University of Cape Town
Department of Computer Science

Parallelisation of Algorithms

by
Alexander Marius Schuilenburg

Thesis submitted in fulfilment of the requirements for the degree of Master of Science in the Department of Computer Science, University of Cape Town, under the supervision of Dr I. M. A. Gledhill and Prof. P. S. Kritzinger.

September, 1990

This thesis was submitted as part of the project "Research Manpower for Computer Science" of the Foundation for Research Development

Acknowledgements

I would like to thank:

The Division of Aeronautical Systems Technology of the CSIR, and my section leader Mr L. Botha at the CSIR who supported my studies with bursaries, provided me with equipment and test examples, and allowed me time to perform my research on my M.Sc. while working for them.

The Division of Building Technology at the CSIR who originally granted me bursaries and who funded the initial part of my research, as well as giving me the opportunity to attend the Advanced OCCAM and Transputer training course at INMOS in Bristol U.K.

The Department of Computer Science, University of Cape Town for the opportunities given to me to continue my post-graduate education.

Mr J. W. Hand and Mr F. S. Higgs who initially introduced me to Transputers and who suggested topics for an M.Sc. thesis.

Dr A. J. Hayzen and Dr D. Sherwell who provided the technique and theory of the numerical simulation studies of airflow which was used as an example parallelisation problem.

Dr I. M. A. Gledhill for her invaluable advice and support as supervisor.

My family and friends, especially Helen Kelly, for their continued support regarding my continual lengthy absences from home, long hours and helping me get the job done.

CONTENTS

1 INTRODUCTION 1

2 PARALLEL ALGORITHMS

2.1 Introduction 3

2.2 Why Parallelise ? 4

2.3 Speedup 5

2.4 Discussion 9

3 THE T800 TRANSPUTER - AN OVERVIEW 10

4 PARALLEL LANGUAGES

4.1 OCCAM 14

4.2 3L Parallel Fortran 15

4.3 Other 3L Parallel Languages 17

4.4 Languages under HELIOS 17

4.5 Summary 18

5 PROCESSES

5.1 Description 20

5.2 Process Networks 22

5.3 Process Placement 23

5.4 Process Farm 25

5.4.1 Introduction 25

5.4.2 Farm Workers 26

5.4.3 Farm Communication 26

5.4.4 Farm Construction - Method 1 31

5.4.5 Farm Construction - Methods 2, 3 and 4 33

5.4.6 Farm Construction - Method 5 35

5.4.7 Farm Construction - Method 6 38

5.5 General Communications Technique 39

6 FARM PARALLELISATION

6.1 Transputer Calculations/Communications Ratio 43
6.2 3L Flood-Fill Farm 47
6.3 SAPF Fortran Farm Structure 49
6.3.1 SAPF Farm Construction 52
6.3.2 Master Process Routines 54
6.3.3 Worker Process Routines 55
6.3.4 Worker Processor Routing Processes 55

7 OCCAM/TURBO PASCAL SERVER 59
7.1 The *Host Server* 60
7.1.1 Reset Mode 61
7.1.2 Booting the Root Transputer 61
7.1.3 Operation 63
7.2 The *Transputer Server* 64
7.3 Summary 65

8 TRANSPUTER WORM
8.1 Link Identification 66
8.2 Determining Memory Size 68
8.3 Transputer Typing 69
8.4 Depth First Worm 70
8.4.1 The Head 70
8.4.2 The Tail 73
8.5 Breadth First Worm 73
8.5.1 The Mouth 75
8.5.2 The Foot 77
8.6 Conclusion 77

9 AIRFLOW MODELLING

9.1 Introduction 79
9.2 Network Configuration 80
9.3 Airflow Network 1 81
9.4 Airflow Network 2 87
9.5 Airflow Network Shell 90
9.6 Data Dependency 92
9.6.1 Case where $p \geq n^2$ 93
9.6.2 Case where $p < n^2$ 95
9.6.3 Alternative Method 96
9.7 Airflow Conclusion 97

10 SAPF - SEMI AUTOMATIC PARALLELISER OF FORTRAN

10.1 Introduction 98
10.2 Parallelisation through SAPF 99
10.2.1 Code Structuring 99
10.2.2 Master/Worker Nomination 100
10.2.3 Static Checking 100
10.2.4 Output 102
10.2.5 Data Relationships 103
10.2.6 Deciding on the (M_i, W_i) pair 106
10.2.7 Multiple Worker Types 107
10.2.8 Implementation 111
10.2.9 MAKECONF 112
10.3 Summary 114

11 PARALLELISATION OF BSCAT

11.1 Introduction 116
11.2 BSCAT Analysis and Worker Determination 117
11.3 Selection of a (M_i, W_i) Pair 118

11.4 Data Communication 120
11.4.1 Work Packet 120
11.4.2 Exception Data 122
11.4.3 BSCAT Results 123
11.5 The Master 124
11.6 The Worker 125
11.7 Results 125
11.8 Conclusion 128

12 CONCLUSION 129

Bibliography 132

Appendix A - SAPF Routine Descriptions 134
1 Master Routines 134
2 Worker Routines 141

Appendix B - OCCAM/Turbo Pascal Server User Manual 145

APPENDIX C - BSCAT 169
BScat Appendix (i) - RELATE output from SAPF output of master and worker 169
BScat Appendix (ii) - RELATE output from pre-loop calculations 172
BScat Appendix (iii) - RELATE output from post-loop calculations . . . 173
BScat Appendix (iv) - List Of Protocol Values 173
BScat Appendix (v) - Description of the Master Process 173
BScat Appendix (vi) - Description of the Worker Process 177

Index 180

Abstract

Most numerical software involves performing an extremely large volume of algebraic computations. This is both costly and time consuming in respect of computer resources and, for large problems, often super-computer power is required in order for results to be obtained in a reasonable amount of time. One method whereby both the cost and time can be reduced is to use the principle "*Many hands make light work*", or rather, allow several computers to operate simultaneously on the code, working towards a common goal, and hopefully obtaining the required results in a fraction of the time and cost normally used. This can be achieved through the modification of the costly, time consuming code, breaking it up into separate individual code segments which may be executed concurrently on different processors. This is termed parallelisation of code. This document describes communication between sequential processes, protocols, message routing and parallelisation of algorithms. In particular, it deals with these aspects with reference to the Transputer as developed by INMOS and includes two parallelisation examples, namely parallelisation of code to study airflow and of code to determine far field patterns of antennas. This document also reports on the practical experiences with programming in parallel.

1 INTRODUCTION

This thesis deals with communication between sequential processes, protocols, message routing and parallelisation of algorithms. In particular, it deals with these aspects with reference to the transputer as developed by INMOS. Several features of the transputer will be discussed and examined, with their usefulness investigated.

As an exercise, two scientific problems will be implemented on a network of Transputers to demonstrate the feasibility of the assorted aspects being investigated. The first problem is that of determining the far field patterns of antennas in the presence of perfectly conducting metal structure at UHF and above. It involves a large amount of numeric computation which requires a large amount of computing power in order for results to be obtained in a reasonable amount of time. In conjunction with this, tools were developed for the semi-automatic parallelisation of Fortran code (SAPF). These tools allow a user to parallelise sequential code which is suited for a processor farm structure, often by "pulling out" loops and dividing the workload amongst several processors.

The second problem is that of studying the airflow around and through obstructions at low and medium speeds. Although the problem itself is not unique, the approach to the solution of the problem is. The air is modelled at a microscopic molecular level, rather than at a macroscopic level. The area being investigated is divided up into cells and the paths of air molecules traced within each cell. The calculations involved in each cell are relatively independent of each other and hence may be performed independently on different Transputers, but more about this later.

This thesis also investigates, in some detail, processor farms and the communication involved. This includes the viability of various farm constructs, the types of farm communication, routing required and different methods of routing. The practical experiences gained when programming in parallel are also reported and are probably representative of the initial experiences of any programmer when introduced into

parallel programming. This includes the parallel pitfalls/problems encountered, probable causes, possible solutions, as well as positive experiences.

Due to the volume and size of the source code often involved, most of it has been omitted from this thesis, as well as from appendices to this thesis. Instead it has been included in a thesis supplement which may be obtained by writing to the author directly. Only information which was determined to be of immediate interest was included in the thesis and appendices. This was done in order to keep the size of the thesis small as, with the source code and other listings included, the thesis would otherwise extend to over 400 pages, of which only the first 120 would be of any descriptive importance.

2 PARALLEL ALGORITHMS

2.1 Introduction

Let us first distinguish between parallelism and concurrency. Concurrency is the *potential* for events to occur simultaneously, while parallelism is the *simultaneous occurrence* of events. Concurrent and parallel algorithms are therefore essentially the same, although a parallel algorithm is executed on more than one processor, while a concurrent algorithm can be executed on one or more processors. In the case of a concurrent algorithm executing on a single processor, the individual processes of the algorithm may be seemingly executed in parallel through time-slicing of the code. The processes will therefore appear to be executing at the same time, although they are in actual executed sequentially bit for bit. A parallel algorithm, however, is executed on at least two processors and therefore consists of processes or parts of processes which are executing simultaneously. It is therefore a refinement of a concurrent algorithm.

Apart from the normal problems encountered when creating normal sequential algorithms, the introduction of parallelism incorporates many further problems which have to be prevented or resolved. Some of them are:

- Avoiding deadlock and livelock
- Preventing communication bottlenecks
- Preventing unwanted race conditions and process lockout
- Avoiding the creation of too many parallel processes
- Preventing processor overloading or underloading
- Detecting program termination

Other issues involved are:

- Program speedup versus number of processors
- Processor efficiency
- Synchronisation overheads
- Communication overheads
- Effect of problem size on speedup, and maximum speedup
- Maximum number of processors which can be kept busy
- Determinism of program execution

2.2 Why Parallelise ?

Much of the scientific code written today involves a large number of complex numeric computations. In order for the results of the execution of this type of code to be obtained in a reasonable amount of time, a large amount of computing power (e.g. CRAY X-MP) is required. Such large systems are not always available to users and can be costly to utilise and run. Furthermore, the size, speed and power of single processors are limited by the laws of physics, and although we are not there yet, we may soon reach a stage that it is not possible or feasible to build faster processors. What happens then ?

The first alternative any good businessman would suggest would be to "hire more processors". If one man takes 1 hour to dig a hole, then why can't two similar men dig the same size hole in ½ the time. This form of problem is dependant on the size and type of the hole. If the hole was to be 5m by 2m wide and 1m deep, it is possible for two or more men to dig

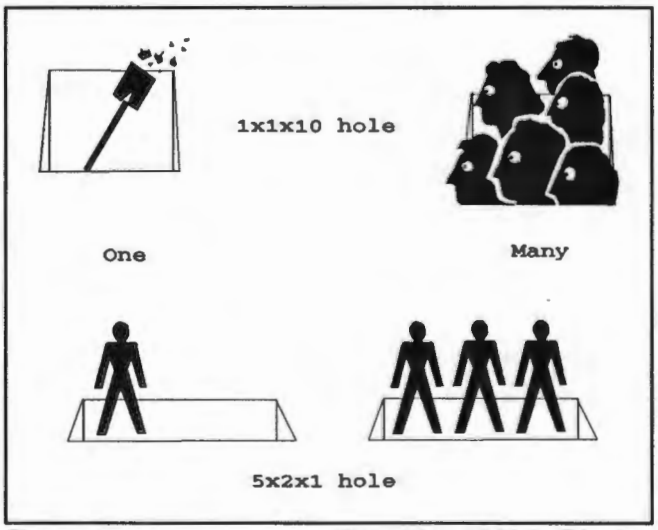

Figure 1 Illustration of Hole Digging

the hole at the same time. If it were to be 1m by 1m wide and 10m deep, it may be difficult to fit more than one man who is able to dig in the hole. In the former case, it is possible to employ more of the same work force to do the task in less time. In the latter case, speedup of the task is only possible by employing a stronger man who can work faster, since if more men were to be placed in the hole, their presence may simply hinder the work progress. This analogy, illustrated in Figure 1, can apply directly to processing. Only certain tasks are suitable for parallelisation. For example, an iterative "root finding" algorithm which requires the results from previous iterative calculations may be totally unsuitable for parallelisation, while an iterative algorithm which draws graphically the mandlebrot set is ideal.

Therefore, in a perfect situation, a program can run n times faster on n processors than on a single processor. This is known as a linear speedup.

2.3 Speedup

Let us now introduce some notation:
Denote the execution time for a task on one processor to be $T(1)$ and on n processors to be $T(n)$.

The speedup is defined by

$$S(n) = \frac{T(1)}{T(n)} \tag{1}$$

and the efficiency by

$$E(n) = \frac{S(n)}{n} \tag{2}$$

Normally, due to communication and synchronisation overheads we have

$$S(n) \leq n \tag{3}$$

$$E(n) \leq 1 \tag{4}$$

Linear speedup is thus when

$$S(n) = n \tag{5}$$

$$E(n) = 1 \tag{6}$$

and superlinear speedup is when

$$S(n) > n \tag{7}$$

$$E(n) > 1 \tag{8}$$

One topic which occasionally re-occurs is the dispute whether superlinear speedup of a parallel implementation of an algorithm over its sequential implementation is possible [Fab86,Par86,Fab87,Jan87]. That is, can n processors do a task more than n times faster than a single processor ? The answer to this is both yes and no, depending on the algorithm and underlying computer architecture which is to implement the algorithm. The mapping of the algorithm to the underlying architecture may result in a greater or lesser efficiency, depending on the suitability and compatibility of the two.

In an attempt to illustrate superlinear speedup Parkinson [Par86] used a vector addition algorithm. In his parallel algorithm, Parkinson assumed that the indices and addresses have already been computed and already reside in the processors. As shown by Faber et al [Fab87], this assumption should be included in the sequential algorithm to a more efficient sequential form which indeed is such that the speedup once again became linear as a maximum. This suggests, that for any algorithm which does achieve superlinear speedup when executed in parallel, it is possible to reduce the sequential algorithm into a more efficient form which is closer to the parallel algorithm and exhibits a speed n times slower than the parallel algorithm running on n processors.

When one considers a sequential program represented in its lowest form (i.e. the implementation level) is to be implemented on a parallel machine without *any* algorithmic modifications to the code, it is not possible to achieve superlinear speedup. This is easily illustrated by taking any algorithm A and breaking it into atomic instruction units $a_1, a_2, a_3, \ldots a_n$ where n is the number of instructions. Assume all of these instructions all take time t to execute (if they do not then we clump together

groups of instructions into equal running time units and set these groups as the instruction units). The total run time of the sequential algorithm is thus tn. If we were to have n processors, each executing one of the above statements simultaneously, the time taken for the parallel execution would, at a minimum, be t. The speedup is $S(n) = tn/t = n$. Since the dividend is at a minimum, the speedup is at a maximum.

Faber et al [Fab86] shows that it is not possible to achieve superlinear speedup in this manner. In this paper, Faber shows that if an algorithm A has a parallel implementation on n processors with running time t, then it has a sequential implementation with running time at most tn. This is proven by simulating the parallel algorithm through a sequential algorithm which executes the first instruction of a_1, then the first instruction of a_2,... then the first instruction of a_n, followed by the second instruction of a_1, the second instruction of a_2 etc. until the problem is solved. This will have a running time no greater than tn.

However, Janβen [Jan87] demonstrated that, using the CREW-PRAM model of Fortune/Wyllie, this is only the case if the cost of memory access is neglected. The algorithm Janβen uses to illustrate his point is the calculation of

$$x^{2^k} \quad ; \quad x \in \Re \quad k \in \mathbb{N} \tag{9}$$

This requires one step to access x, k steps to compute the result and one further step to store the result. Hence the sequential algorithm of (9) requires $k+2$ steps. This algorithm is, however, not parallelisable and would execute in the same time as a single processor if only one of the parallel machine's processors were to perform the algorithm while the others remained idle. The algorithm Janβen gives to simulate the procedure of [Fab86] is given below:

for j = 1 step 1 until k do
for i = 1 step 1 until n do
begin
get $x_i^{2^{j-1}}$
$x_i^{2^j} = x_i^{2^{j-1}} \cdot x_i^{2^{j-1}}$
write $x_i^{2^j}$
end

The time for this simulation algorithm would be $3kn$ steps. However, in this refuting example Janβen assumes that the processor suggested by Faber et al does not have sufficient registers and hence has to store intermediate results in memory. In a RISC processor with a large register memory the cost of context switching would be very small (±0.1%) and could be made non-existent. Furthermore, by interchanging the *i* and *j* loops, the temporary storage requirement would not be necessary. The simulation algorithm for (9) illustrated below clearly has $n(k+2)$ steps and achieves the same as the previous simulation algorithm.

for i = 1 step 1 until n do
begin
get x_i
for j = 1 step 1 until k do
$x_i = x_i \cdot x_i$
write x_i
end

What these examples and counter examples demonstrate is that any comparison on possible speedup obtained by parallelisation should be specific about the underlying computer architecture, the algorithm involved and form of parallelisation employed.

2.4 Discussion

Without getting involved in the debate whether superlinear speedup is indeed possible, the remainder of this thesis will deal with parallelising existing algorithms without making any major logic alterations within the algorithms. The thesis will look at parallelisation as the modifications made to an algorithm in order to break it down into several independent and dependant parts which may be executed concurrently and synchronously, including intercommunication in order to achieve a decrease in execution time.

This is the form of breakdown discussed by Faber et al in [Fab86], although the instruction units may not execute in similar times. This will result in the run time of the units executed in parallel to be, as a minimum, the duration of execution of the longest (time) program unit. Due to data and code dependency, some of these units may also not be able to commence execution until other units have terminated. This is known as code synchronisation.

Another influencing factor is the cost of communication. If the processors do not have access to the same global shared memory but have a local memory of their own in which data/results are stored, processors will need to communicate with each other in order to allow other processors access to this memory and the data/results within, as well as to inform other processors of decisions and progress made and to possibly instruct them on routes to take and/or tasks to perform.

We cannot, therefore, expect superlinear speedup (7) to occur although, ideally, linear speedup (5) is possible. If the cost of synchronisation and communication are included, the speedup and efficiency we can expect will be in the order of (3) and (4). Therefore, in order to achieve a great a speedup as possible, we should attempt to obtain speedup which is as close as possible to linear.

3 THE T800 TRANSPUTER - AN OVERVIEW [INM87,INM88a,INM89]

INMOS is an enterprise set up in 1978 by funding from the British government and chiefly dealt with high-speed static and dynamic RAMs aimed at high performance mainframe and minicomputer markets. However, at its design centre in Bristol, a revolutionary VLSI (very-large-scale-integration) microprocessor chip called the Transputer was developed and is now bearing fruit. Initially the Transputer was to be put in the market against the then current generation of microprocessors, such as Intel's 80286, but now is faring well against Intel's 80386 processor as a micro-controller and in other micro-processing systems. It is currently in use at various divisions of the CSIR and other academic institutions in South Africa as a high power processor.

The design philosophy of the Transputer is, however, very different from its competitors. Its name came from the amalgam of transistor and computer, demonstrating that it is intended to be a programmable component. It is based on the RISC (Reduced Instruction Set Computer) philosophy that it is the simple instructions which are most prevalent in any program, and it is these instruction which are executed most often. Furthermore, it is designed to communicate with other Transputers in parallel so that as many of them as desired may be linked together in a network and applied to a particular task, producing minimal overheads in processor time and physical interconnections. Even more importantly, that same program may be run on a single Transputer or across a network of Transputers.

The concept of using many processors to a common goal is widely recognised as an essential step in the creation of the so-called "fifth generation" of computers. In order to perform natural-language processing and other "intelligent" activities, huge databases will have to be searched, and, in order to achieve reasonableness, these operations may be performed in parallel. Such undertakings of concurrent processing may provide an addition four orders of magnitude of computational capability [Pou84]. The conventional method of connecting microprocessors together to perform parallel processing is through buses. This is however a tedious and onerous chore which is

complicated by the need for even wider parallel buses to be securely shared and by problems of synchronisation.

The Transputer avoids these problems by communication over high-speed serial links and by having the necessary synchronisation built into its instruction set. The T800 has four high-speed bi-directional serial links which can communicate 5, 10 or 20 megabits per second, or effectively a maximum of 2350K per second at full duplex. An eight link transputer is also currently under development.

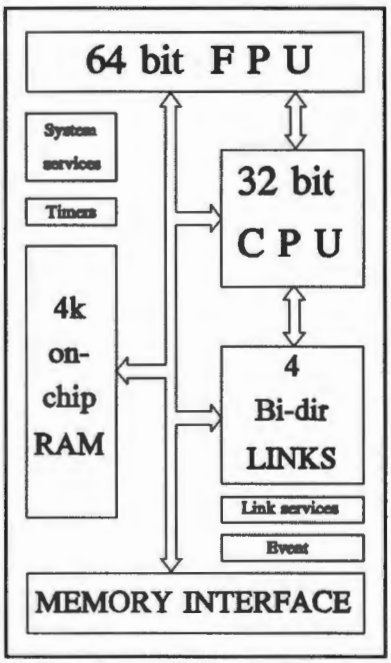

Figure 2 T800 Architecture

Each chip has up to 4K bytes of ultrahigh-speed on-chip RAM which replaces conventional microprocessor registers (agreeing with the RISC philosophy), an external memory interface and a peripheral interface all on chip. The on-chip RAM is divided up amongst different concurrent processes, providing each with their own workspace with windowing being performed on the on-chip RAM. The technique of task-switching then becomes merely a matter of switching a pointer to the current workspace or window. The processor itself contains its own logic to schedule these tasks, time-slicing typically every millisecond. Instructions vary in size, the smallest being a single byte containing a four bit opcode and four bit operand providing 16 byte instructions, to the largest which are often combinations of two or more of the byte instructions. The Transputer has a uniform address space, allowing up to 4 gigabytes of external RAM to be accessed. This method of addressing does not provide for virtual memory operation, although the University of Stellenbosch is currently working on a virtual memory management system for the Transputer, providing the full 4 gigabyte address space. The architecture layout of the T800 is illustrated in Figure 2.

Any number of processes (subject to memory size) may be executed on a T800 concurrently through the use of time-slicing. Each process is assigned a priority. Two types of priority exist on the T800: a high and a low order priority. Any process of high priority is executed until completion, or until it is not possible to continue any further, often due to communication. If more than one process of high priority exists, *no* time-slicing occurs between them. Instead the T800 queues the high priority processes and attempts to sequentially execute each one as far as possible, or until termination. Low priority processes, however, do time-slice amongst each other and operate concurrently. Processes of low priority are only executed when all high priority process have either terminated, or are blocked (cannot execute any further). If the T800 is currently executing a low priority process with blocked high priority processes and a high priority process becomes unblocked, the processor will, at the next deschedule point (± 1 millisecond), resume complete execution of the high priority process. Several independent process will, in high priority, be executed sequentially until termination one after the other, while in low priority will be executed concurrently, using time-slicing, until termination.

The various hardware modules, such as the link processor, numeric co-processor (Floating Point Unit-FPU), normal processor (Central Processing Unit-CPU), etc. all run in parallel, allowing the T800 transputer to run at a speed of 10 MIPS (million instructions per second), or a sustained 2.25 Mflop. Furthermore, the fact that the serial links, memory interface and processor can all operate independently and simultaneously makes transputer to transputer communication inexpensive. As transputers are also asynchronous, they don't have to share a common clock signal and delays in the point-to-point connecting lines can be tolerated within broad limits.

The transputer instruction set was also clearly designed to implement OCCAM efficiently, providing an almost one-to-one mapping of OCCAM to transputer instruction code (TCODE) and including some computer graphic instructions. OCCAM is in fact normally transformed rather than compiled into transputer assembly code where it is assembled and executed. Hence, OCCAM is the most efficiently executed

language, although there are alien languages (to the transputer) such as C, Pascal, Fortran and Lisp for which compilers are available. Benchmarks such as the Sieve of Erasthenes have been implemented in several different languages on the transputer with OCCAM executing 30% faster than the fastest alien language. The speed of execution of the alien languages are grouped together, all running at similar speeds.

The general design of the T800 forms part of the European ESPRIT parallel computer architecture project. This project involves the University of Southampton, Thorn-EMI and RSRE in the U.K., amongst others, and is set to develop a supercomputer constructed from a reconfigurable collection of 256 or more transputers.

4 PARALLEL LANGUAGES ON THE TRANSPUTER

4.1 OCCAM [Bow87,Bur88,INM84,INM88b,INM89]

The language OCCAM was named after William of Occam, a 14th century philosophy student who is known for the saying "*Entia non sunt multiplicanda praeter necessitatem*", better known as Occam's razor. The translation of this is "Entities are not to be multiplied beyond necessity", or rather "Keep it simple", illustrating that the founding principle of OCCAM was a minimalist approach with unnecessary duplication of language definition being avoided.

The OCCAM programming language is a high level language, designed to express concurrent algorithms and their implementation on a network of processing components. It arises from Prof. Tony Hoare's CSP (Communicating Sequential Processes) [Hoa85] at Oxford and David May's EPL (Experimental Programming Language) at INMOS who continues development of the language. OCCAM is sometimes seen as the assembly language of the transputer since OCCAM and the transputer were in fact designed concurrently at INMOS, with the transputer being able to directly execute OCCAM programs.

The OCCAM referred to here is OCCAM 2, which is significantly larger than OCCAM 1 as OCCAM 1 was only designed to carry the basic concepts, with only 22 reserved words. OCCAM 2, however, has a forte' of approximately 50 words and 20 symbols, but remains to be a small and elegant language. The major difference between it and conventional languages is that the concepts of communication, parallel execution and synchronisation are carried in its actual structure, making an ideal language for specification and behavioral description. It also allows the application of mathematical proof techniques to prove the correctness of programs, and will allow transformations, converting a process from one form to a directly equivalent form. The process of transformation hence may be used to any OCCAM program to improve its efficiency in any particular environment and to transform sequential algorithms into parallel ones.

OCCAM is based on the process model of computing, where independent processes communicate to each other over self-synchronising channels, either sequentially, concurrently or alternatively. These channels may have protocols attached, describing the structure of the data that is to be communicated, allowing entire arrays to be communicated as an entire block in one single process. Thus, through OCCAM, an entire system may be described as a collection of concurrent processes which communicate to each other through channels. Furthermore, these systems may be run on single transputers or across arrays of transputers without having to alter their topography. The OCCAM channel consequently describes inter-process communication in the abstract and does not depend or reflect on the particular hardware implementation. Thus OCCAM systems may be developed (written and tested) on a single transputer system before deciding or placing the system on an actual Network of transputers, specifying where individual processes are to be executed. The programmer may therefore be, essentially, unconcerned with the final implementation scheme chosen for the program.

Hence, for our purposes, OCCAM is an ideal language with which to examine communication between sequential processes, protocols, message routing and the parallelisation of algorithms, the topics of this thesis.

4.2 3L Parallel Fortran [3LL88]

The 3L Parallel Fortran is based on the same abstract models of CSP [Hoa85] as the transputer hardware. It is based on Fortran 8*X*, but incorporates several synchronisation, communications and parallel process libraries to administer such functions. Processes are viewed as *tasks*, each with a vector of *input ports* and a vector of *output ports*. The ports are the communication channels which connects the different tasks into an application consisting of concurrently executing tasks. As discussed in Section 5.1, channels or ports are uni-directional and point-to-point. Processes may be placed on processors with configuration software used to specify which processes are

to operate on which processors, the types of processors, the hardware links between the processors, how the tasks are interconnected through their ports, mapping software links onto hardware links etc. etc. Fortran subroutines may also be initiated as tasks, stopped, re-started and terminated, with provision for software links between the different concurrent subroutines.

The major disadvantage of this is that users must specify the amount of memory which is to be used individually by the code and the data sections of each task. The static memory requirements of the tasks and subroutines may be determined from a **decode** utility which is provided, but the dynamic requirements, such as stack and heap space are left up to the users' intuition and common sense. 3L admit themselves that these requirements are difficult to estimate and provide basic rules of thumb in order to allow users to "guestimate" the requirements.

Furthermore users are also required to implement their own routing algorithms to handle communication between different tasks situated on different processes, providing their own multiplexing and routing tasks. Necessary functions, such as task synchronisation, are not inherent in Fortran 8*X* as they are in OCCAM and are left up to the user to implement. Although semaphore libraries are provided, it is left up to the programmer to establish critical sections, prevent accidental sharing, synchronise tasks, implement shared variables and cater for other parallel processing functions. This incorporates an incredible amount of complexity into the programming task and introduces innumerable possible sources of error into otherwise seemingly simple systems. However, if programmers and users are well trained and accustomed to parallel processing, designing and running parallel systems, the possible problems introduced by concurrency and so forth, 3L Parallel Fortran can be a powerful tool through which effective concurrent systems can be created.

4.3 Other 3L Parallel Languages

The other parallel languages, such as C and Pascal, provided by 3L are implemented similar to Fortran with merely the above-mentioned parallel libraries being incorporated to introduce parallelism. Modula-2 is, however, an exception as it incorporates its own form of parallelism. Although the author does not have much experience with Modula-2 on transputers, only Modula-2 on PC based systems, Modula-2 should be a significant improvement on the other 3L languages due to the inherent parallelism of the language. This is important in the construction of concurrent systems as it allows compiler checking and validation of portions of code, as well as making the parallelism more visible, readable and understandable allowing errors to be detected and debugged more easily than systems where parallelism has been patched on.

4.4 Languages under HELIOS

HELIOS is a truly distributed operating system for transputer and other processor networks. It is similar to UNIX in its operating environment and instructions, being written also primarily in C. Inter-process communication is provided not through channels, but through a concept similar to the UNIX pipe. Concurrent processes may communicate directly to each other by merely writing to a "System" file with the receiving file merely reading from the file. HELIOS connects the two communicating processes so that the writing process sends the data directly to the reading process although there is no synchronisation as processes may be placed on different processors, with HELIOS handling all message routing and communication between the different concurrent processes. This provides the users with a great deal of simplicity in concurrent programming when using standard languages such as C, Pascal and Fortran, removing many of the technicalities and administrative functions (such as routing, memory allocation and sizing) of the 3L languages

However, there is a cost to be paid. This is the operating system overhead which is in the order of 30% of the processors' time. When application specific systems are created where the processors are dedicated to a specific function, one would rather not have such an overhead. Applications which take in the order of 3 hours to execute on a bare system will loose an hour to the operating system, giving a real execution time of 4 hours when running under HELIOS. Furthermore, although HELIOS is currently being marketed, it contains a number of documented problems (as well as some undocumented ones) which curtails the system reliability. For example, while I performed simple timing tests, the system would hang one out of seven times due to reasons unknown.

Helios is, however, a force to be reckoned with. The next release (version 2.0) due in July 1990 look promising with most of the problems of the earlier versions being ironed out. Helios removes most of the complexity of concurrent programming from the programmer, allowing them to concentrate the actual problem at hand, rather than the management and the other overhead details present when creating concurrent systems. However, as a reliable version of Helios was not available at the start of this thesis, this thesis will look at the aspects of using bare transputer networks and creating stand-alone applications which require more dedicated and reliable processor time.

4.5 Summary

OCCAM is clearly the best implementation language for systems which are to run on transputer networks. It incorporates parallelism in its construct, but is limited as a functional programming language for use in engineering (no dynamic variables, complex numbers, wordy when cross typing in arithmetic). Fortran on the other hand has been the language of engineers for many years, and will remain to be so for many more. It is, however, limited to the sequential format from which it originated with parallelism being incorporated as a non-standardised feature. Other languages for the transputer, apart from Modula-2, are similarly limited. In the parallelisation of

"dusty-deck" Fortran code onto transputer networks, a code rewrite into Modula-2 or the more suitable OCCAM-2 is not only labour intensive, requiring knowledge of the algorithms employed, but is both costly and time-consuming requiring an entirely new software development cycle. The solution is thus to maintain the Fortran code, but incorporate the CSP features through careful code restructuring or through utilising an OCCAM harness to handle the parallel constructs and process intercommunication [Hon90].

Rather than rewriting such "dusty-decks", this thesis deals with the parallelisation of the existing code using tools created specifically for the task and the implementation of the code using further libraries and methods developed with the intention of being used in such systems.

5 PROCESSES

5.1 Description

A process is an independent computation with its own program and data. It starts, performs a sequence of actions, and then terminates. Each action may be: an assignment, changing the value of a variable; an input, receiving a value through communication; or output, sending a value through communication. It may be, in its simplest form, a single statement, or large and complex like a disk operating system. A process thus represents a building block of the most flexible mode of concurrency known as MIMD (Multiple Instruction Multiple Data). MIMD implies that more than one distinct thread of control exists and that these threads can be executing different instructions and manipulating different data structures concurrently.

The individual processes of a concurrent system may communicate with other processes which are executing at the same time. Communication between process may occur (a) within the same processor if the conversing processes are running on the same single processor, or if the processes are distributed among several processors, communication may be through (b) common memory, or (c) through a communication network. Hybrids of the latter communication systems may exist. We will consider the communication of (c) in the remainder of this thesis since we will be always dealing with more than one processor. (b) has been omitted because communication through a common memory is complicated through the hardware required as well as the memory bandwidth. Furthermore, common memory communication is restrictive when the processors are physically far apart, and when adding or removing processors from a system. Transputers also do not communicate through memory but through high speed serial links.

In the process model of this thesis, illustrated in Figure 3, processes are viewed as tasks with inputs and outputs and communicate with each other through explicitly defined channels using message passing. These channels provide one-way, point-to-point, synchronised communication between concurrent processes and are used to

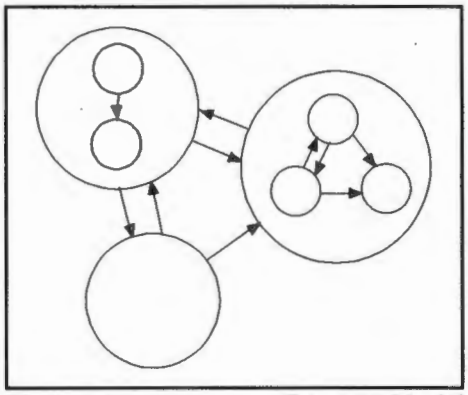

Figure 3 Process Model

connect together separate concurrent processes in the construction of complex concurrent systems. Consider, for example, two concurrent intercommunicating processes P and Q. A channel C_{PQ} connects P to Q where P can only transmit data and Q can only receive data along C_{PQ}. In order for Q to communicate to P, there must exist a second channel C_{QP} where only Q can transmit and P can only receive data along C_{QP}. No other processes can use either C_{PQ} or C_{QP} and P and Q can only use the latter channels as specified. Channels are therefore point-to-point and one-way. Hence, for each channel, there must two unique processes which it connects, one of which can only send data along the channel, and one which can only receive data from the channel.

Furthermore, communication along each channel is synchronised and hence may only occur when both the sending and receiving processes are ready for communication, as illustrated in Figure 4. Here we see two processes, labelled 1 and 2, communicating where process 1 attempts to send some data to process 2. The time increments from left to right. Note that the two processes may start independently of each other and may do some processing before the data transmission is to occur. When process 1 attempts to transmit the data, process 2 is not ready to receive, resulting in process 1 having to wait. After some time, process 2 becomes ready to receive, and the data transmission occurs with both processes resuming execution directly after the data transmission has been completed. The *Transmit time* is the time the data took to be transmitted, which may vary depending on the type of data transmission occurring. Note that the results would have been similar if the roles of Process 1 and 2 were reversed with Process 1 receiving the data and Process 2 sending the data. The process which is ready for communication first will have to wait until the second process becomes ready.

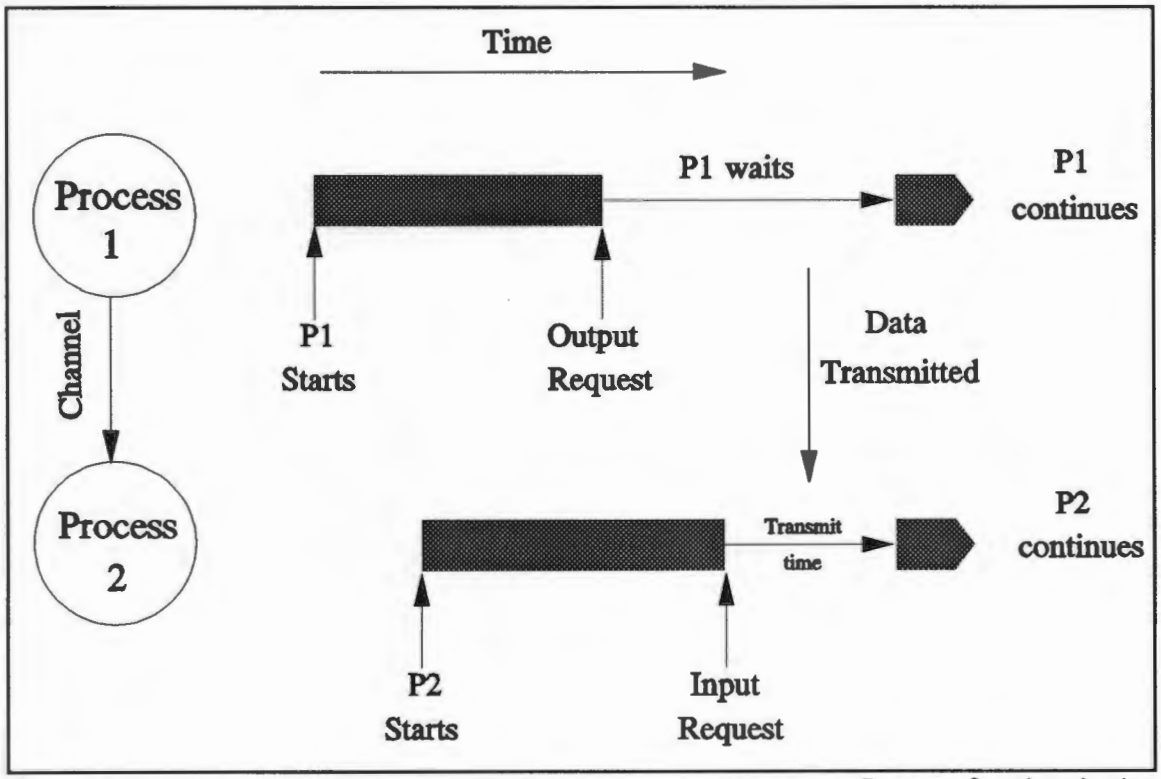

Figure 4 Process Synchronisation

5.2 Process Networks

A process network is a cumulation of interconnected processes working collectively towards achieving a common goal. The processes are connected through uni-directional point-to-point communication links (channels), each channel connecting two processes. The processes may operate concurrently and independently, spawning additional processes and terminating others. Each process typically has its own objectives to fulfil which it may achieve through its program, spawning and terminating processes, assisting and communicating to other processes. It is the culmination of all the process objectives which constitute the network goal or goals. The network communication is through the channels. Processes which are not directly connected through a channel to each other and which need to be able to communicate with each other, do so by communicating with one or more intermediary processes, directly or indirectly, who convey the message to the destination processes.

An example network is illustrated in Figure 5 where the circle represent the processes and the paths represent the uni-directional channels. In this example, messages from *A* to *B* have to travel through intermediaries *C*, *E* and *D*. Note that messages from *A* to *B* may also go through *C*, *E*, *F* and *D*.

