



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

**IMAGE PROCESSING ON A PARALLEL COMPUTER
ARCHITECTURE**

Duraisamy Sundararajan

A Thesis

In

The Department

of

Electrical and Computer Engineering

~~Presented in Partial Fulfillment of the Requirements~~
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

December, 1987

© Duraisamy Sundararajan, 1987

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-41649-1

ABSTRACT

Image Processing on a Parallel Computer Architecture

Duraisamy Sundararajan, Ph. D.
Concordia University, 1987

This investigation is concerned with the design of image processing algorithms and an architecture for their implementation. Traditionally, the requirements of an efficient image processing have been twofold : faster execution time and smaller memory space. Now, with the compelling argument in favour of parallel processing, the algorithms must be designed to have the additional features of reduced data dependency and decomposability for efficient parallel implementation. Keeping these objectives in mind, two new algorithms, one linear and the other nonlinear, are designed. An MIMD type of architecture addressing the problems of parallel processing and designed to match a wide range of image processing algorithms, is proposed.

A design of an algorithm for two-dimensional convolution operation and its implementation on the proposed architecture is presented. In the direct implementation of a convolution function, the data values are first shifted and then multiplied by the appropriate coefficients to obtain the convolution sum. In the proposed implementation, a single data value is multiplied by the coefficients and the resulting products are shifted and added to produce the output. This scheme results in a shorter execution of the software implementation, since it takes less time to access the product values of a single pixel with a set of coefficients than to access the product values of a set of pixels with the corresponding coefficients. The proposed scheme has a feature of reduced number of addition operations by making use of the symmetry properties of windows as encountered in many image processing tasks. It is shown that for convolving an image with a window in different orientations, the algorithm requires a smaller number of multiplications than in a direct implementation.

As an example of a nonlinear image processing operation, design and implementation of a median filtering algorithm is presented. The operations involved in a median filtering algorithm are dependent on the nature of the image data. In order to achieve a high parallel implementation efficiency, the data dependency of the algorithm is reduced in the design of the algorithm itself. In this algorithm, the elements of a window are put into two sets and these sets are updated for each window. The basis of the proposed algorithm is that if the elements of windows, in which the members of individual rows are prearranged in an ascending order, are stored row by row in a one-dimensional array, then the results obtained in partitioning of all the rows of the past window, except one row, can be used to partition and find the median of the current window. The results of the windows immediately above and to the left are used to find the median of the current window. This procedure results in a faster execution time. In addition, it is shown that the execution time is independent of the number of bits (gray levels) used to represent the data and the algorithm is relatively insensitive to noise levels in the image.

The proposed parallel architecture for image processing is a bus-oriented system consisting of a master processor, a number of slave processors, and dual-port interprocessor communication buffer memory modules. These memory modules are included to provide fast and simultaneous communication between adjacent processors.

There are four factors that affect the implementation efficiency of an algorithm on a parallel architecture : (i) contention for a common resource, (ii) unequal work-load distribution among the processors, (iii) processors waiting for partial results and data, and (iv) insufficient concurrency in the algorithm. Parallel processing schemes attempting to resolve these problems on the proposed architecture are developed.

The major communication links between the processors of the architecture are the common memory modules which are contented only by two adjacent processors. This feature of the architecture along with the asynchronous nature of the common memory

modules reduces the contention problem. The contention problem is even further reduced by splitting the common memory space into two regions during the algorithm execution, one for writing and the other for reading the neighborhood data and partial results. The reading area of a common memory module to one processor is the writing area for the other processor connected to it. Algorithms are designed in such a way that at a specific time processors access generally different areas of the same common memory module.

As characteristics of an image may drastically vary from region to region, assigning continuous regions to processors in a parallel computer system leads to an uneven work-load distributions among the processors when executing a data-dependent algorithm. To reduce this problem, a new data partitioning scheme for parallel processing is proposed. The underlying principle of the data partitioning scheme is to force each processor, as much as possible, to work in every region of the image.

The number of synchronization points are reduced by synchronizing the processors after processing a set of windows rather than after each window. This reduces the idling time of the processors, since the execution time is likely to even out over a large number of windows. In addition, the data and partial results are transferred much earlier than when they are required.

ACKNOWLEDGEMENTS

I wish to express my heartfelt and profound gratitude to my supervisor, Professor M. O. Ahmad, for his invaluable guidance, assistance, and inspiration throughout the span of this research and for his advice during the preparation of the manuscript.

I would also like to express my deep appreciation to my family members without whose love, encouragement, and support this work would not have been possible.

This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant A1684 and Fonds pour la Formation de Chercheurs et l'Aide la Recherche under Grant EQ 2007.

TO MY PARENTS

TABLE OF CONTENTS

LIST OF FIGURES	xi
LIST OF TABLES	xiii
LIST OF ABBREVIATIONS AND SYMBOLS	xiv
CHAPTER I: INTRODUCTION	1
1.1 General	1
1.2 Image Processing Algorithms	2
1.3 Image Processing Architectures	2
1.4 Scope and Organization of the Thesis	4
CHAPTER II: THE PROPOSED ARCHITECTURE	7
2.1 Introduction	7
2.2 Image Input and Output	9
2.3 The Processing Elements	9
2.4 Interprocessor Communication	9
2.5 Synchronization	10
2.6 The Master Processor	13
2.7 The Prototype System	13
2.8 Conclusion	14
CHAPTER III: DESIGN AND IMPLEMENTATION OF A TWO-DIMENSIONAL CONVOLUTION ALGORITHM	15
3.1 Introduction	15
3.2 The Proposed Algorithm	16
3.3 The Structure of the Algorithm	22

3.4	Test Results	24	
3.5	Discussion and Summary	24	
CHAPTER IV: DESIGN AND IMPLEMENTATION OF A MEDIAN			
FILTERING ALGORITHM			28
4.1	Introduction	28	
4.2	The Proposed Algorithm	30	
4.3	Test Results and Discussion	38	
4.4	Summary	43	
CHAPTER V: IMAGE PROCESSING ON THE PROPOSED			
PARALLEL ARCHITECTURE			45
5.1	Introduction	45	
5.2	Data Partitioning	45	
5.3	Point Operations	49	
5.4	Neighborhood Operations	49	
5.5	The Parallel Implementation of the Two-dimensional Convolution Algorithm	50	
5.5.1	The Approach of the Implementation	51	
5.5.2	The Algorithm	53	
5.5.3	Test Results	57	
5.6	The Parallel Implementation of the Median Filtering Algorithm	58	
5.6.1	The Approach of the implementation	61	
5.6.2	The Algorithm	64	
5.6.3	Test Results and Discussion	66	
5.6.4	Conclusion	69	

5.7 Implementation of the Two-dimensional Discrete Fourier Transform	71
5.8 Implementation of the Histogram Algorithm	72
5.9 Conclusion	75
CHAPTER VI: CONCLUSION	77
6.1 Summary	77
6.2 Scope for Future Investigation	79
REFERENCES	81
APPENDIX A	85
APPENDIX B	88

LIST OF FIGURES

Fig. 2.1. The proposed parallel computer architecture.	8
Fig. 2.2. Data communication scheme between the PEs through common memories.	11
Fig. 2.3. Synchronization operation.	12
Fig. 3.1. Two-dimensional array representing pixels of an $M \times N$ image	17
Fig. 3.2. Matrix representation of an $R \times S$ window.	17
Fig. 3.3. Terms of the convolution $g(i,j)$, arranged in a matrix form.	18
Fig. 3.4. Multiplication matrix formed by the product values of the pixel $f(i,j)$ with the window coefficients.	19
Fig. 3.5. (a) A 3×3 window. (b) A section of an image.	21
Fig. 3.6. A structure of the direct implementation of two-dimensional convolution.	23
Fig. 3.7. The structure of the proposed implementation of two-dimensional convolution.	23
Fig. 4.1. Pixels in an $R \times S$ window in the neighborhood of pixel $f(i,j)$ of an image.	31
Fig. 4.2. (a) Pixels in a section of the image of Example 4.1 with the window corresponding to the pixel $f(3,3)$. (b) The window of (a) after partitioning.	35
Fig. 4.3. (a) Elements in the window corresponding to the pixel $f(4,3)$ of Example 4.1. (b) Partitioning of the window in (a) after Step 3. (c) Final partitioning of the window in (b) after elements 24, 25, and 26 are transferred from Subset 2 to Subset 1.	36

Fig. 5.1. Data partitioning and their assignment to PEs, transfer of partial results between PEs, and the processing sequence for $N = 16$ and $n = 4$.	47
Fig. 5.2. (a) A 3×3 window. (b) A section of an image. (c) The array $X(i,j)$.	52
Fig. 5.3. (a) A 256×256 input image. (b) An edge output of the image in (a) by implementing the proposed convolution algorithm on the 2-PE prototype architecture.	60
Fig. 5.4. Illustration of hard synchronization. Solid line shows the busy period of a PE and dotted line shows the idling period of a PE.	62
Fig. 5.5. A scheme of soft synchronization and illustration of various operations performed by a PE in a given column cycle for $N = 16$ and $n = 4$.	62
Fig. 5.6. (a) A 256×256 input image. (b) The image after the application of 5×5 median filtering by implementing the proposed algorithm on the 2-PE prototype architecture.	68
Fig. 5.7. Data transfer to transpose an image. (a) The image after column-wise transform. (b) Pixel assignment to PEs after the first cycle of data transfer. (c) Pixel assignment to PEs after the second cycle of data transfer.	73

LIST OF TABLES

Table 3.1	Execution time in seconds of the convolution algorithm for a 256×256 image with 256 gray levels	25
Table 3.2	Execution time in seconds of the convolution algorithm for a 256×256 image with floating-point values	25
Table 4.1	Average execution time in seconds of the proposed (P) and the histogram (H) algorithms and the run-time gain for six different 256×256 images with 128, 256, and 512 gray levels	39
Table 4.2	Actual and worst-case number of comparisons per window in the proposed and the histogram algorithms	42
Table 4.3	The execution time in seconds of the proposed and the histogram algorithms for different noise levels in an image with 256 gray levels	44
Table 5.1	Execution time in seconds and the efficiency of the convolution algorithm for a 256×256 image with 256 gray levels	69
Table 5.2	Average execution time in seconds and the efficiency of the median filtering algorithm for six different 256×256 images	67
Table 5.3	Time (hypothetical) to execute a column of an image	70
Table 5.4	Time taken by PEs to process their assigned segments using the proposed(P) and consecutive column(C) partitioning schemes	70

LIST OF ABBREVIATIONS AND SYMBOLS

CM	Common memory
<i>conout1</i>	One-dimensional array consisting of the convolution output of an image with a window
<i>F</i>	Image matrix
<i>f(i,j)</i>	Pixel value of the image <i>F</i> at position (i,j)
FLIN	Flag used to read from the left CM
FLOUT	Flag used to write into the left CM
FRIN	Flag used to read from the right CM
FROUT	Flag used to write into the right CM
<i>J1</i>	One-dimensional array consisting of the consecutive columns of the image <i>F</i>
<i>G</i>	Number of gray levels in an image
<i>g(i,j)</i>	Convolution of a window with pixel values at position (i,j)
<i>M</i>	Number of rows in an image
MIMD	Multiple-instruction stream multiple-data stream
<i>N</i>	Number of columns in an image
<i>n</i>	Number of processors in the system
<i>pb</i>	One-dimensional multiplication look-up table consisting of the products of gray levels in the image and the window coefficients
PE	Processing element
PG	Percentage run-time gain of the proposed median filtering algorithm over the histogram algorithm

- prest** One-dimensional array with S elements each being a sum of certain number of elements in **pter**.
- pter** One-dimensional array with RS elements each being a sum of certain number of product values of window coefficients in a column and corresponding pixel values
- R** Number of rows in a window
- S** Number of columns in a window
- SIMD** Single-instruction stream multiple-data stream
- T_{hist} Execution time of the histogram median filtering algorithm on a uniprocessor
- T_{prop} Execution time of the proposed median filtering algorithm on a uniprocessor
- $V(i,j,k)$ The k th row of the window at position (i,j) with the elements arranged in ascending order
- W** Window matrix
- $w(i,j)$ Window coefficient at position (i,j)
- w1** One-dimensional array consisting of consecutive rows of **W**
- $X(i,j)$ Matrix consisting of the terms of the convolution function $g(i,j)$
- $Y(i,j)$ Multiplication matrix consisting of the product values of the pixel $f(i,j)$ and the window coefficients

CHAPTER I

INTRODUCTION

1.1. GENERAL

In recent years, computer processing of images has been accepted as a necessary and vital tool in a variety of applications. Digital image processing systems are capable of acquiring and processing images with wider dynamic range than the human eye or photographic film. A digital image is a two-dimensional array of numbers. Each point in the digital image represents the brightness or intensity of the image at that point and it is related to the brightness or intensity of an area around the point. Each element in the digital image is referred to as a picture element, commonly abbreviated as pixel or pel. Image processing consists of altering an image in order to provide it with desirable features, and to classify it into different objects. Image processing is used in such applications as medical diagnosis, industrial inspection, weather analysis, and mineral, undersea and space explorations. One of the reasons for using parallel computers in image processing is the requirement for fast processing of large volume of data. For example, for real-time processing of television pictures with data rate of 25 frames per second, processing of approximately 80 million pixels per second is required for a 3×3 neighborhood operation.

Due to the advent of faster and cheaper VLSI processing elements, large-capacity storage devices, and improved input scanning and output display technology, it has become possible to acquire, process, and display images with increasingly large volume of digital data. Since most of the image processing algorithms are easily decomposable into identical processes depending only on pixel values in a small neighborhood, the image processing problem is ideally suited for parallel processing. However, there are problems to be resolved in efficiently configuring a parallel architecture for specific applications. Some of the problems are partitioning, scheduling, synchronization, and

interprocessor communication. Therefore, an image processing job must be partitioned into tasks; each task must be efficiently scheduled; synchronization of control and data flow must be performed during execution; and interprocessor communication must be established with as little contention as possible. In parallel processing of images, the most crucial problem is the selection and efficient use of an interprocessor communication scheme.

1.2. IMAGE PROCESSING ALGORITHMS

There are two types of operations in image processing — low-level and high-level. The low-level operations involve enhancement of an image and extraction of features from it. The high-level operations then evaluate the extracted features. For low-level operations, such as edge detection, it is difficult to achieve a high throughput using a uniprocessor system, since an output pixel is a function of a set of neighboring pixels in the input image and the neighborhood size may typically vary from 3×3 to 15×15 . Operations required for high-level processing, on the other hand, are symbolic manipulations of the features extracted from the image through low-level processing. The input data rate at this stage is reduced by a factor of 100 to 10,000 from the raw pixel rate, thereby relaxing the requirement of high throughput [1]. Whereas the low-level operations are clearly defined and is the same for all segments of the image, the type of operations performed at high level depends on the evaluated features and may be different for different segments of the image. The two levels of operations require two different types of processing systems. Single-instruction multiple-data stream (SIMD) machines meet the requirements of low-level processing whereas multiple-instruction multiple-data stream (MIMD) machines are better suited for high-level processing [2].

1.3. IMAGE PROCESSING ARCHITECTURES

Broadly speaking, the existing parallel image processing systems can be classified into three categories: (i) array processors, (ii) pipeline processors, and (iii) bus-oriented

multiprocessors. Unger [3] has suggested the use of a two-dimensional array of processing elements (PEs) in which each PE does the required processing for its assigned pixels. The neighboring pixel values are fetched using an interconnection network. All the processors work simultaneously providing parallelism. Despite the decreasing cost of hardware components, implementation of two-dimensional systems is still costly and the performance is limited by the interprocessor communication delays [4]. Further, the communication network becomes very complex if large number of PEs are used. In order to reduce the complexity, most of the existing systems are built with boolean processors and the interconnection is usually limited to a neighborhood of 3×3 [5],[6]. It is possible to process gray level images on such machines by operating serially on the bits of the data. This, however, limits the processing power of the machine due to the requirement of complex software [7]. One-dimensional array of PEs are more efficient, simpler, and economical for parallel processing of images [4]. Pipeline and bus-oriented multiprocessor systems belong to this type of machines.

Most of the architectures developed for image processing are optimized for low-level processing. However, architectures that are capable of executing both low- and high-level processing have also been proposed [8]-[13]. There are three types of such machines: (i) systems having individual sections for processing the two levels, (ii) systems that can be reconfigured into SIMD or MIMD mode of processing, and (iii) systems having only MIMD type of machines. One example of the first type is GOP [8], where two independent sections of pipeline and MIMD machines have been combined to perform low-level and high-level operations, respectively. Another example of the first type is SY.MP.A.T.I [9] built with two independent levels. The first level consists of multiple SIMD-type sections. The second level is of an MIMD type. An example of the second type is a reconfigurable machine, PASM (partitionable-SIMD/MIMD) [10], where the PEs can work in SIMD and/or MIMD modes. PASM is a special-purpose, dynamically reconfigurable, large-scale multimicroprocessor system. Bus-oriented multiprocessor sys-

tems such as CYBA-M [11], ZMOB [12], and the homogeneous multiprocessor [13] are examples of the third type of architecture. CYBA-M consists of sixteen general-purpose microprocessors each connected to a common memory. The central image-memory access is arranged as a five-stage synchronous pipeline. In ZMOB, a shift register bus is used as the communication channel between the PEs. The processors are placed along the bus with an interface that intercepts the data as it is moved along the bus at a high speed. All the PEs are general-purpose microprocessors. In the homogeneous multiprocessor each processing element can access the memory elements of its two immediate neighbors via dynamically created extended buses. One of the parallel implementation problems that has not received much attention in the literature is the load balancing among the PEs in a system. This problem is crucial to the designs of architectures and the algorithms for parallel implementation.

1.4. SCOPE AND ORGANIZATION OF THE THESIS

In this thesis, an image processing machine with an MIMD type of architecture is proposed. It is essentially a bus-oriented architecture in which a number of microcomputers are connected to a common bus. As a single bus is inadequate for the interprocessor communication requirement of neighborhood operations, an asynchronous dual-port memory is used between processors for fast interprocessor communication. The proposed machine has some distinct features both from the architectural point of view as well as in terms of its image processing capability. Specifically, (i) a dual-port memory is used as an interprocessor communication channel between each neighboring pair of PEs, (ii) an efficient scheme of image partitioning and assignment to various PEs is employed, and (iii) an improved synchronization scheme is utilized. As discussed in the following chapters, these features provide some advantages in parallel processing of images. A computer architecture consists of four components: (i) hardware, (ii) algorithms, (iii) operating system, and (iv) the programming language. In this thesis, the problems with respect to the first two aspects as they relate to image processing, are addressed.

Two new algorithms, one for two-dimensional convolution [14] and the other for median filtering [15] are described. These algorithms are presented as representatives of linear and nonlinear neighborhood operations in image processing. Both of these algorithms, while providing faster execution time in uniprocessor implementations, have been designed to achieve a high parallel implementation efficiency as well.

