University of Southampton

Software Migration Aids On Transputer Arrays

by

Andrew James Jackson

A thesis submitted for the degree of
Master of Philosophy

in the
Faculty of Engineering
Department of Electronics and Computer Science

25th September 1991

University of Southampton

ABSTRACT

Faculty of Engineering

Electronics and Computer Science
Master of Philosophy

Software Migration Aids On Transputer Arrays

by Andrew James Jackson

The advent of practical large scale parallel computing has led to a number of advances in thinking and to many new problems. One of the main problems of using massively parallel machines is that of unfamiliarity with code distribution for many programmers. Ideally a compiler would parallelise the users sequential code without any knowledge of special parallel considerations. As no automatically parallelising compilers are available yet, much work is underway to provide tools which allow a programmer to utilise the new technology with the minimum effort. The work includes language extensions, new languages and communication harnesses/templates for certain parallel paradigms. This thesis presents two harnesses written to allow 'coarse farm' and 'geometric' paradigms to be implemented on transputer arrays. A brief study is then given of a selection of languages, harnesses and other tools presently available for programming on transputers hosted by IBM Personal Computer systems (or compatibles).

Contents

1 Introduction 8
- 1.1 Sequential to Parallel Programming . . . 8
- 1.2 The Transputer and Occam . . . 8
- 1.3 'Alien' Languages . . . 11
- 1.4 Standard Parallel Paradigms . . . 12
- 1.5 Philosophy Behind Migration Aids . . . 13
- 1.6 Presently Available Migration Aids. . . . 13
- 1.7 Plan of This Thesis. . . . 14

2 Coarse Farm Harness 15
- 2.1 Adjustments to the Standard Farm Paradigm . . . 15
- 2.2 Perceived User Requirements . . . 16
- 2.3 Coarse Farm Harness . . . 16
  - 2.3.1 Outline of Processes . . . 16
  - 2.3.2 Communication Strategy . . . 19
  - 2.3.3 Writing for the Harness . . . 21
  - 2.3.4 Functionality and Implementation Limits . . . 21
- 2.4 'Benchmark' Case Study . . . 22
  - 2.4.1 The Code . . . 22
  - 2.4.2 The Implementation . . . 23
  - 2.4.3 The Results . . . 23
  - 2.4.4 The Conclusions . . . 29
- 2.5 Diffusion Limited Aggregation (DLA) Case Study . . . 32
  - 2.5.1 The Problem . . . 32
  - 2.5.2 The Code . . . 32
  - 2.5.3 The Implementation . . . 33
  - 2.5.4 The Results . . . 33
  - 2.5.5 The Conclusions . . . 34
- 2.6 Summary of the Coarse Farm Harness. . . . 35

3 Geometric Harness 36
- 3.1 The Geometric Paradigm . . . 36
- 3.2 Perceived User Requirements . . . 36
- 3.3 The Geometric Harness . . . 37
  - 3.3.1 Outline of Processes . . . 37
  - 3.3.2 Communication Strategy . . . 40
  - 3.3.3 Writing for the Geometric Harness . . . 41
  - 3.3.4 Functionality and Implementation Limits . . . 42
- 3.4 'Benchmark' Case Study . . . 43
  - 3.4.1 The Code . . . 43
  - 3.4.2 The Implementation . . . 44

3.4.3 The Results . . . 44
3.4.4 The Conclusions . . . 45
3.5 Conway's Game of Life Case Study . . . 46
3.5.1 A Description . . . 46
3.5.2 The Code . . . 47
3.5.3 The Implementation . . . 48
3.5.4 The Results . . . 48
3.5.5 The Conclusions . . . 49
3.6 Overview of the Geometric Harness . . . 49

4 Review of other Migration Aids and Languages 51
4.1 Introduction and Policy . . . 51
4.2 The Test Applications . . . 51
4.3 Languages . . . 51
4.3.1 The Hardware Used in the Language Tests . . . 52
4.3.2 The New INMOS occam Toolset (D7205) . . . 52
4.3.3 The 3L Parallel FORTRAN Language (v2.0) . . . 56
4.3.4 The Par.C Language (v1.31) . . . 58
4.4 General Harnesses . . . 60
4.4.1 Euler Cycle Configuration Language (v1.08) . . . 61
4.4.2 Virtual Channel Router (v2.0) . . . 64
4.5 Programming Support Environments . . . 66
4.5.1 Express (v3.0) . . . 66
4.6 Operating Systems . . . 70
4.6.1 Helios (v1.1a) . . . 70
4.7 Conclusions of the Review . . . 72

5 Concluding Remarks 74
5.1 Further Work on the Coarse Farm Harness . . . 74
5.2 Further Work on the Geometric Harness . . . 74
5.3 The Need for Migration Aids . . . 74

6 Acknowledgments 75

Bibliography 76

A Coarse Farm Harness Test Code and Data Tables 78

B DLA Code for Coarse Farm Harness 84

C Geometric Harness Benchmark Code. 93

D Geometric Harness Life Code. 99

E Migration Aids Review Code. 105
E.1 The New Occam Toolkit. . . . 105
E.1.1 The Coarse Farm Test. . . . 105
E.1.2 The Geometric Test. . . . 109
E.2 3L Parallel FORTRAN. . . . 113
E.2.1 The Coarse Farm Test. . . . 113
E.2.2 The Geometric Test. . . . 120

E.3 Par.C. 127
E.3.1 The Coarse Farm Test. 127
E.3.2 The Geometric Test. 128
E.4 ECCL 130
E.4.1 The Coarse Farm Test. 130
E.4.2 The Geometric Test. 133
E.5 VCR. 138
E.5.1 The Coarse Farm Test. 138
E.5.2 The Geometric Test 141
E.6 Express. 144
E.6.1 The Coarse Farm Test. 144
E.6.2 The Geometric Test. 145
E.7 Helios. 147
E.7.1 The Coarse Farm Test. 147
E.7.2 The Geometric Test. 148

F List of Company Addresses. **151**

List of Figures

1.1 The Transputer Chip. 9
1.2 Transputer Networks. 10

2.1 Process Diagram for Master Processor. 17
2.2 Process Diagram for Worker Processor. 19
2.3 Benchmark Test 1: Single screen output per job. 24
2.4 Benchmark Test 2: Ten screen outputs per job. 24
2.5 Benchmark Test 3: 100000 floating point multiplies per job. 25
2.6 Benchmark Test 4: 1000000 floating point multiplies per job. 25
2.7 Benchmark Test 5: 1 Kbyte of file i/o per job. 26
2.8 Benchmark Test 6: 10 Kbytes of file i/o per job. 26
2.9 Benchmark Test 7: 100 Kbytes of file i/o per job. 27
2.10 Varying the Number of Processors: Test 1 with 200 jobs. 27
2.11 Varying the Number of Processors: Test 2 with 20 jobs. 28
2.12 Varying the Number of Processors: Test 3 with 100 jobs. 28
2.13 Varying the Number of Processors: Test 4 with 15 jobs. 30
2.14 Varying the Number of Processors: Test 5 with 15 jobs. 30
2.15 Varying the Number of Processors: Test 6 with 10 jobs. 31

3.1 Edge Swap Illustration. 37
3.2 Processor Network for Geometric Harness. 38
3.3 Process Diagram for Master Processor. 39
3.4 Process Diagram for Worker Processor. 40

4.1 Hardware for Coarse Farm Test. 52
4.2 Hardware for Geometric Test. 53
4.3 Process Connections for the ECCL Coarse Farm Test. 63
4.4 Process Connections for the ECCL Geometric Test. 64
4.5 Process Connections for the VCR Coarse Farm Test. 66
4.6 Process Connections for the VCR Geometric Test. 67
4.7 Hardware for the Express System. 68
4.8 Stream Connections for the Helios Geometric Test. 71

List of Tables

2.1 Standalone DLA results. 33
2.2 Faster processors on one million particle cluster. 33
2.3 Coarse Farm Harness DLA. (Continued in table 2.4) 34
2.4 Coarse Farm Harness DLA continued. 35

3.1 All to one test. 45
3.2 One to all test. 45
3.3 All to all test. 45
3.4 Geometric test simulation test. 46
3.5 Average message handling times. 46
3.6 Mflop time estimates. 46
3.7 Stand Alone Life (200 × 200 grid, with display). 48
3.8 Stand Alone Life (200 × 200 grid, no display). 49
3.9 Stand Alone Life (400 × 400 grid, no display). 49
3.10 Geometric Harness Life (200 × 200 grid, with display). 49
3.11 Geometric Harness Life (200 × 200 grid, no display). 50
3.12 Geometric Harness Life (400 × 400 grid, no display). 50

4.1 Coarse Farm Test for the New Occam Toolkit. 55
4.2 Geometric Test for the New Occam Toolkit. 55
4.3 Coarse Farm Test for the 3L FORTRAN. 57
4.4 Geometric Test for the 3L FORTRAN. 58
4.5 Coarse Farm Test for the Par.C Language. 60
4.6 Geometric Test for the Par.C Language. 60
4.7 Coarse Farm Test for ECCL. 62
4.8 Geometric Test for ECCL. 63
4.9 Coarse Farm Test for VCR. 65
4.10 Geometric Test for VCR. 65
4.11 Coarse Farm Test for the Express System. 69
4.12 Geometric Test for the Express System. 69
4.13 Coarse Farm Test for Helios. 71
4.14 Geometric Test for Helios. 72
4.15 System Comparison for Coarse Farm Test with 100 loops. 72
4.16 System Comparison for Geometric Test with 100 loops. 73

A.1 Coarse Farm Benchmark Test 1. 79
A.2 Coarse Farm Benchmark Test 2. 80
A.3 Coarse Farm Benchmark Test 3. 80
A.4 Coarse Farm Benchmark Test 4. 80
A.5 Coarse Farm Benchmark Test 5. 81
A.6 Coarse Farm Benchmark Test 6. 81
A.7 Coarse Farm Benchmark Test 7. 82

A.8 Coarse Farm Benchmark Test 8. 82
A.9 Coarse Farm Benchmark Test 9. 82
A.10 Coarse Farm Benchmark Test 10. 82
A.11 Coarse Farm Benchmark Test 11. 83
A.12 Coarse Farm Benchmark Test 12. 83
A.13 Coarse Farm Benchmark Test 13. 83

1 Introduction

1.1 Sequential to Parallel Programming

The arrival of affordable massively parallel machines has caused a major rethinking of the methods of programming computers.

The traditional Von Neumann computers can only do one operation at a time. To program such a machine a list of instructions has to be written so that they may be executed by the single processing unit in a stream. This is a very unnatural way of thinking. In the real world things do not happen one at a time, everyday life is full of parallel occurrences. The first problem for a new parallel programmer is to overcome his or her sequential background and think parallel.

Programming a parallel machine forces the programmer to think of a number of new problems when coding. The time order in which events occur is no longer obvious when a number of processes are active at the same time. It is important to have control over your program and this new dimension to programming may be expressed as follows : A parallel program is a set of processes which are potentially active at the same time. These processes are a 'work force' and they are usually written in a traditional sequential manner with some form of communication system to transfer data from process to process. Once a work force of processes has been produced it does not amount to a working program. In addition to producing the code to do the job there has to be 'management' of the work force to make sure that the job is done in a sensible manner with maximum benefit to the end user (usually this means maximum speed). A management structure must be created to make sure that no single process gets out of step with the others and that the program communicates and terminates correctly. The management structure in a parallel program is much more complicated than the control structure in a sequential program. Even a simple parallel program needs careful consideration. This is the major difference between parallel and sequential programming.

1.2 The Transputer and Occam

The transputer is a VLSI chip produced by INMOS of Bristol in the United Kingdom. The name comes from the words transistor and computer. A transistor is a device which is used in large numbers to construct more complicated electronic systems. The transputer is a complete computer on a single silicon chip designed to be the building block for larger processing systems, hence the name.

Each transputer has processing power, memory and communications hardware on the chip. The diagram (figure 1.1) shows the major components of the T800 transputer. There is a 10 MIP, 32 bit integer processor and a 1.5 Mflop, 64 bit floating point processor. Communication is provided by four bidirectional INMOS serial links running at 10 or 20 Mbits per second, and there are 4 kilobytes of memory on the chip with

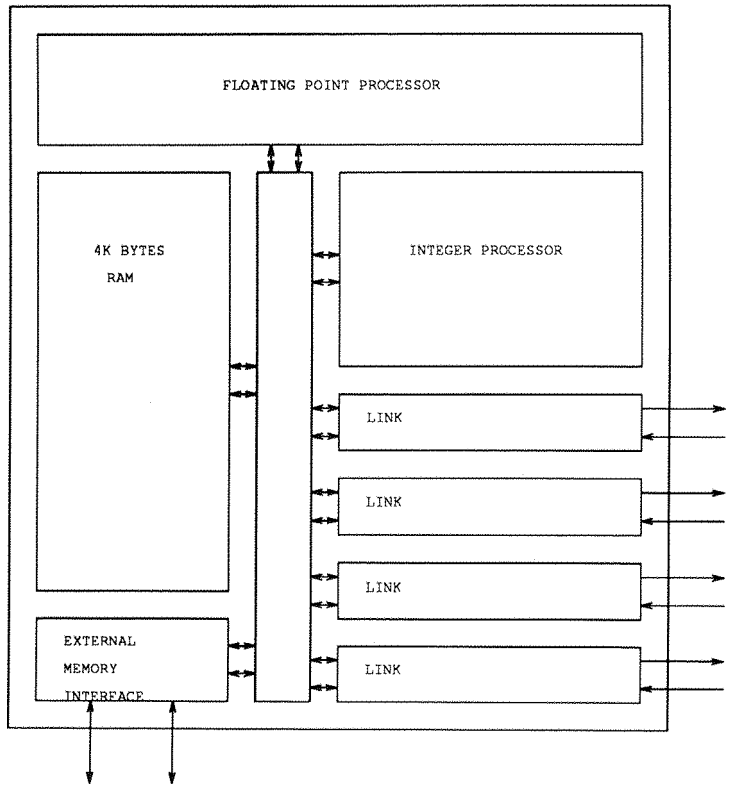

Figure 1.1: The Transputer Chip.

an external memory interface which allows memory up to the 32 bit address limit to be attached [1]. The transputer can thus be thought of as a processing element with four 'hands' which can connect to other transputers. It is obvious then that large networks of transputers can be built with many topologies. (figure 1.2)

Each transputer can execute a unique piece of code so a network of transputers is a multiple instruction, multiple data (MIMD) machine. There is no shared memory between transputers, all data transfer between processors is effected through the links. This feature of distributed memory is a great aid to evaluating the action of a system as there is no memory contention problem.

The transputer's instruction set is similar in concept to a reduced instruction set,'RISC', machine. The sixteen most used instructions are four bits long. Included in the sixteen base instructions is a 'prefix' instruction which doubles the size of the instruction to eight bits. Two prefixes give a twelve bit instruction and so forth. By using this facility the transputer can, potentially, have an infinite instruction set. Operation of the processor is via a three element evaluation stack rather than the more traditional register based operation. This allows for very fast context switching between two processes on the same processor [2].

The occam programming language was inspired by the mathematical model called Communicating Sequential Processes (CSP) [3]. It was developed by INMOS in conjunction with the transputer hardware to allow the programming of massively parallel systems. As well as the usual data types and constructs for a sequential programming language there are also elegant parallel constructs in the language. Parallel processes can be run on the same transputer or on a number of transputers. Communication is

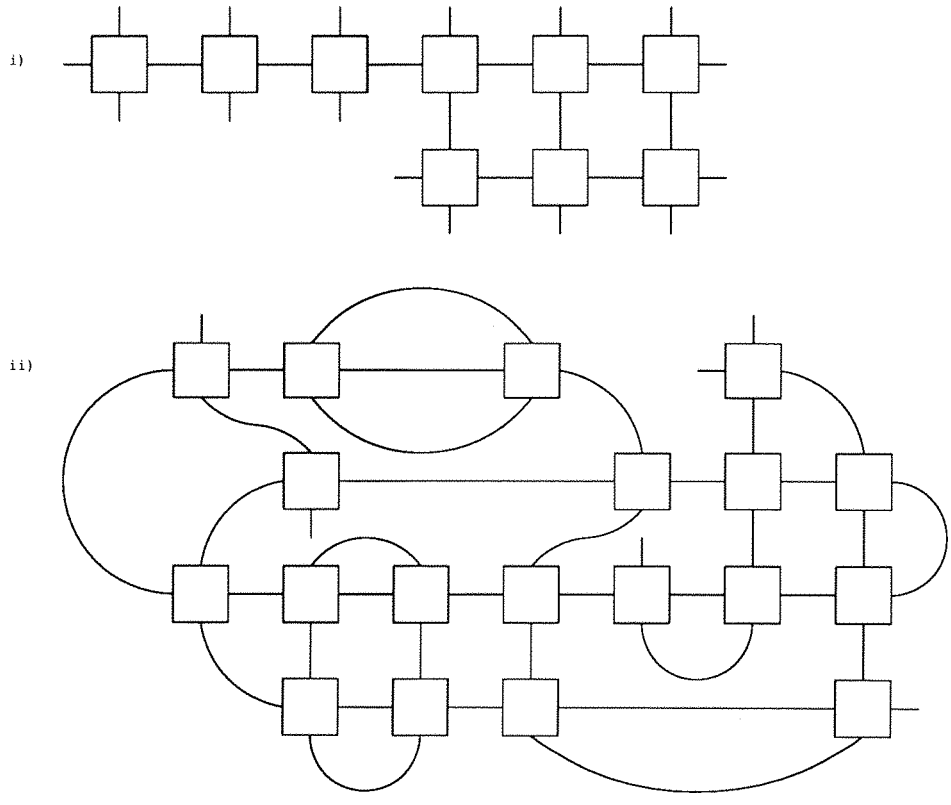

Figure 1.2: Transputer Networks.

handled by the channel data type which effects a point to point one way link between two processes.

Occam is a very simple language built from three basic operations

- Assignment of a value to a variable
- Output of a value down a channel
- Input of a value from a channel

From these simple operations a fully functional high level language has been developed [4]. Program statements are grouped into constructs which define how those statements will execute, in sequence (SEQ) or in parallel (PAR). Procedures can be built up and used as program statements in a familiar way.

Channels provide synchronised communication. A communication is an output on one end of a channel and an input on the other. All channels are one way, and the input end is in one process and the output end in another. Whichever end is reached first causes that process to wait until the other process has reached the communication on that channel as well. The communication will then take place and the processes will go their separate MIMD ways. It can be seen that having communications which operate in this manner aids greatly when thinking about programming. If you have an output without an input (or *vica versa*) the program will hang at the line where that communication occurs. This is called 'deadlock'. If the communications did not synchronise then the process with the failed communication would continue and debugging of the system would be more difficult a problem than it is. An excellent monograph on

deadlock freedom in communicating systems is contained in The Pursuit of Deadlock Freedom [5]. A problem with similar symptoms to deadlock is 'livelock', an infinite communications loop with no interaction with the user.

A parallel program is non-deterministic because of its parallel execution nature. For example, imagine a shared memory program with two processors. Processor A sets variable X to 3 and independently processor B checks the value of X and initiates a destruct sequence on the machine if the value is not 3. For months the program runs such that A always gets to X before B reads it. All is well until one day B runs hot and reaches X first. Self destruct occurs and as far as the program goes nothing has changed. This is a simplistic example. Real shared memory systems have various safeguards to prevent this sort of thing but it illustrates that timing issues have to be carefully considered and that the same program may produce different results from running the same code with the same data.

Because of the problems of nondeterminism, occam confines this type of behavior to only one construct (in normal operation) so that it may be easily controlled. This alternative (ALT) construct allows a number of inputs to be interrogated. If none are ready then the process waits until one of the communications can take place. When more than one input is ready at the same time a random choice is made between them. Only the use of inputs, not outputs, in this construct is allowed for implementation reasons.

So occam and the transputer, therefore, provides a simple and elegant model for parallel systems by virtue of such features as distributed memory, synchronised communication and isolation of the nondeterministic nature of parallel systems. The processor provides speed by operating many of its components (processors, link engines et cetera) in parallel, by its RISC like instruction set and fast context switching between processes.

1.3 'Alien' Languages

Occam and the transputer were developed together and there is a very close relationship between the two. Any other language that runs on a transputer is termed, by INMOS, to be 'alien'. Alien languages at present consist of FORTRAN 77, C, PASCAL and ADA. Work is underway on others such as PROLOG and MODULA-2. To use the parallel aspects of the transputer these languages have either procedural interfaces or language extensions. The company 3L of Edinburgh in Scotland produce a family of compilers (C, FORTRAN and PASCAL) with procedural interfaces to communication libraries and other transputer features. INMOS have an occam system which allows the importing and linking of alien compiled code from the 3L family of compilers with occam code. It is relatively easy using this system to mix the four languages. PARSEC of Leiden in Holland produce a C system for the transputer called Par.C. This has language extensions instead of the procedural interface. ALSYS of Henley in England produce an ADA compiler. ADA is a concurrent language by design.

The work in this thesis was conducted with the INMOS occam toolset D705B and the 3L Parallel FORTRAN version 2.0 using the language mixing facilities of the occam toolset mentioned above [6, chapter 9].

1.4 Standard Parallel Paradigms

Parallel programming, like any other subject, is a diverse field. A new and unique program could be produced for every problem undertaken. This method of progress is wasteful as many ideas from past problems can be reused. Research groups in the field have started to develop sets of parallel paradigms because of this. Having achieved a set of standard ways of programming a parallel machine problems can be categorised according to which paradigm they fit most closely. The list of standard paradigms is largely subjective [7].

At the University of Southampton (UK) we have used three broad paradigms. These are the farm, geometric and algorithmic paradigm. As these are the paradigms I will be referring to in this thesis I will explain them more fully.

The farm paradigm is used when a problem consists of a main loop with many iterations of the same piece of code. Ideally the result of each iteration will be independent of all the other results. This being the case the body of the loop can be replicated onto a number of processors and iterations carried out in parallel. A farmer processor is usually employed to package up the initial information needed by a loop iteration and send out this 'work packet' to a worker processor. When the worker has calculated the result for its packet it sends it back to the farmer. The cycle continues until all the work is done. Because of the nature of the transputer hardware if there is more calculation than communication in one of these cycles then communications may be hidden and high speedups are obtained. This is often the easiest method of making a sequential program parallel. Applications which use this paradigm include ray tracing [8] and Monte Carlo elementary particle event simulations [9].

The geometric decomposition paradigm is used when solving a problem involving large simulations with local interactions. Mini versions of the program are run on a network of processors (usually a grid) each handling a small portion of the simulation. Data from the edges of these small areas must be swapped with neighbours to allow a full simulation step to be calculated. The more local the interactions are the better the result, as a smaller area of neighbour data is then required for edge swapping. Provided that the interaction is local, and sensible sized subareas are used, this method will deliver good speed up with increasing processors. A control processor may be used in a similar way to the farm paradigm to synchronise simulation steps and collect data for display *et cetera*. Greater communication is required in a geometric decomposition than a farm and because of this more work is needed when porting a sequential code to this paradigm. The speedup is usually not as great as for a farm problem solution. The 2DXY [10] simulation of a liquid crystal is an example of an application using this paradigm.

If the two paradigms so far explained are not appropriate then an algorithmic decomposition specific to the problem is the last resort. This kind of decomposition usually takes the form of multiple pipelines. This is the hardest paradigm to port sequential code to as each problem is unique in its character. Examples of this type of paradigm include a vision system [11] and the Bouncing Balls demonstration [12].

Although the last paradigm is problem specific the first two have attributes which will be the same for all problems of their class. All farms will send out work packets, all geometric decompositions will edge swap to some degree and so on. There is a case, then, for writing a 'harness' for these paradigms to provide the usual facilities needed. Some performance penalties may be paid but the average applications programmer

will have much of the parallel programming burden removed by these 'migration aids'.

1.5 Philosophy Behind Migration Aids

In an ideal world compilers would be available to produce parallel executable code from standard sequentially written source code. Programs would be written as always, with an artificial 'single stream' of instructions with no reference to the underlying architecture of the hardware. There may be certain conventions used to make it easier for the compiler to 'parallelise' the code. This would be a similar situation to the present vectorising compilers available for CRAY and other supercomputers. Unfortunately parallelising compilers are not at present available.

The other extreme would be for the programmers to have to do everything themselves. This requires some detailed knowledge of the architecture of the transputer. All communications protocols *et cetera* would have to be set up by the programmer for the specific job in hand. This is the situation provided for by occam, which allows access to the low level hardware features required to program parallel processing transputer arrays.

What would be ideal is the first scenario : what we have to start from is the second.

In an effort to move towards the ideal situation, much work is underway to produce 'migration aids'. These are systems which allow the applications programmer to migrate his or her sequential code to a parallel machine. The first set of aids which became available consists of communication harnesses/programming templates for the farm and geometric paradigms. General harnesses for any paradigm are the next stage leading on to new languages, compilers and operating systems. At present there are general harnesses and parallel operating systems available with leading edge research into self-parallelising compilers and new languages in progress.

The philosophy behind this work is to progressively lighten the burden on an applications programmer, to abstract away from the specific features of the machine in use and allow the programmer to think about how to solve the problem rather than how to program the machine.

1.6 Presently Available Migration Aids.

At start of this thesis most of the migration aids that were available were for the farm paradigm. Meiko produce a FORTRAN compiler which allows the implementation of a farm [13]. The 3L family of compilers all have the Flood Fill Configurer system which allows the user to write two processes, a worker and a master, and have them automatically configured into a farm to run on any attached processor network [26]. FORTNET started life as a farm harness running on a chain of processors produced at Daresbury, UK [14]. With the provision of interprocessor communications between any two processors on the chain it is proposed as a general harness.

