Initial Findings of an Investigation into the Feasibility of a Low Level Image Processing Workstation using Transputers

THESIS

Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF SCIENCE (APPLIED COMPUTER SCIENCE)

of

Rhodes University

by

NICHOLAS DUNCAN COOKE

December 1989

Acknowledgements

The author would like to thank his supervisor, Professor Peter Clayton, for all the help and advice offered as well as the two years of unending patience. All the advice, help and encouragement offered by colleagues, is greatly appreciated. A special thanks to Vanessa Jones for her help in proof reading this document.

Contents

List of Illustrations

Chapter One

1. Introduction

Image processing can be defined as the acquisition, transmission, processing, and display of information, where this information constitutes an image. The image spoken of here is usually one captured by a digitizing camera and is composed of discrete intensity components, commonly known as pixels [ROS76]. This form of picture or image processing, by computer, encompasses a wide variety of techniques and mathematical tools. This must not be confused with the processing of images synthesized by computer, which is the principle concern of computer graphics [ROS76]. Rosenfeld and Kak [ROS76] identify three major problem areas in response to which image or picture processing has developed. These are:

1. Picture digitization and coding: Digitization concerns the conversion of pictures from continuous to discrete form. Coding or *compression* of images arises from the storage problem, as a result of the large amounts of data making up an image. These two areas are discussed in more detail later in this thesis.

2. Picture enhancement and restoration: This concerns the improvement of degraded images. A common application of this is the treatment of blurred and noisy images [COO88].

3. Picture segmentation and description: This deals primarily with the conversion of pictures into simplified *maps*, and the measurement of properties or picture parts.

Areas 2 and 3, form the backbone of the field of image processing. Several authorities provide good detailed descriptions of the techniques required [ROS76][ROS76a][GON77]. It is from these techniques that the present day applications of image processing have been developed. These applications typically include remote sensing [ROS76a], pattern recognition and radiographic image analysis, or more commonly medical imaging [MAC83]. The subject of medical imaging itself encompasses many applications, including Computer Aided Tomography (CAT scanning), Nuclear Magnetic Resonance Imaging (NMR imaging) and cell analysis or cytology. Macovski [MAC83] cites these applications with indepth discussions on each. He mentions that medical imaging, using techniques other than that requiring X-Rays, has arisen out of the limited performance and the destructive capabilities of these X-Rays.

Most standard signal and image processing techniques are dependent on the use of Fourier Analysis. This allows certain characteristics of the image to be determined, so that conclusions can be drawn as to the nature of operation required. This commonly involves the image transform, such as the Fast Fourier Transform [BRA78] [ORA74], or related transforms [GON77], which are usually two-dimensional for images. These image transforms allow the image to be viewed in either the spatial domain, which is the real world picture, or the spatial frequency

domain, that is, the components of light frequency making up the image. It is often more convenient to use an image in the transformed domain when performing operations such as convolution, as this is a highly computationally intensive algorithm involving repeated multiplications. In the transformed domain, a single multiplication can be used [BRA 78], thereby saving computing time. The image can then be simply transformed back to the spatial domain for the result to be seen.

Thus, considering the applications of image processing and the techniques available, all being computationally intensive and demanding on processing power, it is important to investigate efficient computing systems.

1.1. Image Processing

Realizing that the fundamental problems in image processing are the high volumes of data processed and the computationally intensive algorithms, the trend is to look for computer architectures capable of processing more quickly.

Image processing, being a visual subject, has created the demand for the interactive ability of the process. Thus, processing in *less than a day* [JON86], which was acceptable in traditional image processing environments, is no longer so. Silva [SIL89] indicates that conventional architectures cannot cope in the image processing environment because of the amount of data and processing power needed. This is pertinent to the von Nuemann architecture, common to most general purpose computers, not having the power required for image processing applications.

Apart from the need for more powerful processors, the technological development in VLSI chip architectures have made architectures available which are dedicated to the signal processing domain of digital computing. Digital signal processing architectures are frequently used in the image processing environment. Peled [PEL78] describes an image processing facility using a low cost, high speed Signal Processor - the Simple Signal Processor (SSP). This system, however, still requires an interface to a fairly large computer, a minicomputer, to act as a resources manager to this signal processor. Claims of about four seconds for a Discrete Fourier Transform of about 420X420 pixels are made by the author. This is indeed a good figure, although the system described is dedicated to this task. Several other authors cite the use of digital signal processing chips configured as systolic arrays [BRU87] for use in the image processing environment. There is still, however, the need for an architecture which offers generality in terms of processing capabilities, yet which is appropriately powerful to meet the needs if image processing.

1.2. Parallel Processing

Unger suggested, as far back as the early 1960's, that the natural architecture for image processing and recognition is a two-dimensional array of processing elements [ROS83]. This introduced the concept of parallelism in the image processing environment. The idea here is that each processor is responsible for a pixel of the image, and neighbouring processors are responsible for neighbouring pixels. Developments subsequent to this have exploited the concept

of arrays of processors. However, this has not been done specifically for the image processing domain of computing. The ILLIAC III used a 36X36 processor array [ROS83]. With larger numbers of processors, the idea of supercomputing power comes to the fore. Other supercomputer architectures have also emerged, namely the ILLIAC IV, ICL DAP (Digital Array Processor), CRAY-1 and the Connection Machine. Jesshope [JES89] introduces these architectures in a discussion on parallel computing.

Each of these parallel computers, employing multiple processors, comprises an architecture which can be classified according to Flynn's taxonomy [QUI88]. Before discussing these classifications, it is important to look briefly at parallelism and the views and definitions offered by some authors. Duff [DUF78] states that parallel processing is an ill-defined concept. He says that for some, parallelism implies little more than a duplication of arithmetic units, or a provision for simultaneous operations of various parts of an otherwise conventional computer architecture. For others, he states that parallelism is represented by *pipelining*, where data streams through successive computational units, each carrying out its own particular operation on the data. It is important, thus, to look at the differences between parallelism and pipelining. Quinn [QUI88] gives the definition of parallel processing as: *a kind of information processing which emphasizes the concurrent manipulation of data elements belonging to one or more processes, solving a single problem*. This concurrency can be achieved by two methods, namely pipelining and parallelism. Pipelining divides the computation into a number of steps to increase concurrency. Parallelism, on the other hand, uses multiple resources to increase concurrency. There is thus a distinct difference between pipelining and parallelism.

Quinn [QUI88] and Skillicorn [SKI88] both discuss Flynn's taxonomy (1966) used in classifying parallel architectures. Two important classifications to consider are the **Multiple Instruction Multiple Data (MIMD)** architecture and the **Single Instruction Multiple Data (SIMD)** architecture. According to Quinn [QUI88], Flynn categorizes an architecture by the multiplicity of the hardware used to manipulate instruction and data streams. Briefly, these two architectures can be defined as follows [QUI88]:

SIMD: Processor arrays fall into this category. A processor array executes a single stream of instructions, but contains a number of processing units, each capable of fetching and manipulating its own data. Hence in any time unit, a single operation is in the same state of execution on multiple processing units, each manipulating different data.

MIMD: This contains most multiprocessor systems. This is reserved for multiple CPU computers designed for parallel processing; that is, computers designed to allow efficient interactions among the CPU's.

Quinn [QUI88] indicates that Flynn's classification scheme is too vague for strict labelling of supercomputers. He cites several authors who have labelled the CRAY under different classifications.

It is true to say that the processing power of parallel processing architectures has displayed impressive results, not only for image processing, but in general computing as well. Apart from the supercomputer technology displaying parallelism, other systems exist which comprise arrays of processors. Rosenfeld [ROS83] discusses an implementation of cellular arrays, used specifically for image processing. Duff [DUF78] shows the implementation of the CLIP4 which uses a cellular array architecture, also used for image processing. Here the cellular array is a two-dimensional array of processors, or cells, each being able to directly communicate with its neighbours. This is in accordance with the statement made by Unger [ROS83] concerning a two-dimensional array of processing elements being the natural architecture for image processing. Rosenfeld [ROS83] fails to mention that the cellular arrays display a distinct limitation in speed because of communication delays; these systems are also costly.

Kung et. al. [KUN87] describe a wavefront array processor. Their motivation for this topology is twofold, namely: most signal and image processing algorithms can be decomposed into computational wavefronts that can be processed on pipelined arrays. Also, the supervisory overhead incurred in general purpose supercomputers often makes them too slow and expensive for real-time signal and image processing. Kung et. al. [KUN87] criticize the systolic array for signal and image processing on the grounds that they are controlled by global timing reference *beats*. This leads to difficulties in synchronizing large systolic arrays.

Fortes and Wan [FOR87], on the other hand, describe the systolic array as being ideal where high computational throughput is required. This lends itself to image processing, where a high throughput is necessary for fast computation. The high throughput is attributed to the pipelined computations taking place along all dimensions of the array.

Digital signal processing architectures have also been configured for parallel processing. The authors Lang et. al. [LAN88] discuss such an architecture used for high speed digital signal processing.

Smaller array processors have also emerged which show a reasonable processing power, such as the VAX series of computers. Recent architectures, completely inline with the von Nuemann philosophy, such as the INTEL 80386 and the MOTOROLA 68020, display processing powers matching that of some of the VAX processors. Bowman [BOW87] summarizes the processing powers of these recent architectures and the CRAY-1 as:

68020	2-4 MIPS
80386	4 MIPS
VAX 11/780	1 MIPS
CRAY-1	400 MIPS

The von Nuemann bottleneck [OBE88] [BOW87] has brought about the realization of the concepts of parallel processing to computing. This bottleneck is inherent in the von Nuemann architecture (sequential) where there is a single data and instruction pathway between the

processor and memory. This accounts for a slow and inefficient use of the conventional sequential processor. A major problem in parallel computing is the effective operation of more than one processor at a time. There are two methods of realizing multi-processor systems. One is to have the CPU's share a common memory, thereby using a common bus, known as a shared memory system. The second is to allow each CPU to have its own memory, and a high speed interconnection network for the CPU's to communicate on, commonly known as a distributed memory system. These systems are also referred to as highly and loosely coupled multi-processor systems, respectively [HWA85]. Both Stein [STE88] and Obermeier [OBE88] indicate that these formats of multi-processor system lead to several problems. These include the arbitration amongst CPU's, causing contention on the bus as a result of requests for simultaneous use. The bus bandwidth is also saturated. These problems require intervention by introducing expensive hardware such as bus arbiters and cross bar networks [HWA85] into the multi-processor system. There is still, however, a limit to the number of processors that can be added to such a system.

Increased performance can definitely be achieved by parallel processing. However, due to the financial constraints placed on the availability of the supercomputer technology, such as the CRAY-1, as well as the hardware constraints introduced in coupling processors on a common bus, a new technology is sought.

1.3. The Transputer

The design of the transputer overcomes some of the problems present in parallel processing systems based on bus connected multiprocessor devices.

The transputer is a microcomputer designed with parallel processing in mind. It is based on occam's model of concurrency [BUR88], which is in turn based on Hoare's CSP (Communicating Sequential Processes) notation [HOA85].

The transputer can be considered to be a high speed microcomputer, displaying common features such as internal memory (2K on the T414 and 4K on the T800) and central processing unit. The T800 transputer has an on-board 32/64 bit floating point processor. Common to all transputers, and a feature enhancing the transputer's appropriateness in parallel processing environments, are the four high speed serial inter-process communication links. These links are currently rated at 10 $MBits^{-1}$, but recent developments indicate speeds of 20 and 30 $MBits^{-1}$ [CAM88]. This makes the transputer a multicomputer building block, designed with specific intent for the concurrent environment. Communication between processors takes place as synchronized point-to-point communication on the inter-processor links. Dinning [DIN89] compares and summarizes synchronization methods for parallel computers, highlighting this synchronous point-to-point message passing ability of the transputer, eliminating the possibility of multiple processes blocking on the same channel. Rosenfeld [ROS83] indicates that the communications problems inherent in multi-processor systems are overcome by the transparency offered in the transputer's links and the ability to add processors without saturating the communications bandwidth. The links and the message passing method of inter-process communication allow the transputer to be connected in

a number of hardware configurations [ATK86]. The ability to connect the transputer in various topologies allows the granularity of the concurrency and the load balancing to be adjusted and *tuned* to suit the algorithm. These concepts are addressed by Wexler and Prior [WEX89], who advise that the granularity is a problem peculiar to the conversion of existing sequential code into concurrent methods.

The T800 transputer displays features which prove valuable in the context of this project. The on-board full IEEE floating point processor is well suited to the computationally intensive Fourier algorithm implemented here. INMOS [TRA88] make comparisons of the floating point performance of the T800 against several other processors. These are summarized below. Floating point performances are measured in Whetstones.

Processor		**Whetstones/second Single Length**
INTEL 80286/80287	8 MHz	300K
IMS T414	20 MHz	663K
NS 32332-32081	15 MHz	728K
MC 68020/68881	16/12 MHZ	755K
ATT 32000/32100		1000K
IMS T800-20	20 MHz	4000K[1]

Furthermore, the enhanced design of the four serial links on the T800 as opposed to the T414 [BUR88][TRA88] prove to be a valuable asset in aiding the fast distribution of the large magnitude of data inherent in image processing.

Although program development in this project is done in occam[2], under the Transputer Development System (TDS), compilers for other high level languages such as Pascal, Modula-2, Fortran and C are emerging [CLA88][CUL88]. Enhanced operating systems such as Helios [HEL89] are also being introduced. These systems will allow for faster program development.

1.4. Overview of the Project

With the above background in mind, the following can be stated about the aim of this thesis. The research concentrates primarily on a feasibility study involving the setting up of an image processing workstation. As broad as this statement concerning the workstation may seem, there are several factors limiting the extent of the research. This project is not concerned with the design and implementation of a fully-fledged image processing workstation. Rather, it concerns

---

[1] This result was obtained using onboard memory for both code and data. If code and data are stored in external memory, the same whetstone test yields approximately 1000K.

[2] The latest version of occam is occam2 [BUR88a][JON89a].

an initial feasibility study of such a workstation, centered on the theme image processing aided by the parallel processing paradigm.

In looking at the hardware available for the project, in the context of an image processing environment, a large amount of initial investigation was required prior to that concerned with the transputer and parallel processing. Work was done on the capturing and displaying of images. This formed a vital part of the project. Furthermore, considering that a new architecture was being used as the work horse within a conventional host architecture, the INTEL 80286, several aspects of the host architecture had also to be investigated. These included the actual processing capabilities of the host, the capturing and storing of the images on the host, and most importantly, the interface between the host and the transputer [COO89]. Benchmarking was important in order for good conclusions to be drawn about the viability of the two types of hardware used, both individually and together.

On the subject of the transputer as the workhorse, there were several areas which required investigation. Initial work had to cover the choice of network topology on which the benchmarking of some of the image processing applications were performed. Research into this was based on the previous work of several authors, which introduced features relevant to this investigation. The network used for this investigation was chosen to be generally applicable to a broad spectrum of applications in image processing. It was not chosen for its applicability for a single dedicated application, as has been the case for much of the past research performed in image processing [SAN88] [SCH89].

The concept of image processing techniques being implemented on the transputer required careful consideration in respect of what should be implemented. Image processing is not a new subject, and it encompasses a large spectrum of applications. The transputer, with image processing being highly suited to it, has attracted a good deal of research. It would not be rash to say that the easy research was covered first. The more trivial operations in image processing, requiring matrix type operations on the pixels attracted, the most coverage. Several researchers in the field of image processing on the transputer have broken the back of this set of problems. Conclusions regarding these operations on the transputer returned a fairly standard answer[3]. An area of image processing which has not produced the same volume of return as that concerning the more trivial operations, is the subject of Fourier Analysis, that is, the Fourier Transform. Thus a major part of this project concerns an investigation into the Fourier Transform in image processing, in particular the Fast Fourier Transform. The network chosen for this research has placed some constraint upon the degree of parallelism that can be achieved. It should be emphasized that this project is not concerned with the most efficient implementation of a specific image processing algorithm on a dedicated topology. Rather, it looks at the feasibility of a general system in the domain of image processing, concerned with a highly computationally intensive operation. This has had the effect of testing the processing power of the hardware used, and contributing a widely applicable parallel algorithm for use in Fourier Analysis.

---

[3] These are discussed more fully in Chapter 2, which covers the work related to this project.

The results of the investigation are presented along with a discussion of the methods throughout the thesis. The final chapter summarizes the findings of the research, assesses the value of the investigation, and points out areas for future investigation.

Chapter Two

2. Related Work

This project is concerned primarily with an investigation into a cost-effective workstation for image processing. The full design and implementation of a complete workstation is not considered. The central theme of the research is an investigation into the use of transputers as the workhorse to enable suitable cost-effective processing power to be obtained. What is sought, is a general purpose system using a network of transputers, with an INTEL 80286 machine as the host, whose topology allows for a broad spectrum of image processing operations to be executed, by exploiting parallel processing (inherent in the transputer) for maximum gain. The type of workstation that a fully-fledged design project of this nature would be intended to mimic would be one such as the system described by McManis [MAN87] which is an inexpensive but powerful image processing workstation implemented on the Amiga 1000 computer. This differs from the system investigated in this project, in that the transputer has been chosen as the accelerator for the host machine whereas the Amiga 1000 is used for all the processing. The type of parallelism that is to be implemented in this project for the image processing operations is that of *image parallelism* [MOR88a]. The reasons for choosing this type of parallelism will be justified during the course of this thesis.

This chapter of the thesis deals with work related to this project, to show areas of overlap and to provide an indication of the general trend in the subject area concerned. Concerning the subject of parallel processing, several architectures are looked at to justify the use of the use of the transputer for this project. Work covering the implementation of image processing systems, both on transputer networks and other architectures is also discussed.

The bottlenecks imposed by the von Nuemann architecture are discussed by many authors dealing with the subject of parallel processing and parallel architectures. This is especially prevalent in the domain of image processing and other areas of computing where large amounts of data need processing [BAT86] [BOW86]. Several very large systems have been developed specifically for image processing. One such system, the Massively Parallel Processor (MPP), is described by Potter [POT83]. This system uses the **SIMD** architecture, having 16 thousand processors to provide over six billion eight bit adds and 1.8 billion eight bit multiplies per second. This is used primarily for highly computationally intensive pattern recognition algorithms. These processors are connected in a grid topology, thereby allowing processors to communicate with their neighbours. It is also suited to the direct mapping of images (also matrix in format) onto the processors. Duff [DUF78] describes a **SIMD** system which displays a high degree of parallelism. This system, the CLIP4, uses special purpose processors dedicated to each element of the two-dimensional data field. A total of 92X92 processors are used for an image of the same size. Architectures of this sort of magnitude are clearly very impressive. In the scope of this project, however, it merely demonstrates the sort of processing power that can be achieved using parallel architectures and the topology suitable for image processing. The specialized areas of application

of systems like these are not in keeping with the idea of a general purpose system, as is sought for this project. Duff's [DUF78] system, nonetheless, demonstrates some very good ideas for processing images, as well as the magnitude of architecture available for image processing.

Several other large scale image processing machines are described by Kidode [KID83], these being the image processing machines used in Japan. All these systems described have highly complex architectures, mostly parallel, and are definitely not cost effective solutions to small scale image processing requirements. An important point Kidode [KID83] makes, is that if a general purpose digital computer is to be used for image processing, those operations which are least expensive to implement must be determined. This has probably influenced most researchers to implement large workstations for image processing, as a result of the need to solve certain high level tasks such as fingerprint recognition and object analysis, rather than concentrating on smaller low level image processing operations. It is interesting to note that most of these authors have introduced the transputer and mentioned its suitability to image processing. Sleigh, Radford and Harp [SLE88], for instance, look at architectures for computer vision, making comparisons of the DAP architecture and transputer arrays. They indicate that the transputer and its programming language occam offer a high degree of flexibility when engineering algorithms and hardware for computer vision. The same applies to image processing in general. Also, the ability of the transputer arrays to operate in both **SIMD** and **MIMD** configuration, enables the arrays to be used efficiently. This, as well as factors such as ease of programming (using *image parallelism*), data distribution and cost effectiveness, has lead to the transputer being used as the processor in this project.

It is important to look at what sort of processing power is required for some image processing operations. If considering that a display of the image is required to be in real-time, as in motion pictures of say 30 frames a second, then the power requirements are dramatically increased. As an example of the volume of data that would be required for this display rate, consider an image of 256X256 pixels at a resolution of one byte per pixel. This would require approximately 2 $MBytes^{-1}$ of data to be moved in a second. Most image processing applications don't require this amount of data to be displayed at this rate. Visual inspection operations for industry do, however, require some real-time update of the images. To gain an appreciation of the sort of processing power required, the following example is presented to show the requirements for a typical industrial application using an image size of 256X256 pixels at 5 instructions per pixel operation (including data movement) with a throughput of 10 frames per second.

The method of obtaining the processing speed required can be shown by the following formula indicated by Bowman [BOW87]:

$$MIPS = (n_1 x n_2) x I x M x N x F x 10^{-6}$$

where

MIPS = Million of Instructions per Second

$n_1 x n_2$	=	Kernel size (1x1 for point operations)
I	=	Number of instructions per pixel
MxN	=	Number of pixels
F	=	Frame rate required

Bowman [BOW87] shows that the following figures are the requirements for the operations shown below:

Point operations (thresholding, negation)	3 MIPS
3x3 Filter	30 MIPS
Sobel Edge Operator	50 MIPS

From this, it is easy to see that the processing power requirements for typical vision algorithms when 10 to 20 of these operations are required is in the range of 100-1000 MIPS. This sort of requirement can be fulfilled by architectures such as the CRAY-1 (400 MIPS) and the T Series Hypercube of 1000 MIPS. These are both very costly systems to implement.

Bowman [BOW86] [BOW87] and Batchelor et. al. [BAT86] address this problem by introducing the various concepts of parallelism and pipelining as a solution to the von Nuemann bottleneck. They discuss the KIWIVISION [BOW87a] system implemented as a low cost system, targeted at high speed industrial applications. This system is based on the Motorola 68000 Microprocessor and makes use of various image stores and a processing pipeline. This system still, however, has limitations in that it is dedicated, with a set processing power, offering no generality, although individual low level image processing operations can be performed. The authors indicate that these lower level image processing operations usually involve multiple passes through a pipeline of processing modules which must be added for this. An indication is also given of the fact that the high level operations such as Fourier descriptors and classification algorithms are not easily implemented. If implemented, a degree of parallelism is lost. A VME bus is used which limits the connectivity and imposes an upper bound on the processing power. The authors both conclude with an introduction to the INMOS transputer, offering some praise for its capabilities. They cite a performance figure of 30 ms for a 3x3 convolution of a 512x512 pixel image processed on an 8x8 array of transputers implemented by Smiths Associates in the United Kingdom.

Several image processing systems have been implemented using digital signal processing devices. In keeping with the original idea that this project concerns a generalized system for use in an image processing environment, such systems are not considered for implementation. It is nevertheless interesting to investigate some of the work done in this field and to offer some criticism on this, as well as glean information which is of use.

Peled [PEL78] describes a low cost image processing facility, making use of high speed signal processors, known as the Simple Signal Processor (SSP). In terms of actual cost, at present day prices, this system is no longer a low cost one. It makes use of an IBM Series/1 minicomputer

as the overall system resources manager. A Ramtek 9351 512x512x16 bit monitor is used for the display. This is no longer a low cost device. Peled [PEL78] indicates that a 1008 complex Winograd Discrete Fourier Transform (DFT), is executed in about 20 msec. This is impressive, but is a one-dimensional Fourier Transform. It appears that most authors on image processing, implemented on new architectures, tend to avoid the two-dimensional Fourier Transform. Peled [PEL78] indicates that the one-dimensional DFT is a compute bound task and goes on to speculate that a two-dimensional DFT may take about 4 seconds for a 420x420 pixel image, indicating that this is at least an order of magnitude faster than other similar low cost facilities. Two other implementations are indicated on this system; namely histogram equalization and filtering. These results are no more impressive than those which can be achieved by transputer based systems which have been implemented by various authors and will be discussed later.

More recent systems implemented, using digital signal processing chips for image processing, are systems such as those described by Hartenstein et. al. [HAR87a] and Silva [SIL89]. Hartenstein et. al. have implemented the Map Oriented Machine (MOM). This machine is indicated as being a compromise between the purely sequential von Nuemann concept and fully parallel solutions, insofar as the control part has been parallelized and the data manipulation side makes use of sequential access organization. Few actual implementations and results are indicated by the authors. The authors do, however, take a novel approach to the method of operation. Instead of sending the data through an array of processors, as is usually done in systolic array processors, the data is fixed in map organized memory. The authors indicate that this system can solve almost any problem which may be solved by a conventional computer, in principle. They do mention that for non time critical problems it may be better to use a conventional general purpose computer because almost anybody knows how to program it. This systems does not show much promise in terms of generality and flexibility, especially in ease of use.

The system proposed by Silva et. al. [SIL89] makes use of a modular approach to the design, using digital signal processors (the Texas Instruments TMS 32025) in each module. Here the modules are connected using a VME bus, again imposing a bottleneck in terms of connectivity and generality. This system, although described as being a high performance one, is highly specialized, having a dedicated video capture and display system integrated into the hardware design.

Staying on the subject of digital signal processing chips being used in the image processing environment, it is interesting to note that several authors have chosen to combine digital signal processing architecture with the transputer [BRA87] [DOW88] [SAN88a]. The system described by Bramley [BRA87] makes use of the cascadable INMOS A100 correlation chip for digital signal processing. He claims that the transputer is not suited for computationally intensive signal processing. The A100 chip thus acts as the coprocessor to the transputer. Although the system described is primarily intended for signal processing applications, the author describes how the two-dimensional Discrete Cosine Transform (DCT) could be implemented using several cascaded A100 chips. Although impressive results are shown, this system does not allow for generality. The system is further discussed by Sandler and Eghtesadi [SAN88] and Ben-Tzvi and Sandler

[BEN89], where the system is used to perform the Hough transform for shape analysis. This system is described as having the digital signal processing (DSP) chips interfaced to the transputers using the INMOS CO11 link adapter chip. The data, being serial at a rate of 20 $MBits^{-1}$, will introduce a bottleneck in the overall speed considering that the data is transferred, computed by the DSP chips and then sent back, albeit that this DSP chip is dedicated to this type of signal processing operation. This system is similar to that implemented with the A100 signal processing chip [BRA87].

Although the implementations involving digital signal processing chips, as integral components of the complex image processing system, are not suitable for the type of application considered in this project, these devices could certainly be considered for the extension of such a project. Downing and Bennet [DOW88] propose such a system. Their implementation follows the concept of this project very closely, the only difference being that their project is intended for high level feature extraction for object recognition. A system involving quad transputer boards and a CCD (Charge Coupled Device) camera, using a PC as a host computer is discussed. A number of image processing operations have been implemented using *image parallelism* [MOR88a]. Of their results shown, most being for operations specifically applicable to the concept of feature extraction, two are worth noting. To negate an image of 128x128 pixels by 256 grey levels takes 181 ms on one transputer and 49 ms on six transputers. A Laplacian convolution operation using a 3x3 mask for the same dimension as in the previous experiment, takes 637 ms and 131 ms for the respective number of transputers. These results can be compared with the operations performed on the T414 transputer in this project, and against results shown by other authors. The results of experiments carried out on the T414 are shown later in this thesis.

The system implemented by Cok [COK88] is described as a *medium grained* parallel image processing computer. The advantages of this system are listed as being cost effective, flexible in terms of performance and programmability and simple in design. In comparison to many other parallel computers it can be programmed in a pipelined as well as a distributed manner, thereby allowing the choice of granularity to be changed as desired. This is in many respects the same as the system implemented in this project and has offered some interesting ideas for use in this thesis. The applications implemented by Cok range from interactive image manipulation through Mandelbrot set calculations to film chemistry modelling. For the purposes of this project, some of the image processing benchmarks are given below. This is done so that relative performance figures of the system described by Cok can be seen and compared with some of the results in this project, albeit not exactly the same operations. Cok compares several architectures against his results. These are also shown below.

All images are 512x512 pixels of **single colour**. Times are in seconds.

			32 Nodes 20 MHz	
Operation	**Sun 3/160***	**VAX 785**	**T414**	**T800**
Image Boost Mult and Diff	4.1	4.3	0.060	0.060
FFT (Uses FP arithmetic)			13.4	2.1

* - use primarily floating point arithmetic.

Cok [COK88] concludes by indicating that a special parallel implementation of a simple image processing language has been programmed. This, although not immediately considered in this project, would be ideal for a fully implemented workstation. This will be further discussed in the conclusion.

The work done by Morrow et. al. [MOR88] and Crookes et. al [CRO87] is also relevant to this project in certain respects. The system described by these authors indicates a grid array of transputers for image processing with a special purpose language for this, known as the Latin language. This network is described in some detail in the section covering networking in this project (Chapter 5), as it has been adapted for use in this thesis. The Latin language is a set of processes, each having its own array processing capability and can be mapped onto its own array of transputers. Examples of programs are given in the text [MOR88], showing comparisons of the Latin language and occam. There is certainly scope in this project for implementing such a language. Apart from this, the work shown, as well as some of the occam code, has been useful in confirming some of the intuitive ideas considered for the processing of images in this project. This covers the partitioning of the image amongst processors and also the bordering technique in image segments at the edge of the network. This is peculiar to operations such as convolution, although these operations are not implemented in this project. A further publication by Morrow and Perrot [MOR88a] offers a discussion on the concept of *image* and *task parallelism* when implementing low level image processing algorithms. These two types of parallelism are defined as:

Image Parallelism: The image is partitioned over all the available processors with each processor executing a whole algorithm.

Task Parallelism: The image being used is divided into subtasks which are distributed over the processors which can then operate concurrently.

Morrow and Perrot [MOR88a] indicate that a machine which permits both **SIMD** and **MIMD** type processing is suitable for the implementation of a large number of image processing tasks,

ranging from low level local operations to high level global operations. Winder [WIN88] backs this up in his discussion of the dual paradigm architecture, the DisPuter, by saying that efficient hardware architectures must execute all stages of the algorithm efficiently. This can be best achieved by the dual paradigm parallelism. He goes on to state that a truly general purpose machine should exhibit both SIMD and MIMD parallelism. The network chosen in this project offers this in terms of the various ways in which the topology can be exploited. That is, the pipeline, the central theme of this work, and the more complex grid of links in the system. Morrow and Perrot [MOR88a] make some interesting statements which, together with their conclusions, have influenced the way in which the operations are implemented in this project. What is stressed is that efficiency can only be achieved if the load balancing is even. The authors indicate that in *image parallelism* this is not a problem if the image segments are of similar sizes. When introducing parallelism into an algorithm, which was originally sequential, it is important to look at the efficiency that is achieved by introducing this parallelism. This efficiency is related to the number of processors used and is shown by Morrow and Perrot [MOR88a] as:

$$e = \frac{t_1}{n.t_n} 100\%$$

where

t_1 = the time for the sequential version of the algorithm run on a single processor, and

t_n = the time for the concurrent version run on n processors.

The higher the efficiency value, the better the implementation of the algorithm. This efficiency factor is used extensively in the benchmarking tests performed on the Fast Fourier Transform algorithm run on the various size networks in this project. This indicates the overall efficiency of the parallel implementation and determines whether or not there has been any cost effective gain by making use of a generalized network of transputers to implement image processing operations.

Morrow and Perrot [MOR88a] conclude by giving results for some of the low level image processing operations they have implemented, indicating the number of transputers that were used. The tests were performed using both *image* and *task parallelism*. These results are shown below:

		Image Parallelism	Task Parallelism
Entropy Operator		96%	52% (3)
Median Filter		67%	27% (5)
Histogram		13%	37% (3)
Correlation	11x11	95%	44% (4)
	21X21	99%	44% (4)

The figures in brackets indicate the number of transputers used. For the correlation operations, the 11x11 and 21x21 indicate the size of the correlation mask used.

These results indicate that greater gains can be obtained from *image parallelism* (even with a small number of processors), although this is related to the type of operation performed. The authors attribute the low figure for the histogram equalization algorithm, using *image parallelism*, as being due to the fact that it is considered a global operation and thus requires access to the whole image. The low figures for *task parallelism* have been attributed to the following:

1. It is difficult to obtain an equal allocation of work loads for each processor.
2. The algorithm may not have sufficient parallelism inherent in it.
3. The amount of communication required is greater than in *image parallelism*.

The authors emphasize the fact that the solutions for *image parallelism* are more easily produced because the algorithms running on each processor are identical and are straight forward sequential implementations.

It is mainly as a result of the conclusions given by Morrow and Perrot [MOR88a] that *image parallelism* has been chosen as the method of testing the Fast Fourier Transform. This will be covered in detail in the section on the FFT.

To indicate the sort of image processing work that has been done by researchers covering the lower level image processing operations, the work by Sleigh, Radford and Harp [SLE88] and Harp, Palmer and Webber [HAR87] is discussed. Together with other work previously mentioned and the results shown, it is obvious that these low level operations have attracted a great deal of interest in research on the transputer. It is mainly for this reason that the majority of image processing work done in this project covers the Fourier domain of image processing. More specifically the Fast Fourier Transform is dealt with. It is nonetheless important to show the results obtained by these authors as some preliminary familiarization work was done on a T414 transputer and some of these operations were implemented in this project. These results can not only be used for direct comparison, but also allow one to get an overall appreciation of the sort of processing speeds that can be achieved. The results below show some of the comparisons of speeds obtained for the VAX, DAP and 16 node T414 based machines tested by Sleigh, Radford and Harp [SLE88] for the Sobel operator on an image of 128x128 pixels for the VAX and DAP

and 256x256 pixels for the T414 based machine. All times shown are in seconds.

VAX	DAP	T414
1.9	0.016	0.203

The authors mention that each T414 processor has a 64x64 pixel subimage, thereby implementing *image parallelism*. They suggest the use of a T800 to gain the advantage of concurrent I/O and processing capability, and predict a possible speed up of up to 40 times. As only one T414 transputer was available during this project, the initial familiarization work mentioned earlier was carried out on this architecture, and the subsequent experimental work on the Fast Fourier Transform and the network was carried out on the T800, as there were a number of these processors available for use.

Some of the results shown by Harp, Palmer and Webber [HAR87], based on the exact hardware implemented by Sleigh, Radford and Harp [SLE88] are shown below.

The operations were implemented on a network of 4x4 T414B transputers, quoted by the authors as running at 15 MHz and rated at 7.5 MIPS. The times are shown in milliseconds.

	Image Sizes		
Function	**128x128**	**256x256**	**512x512**
Low Pass Filter	36.2	144	593
Grow (Scale)	7.5	24	100
Contrast Stretch	22.8	92	380

Apart from the results above, the work done by Harp, Palmer and Webber [HAR87] is related to the work in this project in respect of the idea of a network of transputers connected to a host computer (IBM PC). Their system is somewhat more advanced as it makes use of a framestore and image display device interfaced directly to the network. This is discussed in more detail later in this thesis as an area of possible expansion in working towards a fully implemented workstation.

In looking at alternatives to the network topology employed in this project, the Perfect Shuffle Machine (PSM) such as that implemented by Schomberg [SCH89] offers a good alternative. This is especially so for the implementation of the Fourier Transform, as Schomberg discusses the implementation of such. He, however, does not show implementation results. As this is directly relevant to the topology chosen for this project and the Fourier Transform implemented, it is discussed in more detail in the appropriate section of the thesis (Chapter 5).

This discussion has covered several architectures which deal with the subject of parallelism and

image processing. It is noted that several authors have introduced the transputer, the architecture chosen for this project, as a good processor for the application of image processing. The central theme of seeking a general purpose, cost effective solution for small scale image processing was considered during the survey of work covered by researchers in the field of image processing and parallelism. It is evident that most of the work covered by others involves low level operations in image processing, such as intensity scaling and filtering. In contrast, the work in this project is concentrated on looking at the effectiveness of the topology chosen. It is also concerned with the Fast Fourier Transform as this is a computationally intensive algorithm and provides an upper bound on the processing requirements of such a system, thereby allowing one to establish the cost effectiveness of the system under investigation.

Chapter Three

3. Development Work Required to Support the Investigation

Although this project is concerned with a feasibility study into an image processing workstation using transputers, some investigation into areas other than that of the transputer was necessary. This work included a study of the capture and displaying of images and the development of software to perform the interface between the host (AT) and the transputer. This chapter discusses these and related issues.

3.1. Linking the Host and Transputer

In order to use the transputer so that it acts as a standalone system, that is, without the Transputer Development System (TDS) running simultaneously with a program on the transputer, a program has had to be developed so that data can be transferred between the host and the transputer. In this project the host referred to is the 80286 machine.