There are several types of operations used in image processing. The proposed architecture, being an MIMD type, can be used for any kind of computation, i.e., it can support an arbitrarily structured parallelism. However, the predominant type of operations in image processing is neighborhood operations and hence the architecture must be tailored to implement these operations efficiently. The neighborhood operations can be classified as: (i) algorithms with fixed number of operations per window, such as convolution and (ii) algorithms with variable number of operations per window, such as median filtering. The execution time of the first type is almost constant while for the second, it is data dependent and varies from window to window. For the second type of algorithm, PEs will finish processing of their assigned segments in different times. The processing of an image is complete only when the PE finishing last has completed the processing of its segment, resulting in a low parallel implementation efficiency. In the design and implementation of the parallel algorithms presented in this thesis, an attempt is made to minimize this problem. To achieve a high efficiency for all types of neighborhood operations, a new data partitioning scheme is proposed. A synchronous execution is chosen for the parallel implementation of the algorithms. The synchronization points have been reduced by synchronizing the PEs after processing a set of windows rather than after each window. This will reduce the idling time of PEs, since the execution time is likely to even out over a large number of windows. In addition, in order to reduce the idling time even further, the data and partial results are transferred much earlier than when they are required.

Chapter II describes the proposed architecture. In Chapter III and IV, two new algorithms and their serial implementations are presented. In Chapter V, the approach and scheme of parallel implementation of image processing algorithms on the proposed architecture are discussed. The design and parallel implementation of convolution and median filtering algorithms, as examples of data-independent and data-dependent operations in image processing, are presented in detail. The parallel implementations of the discrete Fourier transform and histogram algorithms are also discussed. Chapter VI concludes the thesis by summarizing the salient features of the proposed architecture, the algorithms, and their implementations.

CHAPTER II

THE PROPOSED ARCHITECTURE

2.1. INTRODUCTION

A general-purpose image processing architecture suitable for both low-level and high-level processing can be built by choosing an MIMD type of architecture. An MIMD architecture consists of a number of PEs, each having its own control unit, memories, and an interconnection network between processors and/or memories. As a single shared bus is the simplest, least expensive, and most commonly used interconnection network, a bus-oriented multiprocessor architecture is chosen. In the design of this type of architecture there are two main problems to be resolved: (i) communication problem — degradation of the PE utilization time caused by the bus congestion, and (ii) I/O problem — inputting and outputting of a large amount of data. In the design presented in this chapter, an attempt is made to resolve both these problems.

The proposed parallel computer architecture is shown in Fig. 2.1. It consists of a master processor and a number of slave PEs (say n) connected through a bus. Each PE is configured around a 16-bit microcomputer board. In addition to the common bus, each pair of adjacent PEs is also connected through a small communication memory (CM). For most low-level operations, the PEs use this cascade connection. For direct communication between any two PEs, the common bus is used. Employment of a single bus in the system with a reasonably large number of PEs is adequate, since this bus is used as communication channel only during high-level processing when the amount of data (and hence the communication bandwidth) is very much reduced from the initial input image data. However, additional buses can be employed if the communication requirement of the system exceeds the bandwidth of a single bus.

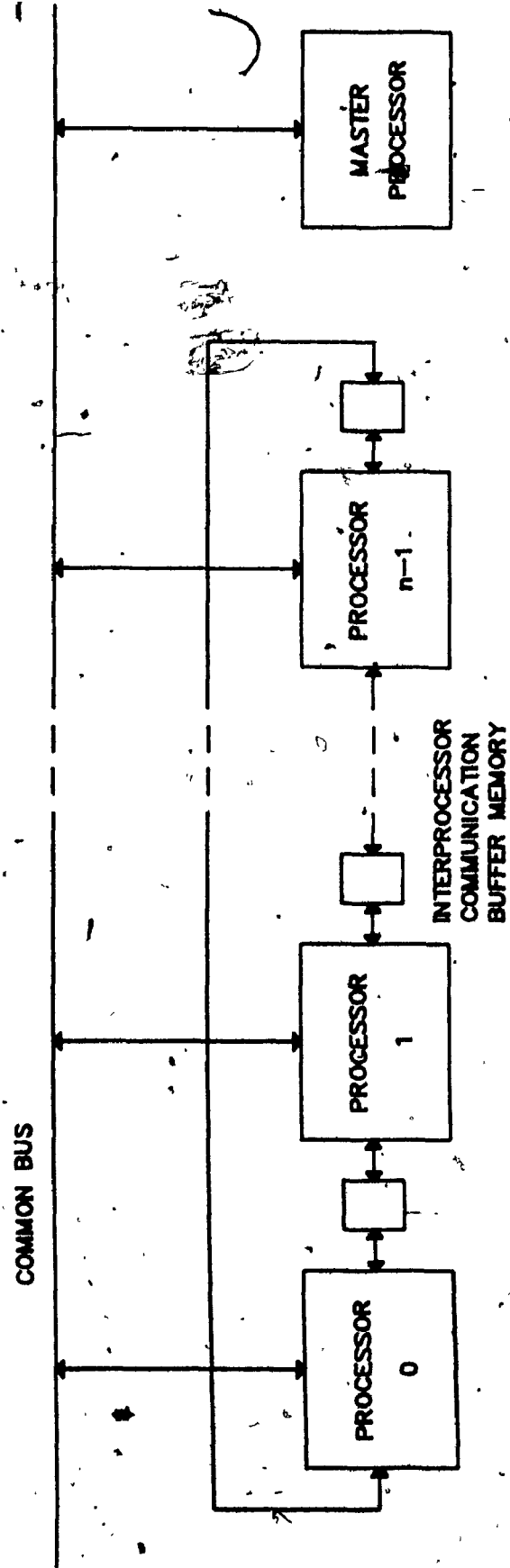


Fig. 2.1: The proposed parallel computer architecture.

2.2. IMAGE INPUT AND OUTPUT

The image input from and output to the master processor or I/O devices are carried out through the common bus. Each PE is assigned with N/n columns of the input image where N is the number of columns in the image. The actual column assignment scheme will be presented in Chapter V.

2.3. THE PROCESSING ELEMENTS

Each PE is based on a 16-bit microcomputer (DB32016 [16]) board. The PEs have a powerful instruction set and built-in multiprocessing features. Each board contains 128K dual-port RAM memory and can be expanded. The common bus shown in Fig. 2.1 is a Multibus [17]. Each board has the necessary interface for connecting it to the Multibus and also has a hardware floating-point unit attached to it. The use of identical PEs and simple interconnections between the PEs provide high reliability, since the system can withstand failures in several PEs as it is a relatively easy task to bypass a defective PE. The number of PEs used depends upon the speed/cost requirement of a specific application. Recent microprocessors such as Transputer [18] have been designed with more features such as built-in random access memory and communication links to communicate with the neighbors. However, this memory cannot be used by two PEs simultaneously as is the case in the present design. Also, the Transputer's communication links are serial resulting in a much slower data transfer rate.

2.4. INTERPROCESSOR COMMUNICATION

There are two paths through which a PE can pass a message to any other PE. One is the common bus which is used for communication while the PEs are executing high-level operations. The other path is the communication memory between the PEs as shown in Fig. 2.1. It is this memory that takes care of all the communications required in executing low-level operations. The CM consists of asynchronous $1K \times 8$ dual-port single-chip RAMs [19]. It has two independent sets of address, data, and control lines.

Two adjacent PEs can access different locations in the CM between them simultaneously. Contention occurs only when both the PEs try to access the same address location at the same time. To avoid the contention problem, algorithms are designed in such a way that at any specific instant the PEs access different areas of the same CM. As shown in Fig. 2.2, a CM is partitioned into two regions, one for writing and another for reading the neighborhood data and partial results. The algorithms are designed in such a way that the reading area of the CM to one PE is the writing area for the other PE connected to it. A PE is connected to two CMs, one CM is used to communicate with the right neighbor and the other CM is used to communicate with the left neighbor. A specific PE always reads the results from one area and writes the results into another area of the same CM. For the other PE connected to this CM, the read and write areas in the memory are interchanged. This way, the CM is available to both the connecting PEs either for simultaneous reading or for simultaneous writing. Although the memory is shared, in effect, to each PE it seems to be a part of a fast private memory. In order to ensure proper data communication, a PE always checks flags FLIN and FRIN before reading the data and checks flags FLOUT and FROUT before writing the data into the CMs.

2.5. SYNCHRONIZATION

For low-level operations, the PEs may have to be synchronized to ensure a sequence of data transfer between them that will lead to correct results. Fig. 2.3 shows a flowchart for the synchronization operation. At the beginning of an execution cycle, each PE reads results from both the CMs connected to it. A PE first reads a flag from one of the memories. If the flag is set, the PE reads results from that memory and resets the flag. If the flag is not set, the PE waits for it to be set. This operation is also repeated for reading results from the other memory. After reading the results, the PE executes the main body of the program producing some results for other PEs. At the

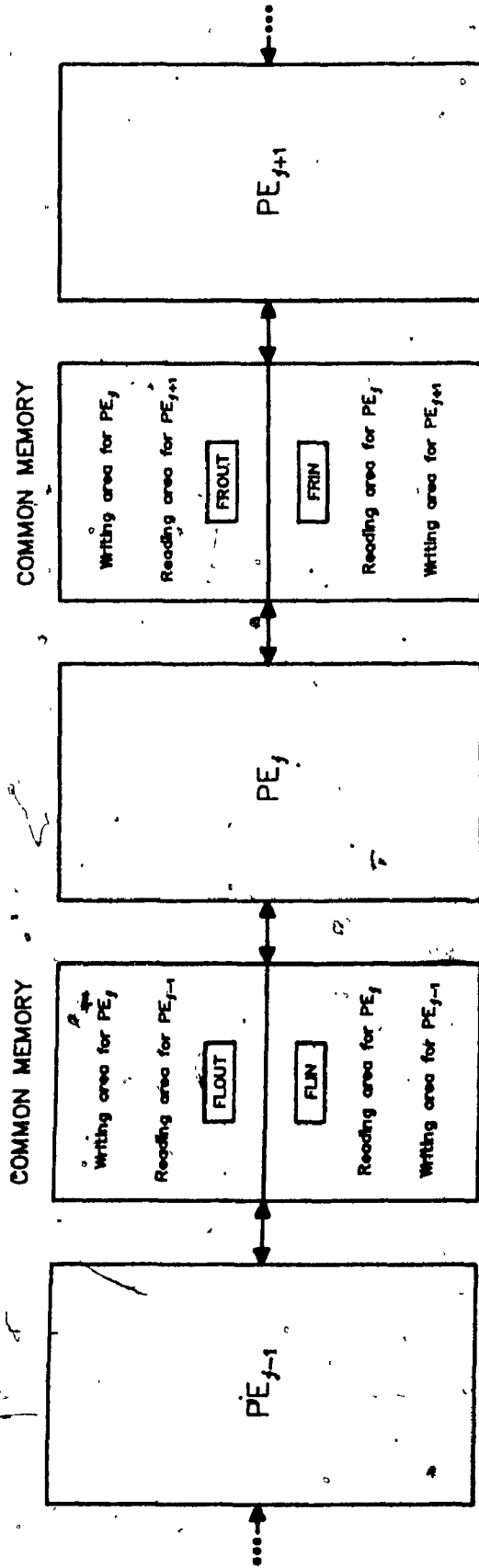


Fig. 2.2. Data communication scheme between the PEs through common memories.

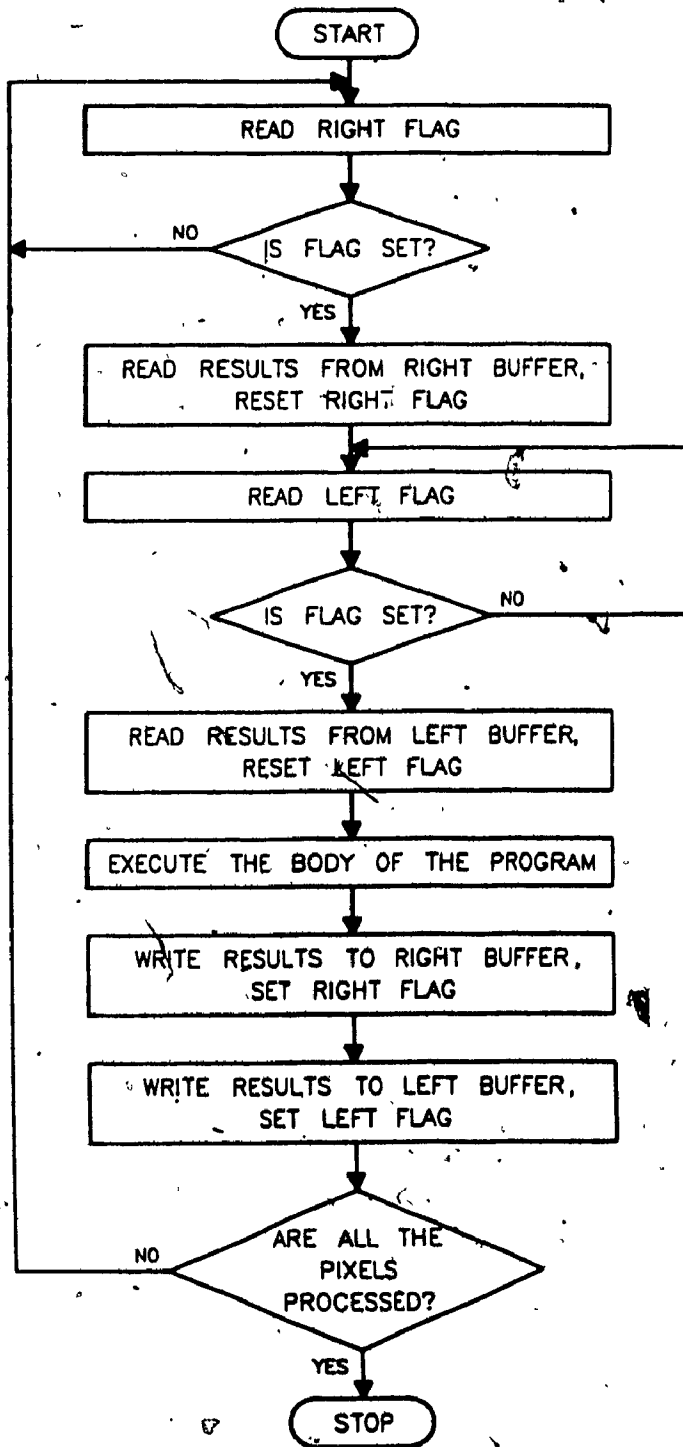


Fig. 2.3. Synchronization operation.

end of a cycle, the PE writes the results to both the CMs and sets the flags so that the neighboring PE can read the results in the next cycle. This sequence of reading and writing of partial results could be reversed depending on the requirement of the algorithm being implemented.

2.6. THE MASTER PROCESSOR

I/O devices such as scanners, image display units, and terminals are connected to and controlled by the master processor. After accepting commands from a user, it initiates the execution of the task by the PEs. In addition, it provides the program development environment such as creation and editing of files, compilation, linking, downloading, and debugging.

2.7. THE PROTOTYPE SYSTEM

A prototype system has been built and is used for testing the efficiency of parallel implementation of algorithms. A VAX 11/780 running on a 4.3 BSD Unix operating system is used as the master processor and two DB32016 microcomputer boards are used as the slave PEs. The NS16032 16-bit microprocessor functions as the central processing unit. Floating-point support is provided by the NS16081 unit. The PEs are connected to the master processor through two independent serial lines. Both data and control information are communicated to and from the master processor through these serial links. The slave PEs are interconnected through the Multibus.

The program and data files are created and edited on the master processor using one of the editors provided by the Unix operating system. The files are stored in the master processor. The cross-support package resident in the master processor, is used to compile or assemble NS16032 programs. The object program is then downloaded to the DB32016 boards (via the serial link) for execution. A high-level symbolic debugging is

possible by using the symbolic debugger. The user programs are written in C language. Pointers, a feature of the C language, are used for addressing specific memory locations of a PE in implementing the algorithms.

2.8. CONCLUSION

The architecture described in this chapter is essentially a bus-oriented system augmented with asynchronous dual-port memories between PEs for interprocessor communication. In order to minimize the contention problem, the memory space of each common memory module has been partitioned into regions. With what has been described so far, this architecture differs from other systems in its use of common memory modules as the main communication link between processors and its simplicity. Other aspects of the architecture such as data partitioning, synchronization, and the contention problem are discussed in Chapter V.

CHAPTER III

DESIGN AND IMPLEMENTATION OF A TWO-DIMENSIONAL CONVOLUTION ALGORITHM

3.1. INTRODUCTION

Two-dimensional convolution is one of the most commonly used algorithms in image processing [1]. Impulse responses in image processing have a finite order of $R \times S$ where R and S are usually in the range of 3 to 20. As a result, direct convolution is preferred to FFT for such filtering operations [20]. The choice of a structure for the implementation of a filter is governed by such factors as the hardware or software complexity, sensitivity, and the required speed. In image processing, speed is an important consideration due to the large amount of data to be processed.

There are two time-consuming operations involved in executing the two-dimensional convolution algorithm. First, as the convolution summation is a sum of products, a large number of multiplications is required. On small machines where multiplication operation is slow, multiplication tables have been used to speed up the execution of the algorithm. The second time-consuming operation is the address calculation of operands with two subscripts. There are two approaches to reduce the operand access time. One approach is to set up an access table whose entries correspond to the row subscript of the two-dimensional data [21]. The other approach is to read the data into a one-dimensional array [22] and to design an algorithm that carries out the operations efficiently. In this chapter, a design and implementation of a two-dimensional convolution algorithm that uses the second approach is presented. The algorithm runs faster than the algorithm with the direct implementation using multiplication and access

tables. While using a look-up table is a common procedure, this is not a necessity for using the proposed algorithm. With or without using the look-up table, the proposed algorithm offers certain advantages, as discussed later, over the direct implementation.

