MULTI-TRANSPUTER IMPLEMENTATIONS OF SELECTED IMAGE PROCESSING TECHNIQUES

by

Christopher J.Turner
Virendra C. Bhavsar
Przemyslaw R. Pochec

TR93-078 Aug 1993

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

Multi-Transputer Implementations of Selected Image Processing Techniques

Christopher J. Turner, Virendra C. Bhavsar, Przemyslaw R. Pochec

Faculty of Computer Science

University of New Brunswick

Fredericton, N.B., E3B 5A3, Canada

Ph. +1(506)453-4566, Fax +1(506)453-3566

email:bhavsar@unb.ca, pochec@unb.ca

Abstract

We present parallel implementations of a local (convolution) and a global (regular moment extraction) image processing technique on a multi-transputer system. Issues relevant to implementation design including computational algorithm selection, initial data pass, and topology selection are discussed. Linear speedups in the convolution implementations are observed for all image sizes whereas such speedups are obtained only for large image sizes for the regular moment implementations. Analysis of the implementations including parallel time complexity functions and observations about data passing and topology selection is given. Two theoretical performance models based on the implementations closely match empirical timing results.

Keywords: parallel processing, transputers, image processing, convolution, moments, performance evaluation.

1 Introduction

Real-time image processing applications require operations to be executed on the thousands of pixels that make up image frames many times each second. The feasibility of transputer based high speed image processing systems has been discussed in [10, 13]. In addition, various commercial systems have been developed. For example, TRANSTECH has marketed an image processing TRAM and an associated library of image processing routines [8]. However, the literature published does not present many details about developing *efficient* transputer

based image processing systems. This paper explores this issue using two image processing techniques.

We have developed parallel implementations of the image processing techniques of convolution and regular moments on an IMS B008 motherboard [9] containing nine IMS T800 transputers programmed with Parallel C. The programs were developed using the ANSI C Toolset [7]. Our main criterion for judging the efficiency of the implementations is execution time. Memory requirements are also considered. Special attention is given to implementation performance as image size and number of transputers increase.

The structure of the paper is as follows. Section 2 reviews the image processing algorithms, *viz.* convolution and regular moments, that we have chosen and presents their single transputer implementations. Section 3 gives our eight multi-transputer implementations. Performance modelling of two of these implementations is presented in Section 4. Conclusions of our work are presented in the last section. For the sake of brevity, we have simply stated some of the results of our work. For a full description of our research, the reader should consult [15].

2 Algorithms and Sequential Implementations

In this section we introduce the image processing algorithms selected in this paper and give timing results for their single transputer implementations. The single transputer programs have been compiled and executed on the root T800 transputer of the B008 motherboard. All single transputer programs have been implemented as Parallel C programs which use none of the parallel features of the language. Compilation was accomplished using the *icconf* command and a configuration source file. Alternatively, transputer programs can be compiled without a configuration source file by using the single transputer option of the *icollect* command. The latter method produces significantly faster running programs. A subsection devoted to convolution and its single transputer implementations is next given. Subsequently, timing results and analysis for the eight regular moment programs are presented.

2.1 Convolution

Convolution is a local feature extraction process. Its implementation involves passing a mask or filter over an image. Some image features extractable using convolution are lines, edges and points. Two commonly used masks for edge point detection are the Laplacian and the Sobel [4, pp. 337-338]. The convolution of two discrete two dimensional functions f and g is defined to be

$$f_e(x,y) * g_e(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f_e(m,n) g_e(x-m, y-n) \tag{1}$$

for $x = 0, 1, \ldots, M-1$ and $y = 0, 1, \ldots, N-1$, where f_e and g_e are extended versions of the functions f and g, with additional zeros added outside the domain of the functions so that the convolution operation can be performed [4, p.88].

The convolution algorithm requires $O(M^2N^2)$ multiplications and additions to convolve an $N \times N$ image with an $M \times M$ mask. The mask is passed over each image pixel. The value of the convolved pixel is set to be the sum of the products of each mask element and the pixels beneath it. It is assumed that $\lfloor (M/2) \rfloor$ zero pixels have been added around the edge of the image. For real-time applications, the procedure may be computationally too intensive for a conventional single processor computer. Our approach to the problem is to spread the computation required over many processors.

We have implemented the convolution algorithm on a single transputer using Parallel C. Both the image and mask are acquired from files on the host. The image pixels are stored as single byte Parallel C *char* data types. The mask elements are stored as *float* data types. In Parallel C these are implemented as 32 bit IEEE floating point numbers. To maintain full flexibility for convolving an arbitrary sized image with an arbitrary sized mask, the mask and image are stored in dynamically allocated buffers rather than in static Parallel C arrays. Offsets into these buffers are maintained in order to reduce the time needed to reference elements.

Due to the large number of multiplications and additions needed, the convolution algorithm runs relatively slowly. The program has been run on images with dimensions equal to powers of two from 64 to 1024 for four different sized real masks. Table 1 gives timing results for the program when run on a 256×256 image for those mask sizes. The times given are for computation only. Figure 1 plots execution times of the program for various mask

sizes versus the number of pixels. The execution times increase linearly with the number of image pixels (quadratically with image size N). We have also found that the times increase slightly less than linearly with the number of mask elements. This is due to the fact that the number of additions required to maintain the buffer offsets increases less than linearly with the number of mask elements. We have modified the convolution program to accept integer masks. Not surprisingly, the execution times are faster than those achieved with floating point masks, since integer arithmetic operations are faster than floating point ones.

2.2 Moments

Statistical moments are widely used in image analysis. They are among the most significant of global image features. Many important geometric attributes of an image can be determined from its moments. Among these are its mass, spread and centre of inertia. Papoulis' uniqueness theorem [12] has established that no two images can have the same set of moments. This property ensures that moments are effective as a means of distinguishing among patterns. In addition, object features called moment invariants, which do not change if a geometric operation is performed on the object, can be derived from moments. Moments are therefore a valuable tool for image analysis. The formula for calculating m_{KL}, the (K, L)th regular moment of a discrete two dimensional function $f(i, j), i = 1, 2, \ldots, N, j = 1, 2, \ldots, N$ is

$$m_{KL} = \sum_{i=1}^{N} \sum_{j=1}^{N} i^K j^L f(i, j) \tag{2}$$

From Eq. (2) it can be seen that moment calculation also requires a large number of multiplications and additions. There are eight algorithms for regular moment calculation used in our work. The most obvious way to calculate moments is the straightforward algorithm [3] which uses nested do loops based on Eq. (2). Because $2N^2$ exponentiations are needed to compute each moment, this algorithm is quite inefficient. Notice that the column coefficient monomials j^l can be calculated using j^{l-1}. In particular,

$$j^l = j j^{l-1}. \tag{3}$$

Reeves' coefficient storage algorithm [14] exploits this fact. The idea here is to create four

$N \times N$ arrays $C(l), l = 0, 1, \ldots, 3$ of column coefficient monomials using Eq. (3). These are then used to calculate the moments.

One drawback of the coefficient storage algorithm is that the products of image elements and coefficients are repeatedly calculated in the final stage of the algorithm. We can solve this problem by incorporating the image values in the four coefficient arrays. We call the resulting algorithm column product storage.

There are two main problems with the storage algorithms. First, to process an $N \times N$ image, the storage arrays require $8N^2$ bytes - a great deal of memory for large values of N. In addition, $O(N^2)$ multiplications would be needed to execute the algorithms on such an image. Multiplications are relatively computationally expensive for many processors.

The double summation in Eq. (2) can be split into two separate summations. Eq. (4) gives the resulting equation for the calculation of m_{KL}.

$$m_{KL} = \sum_{i=1}^{N} i^K \sum_{j=1}^{N} j^L f(i,j) = \sum_{i=1}^{N} i^K RM_{Li}, \tag{4}$$

where

$$RM_{Li} = \sum_{j=1}^{N} j^L f(i,j).$$

The calculation of the (K, L)th moment is thus split into two steps. First, the Lth one dimensional moment RM_{Li} of each image row is calculated. These row moments are then used to calculate the (K, L)th overall two dimensional moment m_{KL}. This is not unlike using the one dimensional Fourier transform to calculate the transform of a two dimensional function. The next three algorithms all use this technique to solve the problems associated with the storage algorithms.

We have created an algorithm similar to the storage algorithms that uses the above two step moment calculation technique. We call it the add and multiply algorithm because addition and multiplication alternate in the process of moment calculation. Figure 2 shows the add and multiply algorithm for the calculation of the one dimensional moments of an $N_1 \times N_2$ image F. Although it has relatively small memory requirements, this algorithm still requires $N^2 + N$ multiplications to calculate each moment of an $N \times N$ image. It would be preferable that an algorithm use only additions to calculate moments. Budrikis and Hatamian have proposed an algorithm for regular moment calculation that does so [1, 5]. The algorithm uses recurrences to build up higher order moments from lower ones. Figure 3

gives pseudo code for a generalized version of the one dimensional recursive algorithm. STM contains the initial values of the moments. SPM contains the starting values of the previous moments (the values that the moments had after the previous execution of the inner loop of the algorithm). M and PRM will contain the values of the moments and the previous moments of F upon completion of the algorithm. To calculate the first 16 overall moments of an $N \times N$ image, the algorithm only requires $8N^2 + 32N$ additions.