The Transputer Development System (TDS) allows for occam[4] programs to be run as bootable programs, on single transputers as well as on networks of transputers. For this, a communications link has to be created between the host and the transputer (root transputer) along which the data can be transmitted or received. Although a file server is available under TDS for standalone programs, for the purposes of this project, it has been necessary to create a file server with features which are easy to utilize. Such features include file handling and good screen manipulation, as well as the tasks of transferring data, enabling the host computer to undertake more specialized computations, and I/O. This is made possible by writing a program on the host in a conventional high level language, with all the standard file and screen handling facilities necessary, and then simply introducing calls to the file server procedures allowing communication with the root transputer to take place.

The program developed is LinkIO.PAS[5], used as a unit in Turbo Pascal (v4.0). The code for LinkIO.PAS appears in Appendix A. This unit can be included in any host program which must communicate with the transputer. As the link adapter chip (INMOS IMS C011) on the transputer (T800 or T414) board, has only a byte wide output onto the bus of the host, the data for any communications has to be sent as packets of single bytes. These packets have the start and stop bits built in by the link adapter chip, totalling 11 bits for every byte transferred. Procedures have thus been developed to allow bytes, 16 bit integers and 32 bit integers to be transmitted to, and received from the transputer. No real number values have been catered for' as the version of

---

[4] The version of occam used in this project is occam2.

[5] Versions are also available in Turbo C (v2.0) and FST Modula-2 in Technical Document 89/12, Department of Computer Science, Rhodes University.

Turbo Pascal used allows only 6 byte real values when no numeric coprocessor is used. The transputer, however, follows the IEEE 754 standard of allowing 4 and 8 byte real values to be used[6]. A method of overcoming this problem, without having to purchase a numeric coprocessor or a new version of the language, would be to write a procedure to perform the conversion of 6 byte real values to 4 and 8 byte values and vice versa, such that these values can be transmitted to, or received from the transputer. The conversion will have to take care of the fact that the mantissa and the exponent of the 6 byte real values is not the same as that for the 4 and 8 byte values which follow the IEEE 754 standard.

In this project the values that are sent to the transputer are converted to real values on the transputer once received from the AT. Once the results have been obtained, the values are converted back to byte or integer values as they represent intensities. The values are converted to real values only for those operation requiring real values, such as the Fourier Transform.

When the occam program is compiled under TDS as a bootable file, it is written to the host as a series of bytes. The LinkIO unit reads these bytes, transmits them to the transputer (root transputer) and then resets the transputer or transputer network. The procedures and functions for the transmission and reception of data then become available for use. All these procedures and functions have been written as inline code so as to allow for the fastest possible data transfer rate. However, this data transfer rate is limited by the bandwidth of the link adapter chip and the transputer link.

The LinkIO.PAS unit for data transmission introduces some strict rules which must be followed when running a host program in conjunction with a program on the transputer.

1. A point of transmission in one program (host or transputer) must have a corresponding point of reception in the other program.

2. The data type being transmitted or received must match on both the host and the transputer, i.e. if a 32 bit integer is being sent from the host to the network, the network must receive this value into a variable declared as a 32 bit integer.

3. A channel, in occam, must match exactly the data type being transmitted or received. That is, if a 32 bit integer is being sent from the host then the channel into which the value is being received on the transputer can only be of this type. To accommodate several data types, a *CHANNEL OF ANY* can be declared, or alternately, the variant protocols in occam2 may be used and a channel declared of such a type. Again, strict point to point communication of the same data type at transmission and reception must be adhered to.

---

[6] The later version of Turbo Pascal, version 5.0, allows for numeric coprocessor emulation, thus incorporating the 4 and 8 byte reals. With this facility available, procedures could easily be written to allow the transfer of real values in accordance with the IEEE 754 standard.

A short example program follows, showing a host program (in Turbo Pascal (v4.0) on the AT) communicating with an occam program on the transputer.

Turbo Pascal program

```
program TestLink;

(* A simple test program *)

uses LinkIO; (* Unit for the communications *)

const
  ValToSend = 5;

begin
  ByteToRoot (ValToSend);  (* Send a byte to the transputer *)
    if LinkDataAvailable then
      writeln (ByteFromRoot);
end. (* TestLink *)
```

occam2 program

```
PROC TestLink ()

-- Simple test program

CHAN OF BYTE ChanIn, ChanOut :
PLACE ChanIn AT 4, ChanOut AT 0 :
BYTE Value :

SEQ
  ChanIn ? Value   -- Get value from the host
  ChanOut ! Value  -- Send it back
:
```

3.2. Graphical Display Devices and Related Hardware

As image processing is largely an interactive subject, images are best analyzed when displayed on graphical display devices. It has thus been necessary to investigate these display devices to make a suitable choice of device for this project.

In the domain of the host architecture used in this project, several graphical display devices are available. Initially, only the Colour Graphics Adapter (CGA) and the Enhanced Graphics Adapter (EGA) [WIL85] were available for use in this project. A Video Graphics Adapter (VGA) [WIL88] system has subsequently become available, thereby allowing some interesting comparisons to be made as well as some good conclusions to be drawn.

The Colour Graphics Adapter (CGA) can be regarded as the pioneering device in terms of graphical display devices for PC type architectures. The CGA provides resolutions of 320x200 pixels in four colours and 640x200 pixels in two colours. When compared to later developments in graphical display devices, the resolutions indicate that the CGA is not suitable to a project such as this, where a reasonably high resolution display device is required.

The Enhanced Graphics Adapter (EGA), however, has an improved display resolution. This

adapter is capable of displaying 640x350 pixels at 16 colours, from a choice of 64 displayable colours. In the early stages of this project, investigation of graphics display devices, showed only these two colour devices to be available, the EGA obviously being the better of the two.

Lawrie [LAW88] an Honours student in the Department of Computer Science at Rhodes University, performed some pioneering work in the department on the EGA. At that stage very little was known about the device, and very little literature regarding the programming of the device was available. Lawrie's [LAW88] investigations provided some very good routines for displaying digital images on the screen. It was discovered that the only way to acquire a good, fast image displaying routine was to avoid the BIOS calls for pixel display. Lawrie provided routines written in assembler, which took control of the hardware on the EGA, thereby allowing faster dumping of images (approximately 6 seconds for a 256x256 pixel image at 16 colours).

The conclusions drawn by Lawrie in his comparison of the EGA versus the CGA are that the EGA is a far superior display device. The EGA can emulate the CGA as well as displaying features unique to itself. A good feature, showing the power of the EGA, is that of the splitscreen facility. Lawrie provided routines which can be used in this project, two of which are particularly applicable to image processing. These routines are as follows:

1. *Rotation of the colour lookup table.* This allows the initial colours assigned to the intensities to be swapped with other colours as the lookup table is rotated. This rotation of the false colouring is good, as areas of detail are sometimes easier seen when displayed in different contrasting colours.

2. *Splitscreen facility*: A routine has also been provided which makes use of the splitscreen facility. This is ideal for simultaneously viewing both the initial image and one on which an image processing operation has been performed, thus allowing immediate comparison of the old and new image to be made.

Considering the relative speed at which images can be displayed on the PC, using the routines provided by Lawrie, the following points should be carefully considered:

1. The page limit inherent in the INTEL range of processors could create a problem with the size of image which can be displayed. Lawrie's investigation was performed on image sizes of 256x256 pixels of 8 bits per pixel (Totalling 64 KBytes of memory). Should larger images need to be displayed, the problem of the 64 KByte page boundary could be overcome by using dynamic data structures. This is discussed in the section relating to some of the work carried out on the AT. This will, however, introduce further time constraints in the speed at which the image is displayed, due to the manipulation of the links.

2. The development of the LinkIO.PAS unit, for use with standalone programs running on the transputer (discussed previously), has shown a distinct bottleneck in the data

transfer bandwidth. It takes approximately 2.75 seconds to transfer a 64 KByte block of data, representing a 256x256 pixel image at 8 bytes per pixel, from the AT memory to the transputer. Considering the relative performance of the display routine, when an image is required to be displayed from the transputer, the time taken for this data transfer must be taken into account. The bottleneck displayed by the link between the PC and the transputer introduces the concept of the video boards available for the transputer (VIDEOPUTER). This issue will be discussed separately in section 3.3.

In looking at the VGA as the standard graphics display device for this project, several new and interesting features can be noted. Jordaan [JOR89], an honours student in the Department of Computer Science at Rhodes University in 1989, has performed some investigative work into this adapter as an update on the EGA. Although a drop in resolution is noted, from 640x350 pixels at 16 colours on the EGA, to 320x200 pixels at 256 colours on the VGA, the resolution is still good for the quality of images captured. This has the added advantage of having 256 colours when false colouring is required. Also to be considered is the fact that the EGA can only display 4 shades of grey, while the VGA is a vast improvement, being able to display 64 shades of grey. Images displayed at 320x200 pixels using the 64 grey scales are very impressive indeed. Again, along the lines of the routines provided by Lawrie [LAW88], Jordaan [JOR89] has implemented routines to provide similar display features.

3.3. Towards Faster Displaying of Images

Due to the relatively slow speed in displaying images and the further delay introduced if the image is sent from the transputer, methods of improving the display time should be considered for a fully fledged workstation. There are two schools of thought regarding improvement in the display time.

3.3.1. Graphics Coprocessor

Graphics coprocessor chips, such as the INTEL 82786 and the Texas Instruments TMS 34020 Graphics Systems Processor, discussed by McNiery [NIE86], are available, offering ideal methods of enhancing the displaying power. These systems would be PC based and thus totally separate from the transputer. However, this would still not overcome the bottleneck existing in the link between the host and the transputer.

Gandhi [GAN88] indicates that the INTEL 82786 graphics coprocessor can achieve a bus bandwidth of 40 MBits per second, thereby allowing drawing speeds of 2.5 MBytes per second at 8 bits per pixel. This is due to the chip acting as a coprocessor to the main processor and the fact that there are two separate and independent processors on the chip. One of these is for drawing and the other for displaying. In this case, the displaying processor would be of use here. This coprocessor can also access system memory directly, thus allowing fast display speeds to be achieved, ideal for a project such as this.

3.3.2. Video Transputer Systems

As the bottleneck present in the link adapter chip (INMOS IMS CO11) and the host bus cannot be overcome, alternative methods of displaying images from the transputer should be considered. Video Transputer systems, such as the graphics system proposed by Nicoud and Schweizer [NIC89], are available. Here, the video output is interfaced directly to a transputer which has dual ported RAM. Although the system proposed by Nicoud and Schweizer [NIC89] is primarily intended for a graphics environment, such a system does not have to be used strictly for graphics per se. The desired part of the architecture would be that part interfaced to the video display device, also linkable to a transputer network.

Harp, Palmer and Webber [HAR87] proposed an image processing configuration which includes both the frame store (image capture device) and display devices connected to the transputer network being used. In this case, the transputer network topology is a mesh configuration, thereby following the matrix structure of images and allowing for easy distribution of the image amongst the processors. This network configuration with the capture and display devices is shown in Figure 1.

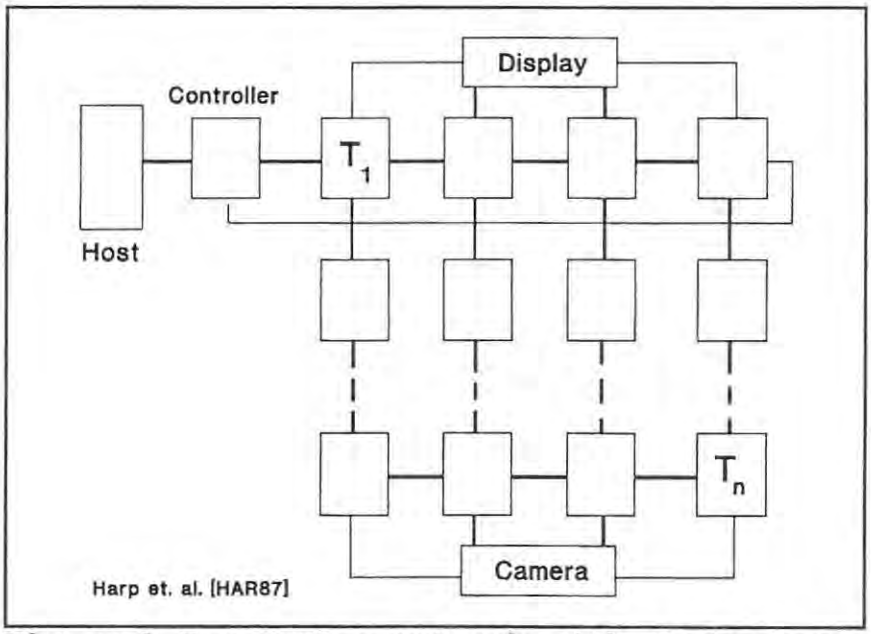

Figure 1 Transputer network with capture and display devices

By looking at the links between the frame store and display boards in Figure 1, it can be seen that these are easily connected into an existing network. Also, having the four links, each going to a separate column, allows for quick distribution of the captured image, and quick gathering of the processed image.

In the context of this project, such systems are not considered for immediate implementation.

Rather, they are meant as an introduction of ideas for further enhancement of a proposed image processing workstation. However, in the case of a fully implemented workstation, it would be advantageous to incorporate the features discussed above. Should image processing operations be performed on both the host and the transputer network, both the graphics coprocessor and the video transputer board could be included.

3.4. Image Capturing

In order to have images available for processing, an image capture device is necessary. In the course of this work, two image capture units were used. Capturing images for image processing requires a Charge Coupled Device (CCD) camera and a compatible frame grabber. Alternatively the images can be captured using an ordinary television camera with hardware for the conversion of the analog signals to digital ones. It is important to understand the format of a digital image when discussing this topic, fundamental to digital image processing. A brief description of this area follows:

Inherent in all image processing/imaging environments is the concept of the digital image. Digital images are captured by means of devices which read the intensities of the light as analog signals and convert these to digital values using Analog to Digital converter hardware. These digital images comprise individual intensities which are the picture elements, or more commonly the pixels. Gonzalez and Wintz [GON77] describe an image as being a two-dimensional light intensity function $f(x,y)$, where x and y denote spatial coordinates and the value of f at any point (x,y) is proportional to the brightness (or grey level) of the image at that point. A digital image is thus an image which is discrete in both the spatial coordinates and brightness. Digital images are usually captured in a matrix format of MxN picture elements, otherwise known as pixels or pels. The intensities are quantized into a range of integer values determined by the number of bits allocated to each pixel. Commonly, 8 bits per pixel are used, giving an intensity range of 0 to 255. The number of quantization levels allocated to the pixel intensities determines the resolution of intensity. Although a greater number of levels provides a higher resolution of intensity, this results in a greater storage requirement. Higher resolution systems also require more complex, and more costly, hardware for capturing images. In the case of an image of low intensity resolution, for example 4 bits per pixel, 16 grey levels of intensity will be available, going from black to white in increasing order of intensity. This low intensity resolution image will appear harsh to the eye. Higher numbers of intensity levels provide a smoother effect.

The first camera and frame grabber board used in this project, was the PC-EYE system, which allows images of varying size, up to 600x400 pixels, to be captured. This CCD camera and frame grabber produce images with pixels 6 bits wide, thus providing intensities in the range 0 to 63. With this intensity scale, it is difficult to capture very good images unless the lighting conditions are very good [COO88]. The reason being that the information required ends up in a very small

range within the 0 to 63 range. The rest of the intensity range is covered by noise from background lighting. In order to overcome this problem, as well as to gain the full benefit of the 0 to 255 intensity range available in data stored at 8 bits per pixel (rather than 6 bits per pixel within each byte), four images must be added together. This results in a slightly improved image with more detail over a wider range of intensities.

The second capture device used in this study offers a far superior quality digital image. The Data Translation DT8251 High Resolution Frame Grabber used in conjunction with a Colour CCD camera, produces images of 512x512 pixels at 8 bits per pixel. This allows an immediate intensity range of 0 to 255, thereby providing good detail. Several colour lookup tables are also provided on this hardware. This allows for a good quality image to be displayed in false colour, especially when used with display hardware such as the EGA or VGA systems.

3.5. Dumping Images to a Laser Printer

In order to display hard copies of captured and processed images, it has been necessary to write software for dumping these images to a laser printer. The laser printer was chosen as the hardcopy device as the dot matrix printer cannot provide as good a resolution. Very little is written in the printer manuals about raster plotting of images on the laser printer. Ross [ROS88] gives a detailed description of the method of controlling a laser printer, in the HP laserJet emulation mode, for image dumping.

In the HP laserJet emulation mode, a laser printer is able to print graphics or text at a resolution of up to 300 dots per inch (dpi). The graphics images are printed using raster graphics printing. Depending on the number of grey scales required, a single raster line must be generated from all the intensities of one line of the image for each row of the grey scale. The scan lines must then be sent to the printer. Thus, if 64 grey scales are chosen in an 8x8 matrix representing the dot of a pixel, eight scan lines must be generated for one row of the image.

Programs have been written allowing the use of 16 or 64 grey scales. These grey scales are generated in 5x5 and 8x8 matrices respectively.

Figures 2 and 3 show two examples of digital images captured using the Data Translation Frame Grabber and Colour CCD camera, dumped to the laser printer using the two scales of resolution.

Figure 2 shows an image dumped onto the laser printer using 16 grey scales in the 5x5 matrix. The negative of the image is shown.

Figure 2 Image displayed using 16 grey scales in a 5x5 matrix.

Figure 3 shows the same image as in Figure 2. Here the image has been printed using the 64 grey scales in the 8x8 matrix as a positive image. Evidence of the sophistication of the image displaying routines are shown by two inset images, placed with software provided by Jordaan [JOR89]. Careful inspection will also reveal that the left inset image has been flipped left to right.

Figure 3 Image displayed using 64 grey scales in an 8x8 matrix.

3.6. Reducing Storage Demands

Digital Images require large amounts of data storage. As an example, consider an image of 512x512 pixels at 8 bits per pixel, a dimension typical of most image capture devices. The total storage requirement for an image of this magnitude is 262 KBytes. Methods of reducing storage requirements should thus be investigated for a fully fledged system. Zigon [ZIG89] describes how run length encoding can be performed on large amounts of data to reduce the storage requirements. Sequences of identical values are looked for and a count of these are recorded with the actual value. Zigon [ZIG89] indicates that the worst case performance of the algorithm results in an output file of double the size. For digital images, however, he claims that this degeneracy has never occurred. As captured images usually have large areas where there are identical intensities, this method would be an ideal and quick way of reducing the demands on storage. Most researchers on the subject of image processing discuss this topic as it is important in the storage of several images [GON77] [ROS76].

Chapter Four

4. Work Performed on the Host Machine (80286) and T414 Transputer

A large amount of work has been carried out on the INTEL 80286 machine and the T414 transputer. In order to make a good comparison of the performance of the 80286 processor as opposed to that of the transputer, several areas had to be covered. The T414 transputer has been used as a direct comparison of processing power, as well as for familiarization of the occam programming language and the parallel processing environment. The T800 processor was used later in the implementation of the Fourier Transforms. With regard to the actual processing power of the two architectures studied here, programs are written on the 80286 architecture implementing some of the more common, low level, image processing techniques. These include the tasks of scaling an image by either adding, subtracting, multiplying, or dividing by a constant. Filtering, using the Laplace Filter, is used, as this is a multiplication intensive algorithm. In the absence of a maths coprocessor (the INTEL 80287), most of the work was done using integer values. There is an immediate increase in processing time when real values are used without a coprocessor, due to the use of the maths emulation libraries built into the high level languages, in both Turbo Pascal and occam2. Also, the low level tasks performed do not always need real values, whereas tasks such as the Fourier Transform do because of the resolution of the values worked with. These Fourier analysis techniques have been implemented, using reals, in both one and two dimensions. These are implemented both with and without coprocessors to get an idea of the speed up, using a coprocessor, and to enable a fair comparison with the T800 transputer which has an on board floating point processor to be made. This is shown in the relevant sections (Chapters 6 and 7).

All the programs that have been implemented on the 80286 have been implemented in Turbo Pascal version 4.0. The reason for using this language is that it is regarded as the fastest object code and has become something of an industry standard. Turbo Pascal also includes all the features that have been necessary for the tests on the 80286 during this project. Although Parker [PAR88] claims, and is correct in doing so, that high level languages do not provide the most compact machine code or the fastest execution time, the file handling features as well as the screen manipulation ability under Turbo Pascal are further features favouring the choice of a high level language as opposed to an assembler language. Software development time is also quicker when using a high level language.

4.1. Reading Images From Memory

The initial work performed on the 80286 processor involved programs for reading digital images, allowing the image processing programs access to the byte wide pixels.

An immediate disadvantage of the INTEL range of processors (up to the 80286) running under

MS-DOS is the 64 KByte page boundary. This has restricted the images to an upper limit of 64 KBytes. Bearing the 2^n image boundary conditions in mind[7], the initial image size has been kept at 256x256 pixels.

The declaration shown below indicates the memory allocation for reading an image of 256x256 pixels at 8 bits per pixel (The declarations are written in Turbo Pascal).

```
type
  Big = array [0..255] OF BYTE;

var
  Block : array [0..1] OF Big ABSOLUTE $5000:0000;
```

To overcome the 0..1 range of the array for the *Block* to contain the image, the range checking has had to be turned off using the {$R-} option under Turbo Pascal. The advantage of the declaration of the Big array is that block reads can be performed on the file containing the image, thereby allowing for quicker reading of the image.

The statement showing **ABSOLUTE $5000:0000** causes the block that is being read into memory to start at **5000:0000**. This is a dangerous practice as problems could arise should any other relevant data or program require that area of memory.

In order to overcome the 64 KByte page boundary, dynamic data structures can be implemented. This also alleviates the problem of a fixed area of memory being taken up. The only disadvantage of using dynamic data structures is that the execution time is slightly slower than the previous method because of the manipulation of the links. An extract of a program is shown below, indicating how images can be read using dynamic data structures. To avoid the waste of space, for example, in having a fixed row length of say a maximum length of 512 pixels, a number of arrays could be declared, all remaining on the 2^n boundary, each to be used by the same pointer structure.

```
const
  n = 512;  (* Largest row length *)

type
  Row  = array [0..n] OF BYTE;
  ARow = ^Row;

var
  Image = array [0..n] OF ARow;
```

In order to use these declarations, the following pseudo code would have to be implemented:

```
New (Image[i])     (* Allocate a new row i *)
for j := 0 to RowLength do
  Image[i]^[j] := read (File, Intensity); (* Read pixels *)
```

To dispose of the image, once it has been used, a simple loop through the image, using the *Dispose* routine available under Turbo Pascal, would free the memory for future use. The heap

[7] This is a restriction in the Fast Fourier Transform to enable easier coding of the algorithm.

can be set to the maximum expected size for the images to be stored or alternately, the maximum heap size can be used, with care. A low cost image processing system is described by MacManis [MAN87] implemented on the Amiga 1000 computer. This computer (68000 based) does not have the 64 KByte page boundary and it is thus easy to address large arrays on this machine. It is obvious that the 64 KByte page limit can only truly be overcome by using a different architecture.

The method described above is ideal in preventing the allocation of images to specific areas in memory, a very dangerous practice which should be avoided.

Benchmarks for reading images of 256x256 pixels are shown below. The clock frequency and type of drive from which the images were read are indicated. All the times are in seconds.

	6 MHz	8 MHz	10 MHz
Hard Drive	3.372	3.323	2.516
Virtual Drive	0.544	0.407	0.846

These results are depicted in Figure 4. From the graph it is evident that the clock speed improves performance, although there is still a dependance on the type of hardware that should be used.

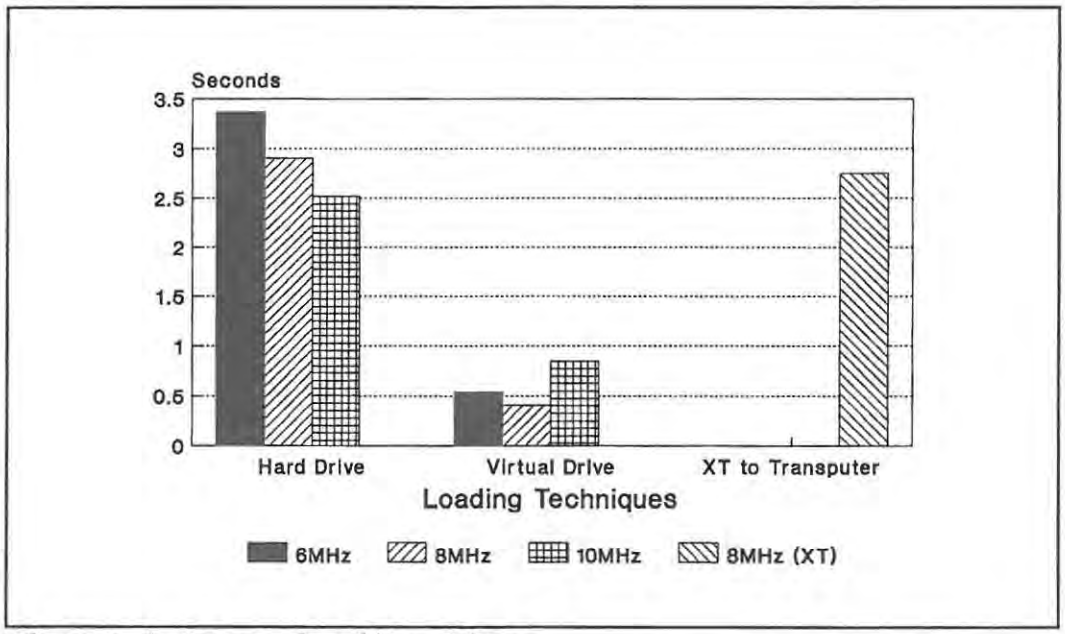

Figure 4 Image loading times.

The 10 MHz test was performed on a machine which had a hard drive access time of 35 msec compared to that of 70 msec for those tests at 6 and 8 MHz. Also, for the 10 MHz test on the virtual drive, extended memory was used. This was not the case for the 6 and 8 MHz tests on virtual drives. This explains why the 10 MHz virtual drive test is slower than the others.

It is important to know the time taken for reading an image into memory. Although the times are specific to the hardware components that the machine comprises, the figures shown above give a good indication of the times one can expect from the hardware.

It can be seen from the above times that the speeds of the various hardware components have some effect on the time taken to read the images. The various manufacturers do not all maintain a set standard for the speeds of the hardware components. The speed of the hard drive, the access time of the RAM and the number of wait states are all important aspects to be considered in finding the optimum host machine to be used in conjunction with the transputer.

The time taken to load an image of 256x256 pixels, totalling 64 KBytes of data, from the memory of the XT (8086), running at 8 MHz, is also shown in Figure 4. This is important when deciding whether an operation should be performed on the transputer or on the host machine. By looking at say the Filtering times, shown below, for the AT and T414, it is obvious that only a small gain in time is achieved by using the transputer. Operations which display smaller differences in time for the two architectures should thus be run on the host machine.

4.2. Low Level Image Processing Tests

Some simple image processing operations were performed on the AT and on a single T414 transputer. These included the intensity scaling, by addition of a constant and multiplication of a constant, and the implementation of a filter. The tests on the AT were run for the three clock speeds as shown earlier.

The benchmark results shown below are for the simple intensity scaling. All times shown are in seconds.

	AT (80286)			T414
	6 MHz	8 MHz	10 MHz	20 MHz
Adding an Integer constant	2.420	1.760	1.430	0.378
Multiplying by a constant				
Real constant	20.810	13.950	11.480	2.974
Integer	5.540	4.150	3.300	0.434

A simple low level image filtering technique, employing a Laplacian filter [GON77] [ROS76], was

used. The reason for choosing a technique of this nature is that it involves a large amount of computation, especially multiplications. As an example, consider a 256x256 pixel image which has 9 multiplications per pixel. A total of 589 thousand multiplications are performed, excluding the additions and special cases at the edges that must be considered. Integer values were used for the computation of the image with the filter mask. A 3x3 element Laplacian filter mask is shown below. Although it can be of any size, the central value must equal the sum of the central edge values in accordance with the Laplacian operation.

0	-1	0
-1	4	-1
0	-1	0

A 3x3 pixel mask was chosen for this test. The program was written so that the user could enter the desired mask. Special cases have to be included for the pixels at the edge of the image, as no wrap around multiplication is performed there. The values outside the edge of the image are set at zero. The benchmark figures for this operation are shown below, using an image size of 256x256 pixels. All times shown are in seconds.

AT (80286)			**T414**
6 MHz	**8 MHz**	**10MHz**	**20 MHz**
9.433	6.980	5.490	2.744

From the above figures, it can be seen that the AT (80286) is fairly impressive in terms of integer multiplication when compared with the T414, which operates at 20 MHz, thereby giving the T414 immediate advantage. The T414 is approximately 10 times faster than the AT for real number multiplication. This is a good performance considering that software libraries have to be used. The T800, having an onboard floating point processor, will show great improvements over the T414 and 80286. This is shown in the tests performed on the one and two-dimensional Fast Fourier Transforms in Chapters 6 and 7.

The results for the low level image processing operations are shown in Figure 5. It is quite evident how much faster the T414 is than the 80286 based AT.

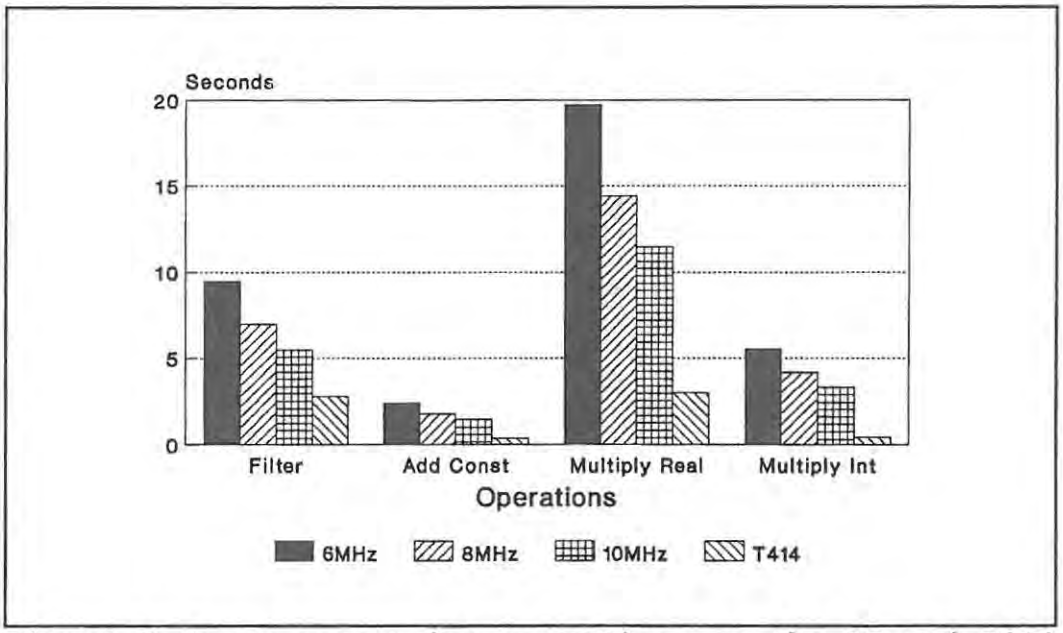

Figure 5 Image processing operations on the AT and T414

When using the transputer, the introduction of parallelism into the operations for a single processor should be avoided. Care must be taken not to introduce a level of parallelism, onto a processor, which is too fine. All the occam processes implemented above were run as sequential processes using the SEQ construct [BUR88]. A typical operation follows:

```
SEQ i = 0 FOR number.of.rows
  SEQ j = 0 FOR number.of.columns
    process.matrix
```

The newcomer to parallel processing is often tempted to use the PAR construct here, that is:

```
PAR i = 0 FOR number.of.rows
  PAR j = 0 FOR number.of.columns
    process.matrix
```

The time taken to perform a context switch, on the transputer, is approximately 0.1 microseconds [TRA88]. The PAR construct, as shown above, simply introduces an overhead as a result of context switching for all the processes enstantiated. Introducing parallelism into the operation would be better performed on a network of processors, again being aware of the granularity in the parallelism. Morrow and Perrot [MOR88a] introduced this in their work on *image* and *task* parallelism in the implementation of low level image processing, using similar tasks to those shown here.

4.3. Investigating the Performance of the AT

The work performed in section 4.2. on the 80286 AT shows relatively good performance for small scale computation. Thus, the question regarding the choice of architecture to be used as the host in the context of this project, can be addressed. Lua, [LUA88] investigated the relative performance measurements of the INTEL 8086, 80286 and 80386 architectures, drawing some

interesting conclusions. These are summarized as follows:

1. Although the processor speeds are a linear function of the clock frequency, each of the INTEL CPU's has an upper bound as to the clock frequency at which it will operate correctly.

2. At the same clock frequency, the 80386 machine is almost 3 times faster than the 8086/8. However, there is hardly any improvement of the 80386 compared to the 80286 when both are running under the REAL 8086 mode.

3. The wait state on the 80286 architecture effects it by 25%. A 1 wait state 12 MHz 80286 is slightly slower than a 0 wait state 10 MHz 80286 machine indicating a 25% slow down.

4. In terms of the average clock speed, a 0 wait state 80286 machine is 3.36 times faster than a 8086. The 0 wait state 80386 machine is only 2.83 times faster than this 8086 machine. Lua's reasoning for this claim is that the 80386 memory design is not as optimized as that of the 80286 machine.

Lua [LUA88] summarizes the MIPS ratings as follows:

10 MHz	8088 XT	0.16 MIPS
10 MHz	80286 (0)	1.21 MIPS
16 MHz	80386	1.63 MIPS

(0) Indicates zero wait states.

Conclusions for this project concerning the speeds of the various components are drawn from the summary of Lua's findings and from the tests performed on the 80286 machine.

Although the 80386 is the fastest processor, and appears to be the ideal choice for a project involving intensive computations (as it can accommodate the highest clock speed), the 80286 proves to have a better cost performance ratio. Also to be considered is the fact that this machine would act as a host to the transputer network most of the time. It would thus be pointless, and somewhat of an overkill, to use an 80386 based host machine. Thus the ideal Intel-based machine would be one comprising an 80286, the fastest hard drive, fastest memory, and highest clock speed.

Chapter Five

5. Transputer Network Topologies

As this project deals primarily with a general purpose transputer network for image processing, it has been necessary to perform a large amount of research in the domain of transputer networks. It is important that a sound topology be chosen for a multiprocessor environment. This is especially so, and somewhat more difficult, when a multiprocessor system is sought which must offer generality in terms of the spectrum of operations that can be performed, and which show a good efficiency in performance. The work described in this section offers justification for the choice of topology. Investigations into the choice of topology are based on the work of several authors in the field of image processing and transputers. Where relevant, details of the methods of testing this network are given, as well as methods of improving the performance.

Image processing requires a large amount of storage for the images to be processed. These images are generally stored in a matrix format, with array processors being best suited to this application. For operations performed on the images to occur, matrix transposition is often required. In order to provide for fast access to the image as well as for fast matrix manipulation, a suitable method of access to the required area of the matrix, as well as close proximity to the matrix, is necessary. As each transputer, in a network, can operate *individually*, a network comprising of a matrix of transputers would be ideal for image processing applications. The image could be evenly distributed amongst the processor nodes in the matrix. A method proposed by Cok [COK88] employs a toroidal configuration of transputers and a single controlling node. This method was chosen by Cok [COK88] because of its simplicity, expandability, and compatibility with image processing. The diagram in Figure 6 shows this transputer network.

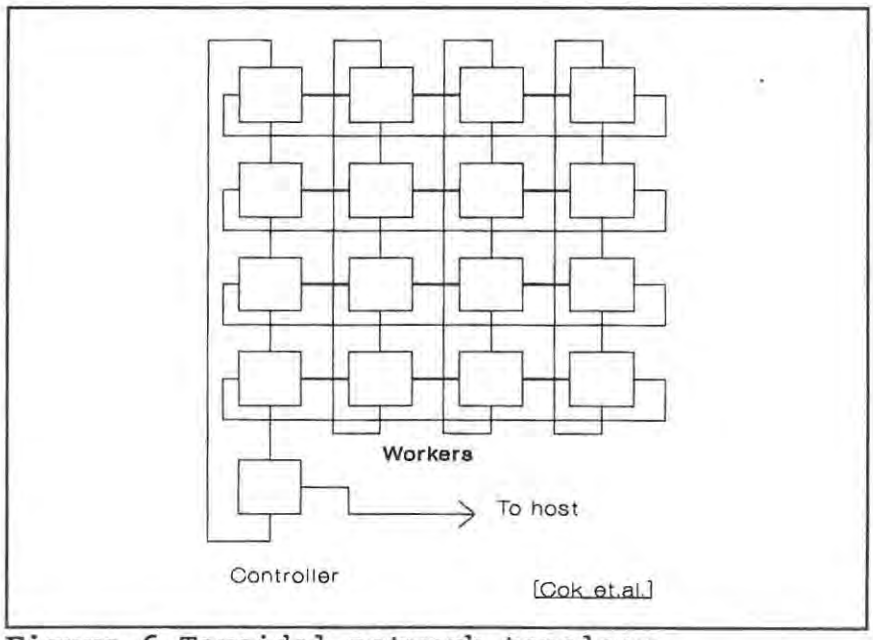

Figure 6 Toroidal network topology.