The other main stream of work going on at the start of this thesis was the development of general communications harnesses. These include ECCL [29] reviewed in its finished state later in this work and Tiny [15] from Edinburgh.

1.7 Plan of This Thesis.

After a general introduction contained in chapter 1 this thesis is organised as shown below.

Chapter 2 : Coarse Farm Harness. A harness is presented, which has been written by the author of this thesis, to implement a special case within the farm paradigm. The system allows the user to destribute a number of identical whole problems on a network of transputers with full transparent access to the single file server from all the processors. Full process descriptions, code templates and examples of use are given. Two case studies are then undertaken. The first is a 'benchmark' to evaluate the functionality and performance of the coarse farm harness. The second is a real application from the subject of chemistry.

Chapter 3 : Geometric Harness. A simple harness, again written by the author of this thesis, for implementing the geometric paradigm is presented. The harness is implemented on a torus of processors. General point to point communications are allowed between any two processors in the system and full process descriptions, code templates, *et cetera* are included in the chapter. The two case studies on the geometric harness were to evaluate its functionality using a 'benchmark' and to implement a typical geometric application, Conway's Game of Life.

Chapter 4 : Review of Other Migration Aids and Languages. In this chapter seven systems of various types are looked at. There are three languages, two general harnesses, a programming support environment and an operating system. Two tests were implemented on each system. The first is an attempt to communicate to the screen from a chain of processors in the way the Coarse Farm Harness of chapter 2 would allow. Secondly an idealised geometric test is used on a torus of processors. This was one of the tests used in chapter 3 on the Geometric Harness. A comparision of the systems is made, including the harnesses written by the author of this thesis, in terms of performance, facilities provided by the system and ease of use.

Chapter 5 : Concluding Remarks. Some final statements on each of the harnesses produced are made. Suggestions for the next steps in the development of the work are put forward.

The thesis concludes with acknowledgements and appendices containing example code for all the work undertaken.

2 Coarse Farm Harness

2.1 Adjustments to the Standard Farm Paradigm

The standard farm paradigm is to split up a single problem which has a central loop. The code body of the loop is the same for every iteration and only the start parameters change. In a parallel processing farm solution this body of the loop is removed and replicated onto a number of worker processors. A processor known as the farmer then assembles the initial parameters into work packets which are communicated to the workers. Results are sent back to the farmer. It helps if the results do not have to be in any particular order as then the worker outputs do not have to be sorted. This method increases the speed of execution of a single problem and requires some restructuring of the code.

For the coarse farm harness we take one step back. Instead of issuing packets for iterations from a single problem we issue packets for whole problems. Here the overall system which needs to be executed must consist of a large number of identical problems run with slightly different initial parameters. The purpose of this is to generate a large number of data sets. Speeding up of the generation of the data sets is achieved by the farming of whole problems and also by the fact that they execute independently in parallel. It can be seen that although the speed of execution of a single problem does not decrease, it is the fact that a number of the problems are run simultaneously that increases the throughput.

The coarse farm harness must provide all the services to each problem or job that is running. Each job must have access to the file systems, keyboard and screen *et cetera*. An effective and efficient method of managing requests to these shared resources must be implemented in the harness. Jobs in the coarse farm harness do not interact with each other, in the same way that packets of work in a standard farm do not interact. There is no need for results in the coarse farm harness to be sorted as each one is an individual job.

The efficiency of the coarse farm harness method has been call naive [7]. The efficiency used in this argument is the financial cost to computer performance ratio. It is concurred that using a parallel machine in this way is no better than a 'sequential machine using similar technology'. I would argue though that the thought behind the coarse farm harness is to increase throughput. The fact that the time for the production of data sets is greatly reduced can have very large advantages. Also as processors are added the speedup can be very good, in line with the usual farm results. Because of these good points I would refute the statement that 'this efficiency is illusionary'.

The coarse farm harness requires even fewer changes to an original sequential program than a standard farm decomposition. The whole program is put in a loop and slightly changed to take into account that a number of jobs may be accessing some shared resources. It will provide efficiency in terms of elapsed time as opposed to cost to performance ratio. In a world where time is money I feel that this is a worthwhile

system to produce.

2.2 Perceived User Requirements

The ideal system, as far as the user is concerned, requires no special action by the programmer and no integral knowledge of the transputer hardware on which the system will run. However as an automatic compiler is not available the user initially requires a template of the paradigm. This template must be simply expressed to allow the programmer to evaluate whether the facilities being offered by the harness are appropriate for his or her application. Harnesses are usually specialised in that they operate for one paradigm only. If the application to be implemented does not fit the paradigm for a certain harness then that tool should be discarded and a more suitable one found. More advanced communication harnesses are available for general use with all paradigms but a performance penalty is usually paid for the generality. General harnesses are frequently more difficult to write for as there is no set pattern for the user's code to fit into.

Once a tool appropriate to the problem has been found the harness should be as easy to write for as possible. All the communications should be provided and a clear specification of the interface given. An outline of the processes should be given with a clear indication of where the user generated processes should go. This gives an idea of how the harness works without too many technical details. The programmer will usually have to write one or two sequential pieces of code to fit into the harness.

A final template is needed for each piece of code to be written by the user. This gives the structure of the code to be inserted into the harness and examples of any procedures provided for use by the programmer. Any special techniques required by the harness need to be explained and examples shown.

To summarise, in the absence of a compiler or preprocessor to do everything automatically, the user needs a clear idea of what paradigm a harness works for and must make a decision as to whether it is appropriate for his or her application. After making that choice, process information is required to allow the programmer to know something about how the harness works. Finally, a fairly rigid template should be provided for any pieces of code to go into the harness and any special features illuminated by examples.

The coarse farm harness paradigm is to have a number of independent problems executing simultaneously, accessing shared file and i/o facilities producing independent data sets. The data sets are collected and analysed at a later time. Process information and code templates appear in the following sections.

2.3 Coarse Farm Harness

2.3.1 Outline of Processes

The coarse farm harness runs on a simple linear chain of processors. The processor at the head of the chain contains the master process, the rest have the worker process loaded onto them.

After the requirements have been finalised a process strategy has to be settled upon. The usual hardware considerations for the transputer are required. These include one process for each link-in or link-out engine. This gives maximum overlap

between communication and calculation. The worker will be handling two broad classes of communication, filer requests and replies, and harness commands. For this purpose, in the workers, the incoming communications will have to be filtered to take out harness commands, job identification numbers for example. As for all transputer programs the number of parallel processes on each single processor should be kept to a minimum. The above considerations lead to the following process designs.

The master process is called `termmux`. (figure 2.1)

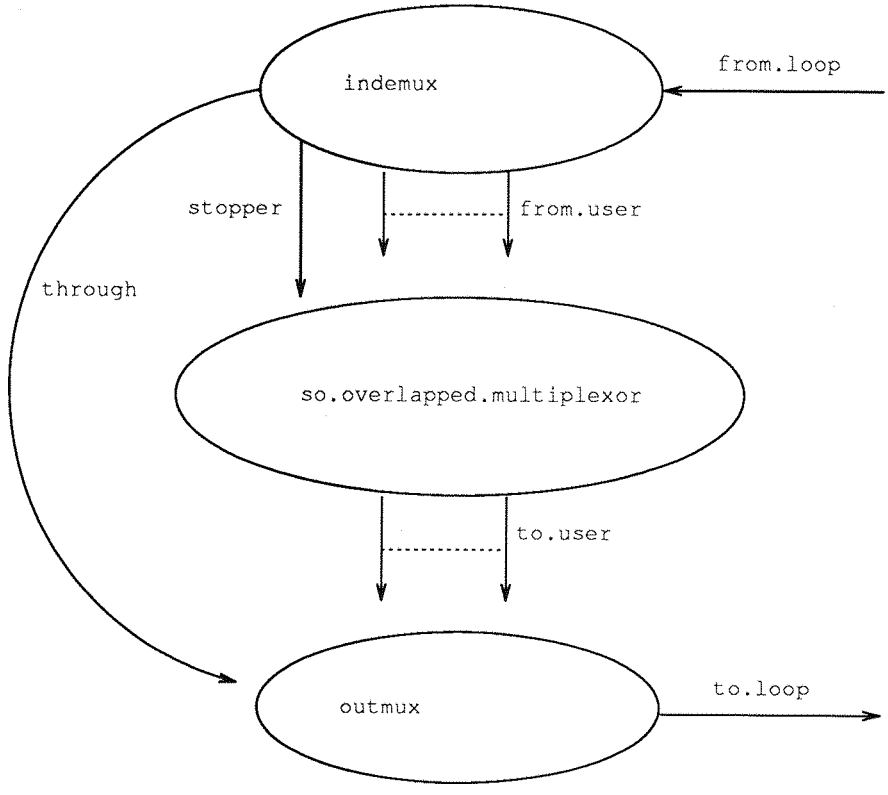

Figure 2.1: Process Diagram for Master Processor.

The processes fulfil the following purposes.

`indemux` : The major part of the work is carried out in `indemux`. Initially, before the branching in the main parallel (`PAR`) construct, the parent process, `termmux`, handles the first interactions with the user and inputting of the number of jobs. The number of jobs is then sent around the system. This allows each worker to calculate its initial `job.id` (job identification number) and thus start work straight away. `Indemux` then handles the calculation of new `job.ids`, after the main `PAR`. It passes them through the channel `through` to `outmux` which then issues the `job.ids` to the network. `Indemux` further handles incoming communications from the worker processors. The communication protocol is the INMOS iserver filer protocol [6] with a few integer tag additions to allow routing.

One of these tags is the processor identification number. There is an array of channels, `from.user`, which take filer requests and pass them on to the INMOS supplied `so.overlapped.multiplexor`. The array size is the same as the

number of workers and the processor identification number is used to index into this array of channels. Termination of the system is again initiated from `indemux` and passes through the channel `through`. From `outmux` the termination signal travels around the loop of workers and back to `indemux`, processes terminating as the signal passes. The channel `stopper` is then output to so that the `so.overlapped.multiplexor` process can terminate. With all the other components of the PAR terminated `indemux` can terminate itself and pass control back to the `termmux` process. Timing of the system is handled by the `indemux` process and the time data is returned to `termmux` when `indemux` terminates. `Termmux` then reports the time to the screen and the whole system terminates.

`outmux` : `Outmux` is a subordinate process in every way. It takes its control signals from `indemux` passing them on to the loop of workers as necessary. An array of server reply channels from `so.overlapped.multiplexor` provides answers to the requests issued via `indemux`. The index which a reply arrives on is used to set the appropriate processor identification tag for routing to the workers.

`so.overlapped.multiplexor`: `So.overlapped.multiplexor` is a routine provided by INMOS for multiplexing a number of processes onto the same file server. An array of input channels for the filer requests, `from.user`, is supplied with a similar array for filer replies, `to.user`. A request passed down `from.user[i]` causes the reply to appear on `to.user[i]`. A signal on channel `stopper` terminates the multiplexor and `from.filer,to.filer` are the usual channel pair for communication with the server program running on the host computer. The supplied multiplexor provides a queue for filer requests thus stopping saturation of any communication structure attached to it. Another multiplexor is supplied with the INMOS D705B which does not have the queueing mechanism. Tests with this non-queueing multiplexor have produced results which are the same as tests using the queueing multiplexor. (The queueing multiplexor was used in the benchmark tests.) This implies that the harness as implemented has a communication system which does not saturate.

The worker process (`worker`) is illustrated in figure 2.2. The `worker` process starts by getting the initial job identification number from the starting up `termmux` process which it passes to the `id.handler` processes via its parameter list. After setting up the first `job.id`, a PAR starts all of the processes.

`up.incomms` : `Up.incomms` takes in communications and passes any job identification numbers for the present processor to `id.handler`, which handles issuing of `job.ids` to the FORTRAN `fworker`. Any filer communications for the processor have their protocol tags stripped off and are passed to the `fworker` process. Any other communications (i.e. not for this processor) are passed to `up.outcomms` which passes them out of the processor.

`id.handler` : `Id.handler` takes in `job.ids` and then puts them out to `fworker`. It knows about the condition to terminate `fworker` (`job.id` = -1) and shuts down after it has shut down the `fworker` by passing on the signal.

`fworker` : `Fworker` contains the FORTRAN work program. Filer requests are issued through `from.fworker` to the communication system and replies returned via

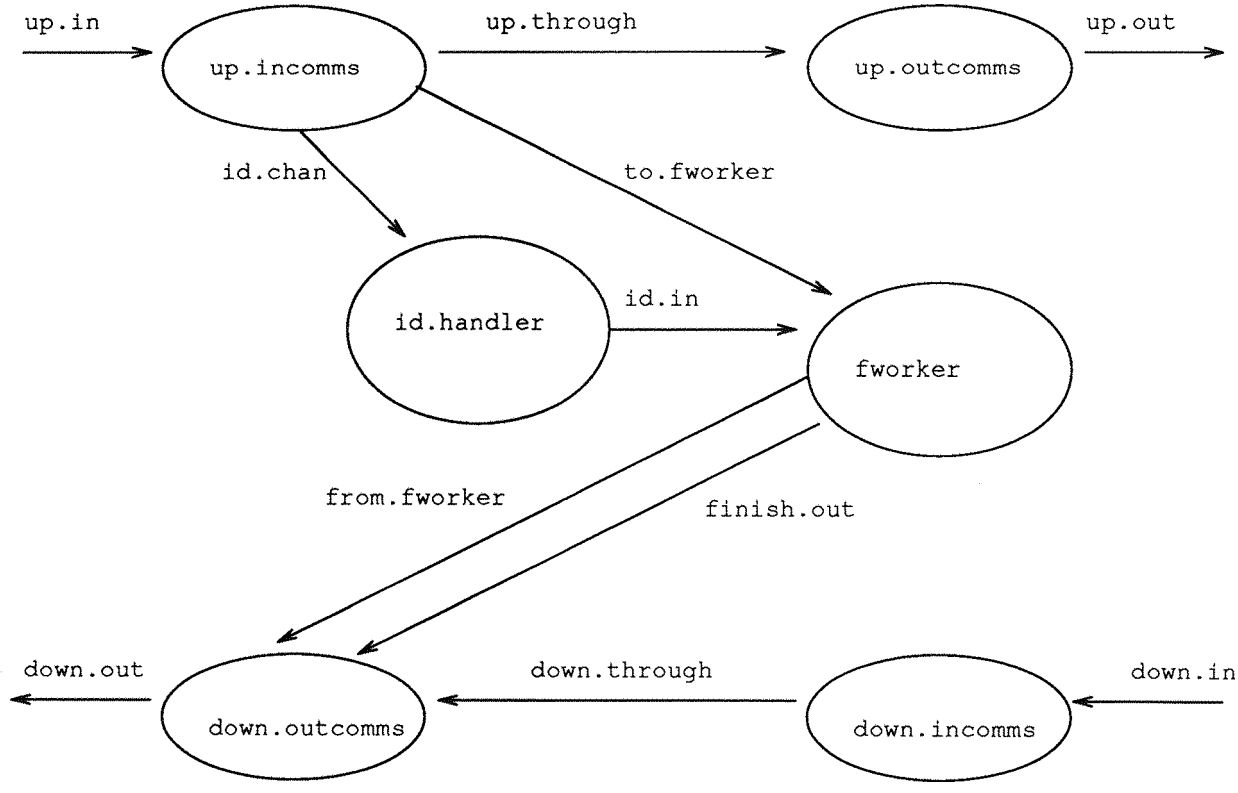

Figure 2.2: Process Diagram for Worker Processor.

the `to.fworker` channel. The `finish.out` channel has a signal placed on it whenever a job is finished.

`down.outcomms`: `Down.outcomms` merges finish signals and filer requests from the present processor and others then passes them back towards the filer.

`up.outcomms` & `down.incomms` : These processes just pass on messages and fulfil the need to have one process per link in/out engine.

2.3.2 Communication Strategy

As the harness was designed to run on a chain of processors the communication strategy is very simple. The usual hardware considerations are taken into account by the provision of a process per linkin/out leading to the most efficient use of the parallelism on the chip itself.

The channels for the main communication path are arranged in a loop through the chain of transputers. Apart from a few special circumstances a message does not travel completely around the loop. Message sending is organised so that whenever a message is sent out it has somewhere to be read. The system, after starting up, fills the worker processors with job identification numbers if they are available. It knows how many to send as the number of worker processors is configured into the system. When working, filer requests move down the loop to the filer and the replies are routed back up the loop to the sending processor. There is no possibility of generating spurious filer messages. The only other messages being generated at this computing stage are end

of job messages in the workers. These messages are replied to by `indemux` with a new `job.id` or a shutdown message to the `fworker` FORTRAN process and its supporting `id.handler` process. The communication processes are NOT shutdown at this stage. When the `job.ids` have been exhausted all the `fworker` and `id.handler` processes have also been shutdown. The final stage of communication is to shut down the communications loop of processes. This is done by sending a flag right around the main channel loop terminating processes as it goes. The master (`termmux`) process can then shutdown and the whole system has cleanly returned control to the host computer.

The protocol used in the harness is the INMOS SP filer protocol with three integer additions.

```
processor.id(INT);message.tag(INT);job.id(INT);
                    SP.len(INT16);SP.mess(BYTE[512])
```

These integers hold the processor identification number, a message type tag and the job identification number. The message type tags are,

- job.tag
- spmess.tag
- finish.tag
- endwork.tag
- terminate.tag

`Job.tag` indicates that a new `job.id` number is contained in the `job.id` field of the message and is sent from `indemux` to the workers.

`Spmess.tag` messages travel in both directions on the loop. They are filer requests from the workers and the subsequent replies from the filer. These are the only messages with have non-empty values in the SP protocol fields of the message.

`Finish.tag` messages go from the workers to `indemux` indicating that a job has been done and requesting a new `job.id`. When there are no more jobs to do `indemux` sends out a `job.id` of -1 to terminate the `fworker` and `id.handler` processes on a processor which has requested a new job.

`Endwork.tag`. The last thing the `fworker` does before it shuts down (`id.handler` has already terminated) is send out an `endwork.tag` message.

`Terminate.tag`. `Indemux` collects `endwork.tag` messages and when it has received one for each worker processor in the system it knows that it is safe to send a `terminate.tag` message and thus close down the whole worker network.

2.3.3 Writing for the Harness

The first thing to establish when attempting to use the harness is if it is appropriate for the problem in hand. This being the case writing for the harness is very easy. The code must fit into the following template.

```
      PROGRAM CORFARM
C
      IMPLICIT NONE
C
C     HARNESS DECLARATIONS
C
      INTEGER PROCID
      INTEGER JOBID
C
C     USERS DECLARATIONS
C
      CALL GETID(PROCID)
      CALL GETID(JOBID)
      WHILE (JOBID.NE.-1) DO 99999
C
C     USERS STANDARD FORTRAN, NO KEYBOARD INPUT
C           & FILE NAMES MUST BE UNIQUE
C
      CALL ENDJOB()
      CALL GETID(JOBID)
99999 CONTINUE
      CALL ENDWORK()
C
      END
```

The user writes standard FORTRAN in the position shown with some limitations as indicated. Porting code into this harness should be simple as the application program is just put into the loop shown above. This template is provided with the harness as well as the library containing the few harness subroutine required. Compilation of the system is fully automated using an appropriate make utility.

2.3.4 Functionality and Implementation Limits

The Coarse Farm Harness gives full access to the filer shared by other processors in the system. Files and screen output are as they would be if each processor had a filer as a single resource. Screen output strings, naturally, appear interleaved as several processors output on a single screen. Keyboard input is available but should not be used. This is because there is no way of telling where the input is going to.

Consider eight processors requesting a number to be input. Good programming practice would be to print a request to the screen. If a number of requests arrive in very quick succession you do not know, when you enter the numbers, where your keyboard input is going. It may not matter in the case of, for example, seeds for random number generator but it usually does matter. For this reason it is suggested that if keyboard

input is needed the input is put into a file beforehand and then taken in by the program using a different unit number for its standard input.

As there is truly only one filer all filenames in a given run of the system must be unique. One way of obtaining this goal is to encode the job and/or the processor number into the file name. A method of coding the job number into the filename is shown in the section on implementing the benchmark tests. (Section 2.4.2)

A limitation given by the host system is the number of files a single program may have open at any one time. For MS-DOS 3.3 (the system used) the limit is twenty files. The server uses a number of these for keyboard, screen et cetera leaving the user with sixteen. A test on SUN Operating System 3.5 allowed fifty files to be opened from a transputer through the server program.

2.4 'Benchmark' Case Study

2.4.1 The Code

It was decided to carry out tests in three areas, screen output, calculation and disk file input and output. The tests were carried out by varying the amount of work contained in a job and by varying the number of worker processors.

The screen output consisted of a number of FORTRAN `PRINT` statements of the form :

```
      PRINT *,'Hello, world from processor '
     *           ,PROCID,JOBID
```

Where `PROCID` is the processor identification number picked up from the system and `JOBID` is the job identification number issued by the system. Tests were run with a job consisting of a single output, and of a loop of ten outputs. It was expected that for screen output, which is contention for a single resource, the time taken for a given system to execute would be related to the number of outputs and not to how they were split into jobs. There may be decreases in performance due to the system saturating the file server with requests.

Tests on the calculation capability of the system were done using the statement:

```
X = X * Y
```

The statement was put in a loop with X set to 0.0 and Y set to 1.0. This is where the harness is expected to perform well. All the calculations in a job can be overlapped with calculations from other jobs.

The disk file input and output tests have a number of stages. The first one of the stages is to construct a filename for the job from the job identification number. This is a technique which should be used generally when using the harness. Data is then loaded into an array of an appropriate size. The file is then opened, the data written and the file closed. Now the file is reopened and the data read back into a new array. The file is again closed and a check on the data is carried out. The result of this check is then reported to the screen. The various calculations should happen in parallel and the access to the file handled by the server should follow the same scenario as accessing the screen, as it is accessing a shared resource.

2.4.2 The Implementation

The benchmark was implemented by the code found in appendix A inserted into the harness as described before. The various tests were carried out by commenting out various parts of the code.

All the code was executed on a TRAM system consisting of six modules on an INMOS motherboard. Each module holds a T800 transputer running at seventeen megahertz. Memory totals were 2 Mbytes of DRAM, 32 Kbytes of SRAM and four kilobytes of the transputers internal memory. The processors were connected together in a linear chain with the link speeds set at twenty megabits per second.

2.4.3 The Results

The following tests were coded in FORTRAN and the results are shown here in graphical form with an explanation for each graph. Tables of all the data from the results are in appendix A.

Screen output tests.

- Test 1. Each job consists of a single output of the hello world string. Batches of up to two hundred jobs were run. (Figure 2.3)
- Test 2. Each job consists of ten outputs of the same string used in test 1. Batches of up to thirty jobs were run. (Figure 2.4)

Calculation tests. The basic element of this test was the multiplication of two real numbers.

- Test 3. Each job was a loop of one hundred thousand iterations. The maximum number of jobs run was one hundred. (Figure 2.5)
- Test 4. One million iterations per job with a maximum number of twenty one jobs in the system. (Figure 2.6)

Disk file input/output tests. Each of these tests consists of a number of jobs which each output a block of data to a file using a FORTRAN unformatted write statement then read the same data back again. The data is then checked and a screen write statement giving the result of the test.

- Test 5. One kilobyte blocks with a maximum of fifty jobs in the system. (Figure 2.7)
- Test 6. Ten kilobyte blocks with a maximum of eleven jobs in the system. (Figure 2.8)
- Test 7. One hundred kilobyte blocks with a maximum of eleven jobs in the system. (Figure 2.9)

Processor power tests. The first seven tests vary the amount of work in the system. Having done this the number of worker processors was varied to see the effect of increasing processor power in the system.

- Test 8. Test 1 for screen output with two hundred jobs. (Figure 2.10)

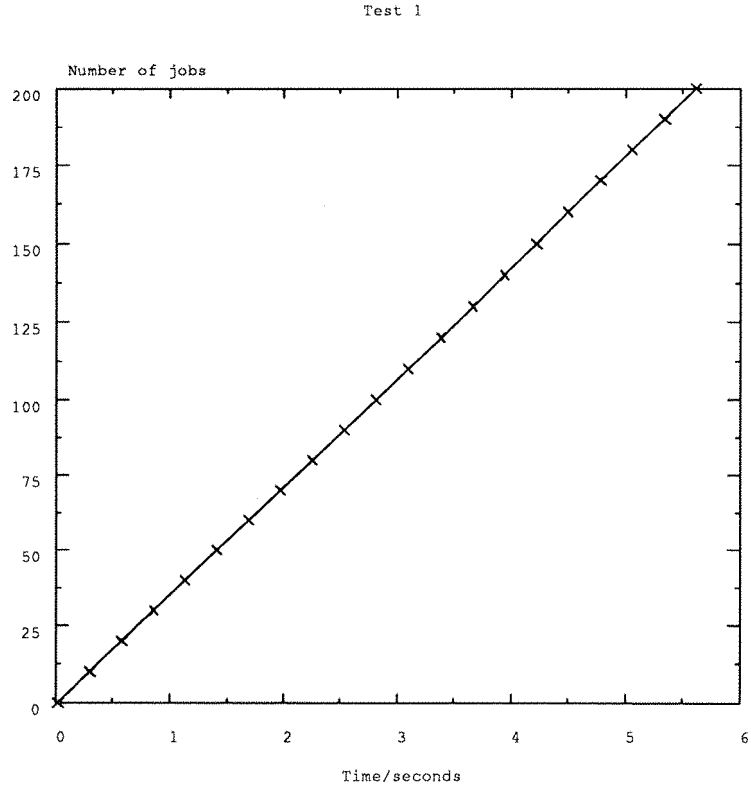

Figure 2.3: Benchmark Test 1: Single screen output per job.

