T103
30/8/88

B

TABLE OF CONTENTS

A. K. Adhikari
22.7.88

MAPPING OF IMAGE PROCESSING ALGORITHMS ON AN ADAPTABLE PIPELINE

Student : S. SRIKANTH

Supervisors :

Prof. D. Dutta Majumder

Dr. Anirban Basu

ACKNOWLEDGEMENTS

I take this opportunity to express gratefulness to a number of individuals whose professional encouragement and assistance have made this work a pleasant endeavour.

Deepest appreciation and thanks must go to my guides Prof. Dutta Majumder and Dr. Anirban Basu for the valuable guidance and encouragement which I received from them.

I wish to thank my friends for their help. I owe a special thanks to Mr. B.B. Arunkumar for drawing the figures.

Finally I thank Mr. Subrata Mukherjee for his efficient and neat typing.

CALCUTTA

S. SRIKANTH

CONTENTS

Chapter 1

Introduction :

The size of the images dealt with in computer vision and image processing problems is usually large, the sizes varying from 128 x 128 to 1024 x 1024 pixels. Many real time applications such as target identification in missile defense systems and robot vision require images to be processed at a very high speed. Conventional computers such as VAX-11/780 with a speed of 1 MFLOPS are grossly inadequate for such applications [1], necessitating the use of parallel processing architectures for high speed execution of these algorithms.

Use of a parallel processing systems for image processing requires mapping of algorithms onto the system. Mapping is an important aspect as the speed of execution is mostly determined by the efficiency with the mapping is done.

A parallel processing system which can be used in image processing is the Adaptable Pipeline whose architectural details are given in [2]. It consists of a linear array of cells all of which can be connected to a Host Computer. The cells are made up of commercially available 32-bit Transputer chips [3] which facilitate an easy and cost-effective implementation of the system. In this dissertation it is shown

how image processing algorithms can be mapped onto the above system exploiting the parallelism in the algorithms to the maximum extent possible.

The basic characteristics of image processing algorithms are studied in Chapter 2. In Chapter 3, a critical survey is made of the existing parallel processing architectures for image processing applications. In Chapter 4, the architectural details of Adaptable Pipeline are discussed together with the features which make it suitable for executing image processing algorithms. Finally in Chapter 5 it is shown how image processing applications can be efficiently mapped onto such an Adaptable Pipeline.

Chapter 2

Characteristics Of Low Level Image Processing Algorithms

Image processing can be thought of as consisting of two levels of computational tasks namely low-level and high level [4]. At low-level, tasks such as image filtering and feature extraction are performed and high-level is concerned with evaluation of extracted feature images. At high-level the information content is greatly reduced and hence processing is less computationally demanding.

For many reasons architectures for image processing are presently concerned with only low-level processing. The set of fundamental operations utilised at low-level are more identifiable and are well understood. Also the two-dimensional representation of an image can be easily mapped onto an array easily. Some of the characteristics of low-level image processing algorithms are considered below.

The input images for low-level image processing algorithms are usually large in size and in some applications they have to be processed fast. On the other hand the number of operations on a pixel are usually small in many low-level algorithms. Because of this, there is no need of using complex data structures.

Another characteristic of low-level vision algorithms is that different parts of the image are treated in the same way. This is explained with an example. Suppose we are smoothing an image by taking the average over its neighbourhood. The final value of each pixel will depend only on its neighbourhood pixel values and for each pixel the amount of computation to be performed is same. This implies that low-level computation has low data dependency.

Low-level algorithms can be of two types - local and global. In local vision algorithms each element of the output image depends only on the corresponding element of the input image and elements in its neighbourhood. Examples of such operations are smoothing, sharpening etc. In global operations output can depend on any input element. Examples are Fast Fourier Transform, Histogramming etc.

These characteristics of low-level image processing algorithms dictate certain properties which the system should possess, so that these algorithms can be efficiently executed on it.

At low-level, different image processing applications tend to use similar routines such as Fast Fourier Transform, Convolution, Histogramming etc. So it would be useful if

a library of these commonly used routines is kept.

Also the image sizes may increase and 3D images may be used. Hence it is absolutely necessary that the system be scalable to handle increased computational demands.

The system should be designed in such a way that it can efficiently handle both local and global operations. Needless generality in the system should be curbed as the low-level algorithms require only a set of restricted operations.

Chapter 3

Review Of Architectures For Image Processing

The characteristics of image processing algorithms indicate the need for parallel architectures for image processing The features of some of the architectures for image processing are reviewed here. The following types of parallel processing architectures are discussed in this chapter.

1. Bit-serial parallel processing system ;
2. Systolic architecture ;
3. Pyramidal architecture.

3.1 Bit-serial parallel processing (BSPP) systems :

BSPP systems have a large number of processing elements and each processing element (PE) is very simple, operating on its data stream bit by bit.

Bit-serial processors can handle data item of any length. Suppose we have an array of M operands with N bits per operand. A parallel processor with PEs of standard word length would suffer losses in efficiency when both M is less than the number of PEs and N is loss than the word length of PE. BSPs suffer only one type of inefficiency in the case

when M is less than the number of PEs. Another inherent advantage with BSPs is got when only part of the operand needs treatment.

Some of BSPP systems and the rationale behind their architectural considerations will be discussed here with special emphasis on Massively Parallel Processor (MPP).

3.1.1 Massively Parallel Processor (MPP) :

The MPP has 16896 PEs arranged on a 128 row X 132 column rectangular array [5]. The PEs are in the array unit. The other major blocks in MPP are the array control unit, the staging memory, the program and data management unit, and the interface to a host computer as shown in Fig. (1).