From Figure 6, it is evident that this network can be expanded infinitely, as the number of interconnections at each node does not change as the toroid is expanded upon. Although this network has a reduced connectivity, this is disadvantageous when communications are required over a larger area. The toroid does not necessarily have to be NxN in dimension but can be any MxN, where M <> N. The toroidal configuration has the links on the edges of the matrix joining the opposite edges of the matrix so it is not a simple mesh configuration. This has several advantages specific to image processing applications. One of these being that, in performing functions such as convolutions or filtering (requiring a mask to be run across the image), there is no need for special case programming at the edges of the image as the mask can wrap around. Another advantage is that this configuration can also be converted to a simple mesh network by simply disconnecting the links at the edges. These spare links can be used for I/O if necessary, or can be configured to act as a pipelined network, allowing for I/O at any of the processor nodes.

Cok [COK88] makes use of a toroidal array architecture with bussed I/O (VME) and dual ported memory. The dual ported memory allows the transputer to process one set of data while another is being loaded. The transputer network connects to a host computer, thereby providing a workstation. This configuration forms a distributed memory system which provides for maximum flexibility. Cok [COK88] explains that if the data is loaded and unloaded from the toroidal array via the bus, speed of transfer becomes a problem. Although there is no evidence of this speed, this problem in the speed of transfer must be due to an attempt to remain within an interactive real-time domain. In the research for this thesis, it was found that with the B004[8] board the speed of transfer of an image is relatively slow when using the standard link to the transputer. To transfer a 64 KByte block of data representing an image takes about 2.75 seconds.

A second network topology for image processing is described by Morrow et.al. [MOR88]. Here the network used is slightly different from the toroidal network in that it does not wrap around. Rather, links on the edge transputers go to the neighbouring edge transputers. Morrow et.al. [MOR88] describes this as *sealing the array*. This is shown in Figure 7.

[8] This was the original board used in the project. It has a single T414 transputer and 2 MBytes of RAM.

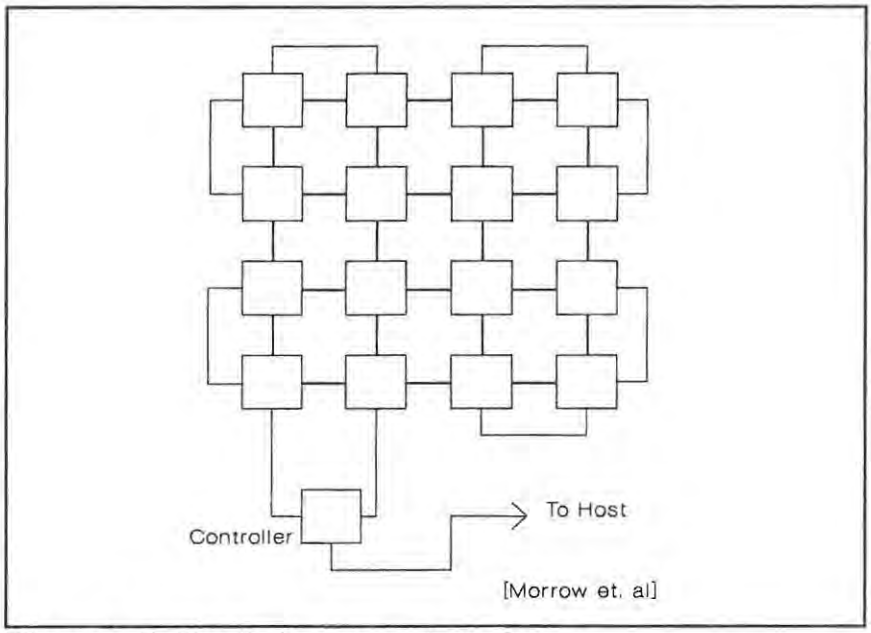

Figure 7 Sealed array topology.

A *controller* node is used in this configuration, acting as the link between the network and the host. It performs the transfer of the data to and from the network. As described earlier, certain image processing algorithms require special case treatment at the edge of the image. Here, the special case treatments would require that the program, such as a convolution, be different on the edge transputers from the inner transputers. This could be done by implementing the *bordering* technique [MOR88]. In other words, the image, when divided amongst the transputers, could automatically be given the edge of the neighbouring sub-image, so as to eliminate the need to transfer the image across when processing of this local image.

The *controller* node, responsible for the communication between the host and the network, can be programmed in such a way that the image is divided evenly amongst the transputers in the network. For example an image of 1024x1024 pixels in a network of 16 transputers in either the toroidal or sealed array topology, could be distributed to the memory of each transputer where each would hold 64x64 pixels totalling 4096 bytes per processor as opposed to approximately 1 Megabyte on a single processor.

5.1. Creating a General Purpose Network

Several factors need to be taken into account when deciding upon the topology of the network to be used. The most important factor is that of the type of operation to be executed on the network. Image processing lends itself to a *mesh* type network configuration. This is due to the matrix structure of an image (usually MxN pixels), thereby allowing ease of distribution. Another factor is that of the reliance of certain image processing operations on neighbouring image data. As shown above, both Morrow et.al. [MOR88] and Cok [COK88] have implemented *mesh*-like topologies, but each with subtle differences, depending on the relevant author's application to be

performed.

It is decided that the *mesh*-like topology be used for this project, with changes. The major difference between this topology and those previously discussed, is the fact that this topology incorporates a pipeline in the network as well. The primary reason for the choice of the pipeline is that data distribution along it is easy. Tests described later in the thesis will show this. Several other factors also influence this choice of topology. In a private communication with Cok [COK88], it was recommended that a pipeline of processors is a good choice when only a few processors are available for use in the network. For ease of programming, the pipeline also seemed an obvious advantage. In choosing the pipeline as the main route for data distribution, this has not divorced the network from the possibility of using the *wrap* around and *bordering* techniques of Morrow et.al. [MOR88] and Cok [COK88]. These edge links are used, thereby keeping the mesh configuration. Further, the *controller* (or *root*) processor is also used, allowing the data to be transferred from the host to the *worker* processors.

The network used in this project is shown in Figure 8.

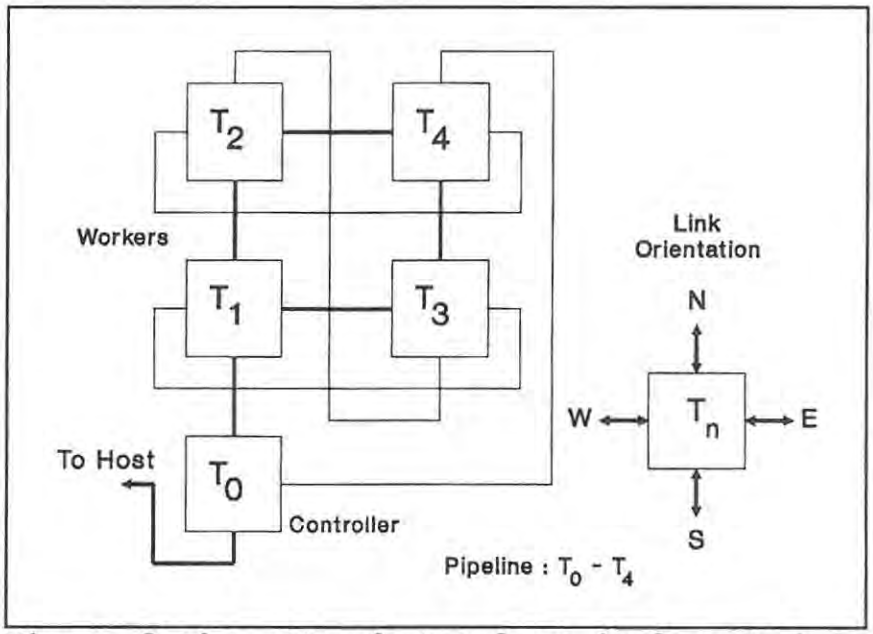

Figure 8 The network topology implemented in this project.

This network is easily expanded, by increasing the length of the pipeline. The edge connections are still maintained in the expansion. The effective *mesh* topology can be expanded in either the number of rows, or columns, or both, thereby maintaining symmetry. The operations in the *controller* process and the *worker* process are kept separate. The *controller* is merely used for data transfer and storage. In cases where there is no advantage of placing an operation on the *workers*, it is placed on the *controller* process. This will be discussed in detail in the section on the Fast Fourier Transform.

This network, discussed above, can be looked at in many states of undress. By simply untwisting the links into a straight pipeline, with the possibility of visibly confusing the links, but maintaining the edge links, the network shown in Figure 9 can be observed.

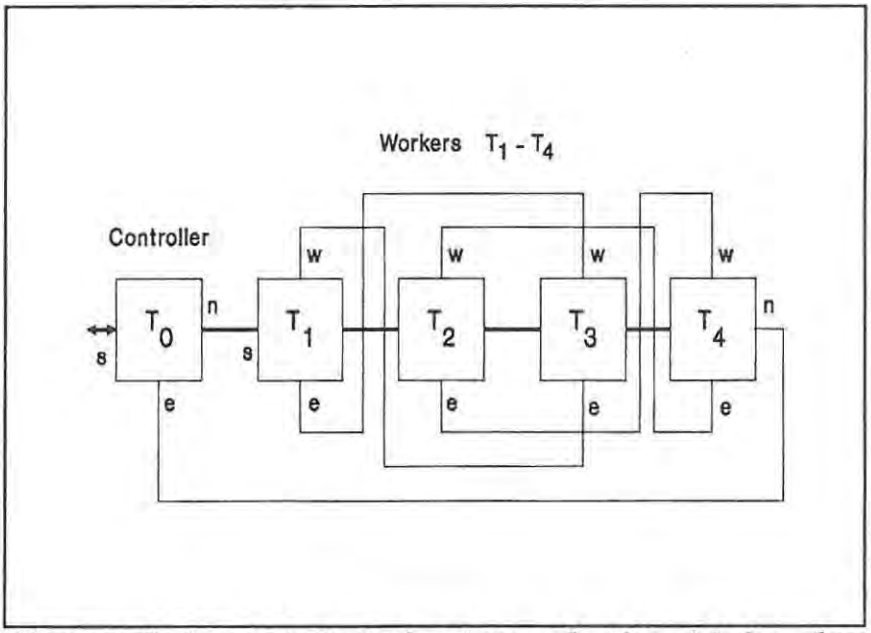

Figure 9 An untwisted view of the links for the network.

The topology shown in Figure 9 closely resembles that of the Perfect Shuffle Machine (PSM) topology as discussed by Schomberg [SCH89]. Sandler and Eghtesadi [SAN88] have implemented a Hough Transform[9] on a similar network, the OSMMA sub-system, achieving good results. A diagram of the Perfect Shuffle Machine topology is shown in Figure 10. This allows for a minimum link usage between processors, which would otherwise be place far apart in a pipeline. Schomberg [SCH89] has implemented this topology for image processing. He indicates the nearest neighbour connections and the additional Perfect Shuffle network. The network used for this project (in complete form) is used as an example. In order to reach $processor_4$ from $processor_1$, along the pipeline, would require communication along three intermediate processor links. Following the linking of the network of this thesis in the untwisted form (Figure 9), $processor_4$ can be reached along two processor links, i.e. from $processor_1$ to $processor_3$ to $processor_4$. The number of links to cross is actually $\log_2 N$, where N represents the number of processors. Although the Perfect Shuffle Machine allows $processor_4$ to be reached directly from $processor_1$, there is still a close resemblance between the two topologies. This saving in the number of links to be crossed may become significant when large amounts of data need to be shuffled.

[9] The theory of the Hough Transform, used for shape analysis by extracting global features, is described by Ben-Tzvi and Sandler [BEN89]. This transform is heavily dependant on Sine and Cosine computations, and is thus computationally intensive.

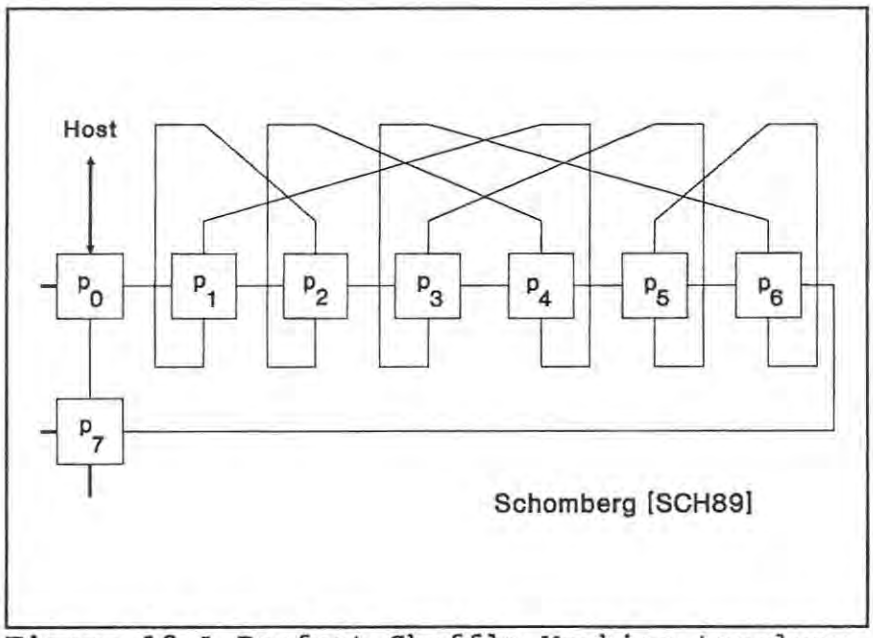

Figure 10 A Perfect Shuffle Machine topology.

5.2. Connecting the Network

Network configurations are often required which are not only general purpose in terms of their applicability, but have general purpose link allocation algorithms, allowing easy expansion of the size of the network. The method of finding a general purpose algorithm for allocating the links when configuring the network is dealt with in this section. The algorithm is general purpose in terms of the size of the network when allocating the links but is strictly specific in terms of the topology considered.

When looking at the links on each transputer, it is easiest to consider the four links as inputs. Here, the input of the transputer neighbouring the one being looked at provides the output for that transputer. The total number of channels needed is four times the number of transputers used, plus the number which have both inputs and outputs. This particularly concerns the link to the host, which must be bidirectional. In a five transputer network as shown in Figure 8, 21 channels are required. The 21st channel acts as the output to the host. This method of allocating links is easier than if 2 channels are considered for each of the four links of a transputer.

Figure 11 shows how all four links on each transputer are considered as inputs.

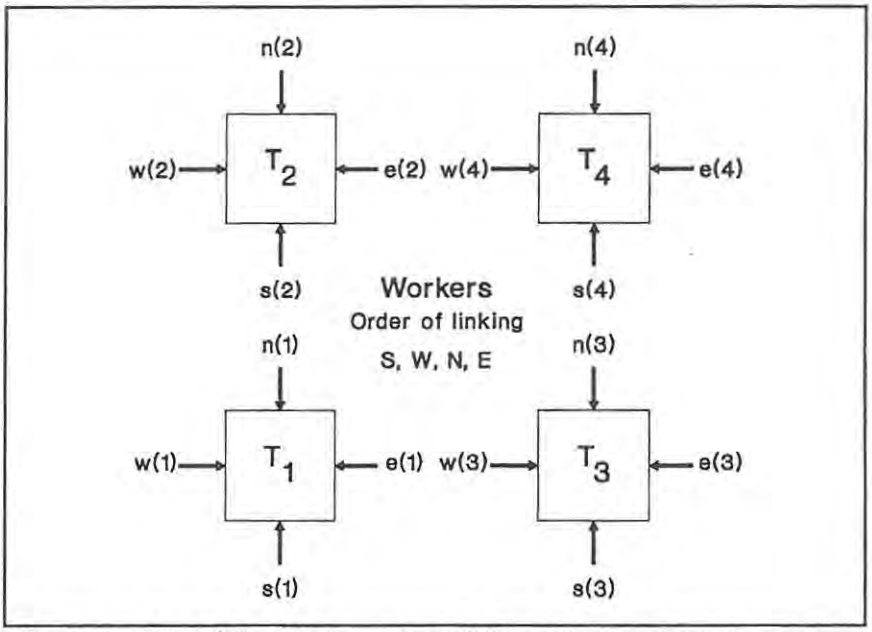

Figure 11 All links considered as inputs.

Considering the physical input and output channels appearing on the transputer as shown in Figure 12,

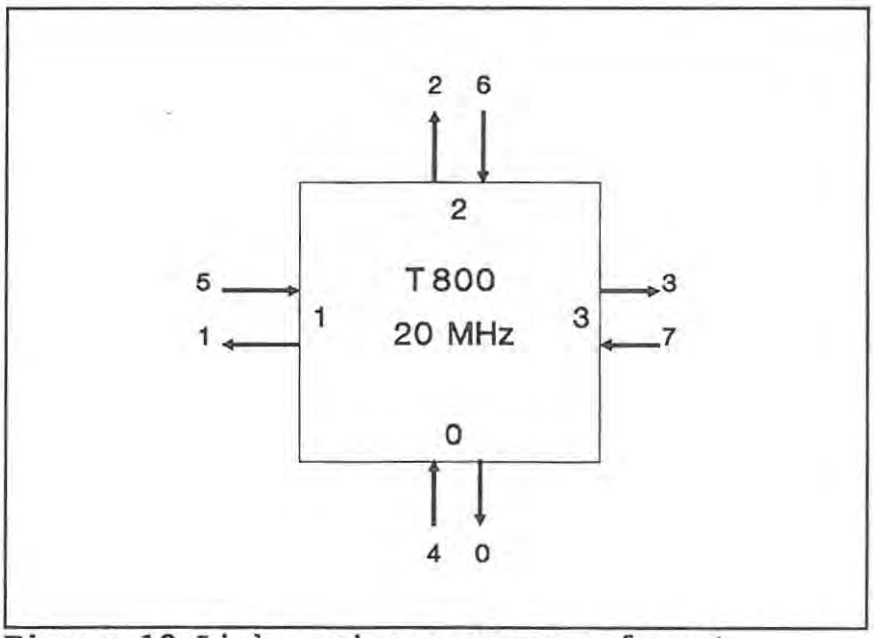

Figure 12 Links using a compass format.

and following a *compass* like association of north, east, south and west for each of the 4 links (0 to 3), an array can be constructed which has the channel assignments. This array allows for easy development of the formula. Automatic assignment of the channels according to the number of rows and columns in the network can be derived from this formula.

The array for allocating the channels can be constructed as shown below. The individual transputers are shown below their respective s, w, n and e channels, which have the array subscripts assigned to them.

S	W	N	E	S	W	N	E	S	W	N	E .....
0	1	2	3	4	5	6	7	8	9	10	11 .....

Root Controller T_1 T_2 Workers T_3 etc.

The following formulae are established for the worker processes by making use of the above array in the following manner. The input links of each processor in the pipeline simply follow in a clockwise manner starting at the southerly link. These are numbered by adding 1 as the links are traversed. These formulae are thus

south : i*4
west : (i*4)+1
north : (i*4)+2
east : (i*4)+3

where

i = the processor number.

These input links are denoted: *s.in, w.in, n.in, e.in* in the occam configuration code.

The output for the north and south links is found by considering the pipeline of links. The north of processor$_i$ connects to the south link of processor$_{(i+1)}$, the next processor in the pipeline. The *s.out* link of processor$_i$ is connected to the *n.in* of the preceding processor, i.e. processor$_{(i-1)}$.

The generalized formula for the *n.out* and *s.out* links are shown below:

s.out = (i*4)-2
n.out = (i+1)*4

where

i again represents the processor.

In considering the *w.out* and *e.out* links, the **MOD** function is required. This is because of the *wrap-around* function created by a *w.out* link of one processor linking to the *e.in* of a processor to its right. The same applies to the *e.out* link. The processor to which one of the east or west links connects is dependant on the number of rows of processors considered for the network. As an example, consider a network with 2 rows and 2 columns, making a total of four *worker* processors. The west link of *worker*$_1$ connects to the east link of *worker*$_3$. The east link of *worker*$_1$

connects to the west link of *worker*$_3$. The **MOD** function with the relevant offset value takes care of this.

The following formula are for the *e.out* and *w.out* links:

$$w.out = \{4*[((i-1)+rows) \text{ MOD } num.procs]+1\}+3$$

$$e.out = \{4*[(num.procs+((i-1)-rows)) \text{ MOD } num.procs]+1\}+1$$

where

i represents the processor identity number.
num.procs represents the number of processors in the network, including the *controller* processor.

The formulae, shown above, apply to the *worker* processors from the second to the second last processor in the network. The *controller*, first and last *worker* have slightly different formula to that shown above. This is to accommodate the odd linking introduced by the *controller* being connected to the host and the fact that the last *worker* connects to the east link of the *controller*. These connections are non standard within the east-west and north-south linking concept of the matrix of *workers*. These formulae are, however, not drastically different.

The following extract of code shows the configuration code for placing the links. The actual link numbers are not shown. The full code for this appears in Appendix B. At compile time the values for the links are calculated and the allocation is performed.

```
PLACED PAR
  PROCESSOR 0 T4  -- Root (controller) transputer is a T414

    VAL link numbers :
    PLACE links :

    controller.process (links for process)

  PROCESSOR 1 T8

    VAL link numbers :
    PLACE links :

    worker.process (links for first worker)

PLACED PAR i = 2 FOR num.procs-2
  PROCESSOR i T8

    VAL link numbers :
    PLACE links :

    worker.process (links for all workers)

  PROCESSOR num.procs T8

    VAL link numbers :
    PLACE links :

    worker.process (links for last worker)
```

A program, written in Turbo Pascal (v4.0) is used to confirm these formula for networks of this topology with up to 1000 links. This allows a network of over 200 *worker* processors to be connected. This number of processors is obviously way beyond the scope of this project in terms of cost and manageability. This is, nevertheless, proof of the correctness of the general linking algorithm established here.

5.3. Data Distribution - Methods and Enhancements

5.3.1. Synchronization Overheads

All communications on channels on the transputer, whether on internal occam channels or across links between neighbouring transputers occurs as point-to-point synchronous communication [TRA88]. Although the T800 displays the enhanced feature of having *overlapped acknowledges*, this format of communication is still relevant. For communication to take place, the process which first becomes ready must wait until the process with which it must rendezvous is also ready [JON87]. This point-to-point communication takes place one byte at a time. On the links, each time a byte is transmitted synchronization must take place [DIN89].

The immediate method of data communication that comes to mind is the transfer of a single sequential data stream, sent element by element, where each element has its relevant number of bytes transmitted. This was a major criticism of earlier versions of occam [BUR88]. The sending of a single element at a time requires a rendezvous for each element sent. Sending a stream of data elements can typically be coded as follows:

```
SEQ i = 0 FOR number.of.elements
  out ! element[i]
```

The vector (array of data) used here is *element* and *number.of.elements* from this vector are sent along channel *out*.

The current version of occam (occam2) allows groups of objects to be transferred during a single rendezvous [BUR88]. Since the data types may vary in length, it is necessary to check the size of the single objects during communication. This type checking requires that a channel be *aware* of the type of input expression and output variable. This is made possible with the introduction of protocols into occam2. The protocols are added to the definition of channels, thereby allowing the channel to know what data is being communicated. Protocols don't make use of a **SEQ** statement in controlling the transfer of data, where more than one data element exists, such as for arrays of data. Comparing this method of data transfer to that where the **SEQ** statement is used, it is evident that synchronization overheads will occur when each individual element is sent when controlled by the **SEQ** statement.

Following the syntax of the occam2 language, occam's channel protocols can take on the three forms shown below [BUR88].

The following results were recorded for these two tests:

SEQUENTIAL	:	T = 40.5132
PROTOCOL	:	T = 22.7678 seconds

It is obvious from these results that the method using channel protocols for data transfer proves to be much faster. This would, however, not be the case when single data elements, instead of vectors are sent. Here it would be pointless as the amount data on the links would be increased because of the header data used in indicating the protocol. If the data being sent is in BYTES, then sending a protocol header (also a BYTE) and the byte of data would simply double the amount of data on the channel.

These protocols do, however, prove to be useful when several types of data must be sent along the same channel. This is the case for most of the image processing and data routing applications implemented in this project. For these, several types of data format need to be sent across the network. This can be seen in the section covering the Fast Fourier Transform and the sections below.

5.3.2. Two Methods of Distributing the Data

Having established that protocols prove to be the faster method of data transfer, it has been necessary to establish which method of distributing data across a network is fastest. Each processor must have some data of the image in order to process it. This is in keeping with the *image parallelism* paradigm discussed by Morrow and Perrot [MOR88a] where an image is partitioned amongst processors and each processor executes the whole algorithm. In testing the distribution of the data, protocols are not used. This does not matter since both tests employ the same data format and data is sent sequentially.

The topology of the network will determine the method by which data can be distributed efficiently and effectively. Looking at the topology of the network used in this project, the immediate path of distribution that comes to mind is that along the pipeline formed by the north-south links joining the processors. Noting, however, the presence of the edge links, a second method of data distribution can be investigated. These two methods are described and evaluated below:

5.3.2.1. Data Distribution along the Pipeline

The data is sent along the pipeline in a simple sequential manner. Each *worker* process has the same code and resides on separate processors. The data consists of a two-dimensional buffer of values representing an *image*. This is so that the distribution process can divide the *image* evenly amongst each of the *worker* processors, in keeping with *image parallelism* [MOR88a]. Each processor can thus execute an image processing algorithm on the data immediately resident. An indication of what sort of times can be expected in distributing a certain size image among a set

number of processors can be gained from this. One can establish whether it is better to perform the task on the host processor (in this case an 80286) or on the network when the task execution time is a known fraction of the total time taken. In cases where the task is trivial and the time taken to get the data onto the network from the host, distribute and process it, takes longer than for the same task to be performed on the host, or alternately on a single T800 processor, the task should be performed on this single processor or the 80286. The purposes of the test are not to establish whether it is better to execute the algorithm on the host or transputer network, but rather to compare the two methods of data distribution for the network.

The extract of code for the *worker* processors, which follows, shows the method of distributing the data along the pipeline:

```
PROC workers (CHAN OF ANY s.out, w.out, n.out, e.out,
                          s.in, w.in, n.in, e.in, VAL INT id.num)

  VAL im.rows IS 64 :  -- Size of image rows and columns
  VAL im.cols IS 64 :
  VAL rows IS 2 :      -- Size of network matrix
  VAL cols IS 2 :
  VAL im.size IS im.rows*im.cols :
  VAL num.procs IS rows*cols :
  VAL rows.per.proc IS im.rows/num.procs : -- Each proc has this many rows
  INT i, j, neighbours.value, number, local.id :
  [im.rows][im.cols] INT buffer : -- Buffer to store values for passing on
  [im.rows/num.procs][im.cols] INT local.buffer : -- Local data buffer

  -- Because the data distribution follows a pipeline the sending
  -- and scaling up of the data and subsequent regathering will
  -- follow a pipeline format - local storage area is used so as
  -- to follow the same format as the other data distribution
  -- method.

  SEQ cnt = 0 FOR 100
    SEQ
      SEQ i = 0 FOR (num.procs - id.num)*rows.per.proc  -- Pass rest on
        SEQ j = 0 FOR im.cols
          SEQ
            s.in ? neighbours.value
            n.out ! neighbours.value
      SEQ i = 0 FOR rows.per.proc     -- Get data for local processor
        SEQ j = 0 FOR im.cols
          s.in ? local.buffer[i][j]
      SEQ i = 0 FOR rows.per.proc     -- Send data from local processor
        SEQ j = 0 FOR im.cols
          n.out ! local.buffer[i][j]
      SEQ i = 0 FOR (id.num-1)*rows.per.proc  -- Pass previous neighbours on
        SEQ j = 0 FOR im.cols
          SEQ
            s.in ? neighbours.value
            n.out ! neighbours.value
:

PROC controller (CHAN OF ANY s.out, w.out, n.out, e.out,
                             s.in, w.in, n.in, e.in, VAL INT id.num)

  VAL im.rows IS 64 :  -- Size of rows and columns for image
  VAL im.cols IS 64 :
  VAL rows IS 2 :      -- Size of network matrix
  VAL cols IS 2 :
  VAL im.size IS im.rows*im.cols :
  VAL num.procs IS rows*cols :
  [im.size] INT local.store :
  INT start.time, stop.time :
  TIMER clock :
```

```
  SEQ
    SEQ i = 0 FOR im.size     -- Must send value to each worker
      s.in ? local.store[i]  -- Get value from PC - put in local storage
    -- Values now obtained so can start timing
    clock ? start.time
    SEQ cnt = 0 FOR 100
      SEQ
        SEQ i = 0 FOR im.size
          n.out ! local.store[i] -- Pass on to workers
        SEQ i = 0 FOR im.size     -- Get value from each worker
          e.in ? local.store[i]  -- Get result from workers
    -- Results now back so can stop timing
    clock ? stop.time
    SEQ i = 0 FOR im.size
      s.out ! local.store[i]   -- Send results back to PC
    s.out ! start.time
    s.out ! stop.time
:
```

The time for sending an *image* of 64 by 64 pixels, each 32 bits in size to four *worker* processors 100 times is

$$T = 9.9868 \text{ seconds.}$$

5.3.2.2. Using the Edge Links

The second method of testing the data distribution involves using the edge links as well as part of the pipeline. Again the data must be evenly distributed amongst the processors. Considering the network used in the project and a two-dimensional *image*, as before, the data can be distributed as follows: The *image* can be equally divided amongst the rows into the number of rows of processors there are in the network. These groups of rows can be sent along the pipeline to temporarily reside on the first column of processors in the pipeline. As soon as each processor has passed the data on that is required by the processor ahead of it, it can send data along to the neighbouring processors (easterly). Here the columns of the image are split into the number of columns of processors in the pipeline. Regathering the data employs the same technique, the only difference being that the data continues to flow in the direction in which it entered the network. That is, it continues along to the right hand side of the network and then gathers at the processor that would be the last in the pipeline of processors and finally back to the controller processor.

An extract of the code for this method of data distribution is shown below:

```
PROC workers (CHAN OF ANY s.out, w.out, n.out, e.out,
                          s.in, w.in, n.in, e.in, VAL INT id.num)

  VAL im.rows IS 64 :  -- Size of image rows and columns
  VAL im.cols IS 64 :
  VAL rows IS 2 :       -- Size of network matrix
  VAL cols IS 2 :
  VAL im.size IS im.rows*im.cols :
  VAL num.procs IS rows*cols :
  INT16 i, j, number, local.id :
  [im.rows][im.cols] INT buffer : -- Buffer to store values for passing on
  [im.rows/rows][im.cols] INT local.buffer : -- Local data buffer
```

```
SEQ
  IF
    id.num = 1
      SEQ cnt = 0 FOR 100
        SEQ
          SEQ i = 0 FOR im.rows
            SEQ j = 0 FOR im.cols    -- So can test how long it takes
                s.in ? buffer[i][j]    -- to send the data to one processor
          SEQ i = 0 FOR im.rows
            SEQ j = 0 FOR im.cols
              s.out ! buffer[i][j]
    TRUE
      SKIP
  SEQ cnt = 0 FOR 100
    SEQ
      IF
        -- Although the data will be jumbled it does not matter
        -- for the purposes of the test
        id.num = 1
          SEQ
            SEQ i = 0 FOR im.rows/rows    -- Get 1/2 values from root
              SEQ j = 0 FOR im.cols       -- and send them straight through
                SEQ
                  s.in ? buffer[i][j]
                  n.out ! buffer[i][j]
            SEQ i = im.rows/rows FOR im.rows/rows
              SEQ j = im.cols/cols FOR im.cols/cols
                SEQ
                  s.in ? buffer[i][j]   -- Get bottom right corner in
                  e.out ! buffer[i][j]  -- Send to right neighbour
            -- Now send your data so east neighbour can collect
            SEQ i = im.rows/rows FOR im.rows/rows
              SEQ j = 0 FOR im.cols/cols
                SEQ
                  s.in ? buffer[i][j]   -- Get your data in and send to neighbour
                  e.out ! buffer[i][j]
        id.num = 2
          SEQ
            SEQ i = 0 FOR im.rows/rows    -- Get in exactly half the image
              SEQ j = 0 FOR im.cols
                s.in ? local.buffer[i][j]
            SEQ i = 0 FOR im.rows/rows
              SEQ j = im.cols/cols FOR im.cols/cols
                e.out ! local.buffer[i][j]  -- Send to right neighbour
            -- Send to easterly neighbour
            SEQ i = 0 FOR im.rows/rows
              SEQ j = 0 FOR im.cols/cols
                e.out ! local.buffer[i][j]
        id.num = 3
          SEQ
            -- Get the amount supposed to have
            SEQ i = 0 FOR im.rows/rows
              SEQ j = im.cols/cols FOR im.cols/cols
                w.in ? local.buffer[i][j]
            -- Collect westerly neighbours
            SEQ i = 0 FOR im.rows/rows
              SEQ j = 0 FOR im.cols/cols
                w.in ? local.buffer[i][j]
            -- Send the half that accumulated
            SEQ i = 0 FOR im.rows/rows
              SEQ j = 0 FOR im.cols
                n.out ! local.buffer[i][j]
        id.num = 4
          SEQ
            -- Put top half in and send it straight out
            SEQ i = 0 FOR im.rows/rows
              SEQ j = 0 FOR im.cols
                SEQ
                  w.in ? buffer[i][j]
                  n.out ! buffer[i][j]
            -- Get lower half of data
            SEQ i = im.rows/rows FOR im.rows/rows
```

```
              SEQ j = 0 FOR im.cols
                SEQ
                  s.in ? buffer[i][j]
                  n.out ! buffer[i][j]
:

PROC controller (CHAN OF ANY s.out, w.out, n.out, e.out,
                            s.in, w.in, n.in, e.in, VAL INT id.num)

  VAL im.rows IS 64 :  -- Size of rows and columns for image
  VAL im.cols IS 64 :
  VAL rows IS 2 :      -- Size of network matrix
  VAL cols IS 2 :
  VAL im.size IS im.rows*im.cols :
  VAL num.procs IS rows*cols :
  [im.size] INT local.store :
  INT start.time, stop.time, hun1.time :
  TIMER clock :

  SEQ
    SEQ i = 0 FOR im.size     -- Must send value to each worker
      s.in ? local.store[i]  -- get value from PC - put in local storage
    -- Values now obtained so can start timing
    clock ? hun1.time
    SEQ cnt = 0 FOR 100
      SEQ
        SEQ i = 0 FOR im.size
          n.out ! local.store[i]
        SEQ i = 0 FOR im.size
          n.in ? local.store[i]
    clock ? start.time
    SEQ cnt = 0 FOR 100
      SEQ
        SEQ i = 0 FOR im.size
          n.out ! local.store[i] -- pass on to workers
        SEQ i = 0 FOR im.size    -- Get value from each worker
          e.in ? local.store[i]  -- get result from workers
    -- Results now back so can stop timing
    clock ? stop.time
    SEQ i = 0 FOR im.size
      s.out ! local.store[i]
    s.out ! hun1.time
    s.out ! start.time
    s.out ! stop.time
:
```

For the same data as the previous test and the same size network, this test provides the following result:

T = 12.4475 seconds

It is evident that for the size network used, the pipeline method of data distribution proves better than the method using the neighbouring links. It should also be noted that the pipeline method of distribution is simpler and amounts to less code than the second method. The second method proves inefficient in terms of the code because it relies on the processor's position in the network to be known.

5.3.3. Fast Data Distribution

The INMOS T800 transputer exhibits certain features making it a superior processor to the T414 transputer. Specific to the subject of data distribution and transfer being discussed, the T800 shows an extreme performance improvement in the method of communication on the links. These enhanced features can be summarized as follows [TRA88]:

1. *Overlapped Acknowledges* - Here the acknowledge packets are sent before the data packet has been fully received. The presence of the overlapped acknowledges, does not affect compatibility with other INMOS products.

2. The T800 uses a DMA (direct memory access) block transfer mechanism to transfer messages between memory and other transputer products via the links.

3. The link interfaces and the processor all operate concurrently. This allows processing to continue while data is being transferred on all the links. This feature is used in the network to test the data distribution.