Chen [3] has proposed another regular moment extraction algorithm that uses only additions, the partial sum algorithm. The algorithm also uses lower order moments to calculate higher order ones. The technique employed here is to build up higher order moments by repeatedly partially summing over lower order moments. To calculate each row moment, each pixel is added to the pixel before it. Next, each pixel is added to the pixel two before it. This process continues with the distance between added pixels doubling at each iteration until the distance is equal to half the image width. In this way, $O(N^2 \log_2 N)$ additions are required to calculate each set of row moments of an $N \times N$ image except the last. They may be calculated by simply summing the previous values of the intensity function F.

In addition to the two dimensional add and multiply, recursive and partial sum overall moment extraction algorithms, six additional two dimensional algorithms can be created. To do so, the one dimensional algorithms are combined in pairs with one algorithm calculating the row moments and the other calculating the overall moments. Only two of these hybrid algorithms are used in our research. Both use the recursive algorithm to calculate the row moments. The first uses the add and multiply algorithm to calculate the overall moments. The other uses the partial sum algorithm to do so. The first will be referred to as the recursive/add and multiply hybrid, the other as the recursive/partial sum hybrid.

The six pure and two of the hybrid moment extraction algorithms described have been implemented on a single transputer. As is the case with the convolution implementation, the images are acquired from host files and stored in buffers of Parallel C *char* types. Offsets into these buffers are again maintained during execution of the programs. The first sixteen moments are found by all the implementations. They are calculated as Parallel C *double* types. Each program was run on the same five image sizes as the convolution program. Table 2 shows the execution times for the programs when run on a 256×256 image. The times given are again for computation only.

We have discovered the following about the single transputer regular moment implementations. The execution time of the straightforward algorithm increases approximately linearly with the number of image pixels. Due to the large number of additions, multiplications and exponent calculations required, this is by far the slowest of the single transputer implementations. Linear increases in the execution times of the coefficient storage and column product storage algorithms with the number of pixels have also been observed. The timing results for the recursive and the add and multiply implementations increase slightly less than linearly with the number of image pixels. However, the execution time of the partial sum implementation increases more than linearly with the number of pixels. These three programs are in fact the fastest single transputer moment implementations.

In order to allow speedup, efficiency and performance analyses to be performed (see Sections 3 and 4) on two of the multi-transputer programs, the recursive/partial sum and recursive/add and multiply hybrid algorithms have also been implemented. The execution times here increase slightly less than linearly. A final observation is that when the data type of the moments is changed from double to single precision in the recursive program, a significant decrease in execution times results. Some loss of accuracy in the moments extracted results when this is done, however.

3 Multi-Transputer Implementations

Four multi-transputer implementations of both convolution and regular moments have been realized. This section begins with a description of the programming technique we have used in the multi-transputer case. Next, descriptions of the issues relevant to image processing on transputers and our strategies for dealing with them are presented. A subsection describing the multi-transputer implementations for convolution is followed by a similar subsection that gives the details of the regular moment multi-transputer implementations. The section ends with some concluding remarks.

3.1 Programming Technique

The entire B008 motherboard is used in the multi-transputer implementations. Our method of programming here is quite different from that used in the single transputer implementa-

tions. In the single transputer case, a Parallel C program running on the root transputer is not only responsible for controlling input and output to the host but also for all computations needed to perform the image processing tasks. The other transputers are not used. However, in our multi-transputer implementations, all nine transputers may be used. Parallel C is again used. In this case, however, one separate asynchronous linked unit process (a compiled and linked Parallel C source file) executes on each transputer used. The root transputer is still responsible for data transfers to and from the host. All of the necessary computations are executed on the other eight transputers.

Each unidirectional hardware link on the T800 transputer has its own controller. It is therefore possible for all sixteen link controllers and the CPU to operate at the same time. In order to take full advantage of this capability, we have allocated one asynchronous Parallel C process to each unidirectional link and to each computational process. The processes communicate with each other and synchronize themselves using local Parallel C channels as in Hoare's Communicating Sequential Process (CSP) model of parallel computation [6]. This method of parallel programming has also been used in [16].

All local channels and processes are declared as global variables in the source file for each transputer. The *main()* function of each source file is used to allocate channels, allocate and run processes and to time the implementation. Processes are run asynchronously in parallel using the *ProcRun()* function. A channel is created for communication between each pair of processes that communicate in each direction they communicate. Aside from channels and processes, global data structures have been kept to a minimum. In this way, problems of mutual exclusion are avoided. As in Hoare's model, each process keeps its own copy of data. This data can be sent to another process using message passing on a local channel. Configuration source files are used to map the linked units to the appropriate transputer on the B008 for execution.

3.2 General Issues and Strategies

Data parallelism is one of the main parallelization techniques used. Under this scheme, many processors execute the same sequence of instructions on separate parts of a large amount of data. Image processing techniques generally require that the same operation be performed on a large number of pixels. For this reason, we have decided to focus on data parallel methods

in our implementations. In the upcoming subsections we describe the following issues specific to multi-transputer image processing: initial data pass, computational algorithm selection, accumulation of final results and topology selection.

3.2.1 Initial Data Pass

Under the data parallel model, each transputer is responsible for processing a part of the image. These image parts might be passed directly to the transputers using the direct link access scheme suggested by Kille, Ahlers and Schneider [10]. In the direct link access system, one link of each transputer is permanently connected to a specially designed frame grabber transputer which directly passes each transputer its part of the image. Because these links cannot be used to configure the network, direct link access reduces the flexibility of the image processing system. Our approach is as follows.

The B008 root transputer acquires the images from files on the host. With a view to having a balanced computational load on the network and a symmetric computational model, we decided that the root transputer would not be used for computational purposes. It is used as a sort of frame grabber transputer that gets images from the host and passes computational results back. It is also responsible for passing the image segments to the computational transputers for processing. There are two main issues to resolve about this initial data pass. The first is how to split up the image. The second issue is which algorithm to use to pass the image segments to the transputers.

Splitting the Image There are three options for splitting the image. It can be done row by row, column by column or subsection by subsection. Convolution can be done equally well on image rows, columns or subsections. However, regular moments can be calculated by independently calculating row moments or column moments, and then using these moments to calculate the overall image moments (see Section 2). This indicates a row or column oriented distribution scheme. In our implementations each computational transputer is passed and is responsible for processing several rows of the image.

Data Passing Algorithm Selection There are at least two general algorithms for passing the image rows: (a) grouped, and (b) packet. In the grouped approach, the root simultaneously passes all the rows needed down each network path all at once, and every

transputer on a path keeps only the rows for which it is responsible, passing the remainder down the path. In the packet scheme, the root simultaneously passes the rows needed down each path as separate packets, one after another. The packet for the last transputer in line is passed first, the packet for the second last second and so on until the packet for the first transputer is passed last.

For the purpose of passing the image rows, two subtopologies are used in our implementations. The first one is the pipeline structure. In this simple topology, the root is connected to one or more transputers in a chain. To distribute image rows evenly among four transputers in a pipeline, the packet algorithm is found to be superior. The second subtopology is treelike. Here, the root is connected to a router transputer which in turn is connected to three more transputers. The packet method is once again better than the grouped method at distributing image rows to the four transputers in the tree topology. An additional advantage to using the packet algorithm is that larger images can be processed with it than can be processed with the grouped algorithm. Based on these results, we have chosen the packet method as the initial data passing algorithm for our implementations.

3.2.2 Computational Algorithm Selection

There are two ways to design parallel computational algorithms. One can either base them on existing single processor algorithms or create new parallel algorithms which have no relation to any known single processor algorithm. We have chosen the former technique, since it is much simpler. In particular, we have considered both data and instructional parallel versions of the sequential algorithms found in Section 2 and combinations of both. This has enabled us to convert actual code from the single transputer implementations for use in our multi-transputer programs.

3.2.3 Accumulation of Final Results

Generally, the final results must be passed back through the network to the root for output on the host. This process is the inverse of the initial data pass. For this reason, we have decided to use the same packet algorithm for the final data pass. In particular, each computational transputer passes its own results back towards the root and then passes back the results of those farther down its path. The root simultaneously gathers the packets from each network

path ending at it.

3.2.4 Topology Selection

We choose topologies only after algorithm selection as a way of enabling and optimizing a potential implementation. In particular, we choose topologies to make possible all necessary data communication and to minimize the time taken for it. This data communication can be done for the initial data pass, computational data communication or for the accumulation of final results.

3.3 Convolution

In this subsection, we describe the multi-transputer programs we have created for performing convolution. The purely local nature of convolution entailed that it was less challenging a process than regular moments to implement on transputers. However, there were still issues about the initial data pass and topology selection to be investigated. In the next subsection we describe the reasoning that led to the design of the convolution implementations. Detailed descriptions and timing results for the implementations follow.