3.2. THE PROPOSED ALGORITHM

The convolution $g(i,j)$ of a two-dimensional discrete image $f(i,j)$ ($i = 0,1,\dots,M-1$; $j = 0,1,\dots,N-1$) and a two-dimensional window $w(i,j)$ ($i = -(R-1)/2,\dots,-1,0,1,\dots,(R-1)/2$; $j = -(S-1)/2,\dots,-1,0,1,\dots,(S-1)/2$) is given by

$$g(i,j) = \sum_{p=-(R-1)/2}^{(R-1)/2} \sum_{q=-(S-1)/2}^{(S-1)/2} f(i-p, j-q)w(p, q) \quad (3.1)$$

The direct software implementation of (3.1) consists of setting up two loops corresponding to the subscripts p and q , fetching the operands using the two subscripts, and finding the sum of the products. The image and window can also be expressed as two-dimensional matrices F and W , shown in Figs. 3.1 and 3.2, respectively. The terms of the convolution function form a matrix, $X(i,j)$, as shown in Fig. 3.3. In the direct implementation of (3.1), all the elements of the matrix $X(i,j)$ are evaluated and then added to find the value of $g(i,j)$, the convolution output corresponding to the pixel $f(i,j)$. In the proposed implementation, instead of multiplying all the pixel values in a neighborhood by the corresponding coefficients (Fig. 3.3), the products of the pixel value $f(i,j)$ with all the coefficients of the window are read from a look-up table producing a matrix $Y(i,j)$ shown in Fig. 3.4. The difference between the direct and proposed evaluation of (3.1) is apparent from the matrices $X(i,j)$ and $Y(i,j)$. In the first matrix, each term is produced by using two different values whereas in the second matrix, the pixel value is fixed. Therefore, a multiplication look-up table can be used more efficiently with the proposed method. Assuming the number of gray levels in the image to be G , the products of each pixel value with each of the coefficients in the W array can be computed

$$F = \begin{bmatrix} f(0,0) & \dots & f(0,j-1) & f(0,j) & f(0,j+1) & \dots & f(0,N-1) \\ f(i-1,0) & \dots & f(i-1,j-1) & f(i-1,j) & f(i-1,j+1) & \dots & f(i-1,N-1) \\ f(i,0) & \dots & f(i,j-1) & f(i,j) & f(i,j+1) & \dots & f(i,N-1) \\ f(i+1,0) & \dots & f(i+1,j-1) & f(i+1,j) & f(i+1,j+1) & \dots & f(i+1,N-1) \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ f(M-1,0) & \dots & f(M-1,j-1) & f(M-1,j) & f(M-1,j+1) & \dots & f(M-1,N-1) \end{bmatrix}$$

Fig. 3.1. Two-dimensional array representing pixels of an $M \times N$ image.

$$W = \begin{bmatrix} w\left(-\frac{R-1}{2}, -\frac{S-1}{2}\right) & \dots & w\left(-\frac{R-1}{2}, -1\right) & w\left(-\frac{R-1}{2}, 0\right) & w\left(-\frac{R-1}{2}, 1\right) & \dots & w\left(-\frac{R-1}{2}, \frac{S-1}{2}\right) \\ w\left(-1, -\frac{S-1}{2}\right) & \dots & w(-1, -1) & w(-1, 0) & w(-1, 1) & \dots & w\left(-1, \frac{S-1}{2}\right) \\ w\left(0, -\frac{S-1}{2}\right) & \dots & w(0, -1) & w(0, 0) & w(0, 1) & \dots & w\left(0, \frac{S-1}{2}\right) \\ w\left(1, -\frac{S-1}{2}\right) & \dots & w(1, -1) & w(1, 0) & w(1, 1) & \dots & w\left(1, \frac{S-1}{2}\right) \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w\left(\frac{R-1}{2}, -\frac{S-1}{2}\right) & \dots & w\left(\frac{R-1}{2}, -1\right) & w\left(\frac{R-1}{2}, 0\right) & w\left(\frac{R-1}{2}, 1\right) & \dots & w\left(\frac{R-1}{2}, \frac{S-1}{2}\right) \end{bmatrix}$$

Fig. 3.2. Matrix representation of an $R \times S$ window.

$$\begin{aligned}
 X(i,j) = & \begin{bmatrix}
 w\left(\frac{R-1}{2}, \frac{S-1}{2}, i+\frac{R-1}{2}, j+\frac{S-1}{2}\right) & \dots & w\left(\frac{R-1}{2}, -1, i+\frac{R-1}{2}, j+1\right) & w\left(\frac{R-1}{2}, 0, i+\frac{R-1}{2}, j\right) & \dots & w\left(\frac{R-1}{2}, 1, i+\frac{R-1}{2}, j-1\right) & \dots & w\left(\frac{R-1}{2}, \frac{S-1}{2}, i+\frac{R-1}{2}, j-\frac{S-1}{2}\right) \\
 w\left(-1, \frac{S-1}{2}, i+1, j+\frac{S-1}{2}\right) & \dots & w\left(-1, -1, i+1, j+1\right) & w\left(-1, 0, i+1, j\right) & \dots & w\left(-1, 1, i+1, j-1\right) & \dots & w\left(-1, \frac{S-1}{2}, i+1, j-\frac{S-1}{2}\right) \\
 w\left(0, \frac{S-1}{2}, i, j+\frac{S-1}{2}\right) & \dots & w\left(0, -1, i, j+1\right) & w\left(0, 0, i, j\right) & \dots & w\left(0, 1, i, j-1\right) & \dots & w\left(0, \frac{S-1}{2}, i, j-\frac{S-1}{2}\right) \\
 w\left(1, \frac{S-1}{2}, i-1, j+\frac{S-1}{2}\right) & \dots & w\left(1, -1, i-1, j+1\right) & w\left(1, 0, i-1, j\right) & \dots & w\left(1, 1, i-1, j-1\right) & \dots & w\left(1, \frac{S-1}{2}, i-1, j-\frac{S-1}{2}\right) \\
 w\left(\frac{R-1}{2}, \frac{S-1}{2}, i-\frac{R-1}{2}, j-\frac{S-1}{2}\right) & \dots & w\left(\frac{R-1}{2}, -1, i-\frac{R-1}{2}, j+1\right) & w\left(\frac{R-1}{2}, 0, i-\frac{R-1}{2}, j\right) & \dots & w\left(\frac{R-1}{2}, 1, i-\frac{R-1}{2}, j-1\right) & \dots & w\left(\frac{R-1}{2}, \frac{S-1}{2}, i-\frac{R-1}{2}, j-\frac{S-1}{2}\right)
 \end{bmatrix}
 \end{aligned}$$

Fig. 3.3. Terms of the convolution $g(i,j)$ arranged in a matrix form.

$$Y(i,j) = \begin{bmatrix}
 w(\frac{R-1}{2}, \frac{S-1}{2})f(i,j) & \dots & w(\frac{R-1}{2}, -1)f(i,j) & w(\frac{R-1}{2}, 0)f(i,j) & w(\frac{R-1}{2}, 1)f(i,j) & \dots & w(\frac{R-1}{2}, \frac{S-1}{2})f(i,j) \\
 w(-1, \frac{S-1}{2})f(i,j) & \dots & w(-1, -1)f(i,j) & w(-1, 0)f(i,j) & w(-1, 1)f(i,j) & \dots & w(-1, \frac{S-1}{2})f(i,j) \\
 w(0, \frac{S-1}{2})f(i,j) & \dots & w(0, -1)f(i,j) & w(0, 0)f(i,j) & w(0, 1)f(i,j) & \dots & w(0, \frac{S-1}{2})f(i,j) \\
 w(1, \frac{S-1}{2})f(i,j) & \dots & w(1, -1)f(i,j) & w(1, 0)f(i,j) & w(1, 1)f(i,j) & \dots & w(1, \frac{S-1}{2})f(i,j) \\
 w(\frac{R-1}{2}, \frac{S-1}{2})f(i,j) & \dots & w(\frac{R-1}{2}, -1)f(i,j) & w(\frac{R-1}{2}, 0)f(i,j) & w(\frac{R-1}{2}, 1)f(i,j) & \dots & w(\frac{R-1}{2}, \frac{S-1}{2})f(i,j)
 \end{bmatrix}$$

Fig. 3.4. Multiplication matrix formed by the product values of the pixel $f(i,j)$ with the window coefficients.

and stored in the look-up table $pd = \{pd(i), i = 0, 1, \dots, GRS-1\}$. It should be noted that the formation of GRS values of the look-up table does not require any multiplication. In fact, the table can be formed only with addition operations. Specifically, the i th entry in the table is the sum of $(i-RS)$ th entry and the $[i/RS]$ th coefficient, where $[i/RS]$ represents the largest integer less than or equal to the quotient i/RS . When a pixel value is accessed, the product values of this pixel with all the coefficients can be obtained readily from the RS consecutive locations of the look-up table. The address of the first location is the product of the pixel value with RS .

Instead of reading the function values from the two-dimensional arrays F and W and executing (3.1), these values are read column-by-column and row-by-row into one-dimensional arrays $f1$ and $w1$, respectively. For each pixel, the convolution output is developed through partial result arrays called $pter$ and $prest$ as defined below.

$$pter(k) = \sum_{p=0,1,\dots,R-1-[k/S]} f1(i+pN)w1(RS-S-pS+\text{mod}(k,S)), \quad k = 0, 1, 2, \dots, RS-1$$

where $[k/S]$ represents the largest integer smaller than or equal to the quotient k/S .

$$prest(k) = prest(k+1) + pter(k), \quad k = 0, 1, 2, \dots, S-2$$

$$prest(S-1) = pter(S-1).$$

An element in $pter$, as defined above, is the sum of a certain number of product values corresponding to a part of a column in a neighborhood. Since a pixel can be part of RS neighborhoods for an $R \times S$ window, there are RS elements stored in the $pter$ at any time.

Consider a 3×3 window function and a section of an image as shown in Figs. 3.5(a) and (b), respectively. The convolution output corresponding to the pixel $f(i,j)$ is produced as follows. When the pixel $f(i-1,j-1)$ is accessed, $pter(8) = f(i-1,j-1)w(1,1)$ is formed. When the pixel $f(i,j-1)$ is accessed, $pter(5) = pter(8) + f(i,j-1)w(0,1)$ is formed. When the pixel $f(i+1,j-1)$ is accessed, $pter(2) = pter(5) + f(i+1,j-1)w(-1,1)$ is formed.

$$W = \begin{bmatrix} w(-1,-1) & w(-1,0) & w(-1,1) \\ w(0,-1) & w(0,0) & w(0,1) \\ w(1,-1) & w(1,0) & w(1,1) \end{bmatrix}$$

(a)

$$\begin{matrix} f(i-1,j-1) & f(i-1,j) & f(i-1,j+1) \\ f(i,j-1) & f(i,j) & f(i,j+1) \\ f(i+1,j-1) & f(i+1,j) & f(i+1,j+1) \end{matrix}$$

(b)

Fig. 3.5. (a) A 3x3 window. (b) A section of an image.

The term $pter(1)$, corresponding to the middle column of the image segment, contains the sum of products of the middle row of coefficients in the W array and the corresponding values in the image segment. Similarly, $pter(0)$, corresponding to the last column of the image segment, contains the sum of the products of the first column of the coefficients and the corresponding data values in the last column of the image segment. The sum of $pter(2)$, $pter(1)$, and $pter(0)$ is the convolution output corresponding to the pixel $f(i,j)$. When the pixel $f(i+1,j-1)$ is accessed, $prest(2) = pter(2)$ is produced. In the next cycle, $prest(1) = prest(2) + pter(1)$ is formed. Finally, when the pixel $f(i+1,j+1)$ is accessed, $prest(0) = prest(1) + pter(0)$ is formed and that is the convolution output corresponding to the pixel $f(i,j)$. The algorithm coded in C language is presented in Appendix A.

3.3 THE STRUCTURE OF THE ALGORITHM

Equation (3.1) represents the difference equation of an FIR filter. Given a difference equation, a filter can be implemented with different structures [23]. A structure giving a direct implementation of (3.1) is shown in Fig. 3.6. The structure assumes a 3×3 filter impulse response and samples are computed row by row. Operators z_1 , z_1^{-1} , z_2 , and z_2^{-1} represent shift down, shift up, shift right, and shift left operations, respectively.

The structure of the proposed convolution algorithm is depicted in Fig. 3.7. The difference between this structure and that of Fig. 3.6 is the order in which the data values are shifted and multiplied. In the direct structure, the data values are first shifted and then multiplied by the appropriate coefficients to obtain the convolution sum. In contrast, a single data value is multiplied by the coefficients and the products are shifted and added to produce the output in the proposed structure. In software simulation of the two structures, the execution time of the proposed structure is found to be shorter. This is due to the fact that it takes less time to access the product values of a single pixel with a set of coefficients than to access the product values of a set of pixels with a set of coefficients.

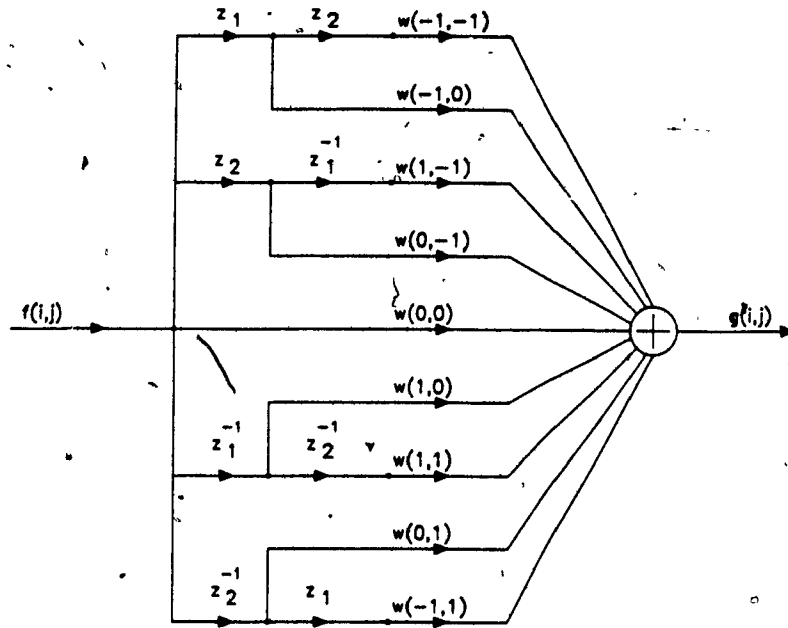


Fig. 3.6. A structure of the direct implementation of two-dimensional convolution.

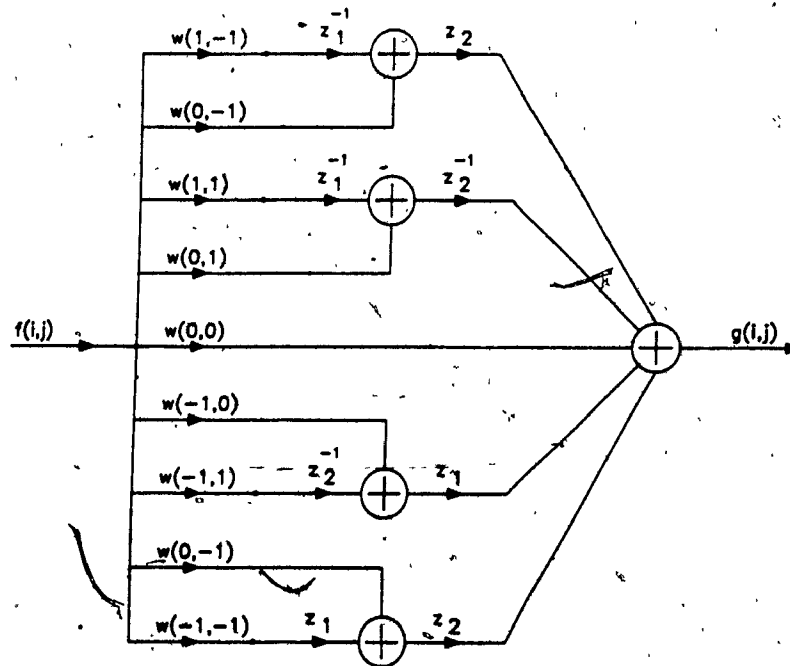


Fig. 3.7. The structure of the proposed implementation of two-dimensional convolution.

3.4. TEST RESULTS

The direct and the proposed implementations were used to convolve a 256×256 image with window functions of different sizes on the VAX 11/780 computer. The execution times for the various cases are shown in Table 3.1. The second and third columns show the execution time in seconds of the direct implementation of (3.1) without and with the use of multiplication and access tables, respectively. The fourth column shows the execution time of the proposed implementation using multiplication tables and one-dimensional indexing. For images with floating-point values, forming a multiplication table will be practically impossible. In that case, the only saving possible in execution time is due to the reduction in the operand access time by using either the access table or one-dimensional indexing. Table 3.2 shows the execution times of the algorithm for the floating-point case obtained by executing the direct and proposed algorithms.

3.5. DISCUSSION AND SUMMARY

In this chapter, an implementation for the two-dimensional convolution algorithm has been presented and its performance compared with the direct implementation. As shown in Table 3.1, the proposed algorithm runs faster by about 30% than the direct implementation using multiplication and access tables. The reduction in execution time is due to the fact that the proposed algorithm is designed in such a way that less time is spent in accessing the required values from the multiplication table in each cycle. This is due to the fact that the product values of a single pixel value with all the coefficients are used in each cycle and these values can be and are stored in consecutive locations. Hence, only the entry point in the multiplication table is to be calculated in each cycle. In the direct method the products of RS different sets of pixel and coefficients are required in each cycle and, therefore, the locations of these values in the multiplication table have to be found individually.

TABLE 3.1
 EXECUTION TIME IN SECONDS OF THE CONVOLUTION ALGORITHM
 FOR A 256 X 256 IMAGE WITH 256 GRAY LEVELS

Window Size	Execution Time for		
	Direct Implementation Without Multiplication and Access Tables	With Multiplication and Access Tables	Proposed Implementation With Multiplication Table and One-dimensional Indexing
3 X 3	16.3	11.4	8.4(26.3%)*
5 X 5	41.0	27.5	19.5(29.1%)
7 X 7	78.9	50.7	35.7(29.6%)

*The numbers in parentheses show the percentage savings in execution time of the proposed algorithm compared with the direct implementation with multiplication and access tables.

TABLE 3.2
 EXECUTION TIME IN SECONDS OF THE CONVOLUTION ALGORITHM
 FOR A 256 X 256 IMAGE WITH FLOATING-POINT VALUES

Window size	Execution Time for	
	Direct Implementation with Access Tables	Proposed Implementation with One-Dimensional Indexing
3 X 3	14.3	13.8
5 X 5	36.0	34.9
7 X 7	67.4	65.6

For the floating-point case, it is not possible to use multiplication table and as seen from Table 3.2, the execution time of the direct and the proposed methods are about the same. This is due to the fact that in this case the operand access time is the same for both the direct and the proposed implementations. The proposed implementation, irrespective of integer or floating-point representation of data, has an advantage: the computations can be carried out in place with less additional memory space as compared to the direct implementation.

Generally, the windows used in image processing are symmetric. The proposed algorithm can make use of this property to reduce even further the number of operations and thus the execution time. In this case, the number of additions required for the proposed algorithm becomes less in comparison with $(RS-1)$ additions required in the direct implementation. Consider a symmetric highpass filter with its coefficients specified as

$$W = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

For a given set of pixels, partial results need to be developed corresponding to the elements of the middle column of the window and only one of the other two columns. Obviously, the saving in the number of operations would be proportionately more for larger windows.