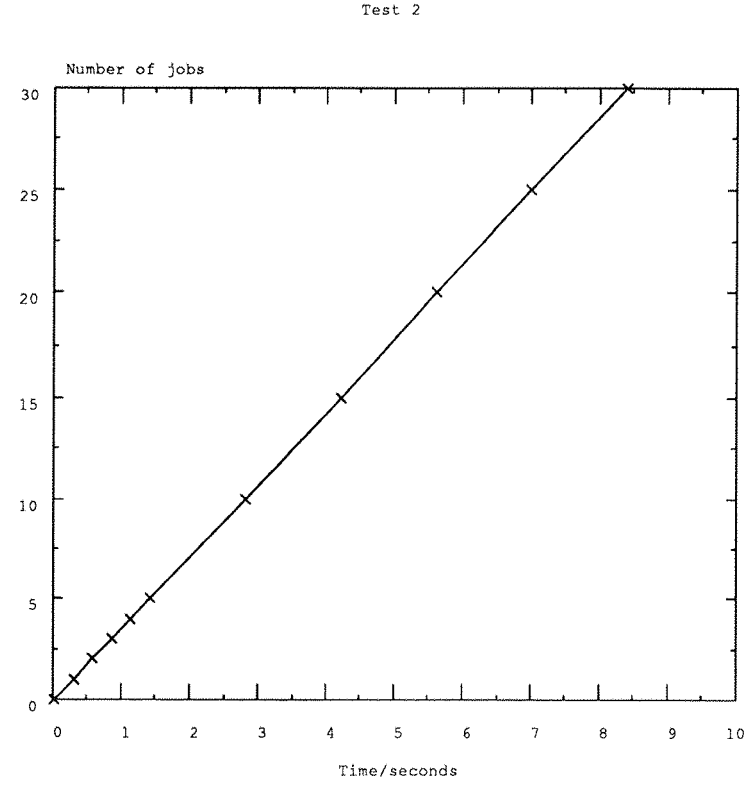

Figure 2.4: Benchmark Test 2: Ten screen outputs per job.

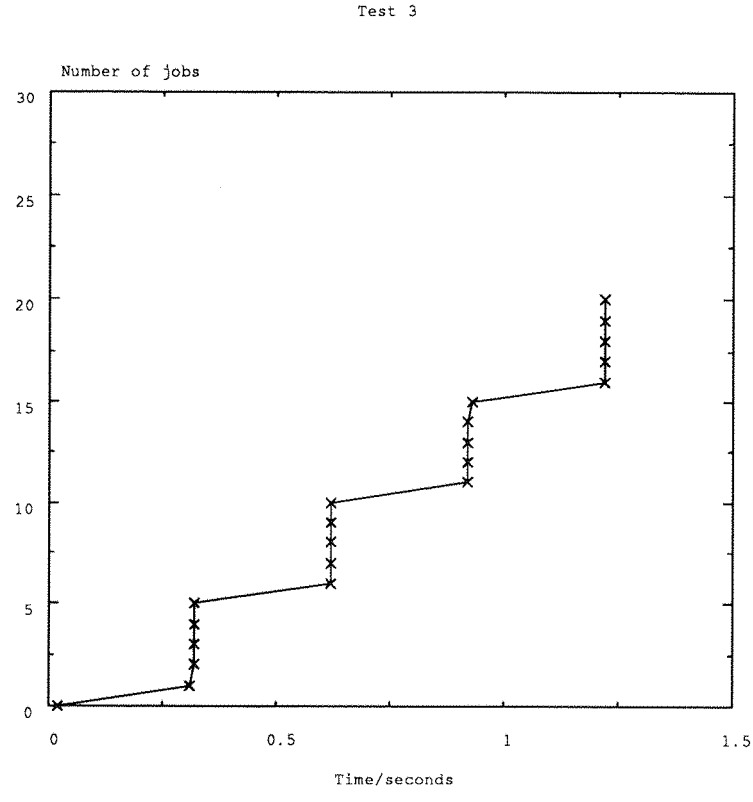

Figure 2.5: Benchmark Test 3: 100000 floating point multiplies per job.

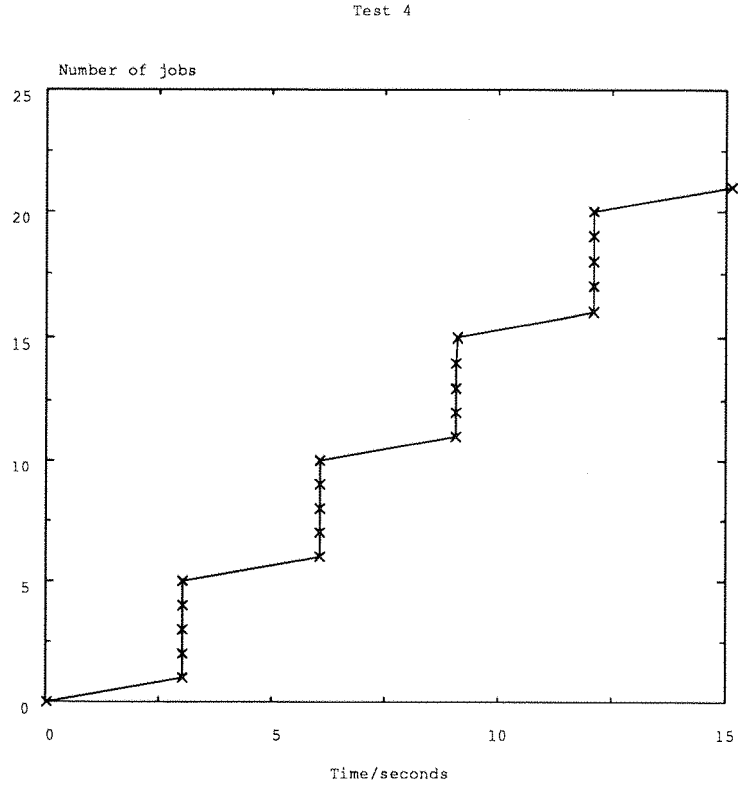

Figure 2.6: Benchmark Test 4: 1000000 floating point multiplies per job.

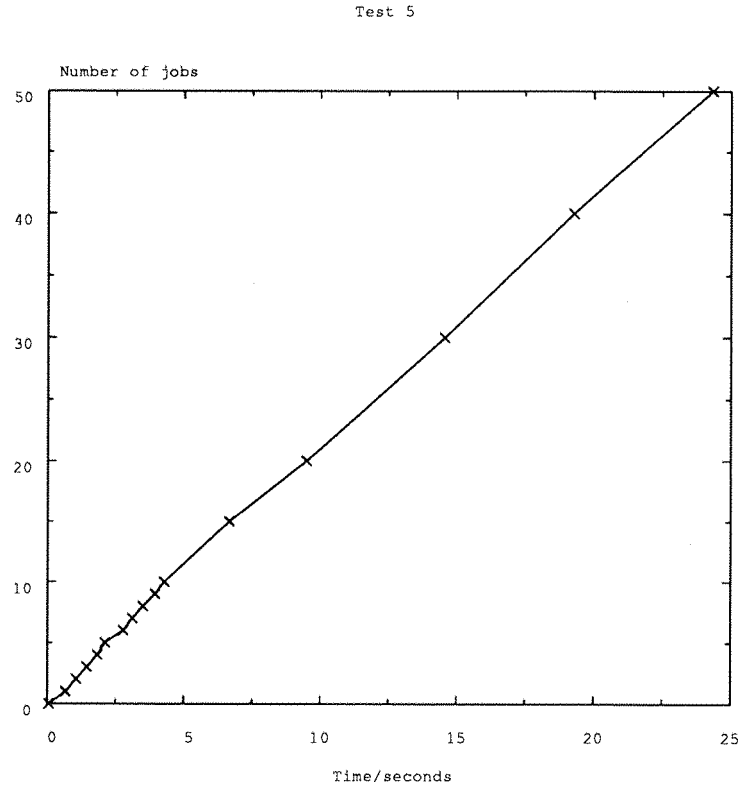

Figure 2.7: Benchmark Test 5: 1 Kbyte of file i/o per job.

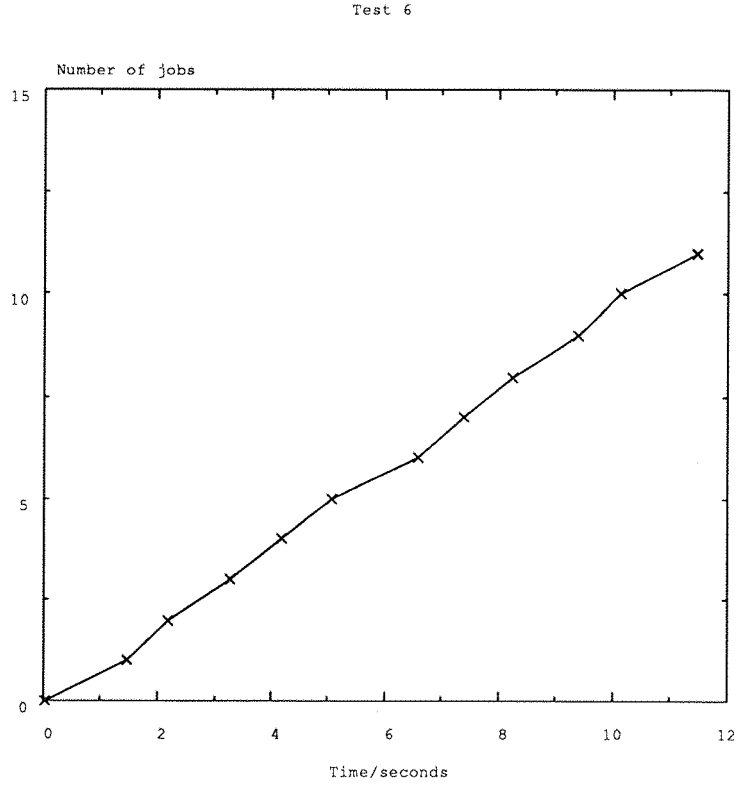

Figure 2.8: Benchmark Test 6: 10 Kbytes of file i/o per job.

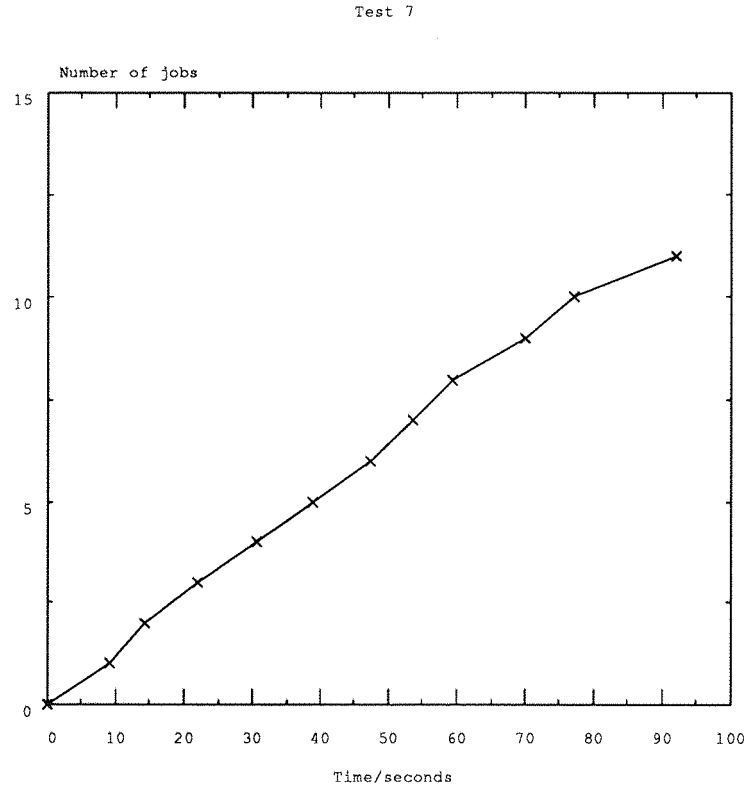

Figure 2.9: Benchmark Test 7: 100 Kbytes of file i/o per job.

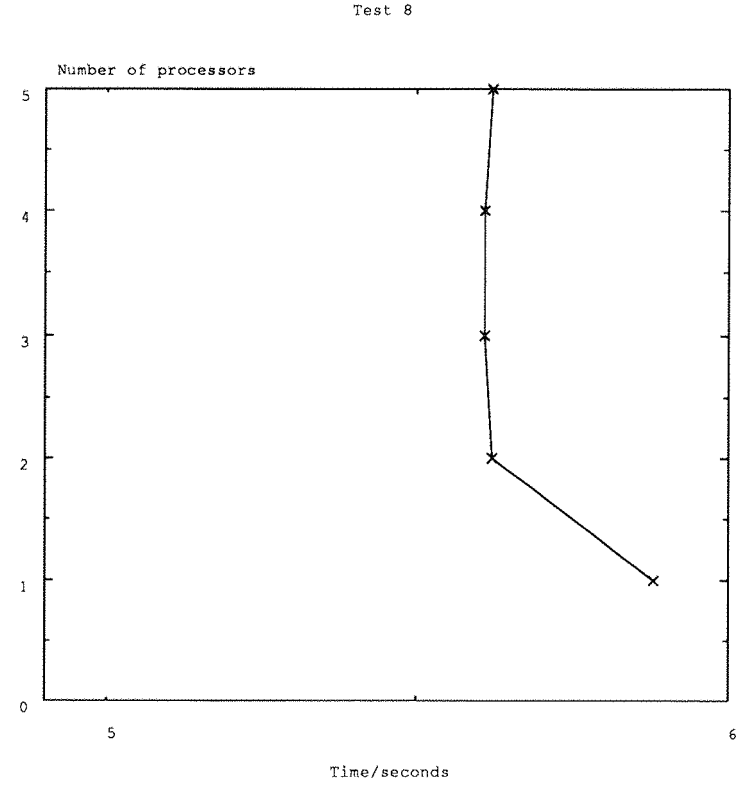

Figure 2.10: Varying the Number of Processors: Test 1 with 200 jobs.

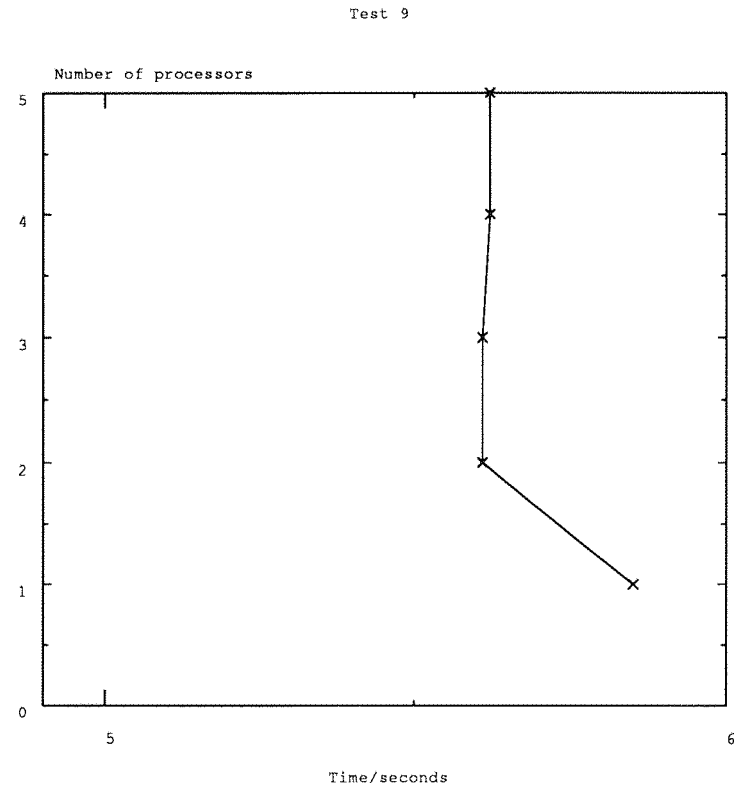

Figure 2.11: Varying the Number of Processors: Test 2 with 20 jobs.

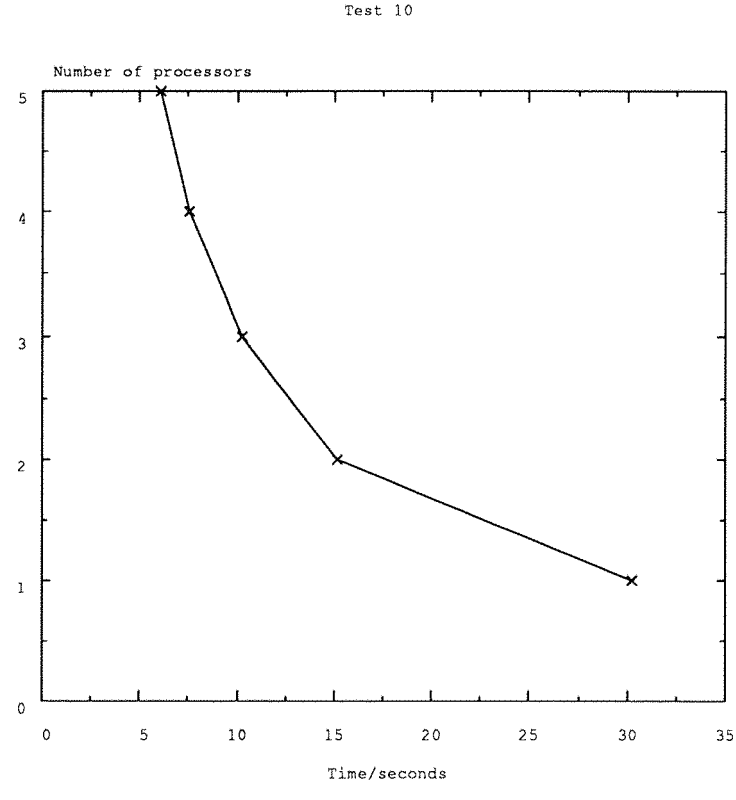

Figure 2.12: Varying the Number of Processors: Test 3 with 100 jobs.

- Test 9. Test 2 for screen output with twenty jobs. (Figure 2.11)
- Test 10. Test 3 for calculation with one hundred jobs. (Figure 2.12)
- Test 11. Test 4 for calculation with fifteen jobs. (Figure 2.13)
- Test 12. Test 5 for file input/output with fifteen jobs. (Figure 2.14)
- Test 13. Test 6 for file input/output with ten jobs. (Figure 2.15)

2.4.4 The Conclusions

It can be seen from the results of tests 1, 2, 8 and 9 that the system is driving the filer as fast as it will go for screen output. From the results of test 1 it can be seen that as the number of communications to the screen increases, the time taken for the system to execute increases in a very close to linear fashion. Correlating data from test 1 and test 2, for example the time taken to do two hundred jobs in test 1 and the time for twenty jobs in test 2, it is evident that the different loading of work content in a job has no effect on the time taken. From tests 8 and 9 it is noted that for increasing processor power the speed is constant. The only departure from this constant speed is for a system with only one worker. This, I feel, is because with more than one processor the handling of job.ids is hidden by overlap but with one worker a small overhead shows up. These results are as expected as the screen is a single resource. There is contention between the processes for the use of the resource and it is obviously running as fast as it can go. It is the total number of messages to the screen which governs the time taken for a system to execute, not how the work is split into jobs or the available processor power.

In the tests on calculation (3, 4, 10 and 11) it can be seen that this is where the harness provides the recognised farming results. Tests 3 and 4 show, convincingly, that all five worker processors operate in parallel. The time for the system to execute jumps whenever the number of jobs in the system is a multiple of five. In tests 10 and 11 the time for a given system to execute is seen to go down as processing power increases. The graphs show execution time decreasing with increasing processing power.

From the results of tests 5, 6, 7, 12 and 13 it is seen that there is some overlap in disk file input/output. This is again a shared resource but it seems that the tests do not drive this feature of the filer as fast as it will go. With tests 12 and 13, as with the equivalent screen output tests, there is an overhead in issuing job.ids with one processor which disappears with more than one. Here, however, the time taken for a constant amount of work decreases slightly as processor power is increased. This implies that there is some extra overlap occurring as the number of processors increases. It can also be seen that the efficiency of disk file input/output increases when the amount of data in a block is increased.

So the following results have been shown.

- There is no speed up in the area of screen output as there is no overlap in accessing this shared resource. (Keyboard input is not allowed)
- For the tests done the usual farming results are obtained for calculation, great increases speed can be achieved here as there is total overlap between the processors.
- There are some gains to be made in disk file input/output. Dealing with the largest blocks possible helps here.

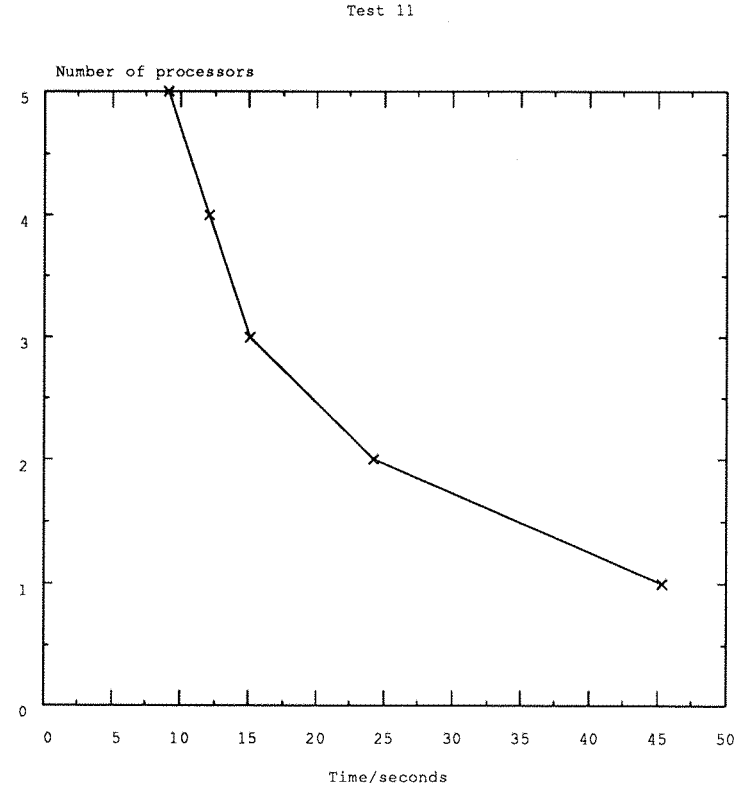

Figure 2.13: Varying the Number of Processors: Test 4 with 15 jobs.

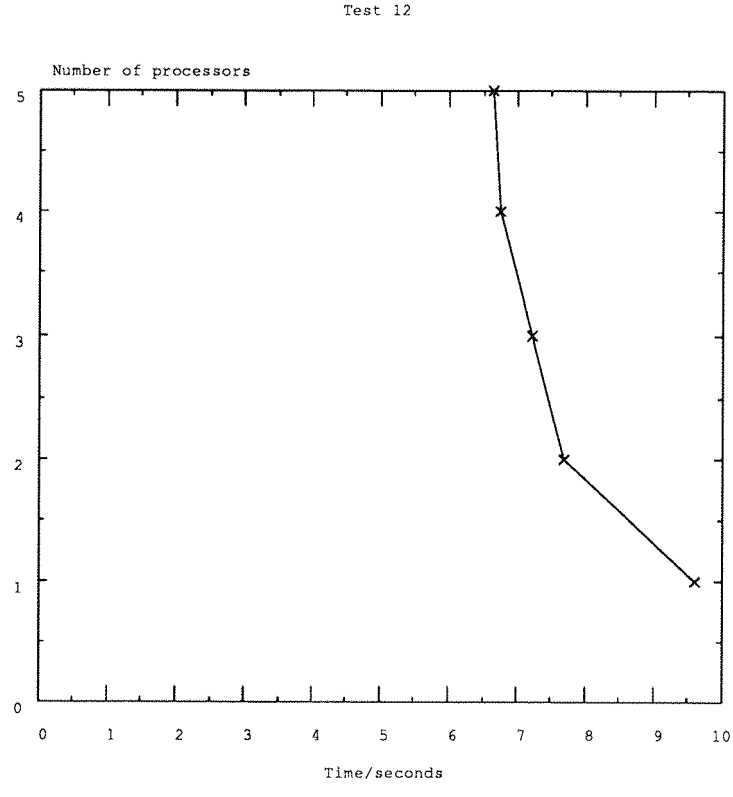

Figure 2.14: Varying the Number of Processors: Test 5 with 15 jobs.

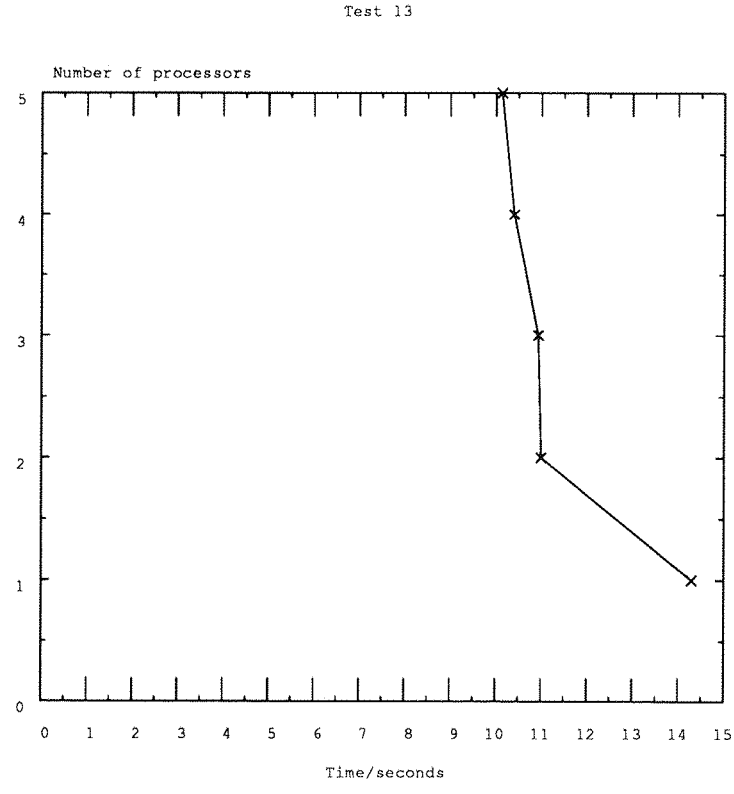

Figure 2.15: Varying the Number of Processors: Test 6 with 10 jobs.