Although the network used in this project has links between the east and the west neighbours, it is clearly in the form of a pipeline when considering the north-south links. For the number of transputers available, and the recommendation by Cok[10], the data must be distributed along the pipeline. The previous experiment compared the performance of the two methods of distributing the data, with the pipeline clearly being the better method for this number of transputers. Considering this pipeline method of data distribution, and the enhanced features of the T800 transputer, as described above, Jones [JON89] compares three methods of data transfer on the links, introducing methods of using the links and processor concurrently. These methods are discussed and compared below, showing how each can be implemented.

In each method data must be input on one link, processed using some process p(), and output on another link. The data used is of any type in a buffer T. This can be either a vector of data or a single data element. The code has been shown here as it is used to establish which method of data transfer is the fastest.

5.3.3.1. Sequential Link Usage

When considering data transfer and processing on a processor the natural method of performing this is to input the data onto the processor, process it and then output it. This is usually thought of in a sequential manner of execution. An example of this method of data transfer follows with the graphical representation appearing in Figure 13.

---

10 Private communication with Ronald Cok of the Eastman Kodak Company.

```
WHILE TRUE
  T x :  -- Buffer x of type T
  SEQ
  in ? x   (1)
  p(x)     (2) -- Some process
  out ! x  (3)
```

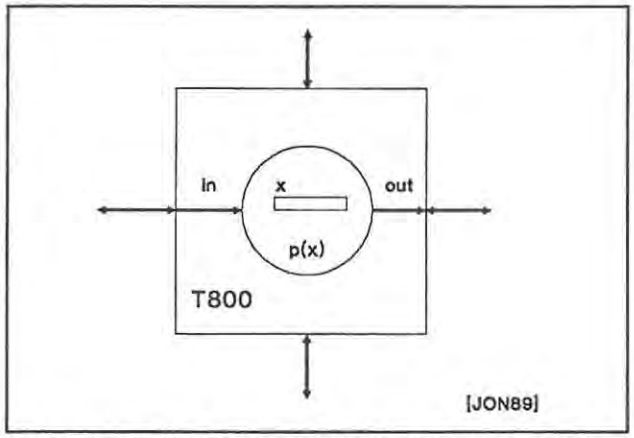

Figure 13 Sequential link usage.

In looking at this simple sequential process, and the fact that the link interfaces can work concurrently with the processor, the following can be noted:

The input link (channel *in*) works while *in* (process 1) executes. The processor works while process *p(x)* (process 2) executes. The output link works while *out* (process 3) executes. Since only one of the three processes (1, 2 and 3) works at a time, the transputer is only working at one third of its capability, if the processor time is equal to the time for transfer of data on one of the links. If the processing time is insignificant in comparison to the data transfer time, then some gain can certainly be achieved. This is, however, still inefficient when considering the fact that the links and the processor can operate concurrently.

5.3.3.2. Parallel Link Usage

The second method discussed by Jones [JON89] utilizes the links on the processor concurrently. This is an adaptation of the three processes above, where they are now made to run concurrently. Figure 14 gives a graphical representation of these three processes. The code for this follows:

```
CHAN OF T c.xy, c.yz :

PAR
  WHILE TRUE      (1)
    T x :
    SEQ
      in ? x
      c.xy ! x
  WHILE TRUE      (2)
    T y :
    SEQ
      c.xy ? y
      p(y)
      c.yz ! y
  WHILE TRUE      (3)
    T z :
    SEQ
      c.yz ? z
      out ! z
```

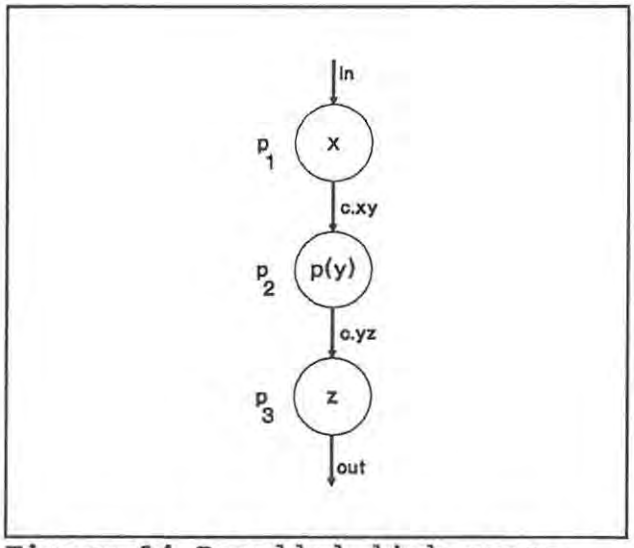

Figure 14 Parallel link usage.

Running these three processes in parallel offers a significant improvement in the performance. This improvement is, however, not quite a three fold one, which can be attributed to the following reasons:

1. The three processes must spend time in communication on channels *c.xy* and *c.yz*, although these are internal channels. The speed at which the transfer can be made is dependant on the memory speed. This will occur on the memory external to the transputer which, although fast, is not as fast as the internal memory. The amount of data being transferred is also a factor influencing the time of this memory transfer. The transfer time becomes significant when the amount of data is large.

2. The processor is guaranteed to be *otherwise occupied* for two block move times per processor call.

3. The link engines are guaranteed to be idle for one block move time per link communication.

Considering point 1 above, Jones [JON89] proposes a new method of data transfer within the processor. For significant amounts of data, this method is guaranteed to save time.

5.3.3.3. Saving Data Transfer Time

Since the data transfer in memory can take up a significant portion of time, Jones [JON89] proposes the use of pointers for the transfer of the data. As pointers are not available within occam, array indexing serves the purpose equally well. It is quicker to transfer the index into the array than the complete data packet associated. In keeping the data, pointed to by the indexing into the array, global, the indexes can be passed along the internal channels between the link engines and the processor. The code for performing this method of data transfer appears as follows with the data structure, using this indexing, appearing in Figure 15.

```
VAL INT n IS 3 :
[n+1] CHAN OF INT c :
PAR
  PAR i = 0 FOR n
    SEQ
      c[i] ! i
      WHILE TRUE
        INT ix.wi              (1,2,3)
        SEQ
          c[i+1] ? ix.wi
          c[i] ! ix.wi
  CHAN OF INT new IS c[0] :
  -- Channel aliasing as another
  CHAN OF INT c.xy, c.yz :
  CHAN OF INT used IS c[n] :
  [n] T b :
  PAR
    WHILE TRUE          (4)
      INT ix.x :
      SEQ
        new ? ix.x
        in ? b[ix.x]
```

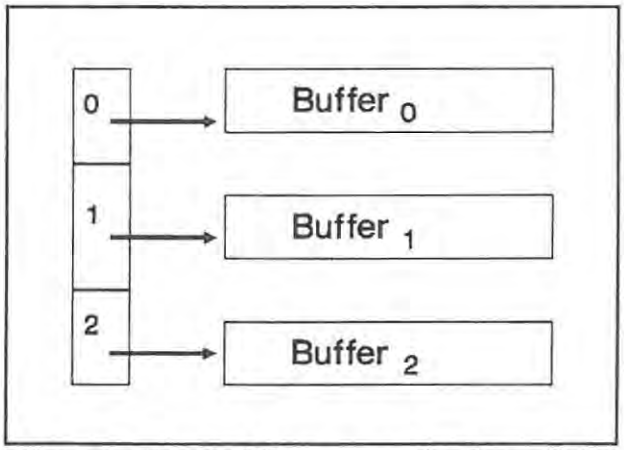

Figure 15 Buffers used for data transfer.

```
    c.xy ! ix.x
WHILE TRUE          (5)
  INT ix.y :
  SEQ
    c.xy ? ix.y
    p(b[ix.y])
    c.yz ! ix.y
WHILE TRUE          (6)
  INT ix.z :
  SEQ
    c.yz ? ix.z
    out ! b[ix.z]
    used ! ix.z
```

The processes that are created can be graphically represented as shown in Figure 16.

In Figure 16, the processes 1, 2, and 3 are responsible for generating the indexing into the data buffer. Processes 4, 5, and 6 perform exactly as those in the second method presented by Jones [JON89], with the exception that they get the data in from the previous processor and the index into this data from the index generating processes (1, 2, 3). The indexing into the data buffer rotates around the network shown here. Thus for three buffers of data, the indexing is generated as 0, 1, 2 and rotated in this order as the data is input, processed and output.

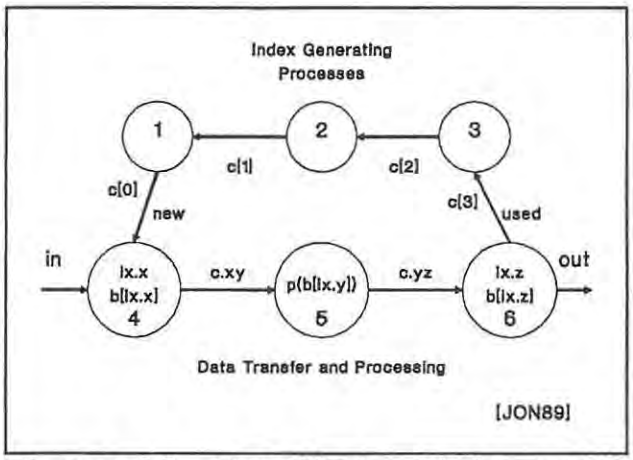

Figure 16 Parallel link usage with indexing buffers.

It should be noted that in this parallel implementation, more context switching will be introduced with these extra processes being present. This should be carefully monitored as it can present a significant overhead in the execution time if the data being transferred is small and there is not much processing to be done. In a case like this, the first method of using the link engine and processor concurrently should be implemented. The experimental results will show this.

When considering this method of data transfer, a summary of the times involved can be made and conclusions drawn from this.

Let

t_p	=	processor time
t_i	=	input time
t_o	=	output time
t_t	=	index transfer time
t_{index}	=	time for generating and transferring the indexing into the array
$t_{context}$	=	time for the context switching of the extra processes

Since the time for transferring the indexes of the arrays, t_t, occurs between the link engine and the processor process, it must be included in the processor time.

The index generation time, t_{index}, can be included in the input and output engine times t_i and t_o.

The ideal execution time would be when

$$(t_i \text{ or } t_o) + t_{index} \approx t_p + t_i$$

Although the input and output engine time normally overlap onto the processor time, care must be taken if the processor time is insignificant in comparison to the link engine time and vice versa. In these circumstances it may not be worth while introducing the extra overheads from the context switching.

5.3.4. Testing the Distribution Methods

The following experiments are performed to test the three data transfer methods discussed. In all the tests, the results appear in the following order:

1. Sequential execution.
2. Parallel execution with memory transfer.
3. Parallel execution using indexing.

5.3.4.1. Investigating Context Switching Overheads

In this test, the data is kept to the smallest possible size (one BYTE per data element) and the processor operating on the data is kept to a single SKIP process, thereby allowing the fastest possible execution time. This data is sent 1000 times to get a good resolution on the time. All times shown are in seconds.

1. 0.0392
2. 0.0904
3. 0.0979 seconds

5.3.4.2. Single Processor Test

A single processor is used to test the three methods discussed. In order to get data on and off of the processor, two other processors are used, merely as output and input devices. For the test two different vector lengths and two different amounts of computation in the processor are used. From this it can be seen what happens when there is a lot of data with both a lot of processing to be done and also for little processing and vice versa. The vectors used contain 32 bit integer values and the two lengths are 10 and 1024 elements respectively. This introduces some significant

variation in the size of the data. The two processes for checking the processor time are a SKIP statement for the shortest processing time and the following process:

```
SEQ i = 0 FOR vector.length
  vector[i] := (vector[i]+Ten)-Ten
```

The process above, introduces a fair amount of computation time when operating on each vector element and effectively leaves the vector elements unchanged.

The two vector lengths used contain 10 and 1024 integers respectively. For each of these lengths two amounts of computation are performed as discussed above. The results for these follow. All times shown are in seconds.

10 Integers

	SKIP statement	**Computation**
1.	0.5845	0.5855
2.	0.0691	0.0777
3.	0.0653	0.0750

1024 Integers

1.	9.5243	11.2373
2.	5.4279	6.8450
3.	4.5320	6.1489

It is evident from the results that the parallel link usage is the only way to perform the data transfer. When comparing the results for the short and long vectors, it is obvious that a large amount of time is spent on the links compared to the amount of computation being performed when the long vectors are used. This causes the ratio in the differences in the sequential and parallel times to be less for the long vectors than for the short. Looking at the differences in times for the two parallel methods, one would expect a vast improvement in the time for the method using the transferring of the indexes of the data buffers when large data vectors are sent. Although there is an improvement, the introduction of the overheads, from the context switching of the index generating processes, causes some delay.

Noting this delay, care should be taken as to which of the two methods is used. For small data elements, the first of the parallel methods would clearly be the better choice. Also, faster memory, although expensive, would allow faster data transfer. This will be especially helpful in the method where data is transferred in memory. The method employing the transfer of the indexes would only be necessary in cases where large amounts of data need transferring.

5.3.4.3. Network Test

The test is performed on the complete network used in this project. Each of the three methods discussed is placed on a *worker* processor in the network. The *controller* processor sends the data to the *workers* and receives the returning data. The code on the *controller* process follows. The *worker* code is simply an adapted version of the three extracts shown earlier. The vectors are sent around the network 1000 times in order to obtain a reasonable time.

```
PAR
  SEQ i = 0 FOR 1000
    n.out ! vector; vector.one
  SEQ j = 0 FOR 1000
    e.in ? CASE
      vector; vector.two
        SKIP
```

Variant protocols [BUR88] are used in order to eliminate the severe overhead of synchronization when individual data elements are sent.

The above process, placed on the *controller* processor, can be depicted graphically as shown in Figure 17.

The transmission of the data must be run in parallel with the reception of the returning data. This allows the pipeline to be completely filled with data. While data continues entering the network, the *e.in* link can gather the data independently.

The following extract of code shows how the data is sent.

```
SEQ i = 0 FOR 1000
  PAR
    n.out ! vector; vector.one
    e.in ? CASE
      vector; vector.two
        SKIP
```

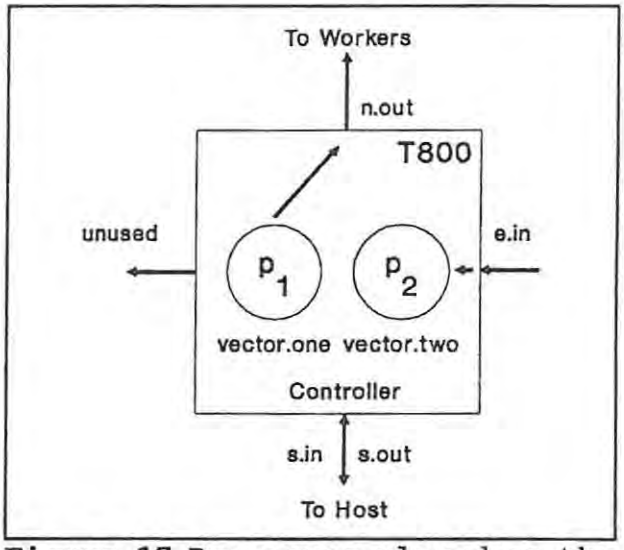

Figure 17 Processes placed on the controller processor.

Since the statement controlling the number of times data is transmitted and received, i.e. SEQ i = 0 FOR 1000, is at the outermost level, absolutely no parallelism is achieved. Although there is a PAR construct, the vector that is transmitted must be received, before the next vector can be transmitted. This test provided very poor data transfer times. The times for the parallel methods are slower than for the sequential method. This is as a result of the lack of parallelism at the outermost level, causing purely sequential execution to take place. Context switching times are also introduced by the presence of the PAR statement. The first of the two parallel methods

proves to be the slowest method of data transmission. Not only is an overhead introduced by the context switching, but the memory transfer adds a great deal to the execution time. The results for this test are shown below.

Vectors containing 1024 32 bit integers are sent around the network 1000 times. The first set of results show the times for the process having only a SKIP statement and the second are for the process having the extra computation added, as shown earlier. All times shown are in seconds.

	SKIP statement	Computation
1.	22.7735	29.6070
2.	26.3802	34.0016
3.	23.8382	30.2002

These times, when compared to those previously shown for the network, are a good indication of the lack of parallelism. Where parallelism has been introduced, within a processor, the overhead as a result of the context switching and the memory transfer becomes evident.

Again, the two vector lengths of 10 and 1024 integers are used, each with two amounts of computation. In the experiment where true parallelism is used on the links, the times show dramatic improvement, and the following results are shown. All times shown are in seconds.

10 Integers

	SKIP statement	Computation
1.	0.1021	0.1183
2.	0.0693	0.0855
3.	0.0656	0.0762

1024 Integers

1.	9.1407	10.8561
2.	5.5390	7.1570
3.	4.6507	6.1705

In comparing these times to those where only one processor (with the two for input and output) is used, an increase in time is evident, but not dramatic. This is as a result of the fact that extra links are used requiring some extra time for propagation across these links and for synchronization between neighbouring transputers. Furthermore, time is taken in filling the pipeline and flushing it at the end. The results of this experiment indicate clearly that the parallel

use of the links and the processor are the obvious method of transferring data between processors.

5.3.5. Clean Process Termination

In each of the methods discussed by Jones [JON89], the data transmission process makes use of infinite looping processes, using the WHILE TRUE statement. This is all very well until a set number of data elements must be transmitted. This is the case in the experiments performed where 1000 vectors are sent. The only way the processes can be terminated with the WHILE TRUE statements being used is by resetting the processor. This is certainly not a very elegant way of terminating the process. This problem of terminating parallel processes is addressed by Welch [WEL89]. He mentions that this is particularly common in occam programs and associated transputer networks.

This method of allowing a set number of data elements to be transmitted can be implemented by replacing the WHILE TRUE statement with the following.

```
SEQ i = 0 FOR number.of.times
```

However, in the third method, simply replacing the code with the SEQ construct, as shown above, does not overcome the problem. Careful inspection of the code will reveal that clean termination of the processes which generate the indexes will not happen because a final index value is output by the process controlling the output link engine. This index value must go back into the processes generating the index as it forms part of the continual wrapping around of these. The problem of taking up the extra index value can be solved by inserting an extra input statement in the index processes.

Jones's code for this third method can be improved as follows, to overcome the problem discussed above:

```
PAR
  PAR i = 0 FOR n
    INT dummy :   -- To soak up the extra value
    SEQ
      c[i] ! i
      SEQ i = 0 FOR 999 -- 1000 would generate 1 too many
        INT ix.wi :
        SEQ
          c[i+1] ? ix.wi
          c[i] ! ix.wi
      c[i+1] ? dummy     -- Soak up the extra value
```

This now offers clean termination of all the processes.

The extra processes, added by Jones [JON89] in the third method, introduce a further overhead in terms of context switching of the processes running concurrently on a single processor. This could be overcome by eliminating the index generating processes and incorporating their operation within the three processes controlling the input, processing, and output. The index

values could simply be rotated within each process. Alternatively, a channel could be created to link the output process back to the input to wrap the value around. Although this method of programming does not exactly follow the occam paradigm (in terms of elegance of concurrent processes performing this task), it will nevertheless eliminate the context switching overhead.

Chapter Six

6. The One-Dimensional Fast Fourier Transform

The Fourier Transform forms a fundamental part of the signal processing environment. It allows the user the ability to analyze signals in order that operations can be performed on these signals. These operations allow the signal to be improved or altered according to some higher level process in an application. The same holds true in image processing, as will be described later. The transform of particular interest in this project is that of the Fast Fourier Transform (FFT). Several authors [ORA74][BRA78] describe the FFT as an efficient way of computing the Discrete Fourier Transform (DFT). The DFT is adapted from the Continuous Fourier Transform in order that a sampled signal can be transformed.

The one-dimensional FFT was implemented as part of this project primarily to compare processor ability in the signal processing environment, because of the computational load which it introduces. Also, it has allowed a familiarization of the algorithm for use in the two-dimensional FFT, the original application intended. The two-dimensional FFT can be performed as a series of one-dimensional FFT's [GON77]. This is described in detail in Chapter 7.

Oran Brigham [ORA74] offers a good detailed description of the FFT and its applications in the signal and image processing environment. A brief mathematical description of how the FFT is arrived at from the DFT follows:

The DFT is defined as follows:

$$X(n) = \sum_{k=0}^{N-1} x_0(k)e^{-j2\pi nk/N} \qquad n = 0,1,2.....N-1$$

The FFT is arrived at by replacing the exponential part of the DFT with

$$W = e^{-j2\pi/N}$$

By expanding the adapted formula in terms of matrices involving the W component, the following can be arrived at

$$X(n) = W^{nk}x_0(k)$$

Careful inspection of the W component reveals that reduction in computation arises from the fact that there exists a relationship in the W component of

$$W^{nk} = W^{nk \text{ MOD } N}$$

This reduction in computation and the ease thereof makes the FFT a good transform for Fourier analysis. A full description of this transform and the proof is offered by Oran Brigham [ORA74]. Although the one-dimensional FFT is not considered for parallel implementation here, it is worth noting the interest it has generated in this domain of computing.

The nature of this Fourier Transform's method of computation lends itself towards a *butterfly* format of computation. This has led several authors to discuss parallel implementation of the one-dimensional FFT, although little in terms of results is published. Pease [PEA68] indicates the adaptation of the transform to parallel processing as far back as 1965. Quinn [QUI88] describes the FFT and the *butterfly* architecture that a parallel implementation of this algorithm is suited to. He mentions that the transform has the important characteristic that no result can be computed without examining every input. This, together with the butterfly format of computation has influenced his choice of architectures to be that of a *butterfly* network. He shows the parallel implementation of the algorithm on a cube connected processor array network, the **SIMD-CC** network. Hockney [HOC88] indicates that a high premium in performance can be achieved if the parallelism of the algorithm can be made to match the hardware parallelism of the array, i.e. the number of processors in the array. Briggs and Sale [BRI89] describe a parallel implementation of a one-dimensional FFT, exploiting the *butterfly* format. The authors use a hard coded implementation, thereby using the exact number of processors as the number of *butterfly* operations, to map the algorithm onto. Here the twiddle factors[11] are precalculated and the multiplications and additions are placed on each processor. In seeking a general solution this method of implementation is far too terse.

Several authors also describe the implementation of the FFT on dedicated hardware. Karwoski [KAR88] shows an implementation of this transform on digital signal processors using a *butterfly* arithmetic architecture. A pipeline FFT, again hardwired onto signal processing devices, is also described by Groginsky and Works [GRO70]. Although the results that are obtained on dedicated hardware such as that indicated here are impressive, the parallel implementation of the transform algorithm is not sought and the use of signal processing hardware is not considered.

The repetitive use of the one-dimensional FFT in obtaining a parallel implementation of the two-dimensional FFT is in keeping with the approach of Gonzalez and Wintz [GON77], and the fact that *image parallelism* [MOR88a] is to be used. The motivation for the use of *image parallelism* is described in detail in Chapter 7. It can, at this stage, however, be attributed to the fact that a general purpose solution for implementing image processing techniques on a pipeline of processors is sought.

Faster implementations of the Fourier Transform, such as the Fast Hartely Transform do exist. This was first described by Bracewell [BRA78]. O'Neill [ONE88] describes this implementation,

[11] The twiddle factors are the bit reversal values and weights (W) which are precalculated to enable faster processing.

indicating that in comparison to the FFT, half the computer resources are used and that this algorithm is twice as fast as the FFT. It could thus be argued that this algorithm should have been implemented in place of the FFT. However, considering the availability of the source code, both locally [JON86] and by authors such as Oran Brigham [ORA74], as well as already implemented versions allowing for verification of correctness, the FFT was the preferred algorithm for test purposes.

What follows are the benchmark times for the implementation of the algorithm executed on an 8 MHz INTEL 80286, both with and without the INTEL 80287 numeric coprocessor and the T800 transputer.

The times are individually shown for the computation of the FFT component and the Magnitude and Phase component. These are shown under the FFT and MAG rows respectively.

6.1. One-dimensional FFT on a 8 MHZ 80286 AT

As Turbo Pascal (v4.0) does not accommodate the emulation of reals in the IEEE 754 standard of 4 and 8 bytes, tests are run for 6 byte values without the use of a numeric coprocessor and for 4 and 8 bytes real values with a numeric coprocessor.

The following times are shown for the calculation of the FFT component and the Magnitude and Phase component for three different sizes of data points. The calculations are performed on the 80286 and 80287 machines for the different size real values. All the times shown below are in seconds.

Data Points		4 Bytes (287)	6 Bytes (286)	8 Bytes (287)
256	FFT	0.6	1.72	0.68
	MAG	0.1	0.99	0.11
512	FFT	1.3	4.01	1.54
	MAG	0.2	1.98	0.22
1024	FFT	2.9	9.06	3.40
	MAG	0.4	4.01	0.44

The same tests are performed on a 16 MHz MITAC 80286 machine. The following results are shown with all times in seconds.

Data Points		**6 Bytes (286)**
256	FFT	0.66
	MAG	0.33
512	FFT	1.59
	MAG	0.77
1024	FFT	3.62
	MAG	1.54

It appears that the computation times are longer for more complex input signals, i.e. more interesting signals. This is evident in that it is easier to multiply by a zero value than a large 4, 6 or 8 byte value.

The standard input signal used for these tests was a symmetrical *top-hat* function. The use of a symmetrical signal eliminates the phase component and the expected resultant *sinc* function can easily be seen.

If the program is compiled for use with a numeric coprocessor, the size of the real values used must be changed from 6 bytes to either 4, 8, or 10 bytes in accordance with the IEEE 754 standard. If this is not done, longer computation times occur as a result of the conversion to either 4, 8, or 10 byte values for the numeric coprocessor to operate on. Tests show what sort of degradation occurs when this is done.

The times, below, for the one-dimensional FFT routine when 6 byte reals are used in conjunction with the numeric coprocessor indicate the extent of the degradation. The Magnitude and Phase times are not shown.

Data Points	**FFT**
256	2.25
512	5.16
1024	12.14

A slow down of approximately 30% can be seen. This is something which must be carefully avoided. The reason for the 30% slow down is that the 6 byte reals have to be converted to the 4 or 8 byte reals when the numeric coprocessor is used which takes some time.

6.2. One-dimensional FFT on a Single T800 Transputer

The results for the one-dimensional FFT executed on a single T800 with 4 MBytes of memory in occam2, using the same algorithm as the Turbo Pascal version are shown below. The program has been run as a single sequential process. Comparisons of these results can be made against the results of the Turbo Pascal version run on the AT.

The table below shows the times for the FFT and Magnitude operations on the T800. The total turn around time for getting the data to the transputer, transforming it, and getting it back to the host is shown as the **AT time**. The reason for doing this is to establish whether it is actually worth sending the data to the transputer across the link considering that this is the bottleneck in the system. The **Data in** and **Data out** times are the times obtained for transferring the data to and from the transputer. The results are grouped for the various word sizes used on each architecture. The two word sizes for the real values on the transputer can be seen. They are 32 and 64 bit real values respectively. In each of these 16 and 32 bit integers have been used.

Data Size	AT time	Data in	T800 Times FFT	MAG	Data Out
32 bit Real values on the T800					
16 bit values on the AT					
256	0.11	0.0155	0.0401	0.0071	0.0420
512	0.22	0.0310	0.0800	0.0143	0.0838
1024	0.39	0.0621	0.1601	0.0296	0.1679
32 bit values on the AT					
256	0.11	0.0189	0.0401	0.0071	0.0526
512	0.27	0.0378	0.0801	0.0143	0.1050
1024	0.44	0.0758	0.1602	0.0296	0.2100
64 bit Real values on the T800					
16 bit integers on the AT					
256	0.11	0.0154	0.0467	0.0103	0.0420
512	0.22	0.0310	0.0938	0.0207	0.0839
1024	0.44	0.0621	0.1905	0.0426	0.1679

Data Size	AT time	Data in	T800 Times FFT	MAG	Data Out
32 bit integers on the AT					
256	0.17	0.0190	0.0467	0.0102	0.0525
512	0.22	0.0378	0.0939	0.0206	0.1050
1024	0.55	0.0758	0.1905	0.0424	0.2100

The two sets of times for the 16 and 32 bit integers being sent from the AT represent the pixels. As the pixel intensity does not exceed 256, implying that the word size for the pixels are bytes, the integers sent from the AT need only be bytes. This has two advantages. Storage space on the AT can be saved and quicker transmission of data from the AT to the transputer can be achieved.

It is important to note that some time is necessary for the conversion of the integers to real values on the transputer. More time is needed to convert to 8 byte than 4 byte real values. As the resolution of the 4 byte values is adequate for the purposes of the computation, as well as requiring less storage space, this word size on the transputer should be chosen.

It is evident from the figures tabulated above that the only time saved when using smaller word sizes on the AT side is the time of data transfer. The computation time still stays relatively constant on the transputer. Noting the gain in data transfer time and data storage saving on both the AT and transputer when using the smaller word sizes, it is obvious that this should be used for the purposes of computation.

Figure 18 shows a direct comparison of the processor ability at computing the FFT on each processor. A logarithmic plot of the times are shown as the T800 results are insignificant in comparison to those of the 80286 and 80287 when depicted graphically. From this diagram it is evident that the T800 has the ability to act as a coprocessor to the AT for computations heavily dependant on real arithmetic.

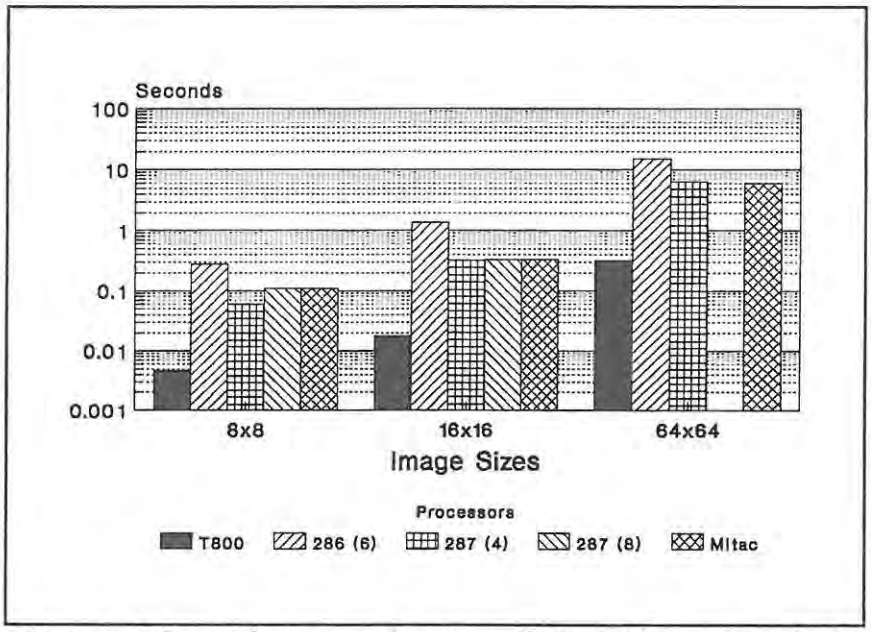

Figure 18 Times shown for the various architectures

It is still important, though, to consider the type of operation to be performed and the time taken for this operation, in order to decide upon the machine on which to execute the task. Should the time taken for the data transfer and the computation, when performed on the transputer, be found to be less than, or equivalent to that for same computation performed on the AT, the operation is better performed on the transputer. The data could be left on the transputer and only the results returned to the AT for analysis or display to save time.

The one-dimensional FFT has proven to be a good testing ground for processor ability. The ability of the T800 to out perform the AT in computations which are heavily dependant on real arithmetic and the fact that there is a bottleneck in the link becomes an important issue when the two-dimensional FFT is to be implemented on a network. The same philosophy regarding the criteria for choosing processor on which to execute the task will, thus, still apply.

Chapter Seven

7. The Two-Dimensional Fast Fourier Transform

The two-dimensional Fourier Transform forms a fundamental part of the image processing environment. It is mostly implemented in the discrete form, known as the Discrete Fourier Transform[12] which is an approximation of the continuous Fourier Transform. The Fourier Transform implemented in this project is that of the Fast Fourier Transform[13]. The two-dimensional FFT is derived from the two-dimensional DFT just as the one-dimensional FFT is, as was shown in the previous chapter. The formula for the DFT of an MxN matrix with substitution of the W component is defined below [JON86]. The full derivation and proof of the two-dimensional formula can be seen in [GON77].

$$H_{k,l} = \sum_{j=0}^{N-1} \sum_{i=0}^{M-1} h_{i,j} W^{(-ik/M)(-jl/N)}$$

where $W = e^{2\pi\hat{\imath}}$ and $\hat{\imath} = \sqrt{-1}$

The resultant matrix $H_{k,l}$ cannot be displayed in its current form. The Magnitude matrix has to be computed from this, taking into account the fact that the origin of the transformed image is at the center of the matrix and has symmetry in the left and right halves of the resultant Magnitude matrix.

The implementation of the two-dimensional FFT for this project is based on the work done by Jonas [JON86]. There are two reasons for this: This work was done in the Department of Physics and Electronics at Rhodes University, thereby making the source code easily available for use. Also, the availability of the VAX on which this FFT was implemented allowed a direct comparison to be made, as well as aiding the debugging of the FFT implemented on transputers. Since a direct comparison is made with the work of Jonas, a short description of his work is given here, indicating its performance relative to the implementation of this thesis.

The two-dimensional DFT is a computationally intensive algorithm, even if optimized for the best computational performance. Jonas [JON86] talks of the computational expense being due to the number of arithmetic operations and the amount of computer memory needed for the storage of image arrays (a typical example is a 256x256 pixel image of 8 bits per pixel requiring 64 KBytes of storage). Jonas [JON86] also mentions that a 512x512 pixel image can be computed

---

[12] The Discrete Fourier Transform will be referred to as the DFT.

[13] The Fast Fourier Transform will be referred to as the FFT.

in less than a day on most machines. Because image processing tends to be interactive, he goes on to say that *less than a day* is not necessarily fast enough for many applications. It is the intention of this experiment to investigate the transform implemented by Jonas [JON86], to make comparisons of the performance obtained, and to suggest possible methods of improving this.

Jonas [JON86] describes how his implementation of the two-dimensional DFT is designed for image processing applications running on a virtual memory machine. This takes advantage of the fact that images do not have imaginary components, since the images are normally derived from the outputs of intensity sensors. Typically these sensors are cameras consisting of Charge Couple Devices, more commonly called CCD cameras. Each sensor on the camera provides a single pixel intensity.

Jonas [JON86] implemented the two-dimensional Fourier Transform on a VAX 11-730 virtual memory machine, under FORTRAN 77 (highly optimized). The Cooley-Tukey [ORA74] algorithm formed the basis of the two-dimensional implementation, essentially performing a one-dimensional FFT on all the columns and then all the rows (or vice versa), each time making use of the necessary bit reversal and weight tables. Gonzalez and Wintz [GON77] show that the transform can be computed using this method because of the separability property allowing successive applications of the one-dimensional FFT to be made. Jonas [JON86] indicates that his implementation performs $MN/2\log_2 MN$ butterfly calculations as apposed to twice the number for the full complex FFT where a real and imaginary vector are used for each row or column. The method of transform described above applies equally to the inverse transform. This will not be dealt with in this project, as the algorithm and its computational complexity is very similar to the FFT.

In FORTRAN on the VAX 11-730, arrays are stored in column major order, whereas in Turbo Pascal and occam2 the ordering is row major. Jonas [JON86] mentions that the column transforms are efficient on a virtual memory machine. Only two rows (or columns) are operated on at a time and this pair of vectors should fit into the working set of the process. The columns are easily mapped into a linear form of memory access, hence the efficiency. The rows, however, are different because of the mapping from the pages in which the columns reside. Jonas [JON86] indicates that in the worst case situation, the memory requirement for the active data is N pages, where N is the number of columns. The 11-730 has 2048 pages of 512 bytes per page, giving a physical memory of 1 Megabyte. Thus performing a 512x512 DFT requires little page swapping if the machine is not busy. Extending the page size will degrade the performance considerably.

Figures are shown below for the performance of the FFT on the VAX 11-730, with 1 Megabyte of memory and no floating point processor. The times shown are in seconds.

n	$N=2^n$	NxN	T_{elapse}	T_{cpu}	page faults
4	16	256	0.45	0.42	20
5	32	1024	1.65	1.62	29
6	64	4096	7.10	6.97	62
7	128	16384	32.49	32.46	1135
8	256	65536	157.79	155.07	11574
9	512	262144	884.20	809.32	103677

Jonas [JON86] mentions that although no floating point processor was available at the time of experimentation, the results obtained were reasonable for his image processing requirements. A floating point processor would make a considerable difference to the results listed[14]. The page faults are also responsible for some of the time degradation.