In the case of algorithms such as edge detection, the image is convolved with different windows for the detection of an edge pixel in different directions. The coefficients of all the windows are the same with the difference being the position of the coefficients in the window. In the proposed method, one set of multiplications is adequate to find the convolution outputs corresponding to a pixel for all orientations of a window. For example, in finding the gradient in four different directions, considerable saving in execution time is achieved because only RS multiplications are required compared to $4RS$ multiplications for the direct implementation.

A look-up table can also be used for images with large number of gray levels. As the data precision increases, a correspondingly larger memory is required. However, with the decreasing cost of memories coupled with the increased addressing capability of processors (with as many as 32 address lines), the approach is feasible for high precision images. Despite the additional memory requirement, the execution time remains invariant with increasing precision. This is in contrast with bit-dependent algorithms where both the required memory capacity and the the execution time increase with the precision of the image data.

CHAPTER IV

DESIGN AND IMPLEMENTATION OF A MEDIAN FILTERING ALGORITHM

4.1. INTRODUCTION

Median filtering is used in image processing to remove noise spikes [24]-[26]. The median of an odd number of m elements $p(i), i = 1, 2, \dots, m$, is the $[(m+1)/2]$ th largest element in the set. In image processing applications of median filtering, a window is moved from one row to the next along the columns of the image (it could also be from one column to the next along the rows) and the median of the pixels contained within the window at each position is computed. Finding the elements of a new window requires the replacement of a number of elements in the previous window equal to the number of columns in the window. The median obtained through this process is called the running median. Since the direct approach of sorting a given set of numbers and finding the median is a time-consuming operation, faster algorithms have been devised [27]-[34]. The method suggested in [27] and [28] to find the median of a set of numbers is to run an iterative procedure that reduces the magnitude of the problem after each iteration by eliminating some numbers in the set that cannot be the median. However, in finding the running median, it would be desirable to make use of the results obtained in computing the median of the previous window to process the current window. Algorithms discussed in [29] and [30] are based on the binary representation of the data for finding the median and they are efficient for array processors. However, the implementation of these algorithms on general-purpose computers will not be suitable, since most of the high-level languages, such as FORTRAN and PASCAL, do not provide bit manipulation facility.

Huang [31] has suggested a running median filtering algorithm based on using an updated histogram of pixel values in a window. A histogram of the pixels in the first window is set up and it is updated as the window moves from one position to another until the end of the column. This operation is repeated for each column in the image. The median of a window is found by adjusting the median of the previous window using the updated histogram. The histogram algorithm provides a fast execution time for images with small number of gray levels. For large number of gray levels in the image, however, the number of steps required to adjust the old median to the new one becomes large resulting in a considerably increased execution time. The worst-case execution time for this algorithm increases exponentially with the number of bits used to represent the data values. In [32], a table is set up for the window elements with a prescribed word-length. This table provides for each number in the given range, the frequency of elements in the window greater than or equal to that number. Using this table, a search is made to find the median. The number of comparisons for finding the median is constant while the number of values in the table to be updated is data dependent. The performance of this algorithm is also dependent on the number of bits used to represent the data values. For one-dimensional median filtering, an algorithm that is suitable for VLSI implementation has been described in [33]. This algorithm is based on storing an ordered list of input data and updating the list as a new data value arrives. The new input value is compared, in parallel, with all the values in the ordered list to find the position where it can be inserted in the list. While this approach can be implemented efficiently in hardware, it would not be suitable for fast software implementation. In [34], a parallel median filtering algorithm designed for use on byte-wide architecture array processors has been presented. In this algorithm, after shifting the neighborhood data, a binary search is made in the neighborhood consisting of K elements for an element that is greater than or equal to $(K-1)$ elements of the set. This algorithm, however, is bit-dependent and is efficient for only large windows.

In this chapter, a new median filtering algorithm is proposed. Instead of using an histogram and updating it as in [31] the elements of a window are put into two sets and these sets are updated for each window. The basis of the proposed algorithm is that if the elements of windows, in which the members of individual rows are prearranged in an ascending order, are stored row by row in an one-dimensional array, then the results obtained in partitioning of all the rows of the past window, except one row, can be used to partition and find the median of the current window. The results of the windows immediately above and to the left are used to find the median of the current window. This procedure to find median results in a faster execution time. Further, the execution time is independent of the number of bits (gray levels) used to represent the data and it is relatively insensitive to the noise level in the image. The invariance of the execution time for a given window size as the data word-length is increased is particularly useful in view of the fact that modern high-precision scanners have led to the processing of images of 10-, 12-, and 14-bit precision [35].

4.2. THE PROPOSED ALGORITHM

Given an $M \times N$ digital image, $f(i,j)$ ($i = 0,1,\dots,M-1; j = 0,1,\dots,N-1$), and an $R \times S$ ($R < M, S < N$) window within the image, the median filtering operation on the image consists of taking a set of RS pixels at each point (i,j) and replacing the pixel value $f(i,j)$ by the $[(RS+1)/2]$ th largest value of the set. The window, $W(i,j)$ corresponding to the pixel $f(i,j)$ expressed in a two-dimensional format, is shown in Fig. 4.1. Following the common practice in image processing, the window dimensions R and S are assumed to be odd. With minor modifications, the proposed algorithm can easily be adapted to the case where R and S are even. In the algorithm to be presented, the median of a window is found from an ordered set of row vectors of the window (the algorithm can also be implemented using column vectors). A row vector is formed from the sorted elements corresponding to each row of the window. Let us define a row vector $V(i,j,k)$

$$W(i,j) = \begin{bmatrix}
 f(i-\frac{R-1}{2}, j-\frac{S-1}{2}) & \dots & f(i-\frac{R-1}{2}, j-1) & f(i-\frac{R-1}{2}, j) & f(i-\frac{R-1}{2}, j+1) & \dots & f(i-\frac{R-1}{2}, j+\frac{S-1}{2}) \\
 f(i-1, j-\frac{S-1}{2}) & \dots & f(i-1, j-1) & f(i-1, j) & f(i-1, j+1) & \dots & f(i-1, j+\frac{S-1}{2}) \\
 f(i, j-\frac{S-1}{2}) & \dots & f(i, j-1) & f(i, j) & f(i, j+1) & \dots & f(i, j+\frac{S-1}{2}) \\
 f(i+1, j-\frac{S-1}{2}) & \dots & f(i+1, j-1) & f(i+1, j) & f(i+1, j+1) & \dots & f(i+1, j+\frac{S-1}{2}) \\
 f(i+\frac{R-1}{2}, j-\frac{S-1}{2}) & \dots & f(i+\frac{R-1}{2}, j-1) & f(i+\frac{R-1}{2}, j) & f(i+\frac{R-1}{2}, j+1) & \dots & f(i+\frac{R-1}{2}, j+\frac{S-1}{2})
 \end{bmatrix}$$

Fig. 4.1. Pixels in an $R \times S$ window in the neighborhood of pixel $f(i,j)$ of an image.

($k = -(R-1)/2, -(R-1)/2+1, \dots, (R-1)/2$) corresponding to the pixel $f(i,j)$, such that it consists of the pixels of the k th row of the window $W(i,j)$ arranged in ascending order. For example, $V(i,j,2)$ consist of the pixel values $f(i+2, j-(S-1)/2), f(i+2, j-(S-1)/2+1), \dots, f(i+2, j-1), f(i+2, j), f(i+2, j+1), \dots, f(i+2, j+(S-1)/2-1),$ and $f(i+2, j+(S-1)/2)$ arranged in ascending order. The approach of the algorithm is now presented with one of its objectives being to minimize the difference between the maximum and minimum number of comparisons required to find the median for different windows. As to be discussed in Section 5.6, this feature of the algorithm would be particularly useful for its parallel implementation.

The median corresponding to the pixel $f(i-1, j)$ is the median of a set of pixels in $W(i-1, j)$. The median corresponding to $f(i, j)$ is the median of RS elements in the neighborhood of $f(i, j)$; that is, in $W(i, j)$. This set of numbers can be obtained from the set of vectors of $W(i-1, j)$ just by deleting the vector $V(i-1, j, -(R-1)/2)$ and adding the row vector $V(i, j, (R-1)/2)$. This new vector is formed from the vector $V(i, j-1, (R-1)/2)$ by deleting the pixel $f(i+(R-1)/2, j-(S+1)/2)$ and inserting the pixel $f(i+(R-1)/2, j+(S-1)/2)$. Therefore, finding the row vectors corresponding to a new pixel, except for the first pixel in each column of the image, consists of deleting and inserting only one pixel from a past vector. Finding the median corresponding to the pixel $f(i-1, j)$ involves the partitioning of the set of pixels in $W(i-1, j)$ into two subsets: Subset 1 contains $(RS+1)/2$ elements and Subset 2 has one less, that are greater than or equal to the largest element in Subset 1.

The formation of the vectors $V(i, j, k)$ from the vectors $V(i-1, j, k)$ ($k = -(R-1)/2, -(R-1)/2+1, \dots, (R-1)/2$), updating of Subsets 1 and 2, and finally finding the median corresponding to $f(i, j)$ involves the following steps.

1. Discard the vector $V(i-1, j, -(R-1)/2)$. This results in reducing the number of elements either in one or both of the subsets. In the extreme case one subset may lose all S elements.

2. Form the vector $V(i,j,(R-1)/2)$ by deleting the pixel $f_{i+(R-1)/2,j-(S+1)/2}$ and inserting the pixel $f_{i+(R-1)/2,j+(S-1)/2}$ in an appropriate position of the vector $V(i,j-1,(R-1)/2)$.
3. The Subsets 1 and 2 of step 1 are updated by appropriately adding the elements from the vector $V(i,j,(R-1)/2)$ either in one of the subsets or in both. This is done by including all the elements less than or equal to the median corresponding to $f_{i-1,j}$ into Subset 1 and by making the remaining elements, which are greater than or equal to this median to be the part of Subset 2.

At this stage, it would be worthwhile to mention the following extreme cases. If S elements of the vector $V(i-1,j,(R-1)/2)$ were in Subset 1 then after the deletion of this vector, Subset 1 would have $(RS+1)/2-S$ elements. Now if all the elements in the new vector $V(i,j,(R-1)/2)$ go to Subset 2, then the number of elements in Subset 1 decreases to $(RS+1)/2-S$. On the other hand, if all the S elements of $V(i-1,j,(R-1)/2)$ were not in Subset 1, then after the deletion of this vector, Subset 1 still has $(RS+1)/2$ elements. Further, if all the S elements in the vector $V(i,j,(R-1)/2)$ go to Subset 1, then the number of elements in Subset 1 increases to $(RS+1)/2+S$.

4. Final partitioning of the set of vectors corresponding to $W(i,j)$, that is, updating of the two subsets and therefore, finding the median corresponding to $f(i,j)$ is done as follows: (i) If the number of elements in Subset 1 is one more than in Subset 2, final partitioning has already been achieved. Then, the new median is the largest element in Subset 1 along its border. (ii) If the number of elements in Subset 1 is less than $(RS+1)/2$, the smallest element of Subset 2 along the border is pushed into Subset 1 and thus the border is extended to the right. This process is repeated until the number of elements in Subset 1 becomes equal to $(RS+1)/2$. The last element brought into Subset 1 is the new median. (iii) If the number of elements in Subset 1 is more than $(RS+1)/2$, then the largest element of Subset 1 along the border is pushed to Subset 2.

and thus the border is extended to the left. This process is repeated until the number of elements in Subset 1 becomes equal to $(RS+1)/2$. The largest element in Subset 1 along the border is the new median.

In view of the two extreme cases mentioned above, determining the median of the new window involves the finding of the S smallest or largest elements along the border in one subset and pushing them into the other. It should be noted that only the determination of the median of the first window for each column of the image would require the finding of the $(RS+1)/2$ number of smallest elements. Even in the case of the first windows, the number of elements to be moved from one subset to the other is made less than $(RS+1)/2$ by first pushing all the elements that are less than the median of the past first window into Subset 1.

From the above the discussion, it is clear that there are three major steps in the implementation of the algorithm: (i) deleting and inserting of an element in forming a new row vector, (ii) finding the position of the partitioning of the new row vector, and (iii) moving the elements from one subset to the other. These operations are repeated for each window. A direct implementation of these operations results in a large execution time. The proposed algorithm coded in C language is presented in Appendix B. As seen from Appendix B and the explanation of the next section, a careful implementation of the algorithm is essential to achieve a fast execution time.

Example 4.1

Consider the image shown in Fig. 4.2(a). Assuming a window size of 5×5 , the elements in the window corresponding to the pixel $f(3,3)$ is shown enclosed in a box. The elements in the window are sorted and partitioned in to two subsets as shown in Fig. 4.2(b) with Subset 1 having 13 smallest elements of the window and the remaining 12 elements belonging to Subset 2. The 13th largest element (23) of Subset 1 is the median corresponding to the pixel $f(3,3)$. The median corresponding to the pixel $f(4,3)$ of Fig. 4.2(a) with its window $W(4,3)$ shown in Fig. 4.3(a), is found using the following steps.

13	21	11	18	15	26
24	23	14	26	24	23
15	26	5	30	22	26
12	22	6	32	30	31
27	23	8	31	29	23
14	24	14	25	22	21
21	30	32	45	53	37

(a)

S						S
u	14	23	23	24	26	u
b	5	22	28	26	30	b
s	6	22	30	31	32	s
e	8	23	23	29	31	e
t	14	21	22	24	25	t
l						2

(b)

Fig. 4.2. (a) Pixels in a section of the image of Example 4.1 with the window corresponding to the pixel $f(3,3)$. (b) The window of (a) after partitioning.

26	5	30	22	26
22	6	32	30	31
23	8	31	29	23
24	14	25	22	21
30	32	45	53	37

(a)

S					S	
u	5	22	26	26	30	u
b	6	22	30	31	32	b
s	8	23	23	29	31	s
e	14	21	22	24	25	e
t	30	32	37	45	53	t
1						2

(b)

S					S	
u	5	22	26	26	30	u
b	6	22	30	31	32	b
s	8	23	23	29	31	s
e	14	21	22	24	25	e
t	30	32	37	45	53	t
1						2

(c)

Fig. 4.3. (a) Elements in the window corresponding to the pixel $f(4,3)$ of Example 4.1. (b) Partitioning of the window in (a) after Step 3. (c) Final partitioning of the window in (b) after elements 24, 25, and 26 are transferred from Subset 2 to Subset 1.

1. Discard the vector $V(3,3,-2) = \{14,23,23,24,26\}$ corresponding to the window $W(3,3)$ (Fig. 4.2(b)). Due to this operation, the number of elements in Subset 1 reduces to 10 (i.e., elements 14, 23, and 23 are removed) and that in Subset 2 also decreases to 10 (i.e., elements 24 and 26 are removed).
2. The elements of $V(4,3,2)$ are found by deleting the element 21 and inserting the element 37 from the vector $V(4,2,2) = \{21,30,32,45,53\}$ giving $V(4,3,2) = \{30,32,37,45,53\}$.
3. The Subsets 1 and 2 of step 1 are updated by adding in them the elements from the new vector $V(4,3,2)$ obtained in step 2. The elements whose values are less than the past median (23), corresponding to the pixel $f(3,3)$ are pushed into Subset 1 while the remaining elements are made to be part of Subset 2. After this operation, Subset 1 has 10 elements and Subset 2 has 15 elements, as shown in Fig. 4.3(b).
4. Since the number of elements in Subset 1 is 10, the three smallest elements of Subset 2 must be pushed into Subset 1, thus extending the border between the subsets to the right. This can be done by repeating the operation of finding the smallest element in Subset 2 and pushing it to Subset 1. As presented in the next section and in Appendix B, in order to be more efficient, this is accomplished as follows. The two smallest elements along the border in Subset 2 are found to be 24 (in the 4th row) and 26 (in the 1st row). The second smallest element (26) along the border may or may not be the second smallest in Subset 2. Hence, the next number in the row vector (the 4th row) where the the smallest element was found, is compared with the second smallest element (26) along the border. Since the element 25 is smaller than 26, the element 25 is the second smallest element in Subset 2. After exhausting the elements of the 4th row, the element 28 becomes the third smallest number in Subset 2 and is pushed into Subset 1. At this point, Subset 1 has 13 elements which are

less than or equal to the smallest number in Subset 2 as shown in Fig. 4.3(c). The largest element (26) in Subset 1 which is the largest element along the final border and also the last element moved to Subset 1, is the median corresponding to the pixel $f(4,3)$.

4.3. TEST RESULTS AND DISCUSSION

The proposed and the histogram algorithms were coded in C language and tested with six different and arbitrarily chosen 256×256 images on VAX 11/780 computer. The images were corrupted by adding Gaussian noise to have a signal-to-noise ratio of about 13. Table 4.1 shows the average execution times for seven window sizes and images with 128, 256, and 512 gray levels. For each of the three gray levels, the proposed algorithm runs faster than the histogram algorithm. As the number of gray levels in the image increases, the difference in execution time between the two algorithms widens in favour of the proposed algorithm. This is due to the fact that, in the histogram algorithm, the smaller the number of gray levels in the image the lesser is the number of comparisons required to adjust an old median to a new value. The percentage run-time gain, PG , of the proposed algorithm over the histogram algorithm is defined as

$$PG = \frac{T_{hist} - T_{prop}}{T_{hist}} \times 100$$

where T_{hist} and T_{prop} are the execution times of the histogram and of the proposed algorithms, respectively. The percentage run-time gain of the proposed algorithm is presented in Table 4.1. Unlike the histogram algorithm, in the proposed method the number of comparisons required remains invariant as the number of gray levels changes. This factor makes the proposed algorithm ideally suited for median filtering in image processing applications over a wide range of gray levels including floating-point values.

TABLE 4.1
AVERAGE EXECUTION TIME IN SECONDS OF THE PROPOSED(P) AND THE
HISTOGRAM(H) ALGORITHMS AND THE RUN-TIME GAIN FOR SIX
DIFFERENT 256X256 IMAGES WITH 128, 256, AND 512 GRAY LEVELS

Window size	128 Gray Levels			256 Gray Levels			512 Gray Levels		
	P	H	Gain	P	H	Gain	P	H	Gain
3X3	12.6	15.2	17.1%	12.6	16.3	22.7%	12.9	19.7	34.5%
5X5	16.3	21.4	23.8%	16.7	21.9	23.8%	16.7	24.8	32.7%
7X7	20.1	25.6	21.5%	19.9	26.5	24.9%	20.7	30.8	32.8%
9X9	25.2	32.5	22.5%	25.5	33.6	24.1%	25.6	36.1	29.1%
11X11	29.5	39.4	25.1%	29.8	39.5	24.6%	30.2	41.6	27.4%
13X13	34.9	45.8	23.8%	35.4	46.7	24.2%	35.5	48.2	26.4%
15X15	41.4	50.9	18.7%	41.2	51.7	20.3%	41.8	54.3	23.0%

In the proposed algorithm, there are basically three operations that are repeated in each cycle. The first operation is to update the bottommost row in a window. This operation consists of deleting one element from a set of S elements and inserting one new element. This operation, using a sequential search, requires at the most S comparisons. The second operation is to search for the position of the element in the bottommost row vector that is greater than or equal to the old median. This operation also needs at the most S comparisons. In the proposed algorithm, the search for the element to be deleted is not started from the first position of the vector but at a position where the deletion occurred in the bottommost vector of the past window. As in typical images, the pixel values in adjacent columns or rows seldom vary abruptly, the number of comparisons required by starting with the previous position of the deletion, is found to be very small. Similarly, only a few comparisons are required to search for the position of an element in the bottommost row whose value is greater than or equal to the past median by starting the search at the partitioning position of the adjacent row.