2.5 Diffusion Limited Aggregation (DLA) Case Study

2.5.1 The Problem

The application chosen to demonstrate the coarse farm harness was Diffusion Limited Aggregation (DLA) from the subject of Chemistry. The specific problem from the realm of DLA used for the test was as follows.

We have a rectangular lattice of sites with a surface at one side. A particle is introduced at a random point a few lattice steps above the surface and random walked until it is adjacent to the surface or has escaped the lattice as defined. When the particle is adjacent to the surface it is said to have 'stuck'. Another particle is introduced and so forth. Particles are said to stick if they are adjacent to the surface or to another stuck particle. The stuck particles form a cluster. New particles are introduced to the system at random points a few steps above the then present maximum height of the cluster above the surface. If a particle escapes the confines of the lattice (ie. walks out of the lattice through the opposite side to the surface) it is restarted. The two sides of the lattice adjacent to the surface form a cyclic boundary.

It can be seen that introducing more than one particle at a time changes the whole character of the problem and that the problem as defined is basically sequential in nature. Variations of DLA include having a single occupied site at the centre of the lattice (which has no surfaces) or to have multiple particles introduced simultaneously. If multiple particles are introduced the problem is very much more complicated as there is then the prospect of particles coagulating to form sub-clusters before they encounter the main cluster.

The purpose in generating these clusters is to model systems such as soot deposition. Large numbers of clusters are generated and subsequently analysed to calculate such things as fractal dimension. The analysis of clusters generated in this way is taken up in other works [16] it is the generating of the clusters which is addressed here.

2.5.2 The Code

The DLA code was written in FORTRAN 77, compiled with the 3L parallel FORTRAN compiler version 2.0 and inserted into the occam harness using the INMOS D705B toolset linker and interface code. Special features of the code are :

- Use of 3L library subroutines to access the transputers clock for timing purposes.
- Use of 3L library subroutines to access integers in a bitwise fashion. The main grid was stored as an integer array and accessed in this way to allow as compact lattice as possible.

The structure of the code was as follows :

```
initialise
setup file
LOOP for the number of particles required
                or until the lattice is full
        write to file if the program buffers are full
        LOOP while a particle is not stuck
                random walk the particle
```

```
                    see if stuck
                    if it is stuck update the lattice
                            and program buffers
            END LOOP
    END LOOP
    write any data in program buffers
    report the programs execution time
```

A full listing of the program may be found in appendix B.

2.5.3 The Implementation

The DLA code was initially implemented in a single transputer 'standalone' form. This code was compiled with the 3L libraries and run to obtain some initial results. The standalone program was then adapted and placed in the harness as indicated earlier. Full code listing for the adapted program in the harness is in appendix B.

2.5.4 The Results

From the standalone program running on a single T800-17 the results in the table were obtained. (Table 2.1)

No. of particles	Time
10000	1min 39secs
100000	41mins 51secs
1000000	1152mins 30secs

Table 2.1: Standalone DLA results.

So a T800-17 takes 19 hours 12 minutes to generate a cluster of one million particles. Moving from this result with naive calculations the estimated time to generate a one million particle cluster on faster T800 processors is shown in table 2.2.

Processor	Time
T800-20	16hrs 30mins
T800-25	13hrs 5mins
T800-30	10hrs 55mins

Table 2.2: Faster processors on one million particle cluster.

Taking the ten thousand particle problem the following results were obtained (Table 2.3 and table 2.4). Each job reports its individual time and the system reports the overall time for execution. The hardware used was the same used in the benchmark section of this chapter.

No. of jobs	job time(secs)	System execution time(secs)
1	160	160
2	131	
	164	164
3	133	
	144	
	150	150
4	132	
	137	
	154	
	186	186
5	132	
	137	
	160	
	161	
	169	169
6	134	
	139	
	144	
	147	
	153	
	122	258
7	133	
	138	
	140	
	163	
	175	
	123	
	154	294

Table 2.3: Coarse Farm Harness DLA. (Continued in table 2.4)

2.5.5 The Conclusions

It can be seen from the results that it pays very well to run multiple jobs on transputers. For the case of ten jobs the total time for executing the jobs in sequence is 1403 seconds compared to the system execution time of 297 seconds. Using the method made available by the Coarse Farm Harness rapid production of data sets is very achievable.

A 'fast' algorithm in a letter to the Journal of Physics [17] produces a 100000 particle cluster in 10 CPU minuites on an IBM 3081. From the results in table 2.1 it can be seen that with only 4 T800-17 processors the average speed of cluster production of the application in the coarse farm harness is matching the IBM 3081 performance. Adding any other transputers to the system will out perform the IBM on overall cluster production time.

No. of jobs	job time(secs)	System execution time(secs)
8	133	
	139	
	140	
	154	
	159	
	124	
	148	
	157	297
9	135	
	137	
	140	
	140	
	159	
	124	
	126	
	145	
	156	297
10	120	
	134	
	140	
	146	
	151	
	125	
	165	
	145	
	141	
	156	297

Table 2.4: Coarse Farm Harness DLA continued.

2.6 Summary of the Coarse Farm Harness.

A harness has been produced for the coarse farm paradigm. This was an occam harness to go around a piece of FORTRAN applications code. The coarse farm paradigm is where a number of whole problems or jobs need to be executed to produce a large number of data sets. These data sets are then analysed at a later time. The program to produce one data set is replicated over a network of transputers. Each transputer must be given access to the file server transparently using the usual input/output subroutines. This system was successfully implemented on a chain of transputers using INMOS D705B occam toolset system and 3L parallel FORTRAN version 2.0. Extensive 'benchmark' tests were run to evaluate the performance and functionallity of the harness. These being successful an application from the field of Chemistry, Diffusion Limited Aggregation, was implemented in the coarse farm harness. Rapid production of data sets was achieved, the system producing ten data sets in 297 seconds compared to the sequencial production time of 1403 seconds.

3 Geometric Harness

3.1 The Geometric Paradigm

The geometric paradigm has also been called domain decomposition which is probably a more descriptive name. The problem must consist of a large space over which calculations are being performed. The usual type of problem is some kind of simulation of particles or calculation of a field locally at a number of grid points.

To make such a problem parallel we split the area or volume of the calculation between the available processors. Each processor may now work on its own subspace of the problem using a copy of the program for the whole space with just the size of the data changed. To make the overall calculation correct there has to be communication at the boundaries of the processors. The interaction used in the simulation will have a 'range'. Ideally this range will be as short as possible. Every processor must receive data from all of its neighbours for the calculation to be correct. Thus the range of the interaction determines the scale of the decomposition. An interaction which has infinite range cannot be parallelised in this manner. Nearest neighbour interactions are the best case for this paradigm. An illustration of the edge swapping needed to give correctness is shown in figure 3.1.

Because of this edge swap, efficiencies gained by the geometric paradigm are not as great as for the farm paradigm. This is because processors have to communicate locally as well as communicating with a central controlling resource. The central resource is usually a graphics screen or a storage device such as a disk with its own associated processor. This processor usually handles any synchronisation necessary to keep the worker processors on the same time step in the calculation. This synchronisation can sometimes be achieved by the interprocessor communications locally, but in most cases a form of 'loose lock step' is imposed by the central processor.

The system is running a single job so resources such as keyboard and disk are not required to be generally accessible to all worker processors, unlike the case with the coarse farm harness. It is however useful to obtain error messages from individual processors on the screen as they occur.

3.2 Perceived User Requirements

As was stated for the coarse farm harness the first thing the user requires is a clear indication of when the harness is an appropriate tool to use. The geometric harness is used to give a parallel solution for a simulation over a large data space. Interactions between entities in the simulation should be at as close a range as possible (nearest neighbour is ideal) as this governs the amount of data to be edge swapped at each iteration of the simulation. Range of the interaction also determines whether this paradigm is appropriate as explained before.

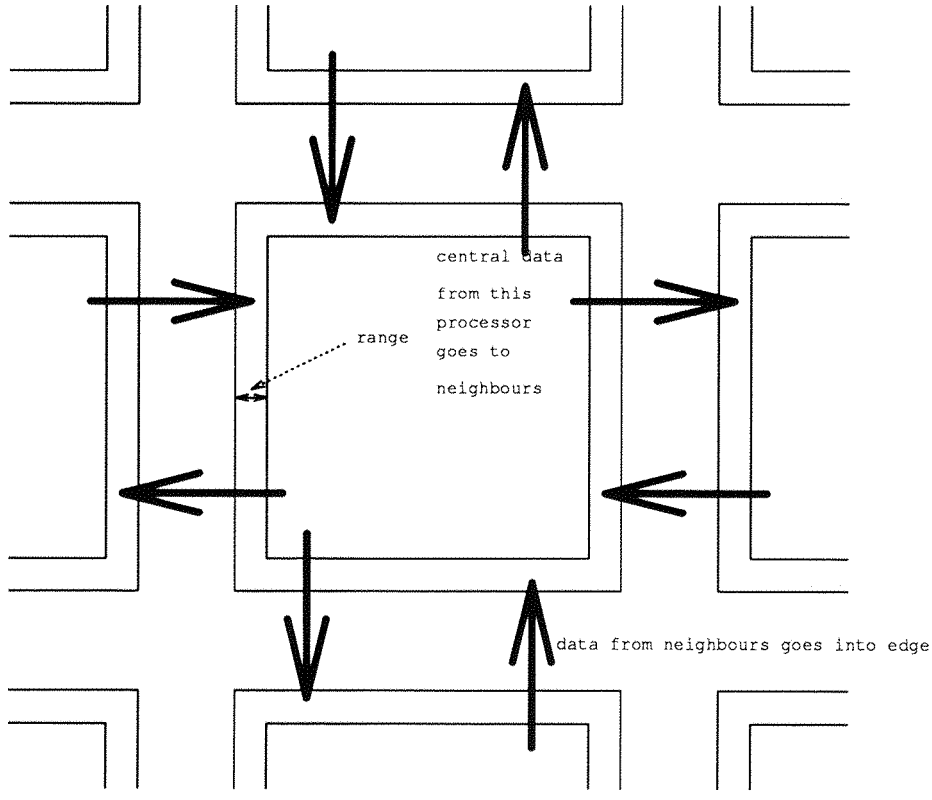

Figure 3.1: Edge Swap Illustration.

Specification of communications, an outline of processes and the necessary templates for writing for the harness are included in the sections following.

3.3 The Geometric Harness

3.3.1 Outline of Processes

The geometric harness is to run on a torus grid with a control processor in one of the loops. The set up of the network is illustrated in figure 3.2.

The usual transputer considerations are taken into account and the master process is loaded onto processor C, the worker process onto processors W. Apart from initial and final subroutine calls the communications in the harness are in the form of point to point message passing between specified processors. All processors have an x,y set of co-ordinates. Messages can be routed in via link 0 or link 1 then accepted or routed through the processor to be output on link 2 or link 3. This very simple routing algorithm allows the minimum number of processes on a processor and very fast routing decisions to be taken.

The master process (`master`) is shown in figure 3.3.

Some initial interaction with the user is undertaken then a `PAR` statement starts the illustrated processes. The processes operate as follows :

`minit` : This short process sends some initial data values to the FORTRAN and then terminates.

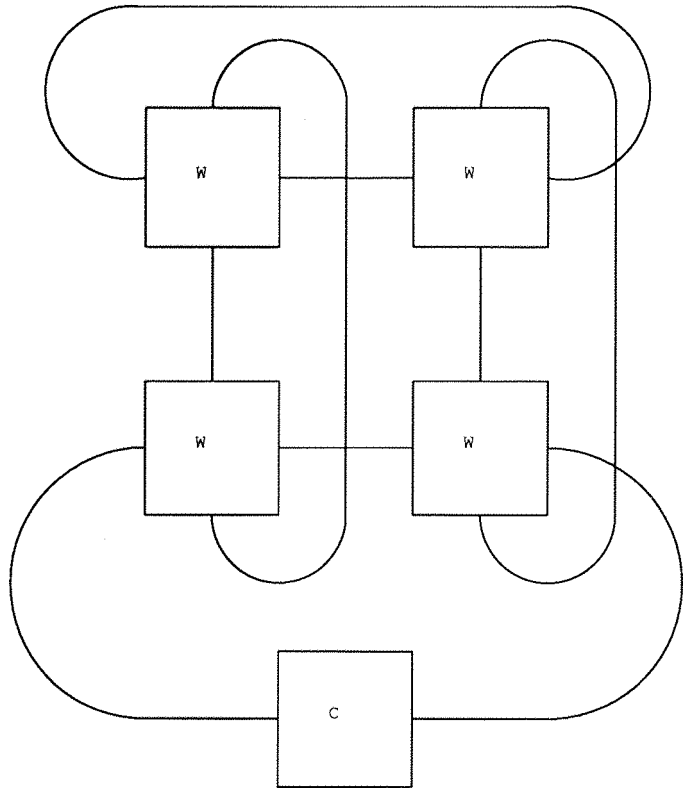

Figure 3.2: Processor Network for Geometric Harness.

`fmaster` : Contains the user written FORTRAN code for the master processor. After gaining the initial data sent by `minit`, via a subroutine call, point to point communication can be undertaken with the worker processors using a procedural interface. At the end of the users code a subroutine call sends a message to the workers instructing them to terminate. A signal is also sent to the `mstopper` process which shuts down all the communications processes. `Fmaster` itself then terminates. Full FORTRAN input and output is available from this process.

`mstopper` : On receiving a signal from `fmaster` this process sends termination signals to `min1` and `mout3` then it terminates itself.

`min1` : This process takes in communications and routes them either to the channel `through` to `mout3` or, if they are for the master processor, into `fmaster`. Termination is by a signal from `mstopper`.

`mout3` : Communications are gathered from `min1` and `fmaster` then output to the worker network which handles onward routing of the messages. A signal from `mstopper` terminates this process.

The worker process (`worker`) is shown in figure 3.4.

The illustrated processes are started up straight away on the worker process by a `PAR` construct and have the following functions :

`winit` : As for the master processor this outputs some initial values to the FORTRAN then terminates.

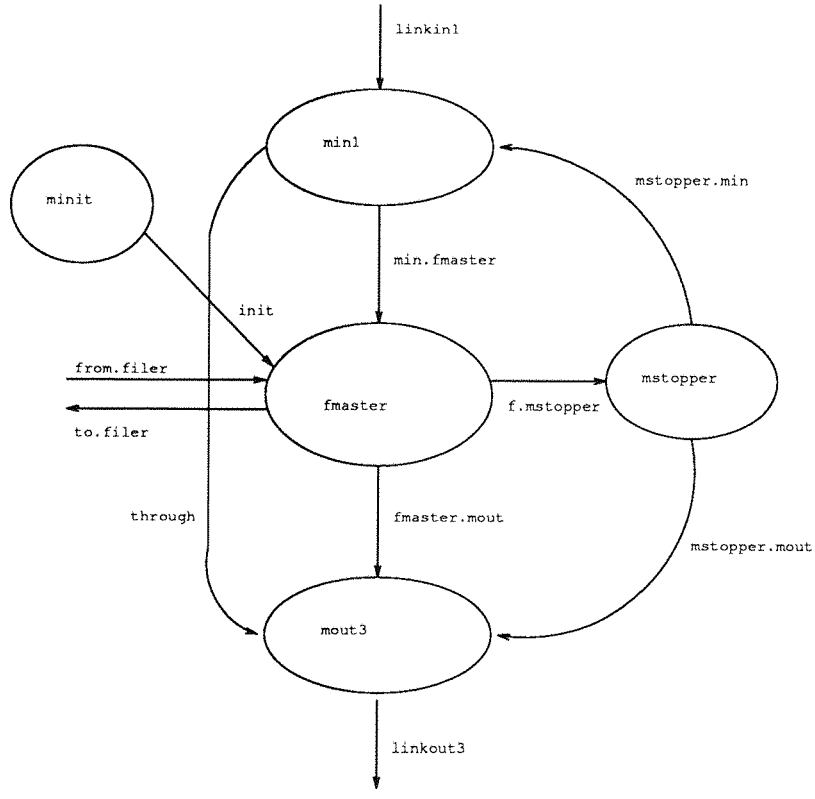

Figure 3.3: Process Diagram for Master Processor.

`win0/win1` : Communications only come into the processor via links 0 and 1. These processes handle incoming messages. The messages are either sent to `winmux` if destined for this processor or are routed via the through channels to the appropriate link output process. Termination of these processes is induced by a signal from `wstopper`.

`winmux` : Messages for this processor are routed here from the `win` processes. All the `winmux` process has to do is merge the streams and sent the messages on to the `fworker` process. A signal from `wstopper` causes termination.

`fworker` : This process contains the users FORTRAN code for the worker processors. Initial data regarding the network size and this processors position in the network is obtained from `winit`. Communications to the master FORTRAN and the FORTRAN on the other worker processors is via library subroutines. At the end of the code a signal is sent to initiate `wstopper` after a suitable interaction with the master processors FORTRAN.

`woutdemux` : On receiving a message from the `fworker` process a routing decision is made and the message is output to a suitable link output process. Termination is by `wstopper` signal.

`wout2/wout3` : These processes just receive messages from the demultiplexor process and the `through` channels which are output onto the links. These actions continue until `wstopper` signals termination.

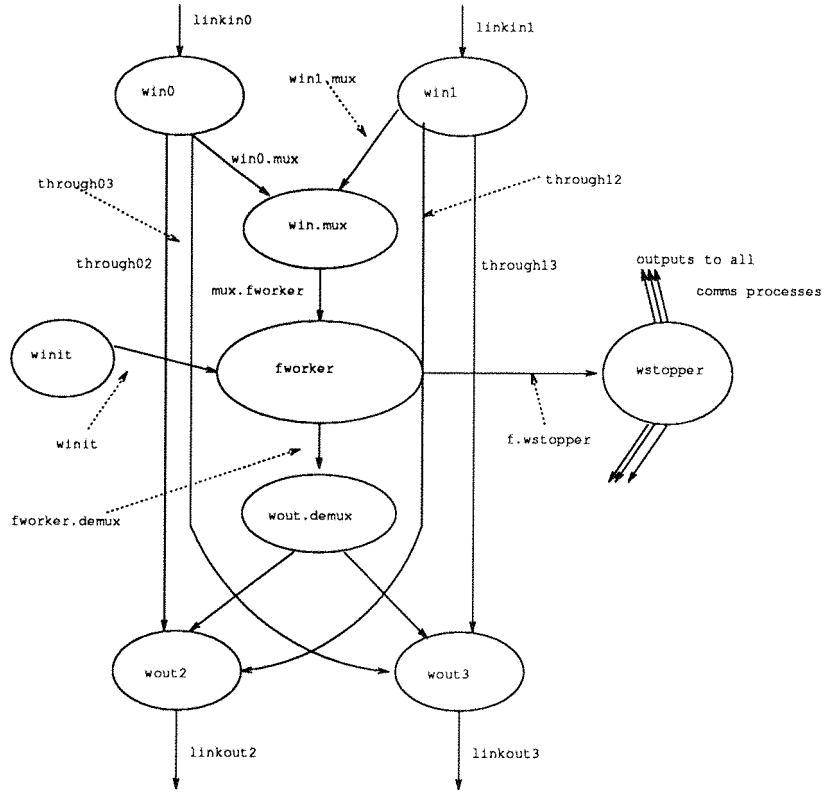

Figure 3.4: Process Diagram for Worker Processor.

`wstopper` : The FORTRAN `fworker` on receiving a termination signal from the master processor sends a signal to `wstopper`. The `wstopper` process then sends signals down all its channels to the communication processes on the processor causing them to terminate. `Wstopper` then terminates and the worker processor has cleanly shut down.

3.3.2 Communication Strategy

The processors for the geometric harness are connected in a toroidal grid. The communications strategy for the system is for messages to travel conceptually 'up and to the right' to get to their destination. This is not the shortest route a message need travel necessarily but it is flow in one direction only and is thus easy to control. Messages move up until they have the correct y co-ordinate for their destination and then right to reach their final target. This strategy means that the master processor is handled implicitly.

After the system has initialised, point to point communication is allowed between any two processors. These communications are asynchronous in nature. At the receiving end no guarantee is given of the order of arrival of messages so some care may be needed when writing for the harness.

The messages which travel round the harness have the following protocol :

```
dx(integer);dy(integer);sx(integer);sy(integer);
   tag(integer);
        len(integer)::buffer(integer array size 128)
```

dx and dy are the destination processor co-ordinates. sx and sy are the co-ordinates of the source processor. Tag may be used by the programmer to mark a message. All values of tag may be used except -1. This is reserved for harness shut down use. A message of length len integers is contained in the integer array buffer.

The point to point communications are handled by two subroutines :

```
SUBROUTINE SENDMESS(DX,DY,TAG,LEN,BUFFER)
```

This subroutine sends a message contained in BUFFER of length LEN integers to the processor with the co-ordinates DX and DY. TAG has a user specified value. If a message other than one of type integer is required then FORTRAN EQUIVALENCE statements should be used to load the message up into BUFFER before transmission.

A message is read by :

```
SUBROUTINE GETMESS(SX,SY,TAG,LEN,BUFFER)
```

It is a message contained in the integer array BUFFER, of length LEN, with a tag value TAG from the processor with co-ordinates SX and SY.

When all the communications have been done and the system is ready to shut down a subroutine is called at the end of the users FORTRAN code. On the master processor subroutine SHUTDOWN is called with the dimensions of the grid as parameters. A point to point communication is sent to each worker processor with a tag value of -1. This causes the wstopper processes to be activated and the workers to terminate. The master processor then shuts down and the whole system returns control to the host computer cleanly. On the worker a subroutine call ENDWORK with no parameters handle its end of the shut down interaction. An error occurs if ENDWORK does not receive a message with the tag value -1 as expected.

3.3.3 Writing for the Geometric Harness

If the harness is deemed appropriate for the job in hand the user must produce two pieces of FORTRAN code.

The master code must fit the following template :

```
      PROGRAM FMASTER
C
      IMPLICIT NONE
C
C     HARNESS DECLARATIONS
C
      INTEGER X,Y,XDIM,YXIM
  INTEGER DX,DY,SX,SY,TAG,LEN,BUFFER(128)
C
C     USERS DECLARATIONS
C
  CALL INIT(X,Y,XDIM,YDIM)
C
C     USERS FORTRAN CODE WITH FULL I/O ACCESS
C     MESSAGES SENT AND RECEIVED BY SUBROUTINES
```

```
C       SENDMESS AND GETMESS
C
        CALL SHUTDOWN(XDIM,YDIM)
C
        END
```

INIT returns the processors X,Y co-ordinates (0,1 in the case of the master) and the maximum x and y dimensions of the grid. The second line of integer declarations gives all the variables necessary for the message passing subroutines. The master is connected to the file server as the root processor in the network and so has full access to the file system facilities. SHUTDOWN signals all worker processors to terminate by sending out a message with a tag value of -1. This is the only reserved tag value any other values may be utilized in normal use of the harness. Between INIT and SHUTDOWN the point to point subroutines may be used freely to implement communications with other processors.

The worker FORTRAN code must fit the following template :

```
        PROGRAM FWORKER
C
        IMPLICIT NONE
C
C       HARNESS DECLARATIONS
C
        INTEGER X,Y,XDIM,YDIM
        INTEGER DX,DY,SX,SY,TAG,LEN,BUFFER(128)
C
C       USERS DECLARATIONS
C
        CALL INIT(X,Y,XDIM,YDIM)
C
C       USERS FORTRAN CODE WITH NO FILE I/O
C       MESSAGES SENT AND RECEIVED BY SUBROUTINES
C       SENDMESS AND GETMESS
C
        CALL ENDWORK()
C
        END
```

INIT works in the same way as when it was used in FMASTER. No file access is provided for in the harness from the worker processors. The subroutine ENDWORK waits to receive a message from the master processor with a tag value of -1 then it shuts down the processor.

These templates appear in the examples with the harness and all the compilation is fully automated with makefiles.

3.3.4 Functionality and Implementation Limits

As it stands the harness will only run on a grid of the type shown in figure 3.2. The master processor must be placed in the loop of the lowest y chain in the position 0,1.

The grid can have any x and y dimensions and the configuration file is set up so that changing two constants followed by a reconfiguration is all that is required to adjust grid size. All worker processors have no access to the filer facilities. A basic point to point communications facility is provided with a subroutine interface. The message passed is a 128 integer array into which the data needing to be transferred should be packed and then unpacked at the receiving end. FORTRAN `EQUIVALENCE` statements should be used to send messages whose type is not integer. Zero length messages are not allowed, the smallest message that can be sent is one integer (four bytes). The harness is not guaranteed to be deadlock free, however if a sensible strategy is used (ie. do not send all your messages and then try to receive them) the harness behaves well. Full FORTRAN access to the file system is achieved through the master process.