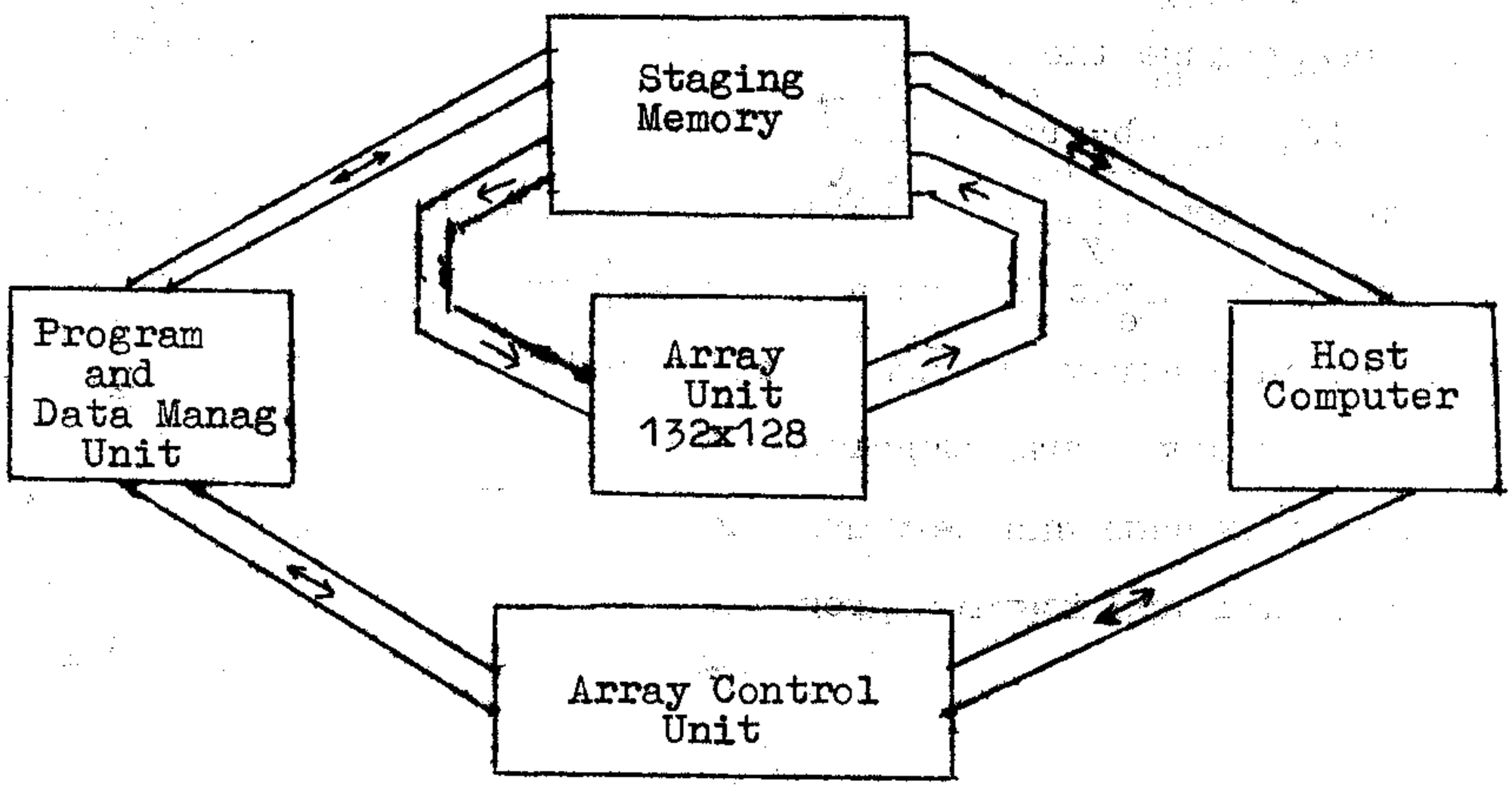


Fig. 1 MPP

Logically, the array unit contains 16384 PEs arranged in a 128 row X 128 column square array. Physically, the array unit contains an extra 128 row X 4 column rectangle of PEs for redundancy. Each PE communicates with its four nearest neighbours north, east, south and west. The square array topology is very useful for local operations. However some image processing algorithms like FFT require communication between pixels located far apart in the image. If one pixel is stored in each PE then the routing time would become prohibitive. However for such operations it would be wise to store several pixels in PE's memory so the number of PEs required to do FFT can be reduced to a small compact sub-array. To avoid idleness of other PEs more than one FFT can be performed at the same time. The routing time thus gets reduced because the longest communication path is only half the width of sub-array. Thus the execution of FFT can be speeded up by using a scrambled layout. Some kind of buffer memory is required in the array unit input-output path to convert data arrays to and from the bit-slice format. Thus a simple square array topology with a buffer memory in its input-output path can perform the permutations required for a particular application program.

3.1.2 CLIP IV :

This was the first bit-serial array processor chip [6]. Data is presented to a two-dimensional array of 96x96 processors. All the parallel data operations, on which the system depends take place here. Local storage of data during processing occurs in a 32-bit RAM. This storage of data during processing is essential for the high speed operation of the array. Connectivity between processors in the array is determined by the required picture tesellation. The CLIP-IV machine allows either square or hexagonal tesellation, under software control. To extract parallelism to the maximum extent possible a special programming language has been developed for handling the operations within the array of processors.

3.1.3 LIPP :

LIPP is a bit-serial parallel processor developed at the Linkoping University, Sweden and is discussed in detail in [7]. Some of the important design aspects which are different from that of MPP and CLIP IV are as follows. Each PE has not only the usual ALU and 1 bit registers but variable length shift registers, a shiftable up/down counter, and an index register for table look up operations. Distributed processor topology is provided, meaning that each processor

deals sequentially with its own sub-image, while in MPP and CLIP-IV all processors are placed side-by-side dealing with neighbourhood pixels. The distributed processor topology has connection scheme where on top of each memory module there is a process.which is connected to eight nearest memory modules.

There are many other bit-serial processors developed like the GRID, NTT, DAP, etc. The architectural details are similar in many respects to the ones discussed above. Of all the chips mentioned only CLIP IV chip allows hexagonal connectivity. While CLIP IV design is biased towards nearest neighbourhood logical operations, MPP offers some logical capability in addition to efficient arithmetic operations. The LIPP architecture is more powerful of all.

In short these machines use a fairly large number of relatively simple PEs which are interconnected in a two-dimensional way. Apart from its 1-bit internal register, each PE has to access its private bit organised memory. There is only one control unit supplying the sequence of PE-instructions and PE-memory addresses. During each instruction cycle all PEs execute the same instruction, using the same PE memory address. Instructions are provided to activate

only those PEs where a program specified condition is satisfied and to deactivate those PEs where this condition is not satisfied. MPP, CLIP IV, LIPP differ in the number of PEs used, PE's interconnection pattern used, PE's complexity, the amount of memory used per PE and the I/O structure used to transfer data between external memories and the PE-memories. Performance results of MPP and CLIP IV are given in [8].

3.1.4 Advantages and Disadvantages of BSPP Systems :

Nearest neighbour access is implicit and hence image processing algorithms involving local operations are done at very high speed on these architectures. The control of each processor is simple and low level, so that computation can b tailored specifically to the application. The bit oriented PEs offer unlimited flexibility of precision and representation of data.

However in such machines because of the large number of PEs involved, the PE architecture needs to be relatively simple. All connections are local and any operation which needs to access neighbourhoods larger than 3x3 are slowed down by inter-PE communication overhead.

In summary bit-serial parallel processing systems improve the performance of some low-level image processing

algorithms particularly those involving local operations but are in general inadequate for doing global operations.

3.2 Systolic Architectures :

The systolic architectural concept was developed at Carnegie-Mellon University and versions of systolic processors are being built [9]. Here one such machine namely the Warp is described together with its applications.

A systolic system consists of a set of interconnected cells, each capable of performing some simple operations. Data in a systolic system flows between cells in a pipelined mode and communication with the outside world occurs only at the boundary cells. Systolic systems provide a realistic model of computation which captures the concepts of pipelining and parallelism. A systolic system can be viewed as a network of processors which rhythmatically compute and pass data through the system. Each processor in a systolic network can be thought of as a heart that pumps multiple streams of data through itself [9]. The crux of the approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell it passes. This is possible for many image processing applications which are compute-bound.

The advantages of systolic structures can thus be summarised as follows. The first and foremost advantage is that multiple use of each input item can be made. This property has many important consequences. Systolic systems can achieve high throughputs with modest I/O bandwidths for outside communication. Many architectures which look fabulous on paper have not met with success because of not giving due importance to I/O problem which is a major bottleneck.

Another advantage of systolic architectures is that it allows concurrent use of many powerful cells rather than sequential use of powerful processors. Concurrency can be obtained by either pipelining, multiprocessing many results in parallel or by both.

To perform well a systolic array needs a large number of cells. The trade-off here might be between using small number of powerful cells or large number of simple cells. A systolic system implemented on a single chip will prefer the former while for board implementation the later is used. Systolic systems also avoid long distance or irregular wires for data communication. For a linear array a global clock parallel to the array is enough for synchronisation no matter whatever might be the length of the array. A systolic array architecture Warp is discussed here.

3.2.1 The Warp Computer :

The Warp machine is a high performance systolic array computer designed for computation - intensive applications. In a typical configuration it consists of a linear systolic array of ten cells, each of which is a 10 MFLOPS programmable processor [1]. The Warp array is an attached processor to a general purpose Host running on Unix operating system.