3.3.1 Design of the Implementations

Keeping in mind the general strategies described in Section 3.2, we have designed and implemented four multi-transputer convolution programs. We have created pure data parallel programs with each computational transputer receiving and processing several rows of the image. In each implementation, the frame grabber root transputer uses the packet algorithm to pass the image rows and the mask. The same convolution algorithm has been implemented in all four programs as a data parallel computational process on all the computational transputers used. The packet method is again used to pass the rows of the convolved image back to the root in all the programs. Two pipeline convolution implementations investigate an issue related to the initial data pass. We have investigated the issue of topology selection by implementing ring and tree versions of one of the two pipeline programs. As in the single transputer case, all the programs use a mask consisting of floating point values and store the image pixels in buffers of *char* data types.

3.3.2 Pipeline Implementations

In the single transputer version of convolution, the image is first padded with a border of zero pixels. In the multi-transputer programs, the borders can either be added in the computational transputers once the image rows have arrived or can be added in the root before the image rows are passed. Cok [2, p. 131] has suggested another approach of initially passing only the rows each transputer is responsible for and having the transputers exchange the bordering rows before beginning convolution. We have implemented pipeline versions of convolution using the first two schemes.

Surprisingly, the more elegant first option proved to be no better than the second in spite of the fact that it requires less data to be passed. In fact, the execution times for the first implementation are inferior for all the mask sizes, image sizes and numbers of processors that we have timed. For example, to convolve a 1024×1024 image with a 7×7 mask, the eight transputer version of the first implementation requires 46.46 seconds while the eight processor version of the second needs 45.67 seconds. We believe that the extra overhead involved in passing the correct rows to each transputer in this case more than makes up for the time saved in the data passes. The times of both implementations decrease dramatically as the number of processors increases from two to four to eight.

In addition, we have discovered the following about the second pipeline implementation. First, the speedups are close to linear for all mask sizes. For a 512×512 image and a 5×5 mask, the speedups range from 1.92 on the two processor program to 7.25 on the eight transputer version. Secondly, the speedups increase as the mask size grows for all image sizes and numbers of transputers. For example, the speedups of the eight processor program improve 14% as the mask size is increased from 3×3 to 13×13 when convolving a 1024×1024 image. Lastly, the efficiencies are all reasonably close to one, decreasing slightly as the number of processors increases. In particular, the efficiency of the implementation ranges from 0.95 to 0.84 when convolving a 512×512 image with a 3×3 mask as the number of processors increases from two to eight.

3.3.3 Ring Implementation

In an attempt to reduce the time taken for data communication, we have created a ring version of the second pipeline implementation. The important feature of the ring implemen-

tation is that the root can distribute image rows and collect convolved rows in two directions at once. This cuts the maximum number of transputers on any one path to or from the root in half as compared to the pipeline version. Reductions in the time needed to distribute the image and to return the convolved image to the host are realized.

The ring timing results for a 512×512 image are given in Figure 4. The results are quite similar to those of the second pipeline implementation. The ring implementation is actually only slightly faster. For example, the eight processor ring program takes 141.9 seconds to convolve a 1024×1024 image with a 13×13 mask as opposed to the 142.5 seconds needed by the eight transputer second pipeline program. This is less than a 0.5% improvement. Speedups for these timing results are shown in Figure 5. The graphs are linear, as is the case with the second pipeline implementation. However, the speedups are slightly greater, reflecting the improved timing results.

3.3.4 Tree Implementation

With the reduction of communication time again in mind, we next converted the ring implementation to an eight transputer tree implementation. Although the maximum number of transputers on both paths from the root is the same as it is for the ring topology, the maximum path length from the root is shortened to three from four. However, in most cases the execution times for the eight transputer tree are no better than the corresponding times for the eight transputer ring. In fact, for a 1024×1024 image and a 13×13 mask, its execution time of 143.2 seconds is worse than the time attained by the second pipeline implementation (142.5 seconds). Only when the program is run on a 3×3 mask on a 64×64 or a 128×128 image does the tree implementation run faster than the ring. We have also noticed that the efficiencies of the convolution implementations increase as the image size increases. For example, the efficiency of the four processor ring program increases from 95.8% for a 64×64 image to 96.3% for a 1024×1024 image when the mask size is 13×13.

It is most important to note that the convolution implementations all have close to linear speedups. This is typical of local techniques with pure data parallel implementations which require no communication between processors during the computational stage. Some observations about topology selection are also indicated. The transputer topology is varied over the last three convolution implementations. The time needed to pass the original and

convolved images is partly determined by the topology used. Yet the timing results for these implementations are quite similar. We conclude from this that for pure data parallel image processing algorithms implemented on the B008, topology selection is relatively unimportant. However, for another transputer board, topology selection may be more important, since the paths along which the image segments are passed may be longer.

During the computational stage, $\frac{N^2M^2}{P}$ floating point multiplications and additions are executed on each of P processors. For an N processor array where each processor is responsible for performing convolution on one row of the image, the parallel asymptotic complexity of our convolution implementations will be $O(NM^2)$ floating point multiplications and additions.

3.4 Regular Moments

The global nature of regular moment calculation made it a more challenging image processing technique to implement on transputers than convolution. In this subsection we give descriptions and the timing results of our four multi-transputer implementations of regular moments.

3.4.1 Design of the Implementations

In designing four regular moment implementations, we have again used the general strategies we have developed. The frame grabber transputer again passes packets of image rows to each computational transputer for processing. The tree topology is used for this initial data pass. The two dimensional recursive algorithm results in the fastest single transputer implementation. We therefore considered using a data parallel version of the two dimensional recursive algorithm for moment calculation. Each transputer might calculate the sixteen two dimensional moments for its image rows using this algorithm. An instructional parallel algorithm could then be used to add those separate sets of moments to obtain the overall moments for the entire image. However, this technique does not work. The one dimensional recursive algorithm can certainly be used to calculate the row moments for each transputer's image rows. However, the one dimensional recursive algorithm cannot be used on these row moments to calculate the overall two dimensional moments of the rows.

Undaunted, we realized that we could still use the one dimensional recursive algorithm to

calculate row moments for each transputer's rows. We also noted that the three algorithms with the fastest single transputer implementations (recursive, add and multiply, and partial sum) share another property besides speed. Each one separates the task of moment calculation into first calculating the moments of each image row and then using them to calculate the overall two dimensional image moments. We realized that we might exploit this in our implementations. We therefore have created three implementations which all use a data parallel version of the one dimensional recursive algorithm to calculate row moments on each computational transputer. These moments are then used by instructional parallel versions of the three algorithms to calculate the overall image moments. We refer to each implementation according to the name of the instructional parallel algorithm used, namely recursive, add and multiply, and partial sum. The fourth multi-transputer regular moment implementation is an alternative version of the partial sum implementation. In all four implementations, the instructional parallel algorithm places the image moments in a transputer connected to the root. Additional transputer connections have been added to the tree structure so that the necessary data passing during the instructional parallel phase of the implementations can be accomplished. As is the case with the single transputer regular moment implementations, the moments are calculated as double precision numbers in the multi-transputer programs.

3.4.2 Partial Sum Implementations

Chen suggested this regular moment extraction algorithm for a linear array of processors in 1990 [3]. It was necessary to augment the basic tree structure heavily for the eight transputer partial sum implementation. P_0 and P_7 are the router transputers in the tree. The following connections between computational transputers were made unless they were part of the basic tree structure. First of all, the transputers were joined into a pipeline. Secondly, with the exception of the third (P_2) and fourth (P_3) transputers, each transputer was connected to one two before or after it. Lastly, every transputer was joined to the transputer four positions before or after it. The resulting topology (shown in Figure 6) is formidable indeed.

We have observed the following about the partial sum implementation. First, increasing the number of transputers in the implementation results in decreases in execution times for all image sizes. Secondly, the implementation is most efficient for larger images. In particular, the efficiency of the eight processor program increases from 43% for a 64×64 image to

74% for a 1024×1024 image. In addition, the partial sum speedups are less than those observed in our convolution implementations. For a 512×512 image, the speedups range from 1.88 on the two transputer program to 5.81 on the eight processor program. Lastly, the efficiency of the implementation decreases dramatically as the number of processors increases for a given image size. For example, the efficiency of the two transputer program is 0.92 for a 256×256 image. The efficiency of the eight transputer program drops to 0.69 when run on a 256×256 image. The reason for the quick decline in efficiency as the number of transputers grows is as follows. Increasing the number of transputers decreases the amount of time needed to calculate the row moments. It also decreases the time needed to perform the additions required in the partial sum stage of the computation which calculates the overall moments. However, the partial sum technique requires a large amount of data communication when executed on more than one transputer. As the number of transputers increases, this requirement increases dramatically. To make matters worse, for the eight transputer program, routers (P_3, P_4) are necessary because of lack of links. This results in a communication bottleneck which further slows down the last stage of the computation. The observed decreasing efficiencies make sense in light of this information.