3.4 'Benchmark' Case Study

3.4.1 The Code

The benchmark set up for the geometric harness was more to test functionality than any performance points. The transputers were set up in the grid illustrated in figure 3.2.

Firstly three communications tests were run.

- All to one test
- One to all test
- All to all test

All these tests ran correctly so I feel safe in saying that the communications strategy of the geometric harness is robust in heavy use.

In the first test all the workers communicate with the master for a number of loops. This means that the number of messages in the system is the number of workers multiplied by the number of loops.

In the second test the first test is reversed. The master sends messages to the workers continuously. Again the number of messages in the system is the number of worker processors multiplied by the number of loops.

In the final test a message is transferred from a processor to all of the processors in the system, including itself. So for each loop there are the total number of processors (including the master) squared messages in the system.

Messages are sent and received with the point to point `SENDMESS` and `GETMESS` subroutines described earlier.

A final test was used to simulate a working geometric system. In this test an edge swap on the workers was undertaken then a work loop of one million floating point operations was executed on each worker. At the end of each work loop a signal was sent to the master processor. We have the following code structure on the worker processors.

```
init
LOOP for number of loops
        edge swap
        calculation loop
```

```
        message to master
END LOOP
message to master
shutdown
```

This simulates a real calculation with a geometric problem being solved on the workers and data collected every iteration onto the master processor.

The master processor has the following code structure :

```
init
LOOP for number of loops
        collect data for each iteration of workers
END LOOP
collect final message from workers
shutdown
```

The success of the tests will show that for heavy communication loads and for a typical geometric problem set up the harness operates correctly.

3.4.2 The Implementation

The benchmark was implemented on a grid of processors as illustrated in figure 3.2. The transputers used were T800-20 parts each with four megabytes of memory (six cycle DRAM). Interface to an IBM PC XT clone (OPUS PC II) was provided by an interface card which like the transputer cards was produced inhouse at Southampton [21],[22].

The code written for the master and worker processors is contained in appendix C. In the same way as the coarse farm harness parts of the code were commented to give the correct test. Appendix C also gives a listing of the library `GEOFLIB.F77` which contains the send and receive subroutines plus the support subroutines called at the start and finish of the users FORTRAN code. The FORTRAN code was compiled with the 3L parallel FORTRAN compiler version 2.0 and then linked into the occam harness using the D705B linker. The INMOS toolset was then used to configure and run the system.

3.4.3 The Results

The results of the functionality tests on the geometric harness are contained in the tables which follow :

- All to one test with a 128 integer message (Table 3.1).
- One to all test with a 128 integer message (Table 3.2).
- All to all test with a 128 integer message (Table 3.3).

The final test results for the geometric calculation simulation are in table 3.4. All messages are 128 integers long.

No. of loops	Time/seconds
1000	2.29
5000	11.44
10000	22.88

Table 3.1: All to one test.

No. of loops	Time/seconds
1000	2.33
5000	11.63
10000	23.27

Table 3.2: One to all test.

3.4.4 The Conclusions

The first three tests (which are wholly communication) give the following conclusions. Firstly they work indicating that even though not formally deadlock free the harness performs under heavy communications loading. As the number of messages in the system grows the time taken grows linearly for all of the communications tests. From this we can calculate an average message handling time for each test (Table 3.5).

The message handling time for the all to all test is better than the tests involving a single source or sink because there are more processors handling more messages more of the time.

The simulated geometric calculation gave a linear increase in time as the number of loops is increased. Each loop has nine communications per processor, eight for an edge swap and one to the master. The work loop is one million floating point operations per loop. Using the average message handling times from above we can produce a worst and best estimate of the time taken per million floating point operations (Table 3.6).

A single transputer benchmark with a work loop of one million floating point operations was implemented to give a value for comparison with the geometric harness. The standalone benchmark was implemented using 3L parallel FORTRAN. The loop took 1.8 seconds to execute. From this figure and the values in table 3.6 the expected speedup when using the harness can be calculated. The speedup from one to four processors is by a factor of approximately 2. This is an efficiency of 50.22%. Thus the

No. of loops	Time/seconds
1000	8.64
5000	43.18
10000	86.38

Table 3.3: All to all test.

No. of loops	Time/seconds
10	36.04
50	180.27
100	360.56

Table 3.4: Geometric test simulation test.

Test	Time/seconds
all to one	0.57
one to all	0.58
all to all	0.34

Table 3.5: Average message handling times.

expected efficiency when using the harness is about 50%.

3.5 Conway's Game of Life Case Study

3.5.1 A Description

The game of life is not so much a game as a simulation. The simulation occurs on an infinite square grid of sites. Each site in the grid has two states. It is either alive or dead. The state of a site in the next step or generation of the simulation is determined by the number of alive neighbours in the previous generation. The rules are as follows :

- If a site is alive and has less than two live neighbours it is dead in the next generation. It has died of loneliness.
- A site with exactly two live neighbours retains its state from the present generation into the next.
- A site with exactly three live neighbours remains alive if it is in the present generation and becomes alive if it is dead in the present generation.
- A site with more than three live neighbours dies in the next generation. It has died of overcrowding.

	Time per Mflop/seconds
best	0.896
worst	0.898

Table 3.6: Mflop time estimates.

So local rules completely define the state of the whole grid at the next generation. The interesting property of life is that these local rules govern the evolution of global structures. Some life patterns die out, some cycle through growth and decline and others grow without limit. The game of life is an example of a systolic array. A systolic array is a system where many simultaneous local calculations produce a global co-operation.

The game starts with some occupied sites and it is then allowed to develop. Full explanations of the rules and some interesting facets of the game can be found in articles by Martin Gardiner [18],[19].

3.5.2 The Code

The code for the game of life was written in FORTRAN 77, compiled with the 3L compiler. The only special features of the code are :

- Use of 3L library subroutines to access the transputers clock for timing purposes.
- Use of a 3L library supplied with the examples which gives some basic access to the cga graphics facilities on the PC.

Pseudo-code structures for the fmaster and fworker processes are as follows :

```
FMASTER

initialise
receive data for initial generation
display initial generation
LOOP for number of generations
        receive data for this generation
        display this generation
END LOOP
shutdown



FWORKER

initailise
send initial generation to master
LOOP for number of generations
        edge swap to get data needed to update
        update to next generation
        send this generation to the master
END LOOP
shutdown
```

The display may be disabled by commenting out the sends and receives before the beginning of the loop and at the end of the loop on each processor. If this is done

the data from the final generation should be sent from the workers after the end of the loop. The master should collect the data from the final iteration for display or storage in a file.

3.5.3 The Implementation

A game of life was initially written in FORTRAN 77 for a single transputer implementation of the problem. The code was then split into a master and worker processes which were placed in the harness. Communications were added to effect the edge swaps and collecting of data on the master for display. The code was then run on the network used for the benchmark illustrated in figure 3.2. For both the standalone and the harness version the following systems were run :

- 200 × 200 sites, periodic boundary conditions, with display to the PC hercules graphics card running cga simulation software.
- 200 × 200 sites, periodic boundary conditions, with no display.
- 400 × 400 sites, periodic boundary conditions, with no display.

Comparisons can then be made between the standalone version and the version in the harness.

Full code listings for the harness life are contained in appendix D.

3.5.4 The Results

The standalone version of life running on one of the transputers described in the benchmark section produced the following results :

- 200 × 200 grid with display (Table 3.7).
- 200 × 200 grid without display (Table 3.8).
- 400 × 400 grid without display (Table 3.9).

No. of loops	Time/seconds
10	48.43
25	115.21
50	226.56
75	337.77
100	449.03

Table 3.7: Stand Alone Life (200 × 200 grid, with display).

The harness life ran on a two by two grid of processors described in the benchmark section. The following results were produced.

- 200 × 200 grid with display (Table 3.10).
- 200 × 200 grid without display (Table 3.11).
- 400 × 400 grid without display (Table 3.12).

No. of loops	Time/seconds
10	11.00
25	26.89
50	53.39
75	79.89
100	106.39

Table 3.8: Stand Alone Life (200 × 200 grid, no display).

No. of loops	Time/seconds
10	46.87
25	115.12
50	228.88
75	342.63
100	456.38

Table 3.9: Stand Alone Life (400 × 400 grid, no display).

3.5.5 The Conclusions

It is noted that for both the standalone and harness versions of life displaying the data on the screen at each iteration costs an awful lot of time. We will consider, therefore, the results with no display. The time taken in all the tables (3.8, 3.9, 3.11, 3.12) increases linearly with the number of iterations of the loop. Comparing values in tables 3.8, 3.9 and 3.11, 3.12 it is seen that a speed up of a factor of 2.0 to 2.2 is achieved. This is in line with the benchmark predictions. The efficiency of the system is between 52% and 54%, again in line with expectations.

3.6 Overview of the Geometric Harness

A harness for the geometric paradigm on a toroidal grid of transputers has been produced. A simple 'one direction of flow' routing algorithm was used to allow point to point message passing between any two processors in the system. Full descriptions of the processes, code templates and use of the harness have been given. A benchmark

No. of loops	Time/seconds
10	43.25
25	101.79
50	199.33
75	296.86
100	394.41

Table 3.10: Geometric Harness Life (200 × 200 grid, with display).

No. of loops	Time/seconds
10	5.29
25	12.83
50	25.37
75	37.92
100	50.46

Table 3.11: Geometric Harness Life (200 $\times$ 200 grid, no display).

No. of loops	Time/seconds
10	21.14
25	52.38
50	104.05
75	155.62
100	207.42

Table 3.12: Geometric Harness Life (400 $\times$ 400 grid, no display).

case study was undertaken to show that the harness has good functionality under different communications and calculation conditions. The results from these tests were favourable so a typical geometric application was placed into the harness. This was Conway's Game of Life, a two dimensional cellular automata system. The application was successfully implemented in the harness system and a speedup by a factor of 2 noted, in line with benchmark predictions.

4 Review of other Migration Aids and Languages

4.1 Introduction and Policy

This section will give a brief insight into some of the presently available programming products for the transputer. The policy will be to try to implement two tests. The first will be communication to a filer from a chain of processors in the manner of the coarse farm harness. The second will be a geometric type test consisting of a cycle of edge swaps and calculation. The object of these tests will be to see what facilities are given to an applications programmer. Ease of use and clear indications of system capabilities are the prime points that will be looked for.

4.2 The Test Applications

The tests are indicative of the coarse farm and geometric harnesses described in the earlier chapters of this thesis. Screen output from a chain of processors and an idealised geometric test will be used. The screen output will consist of the string

```
"Hello world from processor X"
```

The output can be put into a loop to give some performance information.

The geometric test will consist of a one million floating point operation work loop with an edge swap of one kilobyte of data in the four directions of a grid of processors. A screen output may be attempted to give an indication of progress of the work. In each case no attempt will be made to write a harness for the test. Coding will be specific to the job in hand. In a test like this on a large number of systems it is hard to use all of a particular systems specific specialised techniques. Most of the test code can be classed as naively written.

4.3 Languages

The languages which will be used in these investigations all fall into slightly different categories. Occam is a parallel language which was designed with the transputer hardware. 3L parallel FORTRAN and Parsec Par.C are standard languages with extended features. In the FORTRAN the special features of the transputer are accessed by subroutines in libraries which are available at runtime. For Par.C the philosophy is to use language extensions to access the transputer features. With these three languages all the 'types' of language for the transputer are covered, a specialist language, a standard language with a procedural interface and a standard language with a language extension interface. All the systems give a good standard single transputer implementation, it is when programming multiprocessor systems that the methodology differs.

4.3.1 The Hardware Used in the Language Tests

For all the language tests, to allow some performance comparisons to be made, the same hardware was used. The system consisted of an OPUS PC II XT clone with the following transputer hardware.

- An inhouse [21] interface card allowing 20MHz link connection to the personal computers bus.
- Five inhouse 'Trice Transputer' cards [22] consisting of a T800-20 with 4 megabytes of 6 cycle external memory. All the links were set to run at 20MHz

For the coarse farm communication test the processors were connected in a chain from link 2 to link 0. (figure 4.1)

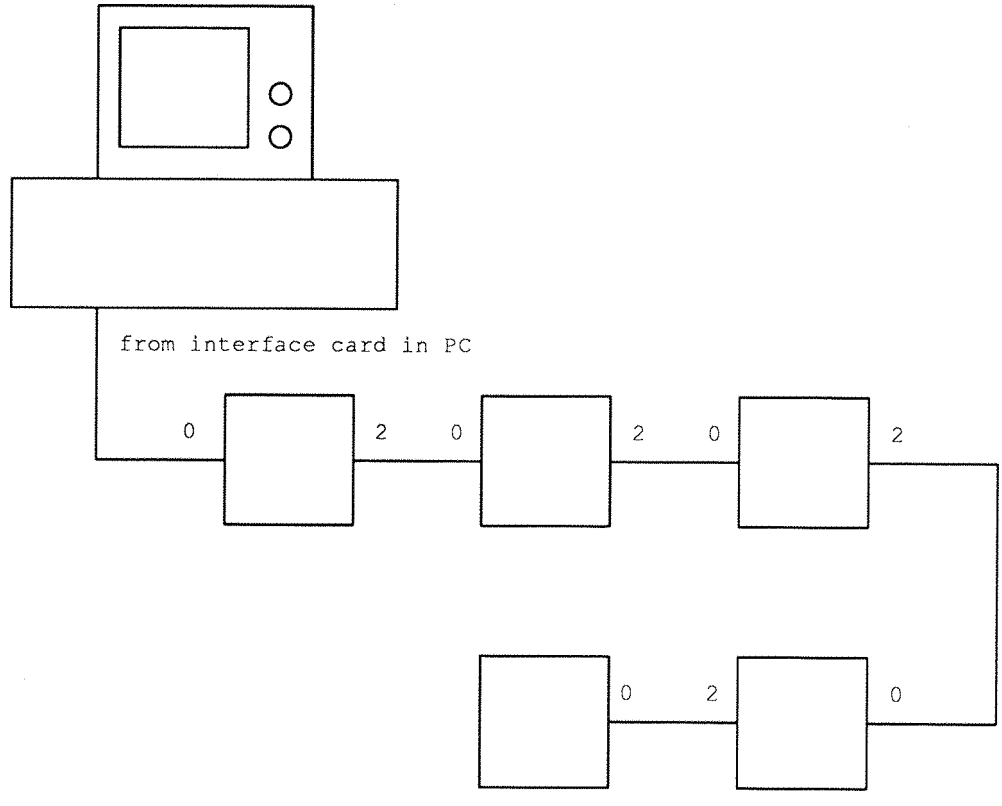

Figure 4.1: Hardware for Coarse Farm Test.

For the geometric test they were connected in a 2 by 2 torus with a control processor inserted in the fashion of the geometric harness. (figure 4.2)

Languages should provide the most flexible but potentially hardest to use route for parallel programming on transputers.

4.3.2 The New INMOS occam Toolset (D7205)

The new INMOS occam toolset implements occam 2 as described in earlier chapters and references. It is a system for building and debugging programs for networks of transputers combining high level language features with the ability to access the transputer hardware at the lowest level.

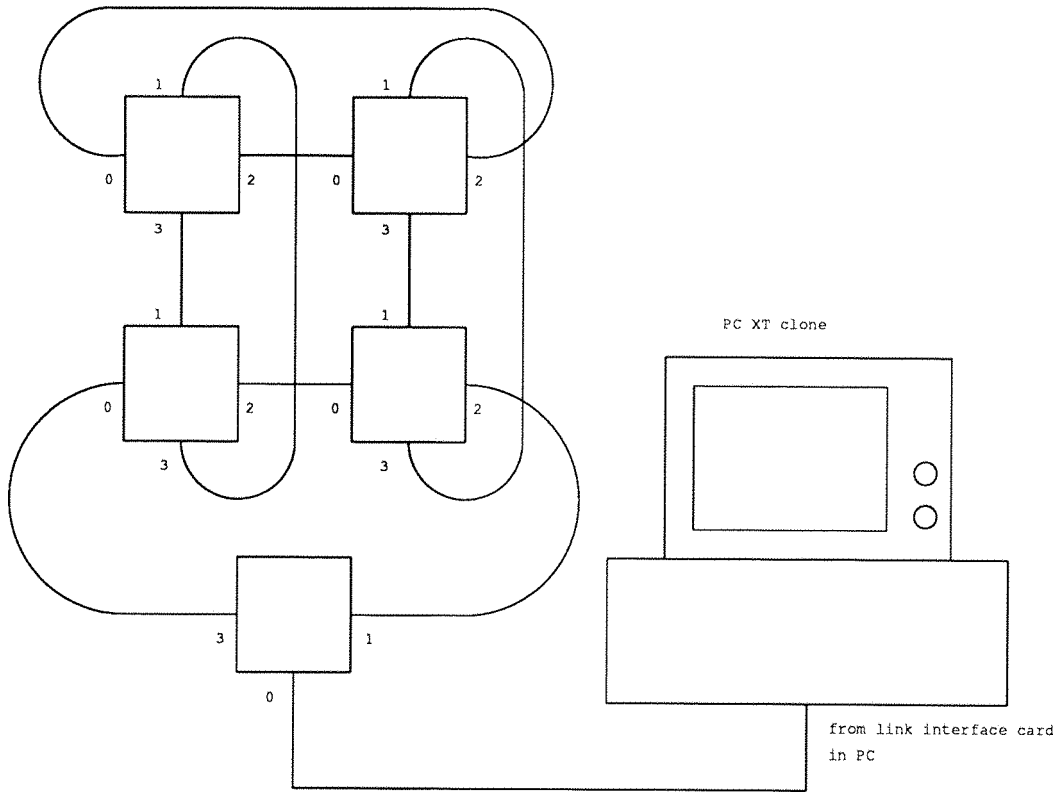

Figure 4.2: Hardware for Geometric Test.

Occam 2 was designed to reflect the architecture of the transputer hardware which leads to 'maximum coding efficiency' [23, page 7]. The occam programming model consists of parallel processes communicating via synchronised channels. As each process is independently executable, multiprocessor programming is a simple extension of the single processor case. A configuration description is used as an extension to occam 2 to allow distribution of processes over processors.

Occam 2 is deemed reliable because of its base in the mathematical theory of Communicating Sequential Processes (CSP). Because of its access to the transputer hardware it may be used for real time programming, fault tolerant link communication and other low level embedded applications [23, chapter 10].

The toolset consists of a compiler, linker, configurer, 'collector' and a server. The collector adds the bootstrap and initial rooting information needed to boot a network of transputers. In addition there is a debugger and some specialised tools such as an EPROM programmer. All tools can be either run on the host computer or on an attached transputer with at least two megabytes of memory. There is a T425 simulator which runs on the host computer to allow testing of code when no transputer hardware is available. There is no special environment or editor with the system and the tools are called from the host computers operating system prompt in the usual way. INMOS do, however, provide a folding editor called origami as freeware. This editor gives functionality similar to the INMOS Transputer Development System integrated environment editor [24].

The system may be run on an IBM PC running DOS, DEC VAX minicomputers running the VMS operating system or under the unix operating system on sun mi-

crosystems workstations of the 3 and 4 architectures.

The main difference from previous releases, (the most recent previous release was used in chapters 2 and 3), is a new configuration language. Instead of an extended occam 2 variant a new concept is employed. The programmer specifies separately a hardware description, a software description and a mapping between the two.

4.3.2.1 The Coarse Farm Test.

Occam is very much a do it yourself language. There are no special communication structures provided to ease the burden of the applications programmer. Access to common resources such as the screen is usually handled by a single processor for each resource. Libraries are provided to drive the a single resource from its controlling processor, but any other processors needing access to the resource must negotiate with the controller. An example of this is the INMOS B007 graphics card [25]. A transputer drives this card through a library of procedures, but any attached transputer network must communicate with the B007 transputer to gain access. No code is provided for the network to access the card so it must be written by the applications programmer. Similarly for the screen on the host computer various library processes are available to output messages to the screen, but only the first transputer in the network is physically connected to the host computer, so only this processor may output to the screen.

For the above reasons when implementing the coarse farm test using the new occam toolset the chain of processors passed the component parts of the message to be output (a string and an integer) to the processor at the head of the chain. This processor has direct access to the screen and so can output the messages made up of the necessary data using the host input/output library routines.

The code was written in two parts. At the head of the chain the processor inputs the strings and integers from the rest of the chain processors and outputs the appropriate messages. On the rest of the chain the component parts of the message are output. Any passing on of the data from processors higher in the chain is also carried out. The whole test was configured using the new configuration language which describes the hardware, software and a mapping from one to the other. The direction of the links is worked out by the configurer by their use in the code.

Although there is no integral environment for the toolset the building of the program from the three source files (root processor, worker processor and configuration sources) can be automated using the provided 'imakef' utility to produce a makefile for an appropriate make program. This was done by the author.

The results of the coarse farm test are shown in table 4.1. The number of loops refers to how many times each processor outputs the message, so for the test as implemented one loop is equivalent to four messages on the screen. Timing of the system was achieved by using the access to the systems clocks provided by occam.

4.3.2.2 The Geometric Test.

Any communications structure required by the applications programmer in the occam toolset must be written by the applications programmer. Thus for the geometric test explicit handling of the message routing through the master processor (see figure 4.2) must be implemented. As the number of workers is only four connected in a two by two torus there is no special routing needed on the main worker network. Again in occam, although many facilities are available in the rich language, the applications

No. of loops	Time/seconds
1	0.15
10	1.61
50	8.46
100	17.16

Table 4.1: Coarse Farm Test for the New Occam Toolkit.

programmer has little or no help in his or her task of implementing a problem solution. All communications have to be explicitly handled by code written by the programmer.

The geometric test implementation code consists of a one kilobyte edge swap along the four links for the worker processor followed by a loop of one million floating point adds simulating work on each worker processor. The occam construct `PAR` was used to elegantly implement the edge swap. The configuration level consists of the three levels described before. A makefile was produced and used. The results are in table 4.2.

No. of loops	Time/seconds
10	13.06
100	143.69

Table 4.2: Geometric Test for the New Occam Toolkit.

The number of loops refers to the number of iterations of the whole system through the edge swap, work loop cycle on each worker.

4.3.2.3 Impressions of the New INMOS Toolset

The main change between the new toolset and the previous versions is the configuration level. This new configuration level, I feel, is more complicated and verbose than necessary. There is no way to define the direction of the link connections as there was in previous releases. This may not seem a problem, but during the development of the hello test the following problem was encountered : The processors were linked in a chain, a loop of channels was declared at the configuration level and passed to the individual processor code. It is the policy of the author to test code piecemeal. With this in mind an integer value was set up to pass along the channels 'down' the loop from the last worker to the root processor. The root processor then would write a message to the screen indicating success. Note the channels going 'up' the loop were not used in this initial test. All proceeded well with the building of the code until the system was configured. The configurer warned that it could not work out a direction for the unused channels then it hung up. Inspection of the information from the compiler confirmed that the product release was in use. The INMOS Business Centre was contacted about the problem.

The new occam toolset provides a powerful and elegant method of programming transputers. All the tools are called from the host computers operating system prompt like any other unbundled toolset. An automatic makefile generater is included in the

system which greatly aids coding if a make utility is available. The configurer automatically determines the placement of channels onto links, but no override is provided so that channels which are unused (for example in the initial hello test development) cause problems as described. Occam provides some of the most efficient code running on transputers but the general programming community have been reluctant to learn this new language. They prefer languages they are familiar with such as C and FORTRAN. Attempts to provide familiar language interfaces to transputers, and so aid code migration, are described in the next two sections.

4.3.3 The 3L Parallel FORTRAN Language (v2.0)

The philosophy of 3L in Edinburgh has been to produce a family of compilers which access the parallel features of the transputer using a procedural interface. The FORTRAN system contains a very strict FORTRAN 77 compiler with associated linker, configurer and server. There is no special editor environment with the system which runs on an IBM PC under DOS. The server 3L supply is 'afserver' which is a customised version of an old INMOS server. INMOS have ported the 3L family of compilers to run with their new server 'iserver'. With this port it is possible to mix 3L compiled code with occam from the old INMOS toolset (D705B). This was the method used in previous chapters.

There are two concepts put forward by 3L in their software model [26]. The first is the concept of tasks, which is similar to occams process model. Each task has its own area of memory for code and data plus vectors. The vectors contain input ports and output ports. There is no sharing of memory between tasks and the ports are connected together to form channels as with occam. Tasks are connected together using a configuration language. The configuration language is not an extension of the language being used, but is the same for all compilers in the family allowing code from the compilers to be mixed. More than one task may be placed on one processor. Any number of connections may be made between tasks on the same processor but the usual four links must be taken into account when connecting tasks on different processors. This is basically the same as occam. The major omissions in the configuration language are replicators and ways of easily passing parameters to tasks at this level.

Access to channel communications is by library subroutines. Libraries are also provided to implement the ALT construct, provide access to the transputers timer and other low level aspects [26, chapter 17].

In addition to tasks the programmer may use threads within a task. Each thread has its own stack but can share code and data with other threads in the same task. This is like a mini shared memory system. Semaphores are used to manage the sharing of memory space.

3L also provide with the system a harness for farm type applications called the 'flood fill configurer'. For this harness the programmer must produce a master task and a worker task. These tasks are then linked into the harness. At load time the transputers network attached to the host computer is analysed or 'wormed' and the master task plus routing code is loaded onto the first available processor. The rest of the processors are loaded with the worker task along with the relevant routing code. The system should then run as a farm. It can be seen that no action is needed if processors are added or removed from the system as the worm will notice and do the loading appropriately. If only one processor is available then a copy of the worker task is

loaded along with the master task onto it.

