod 4

add

mbo

lod 5

add

mbo

lod 6

add

mbo

lod 7

add

mbo

lod 8

add

bcd 9 Broadcast 9 to the processor.

div Divide the sum in register A by 9.

sto 10 Store the result in location 10.

Appendix C

SAL Program for Edge Detection

... Begin of program.

\$ Begin memory allocation.

Declare and load subimages as specified in the previous two sections.

\$ end of memory allocation.

% ... Begin of function body.

Repeat the same segment as in appendix A to access the eight neighboring pixels for every pixel except for those on the edge of the original image.

msk xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx. Change the mask settings as required. If the subimage contains pixels from the edge of the original subimage then at this stage set only those PEs containing those pixels active and mask all the others. Then store pixels in the corresponding locations for the results from the active PEs.

msk xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx. Modify the mask settings as required.

lod 3 Load from location 3 into register A.

bcd 2 Broadcast 2 to all the active PEs.

mul Multiply by 2 the contents of register A of all active PEs.

mbo Move the result to register B.

lod 4 Load the contents of location 4 into register A.

add Add the contents of register A to that of register B.

mbo Move the result to register B.

lod 5 load the contents of location 5 into register A.

add

sto 15 Store the result in the location for G_x .

The execution of this part of the program results in the calculation of the top row of the 3x3 mask for G_x .

lod 6

bcd 2

mul

mbo

lod 7

add

mbo

lod 8

add

This part of the program computes the bottom row of the 3x3 mask for G_x .

mbo

lod 15 Load from location 15 the partial result stored earlier.

sub Subtract the contents of register A from the contents of register B.

sto 15 store the result in location 15.

sto 9 store the result in location 9.

The execution of this part of the program results in the computation of G_x .

In similar fashion G_y and L are computed with appropriate mask settings such the 3x3 masks for G_y and L are computed for each pixel in the subimage. G_y is stored in locations 10 and 16 while the Laplacian is stored in location 18.

msk xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx. Modify the mask settings such that only those PEs in which the contents of location 9 are negative are active.

bcd -1 Broadcast -1 to all active PEs.

mul

sto 9

This part of the program calculates the absolute value of G_x . By multiplying with -1 we are converting all negative values into positive values.

Repeat the same instructions to modify G_y with appropriate mask settings and store the result in location 10.

msk xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx Modify the mask settings as required.

lod 9 load the contents of location 9.

mbo

lod 10

add

sto 17 store the result in the location for gradient G .

Repeat this entire program for all 16 subimages to process the entire image.

Appendix D

Edge Detection on One Pixel Using A Single Processor

This program segment is the same as appendix C as far as the computation of G_x , G_y and L is concerned. The only difference is that we do not have to access the neighboring pixels since the entire image is in the processor's local memory. From appendix C we can determine that this takes 69 instructions.

Bibliography

- [1] Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing*, Mc-Graw Hill, 1989.
- [2] Azriel Rosen Field and Avinash C. Kak, *Digital Picture Processing*, Academic Press, London, 1976.
- [3] David C. Chen and John D. Carpinelli, *MCCS User's Guide*, NJIT, February, 1989.
- [4] Rafael C. Gonzalez and Paul Wintz, *Digital Image Processing*, Addison-Wesley Press, 1987.
- [5] Joseph Kittler and Michael J. B. Duff, *Image Processing System Architectures*, Research Studies Press, 1985.