We have also implemented an alternative version of this partial sum implementation. In this implementation, the partial sum process is done for all four sets of row moments at once, instead of for only one set at a time, as Chen suggests. This reduces the number of data transfers (if not the amount of data sent) by a factor of four. In spite of this reduction, this more elegant implementation is no faster than the simpler method. For most image sizes and numbers of transputers that we have timed, the simpler first method is faster or comparable to the second. In particular, when finding the moments of a 1024×1024 image, the execution times of the two eight transputer programs are virtually identical (2.484 seconds and 2.482 seconds respectively). We attribute the lack of improvement of the second implementation to the larger overhead in buffer management that it requires compared to the first method.

3.4.3 Recursive Implementation

Extra connections are made from the basic tree structure to create a pipeline out of the computational transputers in the recursive topology. The eight transputer recursive topology is shown in Figure 7. Once the row moments have been calculated on each transputer, the one

dimensional recursive algorithm is executed across the computational transputers as follows. The rightmost computational transputer (P_7) begins by executing the one dimensional recursive algorithm on its row moments. (See Figure 3.) Using the algorithm, P_7 determines not only the values of the sixteen overall moments, $MP_{7KL}, K = 0, 1, \ldots, 3, L = 0, 1, \ldots, 3$ for its rows but also those of the middle eight overall moments, $PRMP_{7KL}, K = 1, 2, L = 0, 1, \ldots, 3$ for the rows starting at its second row. When the algorithm ends, P_7 passes these 24 moments to the transputer on its left, P_6.

After receiving these moments, P_6 executes the one dimensional recursive algorithm on its own array of row moments RM with STM set to be MP_7 and SPM set to $PRMP_7$ as in Figure 3. The pseudo code for P_6 in the eight transputer recursive implementation is shown in Figure 8. $STRM$ and $SPRM$, the starting values of the current and previous row moments, are zero arrays. After this second execution of the recursive algorithm on P_6, the resulting 24 moments MP_6 and $PRMP_6$ are passed on to P_5. This process continues down the pipeline of transputers. After each transputer has executed the algorithm, the overall moments for the part of the image starting at its first row have been calculated. When the algorithm has executed on the leftmost transputer P_0, the sixteen overall image moments have been calculated. At this time, P_0 passes the image moments back to the root for output.

As might be expected, the execution times for the recursive implementation have been found to decrease as the number of transputers used increases. As is the case with the partial sum implementations, we have found that the speedup graphs of the recursive implementation are less than those of the convolution implementations. A rapid decline in the efficiency of the implementation as the number of processors increases is also evident. We believe this decline is due to the fact that increasing the number of transputers does reduce the time needed to calculate the row moments but actually increases the time taken to calculate the overall moments. This is because during the overall moment calculation only one computational transputer is active at a time and adding transputers only serves to increase the number of moment data passes required in this last instructional parallel stage of the computation. This is the biggest drawback of the recursive implementation. Perhaps if all the computational transputers were to work together and at once on calculating the overall moments a faster implementation would result. The add and multiply implementation attempts to make use of this idea.

The recursive implementation requires $\frac{8N^2}{P}$ double precision additions to calculate the row moments of an $N \times N$ image on an array of P processors using the one dimensional recursive algorithm. It requires an additional $32N$ double precision additions and $24P$ double precision data transfers to calculate the first 16 overall moments. In particular, for an array of N processors, the algorithm requires $40N$ additions and $24N$ data transfers. Therefore, when implemented on such an array, the algorithm clearly requires $O(N)$ double precision additions and data transfers.

3.4.4 Add and Multiply Implementation

In the description of our design of the regular moment implementations, our disappointment at the impossibility of a pure data parallel version of the two dimensional recursive algorithm to calculate overall moments was expressed. We could not simply use it to calculate the overall moments for each transputer's rows and add the resulting sets of moments together to get the moments for the entire image. The add and multiply algorithm, however, can be used in this way to arrive at the overall image moments. The topology required is relatively simple compared to that needed for the partial sum program. It is simply a tree structure with the fifth (P_4) and seventh (P_6) computational transputers also connected. P_0 and P_7 are again the router transputers. The eight transputer add and multiply topology is shown in Figure 9.

Once the row moments have been calculated in the add and multiply implementation, a data parallel version of the one dimensional add and multiply algorithm is executed simultaneously on the row moments on each computational transputer. The offset of the first row each transputer processes is used. When this has been done, the sets of 16 moments are added up across the processors in partial sum fashion. It is for this addition that the extra connection in the topology is needed. The first transputer (P_0) ends up with the moments and passes them back to the root for host output. Figure 10 gives pseudo code for transputer P_4 in the eight transputer add and multiply implementation. $STRM$ and $SPRM$ are again zero arrays.

In Figure 11 the add and multiply timing results for various image sizes can be found. Again, as the number of processors increases, the times all decrease. Figure 12 shows the speedup graphs for these times. The speedups for the three largest image sizes are very

similar, with the 1024×1024 image having the greatest speedup, and the 512×512 having the next largest speedup. Note that the speedups are linear for these larger image sizes, but less than linear for the smaller ones.

The add and multiply algorithm requires $\frac{8N^2}{P}$ double precision additions to calculate the row moments of an $N \times N$ image on an array of P processors. In addition, $\frac{16N}{P}$ double precision multiplications and additions are needed to calculate the first 16 overall moments of the image rows of each processor. To add these up, it takes $\sum_{i=0}^{\log_2^P - 1} 2^i 16 = 16(P-1)$ double precision data transfers and $16 \log_2^P$ additions. For an array of N processors, the add and multiply algorithm clearly will also have linear asymptotic complexity in the number of double precision additions and data transfers.

In contrast with the convolution implementations, our regular moment implementations suffer from rapidly decreasing efficiencies as the number of transputers increases. The global nature of moments entails that as the number of processors used to calculate them increases, so too does the amount of communication needed among the processors. It is this communication increase that has resulted in the decreased efficiencies of our four and eight transputer moment implementations. Additionally, although we did not vary topologies for our moment implementations, we feel that the moment programs would be quite sensitive to changes in topology. This is also clearly different from the convolution implementations.

In 1990, Chen [3] proposed efficient parallel regular moment algorithms for both linear and two dimensional processor arrays. The algorithm proposed for the linear array is the multi-transputer partial sum technique we have implemented. For a two dimensional array of processors, each of which is given a subsection of the image, Chen recommends that the processors first collectively use the partial sum algorithm to calculate the row moments and again use the partial sum procedure on these moments to calculate the overall image moments. In a more recent article, Pan [11] showed that because Chen forgot to take the time needed for data passing into account, the theoretical timing analysis presented in his article is incorrect. In so doing, Pan demonstrated that Chen's two dimensional algorithm implemented on an $N \times N$ array of processors would have the same parallel asymptotic time complexity of $O(N)$ as his one dimensional algorithm implemented on an N processor linear array when calculating the regular moments of an $N \times N$ image.

We believe that the behaviour of an implementation is most significant as image size grows

and the number of transputers used increases. For the largest image size of 1024×1024, the execution time of the eight transputer add and multiply program is 2.351 seconds and that of the recursive implementation 2.374 seconds. These times excel the corresponding time of 2.484 seconds for the partial sum implementation.

This indicates that when implemented on the B008 in Parallel C, the add and multiply and the recursive multi-transputer algorithms are superior to the partial sum algorithm proposed by Chen. We have also shown that the algorithms have the same linear parallel time complexity as the partial sum algorithm for a linear array of N SIMD or MIMD processors.

4 Performance Models

In this section, we present high level performance models of two of our multi-transputer implementations. We ignore the overhead incurred with Parallel C inter process communication and process switching time on each transputer. Since all the implementations have the same basic structure we have chosen to model only the best implementation for convolution (ring) and regular moments (add and multiply). We believe that the models for the other implementations would be similar. Since the parameters were chosen for runs where $N = 512$ and $M = 7$, the models work best for these values of N and M. However, the models may also be applicable for other values of N and M. In our models, the following notation is used: P denotes the number of transputers used; N denotes the image size and $T_{ps}(B,T)$ denotes the time in seconds needed to send B bytes to each of T transputers on a path from the root using the packet method.

4.1 Convolution Model

4.1.1 Notation

In this model, the following additional notation is used:

M denotes the mask size.

$T_{il}(N,M,P)$ denotes the total time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the ring implementation with P computational transputers.

$T_{cp}(N, M, P)$ denotes the computation time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the ring implementation with P computational transputers.

$T_{cm}(N, M, P)$ denotes the data communication time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the ring implementation with P computational transputers.

$T_{sn}(N, M)$ denotes the time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the single transputer convolution program.