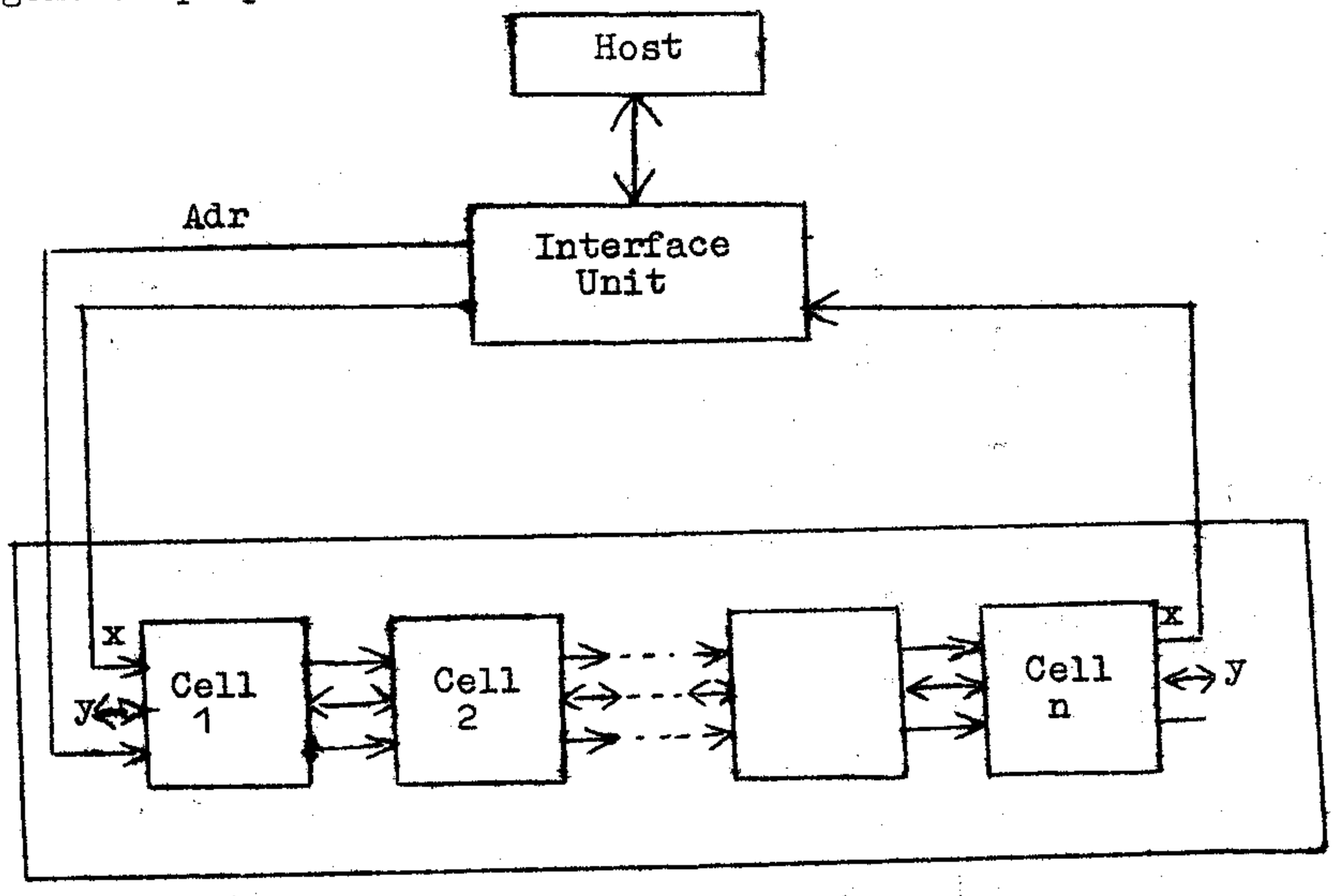


Fig. 2 Warp system overview

The Warp machine has three components : The Warp array, The interface unit and The Host, as shown in Fig. (2). Warp architecture and its relevance to image processing is only

discussed here, further details can be found in [1]. The Warp array performs the computation-intensive routines such as low-level image processing routines. The interface unit handles I/O between Host and Warp array. The Host sends data and receives the results from the array.

Warp is a linear systolic array with identical cells. Each Warp cell is a programmable horizontal microengine with its own micro-sequence and program memory of 8K [1]. The Warp cell consists of 32-bit floating point multiplier, 32-bit floating point adder, two local memory banks, a queue for inter-cell communication and a register to buffer data for each floating point unit.

The Host is a general purpose computer. It is responsible for executing high level routines as well as co-ordinating all the peripherals. The Host has a large memory in which images are stored. The Host performs tasks not suited to Warp array such as those which involve a lot of decision making but little computation.

Warp processor array is a linear array of identical cells. The advantage of using a linear array is, it is easy to implement, easy to extend the number of cells, and it has modest I/O requirement as only two end-cells communicate

with the outside world. However these advantages are outweighed, if the simplicity makes it difficult to program. It is found that many low-level image processing algorithms can be effectively mapped onto a linear array [9].

The I/O BW of Warp is further enhanced by the fact that Warp can transfer 32-bit numbers per each cycle while in most image processing applications the Host deals with 8-bit integers, so that, four integers can be packed and sent in each cycle.

Each Warp cell has a large local memory of 32K to 64K words. This feature is particularly useful for doing global operations where the output depends on a large portion of the input.

Warp is programmed in a language called W2. Programs written in W2 are translated by an optimising compiler into object code for Warp machine. W2 hides low-level details of the machine and allows the user to concentrate on the problem of mapping the algorithm onto the processor array [9].

In summary, Warp is powerful and flexible system because the array is made of powerful, programmable processors with high I/O capability and is capable of executing efficiently both local and global low-level image processing

algorithms. The performance details are given in [1].

3.3 Pyramidal Architectures :

The vision process in human beings takes place across a series of levels begining with retina. Computer vision systems have been structured after this observation. Such systems are called cones or pyramid machines [11]. Hierarchial parallel machine PCLIP is discussod here.

3.3.1 PCLIP :

PCLIP is pyramidal machine that consists of a set of cellular processors and a single controller [11]. The cells are arranged in a pyramid and each cell accepts inputs directly from its thirteen neighbours and computes values that may be stored in local memories or passed to other cells. The cellular operations provided in hardware consist of boolean pattern matching and transferring data from local memory to local registers. PCLIP has three 1-bit registers per cell in addition to local memory of 128 bits per cell. The first is the propagation register, whose value is accesible to neighbours. The second target, register serves both to receive the result of a match instruction and as an additional input to it. The condition register disables its cell when set, causing it to ignore all instructions except a transfer of

data into the condition register. Because of the similarity of PEs many of the bit serial arithmetic techniques used in CLIP IV are used in PCLIP. However, several algorithms seem particular to hierarchial architectures.

One type of algorithm is to create a reduced homomorphic image analyse it and use the results to guide the analysis of the original image. It has been shown in [11] that location of particular feature point can be done in O(log N) time where N is the size of the image. Similarly, point neighbourhood operation can efficiently use both close-neighbouring and more distant information in such tasks as edge detection and region analysis.

Pyramids are built by starting at the lowest level pixel values may be assigned at upper levels by using boolean operations or arithmetic operations such as AVG, MAX, MIN, etc. Certain kinds of feature points can be efficiently located using bottom-up and top-down search. For example, location of maximum intensity in an image is found by building a pyramid by using MAX and tracing the path of maximum element from the root to its original location. A variety of thresholding techniques are simple on pyramid machines. For example, if threshold is chosen as average of ancestors, thus making the threshold adapt to local as well as global intensity variations.