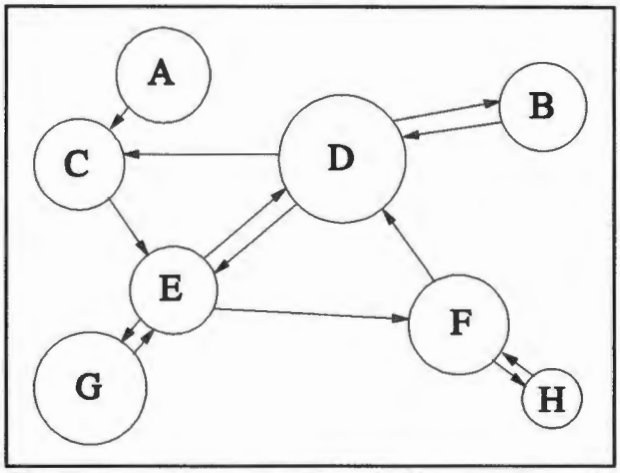

Figure 5 Process Network Example

5.3 Process Placement

Most processors today will execute more than one process concurrently. Thus, in a network of more than one processor, processors will often execute more than one process of a process network concurrently. The choice of which processes to place on which processors is the source of much debate, theory and discussion today and may in fact be a very complex decision [Han89, Hen90]. An example of processor placement for the process network example of Figure 5, using a network of 5 T800's is given in Figure 6. The enclosing boxes represent the processors, the circles represent the processes, the arrows between the boxes and from the boxes to the processes represent the hardware channel connections and mappings respectively, while the arrows between the processes represent the internal software channels.

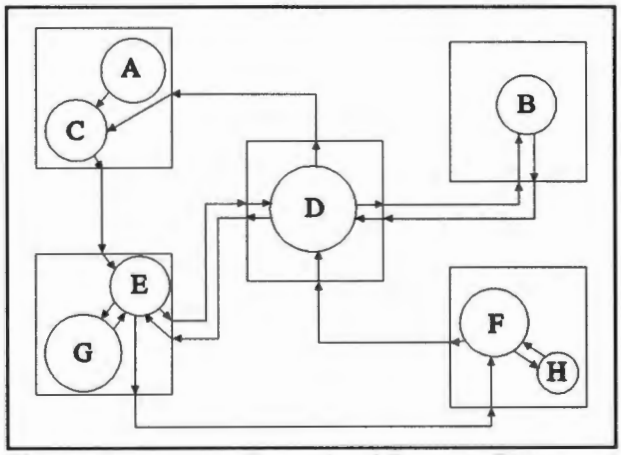

Figure 6 Example of Process Placement

There are several factors which influence the placement of a process on a processor. The two most important are:

1) *Load balancing*

   When more than one processor is involved, as is often the case, it is desirable to place the processes on processors in such a manner that the time spent idle by any processor due to blocking of its processes it at a minimum. Any idle time is processor time which is wasted, resulting in a low processor efficiency, a smaller speedup and hence a greater overall run time than what may actually be realised.

2) *Communication*

   In order to minimise communication overheads and the number of intermediate processors involved with inter-process communication, processes which communicate often to each other should reside either (a) on the same processor, (b) on directly connected processors, or (c) on processors connected by a minimum of intermediary processors. The total time spent by processors on communication must therefore be reduced to a minimum. One method whereby this can be achieved is to rate processes in pairs where each pair is given a weighting according to the amount of process inter-communication. Pairs with the highest weighting should attempt to satisfy (a) first, while those with the weaker weightings should aim to satisfy (b). Process pairs with the least amount of inter-process communication should attempt to satisfy (c).

There is obviously strong interrelationships between the above two factors: Process placement due to load balancing effects the physical positioning of the processes and therefore effects the inter-processor communication overheads, normally substantially greater than intra-processor communication; Process placement minimising inter-processor communication similarly will effect the load balancing. Therefore, in placing processes, one has to try and minimise the sum of the time spent idle and the time spent communicating such that the overall time spent by the network in achieving its goals is at a minimum.

This thesis is not concerned with the topic of parallel process placement as this is a whole complex subject of its own and is the topic of Ph.D. and M.Sc. theses, such as the M.Sc thesis *Parallel Process Placement* of Caroline Handler [Han89]. Instead, this thesis deals with the parallelisation of algorithms, leaving the placement of the individual processes up to the paralleliser and their intuition. In particular, in a process farm environment, the number of processes on processors inherently corresponds to the processor network size. Furthermore, the load created by each process in a farm environment is either very similar or very erratic causing load balancing to become a pointless superfluous exercise due to a high degree of predictability or unpredictability.

5.4 Process Farm

5.4.1 Introduction

A process farm is a network consisting of master and worker processes. The master processes are responsible for creating and distributing work to the workers, ensuring that all the workers are kept busy, and generally also handling an I/O. They must also collect and collate the results from worker tasks which have completed their assignment. The duty of the master is therefore to generate tasks for the workers, distributing the tasks amongst the workers, while simultaneously collecting and collating the results of completed tasks from workers as illustrated in Figure 7.

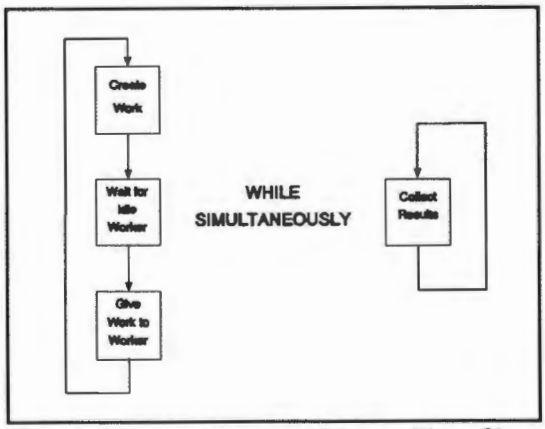

Figure 7 Master Flow Chart

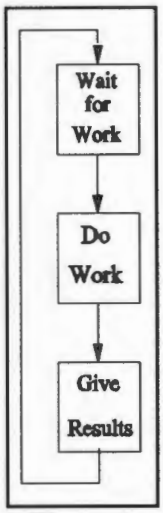

Figure 8 Worker Flow Chart

5.4.2 Farm Workers

Workers generally constitute the task force responsible for solving the algorithm. In general, the greater the number of workers, the greater the amount of work which can be performed by the task force, or alternately, the same amount of work may be performed in a smaller amount of time. Hence the saying "*Many hands make light work*". The workers perform the intense computations required by the algorithm, and with many of the intense computations being performed in parallel (where possible) the time required to solve the problem is reduced. Normally a single worker process will reside on a single processor. If more than one worker process resides on a processor, then the processor will have to divide its time up between the different processes. Although the worker processes will be executing concurrently, they will not be executing in parallel. Hence no speedup and therefore no time benefit will be obtained in this situation. In fact, dependant on the processor, the speedup may be inhibited due to the processor overhead of time-slicing. A possible flow chart for a worker is illustrated in Figure 8.

The workers may all be identical processes performing exactly the same task, or, several different "groups" of identical processes, each group performing a different function but members of the same group performing similar or the same type of function.

5.4.3 Farm Communication

As the speed of algorithms with a process farm design is dependant on the number of worker processes with one per processor, the number of processors is often large in order to facilitate as much of a speedup as possible. Furthermore, normally only one master process is required in this type of environment. Therefore, the master has to be able to communicate with all the workers, even those worker processors which are

not directly connected to the master. It is not the responsibility of the worker processes to handle or be aware of any communication, other than the necessary communication from it to the master and vice versa. It is also undesirable that the worker processes concern themselves with network communication since their task is to solve the problem at hand. Router processes are therefore introduced into the farm network to handle communication and act as support processes, not actually solving the problem, but assisting other processes in doing so. It is the responsibility of the router processes to ensure that messages are conveyed between the master and workers as fast as possible, using the shortest route whenever possible. The worker processes are therefore not required to have any knowledge of the network configuration. The only task which the worker processes should have to perform is that required of them by the master, with the router tasks handling inter-processor communication.

Consider now the type of communication present in such a network:

i) The master, or farmer, must send messages out to his workers. These messages can consist of data which the workers require in order to be able to perform their duties, a list of instructions on what type of operation is to be performed on the data, special instructions such as the abortion of a job, as well as requests for information from the workers, such as "*How far are you now ?*" etc. This is the outgoing traffic from the farmer.

ii) The worker, in turn, may need to send the processed data (results) back to the master. They may also send messages to inform the master of their status, request "*Exception*" data which the farmer may normally not provide them with, request further information which only the farmer may have knowledge of, as well as sending I/O requests to the farmer. This is the incoming traffic to the farmer.

iii) Workers also may need to communicate amongst themselves in order to share and exchange data and information. An example of this is in Cellular Automata where a region is divided into cells with workers typically operating individually on each cell. A worker, in this case, needs to exchange its border information

with the worker processes which are operating on the cells adjoining its cell in order to be able to function correctly. This is the internal network traffic.

As discussed previously, routing tasks are required in the network in order to allow communication between the master and indirectly connected workers. These routing tasks are placed on the same processor as the worker tasks in order to optimise processor efficiency, and run at a higher priority than the worker process. By making the routing tasks high priority tasks, the efficiency of the network is increased due to greater processor utilisation and small processor idle times. As the T800 transputer is being used to implement all of the algorithms in this thesis, the priorities referred to are that of the T800 transputer discussed on page 12. The proof of high priority routing tasks resulting in greater processor efficiency is given below:

High Routing Task Priority

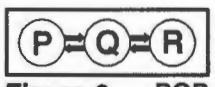

Figure 9 PQR

Consider the illustration in Figure 9 of three processors *P*, *Q* and *R* where there are two bi-directional links between *P* & *Q*, and *Q* & *R* respectively. Therefore, in order for *P* to communicate to *R*, messages must go through intermediary *Q*. Now place a worker task *p* and a routing task p_r on *P*, worker task *q* and routing task q_r on *Q* and worker task *r* and routing task r_r on *R*.

Consider *p* sending a message to waiting *r* with process *q* active with processing time greater than 1 second remaining. To find the time taken for a message to pass from p_r to r_r consider the following where :-

(a) both worker and routing tasks are at the same priority, or the routing task is at low priority and the worker at high priority. (If both were high priority, and *q* is active, this will result in the same effect as the latter case.) In the former case, both processes have to share processor time through time-slicing. *p* passes the message to p_r which then waits an average time $t_{wait\text{-}a}$ for q_r to become active before the message transfer can take place (since *q* is currently utilising processor *Q*'s time). The transfer from p_r to q_r takes time $t_{pq\text{-}comm}$ plus the time t_{q1} which *q* spends active during the transfer operation. Since *r* is waiting, r_r may

fully make use of R's time and the transfer from q_r to r_r takes time $t_{qr\text{-}comm}$ plus the time t_{q2} spent by q being active during the second transfer. The total time that r_r waits for the message from p_r is thus

$$t_{wait\text{-}a}+t_{pq\text{-}comm}+t_{q1}+t_{qr\text{-}comm}+t_{q2}$$

(b) the routing task runs at high priority with the worker at low priority. When p_r receives the message and to passes it on to q_r, the average delay $t_{wait\text{-}b}$ is present since q is busy (even at low priority, there is a delay before the high priority task gains CPU control [Pin88]). Since the q_r is at high priority and q at low priority, we will have $t_{wait\text{-}b} \leq t_{wait\text{-}a}$. q_r then becomes active and exclusively utilises the processor time until the communication has completed. This results in times t_{q1} and t_{q2} being 0 as q remains inactive until q_r terminates. The time for communication in this case will be

$$t_{wait\text{-}b}+t_{pq\text{-}comm}+t_{qr\text{-}comm}$$

Since $t_{q1},t_{q2} \geq 0$ and $t_{wait\text{-}b} \leq t_{wait\text{-}a}$ we get

$$t_{wait\text{-}b}+t_{pq\text{-}comm}+t_{qr\text{-}comm} \leq t_{wait\text{-}a}+t_{pq\text{-}comm}+t_{q1}+t_{qr\text{-}comm}+t_{q2} \qquad \textbf{(10)}$$

The times of all interrupts have been ignored as they are simply the same constants on both sides of (10). The cost of time slicing has also been ignored for similar reasons, although it will intuitively increase the right hand side of (10) as there will be more time slices in (a) with p_r at low priority than (b) above. The cost of time-slicing in today processor is also very small, almost negligible.

Consider now:

(a) If q was active at high priority and q_r at a low priority then t_{q1} and t_{q2} would both be 0 since q would have terminated. However, $t_{wait\text{-}a}$ will then be greater than 1 second as it is the remaining run time of q, hence it is greater than the maximum millisecond wait $t_{wait\text{-}b}$. $\Rightarrow t_{wait\text{-}b} < t_{wait\text{-}a}$

(b) If q and q_r are both at low priority and have to share Q's time, then t_{q1} and t_{q2} both are non-zero. $\Rightarrow t_{q1}, t_{q2} > 0$

(10) then refines to

$$t_{wait-b}+t_{pq-time}+t_{qr-time} < t_{wait-a}+t_{pq-comm}+t_{q1}+t_{qr-comm}+t_{q2} \quad (11)$$

From (11) we can see that the time for communication in (a) is greater than (b) above. Thus, in (b), r spends less time waiting for the message resulting in a greater efficiency. If r is a critical task, then the greater the execution time of r, the greater the total execution time of the algorithm. Thus (b) is more efficient than (a) and all routing tasks should run at a high priority with the workers running at a low priority.

Conclusion

Applying (b) reduces the idle time of the worker processes since messages are conveyed as fast as possible, with the router only allowing the worker on the same processor CPU time when it is blocked, typically when it waits on a channel (for a message to receive and forward) or on a timer. The worker is interrupted when the router has work to do (e.g. forwarding of messages) and has control returned when the router has completed its task and once again is blocked. This also reduces the synchronisation barriers as any process will always be able to send messages immediately if its routing process is idle, and will be able to continue operation after having passed the message to the router since it no longer has to synchronise with the destination process. With synchronisation barriers reduced, the processor performance increases [Axe86] resulting in greater efficiency and a greater speedup. This was confirmed in the Airflow Modelling example of Section 9 where an average increase of performance of 30% was obtained when using the priorities discussed above.

5.4.4 Farm Construction - Method 1

Consider now the construction of a process farm. If *p* processors are available, we will construct a farm with *p* workers and a single master. Since the processor containing the master will be idle when all the workers are occupied, a worker task of low priority is placed on the same processor as the master task (running at high priority) in order to utilise that processor more fully, with the remaining *p-1* workers placed on the remaining processors. Normally a worker is placed with the master as the master task should be able to create and distribute work faster than the workers can complete it. Hence the master will be idle some of the time so a worker is placed on the same processor in order to make use of the master processor idle time. If the master could not distribute work fast enough to the workers then there will always be idle worker processes, resulting in a poor processor utilisation and illustrating that the farm will operate with just as much throughput with less processors. Ideally, all worker processes should be busy at any one time, and the idle time should be kept at a minimum. Therefore, through placement of a worker on the same processor as the master, the master processor is kept busy doing something useful while it is waiting for a worker to become idle in order that it may distribute more work to it.

In order to be able to cater for communication to processors not directly connected to the master processor, consider placing a routing task per processor to intelligently handle the protocols and message passing or communications. Such a farm network is illustrated in Figure 10.

A farm network of the type illustrated in Figure 10 was implemented in the first few farm networks of the Airflow Modelling Example (Section 9). The network would operate to completion 4 times out of 5, but on the 5th time it would "hang". Due to a lack of debugging tools available at that stage, the problem could not be pinpointed or replicated with debugging statements incorporated in the code. The logical conclusion was that the problem was one of synchronisation since the debugging statement temporarily rectified the error by introducing synchronisation delays. After

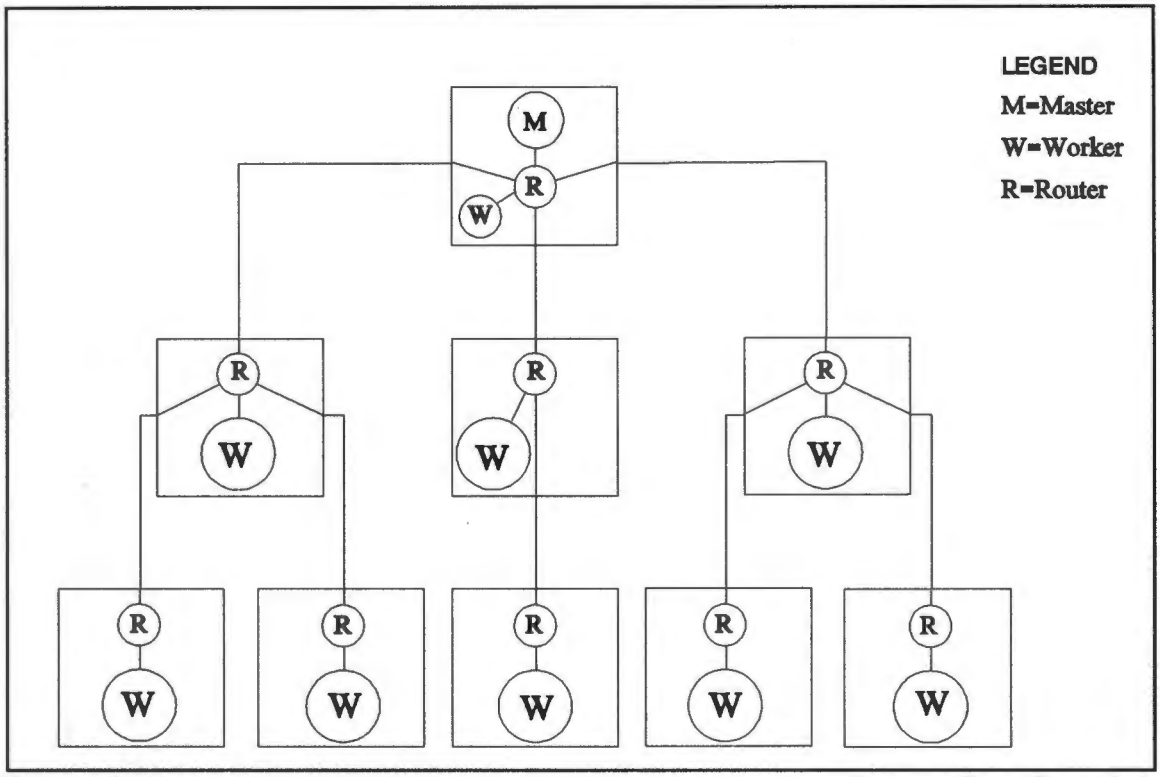

Figure 10 Process Farm Example 1

weeks of debugging and unsuccessful attempts to find the problem, and finally merely continuing work ignoring the problem, a hardware fault was later detected and rectified.

On returning to the initial airflow farm, the code then completely and correctly executed on the test examples 10 times out of 10. It was then decided to see how the network would perform under a greater load and hence greater resultant communication. This was achieved by increasing the amount of work to be performed by each worker, resulting in an increase of work and result packet size. This time at least two of the workers would fail to return their results 1 out of 10 times, returning correct results in the remaining 9 times.

Fortunately this time the problem was found fairly quickly to be a deadlock situation between two routing processes, although the routing process pair which would deadlock varied. The deadlock would result when the two routers attempted to communicate

to each other almost simultaneously. Neither of the routers could accept the message to be forwarded from the other until the other had received the current message. (See Figure 11.) Since routers were idle most of the time, and because the problem would only result when routers actually collided, this resulted in sporadic occurrences of the deadlock which would occur with greater frequencies when the network traffic was increased. This method is therefore unsuitable.

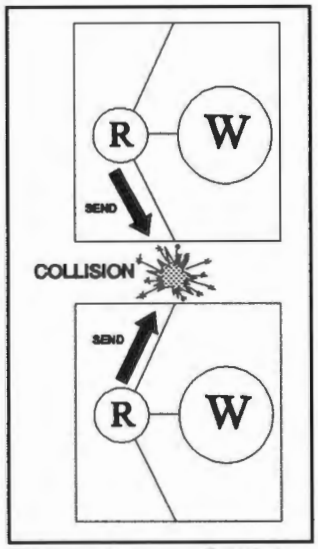

Figure 11 Collision Deadlock

5.4.5 Farm Construction - Methods 2, 3 and 4

Various solutions to the problems above, including the hardware problem thought to be synchronisation trouble, were examined. Most were found to be either inefficient, wasting precious CPU time, or rather complex to implement.

Some of the discarded solutions were:

2) Router Idle

Ensure that the receiving router process is idle before the sending router process were to attempt to send. This could be achieved by sending p work packets (assuming p workers) to the farm, one packet per worker, with the workers only starting on receipt of a broadcasted *go* signal from the master. The master would then only issue work to idle workers if all workers sharing part or all of the path from the master to the idle workers were also idle, ensuring that no result and work packet could collide. This is extremely wasteful of processor time if the work volume per packet varied. All workers with common paths to the master would then have to wait until the slowest worker has completed its task resulting in a large idle time, especially if the slowest worker was in the order of magnitude m ($m \gg 1$) slower than the other workers. The faster workers would be able to complete $m-1$ more tasks if they did not have to wait for the slowest worker. Since m could be very large, this solution was discarded as being inefficient.

3) Pipeline

Adopt a pipeline communication method where work is inserted into the farm at one end, and the results would appear at the other. Communication is thus uni-directional through the pipe, although the pipe may fan-out in the beginning and fan-in towards the end. This does, however, restrict the type of network topologies which may be used in implementing this "pipeline" farm. For example, problems arise with processors which are connected to the network with single links. Furthermore, the route taken for messages will normally not be the shortest, resulting in unnecessary traffic and communication overheads. The decision concerning the construction of the pipe is also problematic. Therefore, it was reasonable to exclude this choice of farm network as a solution.

4) Token Exchange

Utilise a token exchange method whereby tokens are shared between processor pairs requiring to communicate, with the processor owning the token being allowed to communicate. The token may be released by a processor to its sibling processor after a certain number of messages have been sent, or after a certain period of inactivity has been recorded. Each processor may also handle a number of tokens, with a minimum of one, dependant on its connectivity and the message routing method selected.

Release of a token after a certain amount of messages have been sent requires previous knowledge of the network load and communication. It also is restrictive as exception messages cannot be included and incorrect choice of message volume before releasing the token may result in very poor communication, even livelock. The information regarding the message volume can normally only be obtained through simulations or calculations and is restrictive in that these calculations will have to be repeated when processors are added or removed from the network, or when the farm load is varied.

The choice of releasing the token after a certain period of inactivity is a more reasonable one, although the optimal inactivity time for this release to take place is

also dependant on the network load. However, the calculation of the optimal time may be performed dynamically, resulting in an improved performance. If too small an inactivity time is chosen, precious processor time will be wasted by processors merely exchanging tokens, while too large an inactivity time results in messages having to wait an unacceptable extra delay time before being forwarded. Heavy communication load may also result in process lockout when the required delay never occurs due to heavy traffic from a certain direction.

Due to imbalances and exceptions which can occur in a network, both methods of token exchange are wasteful of processor time due to the processor time required to maintain optimal message passing, and because of the delay times encountered when a message is suspended due to the routing process not having the required token in order to communicate.

5.4.6 Farm Construction - Method 5

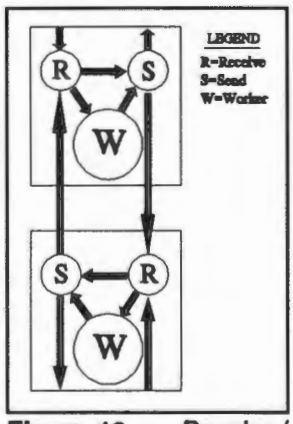

Figure 12 Receive/ Send Routers

In order to prevent the deadlock situation of Section 5.4.4, two router processes at high priority are introduced per processor, one receiving data into the processor with the other transmitting data from the processor, thereby alleviating the deadlock situation. This can be seen when the message collision causing deadlock in Section 5.4.4 occurs. Both transmitting routers may attempt to send their messages simultaneously, however, the receiving processes on either end are ready to receive. One of the receivers then actually accepts its message sent by the other processor's transmitting process. This may then be followed by the transmission of the message travelling in the opposite direction. The two messages thus pass each other without a deadlock collision occurring.

However, messages which require an intermediary processor are, on receipt by the *Receive* router on the intermediary processor, passed on to the intermediary *Send* router for transmission. This introduces a second deadlock complication, similar to that of Section 5.4.4, although here the deadlock results between two intermediary processors. This is easily illustrated using a form of petri-net where the tokens represent messages, and the states represent processes.

Consider part of a network containing four processors, each with processes w_i, s_i and r_i ($1 \leq i \leq 4$) where w_i, s_i and r_i are the worker, sending router and receiving router processes of processor i respectively and where processes w_1 and w_4 are communicating intensively to each other. Also, both w_1 and w_4 can absorb messages at any time. Note that each process handles only one message at a time and must release the message to another process before if can accept the next message. The petri-net for this example is illustrated in Figure 13. w_2 and w_3 have been omitted for simplicity.

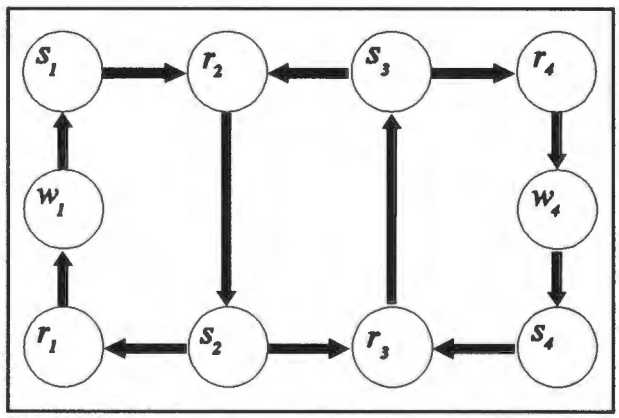

Figure 13 Petri-Net Illustration

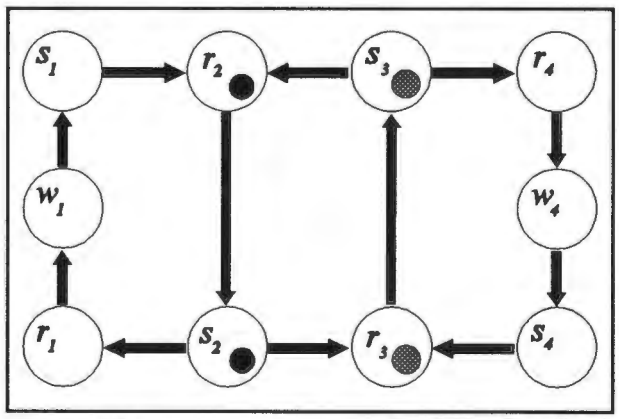

Figure 14 Petri-Net Deadlock

Messages, or tokens, travelling from w_1 to w_4 must travel through s_1, r_2, s_2, r_3, s_3, r_4 in that order, and messages from w_4 to w_1 must travel through s_4, r_3, s_3, r_2, s_2, r_1 in that order. Consider both w_1 and w_4 generating messages, and the following sequence of events:

1) w_1 and w_4 both fire an initial message into s_1 and s_4 respectively.
2) These messages move on to r_2 and r_3 respectively.
3) w_1 and w_4 both fire a second message into s_1 and s_4 respectively.
4) The first two messages move into s_2 and s_3 respectively.

5) The second two messages move into r_2 and r_3 respectively.

We now have a deadlock situation illustrated in Figure 14 where the solid tokens originated from w_1 and the shaded tokens originated from w_4. The message in r_2 cannot move into s_2 until the message in s_2 has moved into r_3. The message in r_3 in turn cannot move into s_3 until the message in s_3 has moved into r_2. None of the messages can be forwarded to their destinations.

This type of network was also used initially in the Airflow Modelling Network of Section 9 and would function perfectly without deadlock until a very high communication load occurred. This did not arise until tests with nine processors, configured in a line, and high communication rate brought the potential deadlock situation to light.

This problem may be overcome, however, by introducing a queue buffer for communication between the r_i and s_i routing processes of processor i. This allows r_i to release its token into the queue buffer, allowing any connected s_j to transmit its message, freeing the deadlock as long as the buffer does not become saturated. This method therefore requires sufficient memory storage from the holding processor for the buffer. The buffer should also be large enough to ensure that deadlock may never occur. As the buffer size may vary greatly due to different communication loads, certain buffers may require a large memory space which the holding processor may not have available. Such buffers may also place an unnecessary communication overhead on the farm.

A simpler technique of solving the deadlock situations of Section 5.4.4 and Section 5.4.6 would be for the routing processes to merely timeout after being locked for a certain period of time and discard their current messages. This, however, results in undesired unreliable communication, resulting in the necessity of acknowledgements, further increasing network traffic load and introducing unnecessary complexity.

The actual process organisation and placement in a farm is therefore a seemingly simple task, it is, however, hidden with all sorts of potential problem areas. Decisions which initially seemed reasonable turned out later to be non-viable solutions. Thus, the behaviour of even quite simple parallel systems can be astonishingly complex. The fact that a program functions correctly today, with some particular set of input values, is no guarantee that it will not fail tomorrow with the same input. Also, simply tracking down a bug in a parallel program can be exceedingly difficult. This is because of the combination of logical complexity, non-repeatable indeterminate behaviour, and the lack of debugging tools [McG88].

5.4.7 Farm Construction - Method 6

Messages in most farms are of type (i) and (ii) discussed in Section 5.4.3. Instead of using two routers to handle incoming messages and outgoing messages from a processor, two routing processes are used to handle incoming and outgoing messages (types (i) and (ii) above) for the farm respectively. This eliminates the connection between the two routing processes and, since messages are either directed to or away from the farmer, eradicates message collisions. Also, the routers used to convey messages away from the farmer are not the same as the routers used to carry messages to the farmer. (By definition, no router is ever used to handle messages in both directions.) Even when the farm network is totally flooded with messages, no deadlock will occur as all messages, assuming that they will always be accepted by their destination, will always have an unblocked path apart from the preceding messages travelling in the same direction. This technique, illustrated in Figure 15, is the most effective and is currently employed in most of the farm structures in AEROTEK (Division of Aeronautical Systems Technology, CSIR) today.

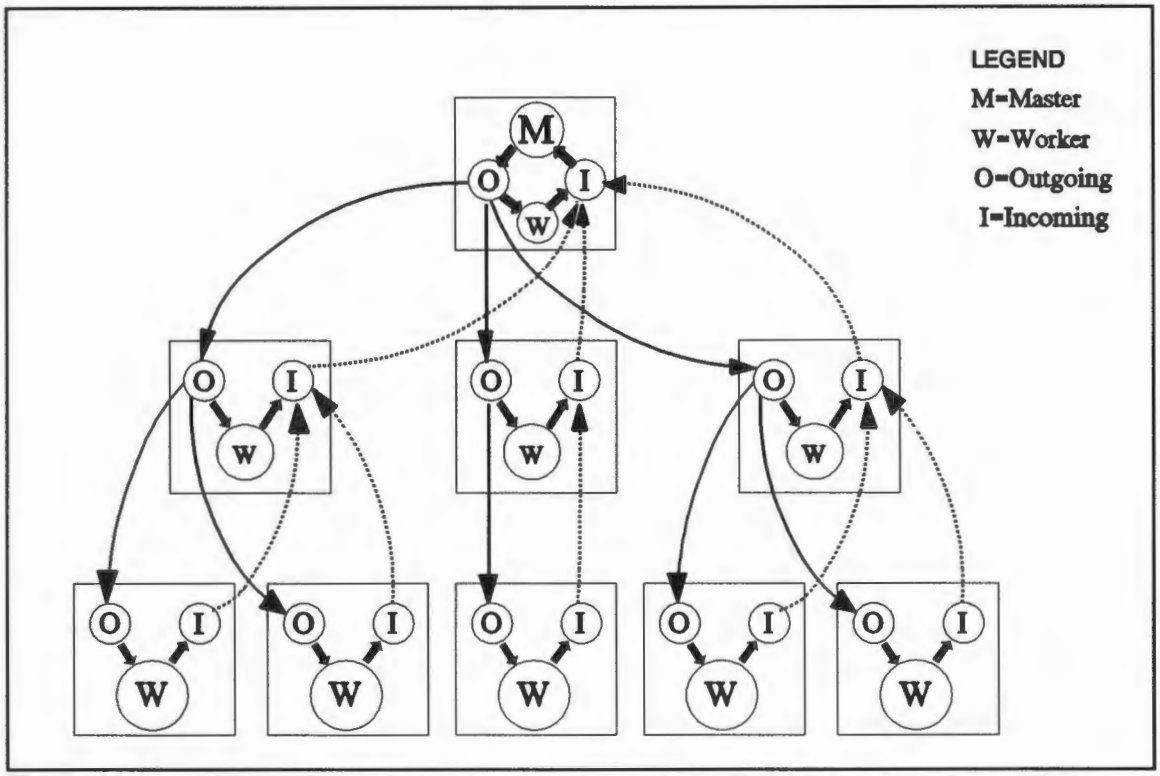

Figure 15 Farm Network, 2 Router Processes

5.5 General Communications Technique

What can be learned from the previous discussion is that in order to prevent deadlock, communication routing processes should always be available to perform their respective functions. Furthermore, the actual tasks performing the work should not be concerned with network communications. Instead dedicated communications processes should control inter-process and inter-processor communication. Experience has shown that in order to reduce the possibility of communications deadlock to a minimum, there should be a high priority receiving process and sending process for each bi-directional transputer link. This gives four receiving and four sending routers per transputer respectively.

Each receiving router monitors its input link for activity, and is linked to the sending routers of the other three links (excluding the same link as input and output on the same link does not make sense unless the input and output wires of the link are

connected to different transputers, hardware requires that they normally are the same processor) as well as any resident process requiring communications through software channels. This gives a fan-out situation with messages arriving and either being distributed for further communications or being passed directly to its destination. The sending router, by contrast, has a fan-in situation where messages for forwarding are accepted from any of the three receiving routers as discussed previously, as well as from processes requiring communication. The messages are then passed through the process' corresponding output link.

The messages are conveyed using a shortest path packet-switching strategy where the message body is prefixed by the links numbers of the different processors through which the message has to travel. For example, in the grid network of Figure 16, a message from processor 0 to processor 8 would have the header (2,2,3) indicating that it must travel through link 2 of the current processor (0), link 2 of the next processor (3) and finally through link 3 of the last intermediate processor (6) in order to get to processor 8. Similarly, a message from processor 5 to processor 6 would have the header (2,1) indicating that it must be output through link 2 of the current processor, then through link 1 of the next processor (8) in order to reach processor 6. Furthermore, messages between any processor pair always follow the same path through the same intermediate processors using a shortest path algorithm.

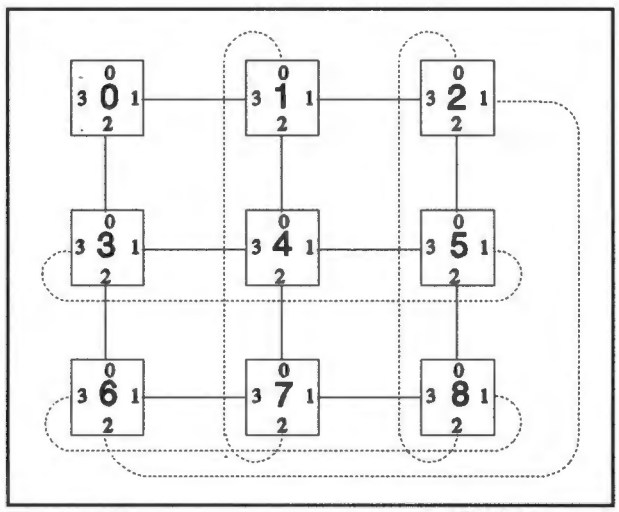

Figure 16 3x3 Grid Network

Even the above configuration does not prevent deadlock from occurring. For example, a circular deadlock occurs in the example below and is illustrated in Figure 17. The solid arrows represent the hardware links, the shaded arrows the internal software channels, the circles at the head of the solid arrows are the receivers while those at

the tail are the senders. Note that only the links, processes and processors concerned are illustrated.

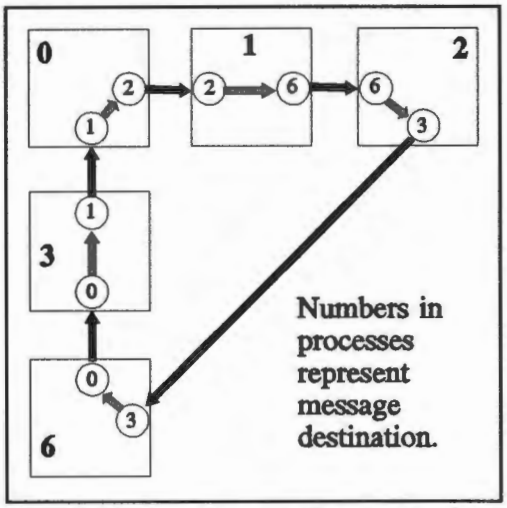

Figure 17 Circular Communications Deadlock

Example

a) P_0 attempts to send two messages in succession to P_2 via P_1.
b) P_1 attempts to send two messages in succession to P_6 via P_2.
c) P_2 attempts to send two messages in succession to P_3 via P_6.
d) P_6 attempts to send two messages in succession to P_0 via P_3.
e) P_3 attempts to send two messages in succession to P_1 via P_0.

Thus, in a network where processors have to communicate with every other processor, using a shortest path algorithm, deadlock prevention schemes are required. Many different deadlock prevention strategies exist for such configurations, however, we shall confine ourselves to a farm network where method 6 is the most effective technique. This technique also allows the farm network to allow processors to communicate directly with their immediate neighbours, possibly in order to share and exchange information, as long as the sending and receiving of this information is conducted concurrently.

As a real time experiment, test runs were made on the network illustrated in Figure 16 where each processor would randomly send a 1K message to any other processor, after a random delay time used to simulate work. The routine scheme used was that previously mentioned with four receiving and four sending routers per transputer. The time before deadlock occurred was determined as an average over 3 test runs, with the internal low priority clock used as the timer and are illustrated in the table below. Where a single delay time appears, a fixed time delay was used, while those where time ranges appear represent any time between the range with equal probability.

Fixed Delay Time (μsecs)	Time to Deadlock (Hr:Min:Sec)	Random Delay Time (μsecs)	Time to Deadlock (Hr:Min:Sec)
0	0:07:25	0-320	0:06:18
320	0:05:43	320-640	0:04:08
640	0:04:53	640-960	0:11:21
960	0:11:21	960-1280	0:23:26
1280	0:16:43	1280-1600	0:27:57
1600	0:45:04	1472-1792	1:17:33
1920	2:06:20	1728-2048	1:24:26
2240	2:33:30	2112-2432	1:36:06
2688	2:19:52	2496-2880	2:01:10
3200	2:25:52	2944-3584	4:52:17

As the above table illustrates, in the cases where fixed delays were used, deadlock occurred sooner than when bounded random delay times were applied. This can be attributed to process and communication synchronisation occurring, due to processes sending messages continually in the same time intervals. This allows the possibility of the above deadlock occurrence to be greater than if the delay times varied within a certain range. This delay time variance reduces the possibility of process synchronisation from occurring, and hence reduces the deadlock occurrence possibility.

6 FARM PARALLELISATION

The most important aspect for parallelisation is whether parallelisation of the code is actually possible. This is determined by investigation of the different sections of code and data flow of the program and the relationships between each. Not all code has calculations which may be performed independently of each other, hence, not all code may be parallelised. However, some parts of a program may be allowed to operate concurrently with each other, sharing and communicating data and program states between themselves, working collectively towards a common goal.