4.1.2 Computation Time

In the single transputer convolution program, N^2M^2 floating point additions and N^2M^2 multiplications and $2N^2M^2$ array references are required. In addition, there are $2N^2M$ array offset additions performed. Finally, N^2 function calls and array references are needed. Therefore,

$$T_{sn}(N, M) \simeq k_1 N^2 M^2 + k_2 N^2 M + k_3 N^2$$

The values of the constants in the above equation are found to be $k_1 \simeq 5.91 \times 10^{-6}$, $k_2 \simeq 3.26 \times 10^{-6}$, and $k_3 \simeq 1.26 \times 10^{-5}$. k_3 was determined by timing the program with only the N^2 loop executing. This was accomplished by running the program with the NM^2 loop commented out. k_3 was set to be the execution time divided by N^2. Next, k_2 was set to be the execution time of the NM^2 loop divided by NM^2. To get this time, the execution time of the N^2 loop was subtracted from the execution time of the NM^2 loop and the N^2 loop. Lastly, k_1 was set to be the execution time of the N^2M^2 loop divided by N^2M^2. Substituting these values of constants, we get

$$T_{sn}(N, M) \simeq 5.91 \times 10^{-6} N^2 M^2 + 3.26 \times 10^{-6} N^2 M + 1.26 \times 10^{-5} N^2$$

These computations are spread evenly among the P computational transputers used. Therefore, we should get

$$T_{cp}(N, M, P) = \frac{T_{sn}(N, M)}{P}$$

$$= \frac{5.91 \times 10^{-6} N^2 M^2 + 3.26 \times 10^{-6} N^2 M + 1.26 \times 10^{-5} N^2}{P}$$

4.1.3 Communication Time

Each image consists of N^2 pixels. In our multi-transputer implementations, each computational transputer is generally passed $1/P$ of the pixels or $\frac{N^2}{P}$ pixels. In the ring implementation, the root passes the pixels to each of $P/2$ transputers down both branches of the ring at the same time using the packet method. This takes $T_{ps}(\frac{N^2}{P}, \frac{P}{2})$. The root must first separate the image into P packets for passing. This takes time proportional to the number of pixels in the array, N^2. Additional time is required to send the packets down the line of transputers. This takes time proportional to the number of bytes in the packet ($\frac{N^2}{P}$) and to the number of transputers in the line ($P/2$). Therefore,

$$T_{ps}(\frac{N^2}{P}, \frac{P}{2}) \simeq k_1 N^2 + k_2(\frac{N^2}{P})(\frac{P}{2})$$

We have created a simple multi-transputer program that simply uses the packet method to distribute sets of image rows to several transputers. Experimentation with this program has enabled us to determine k_1 and k_2. In general, there may be extra overhead involved in the initial data pass than that needed in this simple program. With that fact in mind, we have set k_1 to be 2.25×10^{-7} and k_2 to be 6.42×10^{-7}, slightly higher than the timing results of the data passing program indicate they should be. Substituting these values, we get,

$$T_{ps}(\frac{N^2}{P}, \frac{P}{2}) \simeq 2.25 \times 10^{-7} N^2 + 6.42 \times 10^{-7}(\frac{N^2}{P})(\frac{P}{2})$$

$$= 2.25 \times 10^{-7} N^2 + 3.21 \times 10^{-7} N^2$$

$$= 5.46 \times 10^{-7} N^2$$

The data pass is repeated in reverse at the end of the program when the convolved image is passed back to the root. Therefore, the total communication time,

$$T_{cm}(N, M, P) \simeq 2T_{ps}(\frac{N^2}{P}, \frac{P}{2})$$

$$= 2 \times 5.46 \times 10^{-7} N^2$$

$$= 1.09 \times 10^{-6} N^2$$

4.1.4 Overall Time

The total execution time, T_{tl} is composed of T_{cp} and T_{cm}, *i.e.*,

$$T_{tl}(N, M, P) \simeq T_{cp}(N, M, P) + T_{cm}(N, M, P)$$

Therefore,

$T_{tl}(N, M, P) \simeq \frac{5.91\times10^{-6}N^2M^2+3.26\times10^{-6}N^2M+1.26\times10^{-5}N^2}{P} + 1.09 \times 10^{-6}N^2$. In Figure 13, the last function is plotted for $P = 2, 4, 8$ with $N = 512$ and $M = 7$. The actual timing results for the ring implementation are also plotted. The actual times are slightly more than the estimated times. The discrepancy may be explained as follows. In our communication model, we have not taken into account the time needed to send the convolution mask down each branch of the ring. Nor have we allowed for the extra time needed to send the borders of zeroes to the computational transputers. Our setting of the constants in the model (see Section 4.1.3) may compensate for this. However, we have also ignored the computational time needed to add the borders to the image in the root.

4.2 Regular Moment Model

4.2.1 Notation

The additional notation used in this model is as follows:

$T_{tl}(N, P)$ denotes the total time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the add and multiply implementation with P computational transputers.

$T_{cp}(N, P)$ denotes the computation time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the add and multiply implementation with P computational transputers.

$T_{cm}(N, P)$ denotes the data communication time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the add and multiply implementation with P computational transputers.

$T_{sn}(N)$ denotes the time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the single transputer hybrid recursive/add and multiply program.

4.2.2 Computation Time

In the single transputer implementation of the recursive/add and multiply hybrid algorithm, $8N^2$ additions and N^2 array references are required. There are also $16N$ multiplications and additions and N array offset additions needed. Consequently, we get

$$T_{sn}(N) \simeq k_1 N^2 + k_2 N \simeq 1.32 \times 10^{-5} N^2 + 1.66 \times 10^{-4} N$$

The empirical values are $k_1 \simeq 1.32 \times 10^{-5}$ and $k_2 \simeq 1.66 \times 10^{-4}$. These were determined in a similar manner to the computational constants in the convolution model.

In the last stage of the add and multiply implementation, the 16 moments calculated by the P processors are added up in partial sum fashion across the network. This should take time proportional to $\log_2 P$, *i.e.*, this time equals $k_1 \log_2 P$ for some k_1. For a 512×512 image, $k_1 \simeq 1.47 \times 10^{-2}$. As with the convolution ring implementation, all computation except this last addition is spread evenly among the P computational transputers. Thus,

$$T_{cp}(N, P) \simeq \frac{T_{sn}(N)}{P} + 1.47 \times 10^{-2} \log_2 P$$

$$= \frac{1.32 \times 10^{-5} N^2 + 1.66 \times 10^{-4} N}{P} + 1.47 \times 10^{-2} \log_2 P$$

4.2.3 Communication Time

In the add and multiply implementation, the image is distributed to the P transputers using the packet method as is the case with the ring convolution implementation. This again takes $T_{ps}(\frac{N^2}{P}, \frac{P}{2})$. In the regular moment implementations, however, there is no convolved image to pass back to the root as is the case with convolution. Using the same values for the constants in the convolution ring model for this initial data pass, we get

$$T_{cm}(N, P) \simeq 5.46 \times 10^{-7} N^2$$

4.2.4 Overall Time

We know that $T_{tl}(N, P) \simeq T_{cp}(N, P) + T_{cm}(N, P)$. Hence,

$$T_{tl}(N, P) \simeq \frac{1.32 \times 10^{-5} N^2 + 1.66 \times 10^{-4} N}{P} + 1.47 \times 10^{-2} \log_2 P + 5.46 \times 10^{-7} N^2$$

In Figure 14, the above function is plotted for $P = 2, 4, 8$ with $N = 512$. The actual timing results for the add and multiply implementation are also plotted. This time our estimates are slightly greater than the actual values. We believe that this model is inflated because the regular moment implementations require little extra overhead in the initial data pass. For simplicity, the same values for the constants involved in the initial data pass model have been used in both the convolution and regular moment models. More accurate models might be obtained in the following manner. First, the values of the constants used in the initial data pass model might be reduced. Next, an extra term could be added to the computational time part of the convolution model to account for the padding of the image with zeros done in the root. Lastly, the extra data sent (the zero borders and the convolution mask) might be accounted for in the communication aspect of the convolution model.

5 Conclusion

We have presented parallel implementations of the image processing techniques of convolution and regular moments on multiple transputers programmed with Parallel C. Several single transputer programs were first created in preparation for multi-transputer programming. Eight multi-transputer programs have been developed and analyzed. We have created performance models of two of the multi-transputer programs.

Our results are as follows. Most importantly, we have observed linear speedups for all the multi-transputer convolution programs. The multi-transputer regular moment programs are also considerably faster than their single transputer counterparts. However, as the number of transputers increases, the efficiencies of the programs decrease at a greater rate than in the convolution implementations. This is a symptom of the increase in communication overhead that parallel implementations of global techniques undergo as the number of processors increases. Local techniques, such as convolution, are immune to this effect.

In addition, we feel that the following results are significant. First, we have shown that the packet method of data transmission is superior to the grouped technique of passing data. On the same topic of data communication, the first convolution pipeline implementation and the alternative partial sum implementation indicate that reducing the amount of data transferred in an implementation or the number of transfers required does not always result

in faster multi-transputer programs. We have also learned that topology selection does not greatly influence the efficiency of multi-transputer implementations on the B008 of pure data parallel algorithms such as those we created for convolution. Also, our idea of making each transputer responsible for processing several rows of the image seems to work well, especially for implementing global techniques like moments and the Fourier transform.