7.1. Performance Tests on the AT and a Single T800

The times shown below are for the same algorithm translated from FORTRAN into Turbo Pascal (v4.0) on the AT and occam2 on the transputer. FORTRAN is considered the most efficient high level language implementation on the VAX. Likewise Turbo Pascal on the AT and occam2 on the transputer. The code for the Turbo Pascal (v4.0) and occam2 versions of the algorithm are shown in Appendix C and Appendix D respectively.

Two times have been obtained on each run; the time for the FFT and that for the computation of the Magnitude. The times shown are all in seconds.

Times for the Turbo Pascal version at 8 Mhz on a 80286 AT:

	4 Bytes (287)		**6 Bytes (286)**		**8 Bytes (287)**	
Size	**FFT**	**Mag**	**FFT**	**Mag**	**FFT**	**Mag**
8x8	0.06	0.00	0.28	0.05	0.11	0.06
16x16	0.32	0.05	1.37	0.39	0.33	0.05
64x64	6.32	0.49	15.05	6.26	Too big for memory	

The same test on a 16 Mhz MITAC AT without numeric coprocessor using 6 byte real values produces the following results:

[14] The T800 has an on-board 64 bit floating point processor.

	6 Bytes (286)	
Size	**FFT**	**Mag**
8x8	0.11	0.00
16x16	0.33	0.16
64x64	6.09	2.53

The (287) next to the 4 and 8 byte real values indicates that a floating point processor (INTEL 80287) is used for 4 and 8 byte reals. No floating point hardware was used for 6 byte reals.

The vast differences in the times on the two test machines, can be attributed partly to differences in memory speeds and the number of wait states. Also, the resolution of the 1/18 th of a second clock on the 80286 machine is partially responsible for the large time differences between the longer times on the 8 and 16 MHz machines.

The times for the occam2 version of the algorithm run as a sequential process on a single T800 with on board floating point hardware are shown below. The times shown are in seconds.

	4 Byte Reals	
Size	**FFT**	**Mag**
8x8	0.0047	0.0013
16x16	0.0177	0.0053
32x32	0.0736	0.0205
64x64	0.3230	0.0866
128x128	1.4038	0.3452
256x256	5.7109	1.2914

Figure 19 offers a graphical representation the times obtained for the FFT executed on the various architectures as tabulated above. The Magnitude and Phase times are not shown in the graph. The timings are only shown up to the 64x64 pixel image as this is the upper limit for the AT. An immediate indication of the relative performances of these architectures is visible. A logarithmic scale is used for the times as there is a vast difference in performance speeds between the T800 and the standard 80286 processor operating with and without a coprocessor. On a linear scale the difference is so vast that the time for execution of the FFT on the T800 is hardly visible.

In considering the AT (80286) running at 8 MHz without a numeric coprocessor and then with a numeric coprocessor (80287), obtaining a speedup of approximately 3 times, the AT can be considered as being a fairly powerful *number crunching* machine. The time for comparison must be that for the FFT and the Magnitude combined. The time obtained for the 6 byte real values, without the numeric coprocessor, is somewhat slow, at approximately 21 seconds for a 64x64 pixel

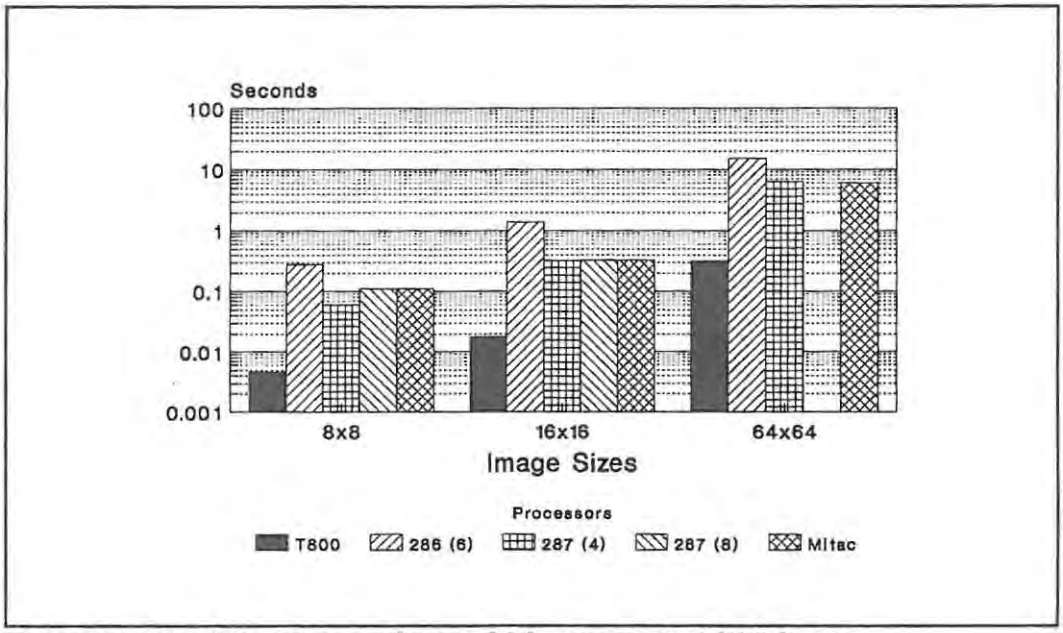

Figure 19 FFT times for T800 versus 80286 AT.

image. However, when considering that when a 80287 numeric coprocessor is added the times start to match that of the VAX, the AT starts becoming more viable for small scale image processing. It must be remembered, though, that the AT has a distinct disadvantage because of the presence of the 64 Kbyte page boundaries. In order to overcome this limitation, dynamically allocated pointers could be used for the vectors. This will, however, introduce longer computation times and will require intense garbage collection.

From the times shown for the 16 Mhz AT, it is obvious that with a numeric coprocessor on a machine like this, speeds equalling or exceeding those on the VAX will be obtained. Once again, however, taking into account the constraining factor of the 64 KByte page boundary, the AT is only viable for small scale image processing applications.

When considering the results obtained for a single T800 processor, running the same algorithm as that run on the VAX, a more concise conclusion can be drawn as to the viability of the transputer for this type of operation. Although the transputer has a floating point processor as an integral part of the hardware, whereas the results for the VAX 11-730 were obtained without a floating point processor, the fact that the transputer and AT combined offers a portable processing unit, at considerably less cost, as well as being expandable (creating networks of transputers), for applications such as the one under discussion, the viability is immediately evident. It must, however, not be forgotten that in such a system, there still exist several limiting factors, not immediately apparent in the context of this experiment. Although the T800 can have memory addressable to 4 Gigabytes, the cost of memory allows a practical limit of up to about 4 Megabytes per transputer. Virtual memory is not within the domain of this architecture, although the physical memory available has the advantage of being contiguous and linearly addressable. Distinct advantages can be seen over having to perform page switching as in architectures such as the VAX 11-730.

The fact that the transputer, although it can be run as a single processor, is designed with parallel processing in mind, introduces further advantages. The construction of networks of transputers allows parallel processing to be fully exploited. This introduces the concept of distributed or shared memory systems. Shared memory systems are not present in transputer based systems. This is because there would be a need for complex hardware for crossbar switching and for performing arbitration. Distributed memory (**MIMD**) systems would be preferred for image processing applications. This allows situations where memory contention could occur to be avoided. It could be argued that a shared memory system would be a better choice for image processing because of the access to a large unit of data, that is the image being processed. However, as data distribution is not difficult, nor is it computationally expensive in transputers, and with the memory requirements being large for image processing, a distributed memory system appears more viable. The transputer offers point-to-point communication allowing for ease of data distribution. Briggs and Sale [BRI89] indicate that a distributed memory system avoids memory access problems and synchronization problems. However, with distributed systems care must be taken to avoid too much time being spent on the communications between the processors, that is, the grain of parallelism must not be too fine [CHE89].

7.2. Tests on a Network of Transputers

Since the project has a workstation for image processing using a network of transputers in mind, the investigation into the execution of the FFT on a network topology was considered.

In the context of this project, dynamically reconfigurable networks [DAS88] were not considered, as the hardware available does not facilitate this. It was also not the intention of this project to produce a fully fledged image processing workstation, but rather to provide evidence of the viability of such a system.

Following the networks proposed by Morrow et. al [MOR88] and Cok [COK88], the network chosen for the purposes of this experiment was the pipeline configuration. This allows for easy expansion and data distribution. The intention was to investigate methods of performing the two-dimensional FFT on the network used in the project using *image parallelism* [MOR88a] in such a manner as to exploit the parallelism to gain performance improvement.

The experiments involving the distribution of the FFT on the network of transputers have involved several stages of successive improvement and *fine tuning*. All the experiments have been implemented on the pipeline configuration involving different numbers of processors. This has been done to determine an optimum number of processors for operations of this nature, that is, involving extensive computation. The choice of the pipeline of processors was discussed in the section concerning the investigation into the network topology. All the experiments involving the FFT on the network of transputers make use of *image parallelism*. This is largely due to the results obtained by Morrow and Perrot [MOR88a] and the conclusions drawn by these authors in their experiments. Another factor influencing this choice of parallelism is the ease of programmability, considering that the Fourier Transform, especially the two-dimensional Fourier

Transform is not a trivial programming task. Also, in seeking a general purpose solution for image processing, and the topology chosen, the grain of parallelism would become too fine if *task parallelism* were used and because of the necessity to keep the data, either a row or column, as a single entity. The difficulties involved in distributing the actual algorithm across several processors would introduce severe restrictions on the choice and flexibility of the topology. Trying to shuffle the data of a row or column around a network in order to keep it together would simply introduce severe overheads into the links. Although the topology, as indicated before, resembles that of the Perfect Shuffle Machine [SCH89], to suddenly break away from the concept of *image parallelism*, utilizing an already sequential algorithm, would simply involve a major rethink of the style of programming already considered.

Thus, considering the fact that a pipeline of processors has been chosen for the topology, although the remaining links allow for the grid structure of the network to be maintained, and the fact that the T800 transputer allows for the links to work concurrently with the processor, the *image parallelism* paradigm is more suitable and is thus the only approach considered here.

The stages of fine tuning and development in the experimentation allow for maximum efficiency in the use of the pipeline to be achieved. This is shown by ultimately arriving at an experiment which exploits the parallelism available in the links and processor on the T800. The steps in the development of an efficient two-dimensional FFT will be shown in the discussion of the experimentation that is to follow.

7.2.1. Initial Testing

The first experiment involving the distribution of the FFT onto the network provides some very good pointers as to how to program the process more efficiently. Each *worker* processor in the network has a complete FFT algorithm. The *controller* processor is responsible for the routing of the data to the *workers* and for getting the results back. Variant protocols [BUR88] are used to allow for the various data types to be sent along the channels on the network. The protocols used are shown below.

The variant protocol declaration for the channel.

```
VAL im.rows IS 512 :
VAL im.cols IS 512 :
VAL im.size IS im.rows*im.cols :
PROTOCOL chan.protocol
  CASE
    N.b.dimen    ; INT    -- Dimensions of the image
    M.b.dimen    ; INT    --
    send.r.image ; BYTE   -- Send rows of image
    send.c.image ; BYTE   -- Send columns of image
    rec.rows     ; BYTE   --
    rec.cols     ; BYTE   --
    rec.a.row    ; [im.cols] REAL32 --   32 bit data matrices
    rec.a.col    ; [im.rows] REAL32 --   32 bit data matrices
    ret.rows     ; BYTE   --
    ret.cols     ; BYTE   --
    ret.r.image  ; BYTE   -- So know for rows
    ret.c.image  ; BYTE   -- For columns
```

```
    comp.image    ; [im.rows][im.cols] REAL32 -- Saves time
    mag.image     ; [im.rows+1][im.cols+1] REAL32
    host.image    ; [im.rows][im.cols] INT16
    m.image       ; [im.rows+1][im.cols+1] INT16
    finished      ; BYTE   --
    strike        ; BYTE   -- Go on strike so don't have to reboot
    fin.stop      ; BYTE   -- Get the stop time
    fin.mag       ; BYTE   -- Get the magnitude time
    start.time    ; INT
    stop.time     ; INT    -- For the actual times
    mag.time      ; INT
:
```

The *controller* process consists of two concurrently operating processes connected by a channel of the variant protocol type shown above. The *transmitter* process gets data from the host (AT) and controls the transmission of the data to the *workers*. The *receiver* process gets the results from the *workers* and communicates these results to the *transmitter* process when further processing s necessary. It is necessary to separate the processes as the compiler won't allow concurrent use of the output of data to the *workers* and input of resultant data from the *workers*.

The image is divided equally amongst the *worker* processors, each processor having a minimum of two rows or columns. The first *worker* in the pipeline is responsible for sending the packets of rows or columns to the neighbouring *workers*. This proves to be very slow because the data is only sent to the neighbour once the full amount for the neighbour is collected. The code for the *worker* and *controller* processors appears in Appendix E. The method of calculating the amount of data each processor should get is based on the number of *workers* and the image size. This is improved in the next stage of experimentation. Once the columns are sent out, the *receiver* process waits until the hybrid results are returned. The *transmitter* is then informed that the rows of the hybrid image must be sent again. The *receiver* waits for the fully transformed image to return before returning it to the host. The computing of the magnitudes is done on one processor on return of the transformed image. It is felt that this is the quickest method of performing this operation, as the amount of data shuffling required to effectively spread the four quadrants of the transformed image out to obtain the magnitudes is too great an overload. This would encompass a grain of parallelism that is too fine. The results shown below indicate that it is in any event a fast computation. It is the FFT algorithm which only really gains the benefit of exploiting some form of parallelism, in this case *image parallelism*.

The protocol here is somewhat wasteful of time as the image is always stored within a 256x256 pixel array, irrespective of the size of the actual image being transformed. The pixels are 4 byte real values, thus indicating the large amount of data always transported on the links.

The results for the experiments are tabulated below. The number of *worker* processors used in the tests and the times for each size are indicated. The efficiencies are given as well, in order to determine which size network is best suited to which size image. The efficiencies are shown as a percentage and are only shown where two, four and eight *worker* processors are used. The Magnitude is not calculated in every test as it is done on the *controller* as a single sequential process and gives the same result each time and has been previously tabulated.

The following results are shown for the test with all the times shown in seconds. The percentage efficiency values are calculated using the formula shown by Morrow and Perrot [MOR88a]. This value indicates the efficiency gained by executing the process on a network of processors. This formula is shown below.

$$\text{Efficiency} = \frac{t_1}{nt_n} 100\%$$

where t_1 = the time for process executed sequentially on a single processor.

t_n = the time for execution on n processors.

Four worker processors simulated on a single T800 transputer. This board has 4 Megabytes of RAM.

Size	FFT	Mag
8x8	0.0464	0.0013
16x16	0.0765	0.0051
32x32	0.1724	0.0193
64x64	0.5154	0.0755
128x128	1.8492	0.2984
256x256	7.2889	1.1872

Single Worker

Size	FFT	Mag
8x8	0.0711	0.0013
16x16	0.1259	0.0051
32x32	0.2703	0.0198
64x64	0.7098	0.0777
128x128	2.2403	0.3082
256x256	Out of memory	

Two Workers

Size	FFT	Mag	Efficiency (%)
8x8	0.0934	0.0012	2.51
16x16	0.1592	0.0045	5.56
32x32	0.3111	0.0174	11.83
64x64	0.7014	0.0682	23.02
128x128	1.8513	0.2698	37.89
256x256		Out of memory	

Four Workers

Size	FFT	Mag	Efficiency (%)
8x8	0.0947	0.0012	1.24
16x16	0.1669	0.0045	2.65
32x32	0.3246	0.0173	5.66
64x64	0.6957	0.0676	11.61
128x128	1.6722	0.2676	20.98
256x256	4.6086	1.0639	30.97

The immediate conclusion that can be drawn from the efficiency figures in these results are that two *worker* processors are more efficient than four for the sizes of data set tested. Figure 20, gives a representation of the improvement in execution times as more processors are added. The cost effectiveness can also be determined from this sort of graph. This will be discussed later for an improved implementation of the FFT.

The timings for some of the 256x256 pixel image tests are not included as memory limitations precluded the execution on every network. These are indicated as *out of memory* in the results tables. The efficiency figures for this experiment are still, however very low. The reasons for this as well as other findings on this experiment are discussed below.

Each *worker* processor (T800 transputer) has only 256 KBytes of memory available. It is only the *controller* processor (or Root board) which has a large amount of memory (either 2 or 4 MegaBytes) primarily because a large amount is required when TDS runs on the root transputer while a user process is executing. This first experiment has indicated that as the Fourier Transform is expensive on memory, care must be taken in the use of this memory. The initial program made use of separate matrices for the rows and the columns, whereas a common storage area could be declared for the rows and columns being transformed, as these are never present at the same time on a *worker* processor. An attempt has, however, been made to keep the matrix size down by declaring the sizes to be dependant on the number of rows or columns in the image and the number of processors in the pipeline. Thus, the largest matrix for holding

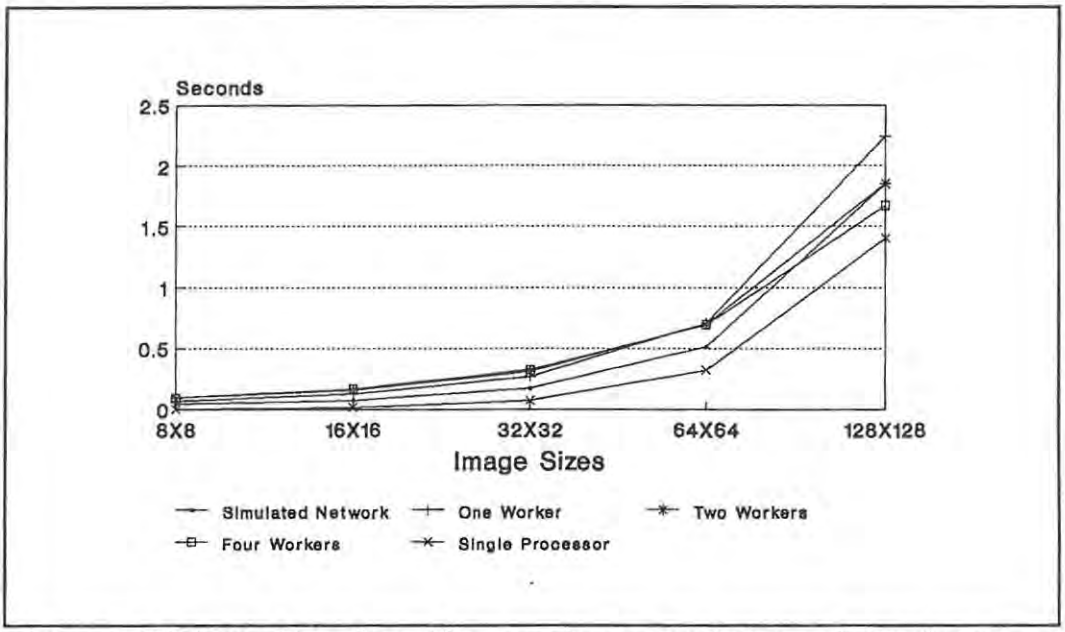

Figure 20 Results for the various numbers of workers.

an image on a *worker* is the number of rows or columns in the image, divided by the number of *workers* in the pipeline. An example of this is a pipeline of four processors and a maximum image of 256x256 pixels, giving a storage of 64x256 elements for the rows and the columns. This still, however, amounts to 64 KBytes of storage when 4 byte real values are used. It is easy to see that declaring separate storage for rows and columns and buffers for performing the FFT soon utilizes the available memory.

A second improvement in this implementation involves the variant protocol. In the above experiment the maximum image size was set at 256x256 pixels. Any image smaller than this was kept inside this storage area. This meant a total of 256x256x4 bytes being transferred on the links when say an 8x8 image was transformed. Transferring large amounts of data simply introduces severe overheads in the time that the links are active, hence the low efficiency figures. This previous method is only advantageous when a general purpose system is required to accommodate any image size. For the purposes of experimentation, though, it is not necessary to keep this generality. The second experiment shows the results for the same process as above, except that the protocols size was set to the actual image size at compile time. As an example of the saving in data, consider an image of 8x8 pixels being sent in the 256x256 storage area. A total of (256x256-64)x4 bytes are saved from being sent along the links.

7.2.2. Optimizing the Protocol

This experiment involved setting the protocol to the limit of the image size i.e. if the image size is 8x8 then the protocol was set for an 8x8 pixel image. Dramatic improvements in the execution time are noted, indicating how wasteful the previous data usage was. The times and percentage efficiency values are shown below where al times shown are in seconds.

One Worker

Size	FFT	Mag
8x8	0.0068	0.0012
16x16	0.0259	0.0044
32x32	0.1047	0.0168
64x64	0.4372	0.0662
128x128	1.8529	0.2648
256x256	Memory limitation	

Two Workers

Size	FFT	Mag	Efficiency (%)
8x8	0.0058	0.0011	40.52
16x16	0.0211	0.0043	41.94
32x32	0.0821	0.0167	44.82
64x64	0.3318	0.0662	48.67
128x128	1.3671	0.2648	51.34
256x256	Memory limitation		

Four Workers

Size	FFT	Mag	Efficiency (%)
8x8	0.0053	0.0011	22.17
16x16	0.0188	0.0043	23.54
32x32	0.0712	0.0167	25.84
64x64	0.2805	0.0662	28.79
128x128	1.1293	0.2647	31.08
256x256	4.6086	1.0639	30.98

Eight Workers

Since the protocols are optimized and more efficient results are obtainable, it is worth looking at larger networks. Here eight transputers are used for the *worker* processors. The minimum image size must be 16x16 pixels as each *worker* must have two rows or columns to work on, because of the real and imaginary arrays used in the Compute-FFT routine. the results for this experiment are shown below.

Size	FFT	Mag	Efficiency (%)
16x16	0.0177	0.0043	12.50
32x32	0.0655	0.0167	14.05
64x64	0.2534	0.0662	15.93
128x128	1.0047	0.2648	17.47
256x256	4.1789	1.1866	17.08

In considering the above results and comparing them to the previous results, it is quite evident just how much time is wasted in sending a complete 256x256 pixel image around the network when say an 8x8 pixel image needs transforming. Taking, for example, the results for a 64x64 pixel image executed on four *worker* processors, and comparing them to those for the same image, but on eight *workers*, a speed up of approximately 40 percent in the execution time is achieved which is impressive, albeit that the overall execution times for both methods are small. The percentage speed up for the smaller images is much higher, indicating the necessity to use this optimized protocol. The results for transforming a 256x256 pixel image will still however be the same. Chen and Yeh [CHE89] discuss the issue of being able to obtain a minimum execution time when there is a balance between the computation time and the communication time as these are the two components involved in the execution time of a parallel algorithm. Although they mention reducing the number of processors if communication time and the computation time are not balanced, the presence of the protocol influencing the communication overhead indicates that it should be optimized, as has been done here. The percentage values for the efficiencies indicate that two processors are the optimum for the data sizes used. This is in keeping with what Chen and Yeh [CHE89] mention. The load balancing for four and eight *workers* is certainly not optimum, albeit that the execution times are faster than when the full protocol is used. Also the degree of parallelism achieved in this experiment is not very high because of the method of distributing the data. This will be discussed in the next experiment, where this problem is overcome.

Figure 21, shows the performance improvement that is obtained by optimizing the protocols. These results can be compared with the previous figure, Figure 20.

The two experiments above are fairly wasteful in terms of the complexity of the protocol structure, although they show results which are impressive when compared to the time for the same operation on the VAX. All the messages indicating the arrival of the transformed rows and columns and the subsequent transferral of them to the neighbouring *transmitter* process, merely introduces further overheads. The method of splitting the image amongst processors also requires a fairly complex amount of code to take care of the ordering of the data and transmission to neighbouring *worker* processors. The code for this appears in Appendix E.

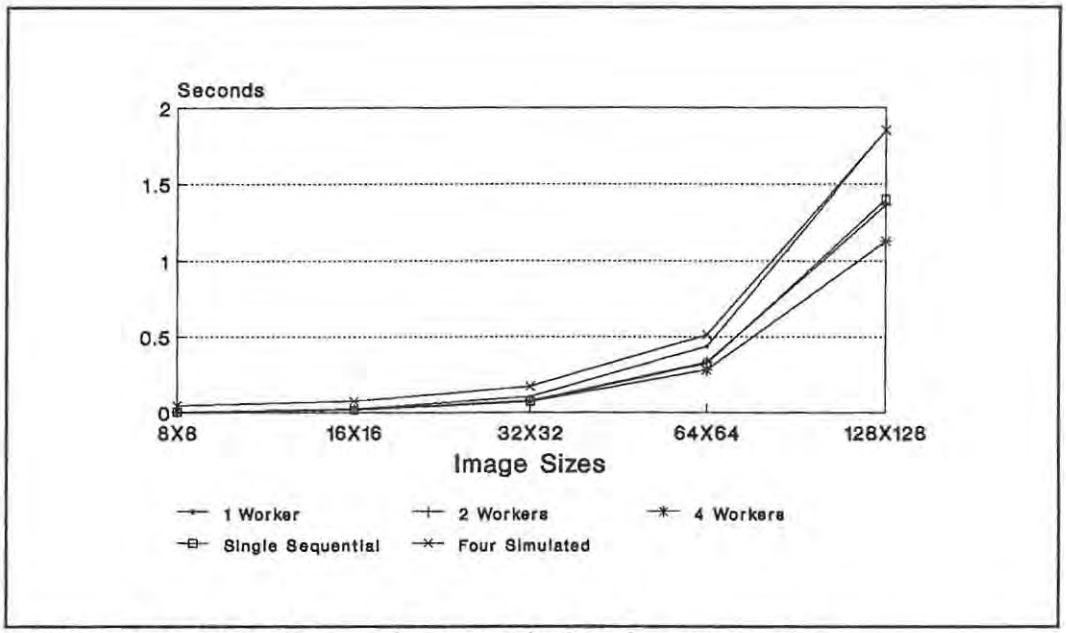

Figure 21 Results using optimized protocols.

7.2.3. Using the Links and Processor Concurrently

The section covering the network used in this project showed the effectiveness of using the links and processor concurrently [JON89]. This next stage of improving the FFT adapts this approach. The second method discussed in Chapter 5 is used here. The introduction of the concurrent use of links and processor also reduces the complexity of the protocol. In this experiment the data is merely sent through the pipeline of processors as a continuous stream. Each *worker* takes two rows or columns, depending on the stage of the FFT, and while processing them passes the data that must go to the neighbours. This method is easier to code as the data exits the pipeline in the exact order in which it entered, thereby returning a correctly ordered image to the *receiver* process. The code for the *worker* and *controller* processors, in Appendix F, indicates the ease of programming. The much reduced protocol for the transmission of data across the network is shown below.

Reduced protocol

```
VAL im.rows IS 256 :
VAL im.cols IS 256 :
VAL im.size IS im.rows*im.cols :
PROTOCOL chan.protocol
  CASE
    N.b.dimen    ; INT    -- Dimensions of the image
    M.b.dimen    ; INT    --
    rec.a.row    ; [im.cols] REAL32 --   32 bit data matrices
    rec.a.col    ; [im.rows] REAL32 --   32 bit data matrices
    comp.image   ; [im.rows][im.cols] REAL32 -- Saves time
    mag.image    ; [im.rows+1][im.cols+1] REAL32
    host.image   ; [im.rows][im.cols] INT16
    m.image      ; [im.rows+1][im.cols+1] INT16
    watch        ; INT    -- For the time
    finished     ; BYTE   --
```

It is quite evident from this protocol that the mechanism involved in sending the data around the network is easier and now conforms with a true pipeline. Comparing the code for this experiment and that of the previous tests will reveal that a higher degree of parallelism has been implemented. The complexity of the previous protocol and the fact that the links were not being used to their full potential resulted in this. If the method of transferring say the columns is examined, it can be seen that the parallelism is lost because a full circuit of the network has to be made by this data before the next batch of data can be sent for processing. The same occurred in the testing of the data transfer when the network was being investigated.

The results for executing the FFT with the links and the processor in parallel are shown below. Times are not given for the computation of the Magnitudes, as this process is always executed on a single processor and the results have already been shown in section 7.2.2.

All times shown in the tests below are in seconds with efficiency values being calculated as before.

4 Workers simulated on a single transputer.

In this test the protocol was set to the maximum image size of 256x256 in order that the experiment follow exactly that of the original experiment. This allows a direct comparison to be made.

Size	FFT
8x8	0.0552
16x16	0.0850
32x32	0.1741
64x64	0.4796
128x128	1.6435
256x256	6.3742

Four Worker processors with a maximum protocol set to 512x512 pixels.

Size	FFT	Efficiency (%)
8x8	0.2062	0.56
16x16	0.2685	1.65
32x32	0.4120	4.47
64x64	0.7608	10.61
128x128	1.7798	19.72
256x256	4.9625	26.05

Four Worker processors with a maximum protocol set at 256x256 pixels.

Size	FFT	Efficiency (%)
8x8	0.0784	1.50
16x16	0.1142	3.87
32x32	0.2047	8.99
64x64	0.4470	18.06
128x128	1.3023	26.95
256x256	3.9366	36.27

It is quite obvious from the two sets of results shown above, that the large image size which the actual image must reside in is responsible for longer times. When the image being transformed is smaller, the difference in times becomes much greater. The percentage efficiency values clearly indicate this. Although the degree of parallelism introduced here shows only a slight improvement in the efficiency for the larger sized images, there is clearly a need for optimized protocols in order that the data being sent across the links be kept to a minimum.

The tests below all use optimized protocols set to the exact image size at compile time.

Two Worker Processors with optimized protocol

Size	FFT	Efficiency (%)
8x8	0.0051	46.08
16x16	0.0195	45.38
32x32	0.0792	46.46
64x64	0.3277	49.28
128x128	1.4099	49.78
256x256	5.0322	56.74

Four Worker processors with optimized protocol

Size	FFT	Efficiency (%)
8x8	0.0054	21.76
16x16	0.0178	24.86
32x32	0.0664	27.71
64x64	0.2607	30.97
128x128	1.0867	32.30
256x256	3.9366	36.27

Eight Worker processors with optimized protocol

The minimum image size permissible here is a 16x16 pixel image. This is because there must always be a minimum of two rows or columns per processor.

Size	FFT	Efficiency (%)
16x16	0.0210	10.54
32x32	0.0680	13.53
64x64	0.2442	16.53
128x128	0.9632	18.22
256x256	3.4581	20.64

From the results above, it is clear that the degree of parallelism obtained here is much improved over that obtained in the previous tests. This is because the concurrency in the link and processor operation on the T800 transputers has been exploited. The times for the smaller image sizes on the four and eight *worker* processor networks is slower than for the two *worker* processor network. This indicates that the balance between the computation time and communication time is not optimal. The time spent in sending the data across links takes longer than the computation of the FFT on a processor, hence the processor sits idle while the data continues across the links. Fewer processors would take care of the load balancing [CHE89]. As the computation becomes more intensive, with the larger images, so the load balancing becomes more optimal, hence the faster times for the four and eight *worker* processors. The non-linearity that can be seen in Figure 21, for the sequential FFT executed on the T800 and the 80286, gives an indication of how optimal load balancing can be achieved for the larger images.

Figure 22, indicates the performance results for the various sized networks and image sizes. Timings are shown for up to 128x128 pixel images as the 256x256 pixel image times tend to cramp the times for the smaller sized images.

One can get a good appreciation of the increase in performance by plotting the actual number of data points versus the times for the various numbers of processors used. This is shown in Figure 23.

Although the percentage efficiency values indicate that two processors are more efficient than four *worker* processors, it must be remembered that a general purpose system is sought. The figures above indicate a high efficiency for the FFT routine only. When considering the results shown by Morrow and Perrot [MOR88a] for a broad spectrum of image processing operations, and the fact that they were using four processors, although in a slightly different configuration, four processors should be considered for a fully implemented workstation. The non-linearity in increase of execution time with increase in the number of processors proves that the return obtained from eight *workers* is not as cost effective as for four. For more intensive calculations for say bigger images, eight may prove more cost effective.

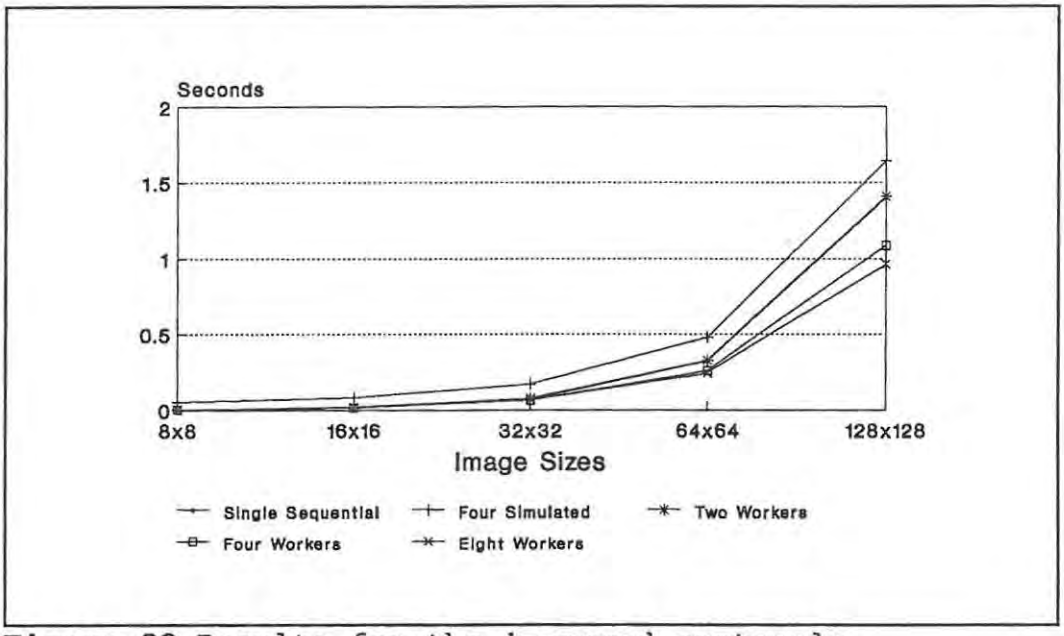

Figure 22 Results for the improved protocols.

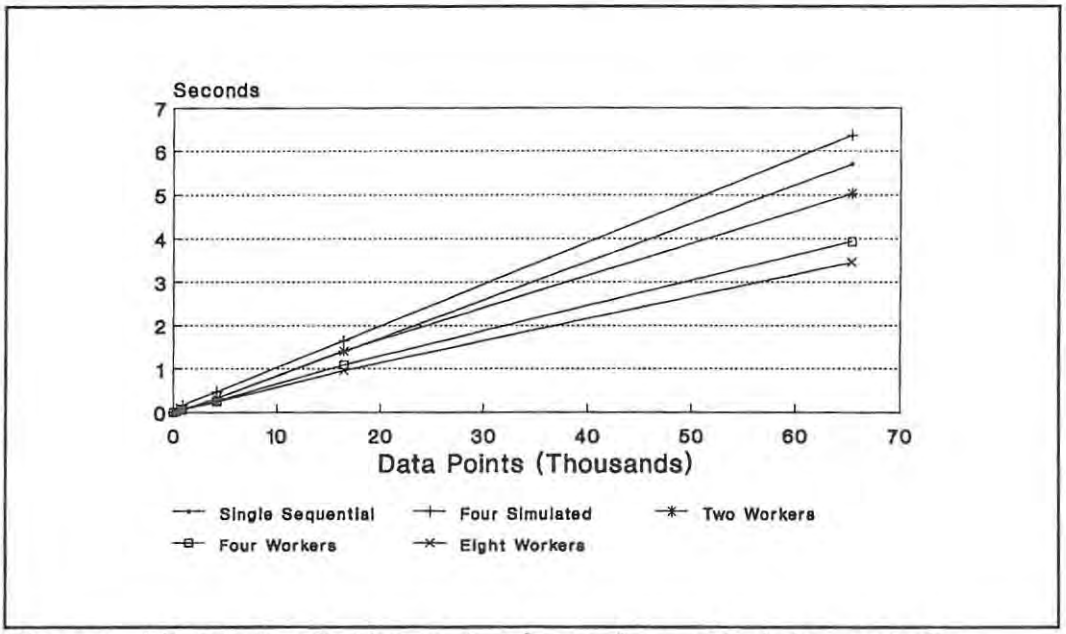

Figure 23 Plot of the actual number of data points.

The graphical representation of the relative increases in performance can also be viewed in such a way that one can see a predicted result for larger images. This can be achieved by simply fitting a curve to the graph, as shown in Figure 24. It should be noted that the scale on the horizontal axis is not linear and distorts the curve somewhat. It nevertheless gives a good indication of the result that can be expected.