At this point, the elements in each row vector of the window are in ascending order. The objective is finally to have the $(RS+1)/2$ elements in Subset 1 and $(RS-1)/2$ elements in Subset 2 and then to find the largest element in subset 1. In view of the earlier operations, the number of elements in the two subsets may not be in that proportion. Therefore, the third operation is to find the largest (if it is Subset 1) or the smallest (if it is Subset 2) in the subset which has more number of elements and push it to the other subset, thereby redefining the border between the two subsets. This operation consists of finding the smallest or the largest of a set of R numbers along the border. The operation is repeated at the most S times. As presented in Appendix B, in order to implement this third operation, the two largest or the two smallest numbers along the border are found in each cycle. There are two reasons for finding two numbers in one cycle instead of only one. First, the average number of comparisons required to find two numbers in the same cycle is less than that in two consecutive cycles. Second, there

is a possibility that one can find in the subset the third largest or smallest element, the fourth etc., with very little additional effort. For example, if the largest element of Subset 1 along the border is found in row vector 1 and the second largest in row vector 2, then all the numbers in vector 1 that are greater than or equal to this second largest number are, in order, the second largest element, the third largest element etc., in Subset 1. It may turn out that the two largest elements along the border are the two largest elements in the subset. In this case only the first advantage is realized. To find the first element, $(R-1)$ comparisons are required. To find the second element, R comparisons are required. This process is repeated at the most $(S+1)/2$ times. The last time, the second largest element is not found and the number of comparisons required is reduced by one. Therefore, for the third operation the number of comparisons required, in the worst case, is $-1+(2R-1)(S+1)/2$. The total number of comparisons required for the proposed algorithm, in the worst case, is $S+S-1+(2R-1)(S+1)/2$. The third operation minimizes the variation between the required minimum and maximum number of comparisons for finding the median of different windows of a given size. The features described above contribute to a fast execution of the algorithm.

The actual and the worst-case number of comparisons required for the proposed algorithm and the number of comparisons required in the worst case for the histogram algorithm are given in Table 4.2. Although on the average, the actual number of comparisons made per window is considerably less than in the worst case, it is a function of the worst-case number. Hence, the execution time of the algorithms will also be a function of the worst case number of comparisons. For larger windows, the number of comparisons required in the proposed algorithm becomes high and the efficiency decreases. But the decrease is relatively small for the window sizes commonly used in image processing applications.

TABLE 4.2
ACTUAL AND WORST-CASE NUMBER OF COMPARISONS
PER WINDOW IN THE PROPOSED AND THE
HISTOGRAM ALGORITHMS

Window Size	Proposed Algorithm		Histogram Algorithm Gray Levels		
	Actual	Worst-case	128	256	512
3x3	10.6	17	133	261	517
5x5	17.4	38	137	265	521
7x7	24.4	65	141	269	525
9x9	31.2	102	145	273	529
11x11	38.3	147	149	277	533
13x13	46.0	200	153	281	537
15x15	56.2	261	157	285	541

Another feature of this algorithm is that the execution time is relatively unchanged for increasing noise levels in the image. The presence of noise in the image makes the difference between the medians of adjacent windows large. This large difference in the medians does not affect the execution time of the proposed algorithm to the extent it does the execution time of the histogram algorithm. This is due to the fact that in the histogram method the search for the current median starts from the previous median and therefore, the number of comparisons and the execution time increase in proportion with the abruptness of the intensity changes. Table 4.3 shows the execution time of the two algorithms for a 256×256 image with different noise levels.

4.4. SUMMARY

A fast algorithm for two-dimensional median filtering has been presented and its performance compared. The main difference between this algorithm and the histogram algorithm is that while both the algorithms use the results obtained from the past window, the proposed algorithm updates two subsets of a window instead of an histogram so that the execution time is bit-independent. The proposed algorithm, in finding the median of a window, makes use of the results obtained from the two previous windows — one immediately above and the other to the left of the current window. The advantage of the algorithm is a fast execution time in comparison with other algorithms. In addition, the algorithm execution time is independent of data word-length and also it is relatively insensitive to the noise levels in the image. The algorithm can be easily adapted for multiprocessing as explained in the next chapter.

TABLE 4.3
THE EXECUTION TIME IN SECONDS OF THE PROPOSED AND THE
HISTOGRAM ALGORITHMS FOR DIFFERENT NOISE LEVELS IN AN
IMAGE WITH 256 GRAY LEVELS

Window Size	Signal to Noise Ratio					
	12		8		4	
	Proposed	Histogram	Proposed	Histogram	Proposed	Histogram
3X3	13.0	16.4	13.2	17.2	13.2	18.3
5X5	17.4	21.6	17.4	22.1	17.5	22.5
7X7	21.0	26.9	21.1	27.3	21.2	27.7

CHAPTER V

IMAGE PROCESSING ON THE PROPOSED PARALLEL ARCHITECTURE

5.1. INTRODUCTION

As stated in Chapter I, image processing operations fall under two classes: (i) low-level operations, and (ii) high-level operations. A characteristic of low-level operations is that the input data, in general, is in the form of large two-dimensional arrays. At this stage, handling of a large amount of data forms a major part of the overall processing time of a specific application. Hence, a parallel architecture must be highly optimized to do the low-level operations efficiently. In addition, it must also have the ability to process high-level operations. The low-level operations can be classified as [2] : (i) point operations, (ii) neighborhood operations, (iii) two dimensional discrete transforms, and (iv) image statistics. In this chapter, the implementations of these operations on the proposed architecture are discussed. Data partitioning is an important consideration in parallel processing. In Section 5.2, a new data partitioning scheme, specifically suited to the type of architecture discussed in Chapter II, is proposed.

5.2. DATA PARTITIONING

In parallel implementation of an algorithm the input data must be partitioned between the PEs in the system. Regardless of the way the image is partitioned between the PEs, the processing time of the complete image is the processing time of the PE that takes the longest time to complete its assigned segment. In a given cycle, the processing time of different PEs could be different if the algorithm execution time is data dependent. It could also vary from cycle to cycle for the same processor. For example in median filtering, the processing time of a specific segment could be considerably longer than that of other segments. This variation in execution time from one segment to the other can only be reduced by designing the algorithm such that, for different

windows, the variation in the number of comparisons required to evaluate the medians is small. Despite this variation in processing time, a high efficiency can be achieved by assigning the input data to the PEs as uniformly as possible from the point of view of algorithm execution time.

Let the number of columns in the image be N and the number of PEs in the system n . Assume that N is an integral multiple of n . Therefore, each PE is assigned N/n columns of the image. In the proposed data assignment scheme, PE_j ($j = 0, 1, 2, \dots, n-1$) is initially allotted with the column whose number is given by $C_j(0) = (N/n)j$. The other columns assigned to and processed by PE_j are then given by the recurrence relation

$$C_j(k) = pN + [C_j(k-1) - ((N/n) - 1)], \quad k = 1, 2, \dots, (N/n) - 1 \quad (5.1)$$

where p has a value of 1 for $k = j+1$ and it is zero otherwise. The column assignment to the PEs, as described above, allows the transfer of partial results of the previous cycle from the left adjacent neighbor. As an example, Fig. 5.1 shows the scheme of data assignment of an image with 16 columns assigned to 4 PEs. It also depicts the sequence in which the columns are processed and the transfer of partial results between the PEs at the beginning of each column cycle.

Broadly speaking, partitioning of images in parallel processing can be classified into two types. In one class, an image is partitioned into equal square segments [36] or into segments comprising consecutive rows or columns [37] requiring overlapped storage for neighborhood operations. In the other class, the rows or the columns of an image are assigned to PEs according to some predefined mapping scheme as described in [9] or in the proposed scheme. The square partitioning has the advantage of minimum overlapping between the segments [36]. In Huang's algorithm [31] or in the proposed median filtering algorithm of Chapter IV, the results of previous windows are used to process the current window. Therefore, the processing of the first windows takes much longer

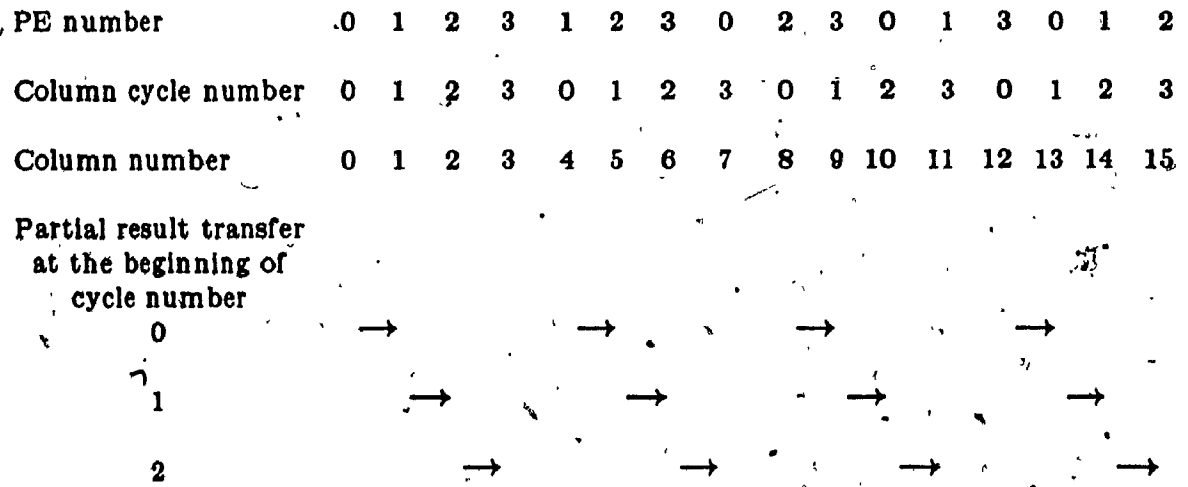
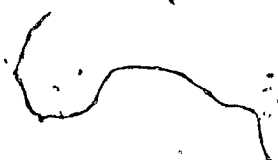


Fig. 5.1. Data partitioning and their assignment to PEs, transfer of partial results between PEs, and the processing sequence for $N = 16$ and $n = 4$.

time than the subsequent windows. As the square partitioning results in a larger number of first windows compared to other partitioning schemes, the processing time of the first windows becomes a larger proportion of the overall processing time of a segment. In the scheme of [36], the image has to be reformatted for executing algorithms such as FFT where a complete row or column is used in processing.

In the second class of data partitioning, neighborhood values and partial results are passed to other PEs during algorithm execution. Irrespective of the number of PEs in the system, the first windows are processed for each column or row only once. In the proposed method, however, the past results from two directions can be used. That is, the results from windows immediately above and left adjacent can be used in processing the current window. This approach, although possible in the first type of partitioning scheme, cannot be used in the partitioning of [9]. This is because of the way the data is assigned to different PEs in [9], the partial results from the left neighboring PE are still being formed when they are required.

The proposed data partitioning scheme provides an efficient parallel implementation of algorithms. In order to illustrate this, assume that there are two PEs in the system and N columns in the image. Each PE works on one half of the image. Let us further assume that the processing time, t , of each column in the first half of the image is twice the time of processing of a column in the second half. The processing of the image is complete only when both halves of the image have been processed, i.e., after $tN/2$ time units using the partitioning of [37]. Using the proposed data partitioning scheme, the processing of the image is complete in $3tN/8$ time units. A uniprocessor will process the image in $3tN/4$ time units. Thus, it is apparent that the proposed data assignment scheme is more efficient for parallel implementation of data-dependent algorithms.



5.3. POINT OPERATIONS

Point operations can be performed on each pixel of the image without the knowledge of neighboring pixels. These operations can be efficiently performed with any type of image partitioning strategy, since no communication between the PEs is required.

5.4. NEIGHBORHOOD OPERATIONS

In neighborhood operations, the output corresponding to a pixel depends not only on the pixel but also on neighboring pixels. For this type of operations, simultaneous exchange of data and partial results between neighboring PEs is required. The CM, the communication link in the proposed architecture, is used to provide a buffer for the data to be transferred. The neighborhood operations can be put into two categories: (i) operations that require almost fixed amount of processing time for each window, such as convolution, and (ii) operations that require varying execution time for each window, such as median filtering. The former type of operations can be done synchronously with good efficiency while the latter type of operations requires additional effort to achieve a high efficiency.

In both types of neighborhood operations, the processing of windows in the current column is divided into two parts. In the first part, that is executed first, partial results using data values and past partial results are produced. In the second part, output pixel values are computed. In the beginning of the execution of the second part, the results of the first part are passed on to neighboring PEs at the earliest possible time when the appropriate CMs are ready to receive them. This division of processing of a column cycle allows the receiving PE to continue with the next cycle, without any idling, if it has already completed its current cycle. Also, the data assignment scheme described above makes this approach to yield a high efficiency, particularly for data-dependent algorithms. The sequence of operations in the proposed implementation of all types of neighborhood algorithms are the same and can be outlined as follows.

1. For a column to be processed, carry out the initialization process (for example, array initialization), if necessary.
2. Process a window of the current column. At the end of processing, check the flags whether partial results and data are ready to be read in or written out. If the flags are set, carry out the read or write operation. Repeat the operations for the next window of the column.
3. If inputting or outputting of partial results and data is not complete, idle until the values are available and then read or write the partial results and data.
4. Update and write the partial results to be used by the neighboring PE in the next cycle. Find the output corresponding to the pixels in the current column.
5. Go to step 1.

The steps described above reduce the idling time of the PEs by using the execution time of a column cycle (the period of processing a complete column) as a buffer. This way, the difference between the execution times of two columns processed by a PE can differ by as much as the execution time of the column taking the longest time. Yet, it is highly probable that the PEs can still proceed with the processing of the next cycle immediately after completing the current cycle. In this study, this kind of synchronization is referred to as soft synchronization. The design and parallel implementation of two neighborhood algorithms, convolution and median filtering algorithms, will be described in Sections 5.5 and 5.6.

5.5. THE PARALLEL IMPLEMENTATION OF THE TWO-DIMENSIONAL CONVOLUTION ALGORITHM

In this section, a parallel implementation of the convolution algorithm described in Chapter II is presented. The PEs are assigned the segments of the input image, according to (5.1). The evaluation of convolution function given by (3.1) consists of finding the sum of products of the coefficients and the corresponding data values in a neighbor-

hood. In a multiprocessor environment, in order to find the convolution output $g(i,j)$, in general, a single PE evaluates all the elements of the matrix $X(i,j)$ and adds them up. In the proposed implementation, a PE accesses only one pixel value $f(i,j)$ at a time and the products of that value with all the coefficients of the window are read from a look-up table producing a matrix $Y(i,j)$ shown in Fig. 3.4. For example, for a 5×5 neighborhood, all the 25 products of a single pixel value with the 25 different coefficients of the window are read from the look-up table by a PE for each neighborhood. As will be shown later, only some of these product values are used by that PE in producing convolution outputs. The rest of the product values are added to partial results and passed on to other PEs using CMs between the PEs. The convolution output for each pixel is produced in a pipelined fashion.

5.5.1. The Approach of the Implementation

The number of PEs, n , in the system can vary from 2 to N . A column cycle is defined in such a way that during the i th column cycle the pixels of the i th column of the columns assigned to a PE are processed by it. During this period, a PE computes the elements of the matrix $Y(i,j)$ (Fig. 3.4), does the partial additions, transfers appropriate partial sums to the neighboring PEs, and computes the convolution output of all the pixels of a column. All the PEs are soft synchronized at the end of each cycle.

The values of all the columns of the two-dimensional $M \times N$ array F assigned to a PE are loaded into the PE as a one-dimensional array $f1 = \{f1(i), i = 0, 1, 2, \dots, (MN/n)-1\}$. The values of the two-dimensional $R \times S$ array W are read row-by-row into a one-dimensional array $w1 = \{w1(i), i = 0, 1, 2, \dots, RS-1\}$. The convolution output is stored in the *conout1* array. For each pixel, the convolution output is developed through partial result arrays called *pter* and *prest* as defined in Chapter III.

Consider an example of a 3×3 window function and a section of an image as shown in Figs. 5.2(a) and (b), respectively. The convolution output corresponding to the pixel $f(i,j)$, the sum of all the terms shown in Fig. 5.2(c), is produced as follows: When

$$W = \begin{bmatrix} w(-1,-1) & w(-1,0) & w(-1,1) \\ w(0,-1) & w(0,0) & w(0,1) \\ w(1,-1) & w(1,0) & w(1,1) \end{bmatrix}$$

(a)

$$\begin{matrix} f(i-1,j-1) & f(i-1,j) & f(i-1,j+1) \\ f(i,j-1) & f(i,j) & f(i,j+1) \\ f(i+1,j-1) & f(i+1,j) & f(i+1,j+1) \end{matrix}$$

(b)

$$X(i,j) = \begin{bmatrix} f(i-1,j-1)w(1,1) & f(i-1,j)w(1,0) & f(i-1,j+1)w(1,-1) \\ f(i,j-1)w(0,1) & f(i,j)w(0,0) & f(i,j+1)w(0,-1) \\ f(i+1,j-1)w(-1,1) & f(i+1,j)w(-1,0) & f(i+1,j+1)w(-1,-1) \end{bmatrix}$$

(c)

Fig. 5.2. (a) A 3x3 window. (b) A section of an image. (c) The array $X(i,j)$.

the pixel $f(i-1, j-1)$ is accessed by PE_k (it is assumed that the $(j-1)$ th column of the image has been assigned to PE_k according to (5.1), where k is a modulo n number), $pter(8) = f(i-1, j-1)w(1,1)$ is formed. When the pixel $f(i, j-1)$ is accessed by PE_k , $pter(5) = pter(8) + f(i, j-1)w(0,1)$ is formed. When the pixel $f(i+1, j-1)$ is accessed by PE_k , $pter(2) = pter(5) + f(i+1, j-1)w(-1,1)$ is formed. The term $pter(1)$ of PE_{k+1} , corresponding to the middle column of the image, contains the sum of the products of the middle row of coefficients in the W array and the corresponding pixel values in the image array. Similarly, $pter(0)$ of PE_{k+2} , corresponding to the last column, contains the sum of the products of the first column of the coefficients and the corresponding data values in the last column of Fig. 5.2(b). The sum of $pter(2)$, $pter(1)$, and $pter(0)$ from PE_k , PE_{k+1} , and PE_{k+2} , respectively, forms the convolution output corresponding to the pixel $f(i, j)$. When the pixel $f(i+1, j-1)$ is accessed, $prest(2) = pter(2)$ is produced and passed on to PE_{k+1} . In the next cycle, when PE_{k+1} accesses the pixel $f(i+1, j)$, $prest(1) = prest(2) + pter(1)$ is produced and passed on to PE_{k+2} . In the last cycle, PE_{k+2} forms $prest(0) = prest(1) + pter(0)$, which is the convolution output corresponding to the pixel $f(i, j)$. It should be noted that the PE holding the last column of pixel values in the window (i.e., PE_{k+2}) forms the final convolution output although the center pixel, $f(i, j)$, of the window is assigned to PE_{k+1} .