References

[1] Budrikis, Z. and Hatamian, M., 'Moment Calculations by Digital Filters', *AT & T Bell Lab Tech. J.*, Vol 63, No. 12, pp. 217-229, 1984.

[2] Cok, R.S., *Parallel Programs for the Transputer*, Prentice-Hall, U.S.A., 1991.

[3] Chen, K., 'Efficient Parallel Algorithms for the Computation of Two-Dimensional Image Moments', *Pattern Recognition*, Vol 23, No. 12, pp. 109-119, 1990.

[4] Gonzalez, R. C. and Wintz, P., *Digital Image Processing, 2nd Edition*, Addison-Wesley, U.S.A, 1987.

[5] Hatamian, M., 'A Real Time Two-dimensional Moment Generating Algorithm and its Single Chip Implementation', *IEEE Trans. Acoust. Speed Signal Process*, ASSP-34, pp. 546-553, 1986.

[6] Hoare, C. A., 'Communicating Sequential Processes', *Communications of the ACM*, Vol. 21, pp. 666-667, 1978.

[7] INMOS Ltd., 'ANSI C Toolset User Manual', Document No. 72-TDS-22400, INMOS Ltd., Bristol, U.K., 1990.

[8] INMOS Ltd., 'Image Processing TRAM IMS B429-16', INMOS Ltd., Bristol, U.K., 1993.

[9] INMOS Ltd., 'IMS B008 User Guide and Referencs Manual', Document No. 72-TRN-22300, INMOS Ltd., Bristol, U.K., 1990.

[10] Kille, K., Ahlers, R.-J. and Schneider, B., 'Experiences with Transputer Systems for High Speed Image Processing', *Proc. SPIE - Int. Soc. Opt. Eng. (USA)*, Vol. 1386, pp. 76-83, 1991.

[11] Pan, Y., 'A Note on Efficient Parallel Algorithms for the Computation of Two-Dimensional Image Moments', *Pattern Recognition*, Vol. 24, No. 9, p. 917, 1991.

[12] Papoulis, A., *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, 1965.

[13] Peng, A. and Kawamura, K., 'Parallel Image Processing on a Transputer-Based System', *Los Alamitos, CA, U.S.A.: IEEE Comput. Soc. Press*, pp. 235-238, 1990.

[14] Reeves, A. P., 'Parallel Algorithms for Real-Time Image Processing', *Multicomputers and Image Processing, Algorithm and Program*, Academic Press, New York, pp. 7-18, 1982.

[15] Turner, C., 'Parallel Implementations of Selected Image Processing Techniques', M. Sc. Thesis, Faculty of Computer Science, University of New Brunswick, Fredericton, N. B., CANADA, TR 93-077, pp. 95, May 1993.

[16] Viswanathan, R., Thube, S., Ogale, S. B., Bhavsar, V. C., and Madhukar, A., 'Simulated Annealing for Semiconductor Surface Relaxations on Multi-Transputer Systems', TR/AP/90/001, Centre for Development of Advanced Computing, Pune University Campus, Pune 411007, India, June 1990.

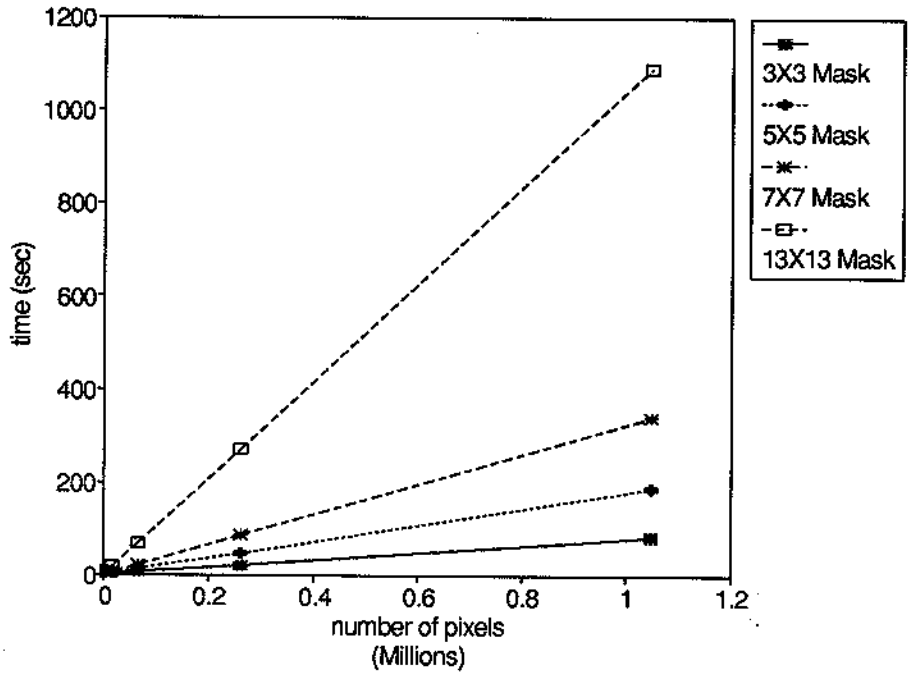

Figure 1: Single Transputer Convolution Execution Times For Various Mask Sizes

MASK SIZE	TIME(SEC)
3×3	5.166
5×5	11.73
7×7	21.31
13×13	68.02

Table 1: Single Transputer Convolution Execution Times in Seconds for a 256×256 Image

```
AAM(F, N1, N2, offset, M)
  for i = 0..N1 − 1
      for l = 0..3
          M(l, i) = 0
          for j = 0..N2 − 1
              M(l, i)+ = F(i, j)
              F(i, j)∗ = (j + offset + 1)
          end for
      end for
  end for
END
```

Figure 2: One Dimensional Add and Multiply Algorithm for Moment Calculation

```
REC(F, N1, N2, STM, SPM, M, PRM)
  for i = 0..N1 − 1
      M0 = STM(0, i)
      M1 = STM(1, i)
      M2 = STM(2, i)
      M3 = STM(3, i)
      PRM1 = SPM(1, i)
      PRM2 = SPM(2, i)
      for j = 0..N2 − 1
          M0+ = F(i, N2 − 1 − j)
          M1+ = M0
          M2+ = M1 + PRM1
          M3+ = M2 + M2 + PRM2 − M1
          PRM1 = M1
          PRM2 = M2
      end for
      M(0, i) = M0
      M(1, i) = M1
      M(2, i) = M2
      M(3, i) = M3
      PRM(1, i) = PRM1
      PRM(2, i) = PRM2
  end for
END
```

Figure 3: One Dimensional Recursive Algorithm for Moment Calculation

ALGORITHM	TIME(SEC)
straightforward	342.8
coefficient	17.01
column products	12.6
recursive	0.8803
add and multiply	2.484
partial sum	9.322
recurse/multiply	0.9055
recurse/partial	1.036

Table 2: Single Transputer Regular Moment Execution Times in Seconds for a 256×256 Image