Having reviewed three types of architectures a brief analysis of the advantages and disadvantages of the three types is made. The bit serial processor arrays are superior as far as local operations are concerned. Algorithms involving local operations are fast and ~~easy~~ to implementat on bit serial processor arrays. However, these processors do not have universal appeal as they are very poor in doing global operations. Another problem with bit serial processor arrays is that they suffer from serious bottleneck in I/O. Though interconnection networks allow flexible topology, they have low bandwidth between communicating processors. However, the problem of I/O is solved in CM-2 [12] by providing a number of disk drives which can feed data into various points in the array simultaneously.

The systolic arrays on the other hand do not have I/O problem as they are basically built on the principle that a data item once accessed should be used effectively in the array. Also they can perform both local and global operations reasonably well. Machines of the type PCLIP are becoming very popular. Being bit serial in nature they are very good in doing local operations and at the same time being organised as a pyramid they have very good global properties.

Chapter 4

A Transputer Based Adaptable Pipeline

A pipeline processor performs overlapped computations to exploit temporal parallelism. The concept of pipeline is similar to assembly lines in an industrial plant. If a pipeline system has the following features

1. each pipeline stage is capable of executing any operation ;
2. the system forms pipelines of variable number of stages ;
3. the system provides for fast exchange of temporary results between pipeline stages ;
4. the system can form a variable number of parallel pipelines.

Then the system is called an Adaptable Pipeline [2]. The architectural details of a Transputer based Adaptable Pipeline with special reference to its applicability for I.P. is given here. Mapping of some I.P. algorithms on such architecture is discussed in the next chapter.

4.1 Hardware Organisation Of The Adaptable Pipeline :

The hardware organisation of the Adaptable Pipeline is shown in Fig. (3). Each stage of the Adaptable Pipeline is made of a commercially available Transputer chip. The salient features of Transputer chip are given in [13]. A linear array of 32-bit Transputers (the T414 chip [13]) forms the consecutive stages of the Adaptable Pipeline.