5.5.2 The Algorithm

During a given column cycle, each PE carries out the same set of operations. The parallel implementation of the algorithm is now presented in pseudocode form, by describing the operations performed by a PE. It is assumed that the image columns have been loaded in individual PEs according to (5.1).

THE VARIABLES

M = Number of rows in the image

N = Number of columns in the image

R = Number of rows in the window

S = Number of columns in the window

$count1$, $count2$, $count3$, and $count4$ are loop counters

G = Number of gray levels in the image

pd = GRS -element product table

$f1$ = MN -element input image array

$w1$ = RS -element coefficient array

$conout1$ = MN -element output image array

$pointer$ = Points to the product values in the look-up table

$pter$ = $R(S-1)$ -element partial values array in which each element is the sum of products corresponding to a part of the column in a window

$ppter$ = MS -element partial values array in which each element is the sum of products corresponding to a full column in a window

$prest$ = MS -element partial result array in which each element is the sum of one or more elements in $ppter$

$offset$ = The sum of the linear displacement between the storage locations of two corresponding partial results of two consecutive windows

$index$ = Index of the current pixel in $f1$

PROCEDURE MAIN

1. Initialize all $prest$ values to zero and reset FLIN.
2. Call Procedure TABLE to set up the multiplication table.
3. For $count1$ from 0 to $N-1$
 4. Initialize all $pter$ values, $offset$, and $index$ to zero.
 5. For $count2$ from 0 to $M-1$
 6. If flag FLIN is reset, read the past $prest$ values from the left CM. Set FLIN.
 7. Call procedure PARTIAL to update and create new partial results: the $pter$ values using the past values of $pter$ and the product values of

pixels in the current column with the coefficients.

8. Go to step 5.

9. Update the values of *prest* and find the output corresponding to the pixels in the current column by calling the procedure PRES.

10. Keep checking the flag FROUT until it is set. Write the past values of *prest* to the right CM. Reset FROUT.

11. Go to step 3.

{ At this point, all the columns have been processed. Since the initial values of partial results that form the output for the first $S-1$ columns have been assumed to be zero, these outputs have to be updated. The required partial results for a PE are stored in the second left neighboring PE. Hence, the partial results are shifted through a PE before the output updating cycle begins. }

12. Keep checking the flag FLIN until it is reset and then read the values of *prest*.

Set FLIN.

13. Keep checking the flag FROUT until it is set and then write the values of *prest*. Reset FROUT. Initialize *index* to zero.

14. For *count1* from 0 to $S-2$

15. Keep checking the flag FLIN until it is reset and then read the values of *prest*. Set FLIN.

16. Call procedure UPDATE.

17. Keep checking the flag FROUT until it is set and then write the values of *prest*. Reset FROUT.

END PROCEDURE MAIN

PROCEDURE TABLE

{ In this procedure, a look-up table containing the products of each gray level with each of the RS coefficients is computed and stored in the *pd* array. }

For *count1* from 0 to *RS-1*

index ← *count1*

sum ← 0

For *count2* from 0 to *G-1*

pd[index] ← *sum*

index ← *index* + *RS*

sum ← *sum* + *w1[count1]*

END PROCEDURE TABLE

PROCEDURE PARTIAL

{ In this procedure, the past values of *pter* are updated and new values of *pter* are created. }

pointer ← *RSf1[index]*

For *count9* from 0 to *S-1*

For *count4* from *count9* to $(R-1)S-1$ by *S*

pter[count4] ← *pter[count4 + S]* + *pd[count4 + pointer]*

pter[count9 + (R-1)S] ← *pd[count9 + (R-1)S + pointer]*

ppter[count9+offset] ← *pter[count9]*

offset ← *offset* + *S*

index ← *index* + 1

END PROCEDURE PARTIAL

PROCEDURE PRES

{ In this procedure, the values of *prest* are created or updated using the values of *ppter*.
The outputs corresponding to the pixels in the current column are computed. }

index ← *index* - *M*

offset ← 0

For *count2* from 0 to *M*-1

For *count3* from *offset* to *offset* + *S*-2

$prest[count3] \leftarrow prest[count3 + 1] + ppter[count3]$

$prest[offset + S-1] \leftarrow ppter[S-1+offset]$

$conout1[index] \leftarrow prest[offset]$

$offset \leftarrow offset + SR$

$index \leftarrow index + 1$

END PROCEDURE PRES

PROCEDURE UPDATE

{ In this procedure, the values of *prest* are shifted. The outputs corresponding to the pixels in the current column are updated. }

$offset \leftarrow 0$

For *count2* from 0 to *M*-1

For *count3* from *offset* to *offset* + *S*-2 - *count1*

$prest[count3] \leftarrow prest[count3 + 1]$

$conout1[index] \leftarrow conout1[index] + prest[offset]$

$offset \leftarrow offset + S$

$index \leftarrow index + 1$

END PROCEDURE UPDATE

5.5.3. Test Results

The percentage efficiency of the parallel implementation of an algorithm is defined as $100T_u/(n.T_p)$, where T_p is the execution time when the algorithm is implemented in parallel using n PEs and T_u is the time taken when only one PE is used. The efficiency of the parallel implementation of an algorithm would be 100% only if there were no overheads such as data communication between PEs, synchronization, and other additional operations required due to the parallel implementation of the algorithm.

The proposed convolution algorithm, coded in C language, was implemented serially and in parallel and tested on a single PE and on the 2-PE prototype of the parallel architecture, respectively. The 1-PE and 2-PE execution time and efficiency for a 256×256 image and for various window sizes are presented in Table 5.1. The serial and parallel execution time of the algorithm are shown, respectively, in the second and the third columns of the table. The efficiency of the parallel implementation, as defined above, is presented in the fourth column and it varies from 93.0% for a 3×3 window to 94.3% for a 15×15 window. Fig. 5.3(a) shows a 256×256 input image. Fig. 5.3(b) displays the edge output using a 3×3 Sobel operator in four directions, obtained by implementing the proposed algorithm on the 2-PE prototype architecture.

5.6. THE PARALLEL IMPLEMENTATION OF THE MEDIAN FILTERING ALGORITHM

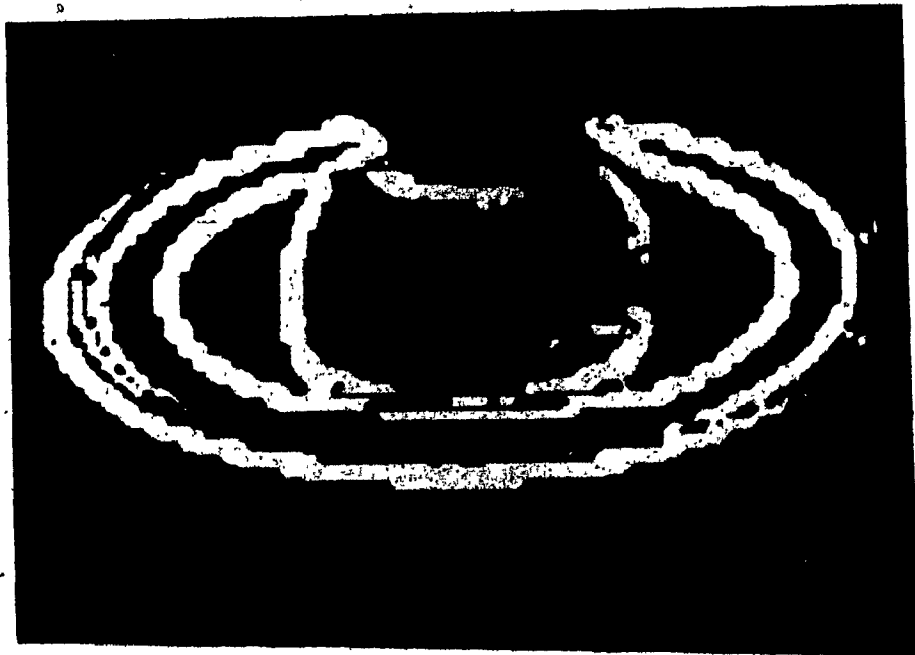
In this section, the problems related to the parallel implementation of the median filtering algorithm presented in Chapter IV are addressed. Since the execution time of the median filtering algorithm is data dependent, in order to achieve a high efficiency of the parallel implementation, an attempt is made in the design of the algorithm to narrow down the difference between the minimum and the maximum processing times of different windows. A synchronous execution is chosen for the parallel implementation. However, the idling time of processors waiting for data or partial results from neighboring PEs is minimized by reducing the number of synchronizing points. This is achieved by synchronizing the processors after processing a set of windows instead of each window. Further, at each synchronization point a hard synchronization is avoided by letting the processors continue, as long as possible, with the execution of the next cycle. The data partitioning scheme of this thesis makes it possible to use the partial results of past windows from two directions.

TABLE 5.1
EXECUTION TIME IN SECONDS AND THE EFFICIENCY OF
THE CONVOLUTION ALGORITHM FOR A 256 × 256 IMAGE
WITH 256 GRAY LEVELS.

Window size	Execution Time		Efficiency
	1 PE	2 PE	
3 × 3	21.4	11.5	93.0%
5 × 5	52.3	28.1	93.1%
7 × 7	97.0	51.9	93.5%
9 × 9	165.6	88.4	93.7%
11 × 11	235.0	125.1	93.9%
13 × 13	322.5	171.4	94.1%
15 × 15	424.2	224.9	94.3%



(a)



(b)

Fig. 5.3. (a) A 256×256 input image. (b) An edge output of the image in (a) by implementing the proposed convolution algorithm on the 2-PE prototype architecture.

5.6.1. The Approach of the Implementation

In a parallel processing environment, all PEs carry out the same set of operations. To start with, each PE writes to and reads from the CMs a number of pixel values in windows (for example, the pixel values from M windows) corresponding to the pixels in the first column of the image segment assigned to it. The PEs begin processing these windows. Each PE, for the first time, sorts the neighborhood pixels in all the row vectors of its first column. As these row vectors are to be used in the next column cycle by the right neighboring PE, a PE at the beginning of each window cycle checks whether the CM is free, i.e., whether the neighboring PE has already read the previously stored data. If the CM is free, the PE writes the row vectors so that the right neighboring PE can start the next column cycle immediately after finishing the processing of the windows of the current column. This process avoids hard synchronization and hence, reduces the idling time of PEs. The hard synchronization is the process of idling of PEs at the points in their current column cycle when they have completed the processing of their respective set of windows and then all the PEs resuming the next cycle by starting the processing of the next set of windows in the next column at the time after the PE taking the longest time has finished the processing in the current column cycle. Fig. 5.4 illustrates an example of a hard synchronization being employed. It is obvious that different PEs get idle at different points in their current cycle. However, it would be possible to resume the next cycle (i.e., the $(i+1)$ st cycle) only after PE₁, which takes the longest time, has completed the processing of its current set of windows in the i th cycle.

In the proposed implementation, a soft synchronization is employed as shown in Fig. 5.5 and explained below.

Initialization : Each PE writes to and reads from the neighboring PEs the required pixel values and sorts all the row vectors in all the windows of its first column.

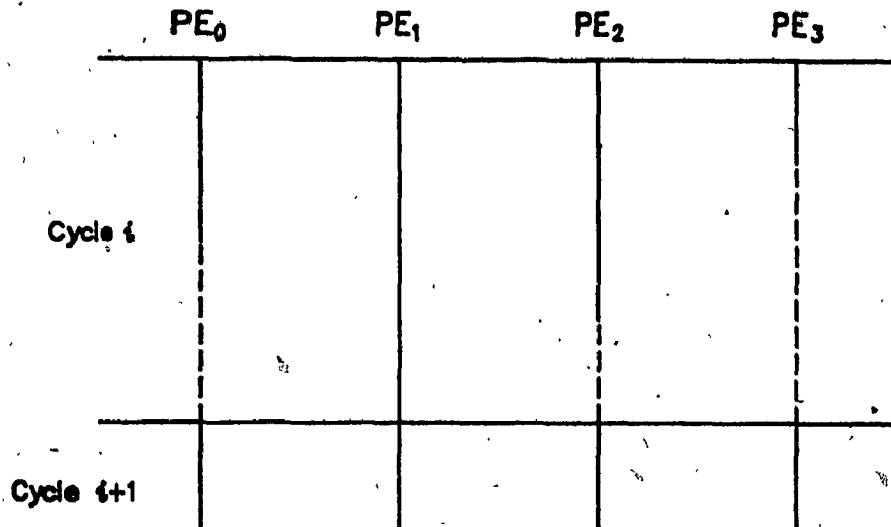


Fig. 5.4. Illustration of hard synchronization. Solid line shows the busy period of a PE and dotted line shows the idling period of a PE.

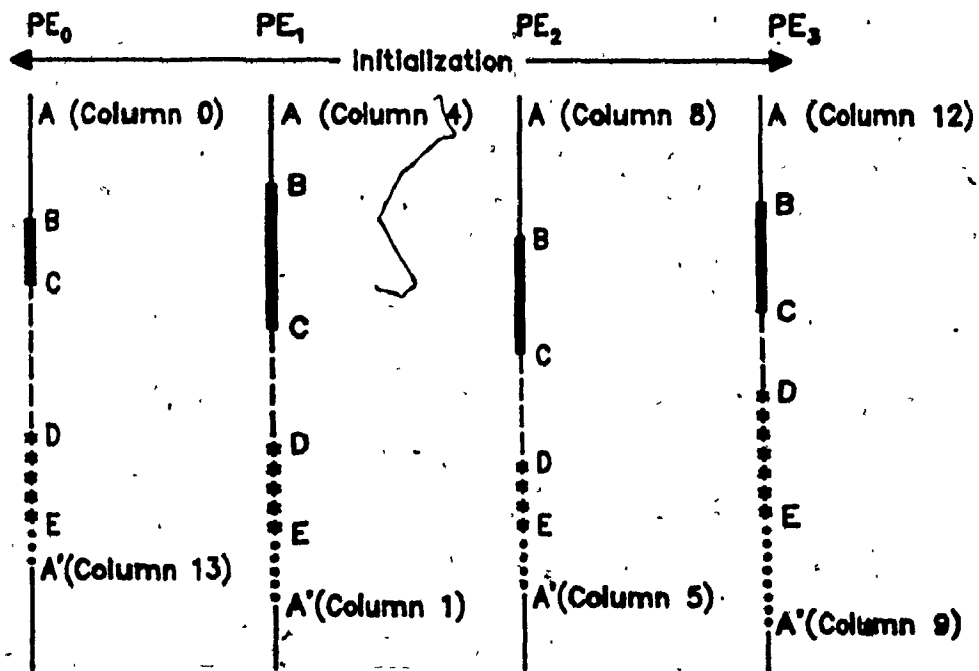


Fig. 5.5. A scheme of soft synchronization and illustration of various operations performed by a PE in a given column cycle for $N = 16$ and $n = 4$.

- A : Starting point of cycle 0.
- A to B : Check the FROUT and FLOUT flags to write in the CMs, the vectors and the delete and the insert elements. If the flags are not reset, process one window and check again. The pixels in the current column are needed by the neighboring PEs processing the $(S-1)/2$ th left adjacent column and the neighboring PE processing the $(S+1)/2$ th right adjacent column, since they are the insert and delete elements, respectively, for these PEs.
- B : Writing to CMs is completed. Set the flags.
- B to C : Processing of the remaining windows of the columns continues.
- C : Processing of all the windows in a column is complete.
- C to D : Keep checking FRIN and FLIN flags until they are set (idling period).
- D : Flags FRIN and FLIN are found set.
- D to E : Reading from the CMs is completed. Reset the flags.
- E to A' : Updating of all the vectors is carried out for the next column cycle. Doing this signals the end of the current cycle for a PE and the beginning of the next cycle.

Unlike hard synchronization, a PE can start its next cycle at point A' without waiting for the other PEs to complete their current cycle, since the required vectors are present in the private memory of the PE. It must be mentioned that a PE may have to idle itself for a longer period of time (segment CD in Fig. 5.5) if due to the nature of data, the difference in the processing times of column j by PE_i and column $(j+1)$ by PE_{i+1} is large for some consecutive values of j . This situation, however, is likely to occur very infrequently in the processing of typical images, since the windows corresponding to neighboring columns are not, in general, significantly different. In a nutshell, in order to reduce the idling times of the PEs, a two-stage buffering has been

employed: (i) synchronizing the PEs less frequently, i.e., after the processing of a set of windows rather than after each window and (ii) by avoiding hard synchronization after processing the set of windows. The parallel implementation of the algorithm is now presented using pseudo codes, incorporating all the points discussed above.

5.6.2 The Algorithm

Assume that the image columns have been assigned to the PEs according to (5.1).

PROCEDURE MAIN

1. Each PE writes the appropriate data in the CMs to form windows corresponding to all the pixels in the first column. Set the flags FLOUT and FROUT.
2. Check the FLIN and FRIN flags for the availability of data in the CM. Read in the data and reset the flags. Sort the pixels and form all the row vectors in all windows of the first column.
3. For all the columns to be processed
 4. Call FIND MIN to find the $(RS+1)/2$ th largest element of the first window.
 5. For all the windows in a column
 6. If flags FLOUT and FROUT are reset, write the vectors and insert and delete elements only once during the processing of the column and set the flags.
 7. Find the partitioning position of the new row with respect to the old median.
 8. Find the number of elements to be moved from one subset to another.
If the movement is from Subset 1 to Subset 2
 call FIND MAX
else
 call FIND MIN
 9. go to step 5.
11. Keep checking the flags FLIN and FRIN until they are set. Read in the

vectors and insert and delete elements and reset the flags. Update the vectors by calling the procedure INSERT DELETE.