Parallelisation of code often will introduce additional overheads, such as that of communication and synchronisation, into the code. These overheads may be large enough to cause the parallel version to run slower than the sequential version. Since our main reason for parallelisation is to obtain a decrease in run time, it is desirable to establish the cut-off point where parallelisation of an algorithm will not obtain any speedup, and also to estimate what sort of decrease in run time will be obtained, if indeed any. The speedup obtained for the parallelisation of code is dependant on the code itself, the data on which the code operates and the target machine on which the code is to be implemented. More importantly, the ratio of calculations to data communication is the determining factor.

6.1 Transputer Calculations/Communications Ratio

Let us assume that the current T800s are being used in a network configuration, running at 20Mhz with the links at 20MBit/sec. This provides a 10 MIPS processor, with a 1 byte data transmission time of 0.3814697 μsecs. The time ratio of 1 instruction execution to a 1 byte data transmission is therefore 1:3.814697.

Consider a system with p processors, each running identical code consisting of x instructions. We assume that each instruction executes in 0.1 μsecs, otherwise we

break the instructions down further so that each instruction takes 0.1 μsecs to execute. Furthermore, assume that the code operates on 1 byte of data and that there are a total of n bytes of data to be processed. The sequential run time of such a code is therefore:

$$t_{sequential} = n \cdot x \tag{12}$$

Now assume that $p=n\geq 2$, so that each processor can execute concurrently on its own byte of data. Assume also that the transmit time for $(n-1)$ data items is less that the time for x instructions to execute. Also, let us assume the best case that we can communicate directly to each processor. Let the transmit time 0.3814679 μsecs be represented by the constant α. The parallel execution time will be the time taken to sequentially transmit the data to each processor $\{\alpha(n-1)$ μsecs$\}$ plus the execution time of x μsecs for the data to be calculated on the data distributing processor, plus the time to receive the data back from the processors $\{\alpha(n-1)$ μsecs$\}$. A data item is left on the processor distributing the data in order that it may perform useful work while waiting for the results from the other processors since we wish to obtain the high degree of efficiency. The parallel run time is therefore:

$$\begin{aligned} t_1 &= \alpha(n-1)+x+\alpha(n-1) \\ &= x+2\alpha(n-1) \end{aligned} \tag{13}$$

If a single processor were included to merely distribute the data and collect the results to the n processors, the run time would be the time to transmit n bytes plus the time to receive the n result bytes plus the time spent waiting for the results. The time spent waiting for the results would be the remaining run time of the worker which was first sent the results. Since $(n-1)$ data packets are to be sent, the remaining run time of this first worker would be the total run time, less the time taken to sent the remaining data. Thus the remaining run time is $x-\alpha(n-1)$ and the run time in this case would be:

$$\begin{aligned} t_{l_2} &= \alpha n+x-\alpha(n-1)+\alpha n \\ &= \alpha n+x+\alpha \end{aligned} \tag{14}$$

However, comparing the efficiency of (13) and (14), using (1) and (2), recalling that n≥2 we prove in (15) that the former method is more efficient than the latter and hence is more suitable.

$$
\begin{array}{lcl}
 & \textit{Efficiency}_1 & \textit{Efficiency}_2 \\
\Rightarrow & \dfrac{x+2\alpha(n-1)}{xn\cdot n} & \dfrac{\alpha n+x+\alpha}{xn(n+1)} \\
\Rightarrow & xn+x+2\alpha n^2-2\alpha & n^2\alpha+xn+\alpha n \\
\Rightarrow & \alpha n^2+x-2\alpha & \alpha n \\
\Rightarrow & n^2+\dfrac{x}{\alpha}-2 & n \\
\textit{but} & n\geq 2 \quad \wedge & \dfrac{x}{\alpha}>0 \\
\Rightarrow & n^2+\dfrac{x}{\alpha}-2 & \geq \quad n \\
\Rightarrow & \textit{Efficiency}_1 & \geq \textit{Efficiency}_2
\end{array}
\qquad (15)
$$

Now assume a worst case scenario where the processors are connected in a long line, with the processor coordinating all the results and data at one end of the line. With processors P_1 to P_n, assume that P_1 is the controller and that it passes data first to P_n, then to P_{n-1}, ... then to P_2 before doing its own work. All data to P_n has to travel through P_2. P_2 then has to pass the data to P_3 before it can accept the data for P_{n-1}. This results in a sending time of $2\alpha(n-2)+\alpha$ μsecs. The computation time on P_1 will be x, and once complete, it will have the same receiving time to obtain the results as the sending time. The receive is performed in reverse order to the send. The worst case then becomes:

$$
\begin{aligned}
t_1 &= 2\alpha(n-2)+\alpha + x + 2\alpha(n-2)+\alpha \\
&= 4\alpha n-6\alpha+x
\end{aligned}
\qquad (16)
$$

We wish times such that the parallel run time is less than or equal to the serial run time. Thus (12) and the more suitable (13) give us:

$$\begin{aligned} & 2\alpha(n-1)+x \leq xn \\ \rightarrow \quad & x(1-n) \leq 2\alpha(1-n) \\ \textit{but} \quad & n \geq 2 \\ \rightarrow \quad & x \geq 2\alpha \end{aligned} \tag{17}$$

Hence, when more than 2α instructions are performed per data item, parallelisation of the code will result in a speedup. If 2α instructions are performed, the run time is the same as the sequential version. Now for the worst case, (12) and (16) gives:

$$\begin{aligned} & 4\alpha n-6\alpha+x \leq xn \\ \rightarrow \quad & x(1-n) \leq 2\alpha(3-2n) \\ \textit{but} \quad & n \geq 2 \\ \Rightarrow \quad & x \geq \frac{2\alpha(2n-3)}{n-1} \end{aligned} \tag{18}$$

Figure 18 is a graphic illustration of the break-even points where the area below the lines signifies where it is not worthwhile to parallelise for the respective case since no speedup will be obtained. The value of α used was 0.3814697 μsecs, the time to transmit 1 byte, with the code length being the number of 0.1 μsec instructions to be performed on the byte with Data Volume being the number of bytes to be operated upon, producing a byte result. The lines for all other possible configurations will fall between the two lines. The above formulas may be altered to fit the configuration and data transmission times for the input and output with the code and/or data size varied in order to establish the break-even division line. If the data and code sizes are known, then the speedup formula (1) may be applied in order to establish the actual benefits which will be obtained.

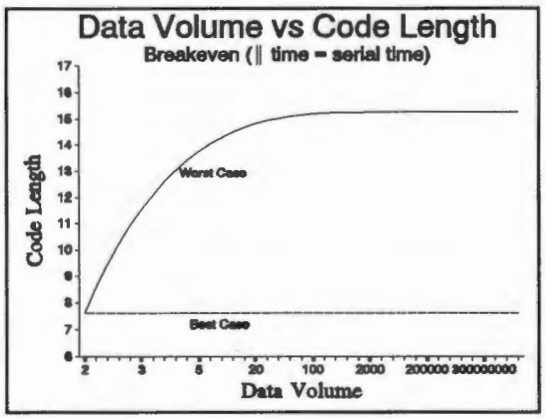

Figure 18

Once the paralleliser has established whether parallelisation of the code is actually feasible and whether suitable benefits will be gained from parallelising the code, they may continue with the next step and commence the parallelisation.

6.2 3L Flood-Fill Farm

The most important processes in a farm network are the master and the workers. The workers are all identical processes which are placed onto different processors. 3L provides a basic farm structure where the paralleliser generates the code for the master and the worker and applies the flood-fill configurer to a generate a boot file for any transputer network. This boot file will place the master and a worker on the first (host) transputer which is connected to the host PC. This is because the master normally handles any I/O requirements of the network and therefore necessitates a direct link to any I/O devices, in our case, a PC.

Network communication is handled internally by flood-fill communication libraries. The master has a generic send which sends a message or group of messages to any free worker. This message is taken to be the work packet and the worker is then marked as busy until the master receives message packets from that worker. Similarly, the worker has a generic send which is used to convey messages to the master. These message packets are assumed to be the results from the worker which is then sent the next work packet from the master. The task of the master in the 3L flood-fill network is thus to repeatedly call the library send routines with the work packets in order to distribute work to the workers, while simultaneously receiving and collating results from the workers. The workers therefore receive the work packet, perform the necessary work, and return the result packet. All the result packets are directed by the communication library to the master, with the communication library similarly distributing message packets from the master to free workers. There is therefore no method through which messages may be addressed to discrete workers, and intermediate and broadcasted messages are not possible due to the indeterminate

message addressing scheme. Intermediate and broadcasted messages are also not possible as each message group sent from the master is considered a work packet and messages sent by the workers to the master are considered the workers' results, marking the worker as being available to perform further work.

However, when performing several tests to establish the performance of the 3L flood-fill networks, I discovered that the expected performance increase when additional transputers were added was not met. In fact, the results indicated that the added transputers were not being utilised and after further investigation and debugging I determined that some worker transputers were either not being booted, or were never given any work. Tests on different network configurations revealed different results with the problem being eliminated on certain configurations. It was then that I wrote a small test network in order to ascertain the number of workers being utilised. The test confirmed the indications that not all the transputers available would be employed as workers. The test and results were then faxed to 3L in Livingston, Britain, who later acknowledged the problem as a simultaneous transputer booting problem. This occurred when, while the network was being flooded with worker processes, two or more transputers would simultaneously attempt to boot a common transputer to which all are adjacent, resulting in the common transputer never being booted and possibly one or more of the booting transputers to be also effected [Mac89]. A summary of the document sent to 3L, which also includes the method to determine the number of transputers utilised, is listed separately from this thesis in a thesis supplement. A copy of this document may be obtained by writing to the author directly.

This form of farm structure is therefore limited in its use. It lacks broadcasting facilities, individual worker addressing schemes as well as intermediate or exception message passing between a specific worker and the master. For example, a worker may attain a stage where a certain limit has been exceeded and requires user input whether it is to continue or abort. Using the 3L farm, the worker cannot pass this information back to the master unless it was sent with the results. In turn, the master could not signal the worker to continue or abort, unless it was sent with a work packet.

The solution to this in 3L is for the worker to give up its work and return all its values calculated, for the possibility that the user wishes the job to be continued. If the users wishes the job aborted, the data can merely be discarded, however, if he wishes the job to be completed regardless, a special work packet containing the partially calculated data is passed as a work packet to the next available worker to complete the work. An alternative would be for the master to simply re-transmit the work, but include with the work the user information on how to cater for the exception when it is again reached.

In the above example, the first solution requires workers to be able to receive several different types of work packets, as well as network communication being unnecessarily increased, while the second solution results in unnecessary re-calculation. Both are undesirable and as they introduce an unnecessary complexity into the parallelisation process, while also increasing network communication. Furthermore, the flood-fill problem would have to be alleviated. It would be far simpler to use a farm network which could incorporate such a scheme.

6.3 SAPF Fortran Farm Structure

In order to overcome the problems foreseen in Section 6.2, a flood fill farm was designed to incorporate the broadcasting of messages, individual worker addressing, as well as intermediate messages. Messages from the master to the worker are received in the same order which they are sent. Workers may reserve the network in order to transmit a series of messages to the master, suppressing messages reaching the master from other workers in order that its messages may arrive consecutively. Library routines are provided in order to allow the master to claim and release workers as they are sent work and return results respectively.

These routines are written in fortran in order to retain consistency, uniformity and portability. The routines are listed in Appendix A with a description of their function and operation, with the source listings provided in the thesis supplement.

The workers are identified by unique consecutive 32 bit integer identification numbers which lie in the range from 1 onwards. Some of the master library routines include a parameter, being the worker's identification number (ID), which indicate the worker to which the message is to be sent, or the worker from where the message was originated.

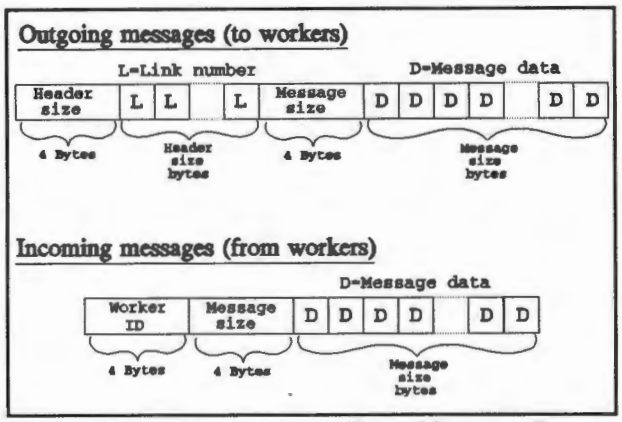

Figure 19 Farm Message Format

Messages are transmitted in the format indicated in Figure 19 where the left-most data items (header size, worker ID) are transmitted first with the right-most (message data) last. The outgoing messages are prefixed by the link header, while the incoming messages require no such prefix as their destination may only be the master. Instead, the incoming messages are prefixed by ID of the worker from where the message originated. If the ID is negative, no message size and message data will be received as this is taken to be a request from the worker number given by the absolute value of ID to reserve the network. In other words, all future data is to be explicitly received by the master from worker *ID* until worker *ID* releases the network. This release is achieved by worker *ID* transmitting (-ID) once again and is also not followed by any message.

The header for outgoing messages comprises of a header length followed by the corresponding number of bytes, each representing the channel number where the message is to be forwarded. As the message progresses through each intermediate transputer, the intermediate transputer removes the link number from the front of the header and forwards this new header with the message down the removed link number. For the case where outgoing messages have a header length of zero bytes, the message

length and data are alone transmitted to the current worker (worker residing on the same transputer as the message router) as this implies that the message has arrived at its destination. A message with header length less than zero implies a broadcasted message. In this case, no header bytes are transmitted (actual length is zero) and each worker transputer gives a copy of the message to the current worker (worker on the same chip) and forwards the message, along with the broadcast header, down each outgoing link to the remaining workers of that branch of the network.

The size of the actual message is limited to the number of bytes given by the library integer constant F77_NETW_MAX_PACKET_LEN and is currently 1K bytes. If a message whose with size is greater than F77_NETW_MAX_PACKET_LEN bytes is to be sent, it must be divided into message segments of size less than or equal to F77_NETW_MAX_PACKET_LEN bytes and each segment sent separately as a message. The library routines initially catered for message packets of any size with the routines breaking down the message into separate segments at the sender, and re-assembling the packets at the destination. The reason this option was excluded was to keep communication time to a minimum since it was found that in over 90% of messages sent, the size of the message was below 1K. Before transmission, each routine would have to ensure the packet would fit within the buffer assigned for the relaying of messages, and thus in over 90% of data transmissions, this checking would be redundant. Therefore, the division of these large data packets, a simple enough task, was left to SAPF and the implementer of the network who would have previous knowledge of when such partitioning is required. Large message packets from a worker to the master could be split into segments and each segment sent separately. This is similar to the technique used by 3L in their farm networks.

In order to prevent the master from receiving a different packet from a second worker while the receiving the split message from the first worker, a network locking facility is provided for incoming messages whereby the first worker may ensure that the master only can receive messages which originated from it. Once all the segments comprising the message have been transmitted, the worker may unlock the communication network

allowing other workers to transmit once again to the master. Since inter-worker communication is not permitted and packets sent by the master arrive at the destination in the order which they are transmitted, a locking facility for outgoing messages is not necessary.

6.3.1 SAPF Farm Construction

The SAPF farm is constructed almost automatically. The master and worker tasks are constructed and compiled separately, with executable code created for each. The exploratory worm (see Section 8) is executed on the target network, with the results being filed. This worm is used to determine the total number of transputers as well as the link connections between each. The pascal program MAKECONF, described in Section 10.2.9 and listed in the thesis supplement, is then used to create a 3L compatible fixed configuration file using the worm output and 3L flood-fill configuration file. The configurer is then used to create a fixed network configuration boot file which will be used to boot up the current network. This step was necessary in order to overcome the problem of the 3L flood-fill configurer. Thus, no recompilation is necessary when creating bootable network code for dissimilar networks, only reconfiguration. In summary, the worm determines the actual network configuration, MAKECONF creates a 3L fixed configuration input file from the worm output and 3L flood-fill configuration file which normally is used in creating a flood-fill network, and finally the 3L configurer is used to actually create the boot code for the network.

The fixed configuration file created by MAKECONF creates a spanning tree of the network from the master, with each transputer hardware link mapped to an outgoing and incoming channel pair. At the root of the tree resides the master, with the worker nodes residing on the branches and leaves of the tree. The path from the master to each of the workers is also ensured to be a shortest path (smallest number of intermediary processors between the master and each worker pair). Worker nodes with

the same number of intermediary processors in the shortest path between it and the master processor are considered to be of the same generation. The links connecting worker nodes of the same generation are considered redundant and are therefore omitted from the fixed configuration file generated by MAKECONF.

Once the network has been booted, the SAPF flood-fill library initialises the network by identifying each worker, the path to each worker and the total number of workers available. This is accomplished through the master which repeatedly sends an unused ID, initially one, down each outgoing port of the master to the workers. (3L provided ports as channels between processes which are linked together in the 3L configurer.) After sending the initial ID ($ID_{initial}$), the master waits on the corresponding incoming port for the result ID (ID_{result}). If ID_{result} is valid (greater than 0), the master will receive a portion of the header path for the worker whose ID is $ID_{initial}$. The master completes the header for this worker by prefixing the outgoing port number of the port down which the ID $ID_{initial}$ was sent. $ID_{initial}$ is then set to ID_{result} which is again dispatched down the same port with the master repeating the process. If an invalid ID_{result} is received, the master moves to the next outgoing port to retransmit $ID_{initial}$ until all the incoming ports have returned an invalid ID_{result}. When the latter occurs, all the workers have been identified (MAKECONF ensures a connected network) with the path to each worker set. The master library then returns control to the actual master process in order for the real work to begin.

As soon as each worker is booted and the worker library initialisation routine is called, the initialisation procedure of network worker communication begins simultaneously with the master network initialisation. Each worker waits on input port 0, which MAKECONF has ensured is the port pair directed to and from the master, for its ID, ID_{mine}. The remaining port pairs are piloted by MAKECONF to workers of the next generation. ID_{result} is then created by incrementing ID_{mine} and is transmitted with a null path down output port 0 to the master. After a worker (say A) has identified itself, it will continue to expect IDs from the master which it forwards in turn down each of its outgoing ports, receiving result IDs and paths. These unidentified workers nodes

(call them *B*) operate in identically the same manner by returning its ID_{result} with a null path. To the path *A* receives from each worker *B*, *A* prefixes the outgoing port number used to address *B* to the path received from *B* and relays ID_{result} and the prefixed path to the master. Using this technique the path to each worker is built up.

When no outgoing ports are found, or all workers along an outgoing port have been identified, an invalid return ID of (-1) is used by *B* to signify *A* that no more unidentified workers exist along the port *A* used to address *B*. *A* then begins forwarding $ID_{initial}$ along the next outgoing port to the next group of *B*. Once all the workers on outgoing ports of *A* have been identified, or no outgoing ports exist, *A* will return an ID_{result} of (-1) to the master. After all workers of following generations on the same branch have been identified, the initialisation stage of the network identification for the worker is complete and the worker may commence with its actual task.

The above-mentioned method enables each worker to be identified, allows the master to determine the path set up by MAKECONF to each worker, and hence establishes the number of workers available.

6.3.2 Master Process Routines

These routines are listed in Appendix A and are called by the programmer of a SAPF farm. The master routines can only be called exclusively from within the master process of a SAPF farm. These routines are used for communication from the master process to the worker processes, and allow the following:

- The ascertainment of the number of worker processes.
- The sending of messages by the master process to individual workers processes.
- The broadcasting of messages to all the worker processes by the master process.
- The receipt of messages by the master process from individual worker processes.

- The claiming of a worker which is marked as being idle in order to be able to distribute work to it.
- The freeing of a worker which is marked as being busy in order for the work distribution sub-process of the master to be able to claim it again. This is performed normally after results have been received by the result-receiving sub-process of the master process.
- The ascertainment of the worker process which has "frozen" the network in order to be able to exclusively communicate to the master process.

6.3.3 Worker Process Routines

These routines are listed in Appendix A and are called by the programmer of a SAPF farm. The worker routines can only be called exclusively from within worker processes of a SAPF farm and exist on each worker processor. These routines are used for communication by each worker to the master, sending and receiving messages to and from the master process. Included is a routine pair which is used to claim exclusive communication to the master process in order to be able to send a sequence of messages which are to arrive concurrently after each other at the master, and to free the network for normal communication once the exclusive communication has ceased.

6.3.4 Worker Processor Routing Processes

The operation of the two message routing processes described in Section 5.5 and illustrated in Figure 15 on page 39 is discussed in this section. These two routines are transparent to the programmer of a SAPF farm, as the programmer does not even need to know of the existence of these routines, and must never in fact call them as they are used exclusively by the SAPF farm communication system. The first routing process handles the outgoing messages, that is, messages from the master to the workers, while the second handles the incoming messages to the master from the

workers. As illustrated in Figure 15, this pair of routing processes resides on each worker processor, and their sole function is to handle the network communication. These processes are introduced by the F77_NETW_WORKER initialisation routine into the network and remain operational even once the workers have completed their tasks and the master has no more work to distribute. Their operation is described below.

6.3.4.1 SUBROUTINE F77_NETW_OUT_MESSAGES(Dummy)

After establishing the address of the input port used to receive message packets from the master, this process thread enters an infinite loop where it distributes message packets sent from the master to the workers on this branch of the network.

In the loop of distributing messages, the thread expects the path header length first, followed by the path header if the header path length is greater than zero. The message length, followed by the message are then received, with the message being stored in the thread's own internal 1K buffer. The thread then establishes the address of the destination port where the actual message is to be sent using the following tests:

a) If the path header is empty (zero length), the destination port address is set to the address of the internal port *to* the worker. The worker receives this message by calling the procedure F77_NETW_RECEIVE which merely reads the message length and message from this internal port.

b) If the path header exists (is greater than zero length), the destination port address is set to the address of the outgoing port whose number appears as the first item in the path header. This first item is then removed from the header and the adjusted header size with the remainder of the header transmitted along the previously determined port.

c) If the path header length is invalid (less than zero), this message must have been broadcasted from the master. In this case, the broadcast header, message length and message are sent down each of the external outgoing ports to the remaining workers of this branch. The destination port address is then set to

the address of the internal port *to* the worker, allowing the worker to also receive a copy of the message.

Once the destination port address has been set by either (a), (b) or (c) above, the message length, followed by the message is then transmitted along that port. This is the base of the loop and the whole process is then repeated infinitely.

6.3.4.2 SUBROUTINE F77_NETW_IN_MESSAGES(Dummy)

The initialisation stage of this thread consists of setting up an array containing, in the first position, the address of the internal port from which messages from the internal worker may be received, and containing in the remaining positions the addresses of the incoming ports from the workers of the following generations. This second internal port is used by the subroutine F77_NETW_SEND to allow the correlating worker to send messages to the master. The messages received from all these ports must be passed on to the master and are therefore incoming messages.

After initialisation, the thread enters into an infinite loop of forwarding messages to the master. In this loop the thread first waits for a worker ID which it inputs from any of these incoming ports. If the ID is valid (positive) the message length and message are both input from the same port with the message placed into an internal buffer. The worker ID, message length and message received constitute an incoming message packet which is then forwarded down the output port to the master.

If the ID is invalid (negative), this is taken to be a request by the worker to reserve network communication to the master. The ID received is forwarded to the master with the thread then entering a loop where it waits only for message traffic originating from the port where the reserve network request was received. The communication

expected by this thread is a second worker ID[1]. Once the second ID has been received it is forwarded immediately to the master. If this second ID is again negative, it is accepted to be a communication release command from the worker which reserved the network and the thread returns to the state of monitoring all the incoming ports. If the second ID is positive, the message length and message are then received in turn from the reserving input port and forwarded to the master, with the thread loop returning to the state of waiting for the second ID.

In this manner, during normal communication, messages are forwarded to the master from the workers. When a worker reserves the network, the request is forwarded to the master and the F77_NETW_IN_MESSAGES router task of that worker ignores any communication from other F77_NETW_IN_MESSAGES routers of following generations. The communication from following generations will continually be ignored until the reserving worker has released the communication network. As the reserve request winds its way to the master, it in turn prevents previous generation workers from transmitting data to their F77_NETW_IN_MESSAGES routing task until the network is released. When the master finally receives the reserve request, it in turn will also only read messages from the reserving port, resulting in messages from any other worker being blocked out.

Note that if requests by other workers are made to reserve the network when it is already reserved, or if several workers simultaneously make requests to reserve the network, these requests will merely be blocked out by the F77_NETW_IN_MESSAGES tasks of previous generation workers as only one request is allowed to pass to the master. The network will remain locked until it is freed by the reserving worker which is the only process permitted to liberate the network. Once released, outstanding reserve network requests will be passed to the master with the first request arriving again blocking out remaining requests.

---

[1]While the communication network is reserved by a worker, the absolute value of the second ID should be the same as the absolute value of the first ID. That is, any communication should originate from the same worker reserving the communications network.

7 OCCAM/TURBO PASCAL SERVER

Most transputer systems today are not stand-alone systems but add-ons to existing systems. These transputer systems are therefore dependant on their host computer for I/O requests and other services. In order for these systems to be able to utilise the services provided by the host, an interface is required between the transputer system and host computer. This type of interface is often known as a *Server*, and consists of two components. The first component, or *Host Server*, is executed on the host computer and translates the protocol received from the transputer system as I/O or other service requests, which it in turn carries out. The second component, or *Transputer Server*, is a set of library routines, providing an interface into the language of the transputer code where calls to these routines will furnish the desired I/O or other service requests. In a network of transputers, often only one transputer is connected to the host computer, and this transputer is labelled the root. This root transputer is therefore often responsible for providing the I/O and other services to the remainder of the network. These two components therefore execute simultaneously on the host computer and root transputer respectively, and communicate with each other in order to achieve the desired results.

The reason for the root transputer often being selected to provide these services is that it can communicate directly to the host and either directly or indirectly with the remainder of the transputer network. If another transputer in the network were to be selected, their communication would have to be routed through the root in any event, resulting in an increase in overheads. Furthermore, if more than one transputer were selected to handle I/O requests, communication paths between these transputers and the host will have to be established. This may also result in more complex I/O services due to the indeterminism of parallelism when performing I/O. For example, suppose two or more transputers require user input from the keyboard. The user, being a sequential device, will not be able to distinguish what input is actually required or where the input is destined to, unless the transputers requesting the input were to include a prompt defining the type of input required. The host also is a sequential

machine so it is standard practice to have the root transputer control all the I/O requirements from other transputers in the network. In fact, all known non-operating system products allow I/O to be performed only on the root transputer. This is highly suited to the farm structure where the root transputer is the master with the remaining transputers being the workers.

The language OCCAM 2 does not include any I/O facilities, therefore inciting a whole set of different I/O servers for the transputer under OCCAM 2. Unfortunately, all of these libraries vary greatly in standard, as well as only providing basic screen, keyboard and file I/O between the host computer and root transputer. The host system used for this thesis was an IBM AT PC and therefore may provide the standard PC utilities to the root transputer. Since the Airflow Model of Section 9 required graphical output device, a graphics output device would have to be used. Such a device for transputers is available, but is costly. Also no server at the time of the original development of Section 9 existed. Therefore, it was decided to create a server whereby the services of the PC may be utilised to a greater extent, including graphic routines not specific to the expensive transputer graphic interface but to the host PC itself. Therefore, this server was based around Turbo Pascal Version 5.5[2], providing calls in OCCAM 2 similar to the library routine calls of Turbo Pascal Version 5.5. Naturally, the *Host Server* was written in Turbo Pascal, while the *Transputer Server* was written in OCCAM 2. This OCCAM/Turbo Pascal server is discussed in the Appendix B.

7.1 The *Host Server*

The *Host Server*, listed in the thesis supplement, provides merely the I/O services of the PC to the root transputer, as well as a few extra system calls and utilities. It is used to boot the root transputer, and possibly connected network, in the same manner as the INMOS AFServer and IServer. It may be passed two file name parameters, the

[2](C) BORLAND SOFTWARE 1989

first of which is used to boot the root transputer with an optional second parameter to boot up a sub-network. Other parameters which may be passed include an option to override the base port address for link 0 of the root transputer, to place a series of keystrokes automatically in the keyboard buffer after booting the root, specifying the path where previous system files may be found as well as allowing the <BREAK> key to be used.

7.1.1 Reset Mode

All transputers in RESET mode not booting from Rom but from their links continually poll all their links for activity. The activity is a message, sent to the transputer, which provides three possible actions according to the first byte received on a link:

0	A poke command followed by the address word and data word. The data word is stored in the address indexed by the address word. The word size is dependant on the word size of the transputer (16 bit for T212 and 32 bit for T414, T800).
1	A peek command followed by the address word, with the data word (contents of the word address) being received on the corresponding link.
2-255	The boot message length, followed by that number of bytes which are to be loaded at MEMSTART. This code, often the pre-loader of the transputer boot code, is then executed from MEMSTART. The pre-loader is often used to load a more complex loader high up in memory. This allows the lower sections of memory to be fully utilised by the actual target code, loaded over the pre-loader, making good utilisation of fast on-chip ram.

7.1.2 Booting the Root Transputer

In order to reset a transputer, both the analyse and reset pins must be held to ground (de-asserted) for a minimum period of 5 milliseconds, followed by reset being held to VCC (asserted) for a further 5 milliseconds before finally being de-asserted. This pulse will place a transputer which has its BootFromROM pin held low in its reset state. This will have the effect described in Section 7.1.1. These pins are made available through the I/O ports of the PC. Thus zero is output on the ports connected to the reset and analyse pins, followed by a minimum delay of 5 milliseconds, with reset then being asserted by outputting a non-zero value on the reset port and holding this signal high for a further minimum of 5 milliseconds in order to allow the signal to stabilise before reset is finally de-asserted, placing the root transputer in a reset state.

Next the boot file is transmitted down the link connected to the root transputer. This is again achieved through the use of two ports which have been connected to an input link on the root transputer. One port indicates the status of the output port down the link, with the other acting as a single byte data port used to carry transmit the byte to the root transputer. Thus, in order for each byte to be transmitted, the status is first examined to determine whether transmission is possible (normally whether the transputer has read the previous byte from the link yet) and once clear, the byte to be transmitted is placed in the output data port. The boot file is thus transmitted byte for byte to the root transputer.

The boot file is often in the format illustrated in Figure 20. The preamble, loaded at MEMSTART, is used to determine the memory size, transputer type etc. and loads the main network loader at the top of memory and finally transferring control to the main network loader. The main loader then loads the code for the root transputer over the preamble code, in order that the actual code may make efficient use of the on-chip RAM. When the network loader has completed its task,

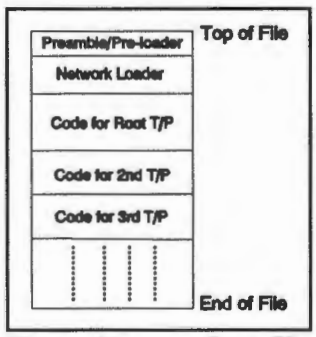

Figure 20 Boot File

it jumps to the actual code to be executed, leaving itself in the vector space where it may be overwritten by program variables etc. These three stages of the booting of a transputer are depicted in Figure 21, with the final memory usage illustrated in stage 3. Before finally handing control to the actual code, the network loader may then distribute code for the other transputers in the network along the remaining links. Each of these codes in turn may also be in the same format as the original boot file, containing preamble and network loader code etc.

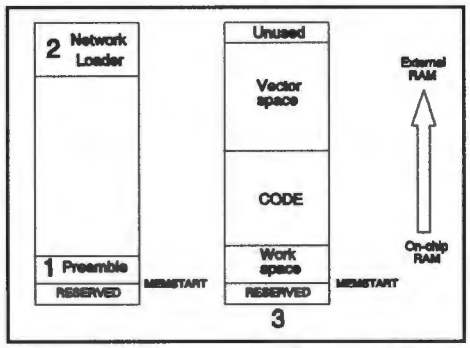

Figure 21 Memory Layout

7.1.3 Operation

Once the *Host Server* has transmitted the boot file, it will wait for input from the output link of the root transputer which is connected to the host PC. This is achieved in the same manner as the transmission of a byte to the root transputer. The host continually polls the input status port to determine whether the host has transmitted a byte. Once available, the byte may be read off a second data input port. This byte is then interpreted as an instruction which is to be performed by the *Host Server*. Once the instruction has been performed, the *Host Server* repeats the process of waiting for instructions until it gets the terminate command, in which case it obviously terminates.

While waiting for a command from the root transputer, known as the monitoring state, the *Host Server* also polls the keyboard and places any keypresses in an buffer, internal to the server, in order that it may at least also do something useful while waiting. This also allows the user to break out of the server and terminate abnormally should they suspect that there is a problem. As an additional feature, the user may exit to dos while still preserving the state of the transputer, and restore the server at a later time to continue its operations. This allows the user to give the transputer a massive task, assuming that the I/O occurs only at the beginning and end of the task, exit to dos

thereby freeing their PC and allowing them to continue with other work, and finally restore the server at a later stage to complete the I/O operating and thereby obtain the results once the task has completed.

This is achieved by saving the screen as well as the file, system and other variables and status' to a temporary file on disk, and then terminating the *Host Server*. When the *Host Server* is restored, the parameter **/*** is used to notify the server that the transputer system is already running and that the screen, file, system and other variables must be restored from the file on disk and that the *Host Server* must return immediately to the monitoring state without attempting to reset or boot the root transputer and/or connected network. The root transputer's communication to the host is frozen when the *Host Server* terminates, allowing the root transputer in turn to establish a system whereby the requests are queued when no response is received from the *Host Server*. When the *Host Server* is restored the queue may then be serviced and the system returned to its original form. This system has successfully been introduced in the Airflow Model discussed in Section 9.

7.2 The *Transputer Server*

The *Transputer Server* is a set of libraries which communicate to the *Host Server* through the channel connected to the host PC in order to perform mainly I/O and some other functions. A protocol has therefore been established between the two servers whereby calls to routines within these libraries will result in the transmission of the relative byte command to the *Host Server*, followed by the relevant data (where applicable) and the receipt of data from the host PC (if also applicable). The OCCAM source of the libraries are listed in the thesis supplement.

The library calls have been deliberately made similar to their actual Turbo Pascal calls in order to enable ease of understanding and use. The only difference is that the first two parameters which must be passed to all of these OCCAM procedures are the

hardware input and output channel from and to the host PC respectively. This gives the procedures the communication path to the host PC, allowing them to initialise the protocol required to achieve their purpose. Finally, no function calls exist within these libraries as the definition of OCCAM 2 does not allow functions to reference channels and global variables. Functions are to be seen as a single entity with no external references. Therefore, they also do not use the PAR construct of OCCAM 2.

7.3 Summary

The OCCAM/Turbo Pascal Server achieved its objective successfully, providing most of the libraries available to programmers of Turbo Pascal V5.5. Furthermore, these libraries could easily be extended by merely including the code to perform the desired I/O operation in the *Host Server*, and the call to activate the operation in the *Transputer Server*.

8 TRANSPUTER WORM

The transputer worm is exploratory, self-replicating code used to determine the configuration of a transputer network. It boots up each connected transputer with a "rubber stamp" copy of itself, making it an ideal generic network loader for farm type structures on any configuration of transputers. Two types of worms were designed to perform the network booting and exploration, differing only by the search method which each employs. The second method was only introduced later in order to reduce the network's boot time, that is, the time taken for all transputers in the network to become active and executing the necessary code.

In order to perform the exploration, the worms had to utilise special techniques in order to test links as well as determine transputer types and memory sizes. These techniques are first described in Sections 8.1, 8.2 and 8.3 respectively before the depth and breadth first worms are documented in Sections 8.4 and 8.5.

8.1 Link Identification

The following methodology, illustrated in Figure 22 by a decision tree, was applied by an exploring transputer, named the *enquirer*, in order to determine whether a transputer device is connected to a link of the *enquirer*:

1) Attempt a 16 bit word peek by trying to transmit the peek byte 1, followed by a 16 bit word. If the attempted transmission fails, there is no device at the other end of the *enquirer's* link, and the link examination terminates.
2) If the transmission succeeded, the corresponding input link is examined for a possible data return. If data is returned, a 16 bit transputer (probably a T212) exists at the other end of the link. The link examination terminates if this is the case as T212 and other 16 bit transputers are not an issue here. The worm may easily be extended to incorporate these transputers.

3) If no data return is received, and the transmission succeeded then the possibility exists that a 32 bit transputer exists on the other end of the link. The *enquirer* completed a peek command for a 32 bit address by transmitting the remaining 16 bits of the address to be peeked. If the transmission of these latter two bytes is successful, and a 32 bit data item is received on the link, the peek command was successful and a 32 bit transputer indeed exists at the other end of the link. If the transmission was unsuccessful, or no data was returned, the link is identified as being disconnected.

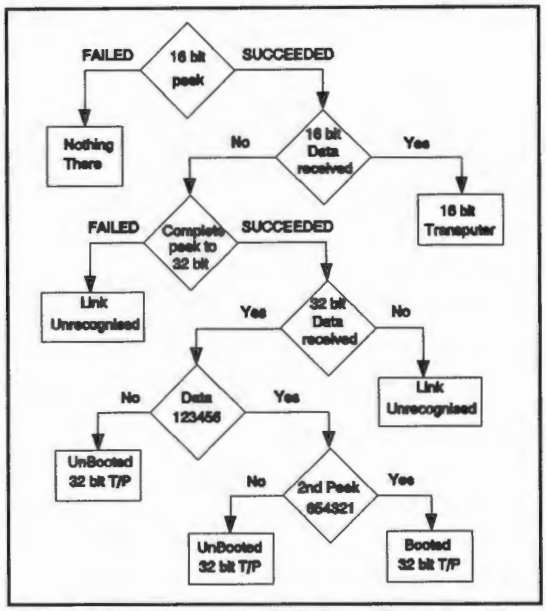

Figure 22 Link Test Decision Tree

4) If a 32 bit transputer was detected by the *enquirer*, the data returned is examined to see it the transputer is not already booted, or if it was already booted and running worm code. This is established by having booted transputers emulate the peek command by returning a predetermined value (123456) as the result of the peek to the *enquirer*. If the *enquirer* does not receive this value the 32 bit transputer is assumed to be unbooted, residing in a reset state. If this first special value is received the *enquirer* must ensure that the value received was not merely coincidence, confirming that the transputer already is booted. This is achieved by performing a second peek command of the same address. When this occurs, the peek emulator returns a second but different predetermined value (654321). If this second value is not received, it merely was coincidence that the address peeked contained the same predetermined value and the transputer is not booted, otherwise the transputer must be booted and executing the worm code.

8.2 Determining Memory Size

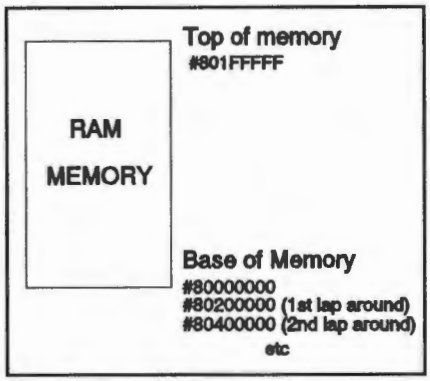

Figure 23 Memory Lap-Around

In order to realise the algorithm, an explanation of the addressing system of the transputer is necessary.