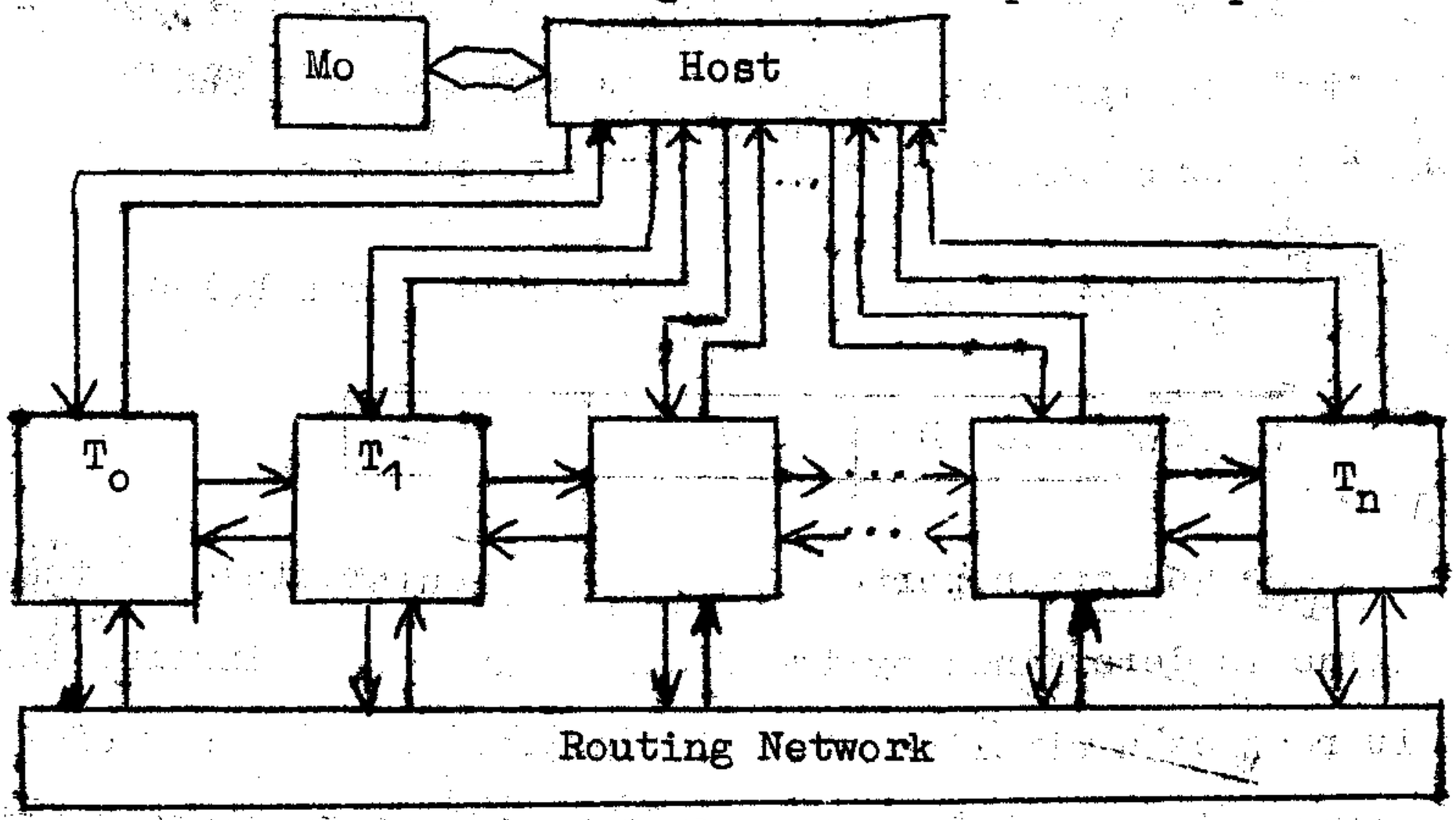


Fig.3 Adaptable Pipeline

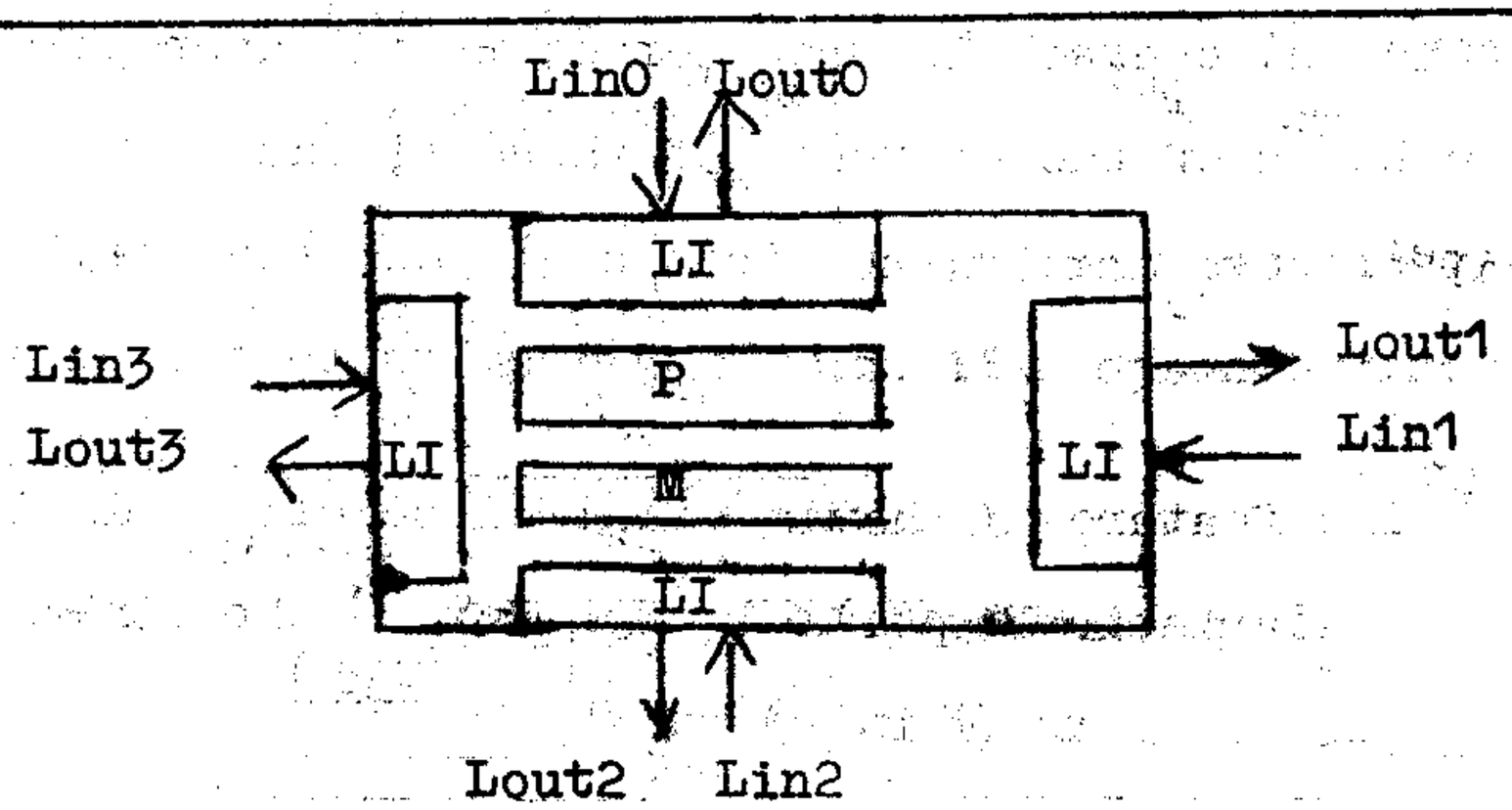
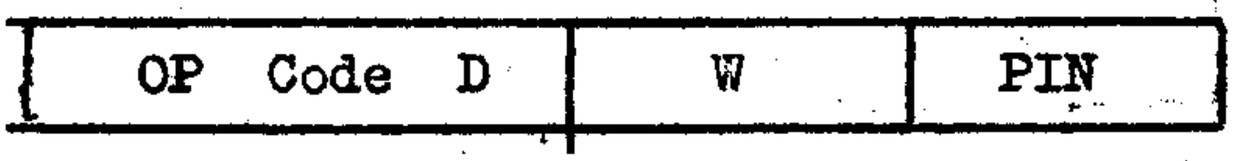


Fig.4 Links of a Transputer

The Transputer array is connected to a Host computer which acts as supervisor and stores data and instructions in memory M_o associated with it. An instruction is initiated for execution from the first stage of pipeline. After being processed at first stage the result and instruction are sent to second stage. The second stage in turn processes it and passes the result to the third stage. The process is continued till the instruction is completely executed and the result is sent back to memory of main computer.

The format of the instruction is as shown below.



The operation performed by a particular stage of the pipeline is determined by two fields D and W. An instruction as it reaches a pipeline stage is interpreted by a set of programs and control is transformed to the appropriate routine on the basis of the contents of these fields. For a computational process partitioned into n stages, the (n+1)-th consecutive stage will send back results to the main memory.

The feature of having parallel pipelines is incorporated in Adaptable Pipeline by connecting the link Lin 0 of all the Transputers (Fig. 4) to the Host computer. Thus any

Transputer can be the first stage of the pipeline. The link LoutO of each Transputer sends results to the Host computer.

The links Lin 1, Lout 1, Lin 2, Lout 2 are used to connect a Transputer stage to its successive and previous stage respectively. Also the last stage is connected to the first stage. Each pipeline stage may send its computational result to any other stage through the routing network to which each Transputer stage is connected through links Lin 2, Lout 2. A Transputer desiring to send a result to stage T_j first sends a code ∂_j to establish connections.

Each pipeline stage T_i uses its 4K local memory for storing data and the programs necessary for the interpretation of the instructions sent. Each Transputer cell can also be connected to an external memory of 4G bytes and is used when the internal memory is not sufficient.

The operation of the adaptable pipeline can be described as follows. Suppose some task is to be performed, it is partitioned into subfunctions for execution in different stages of the pipeline. The Transputer of each stage is pre-loaded with the necessary software for the purpose. It contains all the programs necessary for the execution of the subfunctions of different computational processes that may

be assigned to it to perform on the basis of D and W of the instruction.Each stage selects the program needed for performing a subfunction.

Transputer architecture can be fully utilised if it is programmed in OCCAM language [14]. OCCAM is based on concepts of processes and channels and the software for the whole system can be designed in terms of these. OCCAM has the advantage that concurrent processes are automatically synchronised.

4.2 Use Of The Adaptable Pipeline In Image Processing :

The adaptability of Adaptable Pipeline is discussed here for image processing applications. The architecture of Adaptable Pipeline is flexible enough to be used for many image processing algorithms. For example consider local operations. Here some simple operations are performed on pixel values and its new value depends on its old value and the values of the neighbourhood pixels. In such cases the time taken is more because the size of the images are large. An effective way of solving such algorithms would be by dividing the input image among each Transputer stage so that faster execution on smaller images is possible. Similarly global operations can be easily executed in Adaptable Pipeline

since the large external memory attached to each Transputer stage can be used to store the input image and produce part of the output. Another advantage of having large memory is that most of the standard low-level image processing algorithms can be stored and used under instruction from the Host computer.

The Adaptable Pipeline is easily implementable and is cost-effective as each stage is made up of commercially available Transputer chips. This also makes the system scalable. A system is said to be scalable if its computational capacity can be easily increased without major changes in design.

Each Transputer can communicate with the Host at the BW of 10 M bits/sec. Though this rate is high enough for most applications it is desirable to have a higher BW. If the Host can communicate with Transputer stages concurrently the present BW would be enough for most applications.

In Adaptable Pipeline, as any number of pipelines with varying number of stages can be formed a high amount of efficiency can be obtained in cases where the size of the image is less, by working on more than one image on different pipeline stages.

Finally, though an Adaptable Pipeline is a linear array of Transputer cells it is possible to have multi-dimensional arrays by using routing networks. This is discussed in more details in the next chapter.

Thus it is clear that Adaptable Pipeline can be used with advantage for image processing applications. In the next chapter mapping of image processing algorithms onto the system is discussed.

Chapter 5

Mapping Of Image Processing Algorithms On An Adaptable Pipeline

The power and efficacy of an Adaptable Pipeline for performing low-level image processing algorithms was discussed in the last chapter. In this chapter it is shown how some of the common image processing algorithms can be efficiently implemented on Adaptable Pipeline exploiting the parallelism in the algorithms. All the algorithms discussed here are implemented in OCCAM language.

There are three mapping schemes which are discussed here. They are,

i) Input partitioning
ii) Output partitioning
iii) Pipelining.

All the above partitioning methods are discussed with examples.