The author has used the flood fill configurer in past work and found it not to be very flexible. Problems were encountered when sending large initial data packets out to the workers. Processors were 'frozen out' by the routing software and only a fraction of the available processors returned results. It is not intended to use the flood fill configurer in this work.

4.3.3.1 The Coarse Farm Test

As with occam in the previous section 3L parallel FORTRAN provides no general access to the file server from all the processors in a network. Only one processor has access to the filer. Code for the system is written for each processor and compiled then configured for the network to be used. The implementation method used with the parallel FORTRAN is the same as occam (communicating sequential processes). The component parts of the message to be output were passed down the chain using the communications subroutines provided with the system. The processor at the head of the chain then outputs the messages to the screen using a FORTRAN `PRINT` statement.

There is no replicator and no way to pass parameters into a task called at the configuration level. This lead to a separate code being produced for each processor in the network. The configuration language is made cumbersome by these omissions and coding more difficult.

The old INMOS server 'afserver' (modified by 3L) is also very much slower than the 'iserver' used in the occam toolset (table 4.1). This is seen in the results below. (Table 4.3) Timing was by accessing the transputer timers via the 3L subroutine library.

No. of loops	Time/seconds
1	0.67
10	6.66
50	33.32
100	66.63

Table 4.3: Coarse Farm Test for the 3L FORTRAN.

4.3.3.2 The Geometric Test.

The method of parallelism used by the author for the geometric test was that of tasks rather than threads. For this reason the messages in the test were organised to execute in a set order, differing between processors. The lack of configuration level parameters for the tasks lead to individual code for each processor. Like occam there are no communications structures provided by the system which would be appropriate for this test. (A farm harness is provided with the system - The Flood Fill Configurer.) The results of the one kilobyte edge swap and one million floating point operation work loops are shown in table 4.4. The number of loops refers to whole system iterations.

4.3.3.3 Impressions of 3L Parallel FORTRAN

Parallel FORTRAN provides almost all of the features of occam through subroutines in libraries. It provides heavy weight, task based, parallelism and lightweight, thread

No. of loops	Time/seconds
10	42.88
100	471.68

Table 4.4: Geometric Test for the 3L FORTRAN.

based, parallelism. The author does not like the subroutine interface as much as the concept of language extensions (see Par.C next). This is because much of the elegance of a pure parallel language (one designed for the purpose) is lost when using this method of accessing the parallel features of the hardware.

The configuration language is common for all 3L languages providing the ability to mix FORTRAN, C and Pascal. There are two major omissions in the configuration language which make it cumbersome. There is no way to replicate a statement in the configuration file and no way to pass parameters into the FORTRAN tasks. These omissions really cause great inconvenience.

Parallel Fortran does provide a communications harness for the farm paradigm - the Flood Fill Configurer. Libraries and templates for the code needing to be written by the applications programmer are fully covered in the documentation [26]. An example of the implementation of the drawing of the Mandelbrot set is also given in the manual. The author has had cause to investigate the use of the Flood Fill Configurer during the coarse of an industrial project while employed at Transputer Technology Solutions (see appendix F). Large initial data packets were needed for the application being used and the harness provided performed badly, 'freezing out' processors. Contact with 3L yielded the sources of the harness routing system for modification and the promise of a fix in the next release.

The 3L parallel FORTRAN package provides a familiar language to most programmers with the functionality of occam added by library subroutines. A farm harness is provided with the system. The configuration language is not very flexible and there is no provision of a makefile generater to aid code production.

4.3.4 The Par.C Language (v1.31)

The Par.C language from Parsec Developments, Leiden, Netherlands is a standard C system with language extensions rather than the procedural interface favoured by 3L. The system recognises the Kernighan and Richie C standard with some extensions defined by Harbison and Steele. Most of the draft ANSI C standard is also implemented and any exceptions are listed in the manual [27, section 3-3]. 'The choice of language extensions, as opposed to the inclusion of parallel programming facilities in special runtime libraries, has been made for reasons of clarity of concepts and parallel C source code, and also to provide the maximum of flexibility in the use of the parallel facilities.'

The language extensions consist of a channel data type and two types of statement [27, section 1-2]. The `par` statement has an optional replicator and then one or more program statements to be executed in parallel. This is exactly the same as occam. The `select` statement has a number of `alt` statements contained in it. Each `alt` has a condition and/or an input from a channel guard. This `select` function implements the occam `ALT` construct.

The system consists of a compiler, an assembler (for GUY), a linker, a loader/server and a runtime system. Transparent access is given to files and the standard input/output from all processors in a network. Very thorough explanations of the extensions and examples are given in the manual text.

The Par.C bootsystem dynamically loads code by first worming the transputer network to find out what is there. The code loaded onto each transputer is a full copy of the code written. Execution on individual processors is influenced at runtime. In this way Par.C acts a bit like a cross between a single instruction multiple data (SIMD) machine and a transputer network programmed in occam (MIMD). It is a single program multiple data system. After the boot system worms the network it sets up a a system structure on each processor. This structure contains information such as the processor identification number and its boot link et cetera. This information concerning network layout can be accessed and acted upon at runtime leading to a true multiprocessor multiple data program [27, section 5-9].

As the program loads and starts up it goes through the following stages :

1. Boot the root transputer and investigate the available network.
2. Load the program to all processors.
3. Execute the C startup code
4. Execute the main C program

The fact that all the code is loaded to all the processors and then manipulated at runtime means that there is no configuration level as there is with occam and parallel FORTRAN. The price paid is higher memory usage and loading of code onto a processor which will not necessarily use it.

Compiler directives are used to generate code for different types of transputer. By default the compiler will produce code for a network consisting of a mix of T4 and T8 transputers. Specifying by directive that T8s only will be used greatly improves performance. The usual 'tricks' can be used to improve code performance [20].

No special programming support environment is supplied with Par.C. Standard editors are used to input source code then the tools are called from the operating system prompt as with most compilers.

The Par.C system may be installed on IBM PCs, sun microsystems workstations from the SUN3 and SUN4 ranges and also under the Helios operating systems for tranpsuters.

4.3.4.1 The Coarse Farm Test.

The Par.C system initialises a system structure on each processor after worming the network. Transparent access is provided by the system to the file server from all processors. By consulting the system structure and directly calling the C `printf` function the coarse farm test is implemented. Timing of the system was by the provided runtime loader option to time the execution time of the system. The results of the test are shown in table 4.5.

No. of loops	Time/seconds
1	2.00
10	3.00
50	10.00
100	17.00

Table 4.5: Coarse Farm Test for the Par.C Language.

4.3.4.2 The Geometric Test.

Direct link access is given by the Par.C system via function calls. The `par` function language extension gives an elegant method of implementing the edge swap. No configuration is necessary as the worm and system structure cover this concept. The results for this test are in table 4.6.

No. of loops	Time/seconds
10	58.00
100	562.00

Table 4.6: Geometric Test for the Par.C Language.

4.3.4.3 Impressions of Par.C

Par.C is easy to program in and is supported by excellent comprehensive documentation. It provides a familiar language for applications programmers with language extensions to cover the channel data type and the `PAR` and `ALT` constructs from occam. Library functions are provided to directly access the links and most low level transputer features. There is no configuration level. At boot time the network is wormed and all the code is loaded to all the processors. A structure is setup at load time with system information in it (processor identification number, bootlink *et cetera*) which can be accessed at runtime to influence the running of the code on the processors. This is a single program , multiple data system (SPMD).

4.4 General Harnesses

There will be two general harnesses investigated in this chapter. Both were developed at the University of Southampton (UK) and they provide communication structures for occam programs with some facilities for embedding code written in other languages. The Euler Cycle Configuration Language (ECCL) is a preprocessor for the Transputer Development System (TDS) from INMOS. It allows arbitrary interconnection between processes with no regard for the transputer hardware. Multiplexing of messages down the four serial links per transputer is handled automatically. The Virtual Channel Router (VCR) provides a similar system, where process interconnections are arbitrary, for the new occam toolset from INMOS. It is built into the compiler for the toolset

and allows the simulation of code written for the newly announced T9000 family of transputers [28].

The hardware used for these tests is the same as for the language tests, shown in figure 4.2. As arbitrary channel connections are allowed in both systems there is no need to connect a chain of processors for the coarse farm test. The hardware set up for the geometric test allows more links to be available for the routing system provided by the harnesses. ECCL can be considered a research prototype where as VCR is a 'product' of an Esprit project P2701 PUMA.

4.4.1 Euler Cycle Configuration Language (v1.08)

ECCL is a harness configuration tool which runs under the INMOS Transputer Development System TDS (D700D product). It provides through routing for an arbitrary network of occam processes and was conceived to relieve the burden of consideration of the hardware restrictions when using transputers with the communicating processes style of program development.

Processes communicate through logical channels. This system was designed as a first step towards higher level programming languages [29] and provides synchronised occam style communications. The most important feature is the decoupling of the physical processor network description from the programmers process network description. (The new occam toolset now does a similar thing but ECCL pre-dates it.)

There is one constraint on the target hardware network and that is that it must be a graph on which an euler cycle can be constructed. An euler cycle traverses a graph completely such that each edge of the graph is traveled along exactly once. There are well know formal methods for constructing these cycles [30]. This theory of euler cycles was used at Southampton during the ESPRIT 1 project P1085 'supernode' to design switching strategies which are implemented in hardware in the supernode machines [31]. For practical purposes the ECCL will be able to construct an euler cycle provided there is an even number of connections in the transputer network. The host processor (and root transputer if desired) forms a spur off the main cycle. On the main euler cycle part of the network, with all transputer links connected, it can be seen that the cycle passes through each processor twice. Communications move round the cycle. This is a relatively simple loop of communications processes which can be proved to be deadlock free if eager readership is adhered to. When the network is heavily loaded communications occur only by traveling round the main euler cycle. If there is some spare capacity in the communications network it is possible to 'short cut' on a processor from its first cycle position to its second cycle position if this move is favorable. This is local to a processor and can be checked locally so as not to produce deadlock in the system. This short cutting greatly increases the communications efficiency of the harness.

When the ECCL configurer is run it builds in the routing necessary for the harness automatically. Communications are synchronised, as with occam, and because of this a large number of buffers and large routing tables are not needed. ECCLs memory usage per processor does not grow with network size.

The tool consists of a TDS executable (EXE) which takes an ECCL fold (TDS is a system based on a folding editor) and produces a TDS EXE and associated PROGRAM to implement the desired system (for explainations of PROGRAM and EXE see [24]). These foldsets are then compiled and configured in the usual TDS way. The users pro-

cesses (SCs) have the same restrictions as the usual TDS configuration level processes [24]. Interface procedures are provided to give access to the communication system (which must not be accessed directly) and to implement the ALT construct.

The ECCL runs under the TDS product. TDS may be installed on IBM PCs, VAX computers, SUN workstations, apollo workstations et cetera.

4.4.1.1 The Coarse Farm Test

Code for the processors in ECCL is written in occam. The main difference between 'real' occam and writing for ECCL is that the channel access when using ECCL is by procedure call as opposed to direct access. Templates are provided for any code to be written to go in the harness via examples in the clear documentation from the author of ECCL [29].

The major change to TDS is that the code for the whole system is written in one place. Traditional TDS coding involves writing an EXE to run on the root processor (which is also running TDS) and a PROGRAM to run on the rest of the network. ECCL automatically generates these two parts from a single source.

ECCL does not provide general file server access from all the processors in the system. For this reason code similar to that written for the new occam toolset was produced . The components of the message to be output were sent to the root processor which used the input/output libraries to perform the write to the screen. Messages in ECCL are all byte arrays so loading and unloading of other types of data is left to the user.

At the configuration level there are three parts. The `NETWORK` section describes the hardware connections (figure 4.2). The `HARNESS` section describes the interconnections between the processes. This can bear no relation to the hardware set up. The process connections for the coarse farm test are shown in figure 4.3. The final `PLACED PAR` section places the processes onto the processors. Timing of the system was by using the occam access to the transputers timers. The results are in table 4.7.

No. of loops	Time/seconds
1	0.10
10	1.21
50	7.11
100	14.47

Table 4.7: Coarse Farm Test for ECCL.

4.4.1.2 The Geometric Test

The code for this test was written in two parts. The root processor times the system and receives signals from the workers to achieve this. The workers do the edge swap and work loop. Communications in ECCL are occam like (that is synchronous) and so some attention to the order of message communications must be taken.

The `NETWORK` section for the test is unchanged from the coarse farm test. The processes were configured as shown in figure 4.4.

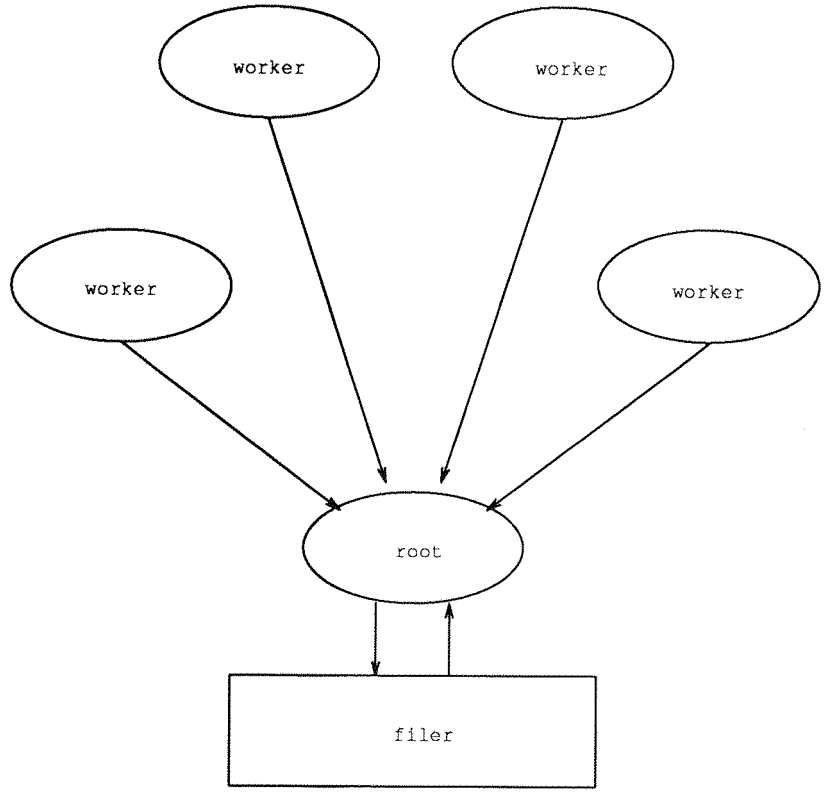

Figure 4.3: Process Connections for the ECCL Coarse Farm Test.

Results from this test are in table 4.8. Performance is close to that of the new occam toolset geometric test.

No. of loops	Time/seconds
10	15.29
100	164.50

Table 4.8: Geometric Test for ECCL.

4.4.1.3 Impressions of ECCL

For the price of moving to a procedural communications interface total freedom of process connections with no regard for the hardware is gained. Performance for the tests is on a par with pure (new toolset) occam. Although an experimental system for researching whether such a system is desirable and practical ECCL provides a level of freedom when programming in occam on multiple transputers under TDS which has only been equalled with the development of VCR, the next system investigated in this review.

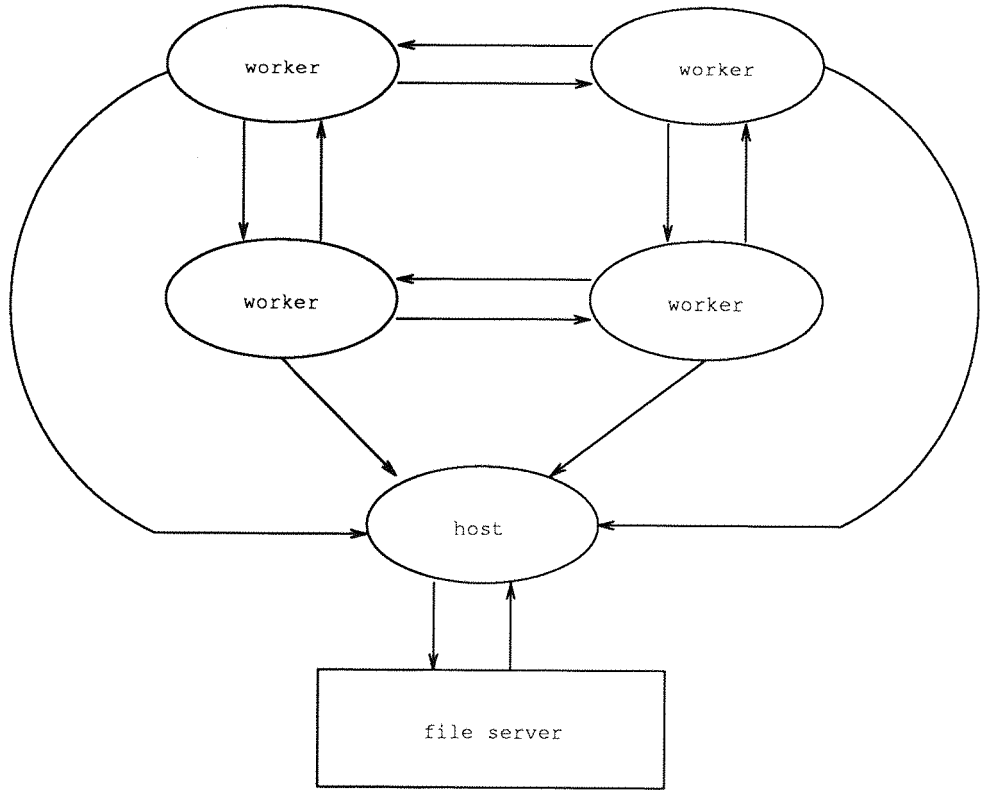

Figure 4.4: Process Connections for the ECCL Geometric Test.

4.4.2 Virtual Channel Router (v2.0)

In a full occam implementation arbitrary process interconnection is implied. This is implemented by the INMOS standard occam compiler for a single transputer, but for multi-transputer systems the hardware restriction of only four links per processor comes into force. The Virtual Channel Router (VCR) system is the first step to implementing full occam for multi-transputer systems. It does this by providing virtual channels and processors. It implements arbitrary message routing on a network of present day 32 bit transputers. The system is built into the new occam toolset which retains its facilities to integrate C from the INMOS compiler. An INMOS FORTRAN compiler will be available in the future. Installations are available for the IBM PC family and for SUN workstations [32].

The inspiration behind VCR is similar to ECCL. It is to facilitate movement of code between one hardware set up and another. It also provides a programming system for present day transputers which will be valid for the next generation of transputers. It provides in software a system which will be available in hardware when the T9000 and its associated through routing link chip, the C104, become available in 1992. The system provides arbitrary process communication and removes the burden of writing the routing software from the applications programmer.

The user of the system writes standard occam processes referencing some adjusted libraries. These processes are then placed on virtual processors connected by virtual channels. Using the INMOS checker software and a router generation tool a hardware description is separately constructed. The resolving of the users process network onto the hardware processor network occurs at load time when both files are given

as parameters. This means that VCR binaries are executable on any VCR installation provided a locally correct hardware description file is available.

The main components of the system are a compiler, a loader and some source generation tools. The main feature of VCR, as with ECCL, is the decoupling of the software process information from the hardware processor information.

4.4.2.1 The Coarse Farm Test

VCR provides a procedure called `hosthook` [32] which handles a connection between the host file server and an arbitrary process in the system. Any number of `hosthook` procedures may be called so any number of processes may have direct access to the server. VCR like ECCL allows totally arbitrary interconnections between processes with no regard for the hardware. Standard occam is written with the channels directly accessed , and this is the case with VCR as the system is built into replacement tools for the new occam toolset.

With direct access to the filer each worker calls the host input/output library procedures (VCR supplies a replacement for this hostio library) to write the message to the screen. Each worker then signals to a master process which is handling the timing of the system. The hardware aspect of the system is handled by the routegen utility. Mapping of the virtual processors onto the physical processors occurs automatically at load time. This leaves just the process description to be written by the applications programmer as a VCR configuration file. The test was configured as shown in figure 4.5.A 'vmakef' utility is supplied with the system to create makefiles and allow the automation of program building. The results of the test are shown in table 4.9.

No. of loops	Time/seconds
1	0.19
10	1.53
50	7.49
100	14.93

Table 4.9: Coarse Farm Test for VCR.

4.4.2.2 The Geometric Test

The geometric test was implemented in occam and had hardware independent process interconnections. (Figure 4.6) The results of the geometric test for VCR are shown in table 4.10.

No. of loops	Time/seconds
10	45.12
100	451.17

Table 4.10: Geometric Test for VCR.

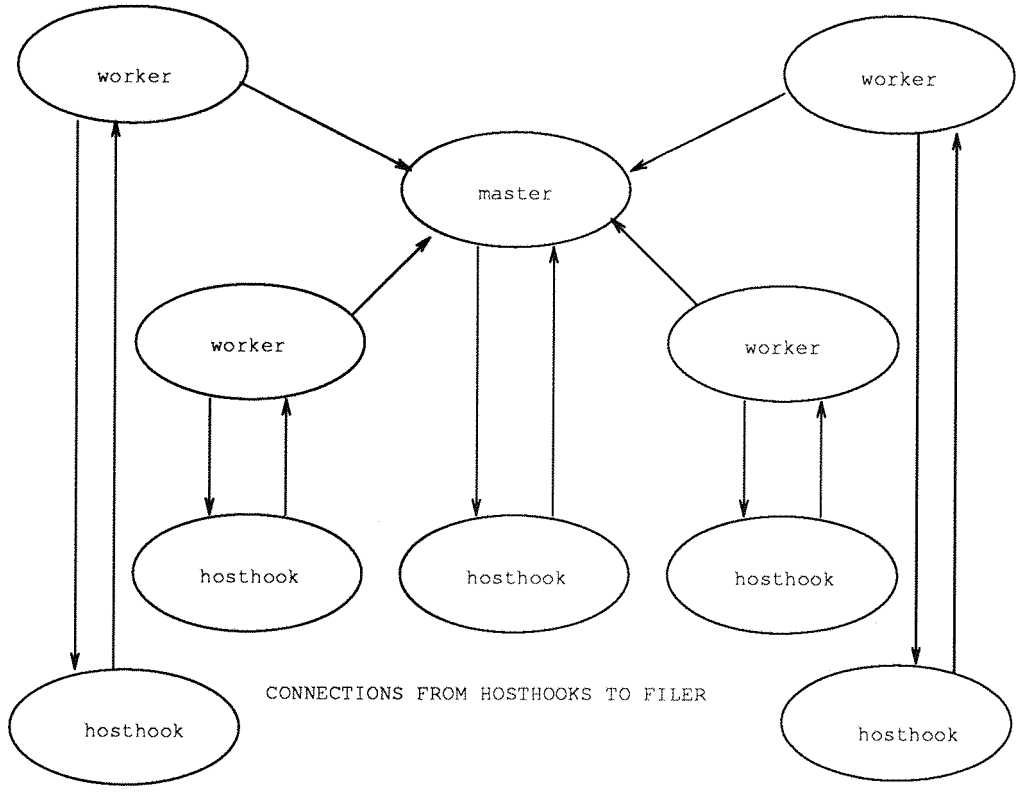

Figure 4.5: Process Connections for the VCR Coarse Farm Test.

4.4.2.3 Impressions of VCR

VCR gives the applications programmer full freedom of process interconnection with no change to the way he or she programs in occam. Certain tools and libraries in the new occam toolset are replaced with VCR equivalents and that is all there is to it. Any compiled VCR code will run on any VCR installation provided a suitable hardware description file is available. True portability is available through VCR. Language mixing is is also available (C and occam now with FORTRAN to come) to provide a truly flexible comprehensive system. From the general work of ECCL, VCR, aimed at the next generation T9000 transputers, is the way forward in transputer programming.

4.5 Programming Support Environments

These are systems which fall between the two classifications of general harness and operating system. A programming support environment usually does not provide an enclosed system like a true operating system does. It is a comprehensive set of tools and libraries to allow a particular progamming task to be undertaken with much of the burden of low level considerations removed from the applications programmer. The express system will be looked at in this review.

4.5.1 Express (v3.0)

The express system provides a large variety of tools and facilities packaged with three different compilers. The 3L C and FORTRAN systems are available plus the Logical

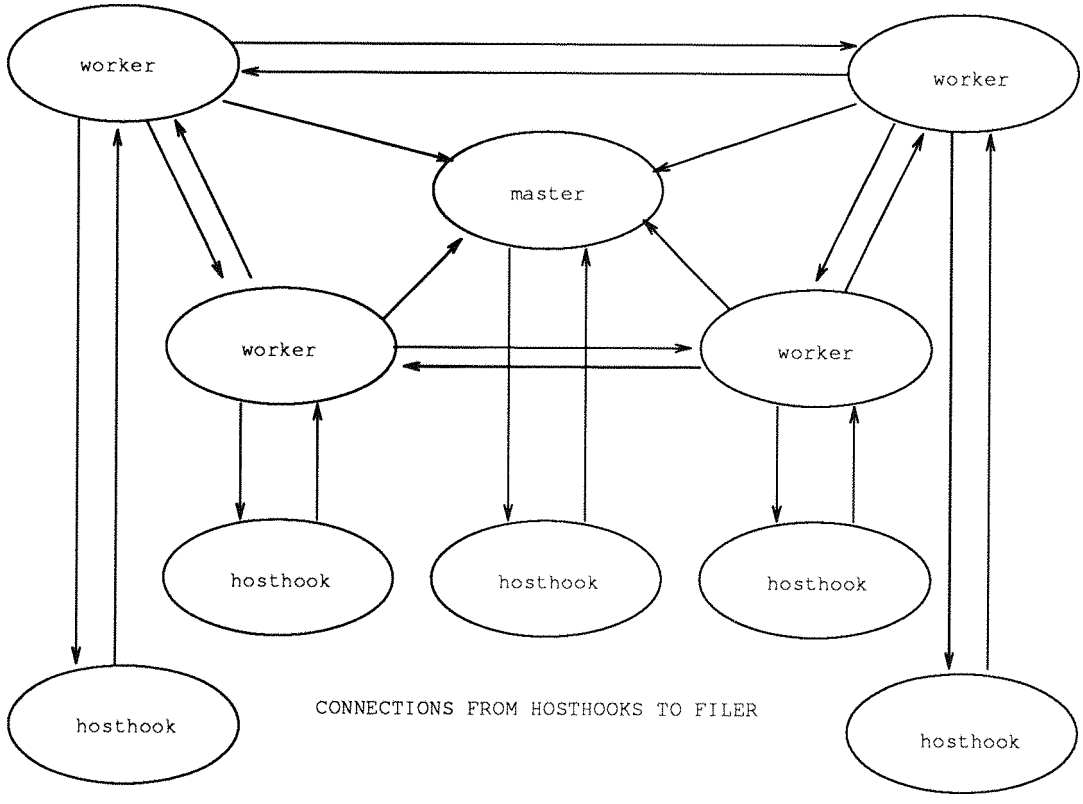

Figure 4.6: Process Connections for the VCR Geometric Test.