12. Go to step 3.

END PROCEDURE MAIN

PROCEDURE FIND MAX

{ In this procedure, *count* largest elements are found in Subset 1 and pushed into Subset 2. }

If *count* > 0 do

 find the two largest numbers along the border

 decrement *count*

 update the border corresponding to the largest element

 If *count* = 0

 return

 while the next number is greater than or equal to the second largest element

 update the border

 decrement *count*

median ← number

 If *count* = 0

 return

 end while

{After all the elements in the row corresponding to the largest element along the border has been exhausted, the second largest element along the border is the next largest element in the set. }

median ← the second largest number

update the border

decrement *count* .

end

END PROCEDURE FIND MAX

PROCEDURE FIND MIN

{In this procedure, *count* smallest elements are found in Subset 2 and pushed into Subset 1. This procedure is similar to FIND MAX.}

END PROCEDURE FIND MIN

PROCEDURE INSERT DELETE

{In this procedure, the delete element is removed and the insert element is put into the corresponding previous vector to form a new vector.}

for each row vector

 find the deleting position in the vector

 discard the delete element

 find the inserting position

 put the insert element at the appropriate place

end

END PROCEDURE INSERT DELETE

5.6.3. Test Results and Discussion

The proposed median filtering algorithm, coded in C language, was implemented serially and in parallel and tested on a single PE and on the 2-PE prototype of the parallel architecture, respectively. The 1-PE and 2-PE average execution times and efficiency for six different 256×256 images and for various window sizes are presented in Table 5.2. The serial and parallel execution time of the algorithm are shown, respectively, in the second and the third columns of the table. The efficiency of the parallel implementation, as defined in Section 5.5.3, is presented in the fourth column and it varies from 76.8% for a 3×3 window to 74.3% for a 15×15 window. Figs. 5.6(a) and

TABLE 5.2
AVERAGE EXECUTION TIME IN SECONDS AND THE EFFICIENCY OF THE
MEDIAN FILTERING ALGORITHM FOR SIX DIFFERENT 256 X 256 IMAGES

Window size	Execution Time		Efficiency
	1 PE	2 PE	
3 X 3	53.2	34.6	76.9%
5 X 5	64.0	44.0	72.7%
7 X 7	76.8	54.2	70.9%
9 X 9	100.4	68.7	73.1%
11 X 11	120.0	81.5	73.6%
13 X 13	144.0	96.0	75.0%
15 X 15	165.6	111.4	74.3%



(a)



(b)

Fig. 5.6. (a) A 256×256 input image. (b) The image after the application of 5×5 median filtering by implementing the proposed algorithm on the 2-PE prototype architecture.

(b), respectively, show an input image and the output image after the application of a 5×5 median filtering by implementing the proposed algorithm on the prototype architecture.

The execution times of the proposed algorithm implemented by using the data partitioning scheme of Section 5.2 and the consecutive row or column-wise partitioning [37] are now compared. Table 5.3 shows the hypothetical execution times of three different sets of 16 columns, assuming that each image has 16 columns which are to be processed by 4 PEs. The time required for interprocessor data movement with the proposed method and the time required to overlap the border area of the segment in the second scheme have been ignored. Table 5.4 shows the execution time taken by each PE to process its assigned column for the two data assignment schemes. For Image 1, the proposed implementation using the proposed scheme finishes the execution in 26 seconds whereas the scheme of [37] requires 32 seconds of execution time. For Image 2 the execution times are the same for both the cases. For Image 3 the execution time using the proposed scheme is 32 seconds as compared to 20 seconds required by the other scheme. While for Image 1, the implementation using the proposed data assignment scheme provides a faster execution time compared to the other scheme, the result is opposite for Image 3. However, it should be noted that Image 1 is typical of practical images whereas Image 3 is not. In practice, two adjacent windows in an image would be quite similar and hence the execution time will also be very close as shown in Table 5.3 for Images 1 and 2. Due to the presence of noise in an image, the execution time of adjacent windows can differ by a large amount, but this is true for some pixels in a column, not for all the pixels. Therefore, the implementation using the proposed data partitioning scheme is likely to provide a faster execution time.

5.6.4. Conclusion

A fast parallel implementation of the median algorithm has been presented. Since the median filtering operations are data dependent, the major problem to overcome in

TABLE 5.3
TIME (HYPOTHETICAL) TO EXECUTE A COLUMN OF AN IMAGE

Image	Column Number															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	5	5	5	5	6	6	6	6	7	7	7	7	8	8	8	8
2	5	6	7	8	7	6	5	6	7	8	7	6	5	6	7	8
3	8	4	4	4	4	4	4	8	4	4	8	4	4	8	4	4

TABLE 5.4
TIME TAKEN BY PES TO PROCESS THEIR ASSIGNED SEGMENTS USING THE PROPOSED(P) AND CONSECUTIVE COLUMN(C) PARTITIONING SCHEMES.

Image	PE ₀		PE ₁		PE ₂		PE ₃	
	C	P	C	P	C	P	C	P
1	20	26	24	26	28	26	32	26
2	26	24	24	26	28	28	26	26
3	20	32	20	20	20	16	20	16

its design and parallel implementation has been to minimize the variation in execution time, for different windows and also of the various PEs as well. A solution to this problem has been achieved by an efficient algorithm design and an effective use of data partitioning. Specifically, the data partitioning scheme has allowed the use of the past results from two neighboring windows, a soft synchronization has reduced the idling time of PEs, and the comparison scheme has minimized the data dependency of the median-filtering operation. In effect, the proposed implementation tends to retain the advantages of synchronous and asynchronous modes of execution while minimizing the disadvantages of both the methods. Although the implementation has been described on a specific architecture with reference to the median filtering algorithm, the approach is general enough to be adapted for implementation of other data-dependent algorithms on any parallel architecture with simultaneous neighborhood communication facility.

5.7. IMPLEMENTATION OF THE TWO-DIMENSIONAL DISCRETE FOURIER TRANSFORM

Different transforms are used in image processing. However, the implementation scheme, in general, is the same for all, since the difference between the transforms is only in the coefficient matrix. Computation of the two-dimensional discrete Fourier transform of an image, by row-column decomposition, consists of two steps: computation of a row (column)-wise one-dimensional transform followed by a column (row)-wise computation. In parallel processing, the PEs are loaded with certain number of complete columns or rows of an image. After finding the transform by using an FFT algorithm in one direction, say column-wise, the image matrix is transposed in order to find the row-wise transform. First, each PE finds the column-wise one-dimensional transform for the columns of pixels it holds. For transposition of the partially (column-wise) transformed image, the CM between the PEs are used. For simplicity, it is assumed that the number of rows in the image is also equal to N , the number of columns. The PEs will be transferring $(N/n)(N/n)(n-1)$ values during the first cycle of data transfer,

$(N/n)(N/n)(n-3)$ values in the second cycle, $(N/n)(N/n)(n-5)$ values in the third cycle, and so on. Assuming n to be even, there will be a total of $n/2$ cycles of data transfer. As the transfers occur simultaneously, the total time for transposing the matrix is $(N/n)(N/n)(n-1+n-3+n-5+\dots+1)T = N^2T/2$, where T is the time required to transfer one value. A transfer is the task of reading and writing one value. Thus, for a given N , the number of transfers remains independent of the number of PEs in the system. After transposing, a row-wise transform results in the transform of the image.

Example 5.1

As an example, assume that there are 4 PEs in the system and that the size of the image is 4×4 . Each PE is allotted one column of the image. The PEs carry out column-wise one-dimensional transform. Assume that the results are as shown in Fig. 5.7(a). As there are 4 PEs, there will be 2 cycles to transpose the image. In the first cycle, each PE sends 3 values to adjacent neighbors and receives 3 values. For example, PE_0 transfers the values 21 and 31 to PE_1 and the value 41 to PE_3 and, receives the values 14 and 24 from PE_3 and the value 12 from PE_1 , as shown in Fig. 5.7(b). In the second cycle, each PE sends one value and also receives one value. PE_0 sends the value 24 to PE_1 and receives the value 13 from PE_3 . At the end of the second cycle, the transposition of the image is complete as shown in Fig. 5.7(c).

The execution time for finding the two-dimensional discrete Fourier transform of a 128×128 image on a single PE and on the 2-PE prototype architecture are 163.7 and 84.5 seconds, respectively. This results in a parallel implementation efficiency of 98.9%.

5.8. IMPLEMENTATION OF THE HISTOGRAM ALGORITHM

The parallel implementation of the histogram algorithm is presented as a representative of image statistics algorithms, since it is the most widely used algorithm belonging to this class. It is assumed that the number of gray levels in the image is G . All the PEs find the histogram of their image segments simultaneously. This is followed by

PE_0	PE_1	PE_2	PE_3
11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44

(a)

11	21	42	41
12	22	32	13
24	23	33	43
14	31	34	44

(b)

11	21	31	41
12	22	32	42
13	23	33	43
14	24	34	44

(c)

Fig. 5.7. Data transfer to transpose an image. (a) The image after column-wise transform. (b) Pixel assignment to PEs after the first cycle of data transfer. (c) Pixel assignment to PEs after the second cycle of data transfer.

$n/2$ cycles of data transfer in which $n/2$ and $(n/2)-1$ transfers take place to the right and left neighbors, respectively, for merging the partial histograms. During the j th cycle ($j = 0, 1, \dots, (n/2)-1$), PE_i ($i = 0, 1, \dots, n-1$) passes the number of pixels in each of the $[1+((n/2)+j+i+1) \bmod n]$ th G/n levels to the left adjacent PE and the number of pixels in each of the $[1+((n/2)-j+i) \bmod n]$ th G/n levels to the right adjacent PE. When a value of a histogram level is received by a PE, it is added with the corresponding value of its partial histogram. Thus, for $(n/2)-1$ cycles, $2G/n$ transfers take place in each cycle in both directions and in the last cycle G/n transfers in one direction only, resulting in a total transfers of $G(n-1)/n$. Obviously, for large number of PEs in the system, the number of transfers approaches the number of levels, G . This way the values are propagated and at the end of $n/2$ cycles, PE_i will hold the merged histogram values for $(i+1)$ st G/n levels. Each PE may need the histogram of the whole image in order to proceed with operations such as histogram equalization, segmentation by thresholding, etc. This is achieved in another $n/2$ cycles of data transfers that are similar to the merging of partial histograms except that when a PE receives a value from its neighbor it only stores the value without performing any addition operations and then passes it to the neighboring PE. In the j th cycle, PE_i transfers the number of pixels in each of the $[1+(i+j) \bmod n]$ th G/n levels to the left adjacent PE and the number of pixels in each of the $[1+(n+i-j) \bmod n]$ th G/n levels to the right adjacent PE.

Example 5.2

As an illustration, assume that there are 4 PEs in the system and number of gray levels in the image is 8. There will be two cycles to merge the partial histograms and the data movement is as follows:

Cycle I

PE_0 sends the number of pixels with levels 4 and 5 to PE_1 and the number of pixels with levels 6 and 7 to PE_3 .

PE₁ sends the number of pixels with levels 6 and 7 to PE₂ and the number of pixels with levels 0 and 1 to PE₀.

PE₂ sends the number of pixels with levels 0 and 1 to PE₃ and the number of pixels with levels 2 and 3 to PE₁.

PE₃ sends the number of pixels with levels 2 and 3 to PE₀ and the number of pixels with levels 4 and 5 to PE₂.

Cycle II

PE₀ sends the number of pixels with levels 2 and 3 to PE₁.

PE₁ sends the number of pixels with levels 4 and 5 to PE₂.

PE₂ sends the number of pixels with levels 6 and 7 to PE₃.

PE₃ sends the number of pixels with levels 0 and 1 to PE₀.

At the end of the two cycles, PE₀, PE₁, PE₂, and PE₃ will have the number of pixels in the whole image with gray levels of 0 and 1, 2 and 3, 4 and 5, and 6 and 7, respectively.

Another two cycles of data transfer are required for each PE to have the complete histogram of the image.

The execution time for finding the histogram of a 256x256 image on a single PE and on the 2-PE prototype architecture are 3.33 and 1.77 seconds, respectively. This results in a parallel implementation efficiency of 94.1%.

5.9. CONCLUSION

There are four factors that affect the implementation efficiency of an algorithm on a parallel architecture : (i) contention for a common resource, (ii) unequal work-load distribution among the processors, (iii) processors waiting for partial results and data, and (iv) insufficient concurrency in the algorithm. In this chapter, parallel processing schemes attempting to resolve these problems on the proposed architecture have been proposed.

In the proposed architecture, the major communication links between processors are the common memory modules. This common resource is contented only by two

processors. This feature along with the asynchronous nature of the common memory modules reduces the contention problem. The contention problem is even further reduced by splitting the common memory space into two regions during the algorithm execution, one for writing and the other for reading the neighborhood data and partial results.

As characteristics of an image may drastically vary from region to region, assigning continuous regions to processors in a parallel computer system leads to uneven workload distributions among processors when executing a data-dependent algorithm. To reduce this problem, a new data partitioning scheme for parallel processing has been proposed. The underlying principle of the data partitioning scheme is the involvement of each processor, as much as possible, in the processing of every region of the image.

The synchronization points have been reduced by synchronizing the processors after processing a set of windows instead of each window. This reduces the idling time of the processors, since the execution time is likely to even out over a large number of windows. In addition, the data and partial results are transferred much earlier than when they are required.

CHAPTER VI

CONCLUSION

6.1. SUMMARY

In this thesis, the design of some image processing algorithms along with a multiprocessor architecture for their implementations has been proposed. The parallel architecture is essentially a bus-oriented multiprocessor architecture with the inclusion of common memories for fast interprocessor communication. It has been found that the use of a dual-port common memory for interprocessor communication gives good performance for most image processing algorithms. Truly asynchronous dual-port memories are one of the recent VLSI products and are simpler than implementing an interconnection network in a multiprocessor system. Since, the low-level operations constitute a major part of all the different types of operations required in image processing, the architecture has been optimized for low-level processing while retaining its ability to process high-level processing as well. The salient features of the proposed architecture are its simplicity, usage of off-the-shelf components, flexibility, efficiency, and expandability.

With modern high precision scanners and processors, it is essential that in addition to providing a fast execution time, the number of operations in image processing algorithms are independent of the word-length. In the design of data-dependent parallel algorithms it is also important that the difference in execution times of different windows are minimized. Another desirable feature would be the capability of storing the input and output data in the same memory space. The algorithms described in this thesis has succeeded, to a large extent, in achieving these objectives.

The main feature of the convolution algorithm is to multiply each pixel value with all the coefficients and forming the output by adding the partial results produced at different cycles rather than multiplying all the window coefficients by the corresponding

pixels in the same cycle. This scheme compounded with the use of multiplication table reduces the processing time. This approach is also capable of making use of the property, that large number of windows used in image processing are symmetric, to reduce the number of addition operations required per window. Further, the number of multiplications required is considerably reduced for convolving an image with the same window in different orientations as in the case of edge detection.

Most of the median filtering algorithms described in the literature use some properties that depend on the number of bits used to represent the pixels and, therefore, the execution time increases with increasing word-length. The main features of the proposed median filtering algorithm are (i) its word-length independence with respect to execution time and memory space, (ii) shorter execution time by using the past results from two neighboring windows, and (iii) its low sensitivity to noise levels in the image.

There are four strategies that have been applied for the parallel implementation of the algorithms: (i) minimization of the data dependency of the algorithm execution time, (ii) an efficient data partitioning scheme, (iii) a reduction of the number of synchronizing points, and (iv) elimination or reduction of the idling time at synchronizing points. The data partitioning scheme makes it possible to use past results from two neighboring windows for synchronous operation resulting in a reduced number of operations for each new window. The idling time is reduced by making each PE to complete the evaluation of partial results required by other PEs and by itself before they are actually required and then move to carry out other operations. In summary, the implementation strategies tend to eliminate the drawbacks of a synchronous execution and provide an efficiency better than in the case of asynchronous execution. Although the parallel implementation strategies have been developed for the algorithms discussed in this thesis, they can be adopted for implementations of other algorithms on parallel architectures capable of providing simultaneous communication among neighboring PEs.

6.2. SCOPE FOR FUTURE INVESTIGATION

General-purpose microprocessors have been used in the design of the prototype architecture to study the efficiency of the parallel implementations of the algorithms. The PEs can be built with bit-slice microprocessors to provide high speed of execution and an instruction set that is more suitable to image processing requirements. A more effective study of the architecture can be made with a larger number of processors, say 16 or 32. The proposed architecture uses a single bus. For applications where bus congestion becomes a problem, a study can be made for the design of an efficient multiple bus architecture. A new operating system and a new language can be developed for the architecture.

In this thesis, two algorithms with bit-independent execution time have been presented. As the use of high precision scanners and processors is becoming more common, faster bit-independent algorithms need to be developed. In addition, the new algorithms should be decomposable to achieve an efficient parallel implementation. The problems of implementing high-level algorithms on the proposed architecture can also be studied.

Looking into the future, new mathematical tools that are better suited to image processing, need to be developed. Mathematical morphology [38,39], a recently introduced tool, is presently under investigation for its suitability to image processing. Residue [40] and distributed arithmetic [41-43] techniques have been applied for faster implementation of multidimensional filters and image processing algorithms. A more comprehensive study can be made as to the applicability of these techniques to MIMD type of image processing architectures. One way to design a parallel image processing system is to model it similar to the human brain. The most significant computing characteristics of the brain are the concurrent and widely distributed data processing, functional modularity, massive parallelism, and a capacity for self-organization [44].

Current research activities in neural networks are directed towards adopting some of these characteristics in the design of parallel architectures.

The computer architectures, parallel programming languages, and the design of algorithms and their implementations are relatively new and challenging areas which are open for extensive research.

REFERENCES

- [1] G. R. Nudd, "Image understanding architectures," in *Proc. Nat. Comp. Conf.*, pp. 377-390, 1980.
- [2] P. E. Danielson and S. Levialdi, "Computer architectures for pictorial information systems," *IEEE Computer*, vol. 14, no 11, pp. 53-67, Nov. 1981.
- [3] S. H. Unger, "A computer oriented towards spatial problems," *Proc. IRE*, vol. 46, pp. 1744-1750, 1958.
- [4] A. Rosenfeld, "Parallel image processing using cellular arrays," *IEEE Computer*, vol. 16, no. 1, pp 14-20, Jan. 1983.
- [5] K. E. Batchler, "Design of a massively parallel processor," *IEEE Trans on Comp.*, vol. C-29, no. 9, pp. 836-840, Sept 1980.
- [6] M. J. B. Duff, "Review of the CLIP image processing system," *Nat. Comp. Conf.*, pp. 1055-1080, 1978.
- [7] K. S. Fu, "Special computer architectures for pattern recognition and image processing - an overview," in *Proc. Nat. Comp. Conf.*; pp. 1003-1013, 1978.
- [8] G. H. Gfanlund, "The GOP parallel image processor," in *Digital Image processing Systems*, L. Bolc and Z. Kulpa, Eds., Berlin: Springer-Verlag, 1981, pp. 201-227.
- [9] J. L. Basille, S. Castan, and J. Y. Latil, "Systeme multiprocesseur adapte au traitement d'images," in *Languages and Architectures for Image Processing*, M. J. B. Duff and S. Levialdi, Eds., London: Academic, pp. 205-213, 1981.
- [10] H. J. Siegel, et al., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. on Comp.*, vol. C-30, no. 12, pp. 934-947, Dec. 1981.
- [11] R. T. Ritchings, "The CYBA-M multiprocessor for image processing," in *Image Processing System Architectures*, J. Kittler and M.J.B. Duff, Eds., London: Research Studies Press, 1985.