5.1 Input Partitioning :

In input partitioning technique the input data is partitioned among each stage and each stage produces output corresponding to this input set. As the size of the input

images is usually large so that much parallelism can be obtained by using the above partitioning technique. Input partitioning ideally suits local operations because output pixel value depends only on a few nearby pixels in the input image.

Parallelism obtained by partitioning the input data among different stages is to an extent offset by three sources of overhead in computation time. They are,

i) Cost of dividing image into parts
ii) Cost of keeping track of these partitions
iii) Cost of combining outputs from each stage.

However these costs are negligible. In most cases the cost of partitioning input is cost of distributing the data among each stage. As each stage of the pipeline is connected to a Host this cost will be negligible. No special book keeping is required as the PIN field of each instruction in Adaptable Pipeline takes care of this. The cost of combining the output is merely the cost of concatenating the output which can be done in the Host as the output from each stage is sent back to the Host after completion of computation.

The speed-up obtained is n, where n is the number of stages in the Adaptable Pipeline (if the overhead costs are

neglected). There are two particular characteristics of Adaptable Pipeline which increase the speed of execution. As each stage is connected to the Host, data can be transferred between Host and each Transputer stage concurrently. As an Adaptable Pipeline can be configured into more than one pipelines, more than one input image can be processed simultaneously. The decision as to when to use a single pipeline or a number of pipelines simultaneously is taken by the Host considering the size of the image and the number of Transputer stages. If the input image size is less then, it would be more appropriate to do more than one computation as input partitioning will result in very small images in each stage. For very large images it would be more beneficial to use the entire pipeline for a single computation.

In many cases the input image need not be partitioned into strictly disjoint data segments. It would be appropriate to keep the border pixels of partitioned data segments in neighbouring Transputer stages. For example suppose we have a 6x6 input image and four Transputer stages. We wish to smoothen the image by replacing each pixel value by the average of its neighbours. Then instead of dividing the image into four 3x3 images it would be better to divide them as four 4x4 images. This would reduce inter-stage communication since if we take 3x3 partition, to compute the new value of border

pixels, some pixel values have to be accessed from adjacent cells. Some algorithms will be discussed now to show how input partitioning technique can be used for their efficient execution on an Adaptable Pipeline.

5.1.1 Median Filtering :

A powerful smoothing technique that does not blur edges is the median filtering, in which the value at a point is replaced by the median of the values in the neighbourhood of the point [15]. Expressed mathematically,

$$f(x,y) = \text{Median}[f(x+1,y), f(x,y+1), f(x-1,y), f(x,y-1)].$$

Median can be found by sorting the values of all neighbourhood pixels and taking the middle value. As a result of this median filtering is done iteratively. The data need be sent only once no matter how many iterations are used.

The input image is first partitioned among various stages of Transputer and median filtering is executed on each sub-image separately. The outputs from each stage when concatenated would give the final filtered image.

5.1.2 Sharpening :

Smoothening would lead to blurring of images and an effective way of counter-acting this is by sharpening the

image. One method of doing it is by applying Laplacian operator given by the equation

$$\nabla^2 f(i,j) = [f(i+1,j)+f(i-1,j)+f(i,j+1)+f(i,j-1)]-4f(i,j)$$

This algorithm implementation in similar to that of median filtering. The image is partitioned and Laplacian operator applied to each pixel of sub-image. The output of all the stages is mixed in the Host to get the sharpened version of the input image.

5.2 Output Partitioning Technique :

In this technique each Transputer stage processes the entire input image and produces part of the output. This partitioning technique can be applied for those image processing applications where the value of each output depends on the entire input data set. Thus the whole of the input data set is given to each Transputer stage and it produces part of the output. The outputs from each Transputer are then transferred back to the Host computer where it is combined to get the final output result. There is no duplication of data-structures in this partitioning technique.

The efficiency of this method will be maximum if each cell can use each data item. Some algorithms which use output partitioning are discussed now.

5.2.1 Histogramming :

Histogram gives a statistical appraisal of the image under consideration. There are many other techniques which manipulate the histogram to enhence the quality of the image. Hence histogramming is a common operation performed on images.

Histogram H(g) of a image f(x,y) is defined as

$$H(g) = n_g \qquad 0 \leq g \leq g_{max}$$

where g_{max} is the gray level resolution and n_g is the number of pixels having gray level g.

Histogramming can be implemented on Adaptable Pipeline by dividing the gray level space among each Transputer stage. If there are n Transputer stages and g gray levels each stage will get nearly g/n gray levels. So, each Transputer stage will form its part of the histogram. All the n histogram can then be concatenated in the main computer to get the final histogram.

5.2.2 Hough Transform :

In Hough transform curve and line detection involves applying a co-ordinate transformation to the picture such that all points belonging to a curve of a given type map into a single location in the transformed space [15]. This can be

illustrated by an example. Suppose it required to detect straight line passing through a point P (which is origin). If a one-dimensional array is constructed in which the value at position W is number of points (P_is) such that $y_i/x_i = W$. Then if we find a peak in the array at W = m, then we can conclude that most of P_is lie on straight line with slope m.

A more general method of line finding is discussed here. For examples lines may be parameterized by values θ and p and the equation is

$$x \cos \theta + y \sin \theta = p$$

Thus for line finding the Hough transform takes the (x,y) location of each significant pixel and for a range of θ calculates the p value using the above formula. The value of (θ,p) is incremented. Once the whole of data is processed, the table is scanned and peaks are found. These peaks correspond to most likely lines in the image. As the most time consuming activity is the mapping between image and parameter space, output partitioning model suits this algorithm the most. The parameterized space is distributed among the various cells. To increase efficiency the Host should send only those pixels which are significant.

After all the data has been processed in Transputer stages each stage selects its significant peaks and sends

them to the Host where the maxima can be selected. Many other global operators such as image rotation etc., can be similarly implemented on Adaptable Pipeline.

5.3 Pipelining :

In pipelining the algorithm is partitioned among the different Transputer stages and each stage performs one stage of processing. Any algorithm which is regular and can be partitioned among various stages can be implemented in this way. Adaptable Pipeline is basically a pipeline and hence implementation of this model is straight forward. However it should not be assumed that this is the only efficient method because it has already been shown how other methods can be efficiently implemented on Adaptable Pipeline. One advantage with pipeline model is that there is no need for spliting or concatenating of input or output data as is done in other methods. Two important algorithms which are most frequently used in image processing are discussed below.

5.3.1 Convolution :

Here one-dimensional convolution is discussed. It is shown in [10] that any higher dimensional convolution can be done by using one-dimensional convolution. The problem is given a sequence of weights $(w_1, w_2, \ldots, w_k)$ and an input

sequence($x_1, x_2, \ldots, x_n$) to evaluate the sequence

$$y_i = W_1 x_i + W_2 x_{i+1} + \cdots + W_k x_{i+k-1} .$$

There are many techniques of doing convolution one such method is discussed here.

Initially all weights are broadcasted to each stage (assuming $k \leq$ number of Transputer stages). At the begining of cycle x_i is broadcasted to each stage and one y_i enters the first stage. After k+1 cycles the final values of y_is are produced one from each cycle.