Systems C system. The system will run on any network of transputers, using one of two models. The 'cubix' model runs a server on the host computer and a transputer program on the transputers. This model is a single program, multiple data system like the Par.C system. Alternatively a specialised host program can be written for the transputer network to communicate with.

The system is started up in the following way :

- A configuration tool is run which worms the network to produce a hardware description file for the attached transputers.
- The system is then initialised. This involves booting a kernel process onto each processor.

When the system is set up programs may be loaded onto the processors. The 3L systems compile on the root transputer, and as the documentation states, are pathological towards express. That is they reset the transputer network. Thus when using the 3L systems the initialisation stage must be gone through whenever an express program needs to be run. The Logical Systems C compiles on the host computer, and this was the system used in the tests in this section.

Express is not guaranteed to be deadlock free on all networks. The documentation [33] claims that for loops, grids and hypercubes the system is deadlock free. An illustration in the manual indicates that a torus with a processor inserted into one of the loops (like figure 4.2) should be acceptable. The figure 4.2 hardware set up was, therefore, settled upon to be used for both tests, as it provides the maximum link

connections. On trying to initialise the system, however, although the worm operated correctly, the loading of the kernel processes failed as the system was unable to resolve the multiple link connections between processors. Removing two connections allowed express to initialise correctly. This gave the hardware set up shown in figure 4.7.

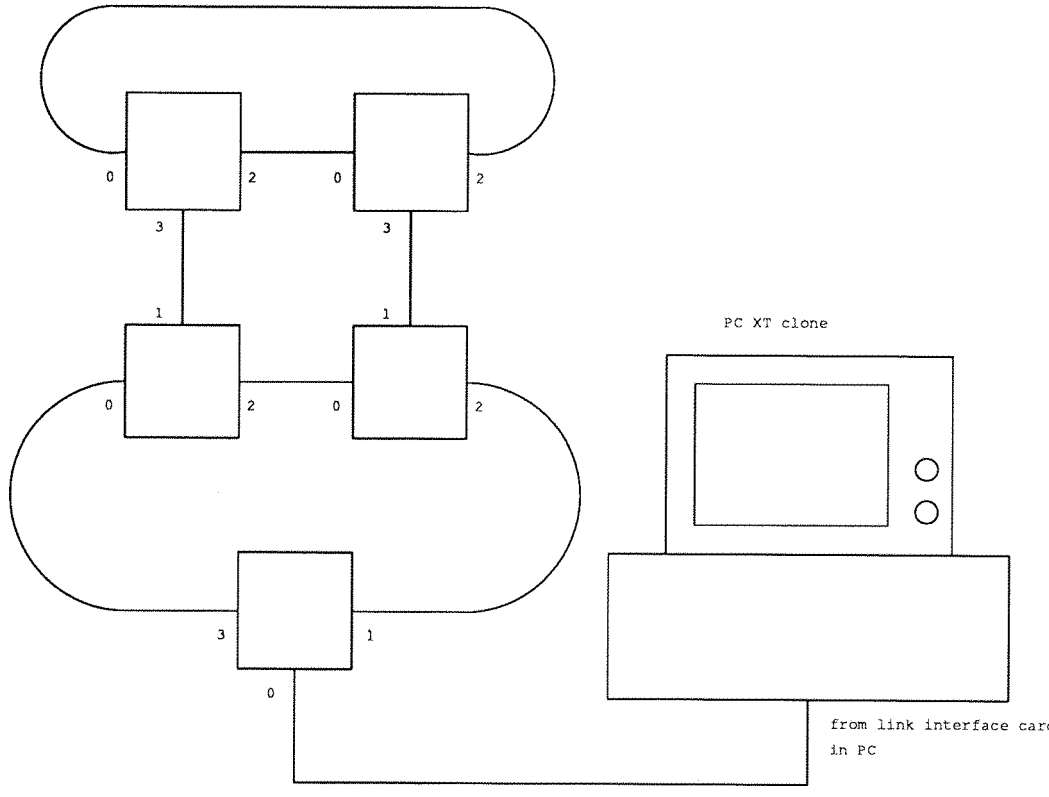

Figure 4.7: Hardware for the Express System.

Express provides both asynchronous and 'loosely' synchronous communications. Multihosting large transputer networks from a local area network is possible for the express system and there is an extensive set of performance monitoring tools provided with the package.

Express can be run on any array of transputers hosted by one or more of the following systems :

- VMS operating system.
- unix operating system.
- MS-DOS operating system.

4.5.1.1 The Coarse Farm Test

As with Par.C a single program, multiple data methodology is used with the express cubix model. The test was programmed in the loosely synchronous communications model. This means that message read and write orders must be paid attention to and that if output to the screen occurs then the message must be exactly the same from all processors. (Only one message will appear on the screen.) For multiple independent output to the screen a function call with the stream as a parameter allows every

processor to write unique messages to the screen. Processor identification numbers and other information is available via a function call to interrogate a system structure on each processor. The coarse farm test is thus easily implemented. Timings of the system were taken on each individual processor. The longest time is shown in the results assuming that the four processors used in the test overlap by one hundred per cent when running. Full results are shown in table 4.11.

No. of loops	Time/seconds
1	0.14
10	1.26
50	6.80
100	13.86

Table 4.11: Coarse Farm Test for the Express System.

4.5.1.2 The Geometric Test

This test was implemented on four processors with individual timings as in the previous section. The loosely synchronous message passing model was used. In express messages are passed to the processor identification number of the destination, all routing is handled by the express kernel. The usual format of iterations of a loop containing a one kilobyte edge swap then a loop of one million floating point adds was used on the workers. The results of the test are shown in table 4.12.

No. of loops	Time/seconds
10	55.89
100	523.24

Table 4.12: Geometric Test for the Express System.

4.5.1.3 Impressions of Express

Express provides a flexible powerful and very comprehensive toolkit for programming parallel systems. It is reasonable easy to write for and use. The documentation is extensive but slightly verbose and unclear in places. The Logical Systems C based package is very slow compiling even the simplest program. The author was using a PC XT clone but a colleague evaluating express on a system hosted by a 386 PC [34] has also noted the slowness of the system. Extensive performance analysis tools are available but these are not investigated in this thesis [34].

The debugger with the system was needed while developing the code for the tests. In the PC hosted version it provides its own windowing system with the familiar break points and single stepping features of most modern debuggers.

4.6 Operating Systems

There are two main distributed operating systems for transputers. Both are based on unix and conform to the posix standard for parallel implementations of unix. IDRIS provides a system which allows the programmer to work with 'raw' processors using INMOS and 3L development tools or by multiple C programs communicating via streams. Helios allows this too but has a high level parallel programming facility provided by the Component Distribution Language (CDL). This allows programs to be written in C and communication is achieved via streams or low level message passing. Thus you may have a number of processes on the same or different transputers running in parallel to produce a single result. This is termed a 'task force'. The usual unix pipe and a few constructs from CSP [3] are used to connect processes. It is Helios and CDL which will be investigated in this section.

4.6.1 Helios (v1.1a)

Helios is an operating system based on unix (with posix conformity) with a high level parallel programming facility called CDL.

The CDL has four parallel constructs :

- A | B provides a uni-directional pipeline.
- A<>B provides a bi-directional pipeline.
- A^^B runs A and B in parallel with no connections between the two defined at this level.
- A | | | B is a farm with master process A and worker process B.

Each component of a CDL line statement runs on a separate processor, if available. More general component descriptions may be written in a CDL file which is then compiled to give the desired system. The full version of CDL amounts to a configuration language [35].

In Helios message passing is achieved at various levels of abstraction [37]. The components of a CDL task force are connected together by streams. These may be input from or output to as either standard C streams, POSIX streams or by using the Helios low level message passing routines. The low level routines are the most difficult to use but give the least message latency [36]. The method of communication used in the geometric test was that of POSIX streams.

Each processors `stderr` stream is connected to the screen allowing diagnostic and other output from all the processors.

As Helios is supposed to provide a high level programming system full use of CDL was not made (that is there was no configuration level), but only the four one line parallel constructs above were considered.

4.6.1.1 The Coarse Farm Test

As direct access to the screen is available from all processors via the stderr stream this test was easily implemented using the C function `fprintf`. Timing occurred on individual processors as with express and each processor output its time after the test had finished. The CDL general parallel construct was used to define the task force.

The results from the test are contained in table 4.13.

No. of loops	Time/seconds
1	0.16
10	1.75
50	8.95
100	17.95

Table 4.13: Coarse Farm Test for Helios.

4.6.1.2 The Geometric Test

The geometric test task force was described using the subordinate (<>) CDL construct. This lead to the following stream connections between the task force member. (Figure 4.8)

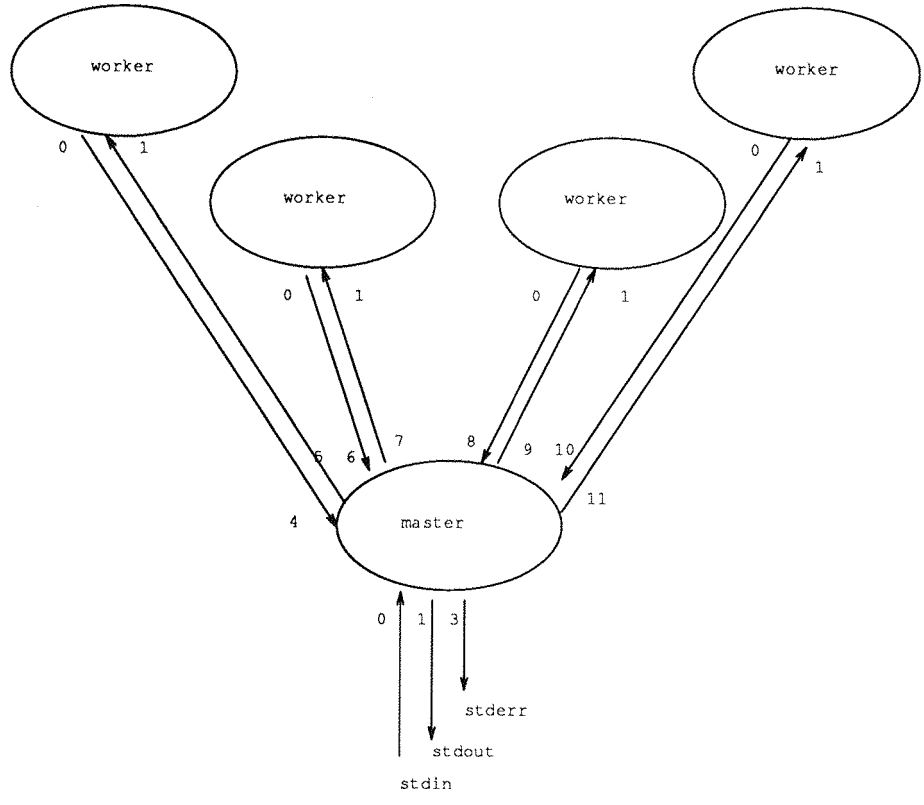

Figure 4.8: Stream Connections for the Helios Geometric Test.

As the CDL basic constructs do not allow connections between workers all the communications had to pass through the master processor. The familiar scenario of one kilobyte edge swaps and one million floating point adds was used.

The results are in table 4.14.

4.6.1.3 Impressions of Helios

Helios provides a completely enclosed unix like environment. Many tools may be run on raw processors from this environment. CDL is supplied as a high level file or

No. of loops	Time/seconds
10	329.15
100	3289.85

Table 4.14: Geometric Test for Helios.

command line based configuration language. It allows a number of C programs to be put together in a task force to implement a parallel solution to a problem. The editor supplied with the system is microemacs.

Helios message passing at the POSIX level has been proved to be very slow. This is the price paid for simple one line CDL commands to configure a parallel system. Also Helios support processes and the kernel are running on the transputer network. Helios is a full operating system of which only a small number of features were tested in this thesis.

4.7 Conclusions of the Review

A comparison of the results for the coarse farm test with one hundred loops is shown in table 4.15. The coarse farm harness result is extrapolated from the values in chapter 3. The chapter 3 results were obtained from T800-17 transputers.

System	Time/seconds
Coarse Farm (extrapolation)	11.24
Express	13.89
ECCL	14.47
VCR	14.93
Par.C	17.00
New Occam Toolkit	17.16
Helios	17.95
3L Parallel FORTRAN	66.63

Table 4.15: System Comparison for Coarse Farm Test with 100 loops.

All the systems perform reasonably well for screen output (no greater than eighteen seconds) except the 3L parallel FORTRAN. That system obviously suffers from the slowness of the old afserver used by 3L. Note how much faster the coarse farm harness is using similar compiled 3L FORTRAN code in an occam harness. The coarse farm harness comes out as the best because it is communicating directly to the server from all its processors using the server protocol.

For the geometric test there is a set of results which vary much more than the output test. The comparison for the geometric test with one hundred loops is shown in table 4.16.

As expected the occam toolset performs best. This is because occam has the closest relationship with the hardware. Message passing and calculation are very efficient when coded in occam.

System	Time/seconds
New Occam Toolkit	143.73
ECCL	164.50
Geometric Harness	361.08
VCR	451.12
3L Parallel FORTRAN	471.68
Express	523.89
Par.C	562.00
Helios	3289.85

Table 4.16: System Comparison for Geometric Test with 100 loops.

The ECCL performs at a level close to that of pure occam. ECCL is very good at nearest neighbor communications as the processors are on a loop. The output from the ECCL configurer is an occam program but it has routing and service processes built in. The price paid in performance here is well worth the flexibility and ease of programming gained.

The geometric harness (chapter 3) had the same edge swap and calculation test run on it to allow comparison with the rest of the systems investigated. This is a 3L FORTRAN based system in an occam harness and performs well showing that harnesses for specific paradigms are well worth producing. The routing strategy for the geometric harness is not ideal but the occam routing code outperforms the pure 3L system by a significant margin.

VCR provides a very convenient and usable system but a price is paid for this. The system is not optimised for nearest neighbor communication. This goes some way to explaining its dissappointing showing.

Both Par.C and 3L parallel FORTRAN require hand written communications code and deliver a disappointing performance.

Express allows random routing between processors and for this convenience the price is a performance one.

Helios pays a terrible price for all its simple CDL configuration and easy to use POSIX streams for interprocessor communications. The basic CDL construct investigated mean that messages need to be routed through the master processor but the large execution time cannot be fully accounted for by this. Helios message passing at the POSIX level must be exceptionally slow.

5 Concluding Remarks

5.1 Further Work on the Coarse Farm Harness

The coarse farm harness has been very successful both in test and on a real application (DLA). The next step to take with the code is to implement properly an interface to 3L parallel pascal and parallel C. Initial work in this direction has encountered some problems with the way these systems communicate with the screen. Screen communication is achieved on a byte by byte basis. Thus a method of buffering a message string at source is needed to stop message garbling and subsequent deadlock.

The diffusion limited aggregation application has proved very worthwhile. Calculations on the results obtained indicate that a run on a large number of processors each with a large memory would yield data sets at a rate comparable or surpassing present production technologies.

5.2 Further Work on the Geometric Harness

The geometric harness has shown good functionality and has behaved as predicted for the implementation of a test geometric application, Conway's Game of Life. The functionality of the harness in its present state is basic point to point asynchronous message passing. This could be improved to provide higher level communications subroutines. These could include message passing subroutines to pass different types of data relieving the applications programmer of the task of packing and unpacking data. Subroutines should also be provided to implement the edge swap. The message passing strategy, although adequate for smaller networks, may suffer from problems of non-optimal routing on large networks. An implementation with a better (shortest route for example) routing strategy should be produced. The ability to get file access from the worker processors is a desirable feature which could be investigated. Conway's Game of Life is a typical geometric application and other applications should be placed in the harness for evaluation.

5.3 The Need for Migration Aids

If applications programmers are to use the presently available parallel hardware they have to be provided with the software tools to do the job. The easier the tools are to use the faster the migration of code to the new technology. Parallel machines must be shown to be as easy, if not easier to use, as a sequential machine. With the limits of sequential technology being reached the only way to gain significant performance improvements is to use parallel hardware. Migration aids provide the first steps to moving to the new hardware. It should not be difficult to program a parallel machine, just slightly different.

6 Acknowledgments

I would like to acknowledge the help of all my colleagues at The University of Southampton, Transputer Technology Solutions and the Novel Architecture Research Centre at Southampton. Particular mention should go to my supervisor Prof. A.J.G. Hey, Dr. David Pritchard, Dr. Michael Surridge, the Marks (Hill and Debbage), Piers Shallow and Flavio Bergamaschi.

Thanks also to Tam.

Bibliography

[1] INMOS Limited, *IMS T800 architecture*, Technical Note 6.

[2] INMOS Limited, *Transputer Instruction Set - A compiler writer's guide*, Prentice Hall, 1988.

[3] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[4] Dick Pountain and David May, *A Tutorial Introduction to Occam Programming*, BSP PROFESSIONAL BOOKS, 1988.

[5] A. W. Roscoe and Naiem Dathi, *The Pursuit of Deadlock Freedom*, Oxford Programming Research Group monograph 57, 1986.

[6] INMOS Limited, *OCCAM 2 Toolset IBM/NEC PC*, 1989.

[7] David J. Pritchard, *Performance Analysis and Measurement on Transputer Arrays*, Proceedings "In Evaluating Supercomputers- UNICOM Applied Information Technology Reports",pp 267, Chapman and Hall.

[8] INMOS Limited, *Exploiting Concurrency: a ray tracing example*, Technical Note 7.

[9] Ian Glendinning and Anthony Hey, *Transputer Arrays as FORTRAN Farms for Particle Physics*, North-Holland Amsterdam, Computer Physics Communications 1987, 45, 367-371.

[10] C. R. Askew et al, *Monte Carlo Simulation on Transputer Arrays*, Parallel Computing, Vol. 6 (1988), p247

[11] S. A. Baker and J. G. Harp, *Parallel Processing on Transputer Arrays for the Recognition of Objects in Infra-red Images*,paper at IEE colloquium on "Transputers for Image Processing Applications", Feb. 1989, digest no. 1989/22.

[12] Charlie Askew, *Real Time Graphics with Tramsputers*, Internal Report 25, Concurrent Computation Group, University of Southampton, UK, 1987.

[13] Meiko Limited, *Running FORTRAN Programs in Parallel*, Bristol, UK, 1989.

[14] R. J. Allen and L. Heck, *Fortnet: a parallel FORTRAN harness for porting application codes to transputer arrays*, Applications of Transputers 1, ed. Len Freeman and Chris Phillips, IOS Press, 1990.

[15] L. J. Clarke, *The Tiny User-Guide*, ECS Project, Edinburgh University Computing Service.

[16] Paul Meakin, *Computer Simulations of Diffusion-Limited Aggregation Processes*, Faraday Discuss. Chem. Soc., 1987, 83, paper 9.

[17] R. C. Ball and R. M. Brady, *J. Phys. A*, 18, L809, 1985.

[18] Matin Gardiner, *Scientific American*, October 1970, p120-123.

[19] Martin Gardiner, *Scientific American*, February 1971, p112-117.

[20] INMOS Limited, *Performance Maximisation*, Technical Note 17.

[21] Link interface PC card with two TRAM sites produced in the Department of Electronics and Computer Science, University of Southampton, UK.

[22] Transputer (T800 + 4MB) 'Trice Ring' card produced in the Department of Electronics and Computer Science, University of Southampton, UK.

[23] INMOS Limited, *Occam2 Toolset User Manual - Part 1*, beta release, October 1990.

[24] INMOS Limited, *Transputer Development System*, Prentice Hall, 1988.

[25] INMOS Limited, *B007 Manual.*

[26] 3L Limited, *Parallel FORTRAN User Guide*, June 1988.

[27] Parsec Developments, *Par.C System*, November 1990.

[28] INMOS Limited, *The T9000 Transputer Products Overview Manual*, 1991.

[29] Mike Surridge, *The Eulerian Channel Configuration Language and Communications System User Guide*, Transputer Support Centre, Department of Electronics and Computer Science, Unit 2, Venture Road, Chilworth Research Centre, Southampton, UK, February 1990.

[30] L. Euler, *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 8, 128, 1736.

[31] D. A. Nicole, E. K. Lloyd, J. S. Ward, *Switching Networks for Transputer Links*, Proceedings of the 8th Technical Meeting of the Occam User Group, ed. J. Kerridge, March 1988.

[32] Mark Debbage, Mark Hill, Denis Nicole, *Virtual Channel Router 2.0 User Guide*, Esprit II PUMA project, Department of Electronics and Computer Science, University of Southampton, UK, June 1991.

[33] ParaSoft Corporation, *Express, Transputer Logical Systems C User Guide*, 1990.

[34] Piers Shallow, work in preparation for the Esprit II GENESIS project, University of Southampton, August 1991.

[35] Perihelion Software Limited, *The CDL Guide*, Helios Technical Guides, Distributed Software Limited, 1990.

[36] Flavio A. Bergamaschi, *Helios Message Passing Performance*, Concurrent Computation Group, University of Southampton, UK, 1990.

[37] Perihelion Software Limited, *The Helios Operating System*, Prentice Hall, 1989.

A Coarse Farm Harness Test Code and Data Tables

This appendix contains the coarse farm harness benchmark code and tables of all the results from the bench tests. The first seven tests vary the amount of work in the system with a constant number of processors. The last six tests keep the amount of work constant and vary the number of processors.

```
C       USER FORTRAN STARTS
        OK=.TRUE.
C       SETUP FILE NAME
        TEMP=JOBID
        DFN(1)='D'
        DFN(8)=CHAR(MOD(TEMP,10)+48)
        TEMP=TEMP/10
        DFN(7)=CHAR(MOD(TEMP,10)+48)
        TEMP=TEMP/10
        DFN(6)=CHAR(MOD(TEMP,10)+48)
        TEMP=TEMP/10
        DFN(5)=CHAR(MOD(TEMP,10)+48)
        TEMP=TEMP/10
        DFN(4)=CHAR(MOD(TEMP,10)+48)
        TEMP=TEMP/10
        DFN(3)=CHAR(MOD(TEMP,10)+48)
        TEMP=TEMP/10
        DFN(2)=CHAR(MOD(TEMP,10)+48)
        FUN=JOBID+10
C       LOAD DATA FOR TEST FILE
        DO 100 C=1,MAXINDEX
          DATA1(C)=JOBID
100     CONTINUE
C       WRITE DATA FOR FILE TEST
        OPEN(UNIT=FUN,FILE=DUFN,FORM='UNFORMATTED')
        WRITE(FUN) DATA1
        CLOSE(FUN)
C       READ DATA FOR FILE TEST
        OPEN(UNIT=FUN,FILE=DUFN,FORM='UNFORMATTED')
        READ(FUN) DATA2
        CLOSE(FUN)
C       COMPARE DATA1 AND DATA2 FOR FILE TEST
        DO 200 C=1,MAXINDEX
          IF (DATA1(C).NE.DATA2(C)) OK=.FALSE.
```

```
200     CONTINUE
C       REPORT RESULT OF FILE TEST
        IF (OK) THEN
        PRINT *,'FILE I/O OK ',PROCID,JOBID
        ELSE
        PRINT *,'FILE I/O NOT OK ',PROCID,JOBID
        END IF
C       LOOP FOR CALCULATION AND SCREEN OUTPUT
        X=0.0
        Y=1.0
        DO 10 C=1,NOLOOPS
          X=X*Y
          PRINT *,'Hello, world from processor ',PROCID,JOBID
10      CONTINUE
C       USER FORTRAN FINISH
```

No. of jobs	Time/seconds	No. of jobs	Time/seconds
0	0.02	110	3.10
10	0.30	120	3.38
20	0.58	130	3.66
30	0.86	140	3.94
40	1.14	150	4.22
50	1.42	160	4.50
60	1.70	170	4.78
70	1.98	180	5.06
80	2.26	190	5.34
90	2.54	200	5.62
100	2.82		

Table A.1: Coarse Farm Benchmark Test 1.

No. of jobs	Time/seconds
0	0.02
1	0.31
2	0.58
3	0.86
4	1.14
5	1.42
10	2.82
15	4.22
20	5.62
25	7.01
30	8.41

Table A.2: Coarse Farm Benchmark Test 2.