Although the execution times for the FFT are very good, when compared to the times achieved

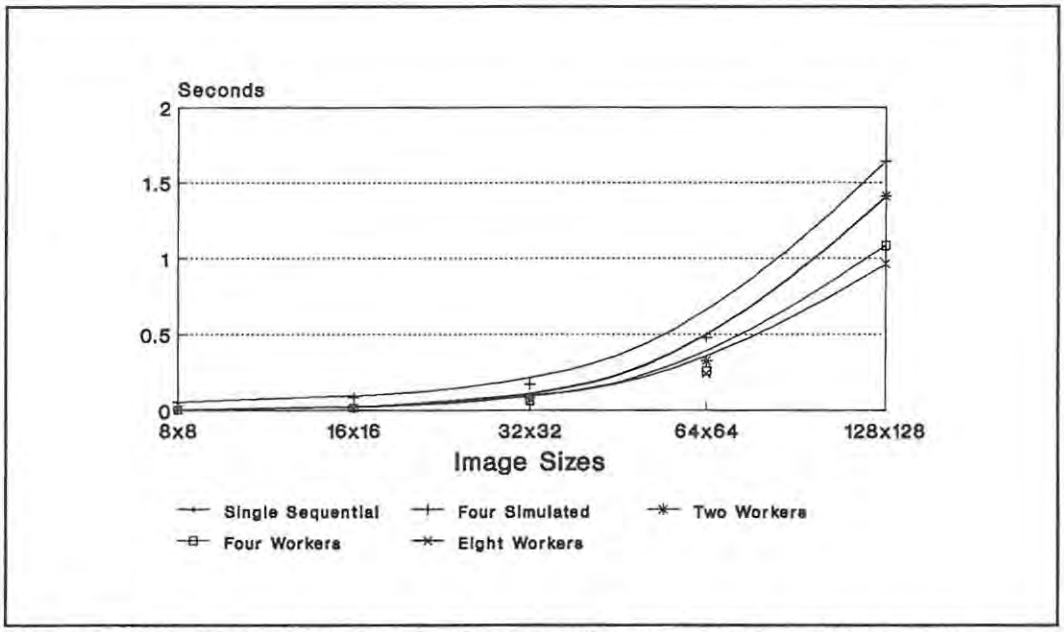

Figure 24 The trend of the performance improvement.

on the VAX and on the AT, care must be taken when the small images are being processed on the network. The time to transfer the image from the host (AT) to the network must be considered unless the image is already on the network. The time to send an image of 256x256 pixels at 8 bits per pixel takes approximately 2.75 seconds. For smaller images this is obviously much quicker. If the times for this combined with the execution and retrieval times is longer than what can be achieved on the host machine then the operation should not be executed here. The limitation of the AT in the image sizes that can be transformed here does, however, favour the use of the transputer network for all the operations to be performed. The image could initially be loaded onto the network and only the computed result retrieved. Furthermore, the T800 transputer becomes more beneficial as an accelerator to the AT for operations which are heavily dependant on real arithmetic.

Chapter Eight

8. Conclusion

An investigation into the feasibility of an image processing workstation comprising a host machine (an INTEL 80286 based AT) and a transputer network has been undertaken. The scope of the project was limited to the feasibility of such a system and no attempt was made to implement a fully fledged workstation. Several areas were investigated in order to provide conclusions as to the viability of such a configuration for image processing, but the investigation concentrated particularly upon the Fast Fourier Transform, an area evaded by previous researchers of parallel image processing techniques.

The two-dimensional FFT was studied as it forms an upper bound in computational intensity in image processing. The results of this investigation allow good conclusions to be drawn as to the size of network which offers the best cost/performance ratio. In comparison to the results for the same sequential algorithm executed on the VAX 11-730 (an architecture commonly used for image processing), the transputer shows very impressive results. From the results obtained for the parallel FFT, keeping the cost/performance ratio in mind, it is evident that the transputer becomes more viable when more intense mathematical calculations are executed on it. The results indicate that for small scale image processing (images limited to 256x256 pixels), four *worker* processors should be considered. This allows a broad spectrum of operations to be executed on the general network, and facilitates easy data distribution using a *quadrant* pattern for splitting the image. In considering the overall operation of the workstation, the FFT measurements have indicated that the time taken to transfer the data from the host to the network will be a substantial portion of the total elapsed time. The data transfer is slow as a result of a bottleneck in the bus of the host. In order to avoid the use of the host computer's bus, a system incorporating transputer-based image capture and display hardware, such as the system described by Harp et al. [HAR87], could be considered as further enhancements to the transputer hardware used for this investigation.

Hardware constraints have limited the scope of exploration in terms of the data size used in the experiments. With only 256 KBytes of RAM per *worker* processor in the test environment, the image sizes have been limited to a maximum of 256x256 pixels. With more memory, larger data sizes (in the region of 1024x1024 pixel images) could be considered and the full potential of the network investigated. Nevertheless, the hardware provided sufficient scope to reveal a clear trend which increases the efficiency of use of a particular size of network as the data set is increased in size. This effect can be attributed largely to the message passing overhead which a particular size of network posseses in filling and flushing the pipeline, irrespective of the data size. For a

particular size of data set, there is clearly a trade-off between transmission time and processing time, from which one is able to ascertain whether to buy more processors, or fewer processors with more memory.

The research has ultimately provided a good distributed FFT algorithm based on *image parallelism* [MOR88a]. No comparable algorithm has yet appeared in the literature, and this algorithm is expected to provide fertile ground for future research.

The network topology chosen for the investigation was a pipeline of processors. The main factor influencing this choice of topology was the generality it offered in terms of the spectrum of image processing operations that could be executed in the *image parallelism* mould. The investigation into alternate methods of data distribution on this network has indicated the benefits of greater simplicity without the loss of efficiency which this topology offers. Not only did the pipelined distribution offer the quickest method of data distribution, but the algorithm, and consequently the code, was very much simpler than when east-west neighbours were used for distribution. Although the pipeline has proven to be very effective because of the generality it offers, the topology still facilitates matrix interconnections for any operations (such as filtering and convolution) where neighbouring components of the image are required for the local calculation. The topology used in this project offers a large degree of flexibility for a wide range of image processing operations.

Some investigation was also performed in the area of graphical display devices. the Enhanced Graphics Adapter (EGA) and the Video Graphics Adapter (VGA) hardware were examined. Both systems accommodate the minimum acceptable resolution required for small scale image processing. The data set is, however, limited to some degree by these hardware devices. Noting these limitations, a suitable avenue for improving the hardware would be the introduction of a system incorporating a video transputer device, such as that described by Harp et. al [HAR87].

During the course of this investigation, several concepts and systems have been noted from the literature which would offer greater flexibility to the scope of this project. These include the introduction of switching networks to allow for greater dynamics in the choice of topology, thereby allowing a close correspondence between the algorithm and the network to be achieved. An indication of the importance attached to such a system is evident from the amount of literature covering this aspect of transputer hardware [DAS88] [FAY87] [KNO88] [NIC88]. This approach also removes the rigid grain of parallelism imposed at run-time [GUP88]. Software development time could be greatly enhanced by introducing high level parallel processing languages such as 3L-C, Pascal and FORTRAN [CUL88] [KUR88], already familiar to most programmers of sequential processing environments. This could be further enhanced with software

systems such as the *pattern oriented parallel debugger* described by Hough and Cuny [HOU88] and simulation software for evaluating parallel programmes [RIZ89]. Also, the availability of parallel languages specific to transputer based image processing, such as the **LATIN** language described by Crookes et. al. [CRO87] [CRO89], would allow for rapid software development on a fully fledged workstation. The need for programming tools in the parallel environment is endorsed by Bershad et. al. [BER88].

This thesis has highlighted the promise which transputer-based image processing holds for providing cost effective and efficient solutions. Already, projects as ambitious as three-dimensional medical imaging systems [TAN88] are being undertaken using this technology.

References

[ATK86] Atkinson, H., (1986), Configuring Multiple CPU's, *Systems International*, March 1986, 48-50.

[BAT86] Batchelor, B.G., et.al., (1986), Developments in Image Processing for Industrial Inspection, *Proceedings of SPIE (The International Society for Optical Engineering), Automated Inspection and Measurement*, Cambridge, Massachusetts.

[BEN89] Ben-Tzvi, D., Sandler, M., (1989), Efficient Parallel Implementation of the Hough Transform on a Distributed Memory System, *Microprocessing and Microprogramming*, **27**, 147-152.

[BER88] Bershad, B.N., et.al., (1988), An Open Environment for Building Parallel Programming Systems, *ACM/Sigplan Notices*, **23**(9), 1-9.

[BOW86] Bowman, C.C., et.al., (1986), Developments in Image Processing for Industrial Inspection, *SPIE (The International Society for Optical Engineering), Automated Inspection and Measurement*, **730**, 34-46.

[BOW87] Bowman, C.C., (1987), Architectures for Image Processing, *SPIE (The International Society for Optical Engineering), Automated Inspection and Measurement*.

[BOW87a] Bowman, C.C., Batchelor, B.G., (1987), Kiwivision - a High Speed Architecture for Machine Vision, *SPIE Proceedings, SPIE Conference on Automated Inspection and High Speed Vision Architectures*, Boston, November 1987.

[BRA78] Bracewell, R.N., (1978), *The Fourier Transform and its Applications*, Second Edition, International Student Edition, McGraw-Hill Kogakusha, LTD.

[BRA87] Bramley, R.G., Creasey, D.J., (1987), A Real-Time Image Compressor using a Modular Signal-Processing System Employing Occam and the Transputer, *Microprocessing and Microprogramming*, North Holland, **21**, 49-55.

[BRI89] Briggs, J.R., Sale, A.H.J., (1989), Fast Fourier Transforms on Multiprocessor Architectures, *The Australian Computer Journal*, **21**(1), 13-18.

[BRU87] Bruschi, O., Negrini, R., Ravaglia, S., (1987), Systolic Arrays for Serial Signal Processing, *Microprocessing and Microprogramming*, North Holland, **20**, 133-140.

[BUR88] Burns, A., (1988), *Programming in occam2*, University of Bradford, Addison Wesley Publishing, Workingham, England.

[CAM88] Cameron, A.G.W., (1988), Transputers and the Sun 386i, *Micro/Systems*, 68-69.

[CHE89] Chen, Y. C., Yeh, Z. C., (1989), Using Fewer Processors to Reduce Time Complexities of Semigroup Computations, *Information Processing Letters*, North Holland, **32**, 89-99.

[CLA88] Clayton, P.G., (1988), *An Overview of Parallel Programming Paradigms for Sharing Information in Distributed Systems*, Department of Computer Science, Rhodes University, Grahamstown, Technical Document No 89/8.

[COK88] Cok, R. S., (1988) A Medium Grained Parallel Computer for Image Processing, *Developments using Occam, Occam Users Group*, Editor. Kerrgie, J., IOS Publishers, Amsterdam, The Netherlands, 113-123.

[COO88] Cooke, N.D., (1988), *Restoring Blurred Digital Text*, Department of Computer Science, Rhodes University, Grahamstown, South Africa, Technical Document No 88/16.

[COO89] Cooke, N.D., Hayes, W.J., Kesterton, A.J., de-Heer-Menlah, F., Clayton, P.G., (1989), *Creating the Link Between the PC and the Transputer*, Department of Computer Science, Rhodes University, Grahamstown, South Africa, Technical Document No 89/12.

[CRO87] Crookes, D., Morrow, P.J., Milligan, P., Scott, N.S., Kilpatrick, P.L., (1987), Notes on Implementing a Language for Transputer Networks, *Microprocessing and Microprogramming*, North Holland, **21**, 559-566.

[CRO89] Crookes, D., Morrow, P. J., Sharif, B., Mc Clatchey, I., (1989), An Environment for Developing Concurrent Software for Transputer Based Image Processing, *Microprocessing and Microprogramming*, North Holland, **27**, 417-422.

[CUL88] Culloch, A.D., (1988), Parallel Programming Toolkit for 3L-C, Fortran and Pascal, *ACM Sigplan Notices*, **23**(9), 23-30.

[DAS88] Das, P.K., Fay, D.Q.N., (1988), Performance Studies of Multi-Transputer Architectures with Static and Dynamic Links, *Microprocessing and Microprogramming*, North Holland, **24**, 281-290.

[DAS88a] Das, P.K., Fay, D.Q.M., (1988), Dynamically Reconfigurable Multi-Transputer Systems, *Microprocessing and Microprogramming*, North Holland, **23**, 247-252.

[DIN89] Dinnig, A., (1989), A Survey of Synchronisation Methods for Parallel Computers, *IEEE Computer*, July 1989, 66-77.

[DOW88] Downing, D. W., Bennet, I. B., (1988), Multi-Transputer based Parallel Implementation of Feature Extraction for Object Recognition, *Developments using Occam, Occam Users Group*, Editor. Kerrgie, J., IOS Publishers, Amsterdam, The Netherlands, 103-111.

[DUF78] Duff, M., (1978), Parallel Processors for Digital Image Processing, *Advances in Digital Image Processing. Theory Applications and Implementation*, Plenum Press, New York, 265-276.

[FAY87] Fay, D.Q.M., Das, P.K., (1987), Hardware Reconfiguration of Transputer Networks for Distributed Object-Oriented Programming, *Microprocessing and Microprogramming*, North Holland, **21**, 623-628.

[FOR87] Fortes, J.A.B., Wah, B.W., (1987), Systolic Arrays - From Concept to Implementation, *IEEE Computer*, July 1987, 12-17.

[GAN88] Gandhi, S., (1988), Graphics Software and hardware Design with the 82786, *Microprocessors and Microsystems*, **12**(5), 271-276.

[GON77] Gonzalez, R.C., Winz, P., (1977), *Digital Image Processing.* Addison-Wesley Publishing Company, Reading, Massachusetts.

[GRO70] Groginsky, H.L., Works, G., (1970), A Pipeline Fast Fourier Transform, *IEEE Transactions on Computing*, **C-19**, 1015-1019.

[GUP88] Gupta, A., Tucker, A., (1988), Exploiting Variable Grain Parallelism at Runtime, *ACM Sigplan Notices*, **23**(9), 212-221.

[HAR87] Harp, J.G., Palmer, K.J., Webber, H.C., (1987), Image Processing on the Reconfigurable Transputer Processor, *7th Occam Users Group and International Workshop on Parallel Programming of Transputer Based Machines*, 1-9.

[HAR87a] Hartenstein, R.W., Hirschbiel, A., Weber, M., (1987), A Flexible Architecture for Image Processing, *Microprocessing and Microprogramming*, North Holland, **21**, 65-72.

[HEL89] *The Helios Operating System*, Perihelion Software Limited, Prentice-Hall, Englewood Cliffs, New Jersey.

[HOA85] Hoare, C.A.R., (1985), *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, Englewood Cliffs, New Jersey.

[HOC88] Hockney, R.W., (1988), *Parallel Computers 2: Architecture, Programming and Algorithms*, 2nd Edition. Consultant Editor: Professor M.H. Rogers, IOP Publishers LTD.

[HOU88] Hough, A.A., Cuny, J., (1988), Initial Experiences with a Pattern-Oriented Parallel Debugger, *Proceedings of the 1988 International Conference on Parallel Processing*, 195-205.

[HWA85] Hwang, K., Briggs, F.A., (1985), *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, United States.

[JES89] Jesshope, C., (1989), Parallel Processing, the Transputer and the Future, *Microprocessors and Microsystems*, **13**(1), 01033-37.

[JON86] Jonas, J., (1986), A 2D DFT Implementation, *Proceedings of the 4th South African Symposium on Digital Image Processing*. Natal University, Durban, South Africa.

[JON87] Jones, G., (1987), *Programming in occam*, C.A.R. Hoare Series Editor. Prentice Hall International Series in Computer Science, London.

[JON89] Jones, G., (1989), Efficient Multiple Buffering in Occam, *Occam Users Group Newsletter*, **11**.

[JON89a] Jones, G., Goldsmith, M., (1989), *Programming in occam2*, Prentice-Hall, Englewood Cliffs, New Jersey.

[JOR89] Jordaan, W., (1989), *An Investigation of the Hardware for the AT, VGA and Frame Grabber (DT2851) with a View to Capture, Process and Display in Real-Time*, Department of Computer Science, Rhodes University, Grahamstown, South Africa.

[KAR88] Karwoski, R.J., (1988), Four-Cycle Butterfly Arithmetic Architecture, *Microprocessors and Microsystems*, **12**(8), 471-480.

[KID83] Kidode, M., (1983), Image Processing Machines in Japan, *IEEE Computer*, January 1983, 68-79.

[KNO88] Knoppers, P., et.al., (1988), A Transputer Network with Flexible Topology, *Microprocessing and Microprogramming*, North Holland, **24**, 275-280.

[KUN87] Kung, S.Y., Lo, S.C., Jean, S.N., Hwang, J.N., (1987), Wavefront Array Processors - Concept to Implementation, *IEEE Computer*, July 1987, 19-33.

[KUR87] Kurver, R., Wijbrans, K., (1987), Developing a Parallel C Compiler, *Micro Cornucopia*, **38**, 14-17.

[LAW88] Lawrie, S.C., (1988), *An Investigation Into the Capabilities of the Enhanced Graphics Adapter*, Honours Project, Department of Computer Science, Rhodes University, Grahamstown, South Africa.

[LUA88] Lua, K.T., (1988), Relative Performance Measurement of 80386, 80286 and 8088 Personal Computer Systems, *Microprocessing and Microprogramming*, **26**, 85-95.

[MAC83] Macovski, A., (1983), *Medical Imaging Systems*. Information and Systems Science Series, Edited by Thomas Kailath. Prentice-Hall, Incorporated, Englewood Cliffs.

[MAN87] McManis, C., (1987), Low-Cost Image Processing, *BYTE*, March 1987, 191-195.

[MOR88] Morrow, P.J., Crookes, D., Kilpatrick, P.L., Milligan, P., Scott, N.S., (1988), A Comparison of Two Notations for Programming Image Processing Applications on Transputers, *Developments using Occam, Occam Users Group*, Editor. Kerrgie, J., IOS Publishers, Amsterdam, The Netherlands, 1-9.

[MOR88a] Morrow, P.J., Perrott, R.H., (1988), The Design and Implementation of Low-Level Image Processing Algorithms on a Transputer Network. *Parallel Architectures and Computer Vision*, Editor. Ian Page, Oxford Science Publications, Clarendon Press, Oxford, 243-259.

[NIC88] Nicole, D.A., Lloyd, K.E., Ward, J.S., (1988), Switching Networks for Transputer Links, *ACM/Sigplan Notices*, **23**(9), 147-165.

[NIC89] Nicoud, J. D., Schweizer, Ph., (1989), Multitransputer Graphics Subsystem, *Microprocessors and Microsystems*, **13**(2), 88-96.

[NIE86] McNiery, E., (1986), New Issues in PC Graphics, *Dr. Dobb's Journal*, November 1986, 30-65.

[OBE88] Obermeier, K.K., (1988), Side by Side, Parallel Processing, *BYTE*, November 1988, 275-283.

[ONE88] O'Neill, M.A., (1988), Faster than Fast Fourier, *BYTE* April 1988, 293-300.

[ORA74] Oran Brigham, E., (1974), *The Fast Fourier Transform*, Prentice-Hall Incorporated, New Jersey.

[PAR88] Parker, K., (1988), Programming the 8086/8088 to Perform, *Micro/Systems*, August 1988, 46-59.

[PEA68] Pease, M. C., (1968), An Adaptation of the Fast Fourier Transform for Parallel Processing, *ACM Journal*, **15**, 252-264.

[PEL78] Peled, A., (1978), A Low-Cost Image Processing Facility Employing a New Hardware Realization of High-Speed Signal Processors, *Advances in Digital Image Processing. Theory Applications and Implementation*, Plenum Press, New York, 301-321.

[POT83] Potter, J.L., (1983), Image Processing on the Massively Parallel Processor, *IEEE Computer*, January 1983, 62-67.

[QUI88] Quinn, M.J., (1988), *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Book Company, New York.

[RIZ89] Rizzo, L., (1989), Simulation and Performance Evaluation of Parallel Software on Multiprocessor Systems, *Microprocessors and Microprogramming*, **13**(1), 39-46.

[ROS76] Rosenfeld, A., Kak, A.C., (1976), *Digital Image Processing*. Computer Science and Applied Mathematics Series, Academic Press, New York.

[ROS76a] Rosenfeld, A., (1976), *Digital Picture Analysis, Topics in Applied Physics*, **11**. Editor. A. Rosenfeld, Springer-Verlag, Printed in Germany.

[ROS83] Rosenfeld, A., (1983), Parallel Image Processing Using Cellular Arrays, *IEEE Computer*, **16**(1), 14-20.

[ROS88] Ross, J., (1988), A Routine to Dump Graphics to the Laser Printer, *Micro/Systems Journal*, July 1988, 35-43.

[SAN88] Sandler, M.B., Eghtesadi, S., (1988), Transputer Based Implementations of the Hough Transform for Computer Vision, *Microprocessing and Microprogramming*, North Holland, **24**, 403-408.

[SAN88a] Sandler, M.B., (1988), Interfacing the Transputer to the TMS320 in an Image Processing Environment, *Microprocessors and Microsystems*, **12**(9), 490-496.

[SCH89] Schomberg, H., (1989), Image Processing on a Transputer-Based Perfect Shuffle Machine, *Microprocessing and Microprogramming*, North Holland, **25**, 277-280.

[SIL89] Silva, V., Dias, J., de Sa, L., (1989), Image Processing System with Multiple DSPs, *Microprocessing and Microprogramming*, North Holland, **25**, 41-46.

[SKI88] Skillicorn, D.B., (1988), A Taxonomy for Computer Architectures, *IEEE Computer*, November 1988, 46-57.

[SLE88] Sleigh, A., Radford, C.J., Harp, G.J., (1988), RSRE Experience Implementing Computer Vision Algorithms on Transputers, DAP and DIPOD Parallel Processors. *Parallel Architectures and Computer Vision*, Editor. Ian Page, Oxford

Science Publications, Clarendon Press, Oxford.

[STE88] Stein, R.M., (1988), T800 and Counting, *BYTE*, November 1988, 287-296.

[TAN88] Tan, A. C., Richards, R., Linney, A. D., (1988), 3D Medical Graphics - Using the T800 Transputer, *Developments using Occam, Occam Users Group*, Editor. Kerrgie, J., IOS Publishers, Amsterdam, The Netherlands, 83-89.

[TRA88] *Transputer Reference Manual*, INMOS Group of Companies. Prentice Hall, New York, 1988.

[WEL89] Welch, P.H., (1989), Graceful Termination - Graceful Resetting, *Applying Transputer Based Parallel Machines, Proceedings of the Tenth Occam Users Group Technical Meeting*, Editor. Andre Bakkers, Amsterdam, 310-317.

[WEX89] Wexler, J., Prior, D., (1989), Solving Problems with Transputers: Background and Experience, *Microprocessors and Microsystems*, **13**(2), 67-78.

[WIL85] Wilton, R., (1985), Programming the Enhanced Graphics Adapter, *BYTE*, Special Issue, **10**(11), 209-220.

[WIL88] Wilton, R., (1988), VGA Video Modes, *BYTE*, Special Issue, Fall 1988, 187-198.

[WIN88] Winder, C.P., (1988), Parallel Processing with the DisPuter, *ACM/Sigplan Notices*, **23**(9), 31-41.

[ZIG89] Zigon, R., (1989), Run Length Encoding, *Dr. Dobb's Journal*, February 1989, 126.

Appendices

Appendix A

Code for LinkIO.PAS to perform the interface between the host and the transputer.

```
(* A unit available for the communications between the Transputer and the AT/XT via the INMOS C011
   link. Various protocols are available, namely for sending and receiving bytes, words, integers
   of various sizes and for resetting the Transputer. Most of the code here is inline code. This
   allows for faster comm's because the code is inserted in the program where calls are made to
   these functions and procedures. This eliminates the setup time for the stack and parameter
   passing. At this stage it appears that REAL values can't be sent across the link because of the
   incompatibility of the REAL types. That is, on the PC a real can only be represented by 6 bytes
   unless a numeric coprocessor is used (under PASCAL) and on the Transputer in occam, the real
   values are REAL32 and REAL64. If I had a numeric coprocessor then real values could be
   transmitted and received.

   Authors : William Hayes and Nicholas Cooke.
             Department of Computer Science.
             Rhodes University.
             Grahamstown.
   Date    : 9 October 1988.
   Updated : 2 November 1988.  *)

(* ------------------------------------------------------------------ *)

unit linkio;

interface

uses crt;

const

  LinkRead      = $150;     (* Addresses for the Transputers comm's *)
  LinkWrite     = $151;
  LinkInStatus  = $152;
  LinkOutStatus = $153;
  LinkReset     = $160;
  LinkError     = $160;
  LinkAnalyse   = $161;
  MaxTimeLink   = 1000;
  Bit0          = $01;

(* ------------------------------------------------------------------ *)

procedure DumpBlock (var block; size : WORD);

(* ------------------------------------------------------------------ *)

function  LinkDataAvailable : BOOLEAN;

(* A function which allows the host program to interrogate the port and
   Link to see whether there is any data available. This function proves
   handy when receiving data because can continue receiving data until
   it decides there is either an error on the line or there is no more
   data available. Thus if it is known how many values should come in
   on the line then a check for the number can be done when this function
   doesn't return a true. *)

inline
  ($BA/LinkInStatus/        {     mov  dx, LinkInStatus      }
   $EC);                    {l1:  in   al, dx                }

(* ------------------------------------------------------------------ *)

function LinkReady : BOOLEAN;

(* A function to see whether the link is ready for data to be sent out
   Care must be taken here when using this function to check whether
   the line is ready to receive. Don't use a REPEAT statement with
   this function as the terminating condition because each time a value
   is sent on the line it automatically becomes ready. Rather indicate
```

```
   exactly how many values must be sent along the line. This can however
   be used as a pre-transmission check, in case the line has hung for
   some reason. *)

inline
  ($BA/LinkOutStatus/       {    mov  dx, LinkOutStatus      }
   $EC);                    {    in   al, dx                 }

(* ------------------------------------------------------------------ *)

procedure ByteToRoot (b : BYTE);

{ A procedure to send a byte to the root Transputer. It makes use of inline
  code for speed purposes only. }

inline
  ($BA/LinkOutStatus/       {    mov  dx, LinkOutstatus      }
   $EC/                     {1:  in   al, dx                 }
   $24/$01/                 {    and  al, 1                  }
   $74/$FB/                 {    jz   1                      }
   $58/                     {    pop  ax                     }
   $BA/LinkWrite/           {    mov  dx, LinkWrite          }
   $EE);                    {    out  dx, al                 }

(* ------------------------------------------------------------------ *)

function ByteFromRoot : BYTE;

(* Receive a byte from the Transputer *)

inline
  ($BA/LinkInStatus/        {    mov  dx, LinkInStatus       }
   $EC/                     {11: in   al, dx                 }
   $24/$01/                 {    and  al, 1                  }
   $74/$FB/                 {    jz   11                     }
   $BA/LinkRead/            {    mov  dx, LinkRead           }
   $EC);                    {    in   al, dx                 }

(* ------------------------------------------------------------------ *)

procedure Int16ToRoot (i : INTEGER);

{ A procedure to send a 16 Bit integer value to the root Transputer. Here
  use is made of the top of stack and the base pointer for obtaining the
  values because they are on the stack on entry to this procedure. }

inline
  ($55/                     {    push bp                     }
   $89/$E5/                 {    mov  bp, sp                 }
   $B9/$02/$00/             {    mov  cx, 2                  }
   $BA/LinkOutStatus/       {11: mov  dx, LinkOutStatus      }
   $EC/                     {12: in   al, dx                 }
   $24/$01/                 {    and  al, 1                  }
   $74/$FB/                 {    jz   12                     }
   $8A/$46/$02/             {    mov  al, [bp+2]             }
   $45/                     {    inc  bp                     }
   $BA/LinkWrite/           {    mov  dx, LinkWrite          }
   $EE/                     {    out  dx, al                 }
   $E2/$EE/                 {    loop 11                     }
   $5D/                     {    pop  bp                     }
   $58);                    {    pop  ax                     }

(* ------------------------------------------------------------------ *)

function Int16FromRoot : integer;

(* Receive a 16 Bit integer from the Transputer *)

inline
  ($B9/$02/$00/             {    mov  cx, 2                  }
   $BA/LinkInStatus/        {11: mov  dx, LinkInStatus       }
   $EC/                     {12: in   al, dx                 }
   $24/$01/                 {    and  al, 1                  }
   $74/$FB/                 {    jz   12                     }
   $BA/LinkRead/            {    mov  dx, LinkRead           }
```

```
   $EC/                    {     in   al, dx              }
   $50/                    {     push ax                  }
   $E2/$F1/                {     loop l1                  }
   $5B/                    {     pop  bx                  }
   $58/                    {     pop  ax                  }
   $88/$DC);               {     mov  ah, bl              }

(* ------------------------------------------------------------------ *)

procedure Int32ToRoot (l : LONGINT);

(* A procedure to send a 32 Bit integer value to the root Transputer. Here
   use is made of the top of stack and the base pointer for obtaining the
   values because they are on the stack on entry to this procedure. *)

inline
  ($55/                    {     push bp                  }
   $89/$E5/                {     mov  bp, sp              }
   $B9/$04/$00/            {     mov  cx, 4               }
   $BA/LinkOutStatus/      {l1:  mov  dx, LinkOutStatus   }
   $EC/                    {l2:  in   al, dx              }
   $24/$01/                {     and  al, 1               }
   $74/$FB/                {     jz   l2                  }
   $8A/$46/$02/            {     mov  al, [bp+2]          }
   $45/                    {     inc  bp                  }
   $BA/LinkWrite/          {     mov  dx, LinkWrite       }
   $EE/                    {     out  dx, al              }
   $E2/$EE/                {     loop l1                  }
   $5D/                    {     pop  bp                  }
   $58/                    {     pop  ax                  }
   $58);                   {     pop  ax                  }

(* ------------------------------------------------------------------ *)

function Int32FromRoot : LONGINT;

(* Receive a 32 Bit integer from the Transputer *)

inline
  ($B9/$04/$00/            {     mov  cx, 4               }
   $BA/LinkInStatus/       {l1:  mov  dx, LinkInStatus    }
   $EC/                    {l2:  in   al, dx              }
   $24/$01/                {     and  al, 1               }
   $74/$FB/                {     jz   l2                  }
   $BA/LinkRead/           {     mov  dx, LinkRead        }
   $EC/                    {     in   al, dx              }
   $50/                    {     push ax                  }
   $E2/$F1/                {     loop l1                  }
   $5B/                    {     pop  bx                  }
   $5A/                    {     pop  dx                  }
   $88/$DE/                {     mov  dh, bl              }
   $5B/                    {     pop  bx                  }
   $58/                    {     pop  ax                  }
   $88/$DC);               {     mov  ah, bl              }


IMPLEMENTATION

(* ------------------------------------------------------------------ *)

procedure DumpBlock (var block; size : word);

(* A procedure to dump a block of code or data to the root Transputer by
   making use of the moving of blocks available only on the AT. Careful
   note should be taken not to allow this to be run on the XT but
   must interrogate for AT or XT so can use different code. *)

begin
 (* Put code in here which will interrogate for an AT and XT *)
end;  (* DumpBlock *)

(* ------------------------------------------------------------------ *)

procedure BootFromT4;
```

```
(* Boot the Root Transputer by first sending the code down as a series
   of bytes and then resetting the Transputer. *)

var
  bootfile : file of BYTE;
  B        : BYTE;
  filename : string;

  procedure ResetRoot;

  (* As the names says, reset the root Transputer. *)

  begin
    port[LinkAnalyse] := 0; delay (50);
    port[LinkReset] := 0;   delay (50);
    port[LinkReset] := 1;   delay (50);
    port[LinkReset] := 0;   delay (50);
  end;  (* ResetRoot *)

begin
  ClrScr;
  write ('Enter the file name : ');
  readln (filename);
  {$I-}
  Assign (bootfile, filename);
  Reset (bootfile);
  {$I+}
  if IOResult <> 0 then
    begin
      writeln ('Cannot find ',filename);
      Halt
    end;
  ClrScr;
  GotoXY (28,10);
  write ('BOOTING ROOT TRANSPUTER');
  ResetRoot;
  while not eof(bootfile) do
    begin
      read(bootfile,b);
      ByteToRoot (b)
    end;
  Close (bootfile);
  ClrScr
end;

begin
  BootFromT4
end. (* LinkIO unit *)

(* ------------------------------------------------------------------ *)
```

Appendix B

The code for configuring and allocating the links for the network used in this project.

```
PLACED PAR

-- 1. Starting at the South of each TxP, going clockwise, start
--    numbering from 0. Thus TxP0 has 0,1,2,3 and TxP1 has
--    4,5,6,7 all corresponding to s,w,n,e.
-- 2. On the root transputer the West link is unused.
-- 3. The computation of the offset into the array of channels
--    makes use of the MOD function (\). This provides a wrap
--    around facility for the transputers on the perimeter.

  -- The link connections follow
  PROCESSOR 0 T4

    -- The configuration for the root transputer. The South
    -- channels talk to the PC.

    VAL s.in  IS 0 :
    VAL w.in  IS 1 : -- Not used
    VAL n.in  IS 2 :
    VAL e.in  IS 3 :
    VAL s.out IS (num.procs+1)*4 : -- The last channel
    VAL n.out IS 4 :
    VAL e.out IS ((num.procs+1)*4)-2 : -- North in of last worker
    PLACE pipe[s.out] AT 0 : -- Data to PC
    PLACE pipe[n.out] AT 2 : -- Data to first worker
    PLACE pipe[e.out] AT 3 : -- Data to last worker
    PLACE pipe[s.in]  AT 4 : -- Data from PC
    PLACE pipe[n.in]  AT 6 : -- Data from first worker
    PLACE pipe[e.in]  AT 7 : -- Data from last worker

    controller (pipe[s.out], pipe[n.out], pipe[e.out],
                pipe[s.in], pipe[n.in], pipe[e.in], 0)

  PROCESSOR 1 T8

    -- The first worker is a special case as this has to deal
    -- with the root transputer. It still, however, follows the
    -- convention of going around South to East in a clockwise
    -- direction.

    VAL s.in  IS 4 :
    VAL w.in  IS 5 :
    VAL n.in  IS 6 :
    VAL e.in  IS 7 :
    VAL s.out IS 2 :
    VAL w.out IS (4*((rows\num.procs)+1))+3 :
    VAL n.out IS 8 :
    VAL e.out IS (4*(((num.procs-rows)\num.procs)+1))+1 :
    PLACE pipe[s.out] AT 0 : -- Data to root
    PLACE pipe[w.out] AT 1 : -- Data to right wing
    PLACE pipe[n.out] AT 2 : -- Data to second worker
    PLACE pipe[e.out] AT 3 : -- Data to right neighbour
    PLACE pipe[s.in]  AT 4 : -- Data from PC
    PLACE pipe[w.in]  AT 5 : -- Data from right wing
    PLACE pipe[n.in]  AT 6 : -- Data from second worker
    PLACE pipe[e.in]  AT 7 : -- Data from neighbour

    workers (pipe[s.out], pipe[w.out], pipe[n.out], pipe[e.out],
             pipe[s.in], pipe[w.in], pipe[n.in], pipe[e.in], 1)

PLACED PAR i = 2 FOR num.procs-2
  PROCESSOR i T8

    -- The second worker to the second last worker all follow
    -- the same pattern of connecting. No special cases are
    -- needed here.

    VAL s.in  IS (i*4)   :
    VAL w.in  IS (i*4)+1 :
    VAL n.in  IS (i*4)+2 :
```

```
    VAL e.in  IS (i*4)+3 :
    VAL s.out IS (i*4)-2 :
    VAL w.out IS (4*((((i-1)+rows)\num.procs)+1))+3 :
    VAL n.out IS (i+1)*4 :
    VAL e.out IS (4*(((num.procs+((i-1)-rows))\num.procs)+1))+1 :

    PLACE pipe[s.out] AT 0 : -- To previous
    PLACE pipe[w.out] AT 1 : -- To right wing
    PLACE pipe[n.out] AT 2 : -- To next
    PLACE pipe[e.out] AT 3 : -- To right neighbour
    PLACE pipe[s.in]  AT 4 : -- From previous
    PLACE pipe[w.in]  AT 5 : -- From right wing
    PLACE pipe[n.in]  AT 6 : -- From next
    PLACE pipe[e.in]  AT 7 : -- From right wing

    workers (pipe[s.out], pipe[w.out], pipe[n.out], pipe[e.out],
             pipe[s.in], pipe[w.in], pipe[n.in], pipe[e.in], i)

PROCESSOR num.procs T8

  -- The last worker is also a special case as it must link
  -- up with the root transputer.

  VAL s.in  IS (num.procs*4)   :
  VAL w.in  IS (num.procs*4)+1 :
  VAL n.in  IS (num.procs*4)+2 :
  VAL e.in  IS (num.procs*4)+3 :
  VAL s.out IS (num.procs*4)-2 :
  VAL w.out IS (4*((((num.procs-1)+rows)\num.procs)+1))+3 :
  VAL n.out IS 3 :
  VAL e.out IS (4*(((num.procs+((num.procs-1)-rows))\num.procs)+1))+1 :

  PLACE pipe[s.out] AT 0 : -- To previous
  PLACE pipe[w.out] AT 1 : -- To right wing
  PLACE pipe[n.out] AT 2 : -- To next = root
  PLACE pipe[e.out] AT 3 : -- To right neighbour
  PLACE pipe[s.in]  AT 4 : -- From previous
  PLACE pipe[w.in]  AT 5 : -- From right wing
  PLACE pipe[n.in]  AT 6 : -- From next
  PLACE pipe[e.in]  AT 7 : -- From right wing

  workers (pipe[s.out], pipe[w.out], pipe[n.out], pipe[e.out],
           pipe[s.in], pipe[w.in], pipe[n.in], pipe[e.in], num.procs)
```