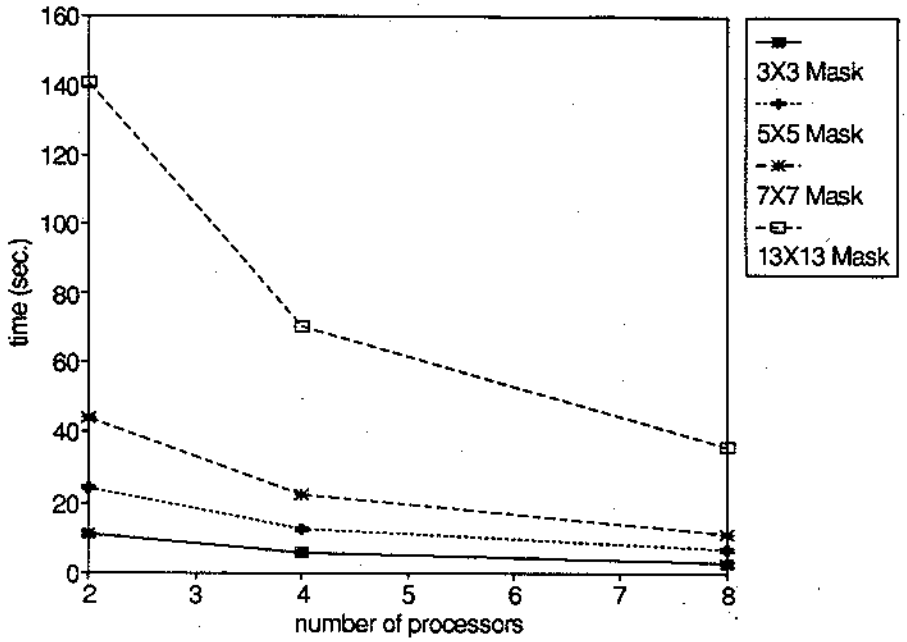

Figure 4: Convolution Ring Execution Times for a 512×512 Image

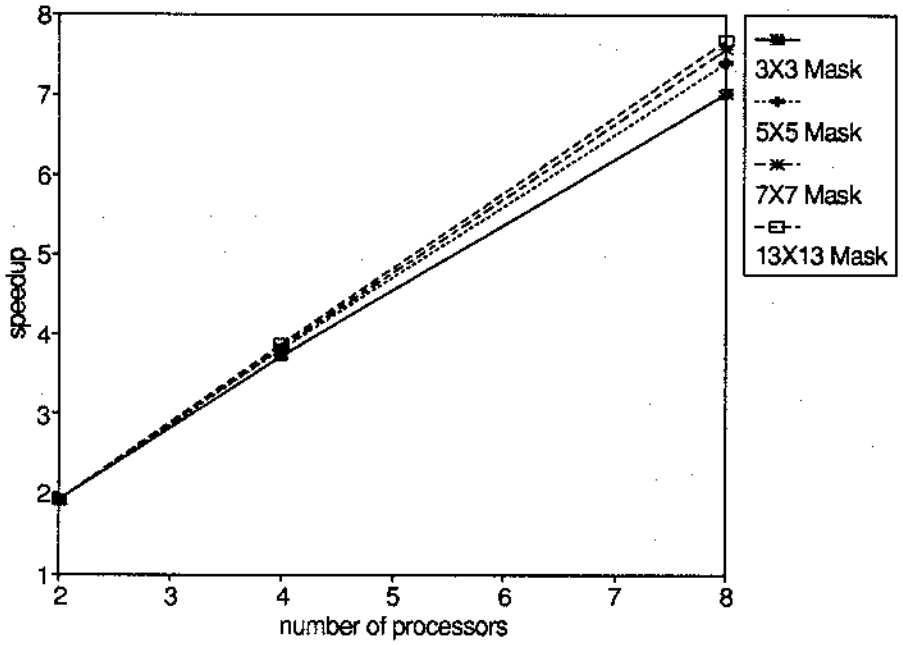

Figure 5: Convolution Ring Speedups for a 512×512 Image

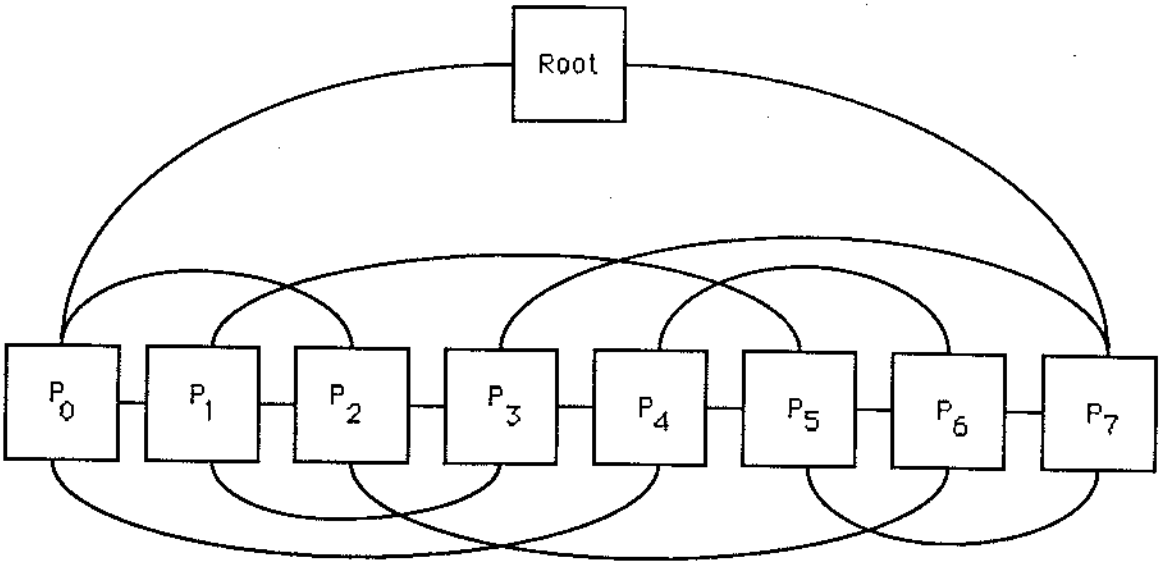

Figure 6: Eight Transputer Partial Sum Topology

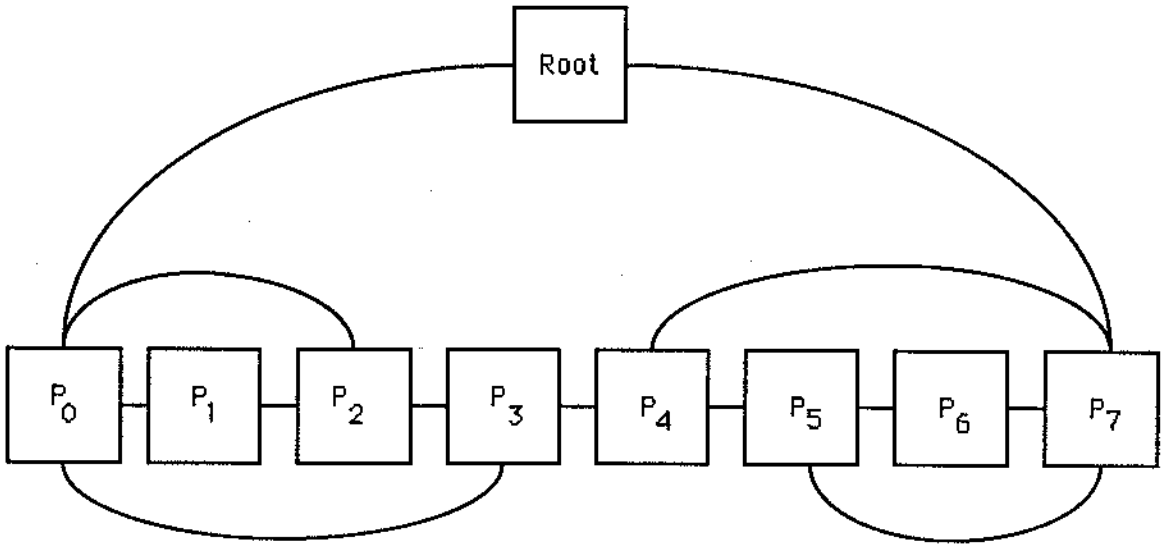

Figure 7: Eight Transputer Recursive Topology

BEGIN
get the image dimension N from P_7
get the image rows $3N/4$ to $7N/8-1$ F from P_7
$REC(F, N/8, N, STRM, SPRM, RM, PRRM)$
get 16 moments MP_7 from P_7
get 8 moments $PRMP_7$ from P_7
$REC(RM, 4, N/8, MP_7, PRMP_7, MP_6, PRMP_6)$
send 16 moments MP_6 to P_5
send 8 moments $PRMP_6$ to P_5
END

Figure 8: Eight Transputer Recursive Pseudo Code for P_6

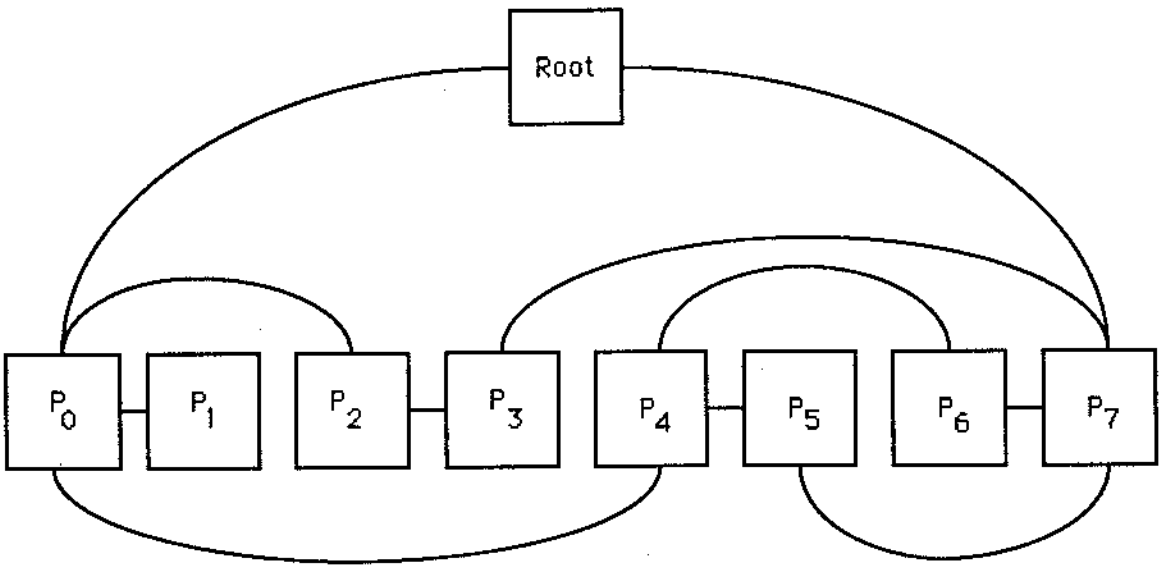

Figure 9: Eight Transputer Add and Multiply Topology

```
BEGIN
   get the image dimension N from P0
   get the image rows N/2 to 5N/8 − 1 F from P0
   REC(F, N/8, N, STRM, SPRM, RM, PRRM)
   AAM(RM, 4, N/8, N/2, M)
   get 16 overall moments MP5 from P5
   for k = 0..3
       for l = 0..3
           M(k, l)+ = MP5(k, l)
       end for
   end for
   get 16 overall moments MP6 from P6
   for k = 0..3
       for l = 0..3
           M(k, l)+ = MP6(k, l)
       end for
   end for
   send 16 overall moments M to P0
END
```