5.3.2 Fast Fourier Transform :

The problem here is to find

$y_i = x_o w^{i(n-1)} + x_1 w^{i(n-2)} + \cdots + x_{n-1}$ $0 \leq i \leq n-1$ given input sequence $x_o, x_1, \ldots, x_{n-1}$. This is called Discrete Fourier Transform. This method would take $O(n^2)$ time. However the Fast Fourier Transform of Cooley and Tukey [1965] takes only O(n log n) time. There are many variations of FFT, the constant geometry version [16] is implemented here. The most time consuming task is the butterfly operation which is described as below.

$$(a_r + ja_i) \pm (b_r \pm jb_i)\ (w_r + jw_i)$$

This involves six real additions and four real additions. FFT can be efficiently implemented on the Adaptable Pipeline as follows. All the butterfly operations in the i-th stage are carried by stage i and the results are stored in the memory of stage i+1. As communication and processing can occur concurrently, the (i+1)th stage can work on butterfly operation of another FFT problem. Thus it is necessary that there should be a constant flow of FFT problems (as in many practical cases) for all the stage to be busy.

Another mode of operation is possible in Adaptable Pipeline where the routing network is used to form multi-dimensional arrays. Though an Adaptable Pipeline has a linear array of Transputer stages it is possible to have a multi-dimensional array. The advantage with such a configuration is that some algorithms like higher dimensional convolution can be mapped very efficiently.

For this, interconnection between various cells has to be made by designing a suitable interconnection network. For example if we want a 3x3 square mesh of Transputer stages the first stage should be connected to fourth, the second to fifth and so on in addition to connections between adjacent stages which already exists. A thorough analysis of the practicability of the above method has to be made before it is used.

CONCLUSION :

The use of parallel processing systems in image processing applications is never doubted, but opinions still differ on the issue of selecting the most appropriate system.

In this dissertation it is shown how Adaptable Pipeline can be used for image processing applications. Adaptable Pipeline has a highly flexible architecture which makes it suitable for many types of low-level image processing applications. Though the Transputer chip, which is used in the Adaptable Pipeline system described here, is a powerful chip there is a mismatch between its computational and communication capabilities. This problem can be alleviated to an extent if the chip uses parallel communication. If such a chip is fabricated then the Adaptable Pipeline would become all the more powerful for image processing applications.

REFERENCES :

1. Annartone M et al. 'The Warp Computer : Architecture, Implementation and Performance', IEEE Trans. on Computers, Dec. 1987, pp. 1523-1537.

2. Basu A 'A Transputer Based Adaptable Pipeline', Proc. Second International Conference on Supercomputing, May 1987, pp. 450-459.

3. C. Whitby - Strevens, 'The Transputer', Proc. 12th Annual International Symposium on Computer Architecture, 1985, pp. 292-300.

4. Edwards M.D in 'Image Processing System Architecture', pp. 85-99, Academic Press.

5. Batcher K.E 'Bit Serial Parallel Processing Systems', IEEE Trans. on Computers, 1982.

6. Fountain T.J 'Computing Structures For Image Processing', pp. 1-13, Academic Press.

7. Dannielsson P.E and Ercisson T.S in 'Computing Structures For Image Processing', pp. 157-178, Academic Press.

8. Gerritsen F.A in 'Computing Structures For Image Processing', pp. 15-30, Academic Press.

9. Kung H.T 'Why Systolic Architectures', IEEE Trans. on Computers, Jan. 1982, pp. 37-46.

10. Kung H.T 'Systolic Algorithms For The CMU Warp Processor'.

11. Tanimoto S.L in 'Multi-Computers and Image Processing', pp. 421-431.

12. Thinking Machines Corporation, CM-2 Technical Summary HA 87-4, Thinking Machines Corp., Apr. 1987.

13. INMOS Ltd., 'Transputer Data Sheet'.

14. INMOS Ltd., 'OCCAM Programming Marual', Prentice Hall International, 1984.

15. Rosenfeld A and Kak A.C
'Digital Picture Processing'
Academic Press, 1982.

16. Rabiner L.R and Gold B
'Theory and Application of Digital Signal Processing'
Prentice Hall, 1975.

T106
30/8/88

IMAGE PROCESSING ALGORITHMS

IMPLEMENTED IN

O C C A M

S. SRIKANTH

T106
30/8/88

IMAGE PROCESSING ALGORITHMS

IMPLEMENTED IN

OCCAM

S. SRIKANTH

```
...   This program performs image enhancement through sharpenening.
...   Sharpening is done by applying the Laplacian operator.
...   Input partitioning technique is used.
...   The  input image is partitioned into different  cells.  Laplacian
...        operator is applied to each pixel of the input image.
...   The output from each stage is sent back to the Host.


VAL total_processors IS 8 :
[2 * max.processors] CHAN (INT) pipe :
[512][512] BYTE image  :
INT temp.row, temp.col, row, col, partition  :


...   The  input image is not divided into perfectly disjoint  sets  but
...        some   redundancy   is   provided   to   avoid   inter-processor
...        communication. This would result in outer cells getting different
...        no. of pixels compared to the inner cells. This is taken care in
...        the following lines.

PAR present.processor = 1 FOR max.processors
  PROC find.laplacian( VAL INT row, col  ,INT laplacian)
     INT temp.col, temp.row, temp  :
     [8] INT temp.array  :
     SEQ
        SEQ
              sum  :=  image[row-1,col]+image[row+1,col]+image[row,col-1]+ image[row,col+1]
              laplacian := sum - (4*image(row,col))
  :
     in.link is pipe[2 * present.processor - 2 ]  :
     out.link IS pipe[2 * present.processor - 1]  :
          SEQ
               in.link ? partiton
               IF
                    present.processor = 1
                         SEQ
                              SEQ row = 1 FOR partition + 1
                                   SEQ col = 1 FOR partition+1
                                        in.link ? image[row,col]
                    present.processor = max.processor
                         SEQ
                              temp.row := 0
                              SEQ row = (present.processor-1)*partition FOR partition
                                   SEQ
                                        temp.col = 0

                                   temp.row = temp.row +1
                                   SEQ col= (present.processor-1)*partition FOR partition
                                        SEQ
                                             temp.col = temp.col +1
                                             in.link ? image[temprow,tempcol]
               TRUE
                    SEQ
                         temp.row := 0
                         SEQ row = (present.processor-1)*partition FOR partition
                              SEQ
                                   temp.col = 0
                                   temp.row = temp.row +1
                                   SEQ    col=    (present.processor-1)*partition   FOR  partition
                                        SEQ
                                             temp.col = temp.col +1
                                             in.link ? image[temprow,tempcol]
          SEQ temp.row = 1 FOR partition
              SEQ temp.col = 1 FOR partition
                   SEQ
                     find.laplacian(temp.row,temp.col,laplacian)
```

```
                        image(temprow,tempcol) = laplacian
                SEQ row = 1 FOR partition
                    SEQ col = 1 FOR partition
                       output ! image [row][col]
    STOP



...  This program evaluates the histogram of the input image.
...  The partitioning technique used is output partitioning.
...  The input image is passed to all the stages.
...  Each stage computes part of the histogram.
...  Each  stage  sends  its  part of the histogram to  the  Host  after
...       computation  is over.


VAL max.processors IS 16  :
PAR processor.no = 1 FOR max.processrs
    [512][512] BYTE image  :
    INT  no.of.grey.levels, max.rows, total.grey.levels  :
    SEQ
         input ? max.rows
         input ? total.grey.levels
         ...   Image is inputted to each stage.
         ...   Image is taken as a square matrix.

         SEQ  row = 1 FOR max.rows
              SEQ col = 1 FOR max.rows
                   input ? image[row][col]
         no.of.grey.levels := total .grey.levels / max.processors
         [histogram FROM  (processor.no-1)*no.of.grey.levels + 1  FOR no.of.grey.levels] IS [value FRO
              SEQ

                   ...  For   its  range  of  the  output   each   stage
                   ...    calculates  the no.  of grey values in  variable
                   ...    count.

                   SEQ loop = 1 FOR no.of.grey.levels
                        count[loop] := 0
                   SEQ row = 1 FOR max.rows
                        SEQ col = 1 FOR max.rows
                             count[image[row][col]] = count[image[row][col]+1
                    SEQ  row = (processor.no - 1) * no.of.grey.levels -1 FOR no.of.grey.levels
                        output ! count[row]
STOP                    [60] CHAN(INT) pipe:



...   This program performs one-dimensional convolution.
...   The program uses pipeline mode of execution.
...   All the weights are stored one per each stage.
...   It is assumed that the no. of weights is less than the size of the
...      array.
...   Each  input element is broadcasted to all cells and the sum of the
...      partial prodouct is accumalated.
...   In short the weights stay and the output moves.

[20] INT x :
INT  weight, max.elements, no.of.elements  :
INT  temp,loop,partial.sum, count  :
VAL  max.processors IS 8 :
  PAR processor.no =
```

```
pipe.input IS pipe[4*processor.no-1];
pipe.output IS pipe[4*processor.no - 2] :
output IS pipe[4*processor.no - 3] :
input IS pipe[4*processor.no];
SEQ  ... Here the no. of weights are broadcasted to all cells.
     input ? weight
     input ? no.of.elements
     input ? max.elements
     IF
          ...    If  the  stage  under consideration is  the  last
          ...       stage, then as many as K-N+1 no.of outputs are
          ...       produced where K is the no. of weights and N is
          ...       the no. of inputs.

          processor.no = max.processors
               count := 1
          TRUE
               count:= no.of.elements - max.procesors +1
IF

          ...   If the stage under consideration is not the last stage
          ...      then each stage gives one output value.

          processor.no = 1
               SKIP
     TRUE
          SEQ loop = 1 FOR processor.no-1
               input ? x[loop]
temp = procesor.no
WHILE temp <= max.elements
     SEQ
          IF
               processor.no = 1
                    partial.sum := 0
               TRUE
                    pipe.input ? partial.sum
              partial.sum := partial.sum + (weight * x[loop])

          ... The outputs are then send to the main host.

             IF
               processor.no = max.processors
                    SEQ
                         output ! count
                         output ! partial.sum
                         count := count + 1
               TRUE
                    IF
                         temp = max.elements
                         SEQ
                              output ! count
                              output ! partial.sum
                    TRUE
                         pipe.output ! partial.sum
STOP
```