Appendix C

Code for the two-dimensional FFT written in Turbo PASCAL version 4.0.

```
program FastFourier;

(* A program to compute a Fast Fourier Transform on an image of a maximum
   of 64x64 pixels or any dimension whose boundary lies on a 2**N bound and
   whose product equals the product of 64x64. The reason for the size limit
   is explained in the note below.

   Ultimately the time taken to perform a 2-D FFT will be taken and a
   comparison made of this time against that on the Transputer for the
   same dimension image. It is immediately evident that because there
   is no page boundary limit on the Transputers memory, the size of the
   image is not limited to product of 64x64 pixels of real value. The only
   limitation will be the total memory size and the fact that some of the
   memory will be needed for the program. The fact that the transputer
   can be run in a network with each network element having it's own memory
   means that fairly large images can be used.

   A compiler directive switch has been included so that auto detection
   for the presence of a math coprocessor is done. Should this be available,
   the REAL's will be converted to the appropriate size as chosen at compile
   time.

   This program has been adjusted to behave exactly as Justin's does. That is,
   the columns are done first, by mapping them into the work arrays. The rows
   are done second, needing no mapping. If it is to be done the other way then
   the Unscaling and separating must still be done with the columns and not
   the rows, as it was done before.

   Note: When using real arrays for the matrices and running the program
         under the Turbo environment, the maximum size of the matrix on
         a 2**N boundary which will fit into memory is 64x64. A matrix of
         128x128 is too big. The problem of memory sizes can be overcome
         if pointers are used. All that is needed is a pointer to the
         rows. They can then be dynamically allocated. Here the row length
         will have to be known, or a maximum row length specified. This can
         be somewhat wasteful.

   This program makes use of the images produced by the CCD Camera hooked up
   to the Data Translation board that Winstone uses. This image must have it's
   first 8 bytes read because this is where the dimensions of the image are
   kept. The images must be read as bytes and then converted to reals by
   multiplying by 1.0. The FFT only works on reals.

   Author  : Nicholas Cooke.
             Rhodes University.
             Grahamstown.
   Date    : 2 March 1989.
   UpDated : 25 August 1989.        *)

(* ------------------------------------------------------------------ *)

uses Crt;

(* ------------------------------------------------------------------ *)

(* Perform the auto detection of the math coprocessor *)

{$IFOPT N+}
    type
      REAL = DOUBLE;
{$ELSE}
  type
    SINGLE   = REAL;
    DOUBLE   = REAL;
    EXTENDED = REAL;
    COMP     = REAL;
{$ENDIF}

(* ------------------------------------------------------------------ *)
```

```
const
  FF        = #12; (* Form Feed for the printer *)
  Escape    = #27; (* Escape code so can set the printer *)
  Condensed = #81; (* Set to condensed mode *)
  Normal    = #78; (* Set printer back to normal *)
  MaxSize   = 63;  (* Don't quite get the freedom of any product equal to
                      the product of 64x64. Maximum edge size is 64 so
                      can go up to this limit staying on 2**N boundaries *)
  Min2      = -32; (* Dimensions for the maximum size of the array to *)
  Max2      = 32;  (* contain the magnitudes - origin at center *)
  Max2M1    = 31;  (* Half maximum size minus one *)

type
  Barray  = array [0..MaxSize] of BYTE;
  General = array [0..MaxSize] of REAL;
  Weight  = array [0..Max2M1]  of REAL;  (* Length of weights is half side *)
  ImRow   = array [0..MaxSize] of REAL;
  Picture = array [0..MaxSize] of ImRow; (* MB goes down and NB across *)
  MagRow  = array [Min2..Max2] of REAL;
  MagPic  = array [Min2..Max2] of MagRow;

var
  PaperWidth,           (* Set the columns depending on printer mode *)
  MB,                   (* Y dimension of the image *)
  NB,                   (* X dimension of the image *)
  MMB2,                 (* Minus half the Y dimension *)
  MNB2,                 (* Minus half the X dimension *)
  MB2,                  (* Half the Y dimension *)
  NB2       : INTEGER; (* Half the X dimension *)
  Image     : Picture; (* Image to be transformed *)
  Magnitude : MagPic;  (* Matrix containing the magnitudes *)
  RealW     : Weight;  (* Real and Imaginary arrays which carry *)
  ImW       : Weight;  (* the weight tables used in the FFT *)
  BitRev    : Barray;  (* Array to store the Bit Reverse table *)


(* ------------------------------------------------------------------ *)

procedure BitRevTable (TableLength, WordWidth : INTEGER);

(* Set up a bit reverse table for use in the FFT when scrambling and
   unscrambling the values. Here the WordWidth is important as it sets
   up the correct length of the table giving the correct bit reversal
   pattern. *)

var
  k, k1, k2, j, br : INTEGER;

begin
  for k := 0 to TableLength - 1 do
    begin
      br := 0;
      k1 := k;
        for j := 1 to WordWidth do
          begin
            k2 := k1 div 2;
            br := br*2+k1-k2*2;
            k1 := k2;
          end;
      BitRev[k] := br;
    end;
end; (* BitRevTable *)

(* ------------------------------------------------------------------ *)

procedure SetWeight (N : INTEGER; Sign : REAL);

(* Set up a weight table for the FFT routine. This makes use of the
   BitReversal function because the weights must correspond to a bit
   reversed order. *)

var
  k, br,
  Nu, Nu1,
  N2, N4   : INTEGER;
```

```
  TR, TI,
  Cosine,
  Phase    : REAL;

  function BitReversal (Number, Width : INTEGER) : INTEGER;

  (* This function performs a single bit reversal on a number of a specified
     width. This is used in the procedure to set the weight table. *)

  var
    k1, k2, j, br  : INTEGER;

   begin
     br := 0;
     k1 := Number;
      for j := 1 to Width do
        begin
          k2 := k1 div 2;
          br := br*2+k1-k2*2;
          k1 := k2;
        end;
     BitReversal := br
   end; (* BitReversal *)

begin
  Nu := Round (Ln(N)/Ln(2.0));
  Phase := (2*Pi)/(N);
  N2 := N div 2;
  N4 := N div 4;
  RealW[0] := 1.0;
  ImW[N4] := Sign;
    for k := 1 to N4-1 do
      begin
        Cosine := Cos(Phase*k);
        RealW[k] := Cosine;
        RealW[N2-k] := -Cosine;
        ImW[N4-k] := Cosine*Sign;
        ImW[N4+k] := Cosine*Sign;
      end;
    RealW[N4] := 0.0;
    ImW[0] := 0.0;
    Nu1 := Nu - 1;
    for k := 0 to N2 - 1 do
      begin
        br := BitReversal(k, Nu1);
          if br < k then
            begin
              TR := RealW[k];
              TI := ImW[k];
              RealW[k] := RealW[br];
              ImW[k] := ImW[br];
              RealW[br] := TR;
              ImW[br] := TI;
            end;
      end;
end; (* SetWeight *)

(* ------------------------------------------------------------------ *)

procedure ComputeFFT (var Re, Im : ImRow; N : INTEGER);

(* This is a general procedure which performs the FFT on two arrays. One
   of the arrays is placed in the Real part and the other in the Imaginary
   part for the computation. The results are passed back to where they are
   called from. *)

var
  Nu,
  ISam, JSam, NSam,
  ISkip, KSkip, MSkip, NSkip,
  Comp                          : INTEGER;
  Tr, Ti,
  Cosine, Sine                  : REAL;

begin
```

```
  Nu := Round(Ln(N)/Ln(2));
  NSkip := 1;
  KSkip := N;
  MSkip := KSkip div 2;
  ISam := -1;
  JSam := MSkip-1;
    for NSam := 1 to MSkip do
      begin
        ISam := ISam+1;
        JSam := JSam+1;
        Tr := Re[JSam];
        Ti := Im[JSam];
        Re[JSam] := Re[ISam]-Tr;
        Im[Jsam] := Im[ISam]-Ti;
        Re[ISam] := Re[ISam]+Tr;
        Im[ISam] := Im[ISam]+Ti;
      end;
    ISam := ISam+MSkip;
    JSam := JSam+MSkip;
    KSkip := MSkip;
    NSkip := NSkip*2;
      for Comp := 2 to Nu do
        begin
          MSkip := KSkip div 2;
          ISam := -1;
          JSam := MSkip-1;
            for ISkip := 0 to NSkip-1 do
              begin
                Cosine := RealW[ISkip];
                Sine := ImW[ISkip];
                  for NSam := 1 to MSkip do
                    begin
                      ISam := ISam+1;
                      JSam := JSam+1;
                      Tr := Re[JSam]*Cosine-Im[JSam]*Sine;
                      Ti := Re[JSam]*Sine+Im[JSam]*Cosine;
                      Re[JSam] := Re[ISam]-Tr;
                      Im[JSam] := Im[ISam]-Ti;
                      Re[ISam] := Re[ISam]+Tr;
                      Im[ISam] := Im[ISam]+Ti;
                    end;
                  ISam := ISam+MSkip;
                  JSam := JSam+MSkip;
              end;
            KSkip := MSkip;
            NSkip := NSkip*2;
        end;
end; (* ComputeFFT *)

(* ------------------------------------------------------------------ *)

procedure UnscaleAndSeparate (var A1, B1, A2, B2 : ImRow; M, N : INTEGER);

(* B1 and B2 will contain the result and can thus be passed back into the
   original matrix. *)

var
  i, k, l    : INTEGER;
  FMN, FMN2  : REAL;      (* Normalizing factor and half that *)

begin
  FMN := M*N;
  FMN2 := FMN * 2.0;
  (* Sort out the DC components *)
  B1[0] := A1[0]/FMN;
  B2[0] := -A2[0]/FMN;
  i := 2;
    while i < M do
      begin
        k := BitRev[i div 2];
        l := BitRev[M-(i div 2)];
        (* Check why always get a range check error here *)
        B1[i]   :=  (A1[k]+A1[l])/FMN2;  (* x2 to go in steps of 2 *)
        B1[i+1] :=  (A2[k]-A2[l])/FMN2;
        B2[i]   := -(A2[k]+A2[l])/FMN2;
```

```
        B2[i+1] := -(A1[1]-A1[k])/FMN2;
        i := i+2;
      end;
  (* Nyquist components *)
  k := BitRev[M div 2];
  B1[1] := A1[k]/FMN;
  B2[1] := -A2[k]/FMN;
end; (* UnscaleAndSeparate *)

(* ------------------------------------------------------------- *)

procedure RowTransform (N, M1, M2 : INTEGER; var Work1, Work2 : ImRow);

(* M1 and M2 indicate which cols must be done. M is the MB value and N
   the NB value. Here the rows are transformed by taking arrays from the
   matrix, firstly removing the values from the matrix because of the
   matrix structure. *)

var
  i : INTEGER;

begin
  (* Remove the first and second column *)
  Work1 := Image[M1];
  Work2 := Image[M2];
  ComputeFFT (Work1, Work2, N);
  (* Now unscramble the result and put back into the image *)
  for i := 0 to N-1 do
    begin
      Image[M1][i] := Work1[(BitRev[i])];
      Image[M2][i] := Work2[(BitRev[i])];
    end;
end; (* RowTransform *)

(* ------------------------------------------------------------- *)

procedure RealTwoDFwd;

(* This is the procedure which starts the ball rolling. It performs the
   -i two-dimensional DFT on the matrix containing the image. The
   dimensions of the image must be on a 2**N boundary for the sake of
   programming ease. the Hermitian result will be placed in this same
   matrix. Rows 1 and 2 of the matrix will have a different format to
   the rest of the matrix. Important to note that because PASCAL uses
   row major ordering, the transform will be performed on the columns first.
   The columns prove to be somewhat more difficult because of the fact
   that for each column that has to be transformed, the matrix has to be
   traversed, taking out the values and transferring them into an array
   which the transform procedure can then operate on. *)

var
  i,j,k,l,
  WordWidth : INTEGER;
  Work1,
  Work2,
  Work1Res,
  Work2Res  : ImRow;  (* Actually only used once for the special case *)


begin
  (* Work on the rows first *)
  (* Set up weight and bit reversal tables *)
  SetWeight (MB, -1.0);  (* Use NB because doing the rows *)
  WordWidth := Round(Ln(MB)/Ln(2.0));
  BitRevTable (MB, WordWidth);
  j := 0;              (* Go across the columns *)
    while j < NB do   (* FOR loop can't increment by 2 *)
      begin
        for i := 0 to MB-1 do
          begin
            Work1[i] := Image[i][j];     (* Going down i getting columns *)
            Work2[i] := Image[i][j+1];
          end;
        ComputeFFT (Work1, Work2, MB);
        UnscaleAndSeparate (Work1, Work1Res, Work2, Work2Res, MB, NB);
```

```
      (* Don't need to pass Image[j] and [j+1] since Image is a global
         matrix and the procedure UnscaleAndSeparate must put the result
         back in this matrix. Leave it for the mean time *)
      for i := 0 to MB-1 do
        begin
          Image[i][j]   := Work1Res[i];
          Image[i][j+1] := Work2Res[i];
        end;
      j := j+2;
    end;
  (* Now work on the rows *)
  SetWeight (NB, -1.0); (* May not be necessary if MB = NB *)
  WordWidth := Round (Ln(NB)/Ln(2.0));
  BitRevTable (NB, WordWidth);
  RowTransform (NB, 0, 1, Work1, Work2);
  (* Now for the special case - remember working on columns *)
  Image[0][0] := Work1[0];
  Image[1][0] := Work2[0];
  j := 2;
    while j < NB do
      begin
        k := BitRev[(j div 2)];
        l := BitRev[NB-(j div 2)];
        Image[0][j-1] := (Work1[k] + Work1[l])/2.0;
        Image[0][j]   := (Work2[k] - Work2[l])/2.0;
        Image[1][j-1] := (Work2[k] + Work2[l])/2.0;
        Image[1][j]   := (Work1[l] - Work1[k])/2.0;
        j := j + 2;
      end;
    k := BitRev[(NB div 2)];
    Image[0][NB-1] := Work1[k];
    Image[1][NB-1] := Work2[k];
  (* Transform the remaining cols *)
  i := 2;
    while i < MB do
      begin
        RowTransform (NB, i, i+1, Work1, Work2);
        (* Work1 and Work2 won't be used here because the result in
           ColTransform will go into the original matrix *)
        i := i + 2;
      end;
end; (* RealTwoDFwd *)

(* ------------------------------------------------------------------ *)

procedure ComplexToMag (M, N : INTEGER);

(* Convert half of the Hermitian in Image to a matrix of magnitudes
   in Magnitude. The origin is at the center of the matrix and it is
   symmetric through this origin. *)

var
  i, j,
  k, l,
  r, c  : INTEGER;

begin
  (* Do the first and second rows first as they are special cases. The
     DC row, DC column is done. *)
  Magnitude[0][0] := Abs(Image[0][N-1]);
  (* Most of DC row *)
  for j := 1 to (N div 2)-1 do
    begin
      k := N-1-(2*j);
      Magnitude[0][j] := Sqrt(Sqr(Image[0][k])+Sqr(Image[0][k+1]));
      Magnitude[0][-j] := Magnitude[0][j];
    end;
  (* DC row, Nyquist columns *)
  Magnitude[0][-N div 2] := Abs(Image[0][0]);
  Magnitude[0][N div 2] := Magnitude[0][-N div 2];
  (* Nyquist rows, DC column *)
  Magnitude[-M div 2][0] := Abs(Image[1][N-1]);
  Magnitude[M div 2][0] := Magnitude[-M div 2][0];
  (* Most of Nyquist rows *)
  for j := 1 to (N div 2)-1 do
```

```
    begin
      k := N-1-(2*j);
      Magnitude[M div 2][j]   := Sqrt(Sqr(Image[1][k])+Sqr(Image[1][k+1]));
      Magnitude[M div 2][-j]  := Magnitude[M div 2][j];
      Magnitude[-M div 2][j]  := Magnitude[M div 2][j];
      Magnitude[-M div 2][-j] := Magnitude[M div 2][j];
    end;
  (* Nyquist corners *)
  Magnitude[M div 2][N div 2]   := Abs(Image[1][0]);
  Magnitude[M div 2][-N div 2]  := Magnitude[M div 2][N div 2];
  Magnitude[-M div 2][N div 2]  := Magnitude[M div 2][N div 2];
  Magnitude[-M div 2][-N div 2] := Magnitude[M div 2][N div 2];
  (* Do the rest *)
  for j := -N div 2 to (N div 2)-1 do
    begin
      k := j+(N div 2);
        for i := 1 to (M div 2)-1 do
          begin
            l := i*2;
            Magnitude[i][j]   := Sqrt(Sqr(Image[l][k])+Sqr(Image[l+1][k]));
            Magnitude[-i][-j] := Magnitude[i][j];
          end;
    end;
  for i := 1 to M div 2 do
    begin
      Magnitude[i][N div 2]   := Magnitude[-i][N div 2];
      Magnitude[-i][-N div 2] := Magnitude[i][N div 2];
    end;
  writeln;
{  for r := -(MB div 2) to (MB div 2) do
    begin
      for c := -(NB div 2) to (NB div 2) do
        write (Magnitude[r][c] :6:2);
      writeln;
    end;}
end; (* ComplexToMag *)

procedure Scaling;

type
  MagRow  = array [Min2..Max2] OF INTEGER;

var
  i, j  : INTEGER;
  Min,
  Max,
  Max1,
  Half,
  Scale  : REAL;
  Mags : array [Min2..Max2] OF MagRow;

begin
  Min  := 255.0;
  Max  := 0.0;
  Max1 := 0.0;
  for i := -(MB div 2) to (MB div 2) do
    for j := -(NB div 2) to (NB div 2) do
      begin
        if Magnitude[i][j] < Min then
          Min := Magnitude[i][j]
        else
          if Magnitude[i][j] > Max then
            Max := Magnitude[i][j];
        if (Magnitude[i][j] < Max) and (Magnitude[i][j] > Max1) then
          Max1 := Magnitude[i][j];
      end;
  writeln ('Min   : ', Min :6:3);
  writeln;
  writeln ('Max   : ', Max :6:3);
  writeln;
  writeln ('Max1  : ', Max1 :6:3);
  writeln;
  Half := (Max-Max1)/2;
  Scale := 255/Max1;
  writeln ('Half  : ', Half :6:3);
```

```
  writeln;
  writeln ('Scale : ', Scale :6:3);
  writeln;
    for i := -(MB div 2) to (MB div 2) do
      for j := -(NB div 2) to (NB div 2) do
        begin
          if Round(Magnitude[i][j]*Scale) > 255 then
            Mags[i][j] := 255
          else
            Mags[i][j] := Round(Magnitude[i][j]*Scale);
          write (Mags[i][j] :4);
        end;
end; (* Scaling *)

(* ------------------------------------------------------------- *)

procedure GetData;

(* Get the dimensions of the matrix in which the image is to reside from
   the image. These dimensions must be on 2**N boundary and the maximum
   size that the image can be is 64x64 - or any dimension whose product
   equals this product. Also read the data. *)

var
  InFile    : file of BYTE;
  InName    : string[40];
  ImDims    : array [0..7] OF BYTE;  (* Contains the image dimensions *)
  Intensity : BYTE;
  i,
  Row,
  Col       : INTEGER;

begin
  writeln;
  write ('Enter the name of the image : ');
  readln (InName);
  Assign (InFile, InName);
  {$I-}
  Reset (InFile);
  {$I+}
  if IOResult <> 0 then
    begin
      writeln;
      writeln ('Cannot find file : ', InName);
      Halt;
    end;
  writeln;
  (* Read the first 8 bytes so can get the dimensions. Winstone stores
     the images captured from the CCD camera and the Data Translation
     board as to integer numbers for the top left hand corner of the
     image and another two for the bottom right hand corner. INTEL stores
     the low byte before the high byte so the bottom right hand values are
     in positions 4 and 6 (when read as bytes) in a 0 to 7 byte scale. *)
  for i := 0 to 7 do
    read (InFile, ImDims[i]);
  NB := ImDims[4]+1;
  MB := ImDims[6]+1;
  writeln ('NB : ', NB);
  writeln;
  writeln ('MB : ', MB);
  writeln;
  writeln ('Reading ');
  writeln;
  for Row := 0 to MB-1 do
    begin
      for Col := 0 to NB-1 do  (* Column first *)
        begin
          read (InFile, Intensity);
          Image[Row][Col] := 1.0*(Intensity);
        end;
      write ('.');
    end;
  Close (InFile);
  writeln;
  writeln;
```

```
{  for Row := 0 to MB-1 do
    begin
      for Col := 0 to NB-1 do
        write (Image[Row][Col] :6:2);
      writeln;
    end;}
  (* Finished reading now work out the half and quarter dimension values *)
  MMB2 := -MB div 2;
  MNB2 := -NB div 2;
  MB2  := MB div 2;
  NB2  := NB div 2;
end; (* GetData *)

(* ------------------------------------------------------------------ *)

begin
  ClrScr;
  GetData;
  TextColor(White+Blink);
  writeln ('Working...');
  TextColor(White);
  RealTwoDFwd;
  ComplexToMag (MB, NB);
  Scaling;
end. (* FastFourier *)
```

Appendix D

Code for the full two-dimensional FFT written in occam2 for a single T800 transputer to run as a sequential process, as adapted from Jonas [JON86].

```
This program performs a 2D FFT following exactly as what Justin Jonas [JON86] does but
taking into account that FORTRAN has column major ordering. This is an update
of the earlier FFT programs as it does not do rows where he does columns and
vice versa, rather it makes use of mapping of columns on vectors so that a
column transform can be done where he does a column transform. He did not
need to map columns as FORTRAN has column major ordering whereas we have row
major ordering. Likewise where he maps his rows I don't have to as I have
row major ordering.

The output of this program is exactly what [JON86] gives.

This program must be used as the basis for the program that must go onto a
network, taking into account the data that is being shuffled on the network.

NB - Now that D700d has been used this program includes function calls which
not only makes things easier but must watch out when upgrade the other prog's
that change the math procedure calls to function calls.

Author : Nicholas Cooke
         Department of Computer Science
         Rhodes University
         Grahamstown
Date   : 20 July 1989.


2 Dimensional Fast Fourier Transform
************************************

This program makes use of a Discrete Fourier Transform to perform a two
dimensional Fast Fourier Transform. It has been adapted from code written
by Justin Jonas of the Department of Physics and Electronics, Rhodes
University, Grahamstown.

The program has been developed under the Transputer Development System,
version D700d, to be run under TDS. The code runs on a single T800 transputer
as a single sequential process, making calls to procedures. There is a large
amount of debugging code present as well as timing. This is in the procedure
which does all the setting up and can be ignored.


The principle behind the operation of the Fourier transform in this case is
that the columns are done first, reshuffled and then the rows. Each time a
single dimensional transform is performed on two rows or columns at a time
and makes calls to the bit reversal and weight tables.

Author : Nicholas Cooke
         Department of Computer Science
         Rhodes University
         Grahamstown
         6140.

Tel    : (0461) 22023 ext 292/295.

Date   : July 1989.

-------------------------------------------------------------------------------

#USE userio
#USE snglmath
#USE mathvals

VAL x IS 256 :
VAL y IS 256 :

[256]INT Bit.Rev :
[128]REAL32 Re.Weight, Im.Weight :
[y][x]REAL32 Image :
[y+1][x+1] REAL32 Mag :
```

```
PROC Continue ()

  INT any :

  SEQ
    newline(screen)
    write.full.string(screen, "Press any key to continue ")
    keyboard ? any
    newline(screen)
:

PROC Bit.Reverse (VAL INT Table.Length, Word.Width)

  -- Set up a bit reversal table to be used to reorder the values after the
  -- FFT has been computed on a single vector.

  INT br, k.1, k.2 :

  SEQ
    SEQ k = 0 FOR Table.Length
      SEQ
        br := 0
        k.1 := k
        SEQ j = 1 FOR Word.Width
          SEQ
            k.2 := k.1/2
            br := ((br*2)+(k.1-(k.2*2)))
            k.1 := k.2
        Bit.Rev[k] := br
:

PROC Set.Weights (VAL INT Number, VAL REAL32 Sign)

  -- Set up a weight table to save time in having to compute the weights every
  -- time they are needed. Also this table is in bit reversed order so that
  -- direct access can be made to it, again saving time.

  PROC B.Reversal (VAL INT Num, Width, INT b.rev)

  -- A small bit reversal procedure to work out the positions for the values
  -- in the weight table when it is constructed.

  -- In the D700D version of TDS this procedure could be substituted with
  -- a genuine function which maybe I should do.

    INT br, k.1, k.2 :

    SEQ
      br := 0
      k.1 := Num
      SEQ j = 1 FOR Width
        SEQ
          k.2 := k.1/2
          br := ((br*2)+(k.1-(k.2*2)))
          k.1 := k.2
      b.rev := br
  :

  INT Nu, Nu.1, N.2, N.4 :
  REAL32 Phase, Cosine, log.num, log.2 :
  VAL Zero IS 0.0(REAL32) :
  VAL One  IS 1.0(REAL32) :
  VAL Two  IS 2.0(REAL32) :

  SEQ
    log.num := ALOG((REAL32 ROUND Number))
    log.2 := ALOG(Two)
    Nu := INT ROUND(log.num/log.2)
    Phase := (Two*PI)/(REAL32 ROUND Number)
    N.2 := Number/2
    N.4 := Number/4
    Re.Weight[0] := One
    Im.Weight[N.4] := Sign
    SEQ k = 1 FOR (N.4-1)
```

```
      SEQ
        Cosine := COS((Phase*(REAL32 ROUND k)))
        Re.Weight[k] := Cosine
        Re.Weight[N.2-k] := -Cosine
        Im.Weight[N.4-k] := (Cosine*Sign)
        Im.Weight[N.4+k] := (Cosine*Sign)
    Re.Weight[N.4] := Zero
    Im.Weight[0] := Zero
    Nu.1 := (Nu-1)
    INT br :
    REAL32 TR, TI :
    SEQ k = 0 FOR N.2
      SEQ
        B.Reversal(k, Nu.1, br)
        IF
          br < k
            SEQ
              TR := Re.Weight[k]
              TI := Im.Weight[k]
              Re.Weight[k] := Re.Weight[br]
              Im.Weight[k] := Im.Weight[br]
              Re.Weight[br] := TR
              Im.Weight[br] := TI
          TRUE
            SKIP
:

PROC Compute.FFT ([]REAL32 Re, Im, VAL INT N)

  -- This procedure forms the guts of the 2D FFT program in that it computes an
  -- FFT on each row and column that is sent to it. It acts exactly as the 1 D
  -- ComputeFFT routine.

  INT I.Sam, J.Sam, N.Sam :
  INT Nu, N.Skip, M.Skip, K.Skip, I.Skip :
  REAL32 Tr, Ti, log.N, log.2 :
  VAL Two IS 2.0(REAL32) :

  SEQ
    log.N := ALOG((REAL32 ROUND N))
    log.2 := ALOG(Two)
    Nu := (INT ROUND (log.N/log.2))
    N.Skip := 1
    K.Skip := N
    M.Skip := (K.Skip/2)
    I.Sam := (-1)
    J.Sam := (M.Skip-1)
    SEQ N.Sam = 1 FOR M.Skip
      SEQ
        I.Sam := (I.Sam+1)
        J.Sam := (J.Sam+1)
        Tr := Re[J.Sam]
        Ti := Im[J.Sam]
        Re[J.Sam] := Re[I.Sam]-Tr
        Im[J.Sam] := Im[I.Sam]-Ti
        Re[I.Sam] := Re[I.Sam]+Tr
        Im[I.Sam] := Im[I.Sam]+Ti
    I.Sam := (I.Sam+M.Skip)
    J.Sam := (J.Sam+M.Skip)
    K.Skip := M.Skip
    N.Skip := (N.Skip*2)
    SEQ Comp = 2 FOR (Nu-1)
      SEQ
        M.Skip := (K.Skip/2)
        I.Sam := (-1)
        J.Sam := (M.Skip-1)
        REAL32 Cosine, Sine :
        SEQ I.Skip = 0 FOR N.Skip
          SEQ
            Cosine := Re.Weight[I.Skip]
            Sine := Im.Weight[I.Skip]
            SEQ N.Sam = 1 FOR M.Skip
              SEQ
                I.Sam := (I.Sam+1)
                J.Sam := (J.Sam+1)
```

```
                Tr := ((Re[J.Sam]*Cosine)-(Im[J.Sam]*Sine))
                Ti := ((Re[J.Sam]*Sine)+(Im[J.Sam]*Cosine))
                Re[J.Sam] := Re[I.Sam]-Tr
                Im[J.Sam] := Im[I.Sam]-Ti
                Re[I.Sam] := Re[I.Sam]+Tr
                Im[I.Sam] := Im[I.Sam]+Ti
              I.Sam := I.Sam+M.Skip
              J.Sam := J.Sam+M.Skip
        K.Skip := M.Skip
        N.Skip := N.Skip*2
:

PROC Un.Sc.And.Sep ([]REAL32 A.1, B.1, A.2, B.2, VAL INT M, N)

  -- The column when they have been FFT'ed are unscaled and separated, making
  -- use of a normalizing factor.

  INT i, k, l :
  REAL32 FMN, FMN.2 :
  VAL Two IS 2.0(REAL32) :

  SEQ
    FMN := (REAL32 ROUND (M*N))
    FMN.2 := FMN*Two
    -- Sort the DC components
    B.1[0] := A.1[0]/FMN
    B.2[0] := -(A.2[0]/FMN)
    i := 2
    WHILE i < M
      SEQ
        k := Bit.Rev[i/2]
        l := Bit.Rev[M-(i/2)]
        B.1[i]   :=  (A.1[k]+A.1[l])/FMN.2
        B.1[i+1] :=  (A.2[k]-A.2[l])/FMN.2
        B.2[i]   := -((A.2[k]+A.2[l])/FMN.2)
        B.2[i+1] := -((A.1[l]-A.1[k])/FMN.2)
        i := i+2
    -- Nyquist components, Real
    k := Bit.Rev[M/2]
    B.1[1] := A.1[k]/FMN
    B.2[1] := -(A.2[k]/FMN)
:

PROC Row.Transform (VAL INT N, M.1, M.2, []REAL32 Work.1, Work.2)

  -- Don't need to map the rows can pass them directly. When put the values
  -- back into the image after doing the row transform on them then must
  -- make use of the bit reversal.

  SEQ
    Work.1 := Image[M.1]
    Work.2 := Image[M.2]
    Compute.FFT(Work.1, Work.2, N)
    -- Must do the bit reversal
    SEQ i = 0 FOR N
      SEQ
        Image[M.1][i] := Work.1[(Bit.Rev[i])]
        Image[M.2][i] := Work.2[(Bit.Rev[i])]
:

PROC Real.Two.D.Fwd (VAL INT M.b, N.b)

  -- This is where it all starts. This procedure is responsible for the overall
  -- control of the 2D FFT. It makes the calls to the row and column'S that are
  -- sent to the ComputeFFT routine which ultimately does the FFT on them.

  PROC Print.Work (VAL []REAL32 Work1, Work2, VAL INT Size, Number)

    -- Just to help see the results.

    SEQ
      write.full.string(screen, "Number ")
      write.int (screen, Number, 1)
      newline(screen)
      SEQ i = 0 FOR Size
```

```
      write.real32(screen, Work1[i], 5, 2)
    newline(screen)
    write.full.string(screen, "Number ")
    write.int(screen, Number+1, 1)
    newline(screen)
    SEQ i = 0 FOR Size
      write.real32(screen, Work2[i], 5, 2)
    newline(screen)
:

INT Word.Width, i, j, k, l :
REAL32 log.num, log.2, Sign :
[256]REAL32 Work.1, Work.2, Work1.Res, Work2.Res :
VAL Minus.One IS -1.0(REAL32) :
VAL Two IS 2.0(REAL32) :

SEQ
  -- Work on the columns first
  -- Appears as though Occam2 has row major ordering so means will have to
  -- map the columns of the matrix onto vectors
  Sign := Minus.One
  -- Set the weight table and bit reversal table up first
  Set.Weights(M.b, Sign)
  log.num := ALOG((REAL32 ROUND M.b))
  log.2 := ALOG(Two)
  Word.Width := (INT ROUND (log.num/log.2))
  Bit.Reverse(M.b, Word.Width)
  j := 0
  WHILE j < N.b
    SEQ
      -- Since are doing the columns will have to map onto Work.1 and Work.2
      SEQ i = 0 FOR M.b
        SEQ
          Work.1[i] := Image[i][j]
          Work.2[i] := Image[i][j+1]
      Compute.FFT(Work.1, Work.2, M.b)
      Un.Sc.And.Sep(Work.1, Work1.Res, Work.2, Work2.Res, M.b, N.b)
      -- Now put Work1.Res and Work2.Res back into main Image
      SEQ i = 0 FOR M.b
        SEQ
          Image[i][j] := Work1.Res[i]
          Image[i][j+1] := Work2.Res[i]
      Print.Work(Work1.Res, Work2.Res, M.b, j)
      Continue()
      j := j+2
  -- Now start on the rows so have to set up a weight and bit reversal table
  -- for this section of the FFT.
  Set.Weights(N.b, Sign) -- Sign still -1.0 and if M.b = N.b then don't need
                         -- to set the weight table again
  log.num := ALOG((REAL32 ROUND N.b))
  log.2 := ALOG(Two)
  Word.Width := (INT ROUND (log.num/log.2))
  Bit.Reverse(N.b, Word.Width)
  VAL Nought IS 0 :
  VAL One IS 1 :
  -- The first and second rows are special cases so do them together
  Row.Transform(N.b, Nought, One, Work.1, Work.2)
  Print.Work(Work.1, Work.2, N.b, 0)
  Continue()
  -- Now for the special case
  Image[0][0] := Work.1[0]
  Image[1][0] := Work.2[0]
  j := 2
  WHILE j < N.b
    SEQ
      k := Bit.Rev[(j/2)]
      l := Bit.Rev[(N.b-(j/2))]
      Image[0][j-1] := ((Work.1[k]+Work.1[l])/Two)
      Image[0][j]   := ((Work.2[k]-Work.2[l])/Two)
      Image[1][j-1] := ((Work.2[k]+Work.2[l])/Two)
      Image[1][j]   := ((Work.1[l]-Work.1[k])/Two)
      j := (j+2)
  k := Bit.Rev[(N.b/2)]
  Image[0][N.b-1] := Work.1[k]
  Image[1][N.b-1] := Work.2[k]
```

```
    -- Transform the remaining rows
    i := 2
    INT i.plus.1 :
    WHILE i < M.b
      SEQ
        i.plus.1 := (i+1)
        Row.Transform(N.b, i, i.plus.1, Work.1, Work.2)
        Print.Work(Work.1, Work.2, N.b, i)
        Continue()
        i := (i+2)
:

PROC Complex.Mag (VAL INT M, N)

  -- Calculate the magnitude of the transformed image so that can display it on
  -- a monitor as the immediate results available from the FFT won't be able
  -- to be displayed. Note it makes use of a FUNCTION to take the absolute value.

  REAL32 FUNCTION Abs (VAL REAL32 Val.To.Abs)

    VAL Zero IS 0.0(REAL32) :
    REAL32 The.Abs.Val :

    VALOF
      SEQ
        IF
          Val.To.Abs < Zero
            The.Abs.Val := -Val.To.Abs
          TRUE
            The.Abs.Val := Val.To.Abs
      RESULT The.Abs.Val
  :

  SEQ
    -- DC row and DC column
    -- Take the absolute value of the Image value and put into Mag matrix
    Mag[M/2][N/2] := Abs(Image[0][N-1])
    -- Most of the DC row
    INT k :
    REAL32 sqr :
    SEQ j = 1 FOR ((N/2)-1)
      SEQ
        k := ((N-1)-(2*j))
        sqr := ((Image[0][k]*Image[0][k])+(Image[0][k+1]*Image[0][k+1]))
        Mag[M/2][j+(N/2)] := SQRT(sqr)
        Mag[M/2][(-j)+(N/2)] := Mag[M/2][j+(N/2)]
    -- DC row, Nyquist columns
    Mag[M/2][0] := Abs(Image[0][0])
    Mag[M/2][N] := Mag[M/2][0]
    -- Nyquist rows, DC column
    Mag[0][N/2] := Abs(Image[1][N-1])
    Mag[M][N/2] := Mag[0][N/2]
    -- Most of Nyquist rows
    INT k :
    REAL32 sqr :
    SEQ j = 1 FOR ((N/2)-1)
      SEQ
        k := ((N-1)-(2*j))
        sqr := ((Image[1][k]*Image[1][k])+(Image[1][k+1]*Image[1][k+1]))
        Mag[M][j+(N/2)] := SQRT(sqr)
        Mag[M][(-j)+(N/2)] := Mag[M][j+(N/2)]
        Mag[0][j+(N/2)] := Mag[M][j+(N/2)]
        Mag[0][(-j)+(N/2)] := Mag[M][j+(N/2)]
    --Nyquist corners
    Mag[M][N] := Abs(Image[1][0])
    Mag[M][0] := Mag[M][N]
    Mag[0][N] := Mag[M][N]
    Mag[0][0] := Mag[M][N]
    -- Do the rest
    INT k :
    SEQ j = -(N/2) FOR N
      SEQ
        k := j+(N/2)
        INT l :
        REAL32 sqr :
```

```
        SEQ i = 1 FOR (M/2)-1
          SEQ
            l := (i*2)
            sqr := ((Image[l][k]*Image[l][k])+(Image[l+1][k]*Image[l+1][k]))
            Mag[i+(M/2)][j+(N/2)] := SQRT(sqr)
            Mag[(-i)+(M/2)][(-j)+(N/2)] := Mag[i+(M/2)][j+(N/2)]
    SEQ i = 1 FOR M/2
      SEQ
        Mag[i+(M/2)][N] := Mag[(-i)+(M/2)][N]
        Mag[(-i)+(M/2)][0] := Mag[i+(M/2)][N]
:

TIMER Clock :
INT Char :
INT NB, MB, NB2, MB2, MNB2, MMB2 :
INT Width, Char, Time.One, Time.Two, Time.Three :
REAL32 Sign, log.N, log.2, FFT.Time, Mag.Time :
VAL Minus.One IS -1.0(REAL32) :
VAL Two IS 2.0(REAL32) :
VAL Sixty.Four IS 64.0(REAL32) :
VAL Million IS 1000000.0(REAL32) :
VAL Conversion IS Sixty.Four/Million :

SEQ
  Char := 0
  write.full.string(screen, "Enter X value : ")
  read.echo.int(keyboard, screen, NB, Char) -- Get the X value from the PC
  newline(screen)
  write.full.string(screen, "Enter Y value : ")
  read.echo.int(keyboard, screen, MB, Char) -- Get the Y value from the PC
  newline(screen)
  NB2 := NB/2
  MB2 := MB/2
  MNB2 := -NB2
  MMB2 := -MB2
  newline(screen)
  write.full.string(screen, "Value of NB2 is  : ")
  write.int(screen, NB2, 1)
  newline(screen)
  write.full.string(screen, "Value of MB2 is  : ")
  write.int(screen, MB2, 1)
  newline(screen)
  write.full.string(screen, "Value of MMB2 is : ")
  write.int(screen, MMB2, 1)
  newline(screen)
  write.full.string(screen, "Value of MNB2 is : ")
  write.int(screen, MNB2, 1)
  newline(screen)
  log.N := ALOG((REAL32 ROUND NB))
  log.2 := ALOG(Two)
  Width := INT ROUND(log.N/log.2)
  write.full.string(screen, "The width of NB is : ")
  write.int(screen, Width, 1)
  newline(screen)
  Continue()
  Bit.Reverse (NB, Width)
  newline(screen)
  newline(screen)
  write.text.line(screen, "The bit reversal table ")
  newline(screen)
  write.text.line(screen, "Number and Bit Reverse ")
  newline(screen)
  SEQ i = 0 FOR NB
    SEQ
      write.int(screen, i, 3)
      write.int(screen, Bit.Rev[i], 3)
      newline(screen)
  newline(screen)
  Continue()
  newline(screen)
  write.text.line(screen, "The input image ")
  newline(screen)
  SEQ i = 0 FOR MB
    SEQ
      SEQ j = 0 FOR NB
```

```
      SEQ
        Image[i][j] := (REAL32 ROUND ((i+1)+(j+1)))  -- Create Image
        write.real32(screen, Image[i][j], 3, 2)
    newline(screen)
newline(screen)
Continue()
Sign := Minus.One
Set.Weights (NB, Sign)
newline(screen)
write.text.line(screen, "Weight Table ")
newline(screen)
write.text.line(screen, "Real and Imaginary ")
newline(screen)
SEQ i = 0 FOR NB/2
  SEQ
    write.int(screen, i, 3)
    write.real32(screen, Re.Weight[i], 3, 2)
    write.real32(screen, Im.Weight[i], 3, 2)
    newline(screen)
newline(screen)
Continue()
newline(screen)
Clock ? Time.One
Real.Two.D.Fwd (MB, NB)
Clock ? Time.Two
Complex.Mag (MB, NB)
Clock ? Time.Three
newline(screen)
write.text.line(screen, "Transformed image ")
newline(screen)
SEQ i = 0 FOR MB
  SEQ
    SEQ j = 0 FOR NB
      write.real32(screen, Image[i][j], 3, 2)  -- Show Transformed Image
    newline(screen)
newline(screen)
Continue()
newline(screen)
write.text.line(screen, "Magnitude of image ")
newline(screen)
SEQ i = 0 FOR (MB+1)
  SEQ
    SEQ j = 0 FOR (NB+1)
      write.real32(screen, Mag[i][j], 3, 2)  -- Show Magnitude
    newline(screen)
Continue()
write.full.string(screen, "Ticks 1 ")
write.int(screen, Time.One, 1)
newline(screen)
write.full.string(screen, "Ticks 2 ")
write.int(screen, Time.Two, 1)
newline(screen)
write.full.string(screen, "Ticks 3 ")
write.int(screen, Time.Three, 1)
newline(screen)
FFT.Time := ((REAL32 ROUND(Time.Two-Time.One))*Conversion)
Mag.Time := ((REAL32 ROUND(Time.Three-Time.Two))*Conversion)
write.full.string(screen, "Time for FFT ")
write.real32(screen, FFT.Time, 5, 5)
write.full.string(screen, " seconds ")
newline(screen)
write.full.string(screen, "Time for Mag ")
write.real32(screen, Mag.Time, 5, 5)
write.full.string(screen, " seconds ")
newline(screen)
Continue()
```