The 32 bit transputer has a 32 bit memory address, allowing up to 4 gigabytes to be accessed. However, only word addressing is permitted. With a word length of 32 bits (4 bytes), the lowest two address lines (0 and 1) are ignored. Furthermore, the upper two address lines (30 and 31) are reserved for external memory mapped devices or ROM, thus we may also ignore the upper two address lines. The start of memory is #80000000 (bit 31 set), which is in on-chip RAM, and works its way upwards. When accessing RAM, only the lower address lines required are mapped onto the actual RAM, with the upper unused address lines being ignored. For example, a 2 Mb memory will have the following 32 bit address space (in binary) where the x lines are ignored:

11xxxxxxxxx11111111111111111xx

Since the x lines are ignored, the following 32 bit address will access the same memory location: (y <> x)

11yyyyyyyyy11111111111111111yy

Thus, the base of memory may be addressed as #80000000, #80200000, #80400000, ... (Hex) as illustrated in Figure 23. The algorithm used uses this feature to search for the memory size, operating as follows:

1) Starting at address *Addr* = #20000000 (upper two address lines ignored), poke the address value (*Addr*) into the address.
2) Bit shift *Addr* right by one bit, and poke the new value of *Addr* into the new address *Addr*.
3) Peek the contents of the old *Addr* and compare the value with the new *Addr*. If the contents are the same a memory lap-around occurred into the base of

memory and the search must continue by going back to step (2). If the values differ, then the new *Addr* accesses the middle of available RAM (RAM must always be a factor of two) as no lap-around occurred. Hence, the old *Addr* is the actual memory size.

8.3 Transputer Typing

Although a T800 can run most of the T414 instruction set, it incorporates its own instructions to reference the on-chip floating-point processor. Thus, the worm must distinguish between the T414 and T800 in order to run the correct code on the appropriate type of transputer. Since the T800 has 4K of on-chip ram, while the T414 (excluding the T424) has 2K, and as on-chip ram is inherently faster than accessing off-chip memory, the transputer type is determined using timing tests in three sections of memory. These are a) known on-chip ram, b) known off-chip ram and c) possible off-chip ram which are located at MEMSTART, MEMSTART+4K and MEMSTART+2K respectively. The time taken to read from values at these three different locations is determined, with the time of the possible off-chip accesses (c) compared to the time of the known on-chip (a) and off-chip (b) locations. If the time in (c) is comparable to (a), then the upper 2K must be on-chip and the processor is a T800, otherwise (c) must be off-chip and the processor is a T414.

This method will not work for a T424 which has 4k of on-chip ram. It does, however, have only two hardware links, distinguishing it from the T800 (excluding the floating point processor), allowing software checking to be performed on the number of links if 4k of on-chip ram is found. Thus the difference between the T424 and T800 may be also determined.

8.4 Depth First Worm

The first method employs a depth first search. The host PC boots up the root transputer (transputer directly connected to the host PC) with the I/O handler and main control system of the worm. This is the base of the worm and has been named the *Tail*, while the actual "rubber stamp" copy which performs the majority of searching, link identification and transputer booting has been named the *Head*. The OCCAM 2 source code for both *Head* and *Tail* is given in the thesis supplement.

Each transputer in the network is identified by a unique identification tag *ID*. *ID* is an eight bit integer which increments for each transputer in the network in the order which they were discovered. That is, the first transputer discovered will have *ID=1*, the second will have *ID=2*, the third *ID=3* etc. for all the transputers in the network.

The purpose of this depth first search is to obtain an identification table for each transputer in the network. The table consists of four entries, one representative of each link on the transputer. Each entry contains the *ID* of the transputer to which the link is connected, as well as the link number of the transputer to which it is connected. Thus each entry contains the link number and *ID* of the transputer to which it connects.

8.4.1 The Head

Head operates according to the method described below. The source listing is provided in the thesis supplement:

1) Once operational on a transputer node, directly after being booted, all links are monitored for the identification tag *ID* from the parent transputer. The parent transputer is the transputer node which booted this current node.
2) The link through which *ID* was received is classified as the parent/booting link and identification information (such as the parent's *ID*, the link number of the

parent which is connected to the parent link, the node's own transputer type and memory size, etc.) exchanged between the parent and child transputers. All other links are marked as being unidentified.

3) Set the token variable *CID* (current available ID), being the next available identification byte, to *ID*+1. Each "rubber stamp" copy of *Head* may only actively explore the network if it has possession of the *CID* token. Only one token exists and therefore only one copy of *Head* may be active at any point in time.

4) Select an unidentified link and perform the sub-steps below, otherwise if there are no unidentified links, skip to step (5).

   a) Determine whether a transputer is connected at the other end of the link.

   b) If a transputer does not exist identify the link as being disconnected and return to the beginning of (4). If one does exist, proceed to (4c).

   c) Determine whether the transputer has already been booted. If so proceed to step (4d), otherwise skip to step (4e).

   d) Swap identification information, such as *ID*s and link numbers, with the already booted transputer, thereby identifying which transputer is connected at the other end of the link, and what link number of that transputer this link is connected to. Thus the link becomes identified and *Head* returns to the beginning of (4).

   e) The transputer type and memory size of the unbooted transputer must be determined, and a request passed to the parent transputer for the "rubber stamp" copy of *Head*. This copy is used to boot the unbooted transputer.

   f) Once booted, the *CID* token is given to the newly booted transputer in order to allow it to identify itself. Link information is then exchanged between the parent and newborn child. The link is thus identified and the parent then passes control to its newborn child and enters a monitoring state described in (g). This relinquishing of control by the new parent is to allow the newborn child to explore the network further,

before attempting to continue its own exploration. Hence the depth first search method.

g) All unidentified links, as well as the currently selected link, are monitored for activity. If activity comes from the newborn child along the currently selected link proceed to (4h). Otherwise, emulate a peek command, returning the value 123456. This emulation, described in Section 8.1, is continued with the emulation of a 2nd peek command, this time returning 654321 as the address contents. This protocol has been set up, as described in Section 8.1, in order to allow an exploring transputer to detect a transputer which has already been booted up. Once the emulation has been completed and the two communication transputers have determined that they are connected, they may exchange identification information with the transputer which obviously sits at the other end of the previously unidentified link. Mark the link through which the identification activity occurred as identified, and continue monitoring.

h) The data received from the currently selected link is the return of the *CID* token. This return is an indication from the child that it has completed it's share of the network exploration and that its parent may continue the network discovery process. The current link, and segment of the network found along this link, have therefore all been identified. The controlling *Head* then returns to the beginning of (4) in order to further its discovery process.

5) Once all links have been identified, *CID* is returned to the parent along with the identification table of the current transputer node, and the identification tables received from any of its children (transputers which were booted by this transputer). In this manner, all the identification tables for each transputer are passed back to the root transputer.

8.4.2 The Tail

Tail executes solely on the root transputer and operates essentially the same as *Head*, except that it handles the user interface as well as any I/O. The source listing is provided in thesis supplement. The results of the search are cumulated here, instead of being passed back to its parent (the PC Host) where they are collated and presented in a decent format to the user. Step (5) of Section 8.4.1 is thus replaced with:

5) Collect and collate all the identification tables from each of its siblings, storing them in a table. Once complete, output the results to the screen, giving the user the option of also storing the results in a file.

8.5 Breadth First Worm

This type of worm was introduced in order to cut down on the time spent by booting the network. Each time it was necessary to boot a transputer, a copy of *Head* had to be passed from the Host PC along the boot path to the transputer which has to be booted. By ensuring that the path which the boot code had to travel was the shortest, as well as distributing the boot code amongst several transputers, the boot time could be reduced. Thus, the depth first worm requires one copy of the head to be read from the host PC per transputer booted, while the breadth first worm may use the same copy read to boot multiple workers.

The breadth first worm consists also of two parts which effectively perform the same operations. The two portions, similar to the depth first worm, are named *Mouth* and *Foot* respectively. *Mouth* performs the majority of the searching, being used as the "rubber stamp" copy, while *Foot* is used to handle the I/O and is executed on the root transputer only. *Foot* operates in a similar manner to *Mouth* except that the results of the search are accumulated here, as well as the I/O requests being translated to host PC requests.

The "breadth first" worm boots the network in an explosive fashion, level by level. As illustrated in Figure 24, each level consists of a group of transputers which have the same smallest number of intermediary transputers between it and the root. For example, if the root transputer is level 0, then level 1 transputers are those transputers directly connected to the root. Level 2 transputers are the transputers directly connected to level 1 transputers, but not connected to level 0 transputers. Thus, after the root transputer is booted, all level 1 transputers are booted from the same *read boot file* request. Then all level 2 transputers are booted, by level 1 transputers, from the same second *read boot file* request etc. etc. Thus the transputer network illustrated in Figure 24 will require one read of the *Foot* boot file, and eight reads of the *Mouth* boot file, as opposed to the single read of the *Tail* file and 24 subsequent reads of the *Head* file of the depth first worm.

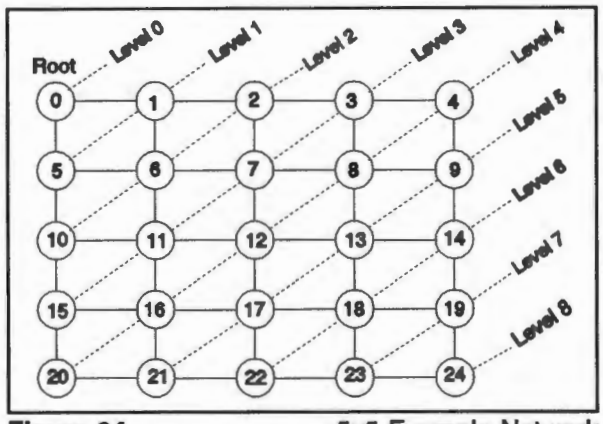

Figure 24 5x5 Example Network

Similar to the worm of Section 8.4, tokens are utilised in order to identify and explore the transputer networks. As before, a *CID* token is used to give an identity the different transputers. However, the "breadth first" worm includes an *explore* token which is used to allow the holder to explore the network. After a level of the network has been simultaneously booted, the *CID* token is passed around in order to give the newly booted transputers their identities. After receiving their *ID*, they enter a monitoring state, similar to the monitoring state of Section 8.4, while waiting for the *explore* token. Once received, they examine each unidentified link in order to further explore the network, but more about this later. The source listings are provided in the thesis supplement.

8.5.1 The Mouth

The technique used by *Mouth* and listed in source format the thesis supplement is as follows:

1) Once operational on a transputer node, directly after being booted, all links are scanned for the identification tag *ID*, which is received as the identification token *CID*, from the parent transputer. Set *ID* to the current token value of *CID*, and increment the *CID* token. This advances the unique identification tag to the next available value.
2) The link through which *CID* was received is classified as the parent/booting link and identification information (such as the parent's *ID*, the link number of the parent which is connected to the parent link, the node's own transputer type and memory size, etc.) exchanged between the parent and child transputers. All other links are marked as being unidentified. *CID* is then returned to the parent to allow the remaining transputers of the same generation (level) to be identified. The transputer then enters a monitoring state whereby the unidentified links are monitored, and identification information exchanged when activity is detected.
3) If the *explore* token is received from the parent, the exploration may continue be examining each link, as described in Section 8.1. If a booted transputer is located, they merely exchange identification information as in (4d) of Section 8.4.1. For each link which has been identified as having an unbooted undiscovered transputer connected, mark that transputer as discovered by poking special identification values at specific locations in memory. Each transputer initially examines these specific locations for every unbooted transputer, before attempting to poke them, in order to determine whether it actually must boot the transputer or whether another transputer will perform the booting. If the special values are found, the transputer must already be discovered so the task of booting it is left to the transputer which initially poked these values into the memory. Each link connected to an unbooted transputer, first discovered by this transputer, is marked as requiring a *Mouth* boot file.

4) The *explore* token is returned along with a *need.boot.file* flag to the parent. This flag is set if any links are marked as requiring *Mouth* boot files, and clear otherwise. This transputer then waits for the copy of the *Mouth* boot file which it forwards down each of these links, booting the necessary transputers.
5) Wait for the *CID* token to return from the parent and, once received, forward the token down each of the links which asked for a copy of the *Mouth* boot file. Once the *CID* token has been to all the necessary links, it is returned to the parent and the *explore* token waited upon from the parent. As with the *CID* token, the *explore* token is passed down each of the links which required the boot file. When the *explore* token is returned from each link, a *need.boot.file* flag accompanies the token, indicating whether each of those siblings require a copy of the *Mouth* boot file themselves. If any siblings require a copy of the *Mouth* boot file, this transputer must require a copy in order to be able to pass the copy to its siblings. This is attained by returning to step (4), otherwise step (6) is executed.
6) Once no more boot files are required, as indicated in the *need.boot.file* flag accompanying the *explore* token on its return to the parent, each transputer forwards all its link identification information to its parent, along with any identification information it receives from its siblings.

This method allows each of the newly booted transputers to identify itself in turn, check their links in turn, and return to its parent whether a copy of the boot file is required. By repeating this process, the transputer network is booted, level by level, with each transputer identified individually within each level, as well as individually exploring their links.

8.5.2 The Foot

The *Foot* operates in almost exactly the same manner as the *Mouth and is listed in the thesis supplement*. The major difference is that all requests to its parent, the host PC, are translated into PC Host requests. The *CID* token is initially received from the parent, being patched onto the end of the *Foot* boot file, but is never passed back. The requests for the *Mouth* boot file from the siblings are translated into actual file read requests from the host PC. The *explore* token is also generated and controlled by the *Foot*. *Foot* also retains any link information which it receives, building up a network identification table, and outputs the results when none of its siblings require boot files. The method allows the entire network configuration to be determined.

8.6 Conclusion

Both worms were successful in their method and operation, being released into public domain where they are used by prominent companies in South Africa to boot and check mixed network configurations. If t is the time required to boot a single transputer, then the time to boot a network of n transputers using the depth first worm is $t.n$. The fastest boot time for the breadth first worm is $t.(1+log_3 n)$ for a tree, with the worst time being $t.n$ for a pipeline. The average time for the breadth first worm was $t.(2\sqrt{n}-1)$, the time to boot a grid network.

The inhibiting factor of both worms was that the **Head** or **Mouth** had to be read from the disc of the host every time a new copy was required by the depth and breadth first worm respectively. This was done to ensure that the worm ran entirely within the on-chip RAM of the transputer (within 2K) since off-chip RAM could not be guaranteed. If there was enough space on the root transputer to keep a copy of the **Head** or **Mouth**, the boot time could be reduced dramatically, being slightly longer than $2t$ for most networks.

Both worms also provide a good harness for a farm network. The master would reside in the root transputer with the **Tail** or **Foot**, with the worker being included with the **Head** or **Mouth**. This ensures that the master resides on the root transputer and that all the remaining transputers are filled with workers, providing a truly generic farm network which could execute on almost any transputer network. When the network configuration information is returned to the root transputer, the master could determine an optimal routing strategy for the farm using this information, and pass the information back to the workers. The workers would then establish their routing processes and wait for their work, continuing as a normal farm.

The worm has already been adapted into a farm which is are used to test the communication channels of any network in order to test data transmission rates on large transputer networks. Once the worm has completed its exploration, the master and workers then begin intensive communication between each other, recording their data transmission rates. This is used as a benchmark to compare assorted transputer systems from different transputer vendors. A copy of this worm adaptation may be obtained by writing to the author directly.

9 AIRFLOW MODELLING

9.1 Introduction

The initial aim of the development of this system was to investigate and implement an alternative simulation method for predicting airflow and thermal effects around and inside buildings which is compatible with the first principles of physics. The system was later extended to include low and medium speed airflow around objects defined by planar surfaces. One method of simulating and analysing low and medium speed airflow and thermal effects in the architectural, building science and aeronautical environment, the essential feature of this system, is through molecular tracing in simulation studies. Simulation studies were used since they tend to simplify problems or events and provide faithful reproduction of physical phenomena. The technique applied was a unique "statistical" method of simulating thermal and air flow in and around objects which is more straightforward to implement than "mathematical" approaches which are generally far more computationally intensive. The actual implementation of the method was further simplified through the use of high Reynolds numbers. The major advantages of this technique over the well used Navier Stokes method is that Navier Stokes requires complicated equations to be developed to describe an object, and these equations to be solved. Furthermore certain assumptions and approximations have to be made about the physics, whereas in the technique developed no approximations are made to the physics and objects may be described by simple linear equations.

The problem was to establish, test and verify the feasibility of such a technique and then extend the technique to cope with complicated problems as well as novel physics and computer approaches. The simulation is still, however, computationally intensive, providing a suitable test case demonstrating the effectiveness of parallelism to achieve real-time responses for the application.

The physics problem was addressed and the feasibility and methodology of numerical simulation demonstrated by physics consultants. The numerical technique for studying

heat and air flows around objects was shown in several reports by the consultants to be feasible within the bounds of reasonableness, providing consistent and realistic results [Hay87,Hay88,She89]. The consultants produced several working version of the code written in Fortran 77, each being a logical succession of the previous version, until finally a model was produced which could simulate the flow of air and heat through a medium containing physical obstacles. Each version was translated into OCCAM 2 in order to obtain the increase in performance given by code written in OCCAM 2 over Fortran 77. The model was implemented for a single, as well as multiple cells, in both Fortran 77 on a PC and in OCCAM-2 on a small network of five transputers and a PC, and the results obtained appeared consistent with physical reality at the molecular level and were within realistic computational times.

Due to the several versions of the code which were produced, a standard parallel shell was developed where each version of the code could fill the shell to produce a parallel equivalent. This chapter of the thesis therefore deals with the parallel implementation of the airflow model. It addresses the different approaches employed in creating the shell, the problems encountered and some of their solutions found, as well as the final implementation. It also discusses the speedup and efficiency obtained and some aspects of load balancing and routing problems.

9.2 Network Configuration

Five transputers were initially made available for the implementation of the thesis. The first was a T414 with 2MB RAM on a B004 board, while the remaining four were T800's with 1MB each on a Microway Quadputer board. The links between the four T800's are hard-wired, with one link open on each. Link 0 of the T414 is always used to communicate to the PC and the remaining open links were initially connected to the open links of T800's W, X and Y. The open link of Z was unconnected. The network is thus configured as illustrated in Figure 25.

9.3 Airflow Network 1

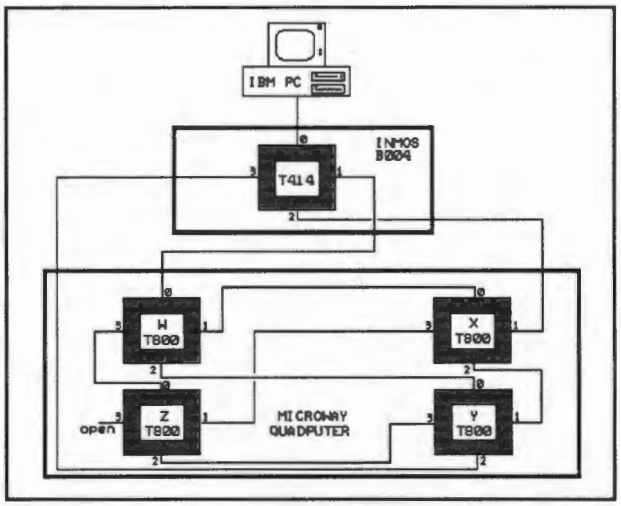

Figure 25 Initial Transputer Network

While the airflow modelling program was still under development, various tests were performed in order to determine the actual parallelism of the problem at that stage. The objective is to trace the path of a molecule in two dimensions in a small area. The area is divided into a grid where individual molecules are traced in each grid cell. When a molecule drifts outside a cell, a new molecule is started off in the same cell. Calculations for a cell are complete when the number of collisions of the traced molecules with other molecules in the cell exceeds a certain pre-determined amount (say MAXCOL). The calculations are performed for all cells before the next iteration begins. Since the calculations for each cell were, at this stage of the analysis, independent of each other they may be performed independently by several processors, hence introducing parallelism. As part of the output of the system, the point of collision for every collision in each cell is plotted on the screen.

Assuming that more than one processor is always available, a farm topology suits this form of algorithm best as several almost identical independent tasks may be completed in parallel. Thus a work scheduler or master is used to distribute work to idle worker processes, and to service the busy workers, accumulating their results once their tasks have been completed. The worker processes, as usual, perform the majority of the calculations. These worker processes reside on individual processors, one per processor, along with a message router of high priority (as discussed in Section 5.4.3) which intelligently handles the protocols and message passing between the processors. An average of 30% improvement of processor usage was obtained by assigning these simple priorities, without which the sending processor will be idle while waiting for the receiving processor's router to become active.

The foreman was placed on a T800 rather than the T414 since, after an initial glimpse, it was determined that a fairly large amount of real calculation would have to be performed in order to create some work and the T414 has no numeric co-processor. The T414 instead simply acts as an interface to the PC, booting the network, determining the maximum number of collisions to be monitored for convergence to be attained, adjusting the screen co-ordinates passed to actual co-ordinates, displaying the results etc. etc. It is therefore excluded from the farm as it does not perform any work, just menial I/O, and therefore has also been omitted from the calculations of speedup and efficiency. The work to be executed involves the computationally intensive tracing of a molecule within a single cell. This task is given to the workers. Therefore there are be four types of processes in operation at any one time. These are:

- The Server/Controller
- The Foreman/Work Scheduler
- The Worker
- The Router

With the transputer network of Section 9.2, the work load is distributed as follows:

- The Server/Controller on the T414,
- The Foreman/Work scheduler on transputer W,
- Workers and Routers on transputers W, X, Y and Z.

Note that a worker is placed on transputer W running concurrently with the foreman since the foreman only allocates tasks when at least one of the workers is idle. The remainder of the time the foreman will be idle. In order to utilise the computing power and time of transputer W more efficiently, a worker was placed on W with low priority allowing a fourth cell to be operated upon when the foreman, a high priority process, is idle. The foreman will become idle when all the workers are operational, as results from a worker are required in order to generate further work. The foreman is made to wait on the links to the workers, allowing the worker sharing the processor with the foreman to become active, utilising processor W's time more efficiently. This

process placement is illustrated in Figure 26.

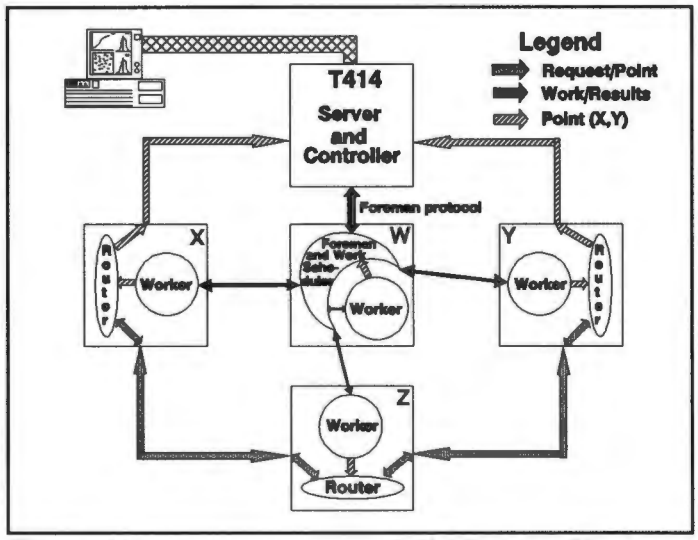

Figure 26 Initial Process Placement

At this stage it must be noted that more than one transputer processor may operate on a single cell. This is due to the fact that the foreman decides on the cell to be operated upon and the starting point of the molecule within the cell. If the molecule drifts outside the cell before the maximum number of collisions is reached, the worker returns the results obtained this far to the foreman, along with the number of collisions that have occurred to that stage of the calculation. That worker (say X) is then idle and is waiting again for a cell on which to operate. If worker Y's (say) molecule also drifts out of its cell at the same time, returning its results to the foreman and becoming idle, worker Y may be given the continuing task of completing the calculations for the cell on which X was previously operating, with worker X similarly being given Y's old cell. This is because the foreman takes no preferences as to which worker is given a task since all the workers are identical.

Two variations of a set of four different tests were done on this network using a 10x10 grid. The tests are numbered from 1 to 4 and use 1 to 4 workers respectively. Test 1 used X as the worker, test 2 used both X and Y, test 3 used X, Y and Z, and test four used the network exactly as shown in Figure 26. The first variation involved plotting the point where every collision occurred in each cell while the second variation of the test involved only plotting every 10th collision. The results are listed in Table II, the times being given in completed seconds, with a discussion of the results following:

Test No	1000 Collisions	1000* Collisions	10000 Collisions	10000* Collisions
1	45	44	452	412
2	30	22	307	206
3	28	15	282	140
4	32	14	317	125

* ⇒ 10th point plotted only

Table II Initial Result Times in Seconds

Results and Deductions

1) There is little difference in the times between the two variation tests for test 1 for one thousand collisions. The slight reduction in time is due to the decreased communication overheads which resulted due to only 1/10 of the points being plotted. The same may be said for the ten thousand collision test.

2) The results of tests 2, 3 and 4 of the first variation show that a bottleneck has occurred in the communication of the points to be plotted to the PC, since the addition of workers gives a small improvement in speed. The increase in time for test 4 is due to the increased overhead of time slicing between worker and foreman. Using simple mathematics, it may be deduced that the PC with a Hercules screen has a maximum throughput of approximately 3600 points per second as given below by Equation (19), using test 3 for 1000 collisions plotting every point.

$$\begin{aligned} \textit{Points per second} &= \frac{\textit{Total Cells} \times \textit{Collisions per cell}}{\textit{Time}} \\ &= \frac{100 \times 1000}{28} \\ &= 3571 \end{aligned} \qquad (19)$$

It must be noted that this bottleneck to the PC cannot be eliminated other than by a reduction of communication between the B004 and the PC since link 0 of the T414 is the only link to the PC from the network. Hence **all** communication to the PC goes via link 0 of the T414.

3) With the introduction of the second variation (only every 10th point is plotted), the bottleneck was eliminated. The times obtained for the different number of

workers are almost classical: Test 2 shows times exactly half of test 1, demonstrating the parallel nature of the problem since two transputers solves the problem in half the time one takes. The foreman processor has been ignored here since the time it spends creating the work is a small percentage of the time spent by the worker performing the work. It is the speedup of the section of the computation intensive code being parallelised which is examined here.

Test 3 also shows favourable results for ten thousand collisions as 1/3 of 412 is 137, the time actually taken being 140 seconds. The three second overhead may be attributed to the fact that Z has no direct link to the T414 and hence is routed via either X or Y, resulting in a small increase in overhead. The X and Y workers on the other hand have to time-slice with the router to allow their points (and Z's) to be forwarded to the T414, resulting in a loss of time. These overheads would not show up in tests 1 and 2 since the routers would not be fully operational at this stage and the time taken for each loop was only the time taken for the workers and the necessary communication.

4) In order to examine the concurrent execution overlap of the foreman and the workers, some timing tests were performed, placing a single worker on the farm on processor W. The times for one thousand and ten thousand collisions, plotting every point, are 53 and 523 seconds respectively. This increase of time from test 1 illustrates the approximate overlap when the foreman and worker are running concurrently.

To determine the overlap between the worker and the master, consider the following:

a) There are 100 jobs (10x10 cell) to be completed, with 10000 points.

b) The total time when both foreman and worker are on the same chip is

$$t_{sequential} + 100t_{generate} + 100t_{work} = 523$$

where $t_{sequential}$ is the time for the sequential, unparallelised part of the algorithm, $t_{generate}$ is the time spent by the foreman to generate one of the 100 tasks, and t_{work} is the time taken by a worker to perform a task (assume all tasks take equal time here).

c) The total time when foreman and worker are on different chips is

$$t_{sequential} + t_{generate} + 100t_{work} = 452$$

since the worker must wait for the first task to be produced, while the remaining tasks are ready to be executed by the worker when it finishes its previous task. This is because the foreman generates the remaining tasks concurrently to the worker performing the work and one can assume that the time required to generate the task is less than the time to perform the work. Therefore work may be produced faster than it can be completed.

d) From (b) and (c) we can deduce that

$$99t_{generate} = 71 \Rightarrow t_{generate} = 0.717172$$

e) Using the time for two workers on separate chips (overlap in work)

$$t_{sequential} + t_{generate} + 50t_{work} = 307$$

and solving for (b) we get

$$50t_{work} = 145 \Rightarrow t_{work} = 2.9$$

Thus the ratio of generating work against performing the work is approximately 1:4.044. Thus the generation of the tasks forms approximately 20% of the time taken for a sequential algorithm, hence it would not be fair to omit the foreman from speedup and efficiency calculations, even though the master will spend ≈75% of its time idle. Thus the time $t_{generate}$ must be reduced.

Another factor to be taken into consideration is that the workers, when placed on X, Y and Z, will be mostly in on-chip RAM, and hence will operate faster than it does in the off-chip RAM of W (the master resides in on-chip ram). Furthermore, there is only a 3% to 9% speedup for one thousand to ten thousand collisions respectively when every 10th point is plotted.

9.4 Airflow Network 2

Further examination of the airflow method revealed that the foreman need not have the current statistics of each cell returned to it by a worker when the molecule drifts outside of the cell and does not necessarily need to calculate the starting position and return the statistics. Instead the worker may decide itself on a starting location for a new molecule through its own random number generator and keep the statistics until the maximum number of collisions (MAXCOL) is reached. This reduces the amount of real arithmetic required by the foreman, as well as the message passing to the foreman, giving it less work in order to create a task and hence allowing it to be moved onto the B004. Furthermore, the actual number of transputers used to solve the problem may not be known, requiring a more global method of worker distribution. Hence, the router must allow for a greater variety of message types, as well as a message forwarding system for those transputers which are not connected directly to the B004, increasing its complexity.

Therefore a second network, illustrated in Figure 27, was designed. The times achieved by the network for 1, 2, 3 and 4 transputers, plotting every tenth point only are listed in Table III. The times represent the average over five iterations to the first decimal point and should be compared against the second variation times of Table II. All times were recorded by the T414 transputer using its lower priority clock and include plotting of the results.

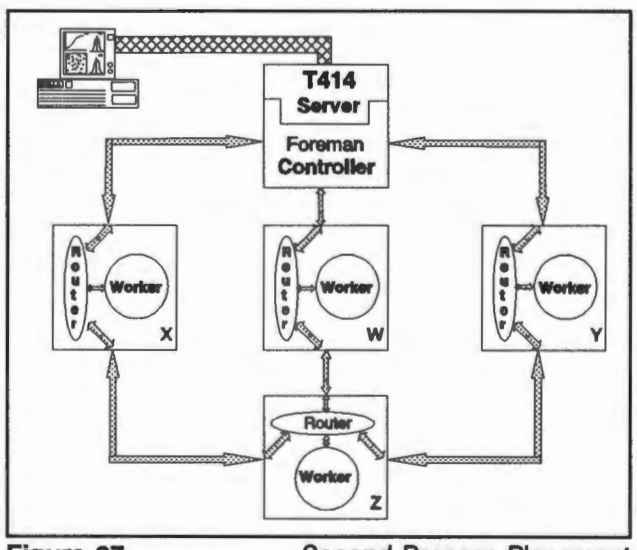

Figure 27 Second Process Placement

Test No	1000 Collisions	10000 Collisions
1	38.2	358.0
2	19.3	179.4
3	13.3	122.0
4	10.1	90.7

Table III Final Test Times in seconds

As may be seen from Figure 27, the controller resides on the T414. All messages to transputer Z are routed through W, although they could also equally be routed through X or Y. Also, due to the controller being on the T414, an initialisation overhead of an extra 13.2 seconds was found. This overhead would only occur once in the initialisation of the program and effected the time for the first iteration only. The initialisation overhead was removed from the times for the iterations given in Table III. The time taken for a completely sequential algorithm plotting every 10th point, on a single T800, using its low priority clock, was 51.0 seconds. If the initialisation time of 13.2 seconds is removed, the remaining time of 37.8 seconds compares favourably to the time of test 1 given in Table III.

Recalling the parallel version time of 53.0 seconds from Section 9.3, we can see that there is an additional 2 second overhead in implementing the first method, while the sequential version was only 0.4 seconds faster than the second parallel version. This latter difference could be diminished even further by using a T800 as the controller which still has to perform some real arithmetic in order to initialise the network and plot the results. The foreman, having completed the initialisation, then merely passes to the workers the cell they are to operate on. The extra overhead of the foreman is thus extremely small when it came to creating work tasks, and results in an insignificant increase in time when it is moved back to one of the T800s. The major increase in performance here was therefore a result of the improved method and reduction of time which was required by the foreman in order to create work tasks. The foreman therefore remained on the T414 in order to retain consistency throughout the farm.

One of the problems encountered was that of deadlock, as described in Section 5.4.4. The problem occurred when W would try to send a message to the T414, and the T414 would attempt to send a message to W. This would occur in one of two instances:

1) Transputer Z had completed its task and was waiting for work which was about to be sent by the controller through W. W on the other hand would still be calculating the points and sending them through to the controller to be plotted. Hence the messages would collide on the link between W and the controller (both routers would be trying to send messages and hence neither would be expecting any messages) resulting in a deadlock between the two.
2) Similar to that above except that W has completed its task and Z was relaying points through W to the controller. Z's point, being relayed by W, and the work from the controller for W would also collide.

One solution to the problem, described in Section 5.4.5, was to send work to W and Z only when they both were idle. When, for example, W had completed its task has work assigned, that work would be placed in a queue until Z became idle. Then a special protocol would be used specifying that a batch of N jobs is being sent, with the jobs being sent and routed through to their respective workers. The highest priority was given to the router to ensure that the entire batch of jobs was forwarded before the worker of that processor could continue. This solution is, however, wasteful of CPU time since it requires that worker processors remain idle until all the workers using the same link for communication become idle, after which time a batch of work would be submitted to the workers along the communication path. A more rudimentary and efficient solution was to sub-divide the router itself into sub-processes, each monitoring a single incoming link. The problem of message collision could therefore be eliminated. For example, the first instance of the message collision example described above, either transputer W's router or the controller (or both) will switch to the sub-process dealing with the incoming message from the controller or W. This allows the messages to be sent and received by both processors, freeing deadlock. This methods is described below in Section 9.5.

Results and Deductions

1) The timing speedups achieved by adding more transputers to the network therefore revealed the actual parallelism of the problem. With N transputers,

the addition of M transputers would provide times of magnitude N/(N+M) of the time N transputers would take to solve the problem with an additional 2% overhead.

2) Furthermore, the movement of the foreman to the T414 gave an average reduced time of 13.5% for 10000 collisions and 12.3% for 1000 collisions for tests 1 to 3. A 28% improvement was achieved for test 4 mainly because the foreman no longer had to time-share with a worker, and because of the reduced message traffic between the controller and the workers.

9.5 Airflow Network Shell

This method, described in Section 5.4.7 and illustrated in Figure 15, was the one finally used by the Airflow Modelling system. However, no worker was placed on the same processor as the master since it is necessary that each worker perform a large volume of floating point operations. Since no floating point unit exists on the T414, all workers were placed on the T800s. With the parallel shell for the model complete and fully operational, it may be applied to the several different versions of the code which exist. The speedup anticipated per transputer added is near-linear as long as the communications required are small with the amount of computations required per data set high, demonstrating the effectiveness of parallelism for this type of code.

The final implementation used the Quadputer, giving a total of five transputers, with the T414 as the controller/master and the remaining four T800s as the workers. Workers *W, X* and *Y* all included routing processes to allow the worker on *Z* to communicate through any one of these three processors to the master T414. *Z* did not have any routing processes as it was connected directly to one of the above processes, demonstrating that the worker processes at the end of each branch in the farm network tree need not include any routing processes, but merely must communicate directly to the routing processes of the worker processor down the same branch closer to the root. These routers are not concerned with the location of the actual worker processes,

whether they are either on chip or on a completely different processor. The routers merely require communication channels whereby messages may be passed to and from the workers. In order to make the shell as transparent as possible, the routing tasks merely looked at the destination of the packet, and would pass the message to the next processor if the message was not destined to the worker residing on the same chip.

Since this network was small, no real routing was required, other than the routing to Z. Should more workers be added to the system, the routing methodology of the SAPF Farm Structure discussed in Section 6.3 could be adopted. In order to make the system completely portable, the Airflow Model could use either version of the worm discussed in Section 8 to boot the code, with the worker and its routers merely being appended on the end of the network search so that once the search has completed and the network configuration been determined, the actual numerically intensive code would be executed. Combining the worm and the routing strategy of the SAPF Farm Structure, the master could easily be modified to extract the routing information from the network search results of the worm. The routing methodology could also be modified to build an efficient routing table as discussed and verified by [Hen90] which would not overload a worker processor with communications, but distribute with communications load amongst the workers at the same level in the network tree.

The OCCAM source code of this final network is listed in the thesis supplement, using the technique described below in Section 9.6 to increase the performance. It must be noted that the code was translated directly from Fortran 77 to OCCAM 2 with no alterations to the structure, algorithm or technique. The Fortran 77 code was provided by the physics consultants. Since it was the parallelism of the problem which was being investigated in this thesis and not the actual problem of Airflow Modelling, the translation was kept as close as possible to the original Fortran 77 code. The only modifications were the introduction of parallelism, related communications and the technique described below. This approach resulted in the development of the Airflow Shell which was filled with the different versions of translated Airflow Code as each version of the code was developed by the physics consultants.

9.6 Data Dependency

Data dependency in the Airflow Modelling System exists, however, in the calculations between each directly connected cell. This effects the rate of convergence of the iterations, as the results of iteration *i* may be used in iteration *i+1*. In the case of the Airflow Model, we are investigating the flow of air and heat through a square medium with an initial stable air environment and uniform temperature distribution, as well as a uniform pressure distribution. A high pressure and temperature source is introduced at the top border of the square and a pressure and temperature sink at the bottom border with the left and right boundaries acting as insulating boundaries. This is illustrated in Figure 28. The flow direction of air and energy, referred to as the *flow*, is examined from the top to the bottom border of the medium, with obstacles contained within the medium, until the airflow pattern and temperature of the entire medium stabilises and reaches a state of equilibrium.

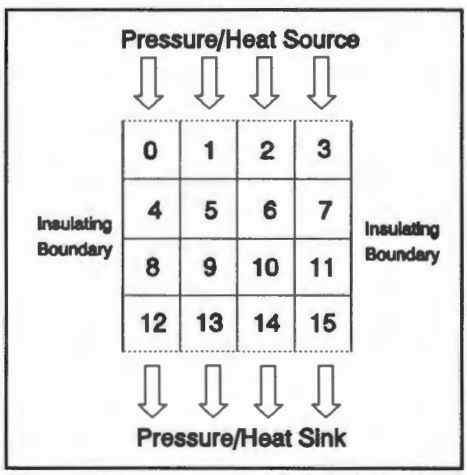

Figure 28 Grid of Medium

The medium which we wish to inspect is divided into smaller regions by a grid, as illustrated in Figure 28, with calculations being performed on each cell in the grid. Each cell uses the information from the adjoining cells (left, right, top and/or bottom) in the calculation of its *flow*. On a sequential machine one may iterate on a loop through the cells 0 to 15. For example, the resultant value of cell *c* iteration *i* is computed using the values of the cells to the left and above produced by iteration *i* and the values obtained by the *(i-1)th* iteration of the cell to the right and below.

Assume that the calculation for each cell takes time t, the grid dimension is n by n and the number of iterations required for all cells to reach equilibrium is I. The time taken for the program to complete on the sequential machine will be

$$time_{sequential} = n^2 \times I \times t \tag{20}$$

9.6.1 Case where $p \geq n^2$

Now consider a parallel machine with p processors where $p \geq n^2$. We wish to duplicate exactly the calculations which would be performed by a sequential machine on the parallel machine, introducing the maximum amount of parallelism. The *flow* for some of the cells may be calculated independently of each other.