- [12] C. Rieger, J. Bane, and R. Trigg, "ZMOB: A highly parallel multi-processor," in *Proc. Workshop on Pict. Data Descrip. and Management, IEEE Comp. Soc.*, pp. 298-304, 1980.
- [13] N. J. Dimopoulos, "On the structure of the homogeneous multiprocessor," *IEEE Trans. on Comp.*, vol. C-34, no. 2, pp. 141-150, Feb. 1985.
- [14] D. Sundararajan and M. O. Ahmad, "A fast implementation of two-dimensional convolution algorithm for image-processing applications," *IEEE Trans. on Circuits Syst.*, vol. 5, CAS-34, no. 1, pp. 577-579, May 1987.
- [15] D. Sundararajan and M. O. Ahmad, "A fast two-dimensional median filtering algorithm," *IEEE Trans. on Circuits Syst.*, CAS-34, no. 11, pp. 1364-1374, Nov., 1987.
- [16] *Series 92000 data Book 1984*, National Semiconductor Corporation, U.S.A.
- [17] *Intel Multibus Interfacing, ISBC Applications Handbook*, Intel Corporation, U.S.A., 1981.
- [18] J. R. Newport, "The inmos transputer," in *32-Bit Microprocessors*, H. J. Mitchell, Ed., New York: McGraw Hill, 1986, pp. 93-130.
- [19] K.W. Pope, "Asynchronous dual-port RAM simplifies multiprocessor systems," *EDN*, vol. 28, no. 18, pp. 147-154, Sept. 1, 1983.
- [20] L. R. Rabiner and B. Gold, *Theory and Applications of Digital Signal Processing*. New York: Prentice-Hall, 1975.
- [21] L. Kruse, *Data Structures and Program Design*. New York: Prentice-Hall, 1984, pp. 105-107.
- [22] C. B. Kreitzberg and B. Shneiderman, *The Elements of Fortran Style*. New York: Harcourt Brace Jovanovich, 1972.
- [23] D. E. Dudgeon and M. Mersereau, *Multidimensional Digital Signal Processing*. New York: Prentice-Hall, 1984.

- [24] A. Rosenfeld and A. C. Kak, *Digital Picture Processing*, Vol. 1, New York: Academic, 1982.
- [25] R. C. Gonzalez and P. Wintz, *Digital Image Processing*. Reading, MA: Addison-Wesley, 1977.
- [26] M. D. Levine, *Vision in Man and Machine*. New York: McGraw-Hill, 1985.
- [27] L. I. Kronsjo, *Algorithms : Their Complexity and Efficiency*. New York: John Wiley, 1979.
- [28] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-wesley, 1983.
- [29] P. E. Danielsson, "Getting the median faster," *Computer Graphics and Image Processing*, vol. 17, no. 1, pp. 71-78, Sept., 1981.
- [30] E. Ataman, V. K. Aatra, and K. M. Wong, "A fast method for real-time median filtering," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-28, no. 4, pp. 415-420, Aug. 1980.
- [31] T. S. Huang, G. J. Yang, and G. Y. Tang, "A fast two-dimensional median filtering algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-27, no. 1, pp. 13-18, Feb. 1979.
- [32] V. V. Bapeswara Rao and K. S. Sankara Rao, "A new algorithm for real-time median filtering," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, no. 6, pp. 1674-1675, Dec. 1986.
- [33] G. R. Arce and P. J. Warter, "A median filter architecture suitable for VLSI implementation," in *Proc. 22nd Annual Allerton Conference on Communication, Control, and Computing*, University of Illinois, pp. 172 - 181, Oct. 1984.
- [34] K. N. Mathews, "Large window median filtering on CLIP7," *Pattern Recognition Lett.* vol. 1, no. 5,6, pp. 341-346, July 1983.

- [35] J. R. Adams, E. C. Driscoll, Jr., and C. Reader, "Image processing systems," in *Digital Image Processing Techniques*, Ed. M.P. Ekstrom, New York: Academic, 1984, pp. 289-360.
- [36] T. Kushner, A. Y. Wu, and A. Rosenfeld, "Image processing on ZMOB," *IEEE Trans. on Comp.* vol. C-31, no.10, pp. 943-951, Oct., 1982.
- [37] C. Rubat Du Merac, P. Jutier, J. Laurent, and B. Courtois, "A new domain for image analysis: VLSI circuits testing, with ROMUALD, specialized in parallel image processing," *Pattern Recognition Letters*, vol. 1, no. 5,6, pp. 347-357, July 1983.
- [38] J. Serra, *Image Analysis and Mathematical Morphology*. New York: Academic, 1982.
- [39] J. F. Bronskill and A. N. Venetsanopoulos, "The Pecstrum," in *Proc. 3rd ASSP Workshop on Spectrum Estimation and Modeling*, Boston, MA, eds. C. L. Nikias and J. G. Proakis, pp. 89-92, 1986.
- [40] C. H. Huang, "High-speed, two-dimensional filtering using residue arithmetic," in *Proc. SPIE*, vol. 241, pp. 215-218, 1980.
- [41] H. Jaggernauth and A. N. Venetsanopoulos, "Real-time time-varying image processing through distributed arithmetic," in *Proc. Int. Workshop Time-Varying Image Process. Moving Object Recogn.*, Florence, pp. 22-25, May 1982.
- [42] H. Jaggernauth and A. N. Venetsanopoulos, "Distributed arithmetic implementation of two-dimensional filters," in *Proc. IEEE Can. Commun. Energy Conf.*, Montreal, pp. 407-410, Oct. 1982.
- [43] H. Jaggernauth and A. N. Venetsanopoulos, "Real-time image processing through distributed arithmetic," in *Proc. 1983 IEEE Int. Symp. Circuits Syst.*, Newport Beach, Calif., pp. 394-397, May 1983.
- [44] J. J. Vidal, "Silicon bains : Whither Neuromimetic computer Architectures," in *Proc. IEEE Int. Conf. on Computer Design : VLSI in Computers*, pp. 17-20, 1983.

APPENDIX A

THE TWO-DIMENSIONAL CONVOLUTION ALGORITHM

This appendix presents the proposed convolution algorithm as a function called CONVOL, coded in C language. This function may be called from another function which incorporates the reading and writing of the image. The only parameters required by the function are the sizes of image and window. These values must be assigned to the appropriate constants in the constant definition part of the program.

THE VARIABLES

ix = MN-element input array

iout = MN-element output array

pter = RS-element partial terms array

prest = MS-element partial result array

row = points to the current row

column = points to the current column

pointer = points to the location of product values corresponding to a pixel

pd = NGRS-element product table array */

/* Constant Definitions */

#define M 256 /* number of rows in the image */

#define N 256 /* number of columns in the image */

#define R 5 /* number of rows in the window */

#define S 5 /* number of columns in the window */

#define G 256 /* number of gray levels in the image */

#define RMS R*S

```
#define MMS M*S
```

```
#define SM1 S-1
```

```
#define RM1MS (R-1)*S
```

```
int out[M*N]
```

```
CONVOL()
```

```
{
```

```
    extern int ix[M*N], coef[RMS];
```

```
    int row, column, x1, y1, sum, index, offset, pointer;
```

```
    int pter[RMS], prest[MMS], pd[NG*RMS];
```

```
/* In the following segment, a product table, pd is set up. The products corresponding to a gray level with each of the coefficients is stored consecutively in the pd array */
```

```
    for(x1 = 0; x1 < RMS; x1++) {
```

```
        index = x1;
```

```
        sum = 0;
```

```
        for(y1 = 0; y1 < G; y1++) {
```

```
            pd[index] = sum;
```

```
            index += RMS;
```

```
            sum += coef[x1];
```

```
        }
```

```
    }
```

```
/* processing is done column by column */
```

```
    for(column = 0, index = 0; column < N; column++) {
```

```
/* For each column the elements in the pter array are initialized to zero */
```

```
        for ( x1 = 0; x1 < RMS; x1++) .
```

```
            pter[x1] = 0;
```

```
        for(row = 0, offset=0; row < M; row++, offset += S; index++) {
```

```
            pointer = RMS * ix[index];
```

/* updating pters */

```
for(x1 = 0; x1 < S; x1++) {  
    for(y1 = x1; y1 < RM1MS; y1 += S)  
        pter[y1] = pter[y1 + S] + pd[y1 + pointer];  
    pter[x1 + RM1MS] = pd[x1 + RM1MS + pointer];  
}
```

/* updating prets */

```
for(x1 = offset; x1 < offset + SM1; x1++)  
    prest[x1] = prest[x1 + 1] + pter[x1 - offset];  
prest[offset + SM1] = pter[SM1];  
out[index] = prest[offset];
```

APPENDIX B

THE MEDIAN FILTERING ALGORITHM

The proposed median filtering algorithm is presented as a function called FAST_MEDIAN; coded in C language. This function may be called from another function which incorporates the reading and writing of the image. The only parameters required by the function are the sizes of image and window. These values must be assigned to the appropriate constants in the constant definition part of the program. Function FAST_MEDIAN calls other functions named SORT, FIND_MIN, FIND_MAX, and INSERT_DELETE.

THE VARIABLES

ix = MxN input array

iout = (M-R+1)x(N-S+1) output array

median = the median of the past window

fmedian = the median of the first window in the previous row

row = points to the current row

column = points to the current column

newcol = the column number assigned to the rightmost column of a window

position = position where the old median partitions the rightmost column

count = number of pixels to be moved from one subset to the other so that Subset 1 has $(RS+1)/2$ number of elements

pointer = points to the column of the window in which the median was found

colvec = 1-dimensional $N(R+2)$ -element array representation of an $(R+2) \times N$ 2-dimensional image segment in which sentinel values have been inserted in the beginning and at the end of each column

border = 1-dimensional array of S elements holding the partitioning position of each column vector of the window. The elements at the borders belong to Subset 2

```
/* Constant Definitions */
```

```
#define M 256 /* number of rows in the image */
```

```
#define N 256 /* number of columns in the image */
```

```
#define R 5 /* number of rows in the window */
```

```
#define S 5 /* number of columns in the window */
```

```
#define RP2 (R+2)
```

```
#define RM1BY2 ((R-1)/2)
```

```
#define RP1BY2 ((R+1)/2)
```

```
#define SM1BY2 ((S-1)/2)
```

```
#define SP1BY2 ((S+1)/2)
```

```
#define RSP1BY2 ((R*S+1)/2)
```

```
#define RP2MULS (RP2*S)
```

```
#define MMRM1BY2 (M-RM1BY2)
```

```
#define NMSM1BY2 (N-SM1BY2)
```

```
#define POSITION (S*RP2+RP1BY2)
```

```
int colvec[N * RP2], border[S], median, count;
```

```
int pointer, row, lout[M - R + 1][N - S + 1];
```

```
FAST_MEDIAN()
```

```
extern int ix[M][N];
```

```
int column, position, newcol, fmedian;
```

```
/* initialization with terminal values, assignment, and sorting of columns in the first
row to be processed. */
```

```
    SORT();
```

```
    fmedian = 0;
```

```
    for (row = RM1BY2; row < MMRM1BY2; row++) {
```

```
/* In the following segment the median of the first window of each row is found. */
```

```
    count = 0;
```

```
    for (column = 0, position = 1; column < S; column++, position += RP2) {
```

```
        border[column] = position;
```

```
        while (colvec[border[column]] < fmedian) {
```

```
            border[column]++;
```

```
            count++;
```

```
        }
```

```
    count -= RSP1BY2;
```

```
    if (count >= 0) {
```

```
        count++;
```

```
        FIND_MAX();
```

```
        border[pointer]++;
```

```
    }
```

```
    else
```

```
        FIND_MIN();
```

```
    iout[row - RM1BY2][0] = fmedian = median;
```

```
    position = POSITION;
```

```
    newcol = 0;
```

```
for (column = SP1BY2; column < NMSM1BY2; column++) {
```

```
/* the number of elements to be moved between subsets is found */
```

```
count = position - border[newcol] - RP2MULS;
```

```
/* the border position in the new column is found such that the element at the border  
is greater than or equal to the past median and the count is minimum */
```

```
if (colvec[position] > median || (colvec[position] == median && count  
>= 0)) {
```

```
/* searching upwards for the border position */
```

```
while (colvec[position - 1] >= median) {
```

```
if (colvec[position - 1] == median && count <= 0)
```

```
break;
```

```
position--;
```

```
count--;
```

```
else {
```

```
/* searching downwards for border position */
```

```
count++;
```

```
while (colvec[++position] <= median) {
```

```
if (colvec[position] == median && count >= 0)
```

```
break;
```

```
count++;
```

```
border[newcol] = position;
```

```
if (count > 0 || (count == 0 && pointer == newcol)) {
```

```
/* Border adjustment by moving the elements from Subset 1 to Subset 2 since the past  
median is the largest number in Subset 1, the first one moved to Subset 2 is the past  
median */
```

```
if (colvec[--border[pointer]] == median)
```

```
else {
```

```
border[pointer]++;
```

```
count++;
```

```
}
```

```
FIND_MAX();
```

```
border[pointer]++;
```

```
else
```

```
/* Border adjustment by moving the elements from Subset 2 to Subset 1 */
```

```
FIND_MIN();
```

```
lout[row - RM1BY2][column - SM1BY2] = median;
```

```
position = border[newcol++] + RP2;
```

```
if (newcol == S)
```

```
newcol = 0;
```

```
if (row < MMRM1BY2 - 1)
```

```
INSERT_DELETE();
```

```
/* In this function, the terminal elements of each column are assigned values. The value at one end is smaller than the smallest pixel value in the image and at the other end the value is larger than the largest pixel value in the image. The pixels in the columns of the windows corresponding to the pixels in the (R + 1)/2th row in the image are stored in the colvec array and sorted. */
```

```
SORT().
```

```
{
```

```
int i, i1, i2, index, temp;
```

```
/* initialize the terminal values of each column */
```

```
for (i = 0; i1 = S + 1; i < N * RP2; i += RP2, i1 += RP2) {
```

```
colvec[i] = (- 2);
```

```
colvec[i1] = 800;
```

```
}
```

```
/* insert and sort the pixels in the colvec array */
```

```
for (i = 0; index = 1; i < N; i++, index += RP2) {
```

```
colvec[index] = ix[0][i];
```

```
for (i1 = index + 1; i1 < index + S; i1++) {
```

```
temp = ix[i1][index][i];
```

```
i2 = i1 - 1;
```

```
while (temp < colvec[i2])
```

```
colvec[i2 + 1] = colvec[i2--];
```

```
colvec[++i2] = temp;
```

```
}
```

/* In this function, count number of smallest elements is moved from Subset 2 to Subset 1

The variables:

median2 = the second smallest element along the border in Subset 2

pointer2 = points to the column of median2 */

FIND_MIN()

{

int median2, pointer2, i3, temp;

while (count < 0) {

/* find the two smallest elements in Subset 2 along the border */

if (colvec[border[0]] < colvec[border[1]]) {

pointer = 0;

pointer2 = 1;

}

else {

pointer = 1;

pointer2 = 0;

}

median = colvec[border[pointer]];

median2 = colvec[border[pointer2]];

for (i3 = 2; i3 < S; i3++) {

```
temp = colvec[border[i3]];
if (temp < median2) {
    if (temp < median) {
        pointer2 = pointer;
        median2 = median;
        median = temp;
        pointer = i3;
    }
    else {
        median2 = temp;
        pointer2 = i3;
    }
}
```

```
if (++count == 0) {
    border[pointer]++;
    return;
}
```

/* find the second, third, etc., smallest element in the column pointed by pointer until
no more elements are left or the count is equal to zero. */

```
while (colvec[border[pointer]] <= median2) {
    if (++count == 0) {
        median = colvec[border[pointer]++];
        return;
    }
}
```

```
/* the second smallest element along the border is the next smallest in Subset 2 */
```

```
median = median2;
```

```
pointer = pointer2;
```

```
border[pointer]++;
```

```
count++;
```

```
}
```

```
}
```

```
/* In this function, which is similar to function FIND_MIN, count number of largest elements in Subset 1 are found and moved to Subset 2 */
```

```
FIND_MAX()
```

```
{
```

```
int median2, pointer2, i3, temp;
```

```
while (count > 0) {
```

```
if (colvec[border[0] - 1] > colvec[border[1] - 1]) {
```

```
pointer = 0;
```

```
pointer2 = 1;
```

```
}
```

```
else {
```

```
pointer = 1;
```

```
pointer2 = 0;
```

```
}
```

```
median = colvec[border[pointer] - 1];
```

```
median2 = colvec[border[pointer2] - 1];
```

```
for (i3 = 2; i3 < S; i3++) {
```



```
temp = colvec[border[i3] - 1];
if (temp > median2) {
    if (temp > median) {
        pointer2 = pointer;
        median2 = median;
        median = temp;
        pointer = i3;
    }
    else {
        median2 = temp;
        pointer2 = i3;
    }
}

if (--count == 0) {
    border[pointer]--;
    return;
}

while (colvec[--border[pointer] - 1] >= median2) {
    if (--count == 0) {
        median = colvec[--border[pointer]];
        return;
    }
}

median = median2;
pointer = pointer2;
```

```
border[pointer]--;  
count--;
```

/* In this function, the columns of the windows corresponding to the the next row to be processed are found by updating the columns corresponding to the present row of the image.

The variables:

delete == the element to be deleted in a column

delpos == the position occupied by the element delete

insert == the element to be inserted

inspos == the position where the element insert is to be inserted */

INSERT_DELETE()

```
{  
    int column, delpos, inspos, insert, delete;  
  
    for (column = 0, delpos = RP1BY2; column < N; column++, delpos += RP2) {  
        delete = ix[row - RM1BY2][column];  
        insert = ix[row + RP1BY2][column];  
  
        if (delete != insert) {  
            if (colvec[delpos] > delete)  
  
/* searching upwards for deleting position */  
  
                while (colvec[--delpos] != delete)
```

else

/* searching downwards for deleting position */

while (colvec[delpos] != delete)

delpos++;

inspos = delpos;

if (colvec[++inspos] < insert) {

/* searching downwards for inserting position */

do {

colvec[inspos - 1] = colvec[inspos];

} while (colvec[++inspos] < insert);

colvec[--inspos] = insert;

}

else {

/* searching upwards for inserting position */

inspos--;

while (colvec[--inspos] > insert)

colvec[inspos + 1] = colvec[inspos];

colvec[++inspos] = insert;

}