Appendix E

Code for the initial version of the FFT run on a network of transputers which made use of the full protocol. The worker and controller processes are shown. The configuration code has been omitted.

```
#USE "\nic\trans\protlib.tsr"
#USE "\nic\trans\eight.tsr"
#USE snglmath
#USE mathvals

PROC workers (CHAN OF chan.protocol s.out, w.out, n.out, e.out,
              s.in, w.in, n.in, e.in, VAL INT id.num)

  -- This process does all the dirty work of doing the FFT. It makes calls
  -- to the procedures for the bit reversal tables, weight tables, computing
  -- of FFT and so forth.

  [im.size] INT Bit.Rev                    : -- Stores the bit reversal table
  [im.size/2] REAL32 Re.Weight, Im.Weight : -- Imaginary and real weight table
  [im.cols] REAL32 row.buffer       : -- Buffer for the rows
  [im.rows] REAL32 col.buffer       : -- Buffer for the columns

  PROC Bit.Reverse (VAL INT Table.Length, Word.Width)

  :

  PROC Set.Weights (VAL INT Number, VAL REAL32 Sign)

  :

  PROC Compute.FFT ([]REAL32 Re, Im, VAL INT N)

  :

  PROC Un.Sc.And.Sep ([]REAL32 A.1, B.1, A.2, B.2, VAL INT M, N)

  :

  -- Start of worker process

  BYTE dummy        :
  INT16 value       :
  BOOL Go.on.strike :
  INT time.taken    :
  INT M.b, N.b      :
  INT rows.per.proc :
  INT cols.per.proc :
  INT row.pass.on   :
  INT col.pass.on   :
  [im.rows/num.procs][im.cols] REAL32 row.store : -- Can't get away with it
  [im.cols/num.procs][im.rows] REAL32 col.store :

  SEQ
    dummy := (BYTE 0)          -- Initialize in case no one else does
    Go.on.strike := FALSE      -- So don't go on strike immediately
    WHILE (NOT Go.on.strike)
      s.in ? CASE
        -- Must get the dimensions first
        N.b.dimen; N.b
          SEQ
            n.out ! N.b.dimen; N.b
            cols.per.proc := (N.b/num.procs)
            col.pass.on := ((num.procs-id.num)*cols.per.proc)
        M.b.dimen; M.b
          SEQ
            n.out ! M.b.dimen; M.b
            rows.per.proc := (M.b/num.procs)
            row.pass.on := ((num.procs-id.num)*rows.per.proc)
        -- When get columns can start FFT
        rec.cols; dummy
          SEQ
            n.out ! rec.cols; dummy
            -- Pass neighbours on first
```

```
    SEQ i = 0 FOR col.pass.on
      s.in ? CASE
        rec.a.col; col.buffer
          n.out ! rec.a.col; col.buffer
    SEQ i = 0 FOR cols.per.proc
      s.in ? CASE
        rec.a.col; col.store[i]
          SKIP
    -- Now that have local columns transform them

    INT Word.Width, i, j, k, l    :
    REAL32 log.num, log.2, Sign   :
    [im.cols] REAL32 Work.1, Work.2   :  -- For the columns
    VAL Minus.One IS -1.0(REAL32) :
    VAL Two IS 2.0(REAL32)        :

    SEQ
      -- Work on the columns first
      Sign := Minus.One
      -- Set the weight table and bit reversal table up first
      Set.Weights(M.b, Sign)
      log.num := ALOG((REAL32 ROUND M.b))
      log.2 := ALOG(Two)
      Word.Width := (INT ROUND (log.num/log.2))
      Bit.Reverse(M.b, Word.Width)
      j := 0
      WHILE j < cols.per.proc   -- Only have this many columns
        SEQ
          Work.1 := col.store[j]
          Work.2 := col.store[j+1]
          Compute.FFT(Work.1, Work.2, M.b)
          Un.Sc.And.Sep(Work.1, col.store[j], Work.2,
                        col.store[j+1], M.b, N.b)
          j := j+2
-- Once finished transforming wait for return call
ret.cols; dummy
  SEQ
    -- Send your columns out first then the neighbours
    n.out ! ret.cols; dummy
    SEQ i = 0 FOR cols.per.proc
      n.out ! rec.a.col; col.store[i]
    SEQ i = 0 FOR ((id.num-1)*cols.per.proc)
      s.in ? CASE
        rec.a.col; col.buffer
          n.out ! rec.a.col; col.buffer
-- Get the rows so can transform them
rec.rows; dummy
  SEQ
    -- Pass neighbours on first
    n.out ! rec.rows; dummy
    SEQ i = 0 FOR row.pass.on
      s.in ? CASE
        rec.a.row; row.buffer
          n.out ! rec.a.row; row.buffer
    SEQ i = 0 FOR rows.per.proc
      s.in ? CASE
        rec.a.row; row.store[i]
          SKIP

    -- Now that have local rows transform them

    INT Word.Width, i, j, k, l    :
    REAL32 log.num, log.2, Sign   :
    [im.rows] REAL32 Work.1, Work.2   :  -- For the rows
    VAL Minus.One IS -1.0(REAL32) :
    VAL Two IS 2.0(REAL32)        :

    SEQ
      -- Now start on the rows so have to set up a weight and
      -- bit reversal table for this section of the FFT.
      Sign := Minus.One
      Set.Weights(N.b, Sign) -- Sign still -1.0 and if M.b = N.b then don't need
                             -- to set the weight table again
      log.num := ALOG((REAL32 ROUND N.b))
      log.2 := ALOG(Two)
```

```
      Word.Width := (INT ROUND (log.num/log.2))
      Bit.Reverse(N.b, Word.Width)
      -- The first and second rows are special cases
      -- so do them together
      IF
        id.num = num.procs  -- The first two rows reside on last proc'
          SEQ
            Work.1 := row.store[0] -- The first and second rows
            Work.2 := row.store[1]
            Compute.FFT (Work.1, Work.2, N.b)
            -- Now for the special case
            row.store[0][0] := Work.1[0]
            row.store[1][0] := Work.2[0]
            j := 2
            WHILE j < N.b    -- Going across the whole row
              SEQ
                k := Bit.Rev[(j/2)]
                l := Bit.Rev[(N.b-(j/2))]
                row.store[0][j-1] := ((Work.1[k]+Work.1[l])/Two)
                row.store[0][j]   := ((Work.2[k]-Work.2[l])/Two)
                row.store[1][j-1] := ((Work.2[k]+Work.2[l])/Two)
                row.store[1][j]   := ((Work.1[l]-Work.1[k])/Two)
                j := (j+2)
            k := Bit.Rev[(N.b/2)]
            row.store[0][N.b-1] := Work.1[k]
            row.store[1][N.b-1] := Work.2[k]
            -- Transform the remaining rows on that processor
            i := 2
            INT i.plus.1 :
            WHILE i < rows.per.proc
              SEQ
                i.plus.1 := (i+1)
                Work.1 := row.store[i]
                Work.2 := row.store[i.plus.1]
                Compute.FFT (Work.1, Work.2, N.b)
                -- Now put results back in bit reversed order
                SEQ cnt = 0 FOR N.b
                  SEQ
                    row.store[i][cnt] := Work.1[(Bit.Rev[cnt])]
                    row.store[i.plus.1][cnt] := Work.2[(Bit.Rev[cnt])]
                i := (i+2)
        TRUE
          SEQ
            -- Transform the rows on the other processors
            i := 0
            INT i.plus.1 :
            WHILE i < rows.per.proc
              SEQ
                i.plus.1 := (i+1)
                Work.1 := row.store[i]
                Work.2 := row.store[i.plus.1]
                Compute.FFT (Work.1, Work.2, N.b)
                -- Now put results back in bit reversed order
                SEQ cnt = 0 FOR N.b
                  SEQ
                    row.store[i][cnt] := Work.1[(Bit.Rev[cnt])]
                    row.store[i.plus.1][cnt] := Work.2[(Bit.Rev[cnt])]
                i := (i+2)
-- Once finished transforming then wait till told to send rows
ret.rows; dummy
  SEQ
    -- Pass your rows on first then the neighbours
    n.out ! ret.rows; dummy
    SEQ i = 0 FOR rows.per.proc
      n.out ! rec.a.row; row.store[i]
    SEQ i = 0 FOR ((id.num-1)*rows.per.proc)
      s.in ? CASE
        rec.a.row; row.buffer
          n.out ! rec.a.row; row.buffer
-- Get the time taken at the start
start.time; time.taken
  n.out ! start.time; time.taken
-- Get the time taken just before calculate the magnitudes
mag.time; time.taken
  n.out ! mag.time; time.taken
```

```
        -- Get the time taken at the end
        stop.time; time.taken
          n.out ! stop.time; time.taken
        -- Now can go on strike
        strike; dummy
          SEQ
            n.out ! strike; dummy
            Go.on.strike := TRUE
    SKIP  -- Go home now
:

#USE "\nic\trans\protlib.tsr"  -- Where the PROTOCOL and VAL declarations are
#USE "\nic\trans\eight.tsr"    -- Where the size of the network is
#USE snglmath
#USE mathvals

PROC controller (CHAN OF chan.protocol s.out, w.out,
                 n.out, e.out, s.in, w.in, n.in, e.in, VAL INT id.num)

  -- This process has two processes running in parallel. One is responsible
  -- for the sending of the data from the host to the workers and also the
  -- hybrid image from the first column transform back to the workers.
  -- The second  receives the data from the workers and either passes it
  -- onto the first process for distribution to the workers or passes the
  -- results back to the host. These two processes are necessary because
  -- cannot output or input on the same link in parallel. These two processes
  -- are linked by two channels.

  CHAN OF chan.protocol link.1, link.2 :   -- For the two processes

  PROC receiver (CHAN OF chan.protocol to.transmitter, from.transmitter)

    -- This process sends the intermediate results to it's neighbouring process
    -- which in turn passes the data to the workers or sends the results back
    -- to the host.

    PROC Complex.Mag ([][] REAL32 Mag, [][] REAL32 Image, VAL INT M, N)

      -- Calculate the magnitude of the transformed image so that can display it on
      -- a monitor as the immediate results available from the FFT won't be able
      -- to be displayed. Note it makes use of a FUNCTION to take the absolute value.

    :

    -- Start of controller code

    BYTE dummy   :
    INT16 result :
    INT M.b, N.b :
    INT Stop, Start, Mag.time :
    BOOL Go.on.strike  :
    VAL Two IS 2.0(REAL32) :
    [im.rows] REAL32 col.buffer :
    [im.rows][im.cols] REAL32 return.image :
    [im.rows+1][im.cols+1] REAL32 magnitudes :
    [im.rows+1][im.cols+1] INT16 mags :
    [im.rows][im.cols] INT16 junk :

    SEQ
      dummy := (BYTE 0)        -- Initialize in case no one else does
      Go.on.strike := FALSE    -- So don't go on strike immediately
      WHILE (NOT Go.on.strike)
        e.in ? CASE
          N.b.dimen; N.b       -- Trap the overflow that comes from workers
            SKIP
          M.b.dimen; M.b
            SKIP
          rec.cols; dummy
            -- Image columns gone all around network so now can tell workers
            -- to return the columns
            to.transmitter ! ret.cols; dummy
          rec.rows; dummy
            to.transmitter ! ret.rows; dummy
          -- Columns back now so can send rows to workers
          ret.cols; dummy
```

```
          -- Have to map each column back into matrix
          SEQ
            SEQ j = 0 FOR N.b
              e.in ? CASE
                rec.a.col; col.buffer
                  SEQ i = 0 FOR M.b   -- Only need to go this far down
                    return.image[i][j] := col.buffer[i]
            -- Now can send the rows to the workers
            to.transmitter ! rec.rows; dummy
            -- Tell worker to receive im rows and so send complete hybrid
            -- image for second transform stage
            to.transmitter ! comp.image; return.image
        -- gather all the rows back and then can find the magnitudes
        ret.rows; dummy
          SEQ
            SEQ i = 0 FOR M.b
              e.in ? CASE
                rec.a.row; return.image[i]
                  SKIP
            -- Quickly work out the magnitudes before returning the image
            to.transmitter ! fin.mag; dummy
            Complex.Mag (magnitudes, return.image, M.b, N.b)
            -- Now that have finished working out the magnitudes can get the
            -- end time by letting the transmitter know that have finished
            to.transmitter ! fin.stop; dummy
            SEQ i = 0 FOR M.b
              SEQ j = 0 FOR N.b
                junk[i][j] := INT16 ROUND return.image[i][j]
            s.out ! host.image; junk
            SEQ i = 0 FOR M.b+1
              SEQ j = 0 FOR N.b+1
                mags[i][j] := INT16 ROUND magnitudes[i][j]
            -- finished converting so can return to host
            s.out ! m.image; mags
        start.time; Start
          s.out ! start.time; Start
        mag.time; Mag.time
          s.out ! mag.time; Mag.time
        stop.time; Stop
          s.out ! stop.time; Stop
        strike; dummy
          Go.on.strike := TRUE  -- Stop the whole process
    SKIP
:

PROC transmitter (CHAN OF chan.protocol from.receiver, to.receiver)

  -- The transmitter process takes data from the host and sends it to the
  -- workers and then also gets the intermediate results from the receiver
  -- process in order to overcome the operation on the same link in parallel

  BYTE dummy :
  BOOL Go.on.strike :
  INT N.b, M.b, Start, Stop, Mag.T   :
  [im.rows][im.cols] REAL32 local.im :
  [im.rows][im.cols] INT16 im.host   :
  [im.rows][im.cols] REAL32 returned.im :
  [im.rows] REAL32 col.buffer :  -- To hold column when need to send it
  [im.cols] REAL32 row.buffer :  -- To hold row when need to send it

  TIMER Clock :

  SEQ
    dummy := (BYTE 0)          -- Initialize in case no one else does
    Go.on.strike := FALSE      -- So don't go on strike immediately
    -- Initialize the real matrix
    SEQ i = 0 FOR im.rows
      SEQ j = 0 FOR im.cols
        local.im[i][j] := 0.0(REAL32)
    -- Now can send the data and start the whole FFT process
    WHILE (NOT Go.on.strike)  -- Will be told when to go on strike
      s.in ? CASE
        N.b.dimen; N.b
          n.out ! N.b.dimen; N.b
        M.b.dimen; M.b
```

```
            n.out ! M.b.dimen; M.b
          host.image; im.host
            SEQ
              -- Convert the image to reals
              SEQ i = 0 FOR M.b
                SEQ j = 0 FOR N.b
                  local.im[i][j] := REAL32 ROUND im.host[i][j]
              -- Can start timing as soon as the image has been received
              Clock ? Start
              -- Can immediately start sending the columns
              n.out ! rec.cols; dummy
              -- First map the column into a buffer and then send it
              SEQ j = 0 FOR N.b
                SEQ
                  SEQ i = 0 FOR M.b  -- Going down a column
                    col.buffer[i] := local.im[i][j]  -- NB mapping a column
                  n.out ! rec.a.col; col.buffer
              -- Tell workers to return the columns
              from.receiver ? CASE
                ret.cols; dummy
                  n.out ! ret.cols; dummy
              -- Tell workers they are to receive rows
              from.receiver ? CASE
                rec.rows; dummy
                  SEQ
                    n.out ! rec.rows; dummy
                    from.receiver ? CASE
                      comp.image; local.im
                        SKIP
                    SEQ i = 0 FOR M.b
                      n.out ! rec.a.row; local.im[i]
              -- Tell workers to return the rows
              from.receiver ? CASE
                ret.rows; dummy
                  n.out ! ret.rows; dummy
              -- The receiver will let us know when the processes are finished
              -- Firstly the magnitude time i.e. before calculate magnitudes
              from.receiver ? CASE
                fin.mag; dummy
                  Clock ? Mag.T
              -- Now get the stop time and then send them around
              from.receiver ? CASE
                fin.stop; dummy
                  SEQ
                    Clock ? Stop
                    n.out ! start.time; Start
                    n.out ! mag.time; Mag.T
                    n.out ! stop.time; Stop
          -- Job over
          strike; dummy
            SEQ
              n.out ! strike; dummy
              Go.on.strike := TRUE
      SKIP  -- Process now finished
:


PAR
  transmitter (link.1, link.2)
  receiver (link.1, link.2)

-- The transmitter and receiver processes are linked by channels
-- so that both channels won't try to send on the south channel
-- to the host.
:
```

Appendix F

Code for the Worker processes using the improved method of data transfer. The code for the processes involved in the actual computation of the FFT has been removed. Only the process names are shown.

```
#USE "\nic\trans\newprot.tsr"
#USE "\nic\trans\two.tsr"
#USE snglmath
#USE mathvals

PROC workers (CHAN OF chan.protocol s.out, w.out, n.out, e.out,
              s.in, w.in, n.in, e.in, VAL INT id.num)

  -- This process does all the dirty work of doing the FFT. It makes calls
  -- to the procedures for the bit reversal tables, weight tables, computing
  -- of FFT and so forth.

  [256] INT Bit.Rev                : -- Stores the bit reversal table
  [128] REAL32 Re.Weight, Im.Weight : -- Imaginary and real weight table
  [im.cols] REAL32 row.buffer      : -- Buffer for the rows
  [im.rows] REAL32 col.buffer      : -- Buffer for the columns

  PROC Bit.Reverse (VAL INT Table.Length, Word.Width)

  :

  PROC Set.Weights (VAL INT Number, VAL REAL32 Sign)

  :

  PROC Compute.FFT ([]REAL32 Re, Im, VAL INT N)

  :

  PROC Un.Sc.And.Sep ([]REAL32 A.1, B.1, A.2, B.2, VAL INT M, N)

  :

  -- Start of worker process

  INT M.b, N.b      :   -- Dimensions common to rows and columns

  SEQ
    -- Pass dimensions on first
    s.in ? CASE
      N.b.dimen; N.b
        n.out ! N.b.dimen; N.b
    s.in ? CASE
      M.b.dimen; M.b
        n.out ! M.b.dimen; M.b

    -- First set up bit reverse and weight tables
    -- For maximum efficiency only done once at beginning

    INT Word.Width :
    REAL32 log.num, log.2, Sign   :
    VAL Minus.One IS -1.0(REAL32) :
    VAL Two IS 2.0(REAL32)        :

    SEQ
      Sign := Minus.One
      Set.Weights(M.b, Sign)
      log.num := ALOG((REAL32 ROUND M.b))
      log.2 := ALOG(Two)
      Word.Width := (INT ROUND (log.num/log.2))
      Bit.Reverse(M.b, Word.Width)

    SEQ i = 0 FOR (N.b/(num.procs*2))   -- i = the number of partitions

      [2][im.cols] REAL32 col.buffer :  -- Store my columns
      [2][im.cols] REAL32 col.pass.buffer : -- Temporary for passing
      -- This buffer alternates positions 0 and 1 thereby eliminating
      -- the transfer of data in memory - effectively transferring an
```

```
  -- index here

  SEQ
    -- Get my two cols
    SEQ my.col = 0 FOR 2
      s.in ? CASE
        rec.a.col; col.buffer[my.col]
          SKIP

    [im.rows] REAL32 Work.1, Work.2   :
    PAR
      SEQ
        -- Perform FFT on own buffer
        Work.1 := col.buffer[0]
        Work.2 := col.buffer[1] -- Only have two columns at a time
        Compute.FFT(Work.1, Work.2, M.b)
        -- Try passing by value at some stage - will save some time
        Un.Sc.And.Sep(Work.1, col.buffer[0],
                      Work.2, col.buffer[1], M.b, N.b)

      -- Pass other data on
      SEQ
        s.in ? CASE
          rec.a.col; col.pass.buffer[0]
            SKIP
        SEQ pass.on = 0 FOR (2*(num.procs-1))-1
          PAR
            n.out ! rec.a.col; col.pass.buffer[(pass.on\2)]
            s.in ? CASE
              rec.a.col; col.pass.buffer[(pass.on+1)\2]
                SKIP
        n.out ! rec.a.col; col.pass.buffer[1]
    -- Send out my cols
    SEQ my.col = 0 FOR 2
      n.out ! rec.a.col; col.buffer[my.col]

-- Set up weight and bit reversal tables only once

INT Word.Width :
REAL32 log.num, log.2, Sign   :
VAL Minus.One IS -1.0(REAL32) :
VAL Two IS 2.0(REAL32)        :

SEQ
  -- Now start on the rows so have to set up a weight and
  -- bit reversal table for this section of the FFT.
  Sign := Minus.One
  Set.Weights(N.b, Sign) -- Sign still -1.0 and if M.b = N.b then don't need
                         -- to set the weight table again
  log.num := ALOG((REAL32 ROUND N.b))
  log.2 := ALOG(Two)
  Word.Width := (INT ROUND (log.num/log.2))
  Bit.Reverse(N.b, Word.Width)

-- Handle the special case first
-- The first worker will have the first and second rows
BOOL first.rows : -- So can pick out the special case rows
[2][im.cols] REAL32 row.buffer :  -- Use for rows
[2][im.cols] REAL32 row.pass.buffer :
SEQ
  first.rows := TRUE
  SEQ i = 0 FOR (M.b/(num.procs*2))
    IF
      (id.num = 1) AND first.rows  -- Special case for the first processor
        SEQ
          -- Get my two rows
          SEQ my.row = 0 FOR 2
            s.in ? CASE
              rec.a.row; row.buffer[my.row]
                SKIP

          INT j, k, l :
          [im.cols] REAL32 Work.1, Work.2   :
          PAR
            SEQ
```

```
        -- Perform FFT on own buffer which are the special cases
        Work.1 := row.buffer[0] -- The first and second rows
        Work.2 := row.buffer[1]
        Compute.FFT (Work.1, Work.2, N.b)
        -- Now for the special case
        row.buffer[0][0] := Work.1[0]
        row.buffer[1][0] := Work.2[0]
        j := 2
        WHILE j < N.b    -- Going across the whole row
          VAL Two IS 2.0(REAL32) :
          SEQ
            k := Bit.Rev[(j/2)]
            l := Bit.Rev[(N.b-(j/2))]
            row.buffer[0][j-1] := ((Work.1[k]+Work.1[l])/Two)
            row.buffer[0][j]   := ((Work.2[k]-Work.2[l])/Two)
            row.buffer[1][j-1] := ((Work.2[k]+Work.2[l])/Two)
            row.buffer[1][j]   := ((Work.1[l]-Work.1[k])/Two)
            j := (j+2)
        k := Bit.Rev[(N.b/2)]
        row.buffer[0][N.b-1] := Work.1[k]
        row.buffer[1][N.b-1] := Work.2[k]

        -- Pass other data on
        SEQ
          s.in ? CASE
            rec.a.row; row.pass.buffer[0]
              SKIP
          SEQ pass.on = 0 FOR (2*(num.procs-1))-1
            PAR
              n.out ! rec.a.row; row.pass.buffer[(pass.on\2)]
              s.in ? CASE
                rec.a.row; row.pass.buffer[(pass.on+1)\2]
                  SKIP
            n.out ! rec.a.row; row.pass.buffer[1]
      -- Send out my rows
      SEQ my.row = 0 FOR 2
        n.out ! rec.a.row; row.buffer[my.row]
      first.rows := FALSE
  TRUE        -- Other processors including the first but on second phase
    [2][im.cols] REAL32 row.buffer :  -- Use for rows
    [2][im.cols] REAL32 row.pass.buffer :
    SEQ
      -- Get my 2 rows
      SEQ my.row = 0 FOR 2
        s.in ? CASE
          rec.a.row; row.buffer[my.row]
            SKIP

      INT j, k, l :
      [im.cols] REAL32 Work.1, Work.2   :
      PAR
        SEQ
          -- Perform FFT on own buffer
          -- Check if need Work.1 and 2 for temp storage since need bit reversal
          -- Again use pass by value to save memory
          Work.1 := row.buffer[0]
          Work.2 := row.buffer[1]
          Compute.FFT (Work.1, Work.2, N.b)
          -- Now put results back in bit reversed order
          SEQ cnt = 0 FOR N.b
            SEQ
              row.buffer[0][cnt] := Work.1[(Bit.Rev[cnt])]
              row.buffer[1][cnt] := Work.2[(Bit.Rev[cnt])]

          -- Pass other data on
          SEQ
            s.in ? CASE
              rec.a.row; row.pass.buffer[0]
                SKIP
            SEQ pass.on = 0 FOR (2*(num.procs-1))-1
              PAR
                n.out ! rec.a.row; row.pass.buffer[(pass.on\2)]
                s.in ? CASE
                  rec.a.row; row.pass.buffer[(pass.on+1)\2]
                    SKIP
```

```
                n.out ! rec.a.row; row.pass.buffer[1]
            -- Send out my rows
            SEQ my.row = 0 FOR 2
              n.out ! rec.a.row; row.buffer[my.row]

    SKIP
:

Code for Controller process for enhanced method of data transfer

#USE "\nic\trans\newprot.tsr"  -- Where the PROTOCOL and VAL declarations are
#USE "\nic\trans\two.tsr"      -- Where the size of the network is
#USE snglmath
#USE mathvals

PROC controller (CHAN OF chan.protocol s.out, w.out,
                 n.out, e.out, s.in, w.in, n.in, e.in, VAL INT id.num)

  -- This process has two processes running in parallel. One is responsible
  -- for the sending of the data from the host to the workers and also the
  -- hybrid image from the first column transform back to the workers.
  -- The second  receives the data from the workers and either passes it
  -- onto the first process for distribution to the workers or passes the
  -- results back to the host. These two processes are necessary because
  -- cannot output or input on the same link in parallel. These two processes
  -- are linked by two channels.

  CHAN OF chan.protocol link.1, link.2 :   -- For the two processes

  PROC receiver (CHAN OF chan.protocol to.transmitter, from.transmitter)

    -- This process sends the intermediate results to it's neighbouring process
    -- which in turn passes the data to the workers or sends the results back
    -- to the host.

    PROC Complex.Mag ([][] REAL32 Mag, [][] REAL32 Image, VAL INT M, N)

      -- Calculate the magnitude of the transformed image so that can display it on
      -- a monitor as the immediate results available from the FFT won't be able
      -- to be displayed. Note it makes use of a FUNCTION to take the absolute value.

    :

    TIMER clock :
    INT M.b, N.b, val :
    INT start.time, stop.time :
    VAL Two IS 2.0(REAL32) :
    [im.rows][im.cols] REAL32 return.image :
    [im.rows+1][im.cols+1] REAL32 magnitudes :

    SEQ
      -- Do some timing - get informed by the transmitter when can start
      from.transmitter ? CASE
        watch; val
          clock ? start.time
      e.in ? CASE
        N.b.dimen; N.b
          SKIP
      e.in ? CASE
        M.b.dimen; M.b
          SKIP
      -- Get the columns back and send to host
      -- Try put some parallelism in sending the columns back so
      -- rows can be worked on when finally do FFT part
      -- Will have to send the column answers to the transmitter proc
      [im.rows] REAL32 col.buffer :
      SEQ
        SEQ j = 0 FOR N.b
          e.in ? CASE
            rec.a.col; col.buffer
              SEQ i = 0 FOR M.b   -- Only need to go this far down
                return.image[i][j] := col.buffer[i]
      -- Send Hybrid image to transmitter process
      to.transmitter ! comp.image; return.image
      -- Get the resultant image from rows being done
```

```
    SEQ i = 0 FOR M.b
      e.in ? CASE
        rec.a.row; return.image[i]
          SKIP
    clock ? stop.time
    -- Send it back to the host
    --s.out ! comp.image; return.image  - Don't send it back
    s.out ! watch; start.time
    s.out ! watch; stop.time
    -- Do magnitudes later
    SKIP
    ************************************************************************
    Magnitudes not done here as it is pretty mundane stuff all done before
    ************************************************************************
            -- Quickly work out the magnitudes before returning the image
            Complex.Mag (magnitudes, return.image, M.b, N.b)
            s.out ! mag.image; magnitudes       -- Mag image
    SKIP
:

PROC transmitter (CHAN OF chan.protocol from.receiver, to.receiver)

  -- The transmitter process takes data from the host and sends it to the
  -- workers and then also gets the intermediate results from the receiver
  -- process in order to overcome the operation on the same link in parallel

  INT N.b, M.b :
  BOOL Go.on.strike :
  [im.rows][im.cols] REAL32 local.im :
  [im.rows][im.cols] INT16 im.host :
  [im.rows][im.cols] REAL32 returned.im :

  SEQ
    s.in ? CASE
      N.b.dimen; N.b
        SKIP
    s.in ? CASE
      M.b.dimen; M.b
        SKIP
    -- Convert to REAL
    SEQ i = 0 FOR im.rows
      SEQ j = 0 FOR im.cols
        local.im[i][j] := 0.0(REAL32)
    SEQ i = 0 FOR M.b
      SEQ j = 0 FOR N.b
        local.im[i][j] := (REAL32 ROUND ((i+1)+(j+1)))
    -- Can start timing once everything sorted out
    to.receiver ! watch; INT 1
    n.out ! N.b.dimen; N.b
    n.out ! M.b.dimen; M.b
    -- Send the columns
    -- First map the column into a buffer and then send it
    SEQ j = 0 FOR N.b
      [im.rows] REAL32 col.buffer :  -- To hold columns
      SEQ
        SEQ i = 0 FOR M.b  -- Going down a column
          col.buffer[i] := local.im[i][j]  -- NB mapping a column
        n.out ! rec.a.col; col.buffer
    -- Get the hybrid image from the receiver process
    from.receiver ? CASE
      comp.image; local.im
        SKIP
    -- Send the rows to complete second part of transform
    SEQ i = 0 FOR M.b
      n.out ! rec.a.row; local.im[i]
    SKIP
:


PAR
  transmitter (link.1, link.2)
  receiver (link.1, link.2)

-- The transmitter and receiver processes are linked by channels
-- so that both channels won't try to send on the south channel
```

```
  -- to the host.
:
```