Allow c_i to be the *flow* of cell c at iteration i. This value will typically be calculated using the *flow* of cells $(c\text{-}1)_i$, $(c\text{-}n)_i$, $(c+1)_{i-1}$ and $(c+n)_{i-1}$. The following discussion concerns the parallelisation of the algorithm using this method.

Using the example 4x4 grid of Section 9.6, we note that once 0_1 has been calculated (in time t), 1_1 and 4_1 may be computed in parallel since all the information they require is available. The value 0_2 may be determined by a third processor concurrently to 1_1 and 4_1. The latter three operations would take time $3t$ on a sequential machine, but since there is a processor available per cell, the operations will take time t as they may all be performed in parallel. Similarly, for the next cycle of t, operations 0_3, 1_2, 4_2, 2_1, 5_1 and 8_1 may be operated in parallel. Thus, for the *flow* $(n^2\text{-}1)_1$ to be calculated, 2n-1 cycles of time t each will have to be performed.

Note that for the next time cycle, each cell c will have the *flows* $(c\text{-}1)_i$ and $(c\text{-}n)_i$, as well as $(c+1)_{i-1}$ and $(c+n)_{i-1}$ available. Thus, for subsequent cycles, each iteration may complete at the end of each cycle. Hence the time for each subsequent iteration will be t. As I iterations are required for the system to stabilise, the time required by the

parallel machine will that of (21) derived from the time *(2n-1)t* for the first iteration to complete and *(I-1)t* for the subsequent iterations.

$$time_{\parallel} = (2(n-1)+I)t \tag{21}$$

More generally, consider a n by n grid where each cell operation takes time t and I iterations are required for convergence. The times for a sequential and parallel machine with n^2 processors are given by (20) and (21) respectively. When greater detail is required of the *flow* distribution in the medium, the granularity *(n)* of the grid may be increased.

Using the speedup formula (1), the speedup for airflow becomes:

$$S_{Airflow}(n) = \frac{n^2 \times I}{2(n-1)+I} \tag{22}$$

This, of course, is based on the following assumptions:

1) The operations for each cell are primary operations, that is, they cannot be parallelised.
2) There will always be one processor available per cell.
3) No time overheads for communications.

The 3D graph of Figure 29 demonstrates the relationship between the number of cells/processors (assuming there is always one processor per cell available), the number of iterations required to reach convergence, and the speedup obtained. Briefly, one can see that the speedup obtained improves exponentially with the number of iterations needed for convergence. An almost linear speedup is obtained when the number of iterations required is much greater than *2n-1*. However the speedup does begin to decrease logarithmically as the number of cells/processors becomes large. This is because when $I < (2n-1)$, the speedup is poor as most of the time is spent filling the cells with the values of the different iterations. Only when all the cells are calculated

concurrently is there a reasonable speedup due to parallelism. The graph depicted has maxima of a 10000 processors (100 by 100 grid), 1000 iterations and speedup 8333.

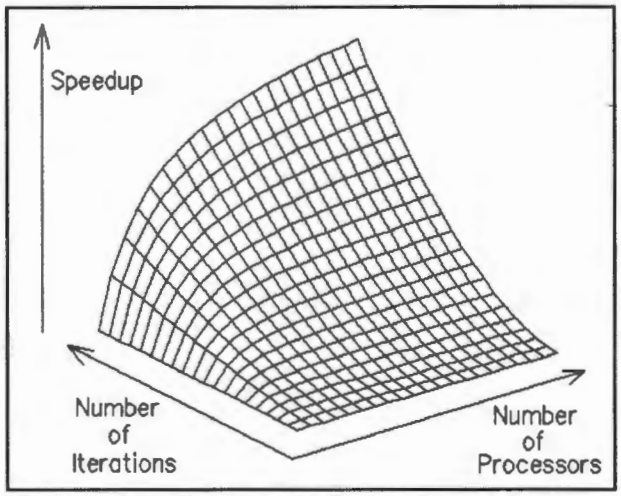

Figure 29 Speedup vs Processors vs Iterations

9.6.2 Case where $p < n^2$

Consider now the case where $p < n^2$.

Define Γ_r to be the number of processors required for any time cycle r preceding the completion of the 1st iteration according to the formulae:

$$\Gamma_r = \begin{cases} \sum_{j=1}^{r} j & \text{for } r \le n \\ \sum_{j=1}^{n} j + \sum_{j=1}^{r-n} (n-j) & \text{for } r > n \end{cases} \qquad (23)$$

Now for r such that $\Gamma_r \le p < \Gamma_{r+1}$ holds, each cycle following cycle r will have more cells to compute concurrently than processors available. When this occurs, each subsequent cycle will utilise every processor fully, resulting in a greater overall efficiency per iteration. Remember that not all the processors will be used for cycles less than r resulting in poor processor efficiency. Since the efficiency is defined to be the speedup divided by the number of processors used, and the number of processors remains constant, the speedup must increase as the number of iterations increases.

The above method described how the results of the sequential algorithm may be duplicated exactly by a parallel machine. If the parallel version of the code is not limited by this exact duplication, there are often alternative methods which may be used to achieve the same final results.

9.6.3 Alternative Method

If the results per cell per iteration are not to be duplicated exactly, several alternative methods of ordering the cell calculations exist. One such alternative is described below:

Consider the inclusion of the condition that the value $(c\text{-}n)_i$ be available for the calculation of c_i, with the surrounding cells to the left and right may either have the values $(c\text{-}1)_i$ or $(c\text{-}1)_{i\text{-}1}$ and $(c+1)_i$ or $(c+1)_{i\text{-}1}$ respectively. The most recently calculated value in either case will be used if it is available. Cell $(c+n)$ may only have the value from iteration $i\text{-}1$ as it is dependant on the value of c_i. In this manner we introduce row dependency, allowing all the cells in the first row to be calculated in parallel for the first time cycle, the second row in the next time cycle along with the first row for the next iteration etc etc. In this manner the last cell will be affected by the *flow* source after n time cycles instead of $2n\text{-}1$. This, in fact, resulted in an overall convergence occurring in fewer time cycles with less iterations as the cells in the last row are affected earlier on in time than the previous methods.

However, the improvement in speedup over the previous parallel method was small with a reduction in the execution time of only 5%. This was mainly because a large number of iterations was required in order for the system to stabilise and that the initial advantage gained was subdued by the sheer volume of iterations which was to follow. In test examples where fewer iterations were required in order for the system to stabilise, it was noted that the improvement of speed was more noticeable. However, these test examples were too simplistic and would not provide the norm for the average run.

9.7 Airflow Conclusion

The Airflow Modelling Shell was successfully implemented, being used as the foundation for each version of the code produced. Each of these versions, after being translated into OCCAM 2, would produce results equivalent to that of their sequential partners. The speedup results obtained were impressive, with an almost linear speedup being obtained for each run. The major factor limiting the speedup was the throughput of the graphics device used to illustrate the actual flow. This could be overcome by reducing the frequency at which each worker would plot its information on the screen (plotting only every 10th point). If the workers were to retain their intermediate results instead of displaying them, only plotting the final results once the model had stabilised, this bandwidth problem would be overcome. This is very feasible since the medium was often divided into a very fine grid where the tracing of the molecules within the cells on the graphical screen would flood the graphics area provided for cell output, resulting in cluttered and illegible output.

10 SAPF - SEMI AUTOMATIC PARALLELISER OF FORTRAN

10.1 Introduction

The initial aim of the development of this system was to produce a tool which would accept, as input FORTRAN 77 code, and produce as output FORTRAN 77 code which may be compiled under 3L Parallel Fortran, using the SAPF Farm Structure of Section 6.3. SAPF is directed at creating processor farms from iterative code where the calculations performed in iteration *j* are not dependant on the results of iteration *j-1*. It was therefore aimed at creating "Processor Farms" where each iteration step can be distributed amongst identical processes running on different processors to be calculated concurrently and independently of each other. Thus the iterative workload is distributed among several processors, and should run in the order of magnitudes faster than the sequential code given as input. This is similar to the tool SUPERB [Zim88], which has been developed for the SUPRENUM supercomputer [Beh88], performing MIMD and SIMD parallelisation of iterative code.

For example, consider a DO loop where the volume of computations contained within the loop and the number of repetitions of the loop is large. This is a suitable candidate for a transformation into a processor farm. The workers may consist of duplicates of the loop contents, with the master being the remainder of the code. The workload, in this case, consists of the consecutive values of the loop variable which are passed individually and uniquely to each worker, such that no two workers perform the same iteration, in order that each worker may perform the stage of the overall iteration passed to it. The results of each iteration could then be passed back to the master, or could be retained by each worker and only returned once the work tasks have been totally distributed and the workers have completed their final tasks. Once the master has received an indication from a worker that the worker has completed the task it was previously given, the master may then send the current value of the loop counter, and update this value for the next available worker.

The final outcome of the development was a tool which may be used in the production of such code, which is where the "semi" portion of the title originated. The user is left to determine which loops they would like "pulled out" and distributed across the worker processes as work jobs. The tool SAPF is used to determine the data flow and relationships between the different sections of code (e.g. between the contents of an iterative loop and the remainder of the code), allowing the user to decide which portions of code may be run in parallel and the data that must flow between the sections of code.

SAPF is not suited for partitioning sequential code into several individual distinct inter-communicating processes, although it may be used as an aid to determine the data relationships between different sections of code which the user may see as candidates for separate concurrent processes. This form of parallelism was avoided in this thesis as it is a more complex and somewhat larger field and may be the topic of numerous other M.Sc and Ph.D theses, such as the M.Sc of C.Handler [Han89] which deals with the optimal placement of processes across processors.

10.2 Parallelisation through SAPF

10.2.1 Code Structuring

The first step of parallelisation when using SAPF involves structuring the code in such a manner that iterative loops and "Hot Spots" are easily identifiable. This task is performed automatically through the utilisation of the commercially available Fortran code unscrambler SPAG which structures the code, making it more comprehensive through the reduction of "Spaghetti" code into readable and structured code. This format change is administered on the code since the Fortran code which is to be parallelised is often in an unreadable state, making the identification of iterative loops and "Hot Spots" difficult, if not almost impossible.

10.2.2 Master/Worker Nomination

Once the code has been structured, iterative loops may easily be identified. The user then manually extracts nominee iterative loops and "Hot Spots" where the contents of such may provide a suitable candidate for a worker. This may also be achieved automatically through a dynamic code profiler where the time intensive iterations can be identified routinely. Once the computation intensive iterations which constitute the majority of time spent in executing the code have been identified, they are nominated as viable candidates for workers. This produces a group of code pairs $\{(M_1,W_1),(M_2,W_2),...\}$ where each W_i is the iteration or "Hot Spot" contents, and the corresponding M_i is the full code with W_i removed. Therefore, for each code pair (M_i,W_i), M_i constitutes the master candidate and W_i forms the worker candidate. The actual code is therefore $M_i + W_i$ which is the code pair (M_i,W_i). Also we have $(M_i,W_i) = (M_j,W_j)\ \forall\ i \neq j$.

10.2.3 Static Checking

Each W_i is then expanded into a fully compilable program with the variable headers (variable and common block definitions) of M_i being included in the W_i code. M_i and W_i are then individually submitted to an analyser, called SAPF (Semi-Automatic Paralleliser of Fortran), which reads in the Fortran code and performs static checking, producing an output file. SAPF was created for the thesis and is passed two parameters, the first is the input file (which may just be a list of files which collectively form M_i or W_i) with the second being the output file.

In the first parse, the static checker SAPF builds a list of local and global variables for each routine of M_i and W_i and notes if and where (in which routine) any assignments, references and/or I/O are performed on the variables within these routines. In the second parse cross-reference routine calls are resolved where variables passed as parameters and global common blocks variables effected by calls to secondary routines

are updated in the calling routine. This is because these variables are affected by calls to secondary routines and must be marked as such in the calling routine. During this second phase, variable types, sizes and common blocks are compared between the routines to ensure a match between the called and calling routines, with any mismatches being reported. This feature has already proved its worth as errors of this type were found in code submitted for parallelism which was frequently utilised and previously assumed to be fully functional.

If any subroutine calls or functions are undeclared, SAPF displays the call to these routines (as called by the program) individually, while attempting to resolve the cross-references, and prompts the paralleliser to manually enter the number of parameters which should be passed to the routines, the name and type of the parameters and the operations which are performed (Assigned, Referenced, Input or Output).

These static checks are not performed by the majority of Fortran compilers but are extremely useful in the development large systems which are written by more than one person. This checking alone ensures that problems arising due to miscommunication are resolved quickly and effectively before the debug and execution testing phase of the code development begins. Thus this feature is a useful tool in itself and in fact is used at AEROTEK as such. Typical problems encountered in code submitted for parallelisation are:

- Incorrect ordering of variables in parameter and common blocks
- Mismatching variable types in parameter and common blocks
- Mismatching array sizes
- The incorrect number of variable appearances in parameter and common blocks

In the event of one of the above errors occurring, SAPF will give a full reference of the file and line number where the error occurred. If a call is involved, both the calling and the called subroutine/functions are listed. All errors may be re-routed through dos by passing a /LOG parameter to SAPF and the using the usual dos re-route notation on the command line.

Once the cross-references have been resolved, the third parse lists any uninitialised variables (variables referenced but never assigned), redundant variables (variables assigned but never referenced) and obsolete variables (variables never referenced nor assigned) are listed and reported on. This is another useful "extra" of SAPF.

10.2.4 Output

The output file generated by the static analyser lists alphabetically all the common blocks used by the code submitted and all the functions and subroutines with their variables, their variable types and dimensions (if arrays). Attached to each variable listed is also an indicator whether the variable was referenced, assigned, input and/or output within the routine, as well as whether the variable is part of a common block, parameter to the routine, external and, in the case of a subroutine or function, whether it was called.

SAPF makes no conjectures about the code flow and makes some assumptions when performing the static analysis. One assumption is that *all* the variables referenced and assigned within a routine will *always* be referenced and assigned by any call to that routine. The variables of a calling routine which are passed through parameters or common blocks to a called routine are thus maximally affected by the operations of the called routine. This is due to the fact that no code flow profiling is performed by SAPF. Furthermore, in the majority of routines, it may be assumed that the code is executed in its entirety and thus the actions on all the variables must be determined for the full length of code. This is especially true in the candidate worker code which is repeatedly executed, increasing the probability of having all conditional sections of its code being executed. A conditional section of code is a part of code which is only executed if certain conditions are met. With code being executed repeatedly a high number of times, the probability of all conditions being met increases, indicating that all conditional sections of the code are likely to be executed with higher probability.

Thus it must be assumed that all sections of code will be executed and this simplistic static code checking is suitable for data flow analysis between master and worker.

The analyser is fairly rigid in the checking of variables. Variables which are parameters of a subroutine or function, as well as common blocks, are not checked for initialisation, redundancy or obsoletism. This is because these variables may be assigned or referenced outside of the routine itself and may appear, incorrectly, to be erroneous within the routine. Also, the parameter and common block variables of a called routine are not effected by the previous operations of a calling routine, since the called routine may be referenced from different parts of the code. However, if the code is structured in a top-down fashion with no GOTO statements etc, this rigidness may be removed and parameter and common block checking performed, with the calling routine passing down the attributes of the shared variables to the called routine. This may easily be added to SAPF by removing the restriction on variables that only non-common and non-parameter variables be examined for initialisation, redundancy and obsoletism, as well as adding code flow analysis.

The major reason for the above rigidity was that SAPF was written to assist in the parallelisation of Fortran code and was not designed to be a static analyser, although this was later introduced in order to reduce the risk of parallelising code which itself was not complete. It is difficult enough to get sequential code running in parallel effectively without having to eliminate problems inherent to the sequential code. Thus SAPF merely performs simple static code checking and maximal data analysis between routines. Furthermore, SAPF need only determine the effect of a call on the variables within the scope of the calling routine, and not vice-versa.

10.2.5 Data Relationships

Once the static analysis is complete for each (M_i ,W_i) pair, the data relationship between the code pair M_i and W_i is determined. This is performed through a second

program called RELATE, listed in the thesis supplement, which forms part of the SAPF package, as does the SAPF Farm of Section 6.3. RELATE reads as input the two output files produced by SAPF for M_i and W_i, being passed the SAPF output file names for the code pair as the first two parameters, with the third parameter being the output file name for RELATE. In doing so, the variables which are actually required by W_i in order for it to perform the *j*th stage of the iteration are determined, as well as the result variables which are to be returned by W_i to M_i.

The output file produced by RELATE lists all variables common to the program of both W_i and M_i (after all, W_i was extracted from M_i) and the relationship between each. These variables are listed in two columns, one column for M_i and the other for W_i, with an indicator between the columns. If the indicator is an arrow pointing between the variable in the column of M_i to the corresponding variable in the column of W_i, this symbolises that the data item must be sent from M_i to W_i in order for W_i to operate. This transmission is required when the variable is either input and/or assigned in W_i and referenced and/or output in W_i. Hence an arrow between a variable pointing from the column of W_i to the column of M_i symbolises that the variable must be passed from W_i to M_i and is probably a result variable, being input and/or assigned in W_i and referenced and/or output in M_i. A double sided arrow indicates that the data item has to be sent in both directions (the latter two conditions apply) while no indicator signifies that the data variable does not have to be transmitted as it is not referenced by either M_i or W_i and are therefore local to either of the two.

At this stage of the parallelisation progress, result variables which are passed from W_i to M_i must be examined to determine whether they have any effect on the variables which have to be transmitted from M_i to W_i. If the latter is true, then there is an inter-iteration data dependency and the code must be modified by the SAPF user to eliminate this relationship. If this modification is not possible, then the (M_i, W_i) pair is not suitable as a master/worker pair since some results from the worker for iteration *j* are required by the master in order to produce the work constituting iteration *j+1*.

The workload of W_i can therefore not be farmed out. SAPF may easily be expanded in order to perform the above checking automatically.

A second check for inter-iteration dependency must also be made. This check involves variables which are effected within W_i. Each variable which is both referenced and assigned within W_i must be assigned before it is referenced, and its assigned value must not depend on its previous value or a value remaining as a result from the previous iteration. The occurrence of this case implies that inter-iteration dependency exists and that some results from the worker for iteration *j* are required by the next worker in order to perform iteration *j+1*. The workload of W_i can therefore, in this case, again not be farmed out. This form of analysis may also be performed automatically by a code profiler.

It is useful to note now that if the code cannot be reconstructed to eliminate the inter-iteration dependency and if parallelisation is to continue on the pair (M_i, W_i), some knowledge of the algorithm and code is essential. This is because it may be possible to reconstruct the algorithm, or create a new algorithm, in order to overcome the inter-iteration dependency. For example, in the case of the parallelisation of BSCAT discussed in Section 11, the result of the previous iteration *j-1* was used as a starting point for the following iteration *j*. It was still possible, in this case, to use the result of an iteration *k* where $k \ll j$ and to still converge on a solution, such that a speedup would be obtained if a process farm strategy were adopted.

One minor inadequacy of RELATE, which may easily be removed, is that it does not include relationship checking of all the common blocks. Thus relationships between two routines which share common blocks which are not listed in the body of either M_i or W_i will not be determined. This problem is easily overcome in the current system by ensuring that all the common block definitions appear within both M_i and W_i, ensuring, of course, that there are no overlaps in variable names etc. This will result in SAPF producing a list of all the common blocks used in the entire program as part of the main program body. This creation of all the common blocks within each routine

should also be performed. This ensures that all common block variables are listed in the body of the code and allows RELATE to check hidden relationships between M_i and W_i as well. An example of a hidden relationship is when M_i calls a routine which in turn calls a second routine which then alters a common block variable which is referenced by a routine of W_i. If the common block was not created globally, RELATE will not currently determine this relationship. Although this problem may be overcome, RELATE should be extended to remove this constraint[3].

Thus the determination of dependence is relatively easy when only scalar variables are involved, although for subscripted variables involving expressions the problem, in general, cannot be solved. Fortunately, however, the vast majority of subscript expressions in real programs is linear in the loop variables and not dependent on any other variables.

10.2.6 Deciding on the (M_i,W_i) pair

The relationship finder's output file thus gives an indication to the user of the volume of flow which has to be passed between each (M_i,W_i) pair. If the loop is placed unfavourably, too much data for too little work may have to be passed showing that parallelisation at this point is undesirable. As discussed in Section 6, there are a number of factors present when deciding on the particular parallelisation of a piece of code. The most important of these factors when dealing with a farm structure is the calculation/communications ratio. That is, the ratio of instructions executed by the workers to the volume of data which has to be transmitted to the workers by the master and to the master by the workers. This ratio has a cut-off point which is dependant on the speed of the processors and the data transmission speed between the

[3]It must be noted that the thesis merely set about to determine and verify a method of parallelisation, and not create a final product. AEROTEK has indicated that it will continue development of this system, with the author, to create a fully operational product without any inadequacies.

different processors. This ratio has been discussed and calculated for transputers in Section 6.1.

In order to assist in the decision of determining which (M_i, W_j) pair to select as the most optimal for the farm structure, knowledge of typical input data is required. This will allow the paralleliser to assess the number of iterations that will typically be performed for each (M_i, W_j) pair. The optimal situation for a processor farm will be where the calculation/communication ratio is high, indicating a high volume of calculations to be performed by the worker with a small amount of data transmission. If, however, the number of iterations is low, a suitable speedup may not be obtained since there may not be enough work to keep all the workers busy. For example, if the number of iterations is less than the number of workers available, it is clear that not all the workers will receive work, resulting in a maximum speedup and poor processor efficiency.

Therefore, the paralleliser should attempt to select a pair (M_i, W_j) such that the calculation/communication ratio, as well as the volume of iterations, is high. This may be achieved by selecting the top few candidates, implementing them, and selecting the pair with the best performance. An alternative to this would be to consider the calculations/communications ratio, the volume of iterations, and the time which each master requires in order to generate the next workload. If the master cannot distribute tasks faster than the workers are executing them, then clearly there will be a poor processor efficiency and hence a poor resultant speedup. If the master can create tasks faster than the workers can complete them, and the number of tasks is suitably large, then the pair with the highest calculation/communication ratio is the best candidate.

10.2.7 Multiple Worker Types

Another factor to consider when parallelising code is that the code may consist of a sequence of iterative loops, each of which is suitable for parallelisation. The farm

network may then contain several different types of worker processes on each worker processor, with the type of worker addressed by the master depending on the current iterative loop being executed by the master. The master/worker pair will then be $(M_i, W^1_i, W^2_i ...)$.

10.2.7.1 Example 1

An example of the above case is given in Listing 1 which gives example pseudo code of the sequential and its corresponding parallel master and worker code.

```
      Sequential                      Master                        Worker
      INITIALISE                      INITIALISE                100 GET_WORK(I,M)
      DO 400 I=1,N1                   DO 400 I=1,N1
        DO 100 J=1,N2                   DO 100 J=1,N2               IF (WORK_TYPE(M,J)) THEN
          PERFORM_WORK_J(I,J)             SEND_WORK(I,J)              PERFORM_WORK_J(I,M)
100     CONTINUE                100     CONTINUE
        DO 200 K=1,N3                   DO 200 K=1,N3             ELSE IF (WORK_TYPE(M,K)) THEN
          PERFORM_WORK_K(I,K)             SEND_WORK(I,K)              PERFORM_WORK_K(I,M)
200     CONTINUE                200     CONTINUE
                                        WAIT_ON_RESULTS
        DO 300 L=1,N4                   DO 300 L=1,N4             ELSE IF (WORK_TYPE(M,L)) THEN
          PERFORM_WORK_L(I,L)             SEND_WORK(I,L)              PERFORM_WORK_L(I,M)
300     CONTINUE                300     CONTINUE                  ENDIF
400   CONTINUE                  400   CONTINUE                    GOTO 100
                                        WAIT_ON_RESULTS
      DISPLAY_RESULTS                 DISPLAY_RESULTS
```

Listing 1 Example of Sequential code and corresponding Farm

The procedure SEND_WORK of the master will send the work packet, along with any necessary data, to the first free worker it finds. If no workers are available, it will wait until one becomes available and send the work to it before returning. The procedure WAIT_ON_RESULTS waits until all the workers are idle before returning to the master. The procedure GET_WORK of the worker is used to receive any work packet send by the master, with any extra data that is necessary to perform the task. The function WORK_TYPE is used to determine the type of work received in order that each worker may perform the correct task.

In above example there is also a relationship between the results of the iterations for 100, 200 and 300. Iteration 300 requires the results of 100 and 200 in order to function. Thus, in the parallel version, the master synchronises with the workers after completing the work distribution for the 200 iteration before continuing to distribute the work for iteration 300. This synchronisation may include the exchange of data between the master and its workers. The synchronisation is also performed in order for all the results to be received by the master before they are displayed.

10.2.7.2 Example 2

If the sequential code produces sub-results which are combined to give a final result, each worker process/processor may contain only one type of worker which exclusively produces one form of sub-result. To illustrate this, consider the analogy of the task required to knit a sweater: two sleeves, one body and one neck must be knitted before they can be sewn together to produce the sweater. If the time required to knit the neck is *t*, the time to knit a sleeve is *2t*, the time to knit the body is *4t* and the time to sew the whole lot together is time *t*, then the time it will take for a single person to produce a sweater is

$$t + 2t + 2t + 4t + t = 10t \quad \text{(Neck + Sleeve + Sleeve + Body + Sew)}$$

For the parallel version of this analogy, we could have 4 people knitting the body, 4 people knitting sleeves, one person to sew the neck and one person to sew the lot together. A time analysis of this "production line" is given in the table below where the first column is the time completed, the next three the number of items produced with the final column containing the activity of the person who sews the sweaters together.

As the table illustrates, after time *1t* only one neck is produced, while the other workers, except for the sewer, are still busy. After time *2t* two necks and four sleeves have been completed while after time *3t* only an extra neck is produced. Through all this time the sewer is idle. However, after time *4t* there are four necks, eight sleeves and four bodies. At this stage the sewer can claim a neck, two sleeves and a body to

sew together, so after time *5t* the first sweater is complete. Thus, after an initial startup time of *5t*, the first sweater is created, and after each subsequent time step *t* a further sweater is produced. This operation continues indefinitely since the sewer will never be idle as there is always a stock of sub-results to feed from in order to keep the sewer busy.

Time	Necks	Sleeves	Bodies	Sewer
1t	1	0	0	Idle
2t	2	4	0	Idle
3t	3	4	0	Idle
4t	4 3	8 6	4 3	Collect neck, 2 sleeves and body. Sew.
5t	4 3	6 4	3 2	1st Sweater complete. Collect and Sew.
6t	4 3	8 6	2 1	2nd Sweater complete. Collect and Sew.
7t	4 3	6 4	1 0	3rd Sweater complete. Collect and Sew.
8t	4 3	8 6	4 3	4th Sweater complete. Collect and Sew.
9t	4 3	6 4	3 2	5th Sweater Complete. Collect and Sew.

This example illustrates the case where several specialised workers can exist concurrently in a farm in order to attain the desired goal with a suitable speedup. However, often results cannot be attained in times which are multiples of each other, resulting in some of the workers being forced to be idle in order to prevent them from producing an over-abundance of sub-results. This is one of the types of tasks encountered when load-balancing is performed in order to ensure that processes are placed on processors such that the total time spent idle by all the processors is at a minimum.

10.2.7.3 Summary

The actual parallelisation of code is a complex task. Numerical and analytical code often consist of a series of complex iterative loops. The loops of this code may each be suitable for parallelisation, often resulting in a farm consisting of numerous types of workers, some existing concurrently on the same processor, with others existing exclusively on their own processor in order to attain optimal load-balancing.

SAPF must be used with a suitable code unscrambler and code profiler when parallelising complex code, such as that described above. The code profiler is used to determine the time spent by each section of code, and hence the number of workers of each type which are required, as well as incorporating some of the techniques of [Han89] in order to try and achieve reasonable process placement. SAPF is used to determine the data relationships between the various sections of code.

10.2.8 Implementation

Once the paralleliser has split up the code, determined the data relationships and code profile, the actual implementation of the farm can commence. This is a fairly simple task since the data which must be transmitted between the master and the worker processes is known (from RELATE) and the SAPF Farm structure has the communication routines required for communication, as well as control routines for the management of the workers. The actual lines of code to perform the transmission required are left for the user to write, although SAPF may also be upgraded to include the automatic production of these lines. This is another cosmetic change which will be made to SAPF in the proposed commercial version.

The master therefore consists of the program shell, with the body of the iteration code removed and distributed across several workers. The contents within these iterative sections of the master instead consists of code to distribute the work, receive the results and synchronise the workers. The receipt of results by the master should be

executed concurrently to the code which distributes the work, and should also cater for exception requests such as request for I/O or data from other workers.

The workers are the body of the iterations which are distributed across several worker processors, and may also exist on the same processor as the master in order to increase the processor efficiency. Each worker is often pre-fixed by code to receive the data and work packet required for the iteration, and post-fixed by data transmission routines to return the results to the master. All I/O operations are also replaced by data transmission requests to the master for the appropriate I/O to be performed. Exception data is similarly handled. Data transmission calls and requests for I/O and exception data must be implemented by the paralleliser manually, although another cosmetic improvement to SAPF would be to perform this process automatically. This may easily be achieved since all the data transmission required is already known by SAPF, as well as all worker I/O.

Finally, the paralleliser must also determine suitable logic and termination code for the workers (e.g. when to return the results to the master, which work to perform and when to terminate) and encode this within the master and worker.

10.2.9 MAKECONF

MAKECONF is a program which has been written in Turbo Pascal, the source code of which appears in the Thesis Supplement, to create a standard configuration file for 3L Parallel Fortran. The standard configurer of 3L does function correctly, up to a point, unlike the 3L flood fill farm. The program reads in the output generated by the worm, as well as the flood-fill configuration file which would normally be passed to the 3L flood-fill configurer and produces as output the standard configuration file for the standard 3L configurer. Worker tasks are placed one per processor as well as a single worker task on the master.

The network configuration created is in the format of a tree using the shortest paths algorithm and balancing the tree. Not all the links between processors are placed in this configuration file as only the links required are used. The program is run passing three parameters. The first is the data file produced by the worm, the second is the flood-fill configuration file in the standard 3L format and the final parameter is the name of the output file (the 3L standard configuration file) which is produced by MAKECONF. A forth parameter "/NWOM" may be passed telling MAKECONF that No Worker On the Master is to exist, in which case the configuration file produced will place only the master and its related tasks on the root processor.

The latter /NWOM option is obligatory due to a bug in the 3L configurer concerning the assignment of priorities to processes. This problem causes work distribution difficulties when attempting to create the master of the farm at a high priority, with the worker at a low priority. The master must run at a high priority in order to ensure that tasks always are distributed when there are workers free, while the worker operates when the master process is idle. This occurs when all the workers are busy and the master cannot distribute any further work. However, when placing both master and worker on the same processor, it becomes evident that the worker runs at high priority with the master running at a low priority. This was verified by testing which of the worker processors received tasks, the result of which illustrated that the only worker to receive tasks was the worker on the same processor as the master. This proved that the worker retained control of the processor, preventing the master from distributing work to the remaining free workers. Attempts were made to force the priorities of the separate processes using the URGENT option provided by the 3L configurer and permutating through all the possibilities. All these attempts failed, indicating that 3L did not actually implement this function correctly, if at all, thereby restricting the farm from including a worker on the same processor as the master. However, if the master spawns a thread running at high priority which actually does the work of the master using the 3L thread library, as well as a worker thread, the problem is eliminated. These library routines therefore do allow different priorities to be attained for processes within a processor.

The latter solution to the problem has been omitted from the thesis since it prevents the discovery process of the SAPF farm, described in Section 6.3.1, from determining the communication routes to the individual workers and interfacing through the standard port interface. Instead, internal channels to this worker have to be established, with a flag and the internal channel addresses to this worker being passed to the SAPF communication library for the master. This may easily be implemented as an improvement to the SAPF library.

When a farm is created using one of the worms as a harness, the MAKECONF and configuration step may eventually be eliminated. This is because the worm may be run on any mixed network of transputers, determine the network configuration as before, and also incorporate the methodology of MAKECONF to establish optimal communication paths. Until such a time, however, MAKECONF remains a useful tool in drawing up the 3L configuration files for large transputer farms.

10.3 Summary

In summary then, using SAPF to parallelise iterative code, the following steps are applied:

(a) Structure the code to be parallelised, enabling the transformation into a farm to progress smoother and allowing easier identification of the major iterative and computational loops.

(b) Either manually, or automatically with a dynamic code profiler, create (M_i, W_i) candidates.

(c) Apply SAPF to both M_i and W_i to perform static code checking, resolving any problems encountered, and extracting the variables (both local and global) effected in each.

(d) Apply RELATE to the output of SAPF when applied to both M_i and W_i to determine the data flow between each.

(e) Ensure that each (M_i ,W_i) pair conforms to the requirements laid out in Section 10.2.5.

(f) Determine the best (M_i ,W_i) pair, or group of pairs as described in Sections 10.2.6 and 10.2.7.

(g) Implement the farm as described in Section 10.2.8.

At this point the paralleliser will have the object code for both the master and the worker. In accordance with the SAPF farm library, the master and worker may be configured in any fashion to produce a farm network. Thus, the code for the master and the worker, as well as the worm, MAKECONF, the 3L fixed network configurer and the 3L flood-fill configuration file is all that is required to make the code truly portable to other transputer networks with different configurations. To create bootable code for a target network, the paralleliser must:

(h) Execute the worm on the target network in order to determine the actual network configuration.

(i) Run MAKECONF on the worm output and flood-fill configuration file to create a fixed configuration data file for the farm.

(j) Use the 3L fixed network configurer to create the bootable farm network code.

The code created in (j) may be used repeatedly for the same target network, without having to execute steps (h), (i) and (j) again. However, if the target network were to change, then steps (h) through (j) must be repeated for the new network. However, steps (a) to (g) never have to be repeated unless the farm was restructered.

Thus a method of Semi Automatic Parallelisation of Fortran has been established for generating farm networks from sequential iterative code. In the following section, an example of parallelisation using the method described above is performed.

11 PARALLELISATION OF BSCAT [Sch90a,Sch90b]

11.1 Introduction

BSCAT (Basic Scattering Code) [Mar79] is a user-oriented computer code for the analysis of the far field patterns of antennas in the presence of perfectly conducting metal structures at UHF and above. The code involves a large number of numeric computations which requires a large amount of computing power in order for results to be obtained in a reasonable amount of time. It was decided to parallelise the code due to a number of reasons. Firstly, larger and faster processors are not only expensive, but also not always available to South Africa, be it for political reasons or other. Secondly, the speed of processors are limited by laws of physics with technology eventually reaching a state that processors cannot be made to run any faster, hence dividing the task amongst several processors is the only alternative, when the latter state is reached, to speed up processing time. Thirdly, a transputer is a relatively fast and cheap processor which has the added capability of being able to communicate to other transputers relatively easily using on-chip hardware. Finally, the Division of Microelectronics and Communications Technology (MIKOMTEK) of the CSIR currently builds and sells transputer systems at a fairly reasonable price, therefore providing us with the resources of obtaining a large amount of computing power at a relatively low cost.

BSCAT reads in data from a file to build up the metal structure and place the antenna sources. Minor calculations are performed on the input data in order to translate the data in a reasonable arithmetic representation for calculation purposes. The calculation of the radar pattern, the time intensive portion of the system, then commences. Once the results have been computed, they are output to two files. The first file contains the execution statistics which reports on errors and other components which must be brought to the attention of the user, while the second file contains the actual radar results in a format which is read in by a second program to display the actual radar pattern obtained. Due to the volume of the source code of //BSCAT, it has not been included in the thesis supplement and instead may be obtained by writing to the author directly.

11.2 BSCAT Analysis and Worker Determination

The first phase of the parallelisation involved an analysis of the code. BSCAT was written in over 9000 lines of Fortran, with the parallel version remaining in Fortran to accommodate the engineers who have to perform the necessary maintenance, as well as eliminating the necessity to re-write the code into OCCAM, the mother language of the transputer. The code first had to be structured and modularised, allowing the easy identification of iterative sections of the code. This was achieved by transforming the code using SPAG, as discussed in Section 10.2.1.

From this transformation, a manual dynamic code profile was performed which revealed that the majority of the time spent by the algorithm was spent within four nested computationally loops. These loops are illustrated in Listing 2, where the "******"s denote areas where calculations are performed. The purpose of the loops, identified by the loop variable are:

```
DO 1200 MS=1,MSX
   ***************
   DO 1150 K=KB,KE
      ************
      DO 1100 J=JB,JE
         DO 1100 II=IBP,IEP,IS
            ******************
            ******************
            ******************
      1100 CONTINUE
   1500 CONTINUE
   *************
1200 CONTINUE
```

Listing 2 Computation Loops

- **MS** Loop through the antenna sources
- **K** Loop through the major GTD (General Theory of Diffraction) fields
- **J** Loop through the minor GTD fields
- **II** Loop through each angle of diffraction

The remainder of the code performs input and minor calculations to the data read from the input file and was therefore not considered for parallelisation.

The contents of each iterative loop was then extracted from the code, building four (M_i, W_i) pairs, where i = 1,2,3,4, for the contents of the **MS**, **K**, **J** and **II** loops respectively. It is clear that only one type of worker may be developed within BSCAT. Each pair was then submitted to SAPF in order to determine the data required by

each iteration (data required by each worker in order for it to perform its work), as well as the data required by the remainder of the code from the iteration (the results of the worker to be transmitted back to the master). This method of this analysis has been discussed in Section 10.2.

11.3 Selection of a (M_i ,W_i) Pair

Each worker pair above was examined in turn for suitability as a farm. The elements of each worker pair noted when making the choice were:

a) The degree of parallelisation should not be so fine that:
   i) The majority of time spent by the farm was the transmission of data.
   ii) The volume of computations performed by each worker is so small that a small percentage of their time is actually spent calculating.

b) The master can produce work at a rate which is faster than the rate the work can be completed by the workers. Thus course grain parallelism, where a high percentage of the time spent by each worker is non-productive (idle), must be avoided.

c) There is enough work in total such that all the workers receive at the very minimum one task.

This immediately ruled out the (M_1 ,W_1) worker pair, since **MSX** of the **MS** loop represents the number of antenna sources, typically only one. This would result in only one task being generated for the average run. Similarly, the (M_2 ,W_2) pair could be eliminated, since the **K** loop would be executed **(KE-KB+1)** times, where **(KE-KB+1)** could only be a maximum of 3. This also applied to the (M_3 ,W_3) pair since the maximum number of iterations provided by the **J** loops would be the maximum of **(JE-JB+1)**, which was also 3. Thus if (M_3 ,W_3) was selected, a maximum of **9*MSX** tasks could be generated for the farm, an average of only 9 tasks. Thus, it appeared that (M_4 ,W_4) was the best choice, using the contents of the **II** loop. The **II** loop

typically produced 360 iterations, looping through 360 degrees, determining the value at every degree. The total number of tasks then produced by the (M_4,W_4) farm is then

$$\frac{MSX \cdot (KE-KB+1) \cdot (JE-JB+1) \cdot (IEP-IBP+1)}{IS} \qquad (24)$$

W_4 fortunately performs a high volume of calculations, so was a valid choice for a worker. However, it was also noted that no calculations are performed between the **J** and **II** loops. This indicated that these two loops could be interchanged and placed in the format illustrated in Listing 3. This would reduce the total tasks of the farm by a factor of **(JE-JB+1)** and ensure that the network does not become flooded with too many work packets. Thus, a new master/worker pair (M_5,W_5) was created where W_5 contains the **J** loop and M_5 the **II** loop. If this pair later results in making the grain of parallelism too large, the original structure and master/worker pair (M_4,W_4) could be reinstated.

```
DO 1200 MS=1,MSX
      ************
      DO 1150 K=KB,KE
            ************
            DO 1100 II=IBP,IEP,IS
                  DO 1100 J=JB,JE
                        ************
                        ************
                        ************
            1100 CONTINUE
      1500 CONTINUE
      ************
1200 CONTINUE
```

Listing 3 Adjusted Loops

Note that the code transformation above should be performed if there was an inter-iteration dependency within the **J** loop, for the corresponding values of **II**. That is **J=1,2,3** for **II=1**, **J=1,2,3** for **II=2**, **J=1,2,3** for **II=3** etc. where each worker operating on a value of **II** would have to perform the work for all the values of **J** since they are related. This would also be the case if subsequent calculations for the next **J** for a particular value of **II** could only continue once the calculations using the previous **J** were complete, due to inter-iteration dependency, and also resulting in some synchronisation. However, this is not the case in BSCAT and the transformation merely reduced the total workload.