This program is for Hough Transform.
In this output partitioning is used.
The program is used for extracting information regarding straight

```
...       lines.
...   The equation of the straight line is taken in the parametric form.
...   For  each  significant  pixel  the Host sends the value  of  p  is
...        caluclated for a range of theta values.
...   For each theta , rho pair the output is incremented.
...   The peaks are found in each stage and sent to the Host. The maximum
...       amongst these is found there.


[32][32] INT  value  :
INT  rho.range, theta.range, theta.max, max.processors  :
[512][512] BYTE image  :
INT  theta,rho, rho.max, max, no.of.rows, row, col, x, y  :
SEQ
     input ? max.processors
     [2*max.processors] CHAN(INT) pipe :
     PAR  processor.no = 1 FOR max.processors
          input  IS pipe [2*(processor.no-1)] :
          output IS pipe [2*(processor.no-1)+1] :
          SEQ
               input ? no.of.rows
               input ? theta.max
               input ? max.processors
               theta.range = theta.max/max.processors
               [count FROM (processor.no-1)*theta.range + 1 FOR theta.range] IS [value FROM 1 FOR theta
          SEQ
             SEQ row = 1 FOR theta.range
                  SEQ col  = 1 FOR rho.range
                       value[theta][row] := 0
             x  := 1
             WHILE x <> 0
                  SEQ
                       input ?  x
                       input ?  y
                       SEQ  theta := (processor.no - 1 ) * theta.range + 1 FOR theta.range
                            SEQ
                                 rho := ( x * cos(theta) + y * sin(theta) )
                                 value[theta][rho] := value[theta][rho] + 1

             ... To find the peak.

             max := 0
             max.rho := 0
             max.theta := 0
               SEQ  theta = (processor.no-1)*theta.range + 1 FOR theta.range
                  SEQ   rho := 1 FOR rho.max
                        IF
                             max <= value[theta][rho]
                                  SEQ
                                       max  := value[theta][rho]
                                       max.rho := rho
                                       max.theta := theta
                             TRUE
                                  SKIP
             output ! max.theta
             output ! max.rho
             output ! max
STOP
```

```
...   This program performs image enhancement by sharpening.
...   Median filtering is performed.
...   The input image is partitioned into different cells.
...   The  value  of  each  pixel  is replaced  by  the  median  of  the
...        neighborhood pixels.


VAL total_processors IS 16 :
[2 * max.processors] CHAN (INT) pipe :
[512][512] BYTE image  :
INT temp.row, temp.col , row, col , partition  :

PAR present.processor = 1 FOR max.processors

  ...   The following procedure caluclates the median of the neighborhood
  ...        pixels of the given pixels.


  PROC find.median( VAL INT pixelrow, pixelcol ,INT median)
     INT temp.col , temp.row, sorti, sortj, temp :
     [8] INT temp.array  :
     SEQ
          SEQ temp.col = pixel.col-1 FOR pixel.col + 1
               temp.array[temp.col-pixel.col+2] = image[pixel.row-1,temp.col]
          temp.array[4] = image[pixel.row,pixel.col-1]
          temp.array[5] = image[pixel.row,pixel.col+1]
          SEQ temp.col = pixel.col-1 FOR pixel.col+1
               temp.array[temp.col-pixel.col+7] = image[pixel.row+1,temp.col]
          SEQ  sorti = 1 FOR 8
             SEQ sortj = sorti+1 FOR 8
                 IF
                    temp.array[sorti] <= temp.array[sortj]
                         SEQ
                              temp = temp.array[sorti]
                              temp.array[sorti]= temparray[sortj]
                              temp.array[sortj]=temp
                    TRUE
                         SKIP
          median = (temp.array[4]+temp.array[5])/2
:
  in.link is pipe[2 * present.processor - 2 ]  :
  out.link IS pipe[2 * present.processor - 1]  :
          SEQ
               in.link ? partiton
               IF
                    present.processor = 1
                    SEQ
                         SEQ row = 1 FOR partition + 1
                              SEQ col  = 1 FOR partition+1
                                   in.link ? image[row,col]
               present.processor = max.processor
                    SEQ
                         temp.row := 0
                         SEQ row = (present.processor-1)*partition FOR partition
                              SEQ
                                   temp.col  = 0
                                   temp.row = temp.row +1
                                   SEQ col= (present.processor-1)*partition FOR partition
                                        SEQ
                                             temp.col  = temp.col  +1
                                             in.link ? image[temprow,tempcol]
               TRUE
                    SEQ
                         temp.row := 0
                         SEQ row =  (present.processor-1)*partition FOR partition
                              SEQ
                                   temp.col  = 0
                                   temp.row = temp.row +1
                                   SEQ    col=    (present.processor-1)*partition  FOR  partition
                                        SEQ
                                             temp.col = temp.col +1
```

```
                                    in.link ? image[temprow,tempcol]
SEQ temp.row = 1 FOR partition
    SEQ temp.col = 1 FOR partition
        SEQ
            find.median(temp.row,temp.col,median)
            image(temprow,tempcol) = median
SEQ row = 1 FOR partition
    SEQ col = 1 FOR partition
       output ! image [row][col]
```