Figure 10: Eight Transputer Add and Multiply Pseudo Code for P_4

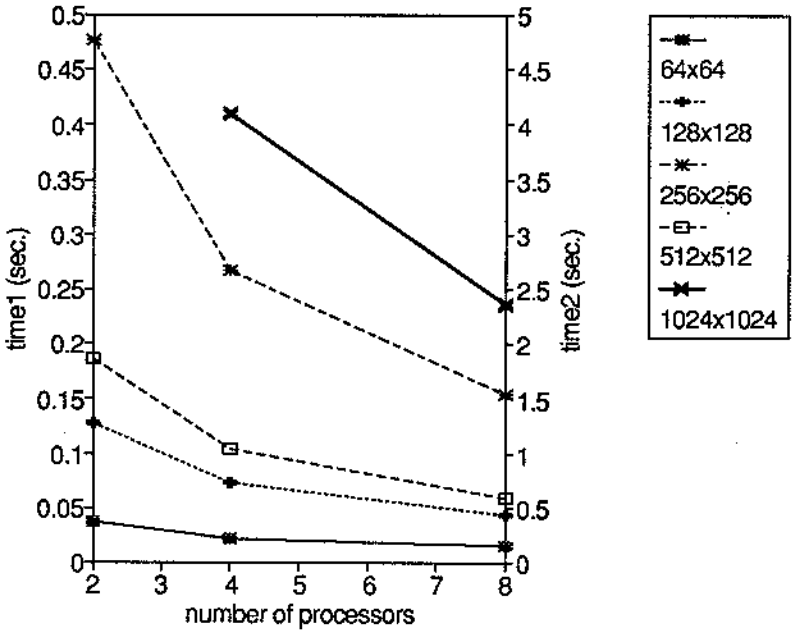

Figure 11: Add and Multiply Execution Times (**time1:** 64×64 Image, 128×128 Image, 256×256 Image; **time2:** 512×512 Image, 1024×1024 Image)

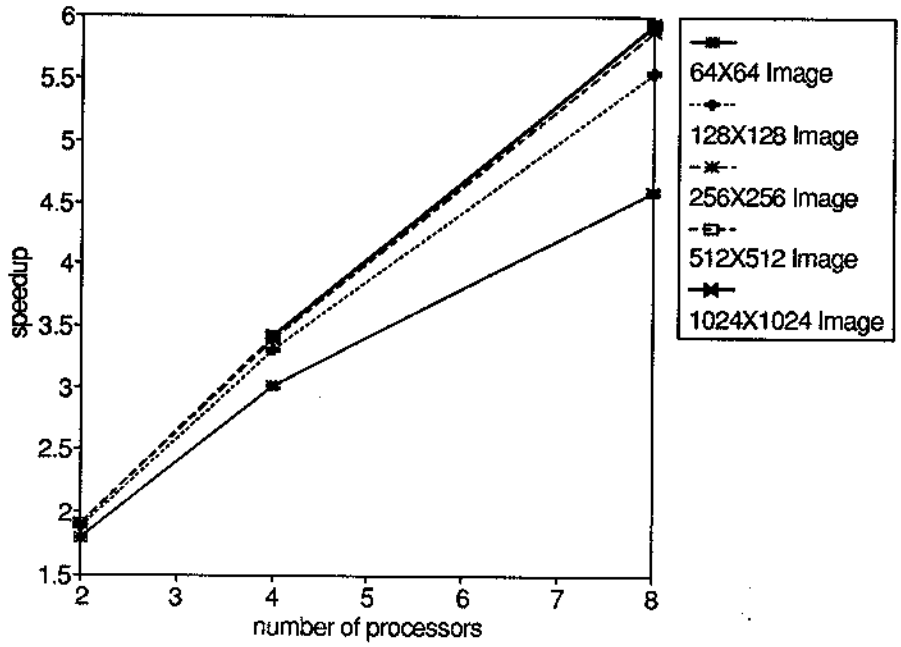

Figure 12: Add and Multiply Speedups

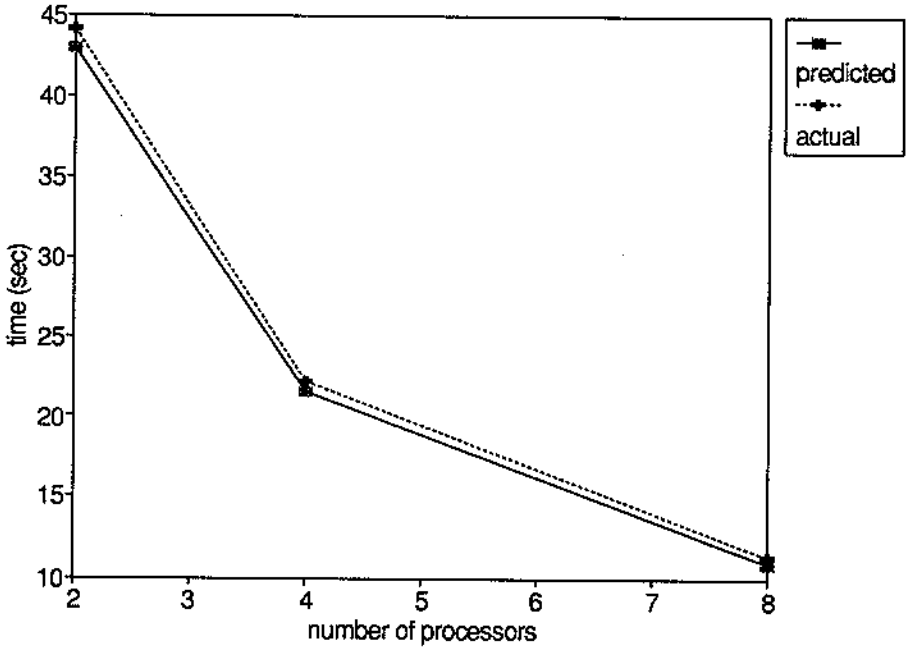

Figure 13: Comparison of Convolution Execution Times for a 512×512 Image and a 7×7 Mask

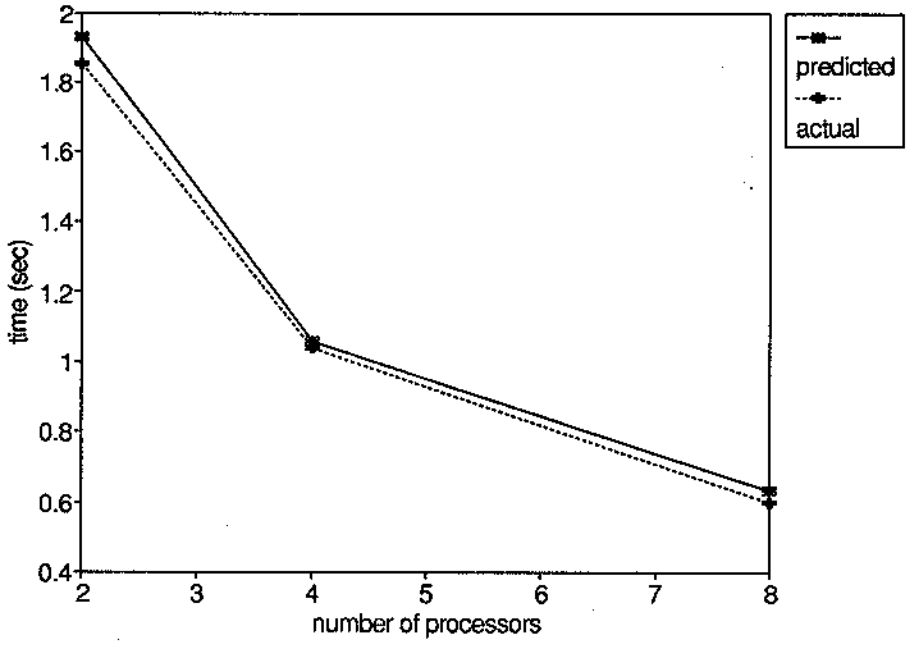

Figure 14: Comparison of Regular Moment Execution Times for a 512×512 Image