11.4 Data Communication

11.4.1 Work Packet

Using the (M_5,W_5) worker pair as the farm candidate, they were each submitted to SAPF, with the pair of output files in turn being submitted to RELATE which produced the output displayed in Appendix C(i), an extract of which appears below. The **J** loop is thus internal to the worker, while the **II** loop resides on the master. All the variables listed with bi-directional arrows (indicating that they must be passed from the master and worker, and back to the master once the worker has completed its task) were individually checked as they would normally indicate inter-iteration dependency. Each of these were found to be either variables within a common BLOCK DATA routine, or were variables which were temporarily assigned and hence are not required to be transmitted.

BSCAT	transfer method	WORKER
A	=>	A
B	=>	B
CJ	<==>	CJ
DPR	<==>	DPR
DTS	=>	DTS
EPHT	<==>	EPHT
ETHT	<==>	ETHT
II	=>	II
J	<==>	J
JB	=>	JB
JE	=>	JE

Extracts of Appendix C(i)

After examining Listing 3, three main regions were found where data, altered by the master M_5, may be required by the workers in order to perform their tasks. These were:

(1) The data read in and effected by calculations performed prior to the **MS** loop.

(2) The data altered by calculations performed before the **K** loop is entered within the **MS** loop.

(3) The data modified by calculations within the **K** loop before the **II** loop is entered.

Each work packet should contain all the variables indicated by Appendix C(i), producing a large work packet size, not a desirable situation. However, the work packet need not consist of all the data of Appendix C(i) as most of it remains unaltered within the **MS** loop. Thus further data flow analysis using SAPF and RELATE was performed between the variables affected in (1), (2) and (3). The data variables effected between (1) and (2) above and their master/worker relationship are listed in Appendix C(ii) and have been named the "Pre-Loop" variables. The few data variables effected between (2) and (3) are listed and described in (iii) below. This indicated that data must be sent to the workers in three stages:

(i) All the variables effected in (1), excluding those variables effected in (2) and (3), are broadcast to the workers just before the **MS** loop is entered. These variables are transmitted once to all the workers.

(ii) All the variables effected in (2), excluding the variables effected in (3). These variables are broadcast to all the workers every time **MS** is changed.

(iii) The variables effected in (3) are transmitted along with the work packet to each worker. These are the three variables **JB**, **JE** and **THP**. These variables were not broadcast since the inclusion of code required and the resultant synchronisation may effect the speedup. It was thus decided to include this data with the work packet. Also, if there are more workers than work packets for the **II** loop, some of the workers may utilise these variables. With this strategy, the work packet becomes **(II,JB,JE,K,MS,THP)** where the size is 24 bytes.

Note that **MS** may be omitted from the work packet as it could have been broadcast with the data from (ii). It has been included in the work packet for completeness in order that it include all the iterative variables. It was also included to act as a farm command since **MS** may never be below zero, so workers receiving negative values of **MS** could interpret the value as a specific request from the master.

A suitable parallelisation point has therefore been achieved where a large amount of computations are performed for a small amount of data sent. As given by the

RELATE output, the following variables and common blocks (in *italics*) are transmitted at each of the above three stages.

Stage (i)
BLR, FACTOR, FNP, *GEOMEL*, *GROUND*, JB, JE, LCNPAT, LDC, LDRC, *LOGDIF*, LOUT, *LPLCY*, LRDC, LRFC, LRFI, LRFS, *LSHDP*, LSOR, *PATDAT*, *TEST*, THP, TPPD

Stage (ii)
BNDDCL, *BNDFCL*, *BNDICL*, *BNDRCL*, *BNDSCL*, *DOUBLE*, *FARP*, *GEOPLA*, *IMAINF*, *IMCINF*, *LSHDT*, LSRFC, LSURF, MPH, PHWR, *SORINF*, VMAG, WI

Stage (iii)
II, JB, JE, K, MS, THP

11.4.2 Exception Data

Each worker cannot perform any I/O as it is not directly connected to a host PC. Therefore, any I/O which is required by workers must be handled by the master. Instead of performing I/O, the workers therefore send the master messages specifying which I/O operations are to be performed along with any data which the master requires. Furthermore, in some cases the master responds to the worker's exception request. This complicates matters slightly as messages received from the workers are therefore either destined for I/O or for the master shell. This problem was overcome by creating a routine analogous to F77_NETW_RECEIVE of Appendix A. This routine, RECEIVE_NETWORK, has the same parameters as F77_NETW_RECEIVE, and receives messages from the workers in the same manner. If it receives any messages bound for I/O, it executes the required I/O operations, and then continues to wait for further messages. It only returns to the master when messages bound for the master shell are received. Messages from

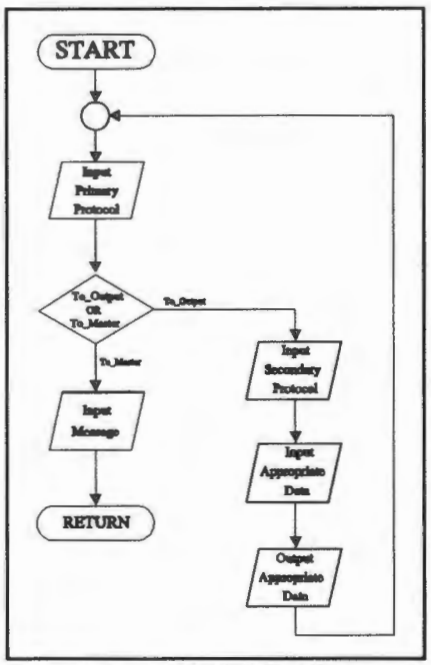

Figure 30 RECEIVE_NETWORK flowchart

workers are therefore prefixed by a primary protocol, listed in Appendix C(iv), specifying whether the message is I/O bound or destined for the master shell. Workers request unique access of the network through a call to F77_NETW_USE, send the primary protocol tag (either **ToMaster** or **ToOutput**) followed by the message/s to the master shell or for output respectively. If the tag **ToMaster** is received, the next message received is returned to the master which initiated the call to RECEIVE_NETWORK. If the tag **ToOutput** is received, a secondary tag specifying what output function is to be performed is expected, as well as the data required for the output. RECEIVE_NETWORK then performs the output operation requested before returning to the initial state of waiting for the primary protocol. The flowchart for RECEIVE_NETWORK is given in Figure 30.

The exception messages are therefore hidden from the master shell which calls RECEIVE_NETWORK instead of F77_NETW_RECEIVE, the former being called in place of the latter. The list of values used as protocols is shown in Appendix C(iv).

11.4.3 BSCAT Results

In order to determine the results produced by each worker, the section of code of M_5 following the **1200 CONTINUE** statement of Listing 3 was run through SAPF and the output file, along with the output of W_5, used as input to RELATE. This is the data which is required from the "Post-Loop" calculations. The resulting communication required is listed in Appendix C(iii). After examining the code and the variables listed, it became evident that all of the variables, apart from the complex arrays **EPHT** and **ETHT**, were used temporarily and thus transmission of these variables is not required. Furthermore, the complex arrays **EPHT** and **ETHT** (the contents of which are all initially (0,0)), simply keep a running total throughout all the loops. Since it is only these variables which are required as results, the workers themselves should keep their own local running totals of these variables and return them as their final results. The final results of all the workers can then be cumulated to produce a single final result.

Thus, the result of each work packet are the complex variables arrays **EPHT** and **ETHT**. The master then applies simple vector addition to all the local arrays returned by the workers to produce a final result, equivalent to the result of the sequential version. Each position of the arrays thus corresponds to the result of each angle **II**, for each loop through **MS**, **K** and **J**.

This latter method is more suitable and makes more sense than each worker returning its result on completion of each task. The main reason for this is that data communication is minimised to the master from the workers with the workers only returning one result. This is analagous to adding a series of numbers together where the numbers are distributed amongst workers where they are totalled, producing subtotals, which the master merely sums to produce a grand total. Imagine the communication load if each worker was merely required to add two numbers and return the result, which the master would then have to add to its total.

With the master/worker pair selected and the data communication determined, the parallelisation of BSCAT into a farm continues into the development of a master and worker.

11.5 The Master

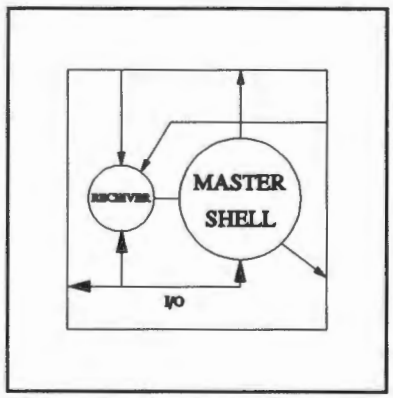

Figure 31 Master Processor

The tasks which constitute the master are illustrated in Figure 31. There is a receiver process which is created by the master which is responsible for concurrently collecting and collating the results from the workers, while the master is responsible for distributing the work. The receiver also handles any exceptions which may occur through its use of the routine RECEIVE_NETWORK of Section 11.4.2, listing also the worker where the exception developed. Note that there is a connection between the master and receiver, but that this is not a communication link. This is

because communication between these two processes is achieved through semaphores, as described in Section 6.3 and Appendix A. The additional semaphore **Results_Sema** is used by the receiver to indicate to the master that all the results have been returned. A detailed description of the Master process may be found in Appendix C(v).

11.6 The Worker

The worker consists of the **J** loop and its contents. Once it has initialised the **EPHT** and **ETHT** arrays, it waits for the broadcasted Stage (i) variables from the master. After these variables have been received, the worker waits for **MS**. As described in Section 11.4.1, the value of **MS** may never be below zero, so the master sends specific negative values of **MS** to the workers which it interprets as different commands. If the value of **MS** is the negative value **Update**, as described in Appendix C(iv), the worker should then expect the Stage (ii) variables. If **MS** was the negative value **Results**, then the worker sends its local copies of **EPHT** and **ETHT** to be added to the master's totals, and resets itself by waiting for the next set of Stage (i) variables for the next set of work. If, however, the value of **MS** is valid, the remainder of the Stage (iii) variables are input and the worker performs its next task. A full description of the worker is given in Appendix C(vi).

11.7 Results

The worker and master processes of BSCAT were retained in object code, with boot files being created by the system for each network configuration used as described in Section 10.2.9. Several different configurations were attempted, all of which were successful.

Thus the inner computation intensive loop was successfully divided among multiple workers, with exciting speedup times being obtained. The parallel farm version of

BSCAT (//BSCAT) was applied to a known problem. This was an example of a Boeing 737 with a single antenna source, and hence was a simple problem with a fairly small workload (a more complex problem would result in a larger degree of speedup). The timing results (excluding network boot time) for the //BSCAT farm, taken over an average of 10 runs using the transputers internal clock, are illustrated in Table V. The estimates included were calculated by extrapolating a best-fit curve for the results obtained, allowing predictions to be made regarding timing, performance and efficiency when increasing the number of worker processors. The formula used to approximate the actual time is given by Equation (25) where n is the number of processors.

BSCAT
Boeing 737 Example 7
Actual and Estimated Time, Speedup and Efficiency
Time in Seconds — Workers on T800-20 with 1Mb each

Number Workers	Actual Time	Est. Time	Actual Speedup	Est. Speedup	Actual Eff.	Est. Eff.
1	200.38	200.37	1.00000	1.00000	1.000000	1.000000
2	101.09	101.10	1.98219	1.98189	0.991095	0.990942
3	68.03	68.08	2.94547	2.94325	0.981823	0.981083
4	51.60	51.60	3.83333	3.88323	0.958333	0.970807
5	41.72	41.73	4.80297	4.80149	0.960594	0.960297
6	35.31	35.17	5.67488	5.67800	0.945813	0.949666
7	30.53	30.48	6.56338	6.57290	0.937626	0.938985
8	26.85	26.98	7.46294	7.42645	0.932868	0.928305

Table V Results

$$t_{//BSCAT} = \frac{Parallisable\ Time}{n} + \frac{Communication\ Time}{\ln(n)} + Overhead\ Time \qquad (25)$$

$t_{//BSCAT}$ thus consists of the time spent in the code which is parallelisable (this may be assumed to be a linear speedup), plus the communication overhead time (where a tree configuration may be assumed), plus the actual overhead time. The best-fit values for these three times obtained were:

Parallelisable time	=	199.0703
Communication time	=	0.382997
Overhead time	=	1.300500

It may be appropriate to note that the time taken on a single T800, for this example, was 200.38 seconds with the execution time obtained for a μVAX for the same input data was 218.53 seconds. This illustrates that the T800, by itself, is a powerful

processor for executing sequential code, even with no code parallelisation or modifications.

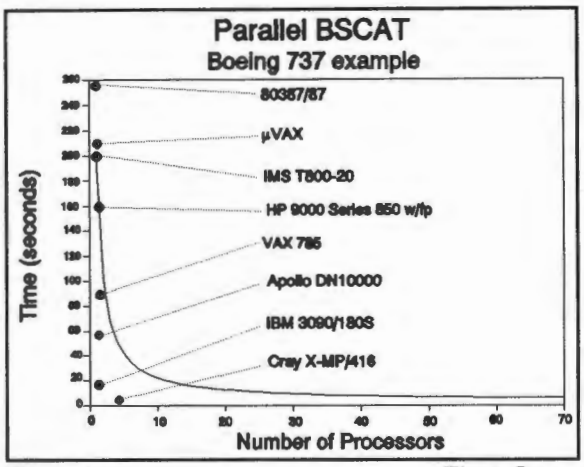

Figure 32 Time Curve

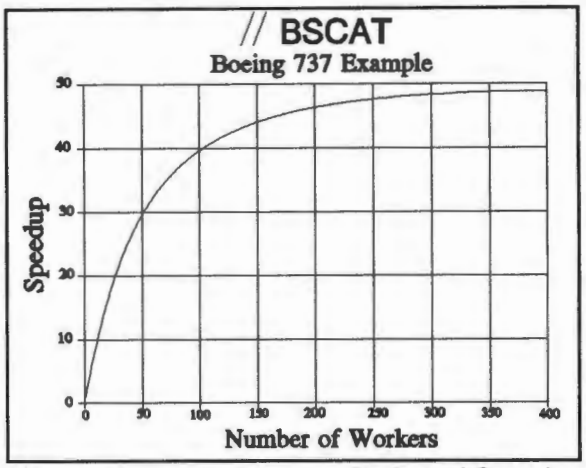

Figure 33 Projected Speedup

The best fit for the time obtained was extended for up to 100 workers and is illustrated for up to 70 workers in Figure 32 by the solid curved line. The circles represent the timing which would be obtained if BSCAT were to be executed in Fortran on the computers indicated. These figures were calculated using known benchmarks [Don89] which were then applied to the relevant computers in order to obtain approximate values.

In projecting the speedup for //BSCAT using the Boeing 737 example, the graph illustrated in Figure 33 was obtained. Note that the graph curve levels out at ±350 workers, which, for an input of one source through 360 degrees conforms almost exactly to the speedup which could be expected. This is due to the fact that only 360 work packets could be generated and hence a maximum of ±360 processors may be kept busy with the remaining processors lying idle throughout the operation. The reason seen for the levelling out at ±350 may be attributed to the communication overheads which are incurred.

Note that the speedup also never exceeds the 50 mark (50 times faster than the sequential version). This is due to the parallelism of the problem as there will always be code which must be executed sequentially for any algorithm, resulting in a maximum speedup for the algorithm. This is known as Amdahl's law. To illustrate, if an algorithm is broken down into n primary (indivisible) operations, each requiring time t to execute, and each of these operations is executed in parallel by n processes (not that this would make any sense), then the program will never execute in parallel fast than the time t. If $n+1$ processors were applied to the algorithm, the $n+1$th processor would not have any operation to perform, other than a redundant duplicate operation.

11.8 Conclusion

The parallelisation of BSCAT was successful, with the results and execution times of several different input sets comparing favourably to the original sequential version with the same input sets. Thus super-computer power may be obtained at the fraction of the normal cost.

Due to an initial error, it was later found that the results obtained from iteration i of the algorithm are carried to iteration $i+1$ to allow for a faster convergence, and hence faster execution. However, it was found that workers using their previous iteration i results in converging on a solution for iteration $i+x$, where x is the number of packets distributed between receiving the i and $i+x$ work packets, would still converge in just as fast a time. Thus the "outdated" data from iteration i was found to be a suitable starting point for the next iteration received. If this effect resulted in a poor speedup, the speedup could be increased further by creating a group of consecutive work packets using consecutive iterations which would be sent to the same worker, giving each worker the opportunity of using the latest information as the starting point for a limited number of work packets.

12 CONCLUSION

This thesis demonstrated successful semi-automatic parallelisation of iterative code. It provides methodology and support for the parallelisation of some forms of sequential iterative code, reducing substantially the quantity of manual work normally required to produce a parallel version. Several advances were made in the field of parallelisation and in the software support field for transputers. A number of tools were developed, including a transputer worm, a Turbo Pascal server, a static code and data flow analyser as well as an automatic farm configurer.

The first tool developed was the transputer worm which allows generic farm code to be produced which may execute on any transputer network complying to the INMOS standard, as well as a few others which can be remotely reset. This reduces the developer's concerns regarding different topologies, since the network topology is determined automatically. A suitable farm network configuration for the target transputer network is produced from the results of the worm's search. This ensures that farm communication follows the shortest route efficiently, automating the parallelisation task even further. The worm may also be extended to provide an OCCAM harness for a farm network written in Fortran, C or any other language, thereby eliminating the inadequacies and problems of the 3L farm, and incorporating automatic routing strategies. The worm is a useful stand-alone tool and has been released into public domain where it is used to determine actual transputer network configurations.

The Turbo Pascal Server was another aspect of the thesis which benefitted the field of parallelism with respect to transputers. Its objective of providing the OCCAM programmer with all the utilities and facilities of the IBM PC provided by Turbo Pascal V5.5, in particular the PC's graphics, was achieved with great success. It is currently marketed as a stand-alone product, in full source format, with many well-known companies in South Africa utilising the server in their research. A mark of its success, value and flexibility.

SAPF, the Semi Automatic Paralleliser of Fortran, was developed in order to reduce the level of understanding of the sequential iterative code and work required in order to parallelise the code into a farm format. This was successful and reduced the time spent in parallelising such algorithms from months to weeks. It also decreased the element of human error considerably, not only as a static code and data flow analyser, but also by performing the rigid type checking, found in structured languages such as Pascal and Modula-2, on Fortran code. This feature alone discovered problems in well utilised sequential code. The SAPF farm routines also provide a good foundation from which many farm networks can and have been developed. These routines are also being marketed as an individual product, mainly as a substantial advancement over the 3L farm, but also as a solution to the 3L bug.

The user involvement is SAPF is not unwarranted since minor modifications to the code structure may have to be made in order to make the algorithm more amenable to parallelisation. In order to produce an optimal parallel version, the user may also be required to reduce inter-iteration data dependencies and introduce alternative algorithms, such as those found and resolved in BSCAT and the Airflow model.

Typical results achieved through parallelisation using SAPF, its tools and the worm reflected high degrees of speedup. This speedup was only limited by Amdahl's law which states that the speedup which may be attained is always limited by the series of instructions which must be executed sequentially. The speedup obtained in the farms generated through SAPF produced gratifying increases in the speed of execution, in orders of magnitudes faster. Both the Airflow Modelling problem and BSCAT were successfully parallelised and proved to be good test examples, validating the communication, load balancing and parallelisation techniques developed in the thesis. Figure 33 shows that the speedup obtained by adding more processes to //BSCAT is well related to the number of iterations (workload) available, illustrating Amdahl's law. The resultant speedup observed for BSCAT, by far the larger of the two test examples, correlated closely to the predictions made, reflecting a high degree of confidence in achieving the projected speedup for larger networks.

This thesis opened up a number of research areas in the development of a semi-automatic paralleliser for Fortran code, as well as support tools for transputers and transputer networks. Future development of SAPF has already been planned where the improvements and cosmetic changes recommended are to be implemented. This development includes the automatic generation of code for farm communication (including the handling of worker I/O and exceptions), a dynamic code profiler to extract the iterative loops and automatically submit them to SAPF, as well as modifications to the server to make it IServer compatible in order that other languages on the transputer may also benefit from its services. Development of the automatic code generation and IServer modifications are already under way. These changes and extensions will then provide the paralleliser with a series of farms, with the corresponding data flow and workload, from which a suitable choice can be made. The farm finally selected will then be integrated into the worm, producing a truly generic farm code which can be executed on multiple network topologies without any modifications.

Bibliography

[3LL88] 3L Ltd and EPCL, *3L Parallel Fortran User Manual -Version 2.0*, 17 June 1988

[Axe86] T.S.Axelrod, *Effects of synchronisation barriers on multiprocessor performance*, Parallel Computing, Vol 3, pp. 129-140, 1986

[Bab88] R.G.Babb II, *Programming Parallel Processors*, Addison-Wesley, ISBN 0-201-11721-5, 1988

[Beh86] P.M.Behr, W.K.Giloi and H.Mühlenbein, *SUPRENUM: The German supercomputer achitecture- Rationale and concepts*, IEEE Proceedings of the 1986 International Conference on Parallel Processing, pp. 567-575, 1986

[Bow87] K.C.Bowler et al, *An Introduction to OCCAM 2 Programming*, Chartwell-Bratt, ISBN 0-86238-137-1, 1987

[Bur88] A.Burns, *Programming in OCCAM 2*, Addison-Wesley, ISBN 0-201-17371-9, 1988

[Don89] J.J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*, Technical Memorandum No.23, Mathematics and Computer Science Division, Argonne National Laboratory, June 1989

[Fab86] V.Faber, O.M.Lubeck and A.B.White, *Superlinear speedup of an efficient sequential algorithm is not possible*, Parallel Computing, Vol 3, pp. 259-260, 1986

[Fab87] V.Faber, O.M.Lubeck and A.B.White, *Comments on the paper "Parallel efficiency can be greater than unity"*, Parallel Computing, Vol 4, pp. 209-210, 1987

[Han89] C.Handler, *Parallel Process Placement - M.Sc Thesis*, Rhodes University, January 1989

[Hay87] A.J.Hayzen, *Report NBRI-1 to NBRI-2*, CSIR Pretoria, 1987

[Hay88a] A.J.Hayzen, *Report NBRI-3 to NBRI-4*, CSIR Pretoria, 1988

[Hay88b] A.J.Hayzen, *Personal Communication*, CSIR Pretoria, 1988

[Hen90] M.Henning, *Personal Communication*, University of Natal, February 1990

[Hoa85] C.A.R.Hoare, *Communicating Sequential Processes*, Prentice Hall, ISBN 0-13-153271-7, 1985

[Hon90] M.A.Honman, *Multiple-Language Programs under the OCCAM Toolset*, Internal Report DAST 90/26, AEROTEK - CSIR, Pretoria, South Africa, January 1990

[INM84] INMOS, *OCCAM Programming Manual*, Prentice Hall, ISBN 0-13-629296-8, 1984

[INM87a] Technical note 6, *IMS T800 Architecture*, INMOS Bristol, March 1987

[INM88a] INMOS, *Transputer Reference Manual*, Prentice Hall, ISBN 0-13-929001-X, 1988

[INM88b] INMOS, *OCCAM 2 Reference Manual*, Prentice Hall, ISBN 0-13-629312-3, 1988

[INM89] Technical note 47, *The role of OCCAM in the design of the T800*, INMOS Bristol, 1989

[Jan87] R.Janβen, *A note on superlinear speedup*, Parallel Computing, Vol 4, pp. 211-213, 1987

[Kri89] E.V.Krishnamurthy, *Parallel Processing: Principles and Practice*, Addison-Wesley, 1989

[Mac89] J.W.Machar, *Personal Communication*, 3L Ltd, Livingston, U.K., August 1989

[Mar79] R.J.Marhefka and W.D.Burnside, *NUMERICAL ELECTROMAGNETIC CODE (NEC) - BASIC SCATTERING CODE (Part I and II)*, Ohio State University, Technical Reports 784501-18, 784508-14, September 1979.

[McG88] J.R.McGraw and T.S.Axelrod, *Exploiting Multiprocessors: Issues and Options*, Programming Parallel Processes, Addison-Wesley, pp. 7-25, ISBN 0-201-11721-5, 1988

[Par86] D.Parkinson, *Parallel efficiency can be greater than unity*, Parallel Computing, Vol 3, pp. 261-262, 1986

[Pin88] A.Pinder, *Personal Communication*, INMOS, Bristol, U.K., July 1988

[Pou84] D.Pountain, *Microprocessor Design*, BYTE, pp. 361-366, August 1984

[Sch89] A.M.Schuilenburg, M.SC *Technical Presentation*, AEROTEK - CSIR, Pretoria, South Africa, August 1989.

[Sch90a] A.Schuilenburg, *Parallelisation of BSCAT*, AEROTEK - CSIR, Pretoria, South Africa, Internal Report DAST 90/12, January 1990

[Sch90b] A.Schuilenburg, *Parallelisation of BSCAT*, IEEE AP/MTTS-90 Proceedings, pp. 259-266, August 1990

[She89] D.Sherwell, *Personal Communication*, CSIR, Pretoria, South Africa 1989

[Zim88] H.P.Zima, H.J. Bast and M.Gerndt, *SUPERB:A tool for semi-automatic MIMD/SIMD parallelisation*, Parallel Computing, Vol 6, pp. 1-18, 1988

Appendix A - SAPF Routine Descriptions

1 Master Routines

The routines described here are the routines which may be called by the master process of a SAPF farm.

1.1 SUBROUTINE F77_NETW_MASTER()

Description

This subroutine must be called in order for the master to initialise the farm network as discussed in the previous section. If this is not performed before any of the master library routines below are called, the farm network will not be able to function correctly.

Operation

The routine initialises the channels, establishes the number of workers available, identifies them and determines the path to be taken by messages to each worker using the method described in Section 6.3.1. The messages are packet switched to each worker and hence are prefixed by the path which is used as the header. The routine also initialises the semaphores and variables which the master may use.

In the initial implementation, each of the master routines below would check an initial flag to ensure that the initialisation had occurred. If this were not the case, the initialisation would commence. This was later excluded in favour of speed of execution since initialisation occurs only once and for each communication routine to ensure initialisation results in excessive redundant checking. The single calling of this initialisation routine is left to SAPF or the implementer of the network.

1.2 INTEGER FUNCTION F77_NETW_NUM_WORKERS()

Description

This integer function returns the number of workers found in the farm.

Operation

The value of the common block variable containing the number of workers is returned. The common block is not normally accessible outside of the library.

1.3 SUBROUTINE F77_NETW_BROADCAST(Length, Buffer)

Description

This subroutine will broadcast the contents of *Buffer* to all the workers in the network. *Buffer* may be any variable, from a simple integer to multi-dimensional complex arrays. The size of *Buffer* in bytes must be contained in the 32 bit integer *Length* and must not exceed the value of F77_NETW_MAX_PACKET_LEN, otherwise this may result in the communication network to fail.

Operation

The broadcast header is sent with the message contained in *Buffer* down each of the outgoing channels to the workers. The broadcast header consists of a negative header length which has the implicit connotation of no header bytes as well as for a copy of the message to be sent to the current worker with further copies distributed along all outgoing links. As discussed previously, messages may not exceed F77_NETW_MAX_PACKET_LEN bytes in size, otherwise network communication failure may result.

1.4 SUBROUTINE F77_NETW_SEND(Length, Buffer, Worker)

Description

Length is a 32 bit integer constituting the number of bytes of *Buffer* which are to be transmitted to the worker with the ID *Worker*. *Buffer* may be any variable where the size of it in bytes is contained in *Length*. *Length* may also not exceed

F77_NETW_MAX_PACKET_LEN otherwise network communication failure may result.

Operation

The output channel number is obtained and removed from the header, decreasing the header size. The new header size, followed by the remainder of the header, the *Length* of buffer, and *Length* bytes of *Buffer* are then transmitted along the relative output channel. As previously mentioned, the initial implementation of this routine would split up packets greater than F77_NETW_MAX_PACKET_LEN into more than one smaller sized packets and distribute these to the necessary worker. The current implementation requires the implementer or SAPF to split up larger packets (as does the 3L farm library), ensuring that message packet sizes fall within the correct range, since it was found that over 90% of messages sent were less than F77_NETW_MAX_PACKET_LEN in size. The exclusion of this option resulted in faster communication due to the removal of over 90% of redundant checking previously required to ensure that the size of messages fell within the correct range.

1.5 SUBROUTINE F77_NETW_RECEIVE(Length, Buffer, Worker)

Description

This subroutine will wait for any data to be received from any incoming channel, and hence any worker. The data received is placed in the variable *Buffer* with the length of bytes received and worker ID which created the message returned in the two 32 bit integer variables *Length* and *Worker* respectively.

Operation

In normal operation when the network is not reserved, this subroutine monitors all incoming channels for messages to the master. This is the monitoring stage. If any activity is detected, it inputs a worker ID. If the ID is positive, the 32 bit integer message length *Length* is read from the channel, followed by a message of *Length* bytes which are stored in *Buffer*. The subroutine then exits.

If the ID is negative this is regarded as being a request to reserve or release communication to the master. If communication is not reserved, it is reserved for the worker whose ID is the absolute value of the ID received with the subroutine returning to the monitoring stage. If the master is in the monitoring stage and network communication is reserved by a worker, only the channel coming from the worker reserving the network is monitored, blocking any message traffic from the remaining incoming channels. The reserving worker therefore has exclusive communication rights to the master. If the network is already reserved and a negative ID is received from the monitoring stage, this request must have come from the reserving worker. The negative ID therefore indicates that the worker wishes to release the network, allowing the remaining workers to communicate to the master as well. The communication network is then released by the subroutine which again returns to the monitoring state of monitoring all incoming channels.

1.6 INTEGER FUNCTION F77_NETW_RESERVED()

Description

This integer function returns the worker ID of the worker which has the network reserved for communication purposes. An ID of zero is returned if the network is not reserved by any workers.

Operation

The value of the common block (not available outside the library) variable containing the worker ID of the worker which has the network reserved is returned.

1.7 INTEGER FUNCTION F77_NETW_FREE_WORKERS()

Description

The master library also provides functions and subroutines whereby the workers may be managed. These management routines can identify and claim idle workers in order to distribute work to them as well as making the workers available again once they

have performed their required tasks. If the management routines are used, this function will return the number of idle workers. All the workers are marked initially as being idle and are claimed through the function F77_NETW_USE_WORKER and released by the subroutine F77_NETW_FREE_WORKER.

Operation

Shared variables are used by the worker management routines to keep track of the number of idle workers and which of the workers have been claimed. Since these variables may be altered by concurrent sections of code, a semaphore is used to protect and ensure unique access to these common variables. This function therefore waits on (reserves) the semaphore, makes a copy of the variable used to store the number of workers which are available (contained within the common block), signals (releases) the semaphore and returns the value of the copy as the result.

A second semaphore, *Workers_Free*, is used to manage the number of free workers. Its purpose is to prevent the busy wait which the function F77_NETW_USE_WORKER will have to perform if all the workers are busy and an idle worker has been requested. Although it may be accessed instead of the shared variable, the implementation of semaphores may not always guarantee that the value of the semaphore is accessible. The 3L parallel libraries implement semaphore variables as an integer array and do not provide a method whereby the value of a semaphore may be returned, although testing has shown that this value can be found in the first array position of the semaphore variable. Therefore an autonomous variable is used in the common block to administer the number of idle workers.

1.8 INTEGER FUNCTION F77_NETW_USE_WORKER()

Description

This function returns the ID of a worker which is marked as idle and indicates it as being active. Initially the library routines have all the workers marked as idle and may return any worker ID. After some workers have been marked as active, this function

will return the ID of the first worker it finds marked as inactive. Therefore this function is called in order to reserve a worker for a task. If no workers are available, the function waits until at least one does become open and will return the with an idle worker's ID. The workers are marked as free by a call to the subroutine F77_NETW_FREE_WORKER. When F77_NETW_USE_WORKER is used in conjunction with F77_NETW_FREE_WORKER an effective worker management system will result where the possibility of sending work to a worker which is already active with is eliminated. Furthermore, this management system ensures that the workers may be maximally utilised, provided the master can create tasks faster than the workers complete the tasks.

Operation

The function first waits on the semaphore *Workers_Free*, attempting to decrease it by one to a non-negative value. This semaphore, set to the number of workers found on initialisation, allows the busy wait loop to be avoided by inducing the function to wait until a worker is freed by F77_NETW_FREE_WORKER if no workers are idle on entry to the function. This done using the 3L parallel fortran library routine F77_SEMA_WAIT. The busy wait would involve the function constantly looping, achieving nothing, until the variable containing the number of free variables becomes greater than zero.

Once *Workers_Free* has been decremented, indicating that a worker is free, the semaphore protecting the shared variables is also waited on. When access to the shared variables has been granted, the variables are inspected to determine which worker is available and this worker marked as active. The variable storing the number of free workers is also decremented before the semaphore protecting the shared variables is signalled in order to release these shared variables. The ID of the worker found to be inactive and marked as active is then returned as the result of the function.

1.9 SUBROUTINE F77_NETW_FREE_WORKER(Worker)

Description

The subroutine marks as inactive the worker whose ID is passed in the integer variable *Worker*. This frees the worker so that it may again be claimed by F77_NETW_USE_WORKER and furnished with another task. If this function is used in conjunction with F77_NETW_USE_WORKER it will provide an effective book-keeping system of the workers available and busy.

Operation

Firstly, the semaphore protecting the shared variables is waited on to ensure unique access. Once this is granted, the worker is checked to see if it is not already marked as free, in which case the subroutine exits, after unlocking the shared variables, without performing any further operations. If the worker is marked as active, it is reset as being inactive and the variable containing the number of free workers incremented. The shared variables are then unlocked and the *Workers_Free* semaphore signalled (incremented).

At this point it can be noted that the ID of free workers may be kept on a stack and may be popped off the stack by F77_NETW_USE_WORKER in order to be claimed and replaced by this routine. The stack initially must contain the IDs of all the workers. The actual implementation, however, operates on a round robin system to ensure all the workers are sent tasks, not merely the workers nearer the root which may receive the work and return the results fast than transputers further away. This routine thus expects workers to be freed in the order they were claimed (a reasonable assumption) and merely scans through the list for free workers.

2 Worker Routines

The routines described here are the routines which may be called by a worker process of a SAPF farm.

2.1 SUBROUTINE F77_NETW_WORKER()

Description

Similar to the master processor routine F77_NETW_MASTER, this routine is used to initialise the routing tasks, channel communication variables and other variables which will be used by the message routing libraries of the workers. Each worker must therefore call this routine when initialising otherwise the farm network will not be able to function correctly.

Operation

The routine initialises the channels, and then begins the network communication initialisation process as described in Section 6.3.1. This involves waiting for its own ID *IDmine* from the input channel from the master and returning the value of $ID_{mine}+1$ with a null path down the output channel to the master. Thereafter the worker waits for further IDs from the input master channel which it forwards continuously down each of output channels to the next generation of workers. If the corresponding input channel returns a negative ID_{result}, the worker forwards the IDs down the next outgoing channel. If no outgoing channels exist, or all outgoing channels have returned a negative ID_{result}, a negative ID_{result} will be returned down the output channel to the master, signifying that this branch of the network has no more workers to be identified.

If a valid ID_{result} is returned, a path is expected to follow. The worker then prefixes the path with the channel number which it received ID_{result} from (and sent $ID_{initial}$ down) and forwards ID_{result} with the new path back down the output channel to the master. Once all the workers in this branch have been identified and a negative ID_{result} returned down the channel to the master, the routine initialises the internal communication channels

and initiates the two routing tasks/threads which reside on each worker processor. The routine then exits, returning control to the worker.

As in the initial implementation of the equivalent master routine, the initial implementation of the worker routines included a check to ensure that this worker initialisation procedure was called. This was also excluded in favour of speed of execution, leaving the opening call of this routine to SAPF or the implementer.

2.2 SUBROUTINE F77_NETW_SEND(Length, Buffer)

Description

This subroutine will send *Length* bytes of *Buffer* to the master. *Buffer* may be any variable where *Length* is an integer variable whose value must not exceed F77_NETW_MAX_PACKET_LEN. As previously discussed, if the size of *Buffer* exceeds this value, *Buffer* must be split up into messages whose size does fall within the required range.

Operation

The worker number, followed by the *Length* and *Buffer* are sent along the internal channel to the routing task whose responsibility it is to forward messages to the master (See Section 6.3.4.2).

As previously mentioned, the initial implementation of this routine would split up packets greater than F77_NETW_MAX_PACKET_LEN into more than one smaller sized packets and distribute these, after reserving the network, to the master. Once all the packets constituting *Buffer* were sent, the communication network would be freed. Since it was found that more than 90% of messages sent were less than F77_NETW_MAX_PACKET_LEN in size, the current implementation now requires that the implementer or SAPF split up larger packets (as does the 3L farm library) to ensure message packet sizes fall within the correct range. The exclusion of this option resulted in faster communication due to the removal of over 90% of the redundant

checking previously required to ensure that the size of messages fell within the correct range. Network communication failure will occur should *Length* exceed this maximum.

2.3 SUBROUTINE F77_NETW_RECEIVE(Length, Buffer)

Description

This subroutine receives from the master data sent to the worker calling this routine. The data received is returned in *Buffer* with the length of the data returned in the integer *Length*.

Operation

The data is received along the internal channel from the routing task whose responsibility it is to distribute the work or message packets to the workers along this branch of the network (See Section 6.3.4.1). *Length* is received first, followed by *Length* bytes of *Buffer*.

2.4 SUBROUTINE F77_NETW_USE()

Description

This subroutine reserves the communications path to the master, ensuring that the master only receives future messages sent by this worker. It must be used in conjunction with F77_NETW_FREE which is used to free the network hence allowing the remaining workers to continue data transmission to the master.

Operation

The subroutine first checks to ensure this worker has not already reserved the network, in which case it merely exits without performing any operation. If the network is not reserved, the negative value of the workers ID is transmitted along the internal channel to the router which forwards messages to the master. The transmission of such data is taken to be a request by the worker to reserve the communication path to the master for its exclusive use.

2.5 SUBROUTINE F77_NETW_FREE()

Description

This subroutine is used to free the communication path to the master after it has been reserved by F77_NETW_USE.

Operation

The subroutine initially checks whether this worker currently is reserving the network. If not, it returns without performing any operation. If so, the negative value of the worker's ID is transmitted along the internal channel to the router which forwards messages to the master. The transmission of such data is taken to be a request by the worker to release the communication path to the master, permitting other workers along this branch to communicate to the master once again.

Appendix B - OCCAM/Turbo Pascal Server User Manual

This appendix consists of the User Manual for the server. The Turbo Pascal Source of the *Host Server* as well as the OCCAM 2 source of the *Transputer Server* and transputer libraries appear in the thesis supplement.

1 INTRODUCTION

This document covers the OCCAM procedures available to the programmer using the TURBO PASCAL TRANSPUTER SERVER. This transputer server, called **ALSERVER**, provides OCCAM programmers with most of the features supported by TURBO PASCAL V5.5[4]. The OCCAM procedures are called in a similar manner to the TURBO PASCAL routines, with the exception that the OCCAM procedures all include two extra parameters in order to allow the programmer to specify the channels where the link to the PC exists. The first channel is that along which the OCCAM procedure requests are sent to the PC, while the second is the channel along which the results are returned from the PC to the procedure. Each OCCAM library routine therefore must be passed the channel where data will come from the PC (1st parameter), and the channel along which to send data to the PC (2nd parameter). The remaining parameters (if any) are those required by the Turbo Pascal Library routines themselves.

An example is given below of the two channels required, and their placement for the INMOS B004 board.

```
CHAN OF ANY  From.PC, To.PC:
PLACE From.PC AT 0:
PLACE To.PC   AT 4:
```

The OCCAM procedures are called passing From.PC first, then To.PC, followed by whatever parameters are required by the procedure itself. For example, the procedure that can be called in order to clear the text screen ClrScr is called in the following manner.

ClrScr(From.PC, To.PC)

In the following examples, the channel carrying data in from the PC to the transputer will be referred to by **in**, while the channel leading out to the PC from the transputer will be referred to as **out**.

---

[4] (C) BOLAND SOFTWARE 1989

2 THE SERVER

The Turbo Pascal Server is called **ALSERVER** and is called as follows:

ALSERVER [boot1] [boot2] [/Llink.address] [/Kkeyboard] [/B±] [/Ppath]

The server boots programs compiled for the host transputer network, similar to the method of the alien file server AFSERVER supplied by INMOS. Only two file name parameters may be passed to ALSERVER, the first **boot1** is the file name of the boot file with which the host is to be booted, and the second **boot2** is the boot file for the first sub-system connected to the host system which is to be subsequently booted by the host, using the quadputer library. Either of these file names may be replaced by **/*** which indicates to ALSERVER that the host/sub-system is already running.

The server has an additional feature which allows the user to terminate or exit to a dos shell while the host is running. When **Control-C** or **Break** is pressed the following bottom line will appear:

(T)erminate (E)xit to Dos (C)ontinue (I)gnore (F)lush Keyboard

The options are as follows:

(T) Terminate Server, aborting communications to the host. This option should be selected when the host "hangs" and the user wishes to exit to dos.

(E) Exit to Dos, saving the current screen and communication status between the host and the PC. The screen is preserved in the file "ALSERVER.DAT" and the communication status in "ALSERVER.DAT". This command results in an ordered shut-down of the PC server, cutting communications to the host. Communications may be resumed by passing "/*" as the first and/or second parameter to the server instead of the file name which originally was used to boot the transputer, indicating that either the host or sub-system already is running. The user should ensure that these two files exist in the current directory where the server was started and exited to the Dos shell. If these files are not in the current directory (the user has changed directories), the **/Ppath** option should be included where **path** specifies the path where these two files may be found.

(C) Continue allows the user to enter the **Control-C** or **Break** keypresses which are to be interpreted by the host program.

(I) Ignore the **Control-C** or **Break** key and continue.

(F) Flush the keyboard buffer, removing any keys that have been pressed and placed in the buffer. The buffer may contain up to 255 characters.

The /L option specifies the base port address for link 0 of the host transputer from the PC. **link.address** may be in decimal, or in hexadecimal, in which case the address must be preceded by a #. The default for this address is #150.

The /K option specifies a list of characters which are to be placed in the keyboard buffer. This implies that immediately after booting, the keys represented by **keyboard** will be simulated as having been pressed. Control characters are preceded by a ^ and

the decimal (3 digits) or hexadecimal (2 digits) number for that character. The actual character "^" is inserted by ^^. Thus, for example, <RETURN> will be ^013 or ^#0D, <ESC> will be ^027 or ^#1B .

For example: /KYNFAlex^013Q
will place the characters "Y", "N", "F", "A", "l", "e", "x", <RETURN> and "Q" in the keyboard buffer, as if those keys had actually been pressed.

The /B followed by a + or - specifies whether the break mode is to be turned on or off. Break mode allows the server to react to a Control-C or <BREAK> keypress. When on, the server responds to <BREAK> and when it is off it simply places a Control-C in the keyboard buffer. Hence /B+ turns break checking on and /B- turns it off. The default is ON.

The /P followed by a path is used only when re-starting ALSERVER when the Exit to Dos option was selected. **path** specifies the path to the directory where the files ALSERVER.DAT and ALSERVER.SCR may be found.

NOTE: The routine TerminateServer must be called at the end of every OCCAM program which uses ALSERVER in order to terminate the server running on the PC and to return to dos. If this is not called, the PC server will "hang", waiting for further messages which will never come from the host transputer since that has terminated.

3 PROTOCOL LIBRARIES

Listed below are the OCCAM protocol libraries used by the OCCAM and Turbo Pascal server routines, headed by the file name under which the they are stored, and their respective file contents. The Turbo Pascal server, running on the PC, simply waits for a command to be sent to it from the program (normally written in OCCAM) running on the host system which the Turbo Pascal server is monitoring. Commands are sent to the server in the byte format listed below, followed by the communication necessary to convey information to and from the PC to the program running on the host transputer system.

"TPKEYPRO.TSR"

```
-- KEYBOARD PROTOCOL    1 -  50
VAL TPKeypressed            IS BYTE   1:
VAL TPReadKey               IS BYTE   2:
VAL TPReadEchoKey           IS BYTE   3:
VAL TPReadString            IS BYTE   4:
VAL TPReadEchoString        IS BYTE   5:
VAL TPReadNumber            IS BYTE   6:
VAL TPReadEchoNumber        IS BYTE   7:
VAL TPBreakOn               IS BYTE   8:
```

"TPSCRPRO.TSR"

```
-- SCREEN PROTOCOL     51 - 100
VAL TPClrScr                IS BYTE  51:
VAL TPClrEOS                IS BYTE  52:
VAL TPClrEOL                IS BYTE  53:
VAL TPGotoXY                IS BYTE  54:
VAL TPNormVideo             IS BYTE  55:
VAL TPHighVideo             IS BYTE  56:
VAL TPLowVideo              IS BYTE  57:
VAL TPSound                 IS BYTE  58:
VAL TPNoSound               IS BYTE  59:
VAL TPWhereX                IS BYTE  60:
VAL TPWhereY                IS BYTE  61:
VAL TPTextBackground        IS BYTE  62:
VAL TPTextColor             IS BYTE  63:
VAL TPTextMode              IS BYTE  64:
VAL TPWriteString           IS BYTE  65:
VAL TPWriteLnString         IS BYTE  66:
VAL TPPrintScreen           IS BYTE  80:
```

"TPFILPRO.TSR"

```
-- FILE PROTOCOL      101 - 150
VAL TPAssignFile            IS BYTE 102:
VAL TPResetFile             IS BYTE 103:
VAL TPRewriteFile           IS BYTE 104:
VAL TPAppendFile            IS BYTE 105:
VAL TPCloseFile             IS BYTE 106:
VAL TPRenameFile            IS BYTE 107:
VAL TPDeleteFile            IS BYTE 108:
VAL TPEndOfFile             IS BYTE 109:
VAL TPFileError             IS BYTE 110:
```

```
VAL TPReadTextLine          IS BYTE 111:
VAL TPReadTextNum           IS BYTE 112:
VAL TPReadBlock             IS BYTE 113:
VAL TPReadBool              IS BYTE 114:
VAL TPReadByte              IS BYTE 115:
VAL TPReadInt               IS BYTE 116:
VAL TPReadInt16             IS BYTE 117:
VAL TPReadInt32             IS BYTE 118:
VAL TPReadInt64             IS BYTE 119:
VAL TPReadReal32            IS BYTE 120:
VAL TPReadReal64            IS BYTE 121:

VAL TPWriteTextLine         IS BYTE 131:
VAL TPWriteText             IS BYTE 132:
VAL TPWriteBlock            IS BYTE 133:
VAL TPWriteBool             IS BYTE 134:
VAL TPWriteByte             IS BYTE 135:
VAL TPWriteInt              IS BYTE 136:
VAL TPWriteInt16            IS BYTE 137:
VAL TPWriteInt32            IS BYTE 138:
VAL TPWriteInt64            IS BYTE 139:
VAL TPWriteReal32           IS BYTE 140:
VAL TPWriteReal64           IS BYTE 141:

"TPDOSPRO.TSR"
-- DOS PROTOCOL       151 - 200
VAL TPExec                  IS BYTE 151:
VAL TPGetDate               IS BYTE 152:
VAL TPSetDate               IS BYTE 153:
VAL TPGetTime               IS BYTE 154:
VAL TPSetTime               IS BYTE 155:
VAL TPDiskFree              IS BYTE 156:
VAL TPDiskSize              IS BYTE 157:
VAL TPGetFAttr              IS BYTE 158:
VAL TPSetFAttr              IS BYTE 159:
VAL TPFileExists            IS BYTE 160:
VAL TPInterrupt             IS BYTE 161:
VAL TPReadPort              IS BYTE 162:
VAL TPWritePort             IS BYTE 163:

"TPGRFPRO.TSR"
-- GRAPH PROTOCOL     201 - 240
VAL TPInitGraph             IS BYTE 201:
VAL TPSetGraphMode          IS BYTE 202:
VAL TPRestoreCrtMode        IS BYTE 203:
VAL TPCloseGraph            IS BYTE 204:
VAL TPSetViewPort           IS BYTE 205:
VAL TPClearViewPort         IS BYTE 206:
VAL TPClearDevice           IS BYTE 207:
VAL TPDetectGraph           IS BYTE 208:
VAL TPSetLineStyle          IS BYTE 209:
VAL TPSetFillPattern        IS BYTE 210:
VAL TPSetFillStyle          IS BYTE 211:
```

```
VAL TPArc                IS BYTE 212:
VAL TPBar                IS BYTE 213:
VAL TPBar3D              IS BYTE 214:
VAL TPCircle             IS BYTE 215:
VAL TPDrawPoly           IS BYTE 216:
VAL TPLine               IS BYTE 217:
VAL TPLineTo             IS BYTE 218:
VAL TPMoveRel            IS BYTE 219:
VAL TPMoveTo             IS BYTE 220:
VAL TPPutPixel           IS BYTE 221:
VAL TPOutText            IS BYTE 222:
VAL TPOutTextXY          IS BYTE 223:
VAL TPRectangle          IS BYTE 224:
VAL TPSetTextJustify     IS BYTE 225:
VAL TPSetTextStyle       IS BYTE 226:
VAL TPGetAspectRatio     IS BYTE 227:
VAL TPGetMaxX            IS BYTE 228:
VAL TPGetMaxY            IS BYTE 229:
VAL TPGraphResult        IS BYTE 230:
VAL TPSetColor           IS BYTE 231:
VAL TPSetBkColor         IS BYTE 232:
VAL TPGetColor           IS BYTE 233:
VAL TPGetBkColor         IS BYTE 234:
VAL TPEllipse            IS BYTE 240:
VAL TPFillEllipse        IS BYTE 241:
VAL TPFillPoly           IS BYTE 242:
VAL TPFloodFill          IS BYTE 243:
VAL TPGetArcCoords       IS BYTE 244:
VAL TPLineRel            IS BYTE 245:
VAL TPPieSlice           IS BYTE 246:
VAL TPSector             IS BYTE 247:

"TPSERPRO.TSR"
-- SERVER PROTOCOL
VAL TPSystemInfo         IS BYTE 250:
VAL TPBootSubsystem      IS BYTE 251:
VAL TPBootFileSubSys     IS BYTE 252:
VAL TPEndServer          IS BYTE 255:
```

4 OCCAM LIBRARIES

The libraries available are listed below, with the procedure declarations in bold print, and a brief explanation following.

4.1 Keyboard Library (TPKeyboard)

```
PROC KeyPressed (CHAN OF ANY in,out, BOOL Pressed)
```

The boolean **Pressed** will return TRUE if one or more keys have been pressed on the keyboard and FALSE otherwise.

```
PROC ReadKey (CHAN OF ANY in,out, VAL BOOL echo, BYTE ch)
```

This procedure will return the key which has been pressed in **ch** if `KeyPressed` returns a TRUE. If `KeyPressed` returns FALSE then the procedure will wait until a key is pressed. If the boolean **echo** is set, the character will be echoed onto the screen.

```
PROC ReadString(CHAN OF ANY in,out, VAL BOOL echo, INT len,
                                    []BYTE string)
```

A string of ASCII characters will be read in from the keyboard. The string is defined in the OCCAM format (ASCII codes representing the string in consecutive locations of the array), but for the purposes of interfacing to Turbo Pascal, they are also treated as ASCIZ. That is, a string will additionally have a character 0 at the end if it does not fill the array in which it is stored entirely. **string** is the array where the ASCIZ string is to be read into, **len** will contain the number of characters in the string (excluding the ending 0) and finally **echo** will determine whether the string to be read is to be echoed on the text screen or not. The server will prevent the user from entering a string which is too long to be stored in **string**.

```
PROC ReadNumber(CHAN OF ANY in,out, VAL BOOL echo, INT len,
                                    []BYTE string)
```

This procedure is similar to that of above, except that it will return only a valid numeric (real or integer) value in the string. The string will have to be converted into an actual integer/real value using the standard OCCAM libraries.

```
PROC BreakOn(CHAN OF ANY in,out, VAL BOOL on)
```

This procedure allows the user to turn the break checking on and off through software. Typically it would be turned off when entering a critical section and then turned back on. Default is ON.

4.2 Screen Library (TPScreen)

```
PROC ClrScr (CHAN OF ANY in,out)
```

This will clear the text screen, if the PC host is in text mode, and return the cursor to the top left hand corner of the screen (position 1,1).

```
PROC ClrEOS (CHAN OF ANY in,out)
```

This procedure will clear the screen from the current cursor position, to the end of the screen.

```
PROC ClrEOL (CHAN OF ANY in,out)
```
This will clear the line on which the cursor currently resides, from the current cursor position to the extreme right hand side of the screen.

```
PROC GotoXY(CHAN OF ANY in,out, VAL INT x,y)
```
The cursor is moved to position (**x**,**y**) on the screen where **x** is the horizontal position and **y** the vertical, with the top left hand corner of the screen being position (1,1) and the bottom right being position (80,25) on a normal PC screen.

PROC HighVideo (CHAN OF ANY in,out)
This is similar to `NormVideo`, except that text output following this command will appear bright in intensity.

```
PROC LowVideo (CHAN OF ANY in,out)
```
This is similar to NormVideo, except that text output following this command will appear low in intensity.

```
PROC NormVideo (CHAN OF ANY in,out)
```
Once called, all text output to the screen following this command will appear in normal intensity.

```
PROC NoSound (CHAN OF ANY in,out)
```
If the PC Host's speaker is busy sounding, then a call to this procedure will terminate the speaker, otherwise it has no effect.

```
PROC PrintScreen(CHAN OF ANY in,out)
```
A call to this procedure will result in the dos **PrtSc** function being called (as if **PrtSc** had been pressed). However, if the screen is in graphics mode, a separate print screen routine will be called which will print the graphics screen to the printer in Epson compatible format.

```
PROC Sound(CHAN OF ANY in,out, VAL INT hertz)
```
A call to this procedure will start the PC Host's speaker sounding at the frequency specified by hertz. The speaker will continue producing the sound until either the frequency is changed, or `NoSound` is called.

```
PROC TextBackground(CHAN OF ANY in,out, VAL INT color)
```
In the case where the PC Host has a color screen with a color card, the background appearance of the text output to the screen following this command will appear in the color passed. The following color constants have been defined and are available to the programmer in both **TPScreen** and **TPGraphics** :

```
VAL Black          IS INT 0:
VAL Blue           IS INT 1:
VAL Green          IS INT 2:
VAL Cyan           IS INT 3:
VAL Red            IS INT 4:
VAL Magenta        IS INT 5:
VAL Brown          IS INT 6:
```

```
VAL LightGray     IS INT 7:
VAL DarkGray      IS INT 8:
VAL LightBlue     IS INT 9:
VAL LightGreen    IS INT 10:
VAL LightCyan     IS INT 11:
VAL LightRed      IS INT 12:
VAL LightMagenta  IS INT 13:
VAL Yellow        IS INT 14:
VAL White         IS INT 15:
VAL MaxColor      IS INT 15:
```

`PROC TextColor(CHAN OF ANY in,out, VAL INT color)`
This is similar to that of `TextBackground` except that the foreground color (actual color of the text to be displayed) of the text displayed will appear in the color passed.

`PROC TextMode(CHAN OF ANY in,out, VAL INT mode)`
This selects a specific text mode. The following constants are defined and are available to the programmer:

```
VAL BW40     IS  0:    -- 40x25 B/W on color adaptor
VAL CO40     IS  1:    -- 40x25 color on color adaptor
VAL BW80     IS  2:    -- 80x25 B/W on color adaptor
VAL CO80     IS  3:    -- 80x25 color on color adaptor
VAL Mono     IS  7:    -- 80x25 on monochrome adaptor
```

`PROC WhereX(CHAN OF ANY in,out, INT x)`
`x` will return the current x location on the screen of the cursor.

`PROC WhereY(CHAN OF ANY in,out, INT y)`
`y` will return the current y location on the screen of the cursor.

`PROC WriteString(CHAN OF ANY in,out, VAL []BYTE str)`
This procedure will display the string passed at the current cursor location on the screen, in the current foreground color with the appropriate background color. The cursor position is also advanced respectively. The string may be complete (an entire array), or it may be in ASCIZ format. In any case, a 0 character in the byte array **str** signifies the end of the string.

`PROC WriteLnString(CHAN OF ANY in,out, VAL []BYTE str)`
The string passed is displayed on the screen at the current cursor position, similar to that of `WriteString`, with a carriage return and line feed also being issued at the end of the string.

4.3 File Library (TPFile)

The filing utilities provided here are listed below. A maximum of 20 files (text and other) may be assigned at any moment in time. Each file is to be identified by a unique file handle which is simply a byte, identifying it to a corresponding file on the server. The value of the byte is determined by the server when the file is assigned, hence at no stage should this handle be altered by the programmer unless they are fully aware of the consequences of their actions.

All files are simply seen as files of bytes. They may be read as text files, or files consisting of a mixture of bytes, words (16/32/64 bit), and reals (both 32 and 64 bit) and blocks of bytes. These bytes/integers/reals/blocks may be written to the file in any order, making it necessary to retrieve the data from the file in the same order. Furthermore, in order to allow flexibility, text may be included in the latter file, and bytes/integers/reals/blocks in text files. The text will simply be written in the file as a series of ASCII characters, depending on the length of the string. If the `WriteTextLine` option is used, the text will be written followed by the carriage return (#13) and line feed characters (#10) respectively. When reading a text string from a file, the text is read up to the carriage return and/or line feed, with the latter characters being excluded from the string. The file pointer therefore lies at the position past the line feed. For further details refer to the respective output procedures.

The standard file handle **PRINTER** is also available for the programmer to send output directly to the printer using `WriteString` and `WriteLnString`.

4.3.1 Standard Routines

```
PROC AssignFile(CHAN OF ANY in,out, VAL []BYTE str, BYTE handle)
```

This procedure assigns to **handle** the unique byte which is to identify the file whose name is in **str**. **handle** should therefore be treated as the file variable once the file has been assigned and should *never* be altered by the user. It may however be copied to other BYTE variables, and passed as a value parameter to procedures since it is the value of the handle which determines the file being referenced.

```
PROC ResetFile(CHAN OF ANY in,out, VAL BYTE handle)
```

The file referred to by **handle** is opened for reading.

```
PROC RewriteFile(CHAN OF ANY in,out, VAL BYTE handle)
```

The file referred to by **handle** is opened for writing.

```
PROC AppendFile(CHAN OF ANY in,out, VAL BYTE handle)
```

The file referred to by **handle** is opened for writing with the file pointer being positioned at the end of the file. This allows users to add to the end of an existing file.

```
PROC CloseFile(CHAN OF ANY in,out, VAL BYTE handle)
```
A file is closed and the file buffer flushed. **handle** then becomes invalid and must be re-assigned to the file of the same name if the file that was closed is to be re-opened for reading.

```
PROC RenameFile(CHAN OF ANY in,out, VAL BYTE handle,
                                    VAL []BYTE str)
```
This will rename the file which has been assigned using `AssignFile` to the name passed in **str**. The file must not be open for reading or writing.

```
PROC DeleteFile(CHAN OF ANY in,out, VAL BYTE handle)
```
This will delete the file which has been assigned using `AssignFile`. The file must not currently be open for reading or writing.

```
PROC EndOfFile(CHAN OF ANY in,out, VAL BYTE handle, BOOL eof)
```
This will return TRUE in **eof** if the file is open for writing, or if the file is open for reading and the file pointer is past the end of the file. Otherwise **eof** will return FALSE.

```
PROC FileError(CHAN OF ANY in,out, INT error.code)
```
This procedure will return the IORESULT (See Turbo Pascal Manual) of the last IO operation performed. The byte **error.code** may contain the following values: (These constants are available when using **TPFile**)

```
VAL NoError         IS    0:
VAL FileNotFound    IS    2:
VAL PathNotFound    IS    3:
VAL TooManyFiles    IS    4:
VAL AccessDenied    IS    5:
VAL InvalidHandle   IS    6:
VAL InvalidAccess   IS   12:
VAL InvalidDrive    IS   15:
VAL CantRemoveDir   IS   16:
VAL CantRename      IS   17:
VAL DiskReadError   IS  100:
VAL DiskWriteError  IS  101:
VAL FileNotAssign   IS  102:
VAL FileNotOpen     IS  103:
VAL FileNotInput    IS  104:
VAL FileNotOutput   IS  105:
VAL InvalidNumeric  IS  106:
```

4.3.2 Read Routines

In all the routines that follow, the file handle passed must be a valid handle of a file which has been assigned and opened for reading. In all cases of reads or writes, the error code is set signifying whether the operation was completed successfully, and if not then why. The error code may be returned by a call to `FileError`.

```
PROC ReadTextLine(CHAN OF ANY in,out, VAL BYTE handle, INT len,
                                      []BYTE str)
```

Reads in a string from a file (not necessarily text) into **str**. **handle** must be the handle of a valid file which has been opened for reading. The ASCII characters are read from the file until a carriage return (#13) and/or a line feed (#10) (or both, in that order) are encountered. If, however, the array **str** contains less positions than are available to facilitate the number of characters in the line of text, **str** will be filled entirely by the characters and the file pointer will reside at the next available character to be read in. The string will be stored in **str** in ASCIZ format and the length of the string read returned in **len.**

For Example: Consider the line of text

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz",#13,#10
```

A call to this procedure where **str** is [10]BYTE will return to **str** in successive calls:- `"ABCDEFGHIJ"` , `"KLMNOPQRST"` , `"UVWXYZabcd"`, `"efghijklmn"` , `"opqrstuvwx"` and `"yz"`. The latter call will have a #0 in position 3 of the array and **len** = 2, while the former will have **len** = 10.

Note: Any byte may be read into the string, except for #13 and #10 which are used to signify the end of a line of text in a file.

```
PROC ReadTextNum(CHAN OF ANY in,out, VAL BYTE handle, INT len,
                                     []BYTE str)
```

Reads in a string representing a numeric from the open file passed in **handle**. Leading spaces are omitted and the first non-printable ASCII character encountered signifies the end of the number being read in string format. The string is returned in ASCIZ format, with the length of the string being returned in **len.** The file pointer will be at the first non-printable ASCII byte after the numeric string. Any #10's or #13's and spaces will be skipped over and ignored when searching for a text number.

```
PROC ReadBlock(CHAN OF ANY in,out, VAL BYTE handle,
                                   VAL INT blocksize,
                                   []BYTE block,
                                   INT numread)
```

Reads in a block of bytes from the file **handle. handle** must be opened for reading. The number of bytes to be read in must be passed in **blocksize**, the array into which the bytes are to be read is **block**, and the actual number of bytes that were read from the file are returned in **numread.** Normally **blocksize** = **numread,** except in the case where the end of the file has been reached, in which case only the remaining unread bytes are returned, or in the case of a read error. It must be noted that the largest block size which may be read is 64K (65535 bytes).

```
PROC ReadBool(CHAN OF ANY in,out, VAL BYTE handle, BOOL val)
```
Read in a boolean value val from the file **handle**. **val** is stored as a single byte in the file. If the byte is 0 then **val** will be set to FALSE, otherwise it will be TRUE.

```
PROC ReadByte(CHAN OF ANY in,out, VAL BYTE handle, BYTE val)
```
Read a single byte from the file **handle** and return its value in **val**.

```
PROC ReadInt(CHAN OF ANY in,out, VAL BYTE handle, INT val)
```
Read a single 32 bit integer from the file **handle** and return its value in **val**.

```
PROC ReadInt16(CHAN OF ANY in,out, VAL BYTE handle, INT16 val)
```
Read a single 16 bit integer from the file **handle** and return its value in **val**.

```
PROC ReadInt32(CHAN OF ANY in,out, VAL BYTE handle, INT32 val)
```
Read a single 32 bit integer from the file **handle** and return its value in **val**.

```
PROC ReadInt64(CHAN OF ANY in,out, VAL BYTE handle, INT64 val)
```
Read a single 64 bit integer from the file **handle** and return its value in **val**.

```
PROC ReadReal32(CHAN OF ANY in,out, VAL BYTE handle, REAL32 val)
```
Read a single 32 bit real from the file **handle** and return its value in **val**.

```
PROC ReadReal64(CHAN OF ANY in,out, VAL BYTE handle, REAL64 val)
```
Read a single 64 bit real from the file **handle** and return its value in **val**.

4.3.3 Write Routines

In all the routines that follow, the file handle passed must be a valid handle of a file which has been assigned and opened for writing. In all cases the error code is set signifying whether the operation was completed successfully, and if not then why.

```
PROC WriteTextLine(CHAN OF ANY in,out, VAL BYTE handle,
                                       VAL []BYTE str)
```
The ASCIZ text string passed in **str** is written to the file **handle** which must be opened for writing. **handle** may be **PRINTER** in order to write to the printer instead of a file. A carriage return and line feed is also written at the end of writing the string to the file.

```
PROC WriteText(CHAN OF ANY in,out, VAL BYTE handle,
                                   VAL []BYTE str)
```
The text string passed in **str** is written to the file **handle** which must be opened for writing. The difference between this routine and `WriteTextLine` is that a carriage return and line feed is not issued at the end of writing the string to the file, allowing the programmer to make several calls to `WriteText` with each string being kept on the same line until a `WriteTextLine` is issued or the file closed. As before **handle** may be **PRINTER** in order to print text on the printer.

```
PROC WriteBlock(CHAN OF ANY in,out, VAL BYTE handle,
                                    VAL INT blocksize,
                                    VAL []BYTE block,
                                    INT numwritten)
```

Write out a block of bytes to the file **handle**. **handle** must be opened for writing. The number of bytes to be written in must be passed in **blocksize**, the array in which the bytes are to be written is **block**, and the actual number of bytes that are written to the file are returned in **numwritten**. Normally **blocksize** = **numwritten**, except in the case where the disk becomes full. It must be noted that the largest block size which may be written is 64K (65535 bytes).

```
PROC WriteBool(CHAN OF ANY in,out, VAL BYTE handle, VAL BOOL val)
```

Write a boolean value **val** to the file **handle**. **val** is stored as a single byte in the file. If **val** is TRUE the byte will be 1, otherwise it will be 0.

```
PROC WriteByte(CHAN OF ANY in,out, VAL BYTE handle,val)
```

Write a single byte **val** to the file **handle**.

```
PROC WriteInt(CHAN OF ANY in,out, VAL BYTE handle, VAL INT val)
```

Write a 32 bit integer **val** to the file **handle**.

```
PROC WriteInt16(CHAN OF ANY in,out, VAL BYTE handle,
                                    VAL INT16 val)
```

Write a 16 bit integer **val** to the file **handle**.

```
PROC WriteInt32(CHAN OF ANY in,out, VAL BYTE handle,
                                    VAL INT32 val)
```

Write a 32 bit integer **val** to the file **handle**.

```
PROC WriteInt64(CHAN OF ANY in,out, VAL BYTE handle,
                                    VAL INT64 val)
```

Write a 64 bit integer **val** to the file **handle**.

```
PROC WriteReal32(CHAN OF ANY in,out, VAL BYTE handle,
                                     VAL REAL32 val)
```

Write a 32 bit real **val** to the file **handle**.

```
PROC WriteReal64(CHAN OF ANY in,out, VAL BYTE handle,
                                     VAL REAL64 val)
```

Write a 64 bit real **val** to the file **handle**.

4.4 Dos Libraries (TPDos)

```
PROC Exec(CHAN OF ANY in,out, VAL []BYTE path, cmd)
```

Executes a specified program on the host PC with a specified command line. The program name is given by the **path** parameter and the command line is given by **cmd.** The command will not execute if the PC does not have enough memory in which to execute it.

```
PROC GetDate(CHAN OF ANY in,out, INT year, month, day, dayofweek)
```

This will return the current date of the host PC. **year** may be from 1980..2099, month 1..12, **day** 1..31 and **dayofweek** 0..6 where 0 corresponds to Sunday.

```
PROC SetDate(CHAN OF ANY in,out, VAL INT year, month, day)
```

This will set the current date of the host PC to the date specified. **year** may be from 1980..2099, **month** 1..12 and **day** 1..31.

```
PROC GetTime(CHAN OF ANY in,out, INT hour, minute,
                                 second, sec100)
```

This will return the current time of the host PC in a 24 hour clock format. **sec100** is the number of 100ths of a second elapsed.

```
PROC SetTime(CHAN OF ANY in,out, VAL INT hour, minute, second,
                                     sec100)
```

This set the current time of the host PC to the time specified in a 24 hour clock format. **sec100** is the number of 100ths of a second elapsed.

```
PROC DiskFree(CHAN OF ANY in,out, VAL INT disk,  INT free)
```

This will return the size in bytes of the amount of free space available on the host PC's drive devices. **disk** is a byte representing the current disk drive where 0 implies the current logged drive of the PC, 1 specifies drive A, 2 drive B, 3 drive C etc.

```
PROC DiskSize(CHAN OF ANY in,out, VAL INT disk, INT size)
```

This will return the size in bytes of the amount of entire disk space on the host PC. **disk** is a byte representing the current disk drive where 0 implies the current logged drive of the PC, 1 specifies drive A, 2 drive B, 3 drive C etc.

```
PROC GetFAttr(CHAN OF ANY in,out, VAL BYTE handle,  INT attr)
```

The current file attribute for the file **handle** is returned in **attr.** The file must be assigned but not open for reading or writing. The following are defined as constants.

```
VAL ReadOnly  IS  1:
VAL Hidden    IS  2:
VAL SysFile   IS  4:
VAL VolumeID  IS  8:
VAL Directory IS  16:
VAL Archive   IS  32:
VAL AnyFile   IS  63:
```

`PROC SetFAttr(CHAN OF ANY in,out, VAL BYTE handle, VAL INT attr)`
The file attribute passed in **attr** is assigned to the file **handle**. The file must be assigned but not open for reading or writing. For example, in order to make the file Read Only and Hidden, the procedure is called as follows:

```
SetFAttr(In,Out,File,ReadOnly + Hidden,error.code)
```

`PROC FileExists(CHAN OF ANY in,out, VAL []BYTE str, BOOL does)`
A file or directory name (including wild characters and path specifications) is passed in **str** and this procedure will return TRUE in **does** if the file or directory is found, and FALSE otherwise.

4.5 Graphics Library (TPGraphics)

A subset of the Turbo Pascal V5.5 graphics libraries have been incorporated. The library is called in exactly the same method as the Turbo Graphics library, with the exception that the first two parameters are as before the input and output channels from and to the PC Host.

4.5.1 Graphics Constants

The following graphics constants have been defined and are available.

```
-- GraphResult error return codes
VAL grOk                IS INT     0:
VAL grNoInitGraph       IS INT  (-1):
VAL grNotDetected       IS INT  (-2):
VAL grFileNotFound      IS INT  (-3):
VAL grInvalidDriver     IS INT  (-4):
VAL grNoLoadMem         IS INT  (-5):
VAL grNoScanMem         IS INT  (-6):
VAL grNoFloodMem        IS INT  (-7):
VAL grFontNotFound      IS INT  (-8):
VAL grNoFontMem         IS INT  (-9):
VAL grInvalidMode       IS INT (-10):
VAL grError             IS INT (-11):   -- generic error
VAL grIOerror           IS INT (-12):
VAL grInvalidFont       IS INT (-13):
VAL grInvalidFontNum    IS INT (-14):
VAL grInvalidDeviceNum IS INT (-15):

-- define graphics drivers
VAL Detect    IS INT 0:    -- requests autodetection
VAL CGA       IS INT 1:
VAL MCGA      IS INT 2:
VAL EGA       IS INT 3:
VAL EGA64     IS INT 4:
VAL EGAMono   IS INT 5:
VAL RESERVED  IS INT 6:
VAL HercMono  IS INT 7:
VAL ATT400    IS INT 8:
VAL VGA       IS INT 9:
VAL PC3270    IS INT 10:
```

```
-- graphics modes for each driver
VAL CGAC0       IS INT 0:   -- 320x200 pal 0 L/Green,
                               L/Red, Yellow:    1 pg
VAL CGAC1       IS INT 1:   -- 320x200 pal 1 L/Cyan, L/Magenta,
                               White:    1 pg
VAL CGAC2       IS INT 2:   -- 320x200 pal 2 Green, Red, Brown:
                               1 pg
VAL CGAC3      IS INT 3:  -- 320x200 pal 3 Cyan, Magenta, L/Gray:
                               1 pg
VAL CGAHi       IS INT 4:   -- 640x200 1 pg
VAL MCGAC0      IS INT 0:   -- 320x200 pal 0 L/Green, L/Red,
                               Yellow:   1 pg
VAL MCGAC1      IS INT 1:   -- 320x200 pal 1 L/Cyan, L/Magenta,
                               White:  1 pg
VAL MCGAC2      IS INT 2:   -- 320x200 pal 2 Green, Red, Brown:
                               1 pg
VAL MCGAC3     IS INT 3:  -- 320x200 pal 3 Cyan, Magenta, L/Gray:
                               1 pg
VAL MCGAMed     IS INT 4:   -- 640x200 1 pg
VAL MCGAHi      IS INT 5:   -- 640x480 1 pg
VAL EGALo       IS INT 0:   -- 640x200 16 color 4 pg
VAL EGAHi       IS INT 1:   -- 640x350 16 color 2 pg
VAL EGA64Lo     IS INT 0:   -- 640x200 16 color 1 pg
VAL EGA64Hi     IS INT 1:   -- 640x350 4 color  1 pg
VAL EGAMonoHi   IS INT 3:   -- 640x350 64K on card, 1 pg: 256K on
                               card,  2 pg
VAL HercMonoHi IS INT 0:   -- 720x348 2 pg
VAL ATT400C0    IS INT 0:   -- 320x200 pal 0 L/Green, L/Red,
                               Yellow: 1 pg
VAL ATT400C1    IS INT 1:   -- 320x200 pal 1 L/Cyan, L/Magenta,
                               White:  1 pg
VAL ATT400C2    IS INT 2:   -- 320x200 pal 2 Green, Red, Brown:
                               1 pg
VAL ATT400C3   IS INT 3:  -- 320x200 pal 3 Cyan, Magenta, L/Gray:
                               1 pg
VAL ATT400Med   IS INT 4:   -- 640x200 1 pg
VAL ATT400Hi    IS INT 5:   -- 640x400 1 pg
VAL VGALo       IS INT 0:   -- 640x200 16 color 4 pg
VAL VGAMed      IS INT 1:   -- 640x350 16 color 2 pg
VAL VGAHi       IS INT 2:   -- 640x480 16 color 1 pg
VAL PC3270Hi    IS INT 0:   -- 720x350 1 pg

-- Colors for SetPalette and SetAllPalette
VAL Black           IS INT 0:
VAL Blue            IS INT 1:
VAL Green           IS INT 2:
VAL Cyan            IS INT 3:
VAL Red             IS INT 4:
VAL Magenta         IS INT 5:
VAL Brown           IS INT 6:
VAL LightGray       IS INT 7:
VAL DarkGray        IS INT 8:
VAL LightBlue       IS INT 9:
```

```
VAL LightGreen      IS INT 10:
VAL LightCyan       IS INT 11:
VAL LightRed        IS INT 12:
VAL LightMagenta    IS INT 13:
VAL Yellow          IS INT 14:
VAL White           IS INT 15:

-- Line styles and widths for Get/SetLineStyle
VAL SolidLn         IS INT 0:
VAL DottedLn        IS INT 1:
VAL CenterLn        IS INT 2:
VAL DashedLn        IS INT 3:
VAL UserBitLn       IS INT 4:       -- User-defined line style


VAL NormWidth       IS INT 1:
VAL ThickWidth      IS INT 3:

-- Set/GetTextStyle constants
VAL DefaultFont     IS INT 0:       -- 8x8 bit mapped font

VAL TriplexFont     IS INT 1:       -- "Stroked" fonts
VAL SmallFont       IS INT 2:
VAL SansSerifFont   IS INT 3:
VAL GothicFont      IS INT 4:

VAL HorizDir        IS INT 0:       -- left to right
VAL VertDir         IS INT 1:       -- bottom to top

VAL UserCharSize    IS BYTE 0:      -- user-defined char size

-- Clipping constants
VAL ClipOn          IS TRUE:
VAL ClipOff         IS FALSE:

-- Bar3D constants
VAL TopOn           IS TRUE:
VAL TopOff          IS FALSE:

-- Fill patterns for Get/SetFillStyle
VAL EmptyFill       IS INT 0:  -- fills area in background color
VAL SolidFill       IS INT 1:  -- fills area in solid fill color
VAL LineFill        IS INT 2:  -- --- fill
VAL LtSlashFill     IS INT 3:  -- /// fill
VAL SlashFill       IS INT 4:  -- /// fill with thick lines
VAL BkSlashFill     IS INT 5:  -- \\\ fill with thick lines
VAL LtBkSlashFill   IS INT 6:  -- \\\ fill
VAL HatchFill       IS INT 7:  -- light hatch fill
VAL XHatchFill      IS INT 8:  -- heavy cross hatch fill
VAL InterleaveFill  IS INT 9:  -- interleaving line fill
VAL WideDotFill     IS INT 10: -- Widely spaced dot fill
VAL CloseDotFill    IS INT 11: -- Closely spaced dot fill
VAL UserFill        IS INT 12: -- user defined fill
```

```
-- BitBlt operators for PutImage
VAL NormalPut       IS INT 0:     -- MOV
VAL XORPut          IS INT 1:     -- XOR
VAL OrPut           IS INT 2:     -- OR
VAL AndPut          IS INT 3:     -- AND
VAL NotPut          IS INT 4:     -- NOT

-- Horizontal and vertical justification for SetTextJustify
VAL LeftText        IS INT 0:
VAL CenterText      IS INT 1:
VAL RightText       IS INT 2:

VAL BottomText      IS INT 0:
VAL TopText         IS INT 2:

VAL Black           IS INT  0:
VAL Blue            IS INT  1:
VAL Green           IS INT  2:
VAL Cyan            IS INT  3:
VAL Red             IS INT  4:
VAL Magenta         IS INT  5:
VAL Brown           IS INT  6:
VAL LightGray       IS INT  7:
VAL DarkGray        IS INT  8:
VAL LightBlue       IS INT  9:
VAL LightGreen      IS INT 10:
VAL LightCyan       IS INT 11:
VAL LightRed        IS INT 12:
VAL LightMagenta    IS INT 13:
VAL Yellow          IS INT 14:
VAL White           IS INT 15:
VAL MaxColors       IS INT 15:
```

4.5.2 Graphics Procedures

```
PROC Arc(CHAN OF ANY in,out, VAL INT X, Y, StAngle,
                                        EndAngle,Radius)
```

Draws a circular arc around **StAngle** to **EndAngle** using **(x,y)** as the center point with a radius of **Radius**. A start angle of 0 and end angle of 360 will draw a complete circle. The angles are counterclockwise with 0 at 3 o'clock, 90 at 12 o'clock etc.

```
PROC Bar(CHAN OF ANY in,out, VAL INT X1, Y1, X2, Y2)
```

Draws a filled in rectangle (used in bar charts for example) using the patterns and color defined by **SetFillStyle** or **SetFillPattern**. **(x1,y1)** is the top left corner of the bar and **(x2,y2)** the bottom right hand corner.

```
PROC Bar3D(CHAN OF ANY in,out, VAL INT X1, Y1, X2, Y2, Depth,
                               VAL BOOL Top)
```
Draws a filled-in three dimensional bar using the pattern and color defined by **SetFillStyle** or **SetFillPattern**. The 3-D outline of the bar is drawn in the current line style and color. **(x1,y1)** is the top left corner of the bar and **(x2,y2)** the bottom right hand corner, while **Depth** is the number of pixels deep of the 3-d outline and **Top** a boolean indicating whether a top is to be put on the bar (TRUE) or not (FALSE).

```
PROC Circle(CHAN OF ANY in,out, VAL INT X, Y, Radius)
```
A circle is drawn in the current color using **(x,y)** as the center point using the aspect ratio in order to make a perfect circle rather than an elliptical one.

```
PROC ClearViewPort (CHAN OF ANY in,out)
```
Clear the current viewport, setting the fill color to the background color and moves the current pointer to (0,0).

```
PROC ClearDevice (CHAN OF ANY in,out)
```
Clears the graphics screen and prepares it for output, moving the current graphics pointer to (0,0).

```
PROC CloseGraph (CHAN OF ANY in,out)
```
Shuts down the graphics system, restoring the screen mode before graphics was initialised and frees the memory allocated on the PC Host's heap for the graphics scan buffer.

```
PROC DetectGraph(CHAN OF ANY in,out, INT GDriver, GMode)
```
Checks the graphics hardware and determines which graphics driver and mode to use. When **GDriver** in `InitGraph` is set to **Detect** then a call to `DetectGraph` is made to determine which driver should be used.

```
PROC DrawPoly(CHAN OF ANY in,out, VAL INT NumPoints,
                                  VAL [][2]INT points)
```
Draws the outline of a polygon using the current line style and color. **NumPoints** specifies the number of coordinates which are passed in **points** with the x location being in the first dimension and the y in the second.

```
PROC Ellipse(CHAN OF ANY in,out, VAL INT X, Y, StAngle, EndAngle,
                                                XRadius, YRadius)
```
An ellipse will be drawn from the starting angle to the ending angle given in an anti-clockwise rotation with 0 being the positive x axis. The center point is **(X,Y)**.

```
PROC FillEllipse(CHAN OF ANY in,out, VAL INT X, Y,
                                             XRadius, YRadius)
```
A complete ellipse will be drawn with center point **(X,Y)** and filled in the current fill style/pattern and color.

```
PROC FillPoly(CHAN OF ANY in,out, VAL INT NumPoints,
                                  VAL [][2]INT points)
```
Draws the outline of a polygon using the current line style and color and fills the area enclosed with the current fill color in the current fill style or pattern. **NumPoints** specifies the number of coordinates which are passed in **points** with the x location being in the first dimension and the y in the second.

```
PROC FloodFill(CHAN OF ANY in,out, VAL INT X, Y, Border)
```
This procedure is called to fill an enclosed area, bounded by the color **Border** on bitmap devices. **(x,y)** is a seed within the enclosed area. The region is filled by in the current fill style/pattern and color.

```
PROC GetArcCoords(CHAN OF ANY in,out, VAL INT X, Y, StX, StY,
                                                EndX, EndY)
```
The coordinates of the last arc or ellipse drawn will be returned. The center points is **(X,Y)** and the start and end coordinates of the arc or ellipse given in **(StX,StY)** and **(EndX,EndY)** respectively. This procedure is useful if lines have to be drawn from the start and/or end points of the last drawn arc/ellipse.

```
PROC GetAspectRatio(CHAN OF ANY in,out, INT AspX, AspY)
```
This returns the effective resolution of the graphics screen from which the aspect ratio **(AspX,AspY)** can be computed. This ratio is used to make circles, arcs etc. round.

```
PROC GetBkColor(CHAN OF ANY in,out, INT BkColor)
```
Get the color of the background of items which are plotted on the screen and return it in **BkColor**.

```
PROC GetColor(CHAN OF ANY in,out, INT Color)
```
Get the color in which lines, polygons, fills etc. are to be performed and return it in **Color**.

```
PROC GetMaxX(CHAN OF ANY in,out, INT MaxX)
```
This will return in **MaxX** the right-most column (x resolution) of the current graphics driver and mode.

```
PROC GetMaxY(CHAN OF ANY in,out, INT MaxY)
```
This will return in **MaxY** the bottom-most row (y resolution) of the current graphics driver and mode.

```
PROC GraphResult(CHAN OF ANY in,out, INT Result)
```
This will return an error code for the last graphics operations.

```
PROC InitGraph(CHAN OF ANY in,out, INT GDriver, GMode,
                                   VAL []BYTE path)
```
This procedure initialises the graphics system and puts the PC Host's hardware into graphics mode. Both **GDriver** and **GMode** are passed to `InitGraph`. If **GDriver** is set to **Detect** (0), then the graphics module will automatically detect which graphics card is present using a call to `DetectGraph` and initialise it appropriately. The graphics mode selected returned in **GMode**. If **GDriver** is not **Detect** then the value of **GDriver** is assumed to be a driver number and that driver is selected and set to the mode in **GMode**. **path** specifies the directory path where the graphics drivers may be found. If **path** is null, the driver files must be found in the current directory.

```
PROC Line(CHAN OF ANY in,out, VAL INT X1, Y1, X2, Y2)
```
Draws a straight line from the point **(X1,Y1)** to **(X2,Y2)** in the current line style and thickness set by `SetLineStyle` and color set by `SetColor`.

```
PROC LineRel(CHAN OF ANY in,out, VAL INT X, Y)
```
Draws a straight line from the current graphics cursor position (CP) to (CP(X)+X,CP(Y)+Y) in the current line style and thickness set by `SetLineStyle` and color set by `SetColor`.

```
PROC LineTo(CHAN OF ANY in,out, VAL INT X, Y)
```
Draws a straight line from the current graphics cursor position (CP) to **(X,Y)** in the current line style and thickness set by `SetLineStyle` and color set by `SetColor`.

```
PROC MoveRel(CHAN OF ANY in,out, VAL INT X, Y)
```
Move the current graphics pointer (CP) to a point that is a relative **(X,Y)** distance from the current pointer position.

```
PROC MoveTo(CHAN OF ANY in,out, VAL INT X, Y)
```
Move the current graphics pointer (CP) to a point **(X,Y)** that is an absolute position in the current viewport.

```
PROC OutText(CHAN OF ANY in,out, VAL []BYTE str)
```
Displays the string str at the current pointer (CP) using the current justification settings as defined by `SetTextJustify` in the current style, size and direction as defined by `SetTextStyle`.

```
PROC OutTextXY(CHAN OF ANY in,out, VAL INT X, Y,  VAL []BYTE str)
```
Displays the string str at the position **(X,Y)** using the current justification settings as defined by `SetTextJustify` in the current style, size and direction as defined by `SetTextStyle`.

```
PROC PieSlice(CHAN OF ANY in,out, VAL INT X, Y, StAngle,
                                  EndAngle,Radius)
```
Draws a pie slice starting at **StAngle** through to **EndAngle** using **(X,Y)** as the center point with a radius of **Radius**. The pie slice is then filled with the current fill style or pattern in the current fill color. A start angle of 0 and end angle of 360 will draw a complete filled circle. The angles are counterclockwise with 0 at 3 o'clock, 90 at 12 o'clock etc.

```
PROC PutPixel(CHAN OF ANY in,out, VAL INT X, Y, Pixel)
```
Plots a pixel in the color defined by **Pixel** at the point **(X,Y)**.

```
PROC Rectangle(CHAN OF ANY in,out, VAL INT X1, Y1, X2, Y2)
```
Draws a rectangle using the current line style and color with **(X1,Y1)** in the upper left corner and **(X2,Y2)** in the lower right.

```
PROC RestoreCrtMode (CHAN OF ANY in,out)
```
Restores the screen mode to its original state before graphics was initialised. `RestoreCrtMode` may often be used in conjunction with `SetGraphMode` in order to switch back and forth between text and graphics modes.

```
PROC Sector(CHAN OF ANY in,out, VAL INT X, Y, StAngle, EndAngle,
                                  XRadius, YRadius)
```
An ellipse will be drawn from the starting angle to the ending angle given in an anti-clockwise rotation with 0 being the positive x axis. The center point is **(X,Y)**, and the ellipse will be filled in the current fill style/pattern and color.

```
PROC SetBkColor(CHAN OF ANY in,out, VAL INT BkColor)
```
Set the color of the background of items which are plotted on the screen to **BkColor**.

```
PROC SetColor(CHAN OF ANY in,out, VAL INT Color)
```
Set the color in which lines, polygons, fills etc. are to be performed to the color **Color**.

```
PROC SetFillPattern(CHAN OF ANY in,out, VAL [8]BYTE Pattern,
                                  VAL INT Color)
```
Selects a user-defined fill pattern for all filling done by `FillPoly`, `FloodFill`, `Bar`, `Bar3D`, and `PieSlice`. The Pattern array is 8 bytes long, each byte corresponding to 8 pixels in (one per bit) in the pattern.

```
PROC SetFillStyle(CHAN OF ANY in,out, VAL INT Pattern, Color)
```
Selects the fill pattern and color for all filling done by `FillPoly`, `FloodFill`, `Bar`, `Bar3D`, and `PieSlice`.

```
PROC SetGraphMode(CHAN OF ANY in,out, VAL INT GMode)
```
Sets the PC Host system to graphics mode and clears the screen. **GMode** must be a valid mode for the current device driver. `SetGraphMode` may often be used in conjunction with `RestoreCrtMode` in order to switch back and forth between text and graphics modes.

```
PROC SetLineStyle(CHAN OF ANY in,out, VAL INT Style, Pattern,
                                                  Thickness)
```
Sets the current line width and style. All lines drawn by `Line`, `LineTo`, `Rectangle`, `DrawPoly`, `Arc`, `Circle`, etc. will be drawn in this line style specified, When **Style** is set to **UserBitLn** then the line is output using the 16 bit pattern defined in **Thickness**.

```
PROC SetTextJustify(CHAN OF ANY in,out, VAL INT Horiz, Vert)
```
This sets the text justification values used by `OutText` and `OutTextXY` so that the text is justified around the current pointer or the point given.

```
PROC SetTextStyle(CHAN OF ANY in,out, VAL INT Font, Direction,
                                      VAL BYTE Size)
```
Sets the current text font, style and character magnification factor.

```
PROC SetViewPort(CHAN OF ANY in,out, VAL INT X1, Y1, X2, Y2,
                                      VAL BOOL Clip)
```
Set the current output viewport or window for graphics output. **(X1,Y1)** define the upper left corner of the viewport and **(X2,Y2)** define the lower right corner (0 <= **X1** < **X2** and 0 <= **Y1** < **Y2**). The upper left corner of a viewport is (0,0). The boolean **Clip** determines whether drawings are clipped at the current viewport boundaries. All graphics commands are viewport relative. The initial graphics window or viewport is set to (0,0,`GetMaxX`,`GetMaxY`) where `GetMaxX` and `GetMaxY` are the values returned by the OCCAM procedures of that name respectively.

4.6 Server Library (TPServer)

The following list of routines are included as various utilities. The most notable is the `TerminateServer` procedure which is used to terminate the server running on the PC, otherwise, if an OCCAM program finished on the host transputer, the server would sit and wait forever for commands from the transputer. This is normally done as the last statement of any OCCAM program which uses the server and tells the server that it must terminate.

```
PROC SystemInfo(CHAN OF ANY in,out, INT memsize)
```
This procedure returns the size of the memory (in bytes) in **memsize** of the host transputer that was booted.

```
PROC TerminateServer (CHAN OF ANY in,out)
```
This signals the server running on the PC to complete. Any files left open by the OCCAM program will be automatically closed, any the PC will return to the DOS level.

APPENDIX C - BSCAT

BScat Appendix (i) - RELATE output from SAPF output of master and worker

BSCAT	transfer method	WORKER
A	=>	A
AS		AS
B	=>	B
BCD	=>	BCD
BD	=>	BD
BLR	=>	BLR
BTDC	=>	BTDC
BTI	=>	BTI
BTS	=>	BTS
B_BNDDCL		B_BNDDCL
B_BNDFCL		B_BNDFCL
B_BNDICL		B_BNDICL
B_BNDRCL		B_BNDRCL
B_BNDSCL		B_BNDSCL
B_DATA		B_DATA
B_DOUBLE		B_DOUBLE
B_ESTOR		B_ESTOR
B_FARP		B_FARP
B_GEOMEL		B_GEOMEL
B_GEOPLA		B_GEOPLA
B_GROUND		B_GROUND
B_IMAINF		B_IMAINF
B_IMCINF		B_IMCINF
B_LOGDIF		B_LOGDIF
B_LPLCY		B_LPLCY
B_LSHDP		B_LSHDP
B_LSHDT		B_LSHDT
B_PATDAT		B_PATDAT
B_SORINF		B_SORINF
B_TEST		B_TEST
CAS		CAS
CJ	<==>	CJ
CNC	=>	CNC
COMPLEX_LENGTH		COMPLEX_LENGTH
CPI4	=>	CPI4
CPS		CPS
CTC	=>	CTC
CTHS		CTHS
D		D
DDC	=>	DDC
DP		DP
DPR	<==>	DPR
DT		DT
DTDC	=>	DTDC
DTI	=>	DTI
DTS	=>	DTS
EDCPH		EDCPH
EDCRPP		EDCRPP
EDCRPT		EDCRPT
EDCTH		EDCTH
EDDPH		EDDPH
EDDTH		EDDTH
EDPCPH		EDPCPH
EDPCTH		EDPCTH
EDPPH		EDPPH
EDPTH		EDPTH
EDRCP		EDRCP
EDRCT		EDRCT
EDRPP		EDRPP
EDRPT		EDRPT
EIPH		EIPH
EITH		EITH
EPH		EPH
EPHT	<==>	EPHT
ERCAP		ERCAP
ERCAT		ERCAT
ERCPP		ERCPP

ERCPT		ERCPT
ERDPH		ERDPH
ERDTH		ERDTH
ERPCP		ERPCP
ERPCT		ERPCT
ERPDCP		ERPDCP
ERPDCT		ERPDCT
ERPDP		ERPDP
ERPDT		ERPDT
ERPH		ERPH
ERPPH		ERPPH
ERPSP		ERPSP
ERPST		ERPST
ERPTH		ERPTH
ERRPP		ERRPP
ERRPT		ERRPT
ERSPP		ERSPP
ERSPT		ERSPT
ERTH		ERTH
ESPH		ESPH
ESTH		ESTH
ETH		ETH
ETHT	<==>	ETHT
F77_NETW_MAX_PACKET_LEN		F77_NETW_MAX_PACKET_LEN
F77_NETW_MAX_WORKERS		F77_NETW_MAX_WORKERS
F77_NETW_NUM_WORKERS		F77_NETW_NUM_WORKERS
F77_NETW_RESERVED		F77_NETW_RESERVED
F77_NETW_SEND		F77_NETW_SEND
FACTOR		FACTOR
FNP	=>	FNP
H		H
HAW		HAW
I	<==>	I
IC	<==>	IC
ID	=>	ID
IDD	=>	IDD
IDG		IDG
II	=>	II
IM		IM
INTEGER_LENGTH		INTEGER_LENGTH
I_TOTAL	<==>	I_TOTAL
J	<==>	J
JB	=>	JB
JE	=>	JE
K	=>	K
LCNPAT	=>	LCNPAT
LCORNR	=>	LCORNR
LCYL		LCYL
LDC	=>	LDC
LDEBUG		LDEBUG
LDRC	=>	LDRC
LGRND	=>	LGRND
LIHD	=>	LIHD
LOUT	=>	LOUT
LPLA	=>	LPLA
LPRAD		LPRAD
LRANG		LRANG
LRDC	=>	LRDC
LRFC	=>	LRFC
LRFI	=>	LRFI
LRFS	=>	LRFS
LSHD	=>	LSHD
LSLOPE	=>	LSLOPE
LSOR	=>	LSOR
LSRFC	=>	LSRFC
LSTD		LSTD
LSTS		LSTS
LSURF	=>	LSURF
LTEST		LTEST
ME	<==>	ME
MEP	=>	MEP

MEX	<==>	MEX
MP	<==>	MP
MPH	<==>	MPH
MPX	=>	MPX
MPXR	=>	MPXR
MS	=>	MS
PDCR	=>	PDCR
PHSR		PHSR
PHWR		PHWR
PI	<==>	PI
PRAD		PRAD
P_DFRFPT1W		P_DFRFPT1W
P_DFRFPT2W		P_DFRFPT2W
P_OUTP1W		P_OUTP1W
P_OUTP2W		P_OUTP2W
P_PCLYRTW		P_PCLYRTW
P_PRIOUTW		P_PRIOUTW
P_RCLDPLW		P_RCLDPLW
P_REFBPW		P_REFBPW
P_RFDFPTW		P_RFDFPTW
P_RFPTCLW		P_RFPTCLW
P_TANGW		P_TANGW
RANG		RANG
REAL_LENGTH		REAL_LENGTH
RESULTS		RESULTS
RPD	<==>	RPD
SAS		SAS
SASP		SASP
SNC	=>	SNC
SPS		SPS
STHS		STHS
SYNC		SYNC
TDCR	=>	TDCR
THP	=>	THP
THSR		THSR
TOMASTER		TOMASTER
TOOUTPUT		TOOUTPUT
TPI	<==>	TPI
TPPD	=>	TPPD
UCD	=>	UCD
UDC	=>	UDC
V	=>	V
VCD	=>	VCD
VDC	=>	VDC
VMAG	=>	VMAG
VN	=>	VN
VP	=>	VP
VTI	=>	VTI
VTS	=>	VTS
VXI	=>	VXI
VXIC	=>	VXIC
VXS	=>	VXS
WI	=>	WI
WL		WL
WORKDONE		WORKDONE
X	=>	X
XCL		XCL
XI	=>	XI
XIC	=>	XIC
XPC	=>	XPC
XS	<==>	XS
YCL		YCL
YPC	=>	YPC
ZC	=>	ZC
ZCL		ZCL
ZPC	=>	ZPC

BScat Appendix (ii) - RELATE output from pre-loop calculations

BSCAT	transfer method	WORKER
BCD	=>	BCD
BD	=>	BD
CJ	<==>	CJ
CPI4	=>	CPI4
DDC	=>	DDC
DPR	<==>	DPR
DTDC	=>	DTDC
DTI	=>	DTI
DTS	=>	DTS
FNP	=>	FNP
I	<==>	I
ID	=>	ID
IDD	=>	IDD
II	=>	II
J	<==>	J
JB	=>	JB
JE	=>	JE
LDC	=>	LDC
LIHD	=>	LIHD
LSHD	=>	LSHD
LSRFC	=>	LSRFC
LSURF	=>	LSURF
MPH	<==>	MPH
MS	=>	MS
PDCR	=>	PDCR
PI	<==>	PI
RPD	<==>	RPD
TDCR	=>	TDCR
THP	=>	THP
TPI	<==>	TPI
UCD	=>	UCD
UDC	=>	UDC
V	=>	V
VCD	=>	VCD
VDC	=>	VDC
VMAG	=>	VMAG
VN	=>	VN
VP	=>	VP
VTI	=>	VTI
VTS	=>	VTS
VXI	=>	VXI
VXIC	=>	VXIC
VXS	=>	VXS
WI	=>	WI
X	=>	X
XI	=>	XI
XIC	=>	XIC
XS	<==>	XS

BScat Appendix (iii) - RELATE output from post-loop calculations

```
BSCAT                    transfer method   WORKER
CJ                           =>            CJ
CPI4                         =>            CPI4
DPR                         <==>           DPR
EPHT                        <=             EPHT
ETHT                        <=             ETHT
PI                          <==>           PI
RPD                          =>            RPD
TPI                         <==>           TPI
```

BScat Appendix (iv) - List Of Protocol Values

```
C
C     COMMUNICATION CONSTANTS - PRIMARY PROTOCOL
C
      INTEGER Integer_Length, Real_Length, Complex_Length,
     *        ToMaster, ToOutput, Results, WorkDone, Sync, Update
      PARAMETER (Integer_Length=4, Real_Length=4, Complex_Length=8,
     *           ToMaster=-1, ToOutput=-2,
     *           Results=-5,  WorkDone=-6, Sync=-7, Update=-8)
C
C     SECONDARY PROTOCOL:
C     LABEL CONSTANTS FOR RUN TIME OUTPUT
C
      INTEGER   P_PCLYRTW,  P_PRIOUTW, P_TANGW,  P_DFRFPT1W,
     *          P_DFRFPT2W, P_RCLDPLW, P_REFBPW, P_RFDFPTW,
     *          P_RFPTCLW,  P_Outp1W,  P_Outp2W
      PARAMETER (P_PCLYRTW=1,  P_PRIOUTW=2,  P_TANGW=3,
     *           P_DFRFPT1W=4, P_DFRFPT2W=5, P_RCLDPLW=6,
     *           P_REFBPW=7,   P_RFDFPTW=8,  P_RFPTCLW=9,
     *           P_Outp1W=10,  P_Outp2W=11 )
```

BScat Appendix (v) - Description of the Master Process

The Master Shell Process

The master shell process assigns work as follows:

1) Create the next work packet for a worker.
2) Claim a free worker, using the SAPF library routine of Appendix A.
3) Send the work packet to the worker claimed.
4) If there is still work left to be done, return to step (1) otherwise continue with step (5).
5) Claim a free worker as in step (2), and send it a message that its results are required. This must be performed until all the workers have been sent the request for its results.

6) Try to decrement the semaphore **Results_Sema.** This semaphore is initially 0 and will only be incremented by RECEIVER once all the results have been received from the workers. The MASTER process will thus be forced to wait until RECEIVER has all the results. This process results in synchronisation between the two processes.

7) Output all the data as collected by RECEIVE and continue as normal.

The flow chart of the Master Shell is illustrated in Figure 34.

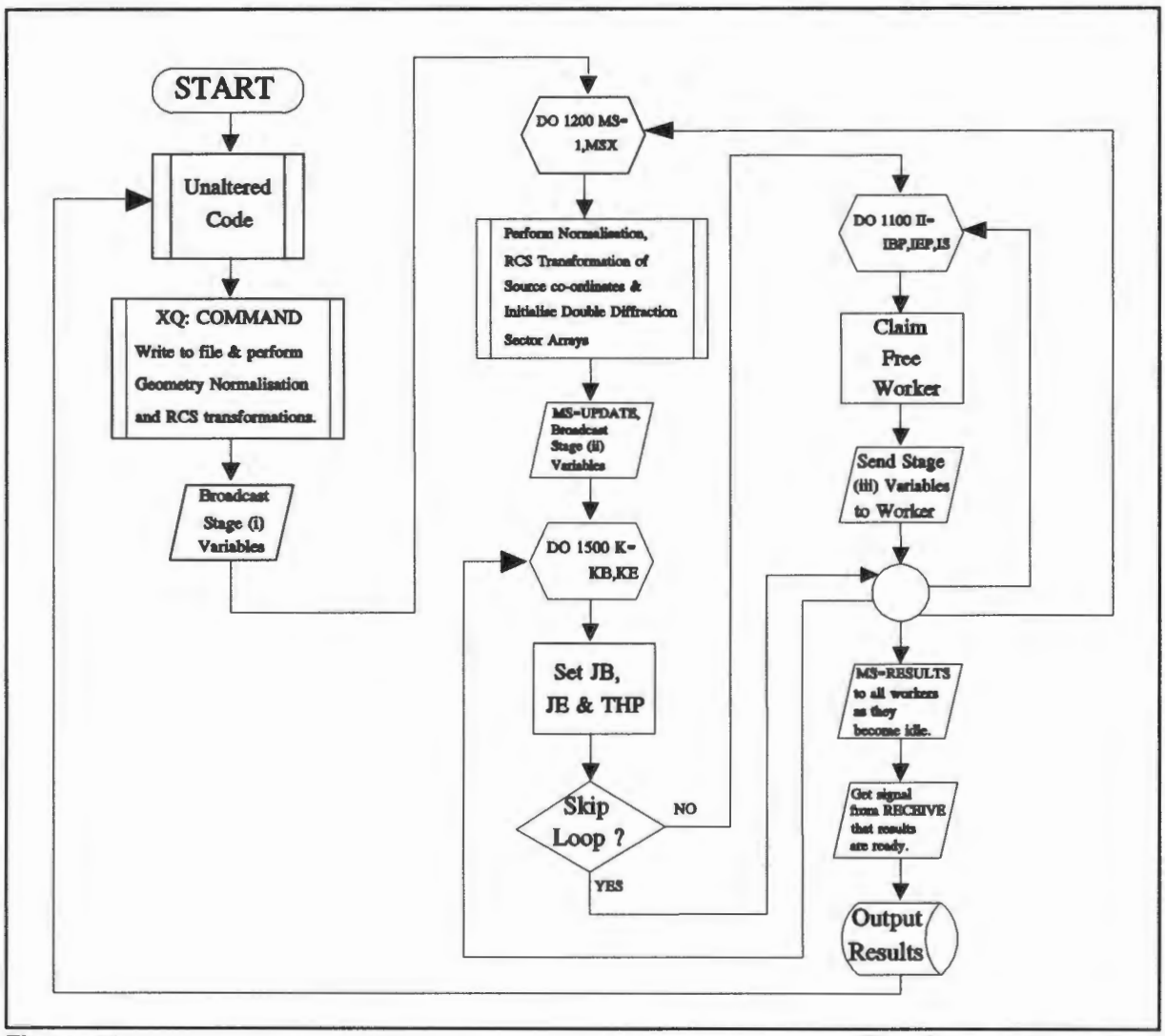

Figure 34 Flow Chart of Master Shell

The following steps corresponding to the process/input symbols of Figure 34 given on the right.

Step	Process/Input Symbol
(1)	The various combinations of all the flowchart symbols, except for the ones indicated below.
(2),(3),(4)	The "**Claim Free Worker**" process symbol.
(5)	The "**Send Stage 3 Variables to Worker**" I/O symbol.
(6)	The iteration marker containing "**1100, 1500** and **1200**".
(7)	The "**MS=RESULTS to all workers as the become idle**" I/O symbol.
(8)	The "**Get signal from RECEIVER that results are ready**" I/O symbol.
(9)	The "**Output Results**" disk access symbol.

As may be seen from the flowchart in Figure 34, the master uses a method whereby it can inform the worker that the message it just has received is not a work packet, but a request for results or an update of information. This is easily implemented as the value of **MS** may never be less than one. Therefore if a constant negative number were to be sent to the worker, the worker could interpret this as a command, such as a request for the results etc. Different negative constants could have different meanings. If the value of **MS** sent is positive, a standard work packet is assumed. The value used as protocols are listed in Appendix C(iv). If **MS=Results** is sent to the worker, the worker will respond by sending its totals of the arrays **ETHT** and **EPHT** and re-initialise itself waiting for the initial broadcast of data from Stage (i) of Section 11.4.1. If **MS=Update** were sent, the worker would then expect the data as sent in Stage (ii). Hence the master, each time **MS** is incremented, will sent an **Update** followed by the data which has been changed. Once the **MS** loop has been completed, the protocol **MS=Results** is sent to each worker once it becomes free and the RECEIVER process then collects and collates all the **ETHT** and **EPHT** totals before signalling the Master Shell process that it may continue.

The RECEIVER process

The RECEIVER process is spawned by the Master just after the Stage (i) variables of Section 11.4.1 have been sent. As discussed above, the RECEIVER task is responsible for receiving all the messages from the workers and does this in parallel to the Master. The workers use a secondary protocol (the primary protocol is used in RECEIVE_NETWORK to specify whether the message is I/O bound, or for the master) to messages to RECEIVER.

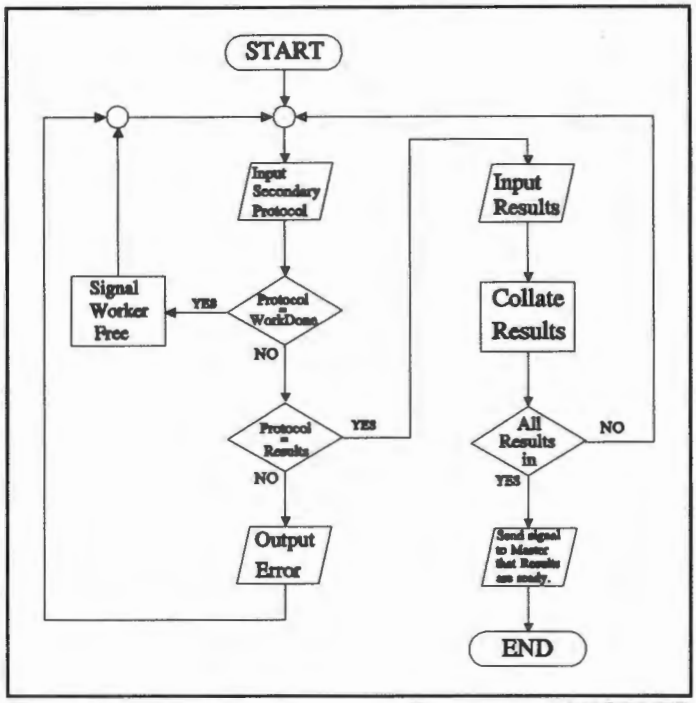

Figure 35 Flowchart of RECEIVER

If the protocol **WorkDone** is received, RECEIVER will release the worker which returned the results using the routines of Appendix A. If, at this point, the master was waiting for a worker to become idle, it will continue and claim the now idle worker and send it the work packet it created.

If the message **Results** (both a primary and secondary protocol) is received, the **EPHT** and **ETHT** totals sent by the worker are received and added to the master's running total of these two variables. Any other protocol received is an error. The receiver also keeps track of the number of result packets it has received. If there are *N* workers, then *N* result packets must be received by RECEIVER in total. Hence, when RECEIVER has received the same number of result packets as there are workers, it can justify that the totals of **EPHT** and **ETHT** has kept are the final results required. Thus RECEIVER can signal the Master Shell that all the results have been received. After doing the latter RECEIVER is no longer required and therefore terminates. The flowchart for RECEIVER is given in Figure 35.

BScat Appendix (vi) - Description of the Worker Process

A flowchart of the worker process is illustrated in Figure 36. The Stage (i) variables of Section 11.4.1 are input when broadcasted by the master. The worker then sits in its GET_WORK/DO_WORK loop. As discussed previously, the value of **MS** received determines whether actual work is to be performed, whether the Stage (ii) variables are to be input, or finally whether the master has requested the results from the worker and the worker should therefore oblige. In the latter case, the worker, after sending the results, "restarts" and waits for the next set of work from the master. The whole work process is then repeated.

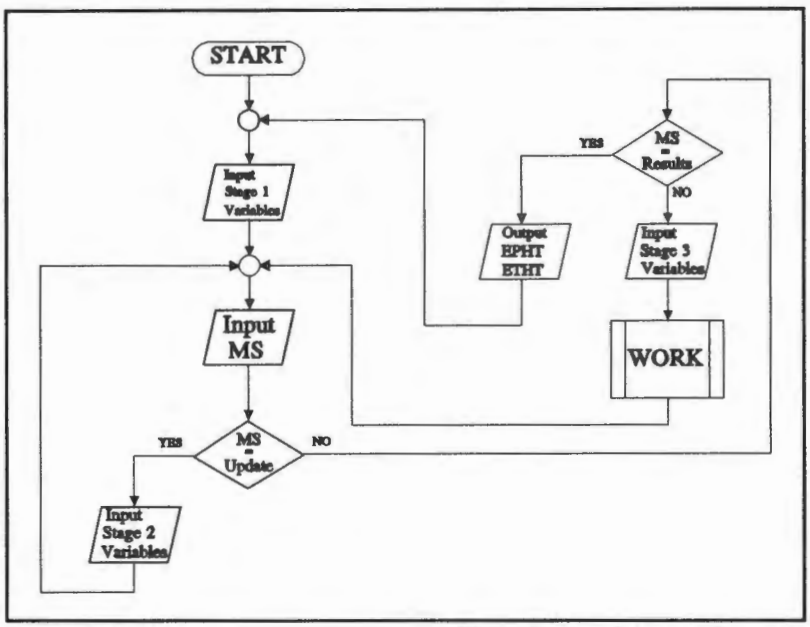

Figure 36 Flowchart of Worker Process

On receiving **MS**, the worker checks whether **MS = Update** in which case the Stage (ii) variables will be updated. This is performed each time MS has been changed. After all the Stage (ii) variables have been updated, the worker returns to an idle state waiting for the next **MS**.

If **MS = Results** is received, the worker immediately returns the **Results** message along with the running totals it kept of **EPHT** and **ETHT**. The worker then "restarts", resetting the totals of **EPHT** and **ETHT** and waits for the next set of work.

Any other value of **MS** is assumed to be a valid value (greater than 0) and therefore the remainder of the Stage (iii) variables are expected. The worker performs the work

required and, on completion, includes the results to the workers' local total of **EPHT** and **ETHT**. The worker then sends the secondary protocol **WorkDone** to the master, signalling the master that the worker is now idle.

Since the messages sent by the worker to the master are split into several smaller parts (e.g. Several Small Variables for Output), each group of messages is always prefixed by a call to F77_NETW_USE and post-fixed by a call to F77_NETW_FREE. This ensures that no other messages from other workers will be received while RECEIVER is gathering either data for exception I/O, results, or simply the message indicating a worker has become free. As the message sizes are small, and the exception messages are few and far between (the majority being sent on termination of the GET_WORK/DO_WORK loop), performance of the network will not be much effected.

A typical message may consist of **(ToMaster, WorkDone)** or **(ToOutput, Secondary_Protocol, Related_Data, Related_Data, Related_Data)**. Thus it is easy to see why the network has to be reserved before messages are sent from a worker. If this were not the case, consider the example where 2 workers transmit **(ToMaster, WorkDone)** and **(ToMaster, Results, EPHT, ETHT)** simultaneously. RECEIVER may receive the messages in the order (subscripts denote originator) **(ToMaster_1, ToMaster_2, Results_2, WorkDone_1, EPHT_2, ETHT_2)** which will have to be unscrambled before any sense can be made of the messages. Imagine the unscrambling of messages from 10 workers. Thus, by each worker reserving the network before sending each group of messages, the worker ensures that messages arrive in the correct order. For example, the message received by the master may be either of the following: **(ToMaster_1, WorkDone_1, ToMaster_2, Results_2, EPHT_2, ETHT_2)** or **(ToMaster_2, Results_2, EPHT_2, ETHT_2, ToMaster_1, WorkDone_1)**. This makes more sense.

Exceptions

As workers can reach exception cases, such as roots of equations not being found or too many iterations performed, these cases must be handled and reported on. These

exception cases occur where either some sort of error has developed or the user is to receive a warning that a certain "undesirable" condition exists due to either accidental or deliberate "mistakes" in the input data. Workers cannot perform any I/O themselves and hence must inform the master of these conditions in order that the master may perform the required I/O.

Several different forms of I/O exist, the most common being debug statements. As the number and type of debug statements were large and varied, it was decided to exclude debug statements. All debug statements were therefore commented out of the code (possibly to be included at a later stage). The remaining I/O messages therefore had to be implemented. The first problem was an overlap between routine which performed I/O in both the worker and the master code. This was overcome by duplicating the routines, re-naming the second routine by post-fixing the current name by a "**W**" (for Worker). For example, TANG is used in both thus there are two procedures, TANG and TANGW (essentially the same) for the master and worker respectively. The second routine then had the I/O operations commented out and a secondary I/O protocol created and sent, along with the data (if any), to the master with the primary protocol **ToOutput.** The I/O operations were copied to RECEIVE_NETWORK which, for a given secondary I/O protocol, would receive and output the data as required.

The secondary I/O protocol is listed in Appendix C(iv) with the following naming convention. The protocol name would correspond to the name of the routine in the worker which would request the I/O operation. In order to prevent confusion, all secondary I/O protocol is pre-fixed by a "**P_**". In cases where more than one different type of I/O is required by the same routine, a number would appear at the end of the name, before the "**W**" post-fix (if present).

Index

Airflow Modelling 79
Boot 61
BSCAT 105, 116
    analysis 117
    parallelisation 116
Code structuring 99
Communication 39
Configurer 112
CSP 14
Data
    communication 120
    dependency 92
    exceptions 122
    relationships 103
Deadlock 32, 36, 40
Efficiency 5
EPL 14
ESPRIT 13
Farm
    construction - method 1 31
    construction - method 2 33
    construction - method 3 34
    construction - method 4 34
    construction - method 5 35
    construction - method 6 38
    SAPF 49
Flood-Fill 47
Foot 77
Fortran 15
Head 70
HELIOS 17
INMOS 10
Languages 14
Link Identification 66
Links 11
MAKECONF 52, 112
Master Process 25
Memory Size 68
Microway 80
Mouth 75
Navier Stokes 79
Networks 22
OCCAM 12, 14
    razor 14
Parallelisation
    BSCAT 116
Priority 12
Process 20
    communication 20
    farm 25
    master 25
    network 22
    placement 23

priority . . . 12
router . . . 27
synchronisation . . . 21
worker . . . 26
Quadputer . . . 80
Reset . . . 61
Router . . . 27
priority . . . 28
SAPF . . . 1, 49, 98
data relationships . . . 103
farm construction . . . 52
farm structure . . . 49
implementation . . . 111
messages . . . 50
output . . . 102
Server . . . 59, 145
host . . . 60
transputer . . . 64
user manual . . . 145
Simulation . . . 79
Speedup . . . 5
airflow . . . 94
BSCAT . . . 127
Static checking . . . 100
T800 . . . 10, 11
architecture . . . 11
Tail . . . 73
Token Exchange . . . 34
Transputer . . . 10
Transputer Typing . . . 69
Worker
selection . . . 106, 118
types . . . 107
Worker Process . . . 26
Worm . . . 66
breadth first . . . 73
depth first . . . 70