No. of jobs	Time/seconds	No. of jobs	Time/seconds
0	0.02	11	0.92
1	0.31	12	0.92
2	0.32	13	0.92
3	0.32	14	0.92
4	0.32	15	0.93
5	0.32	16	1.22
6	0.62		
7	0.62	50	3.05
8	0.62	100	6.07
9	0.62		
10	0.62		

Table A.3: Coarse Farm Benchmark Test 3.

No. of jobs	Time/seconds	No. of jobs	Time/seconds
0	0.02	11	9.08
1	3.03	12	9.08
2	3.04	13	9.08
3	3.04	14	9.08
4	3.04	15	9.09
5	3.04	16	12.10
6	6.06	17	12.10
7	6.06	18	12.11
8	6.06	19	12.11
9	6.06	20	12.11
10	6.06	21	15.13

Table A.4: Coarse Farm Benchmark Test 4.

No. of jobs	Time/seconds
0	0.03
1	0.64
2	1.03
3	1.44
4	1.86
5	2.10
6	2.77
7	3.13
8	3.51
9	3.97
10	4.29
15	6.65
20	9.51
30	14.55
40	19.26
50	24.33

Table A.5: Coarse Farm Benchmark Test 5.

No. of jobs	Time/seconds
0	0.03
1	1.46
2	2.19
3	3.27
4	4.20
5	5.07
6	6.58
7	7.39
8	8.24
9	9.38
10	10.14
11	11.48

Table A.6: Coarse Farm Benchmark Test 6.

No. of jobs	Time/seconds
0	0.03
1	9.19
2	14.24
3	22.07
4	30.62
5	38.96
6	47.39
7	53.61
8	59.30
9	69.92
10	77.08
11	92.00

Table A.7: Coarse Farm Benchmark Test 7.

No. of worker processors	Time/seconds
1	5.88
2	5.62
3	5.61
4	5.61
5	5.62

Table A.8: Coarse Farm Benchmark Test 8.

No. of worker processors	Time/seconds
1	5.85
2	5.61
3	5.61
4	5.62
5	5.62

Table A.9: Coarse Farm Benchmark Test 9.

No. of worker processors	Time/seconds
1	30.25
2	15.14
3	10.30
4	7.58
5	6.07

Table A.10: Coarse Farm Benchmark Test 10.

No. of worker processors	Time/seconds
1	45.35
2	24.19
3	15.13
4	12.11
5	9.09

Table A.11: Coarse Farm Benchmark Test 11.

No. of worker processors	Time/seconds
1	9.59
2	7.70
3	7.23
4	6.75
5	6.65

Table A.12: Coarse Farm Benchmark Test 12.

No. of worker processors	Time/seconds
1	14.31
2	10.99
3	10.92
4	10.39
5	10.14

Table A.13: Coarse Farm Benchmark Test 13.

B DLA Code for Coarse Farm Harness

The following code was run with great success in the Coarse Farm Harness. It is written for the 3L Parallel Fortran 2.0 compiler. The only non-standard (non FORTRAN77) features used were the bitwise addressing of data and accessing the transputers timer. Bitwise addressing allowed the lattice to be compacted so that each bit in the main data structure represents a site.

```
PROGRAM DLA

IMPLICIT NONE

INCLUDE 'TIMER.INC'
INTEGER JOBID,PROCID
C
C CONSTANTS
C
INTEGER XDIM,YDIM,MAXX,MAXY,DIM
INTEGER MAXNPARTS,BLOCKSIZE,SECONDS,MINS
PARAMETER (XDIM=7)
C 2 7 27 100
PARAMETER (YDIM=247)
C 71 247 951 3520
PARAMETER (MAXX=XDIM*32)
PARAMETER (MAXY=YDIM)
PARAMETER (DIM=2)
PARAMETER (MAXNPARTS=10000)
C 1000 10000 100000 1000000
PARAMETER (BLOCKSIZE=500)
PARAMETER (SECONDS=15625)
PARAMETER (MINS=60)
C
C ALL CHANGES TO CONSTANTS MUST BE COPIED TO
C SETBIT AND BITISSET
C
C VARIABLES
C
INTEGER F77_TIME_NOW
C 3L LIBRARY FUNCTION
INTEGER SPACE(XDIM,YDIM)
INTEGER XCOORD(BLOCKSIZE),YCOORD(BLOCKSIZE)
INTEGER I,J
```

```
INTEGER TEMP,FUN,TIME1,TIME2
CHARACTER DFN(8)
CHARACTER*8 DUFN
EQUIVALENCE (DUFN,DFN)
LOGICAL ABOVETOP
INTEGER NPARTS,TOP,CPARTS,SENDPART
LOGICAL STUCK,SET0,SET1,SET2,SET3
INTEGER DTOTOP,DIREC,DELTA
INTEGER X,Y
C
CALL GETID(PROCID)
CALL GETID(JOBID)
DO 99999 WHILE(JOBID.NE.-1)
C
C USER FORTRAN STARTS
C
C INITIALISE
C
C INIT SPACE
C
DO 100 I = 1,XDIM
  DO 200 J = 1,YDIM
    SPACE(I,J) = 0
200   CONTINUE
100 CONTINUE
C
C INIT XCOORD,YCOORD
C BUFFERS TO SEND COORDS TO FILE
C
DO 300 I = 1,BLOCKSIZE
  XCOORD(I) = 0
  YCOORD(I) = 0
300 CONTINUE
C
C SET UP FILE
C FILE IS UNFORMATTED SEQUENTIAL
C
TEMP = JOBID
DFN(1) = 'D'
DFN(8) = CHAR(MOD(TEMP,10)+48)
TEMP = TEMP / 10
DFN(7) = CHAR(MOD(TEMP,10)+48)
TEMP = TEMP / 10
DFN(6) = CHAR(MOD(TEMP,10)+48)
TEMP = TEMP / 10
DFN(5) = CHAR(MOD(TEMP,10)+48)
TEMP = TEMP / 10
DFN(4) = CHAR(MOD(TEMP,10)+48)
```

```
TEMP = TEMP / 10
DFN(3) = CHAR(MOD(TEMP,10)+48)
TEMP = TEMP / 10
DFN(2) = CHAR(MOD(TEMP,10)+48)
FUN = JOBID +10
OPEN(UNIT=FUN,FILE=DUFN,FORM='UNFORMATTED')
C
C INIT VARIABLES
C
C NUMBER OF PARTICLES TO PUT DOWN
C
NPARTS = MAXNPARTS
C
C BOOLEAN ABOVE TOP OF CLUSTER
C
ABOVETOP = .TRUE.
C
C TOP OF CLUSTER
C
TOP = 0
C
C NUMBER OF PARTICLES LAYED DOWN
C
CPARTS = 0
C
C NUMBER OF PARTICLES IN SEND BUFFERS
C
SENDPART = 0
C
C START CLOCK
C
TIME1 = F77_TIME_NOW
C
C MAIN LOOP
C
DO WHILE ((CPARTS.LT.NPARTS).AND.(TOP.LT.(YDIM-4)))
C
C PRINT MESSAGE EVERY 10000 PARTICLES
C
  IF (MOD(CPARTS,10000).EQ.0.AND.CPARTS.NE.0)
     *       WRITE (*,150) PROCID,JOBID,CPARTS
150   FORMAT ('Processor ',I2,' job ',I4,
     *               ' put down ',I8,' particles.')
C
C UPDATE COUNTS
C
  CPARTS = CPARTS + 1
  SENDPART = SENDPART + 1
```

```
C
C WRITE TO FILE IF BUFFERS FULL
C
   IF (SENDPART.EQ.BLOCKSIZE+1) THEN
     WRITE (FUN) XCOORD,YCOORD
     SENDPART = 1
   END IF
C
C INIT PARTICLE
C
   STUCK = .FALSE.
   CALL RAN(X,MAXX)
   X = X + 1
   Y = TOP +2
C
C WALK PARTICLE
C
   DO WHILE (.NOT.STUCK)
C
C GET DIRECTION AND HOW HIGH
C
     CALL RAN(DIREC,4)
     DTOTOP = Y - TOP
C
     IF (DTOTOP.GT.2) THEN
C
C LONG MOVES
C
C DO MOVE
C
       DELTA = MOD(DTOTOP,10) + 1
       IF (DIREC.EQ.0) THEN
Y = Y + DELTA
       IF (DIREC.EQ.1) THEN
Y = Y - DELTA
       IF (DIREC.EQ.2) THEN
X = X + DELTA
X = MOD(X,MAXX)
       IF (DIREC.EQ.4) THEN
X = X - DELTA
X = MOD((X+MAXX),MAXX)
IF (X.EQ.0) X = MAXX
       END IF
C
C RESET IF TOO HIGH
C
       IF (Y.GT.(TOP+100)) THEN
Y = TOP + 2
```

```
CALL RAN(X,MAXX)
X = X + 1
      END IF
C
    ELSE
C
C SHORT MOVES
C
C DO MOVE
C
      DELTA = 1
      IF (DIREC.EQ.0) THEN
Y = Y + DELTA
      IF (DIREC.EQ.1) THEN
Y = Y - DELTA
      IF (DIREC.EQ.2) THEN
X = X + DELTA
X = MOD(X,MAXX)
      IF (DIREC.EQ.4) THEN
X = X - DELTA
X = MOD((X+MAXX),MAXX)
IF (X.EQ.0) X = MAXX
      END IF
C
C STUCK ??
C
      IF (Y.EQ.0) THEN
STUCK = .TRUE.
      ELSE
CALL BITISSET(SPACE,X,Y+1,SET0)
CALL BITISSET(SPACE,X,Y-1,SET1)
IF (MOD(X+1,MAXX).EQ.0) THEN
  CALL BITISSET(SPACE,MAXX,Y,SET2)
ELSE
  CALL BITISSET(SPACE,(MOD(X+1,MAXX)),Y,SET2)
END IF
IF (MOD(X-1+MAXX,MAXX).EQ.0) THEN
  CALL BITISSET(SPACE,MAXX,Y,SET3)
ELSE
  CALL BITISSET(SPACE,(MOD((X-1+MAXX),MAXX),Y,SET3)
END IF
STUCK = SET0.OR.SET1.OR.SET2.OR.SET3
      END IF
C
C SET BIT IF STUCK AND PUT COORDS IN BUFFERS
C
      IF (STUCK) THEN
CALL SETBIT(SPACE,X,Y)
```

```
XCOORD(SENDPART) = X
YCOORD(SENDPART) = Y
IF (Y.GT.TOP) TOP = Y
      END IF
C
    END IF
C
  END DO
C
END DO
C
C END OF MAIN LOOP
C
C SEND ANY PARTICLES LEFT AND FINISH MESSAGE
C
IF (SENDPART.GT.0) THEN
  WRITE (FUN) XCOORD,YCOORD
END IF
C
TIME2 = F77_TIME_NOW
TIME2 = ABS(TIME2 - TIME1) /SECONDS
TIME1 = TIME2 / MINS
TIME2 = TIME2 - TIME1 * MINS
WRITE (*,250) PROCID,JOBID,CPARTS
250 FORMAT ('Processor ',I2,' job ',I3,
     *            ' put down ',I8,' particles total.')
WRITE (*,350) JOBID,TIME1,TIME2
350 FORMAT ('Job ',I3,I5,' minutes ',I3,' seconds.')
C
C USER FORTRAN FINISH
C
CALL ENDJOB()
CALL GETID(JOBID)
99999 CONTINUE
CALL ENDWORK()
END
```

```
      SUBROUTINE SETBIT(SPACE,X,Y)

      IMPLICIT NONE
C
C CONSTANTS
C
      INTEGER XDIM,YDIM,MAXX,MAXY,DIM
      INTEGER MAXNPARTS,BLOCKSIZE,SECONDS,MINS
      PARAMETER (XDIM=7)
C 2 7 27 100
      PARAMETER (YDIM=247)
C 71 247 951 3520
      PARAMETER (MAXX=XDIM*32)
      PARAMETER (MAXY=YDIM)
      PARAMETER (DIM=2)
      PARAMETER (MAXNPARTS=10000)
C 1000 10000 100000 1000000
      PARAMETER (BLOCKSIZE=500)
      PARAMETER (SECONDS=15625)
      PARAMETER (MINS=60)
C
C VARIABLES
C
      INTEGER X,Y,MAJX,BIT
      INTEGER SPACE(XDIM,YDIM)
C
C BODY
C
      BIT = MOD(X-1,32)
      MAJX = ((X-1)/32) + 1
      SPACE(MAJX,Y) = IBSET(SPACE(MAJX,Y),BIT)
      RETURN
      END
```

```
      SUBROUTINE BITISSET(SPACE,X,Y,BSET)

      IMPLICIT NONE
C
C CONSTANTS
C
      INTEGER XDIM,YDIM,MAXX,MAXY,DIM
      INTEGER MAXNPARTS,BLOCKSIZE,SECONDS,MINS
      PARAMETER (XDIM=7)
C 2 7 27 100
      PARAMETER (YDIM=247)
C 71 247 951 3520
      PARAMETER (MAXX=XDIM*32)
      PARAMETER (MAXY=YDIM)
      PARAMETER (DIM=2)
      PARAMETER (MAXNPARTS=10000)
C 1000 10000 100000 1000000
      PARAMETER (BLOCKSIZE=500)
      PARAMETER (SECONDS=15625)
      PARAMETER (MINS=60)
C
C VARIABLES
C
      INTEGER X,Y,MAJX,BIT
      INTEGER SPACE(XDIM,YDIM)
      LOGICAL BSET
C
C BODY
C
      BIT = MOD(X-1,32)
      MAJX = ((X-1)/32) + 1
      BSET = BTEST(SPACE(MAJX,Y),BIT)
      RETURN
      END
```

```
      SUBROUTINE RAN(NUM,NUMRAN)
      C
      C Hack of ran2 from Numerical Recipes,
      C W.H.Press et al p197 Cambridge Uni. Press.
      C
      IMPLICIT NONE
      C
      C VARIABLES
      C
      INTEGER RAND,NUMRAN
      INTEGER M,IA,IC,IFF,IR(97),J,IY,IDUM
      REAL RM,RRAND
      PARAMETER (M=714025,IA=1366,IC=150889,RM=1./M)
      SAVE IDUM
      DATA IFF /0/
      C
      C INITIALISE
      C
      IF (IFF.EQ.0) THEN
        IFF = 1
      C     PRINT *,'Enter random number seed.'
        READ *,IDUM
        IDUM = MOD(IC-IDUM,M)
        DO 11 J = 1,97
          IDUM = MOD(IA*IDUM+IC,M)
          IR(J) = IDUM
      11    CONTINUE
        IDUM = MOD(IA*IDUM+IC,M)
        IY = IDUM
      END IF
      C
      C GENERATE
      C
      J = 1 + ( 97*IY) /M
      IF (J.GT.97.OR.J.LT.1) J = MOD(J,97) + 1
      IY = IR(J)
      RRAND = IY * RM
      RAND = MOD(IMT(REAL(NUMRAN)*RRAND),NUMRAN)
      IDUM = MOD(IA*IDUM+IC,M)
      IR(J) = IDUM
      C
      RETURN
      END
```

C Geometric Harness Benchmark Code.

This appendix contains code listings of the master, worker and library FORTRAN used to test the functionallity of the geometric harness. Non-standard FORTRAN was used to access the transputers timer in the master code. Both pieces of code fit their respective templates shown in section 3.3.3.

The FMASTER code, full i/o is available from this processor.

```
      PROGRAM FMASTER

      IMPLICIT NONE
      INCLUDE 'TIMER.INC'
C
      INTEGER TICKS
      PARAMETER (TICKS = 15625)
      INTEGER X,Y,XDIM,YDIM
      INTEGER SX,SY,DX,DY,TAG,LEN,BUFFER(128)
      INTEGER I,J,T1,T2
      REAL RTIME
C
      T1 = F77_TIMER_NOW()
      CALL INIT(X,Y,XDIM,YDIM)
C
C     *** ALL TO ONE TEST
C      DO 999 J=1,10000
C        DO 99 I =1,(XDIM*YDIM)
C          CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C 99     CONTINUE
C 999   CONTINUE
C
C
C     *** ONE TO ALL TEST
C      DO 999 J=1,1000
C        CALL SENDMESS(1,1,5,128,BUFFER)
C        CALL SENDMESS(1,2,5,128,BUFFER)
C        CALL SENDMESS(2,1,5,128,BUFFER)
C        CALL SENDMESS(2,2,5,128,BUFFER)
C 999   CONTINUE
C
C     *** ALL TO ALL TEST
C      DO 99 I=1,1000
C        CALL SENDMESS(0,1,5,128,BUFFER)
C        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
```

```
C         CALL SENDMESS(1,1,5,128,BUFFER)
C         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C         CALL SENDMESS(2,1,5,128,BUFFER)
C         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C         CALL SENDMESS(1,2,5,128,BUFFER)
C         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C         CALL SENDMESS(2,2,5,128,BUFFER)
C         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C 99    CONTINUE
C       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C
C      *** EDGE SWAP AND CALC TEST
       DO 99 I=1,100
         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
         CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
 99    CONTINUE
       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
       CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C
       CALL SHUTDOWN(XDIM,YDIM)
       T2=F77_TIMER_NOW()
       T1=T2-T1
       RTIME = REAL(T1)/REAL(TICKS)
       PRINT *,RTIME

       END
```

The FWORKER code, this has no i/o.

```
      PROGRAM FWORKER

      IMPLICIT NONE
C
      INTEGER X,Y,XDIM,YDIM
      INTEGER SX,SY,DX,DY,TAG,LEN,BUFFER(128)
      INTEGER I,J
      INTEGER B1(128),B2(128),B3(128),B4(128)
      INTEGER B5(128),B6(128),B7(128),B8(128)
      REAL RNUM
C
      CALL INIT(X,Y,XDIM,YDIM)
C
C     *** ALL TO ONE TEST
C      DO 99 I=1,10000
C        CALL SENDMESS(0,1,5,128,BUFFER)
C 99   CONTINUE
C
C     *** ONE TO ALL TEST
C      DO 99 I=1,1000
C        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C 99   CONTINUE
C
C     *** ALL TO ALL TEST
C      DO 99 I=1,1000
C        CALL SENDMESS(0,1,5,128,BUFFER)
C        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C        CALL SENDMESS(1,1,5,128,BUFFER)
C        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C        CALL SENDMESS(2,1,5,128,BUFFER)
C        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C        CALL SENDMESS(1,2,5,128,BUFFER)
C        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C        CALL SENDMESS(2,2,5,128,BUFFER)
C        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C 99   CONTINUE
C      CALL SENDMESS(0,1,5,128,BUFFER)
C
C
C     *** EDGE SWAP AND CALC TEST
C
      RNUM = 0.0
      DO 99 I = 1,100
        DX = X-1
        IF (DX.EQ.0) DX=XDIM
```

```
          CALL SENDMESS(DX,Y,1,128,B1)
          CALL GETMESS(SX,SY,TAG,LEN,B5)
          DY = Y+1
          IF (DY.GT.YDIM) DY=1
          CALL SENDMESS(X,DY,2,128,B2)
          CALL GETMESS(SX,SY,TAG,LEN,B6)
          DX = X+1
          IF (DX.GT.XDIM) DX=1
          CALL SENDMESS(DX,Y,3,128,B3)
          CALL GETMESS(SX,SY,TAG,LEN,B7)
          DY = Y-1
          IF (DY.EQ.0) DY=YDIM
          CALL SENDMESS(X,DY,4,128,B4)
          CALL GETMESS(SX,SY,TAG,LEN,B4)
          DO 9 J=1,1000000
            RNUM = RNUM + 1.0
 9        CONTINUE
          CALL SENDMESS(0,1,5,128,BUFFER)
 99     CONTINUE
        CALL SENDMESS(0,1,5,128,BUFFER)
C
        CALL ENDWORK()

        END
```

The GEOFLIB code. This code library contains the support subroutines for the geometric harness.

```
      SUBROUTINE INIT (X,Y,XDIM,YDIM)

      INTEGER X,Y,XDIM,YDIM

      CALL CHANINMESSAGE (3,X,4)
      CALL CHANINMESSAGE (3,Y,4)
      CALL CHANINMESSAGE (3,XDIM,4)
      CALL CHANINMESSAGE (3,YDIM,4)

      END


      SUBROUTINE SENDMESS (DX,DY,TAG,LEN,BUFFER)

      INTEGER DX,DY,TAG,LEN,BUFFER(*)

      CALL CHANOUTMESSAGE (2,DX,4)
      CALL CHANOUTMESSAGE (2,DY,4)
      CALL CHANOUTMESSAGE (2,TAG,4)
      CALL CHANOUTMESSAGE (2,LEN,4)
      CALL CHANOUTMESSAGE (2,BUFFER,LEN*4)

      END


      SUBROUTINE GETMESS (SX,SY,TAG,LEN,BUFFER)

      INTEGER SX,SY,TAG,LEN,BUFFER(*)

      CALL CHANINMESSAGE (2,SX,4)
      CALL CHANINMESSAGE (2,SY,4)
      CALL CHANINMESSAGE (2,TAG,4)
      CALL CHANINMESSAGE (2,LEN,4)
      CALL CHANINMESSAGE (2,BUFFER,LEN*4)

      END
```

```
      SUBROUTINE ENDWORK ()

      INTEGER SX,SY,TAG,LEN,BUFFER

      CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
      IF (TAG.NE.-1) TAG=TAG/0
      CALL CHANOUTMESSAGE (3,TAG,4)

      END



      SUBROUTINE SHUTDOWN (XDIM,YDIM)

      INTEGER XDIM,YDIM,BUFFER,I,J

      DO 100 J=YDIM,1,-1
        DO 10 I=XDIM,1,-1
          CALL SENDMESS(I,J,-1,1,BUFFER)
10      CONTINUE
100   CONTINUE
      CALL CHANOUTMESSAGE (3,-1,4)

      END
```

D Geometric Harness Life Code.

The code contained here is the FORTRAN 77 source for the FMASTER and FWORKER user processes for the game of life implemented in the geometric harness described in chapter 3.

```
      PROGRAM FMASTER

      IMPLICIT NONE
      INCLUDE 'TIMER.INC'
      INCLUDE 'CGA.INC'
C
      INTEGER TICKS
      PARAMETER (TICKS = 15625)
      INTEGER X,Y,XDIM,YDIM
      INTEGER SX,SY,DX,DY,TAG,LEN,BUFFER(128)
      INTEGER LIFE(202,202)
      INTEGER I,J,K,T1,T2,STARTX,STARTY,LOOP
      REAL RTIME
C
      T1 = F77_TIMER_NOW()
      CALL INIT(X,Y,XDIM,YDIM)
C
C     *** LIFE MASTER
C
      CALL VIDEO_MODE(MONO_80COL_TEXT_MODE)
      PRINT *,'Geometric Harness Life'
      PRINT *,'Version 1.0'
      PRINT *,'22/8/91'
      DO 100 I=1,202
        DO 10 J=1,202
          LIFE(I,J)=0
 10     CONTINUE
 100  CONTINUE
C
C     PLOT FIRST
C
      DO 501 I=1,400
        CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
        STARTY = ((SY-1)*100)+1
        K=1
        DO 51 J = STARTY,STARTY+100
```

```
            LIFE(TAG,J)=BUFFER(K)
            K=K+1
 51       CONTINUE
 501    CONTINUE
        CALL VIDEO_MODE(CGA_LORES_GRAPHICS_MODE)
        DO 500 I = 2,201
          DO 50 J = 2,201
            CALL CGA_LORES_PLOT (I+90,J,LIFE(I,J))
 50       CONTINUE
 500    CONTINUE
        CALL CGA_UPDATE
C
        DO 9999 LOOP=1,100
C
C       PLOT REST
C
        DO 601 I=1,400
          CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
          STARTY = ((SY-1)*100)+1
          K=1
          DO 61 J = STARTY,STARTY+100
            LIFE(TAG,J)=BUFFER(K)
            K=K+1
 61       CONTINUE
 601    CONTINUE
        CALL VIDEO_MODE(CGA_LORES_GRAPHICS_MODE)
        DO 600 I = 2,201
          DO 60 J = 2,201
            CALL CGA_LORES_PLOT (I+90,J,LIFE(I,J))
 60       CONTINUE
 600    CONTINUE
        CALL CGA_UPDATE
C
 9999 CONTINUE
C
C       END OF WORK SYNCHRONIZE
C
        DO 999 I=1,4
          CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
 999    CONTINUE
C
C       RECEIVE FOR NO OUTPUT 80
C
C        DO 801 I=1,400
C          CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
C          STARTY = ((SY-1)*100)+1
C          K=1
C          DO 81 J = STARTY,STARTY+100
```

```
C           LIFE(TAG,J)=BUFFER(K)
C           K=K+1
C 81      CONTINUE
C 801   CONTINUE
C
      CALL SHUTDOWN(XDIM,YDIM)
C
C     FINISH UP
C
      T2=F77_TIMER_NOW()
      T1=T2-T1
      RTIME = REAL(T1)/REAL(TICKS)
      PRINT *,RTIME
      PRINT *,'Hit a digit'
      READ *,K
      CALL VIDEO_MODE(MONO_80COL_TEXT_MODE)
      PRINT *,'Thankyou and goodnight'
C
      END
```

```
      PROGRAM FWORKER

      IMPLICIT NONE
C
      INTEGER X,Y,XDIM,YDIM
      INTEGER SX,SY,DX,DY,TAG,LEN,BUFFER(128)
      INTEGER I,J,LOOP,COUNT
      INTEGER OLD(102,102),NEW(102,102)
      REAL RNUM
C
      CALL INIT(X,Y,XDIM,YDIM)
C
C     *** LIFE WORKER
C
C     INIT
C
      DO 100 I=1,102
        DO 10 J=1,102
          OLD(I,J)=0
          NEW(I,J)=0
 10     CONTINUE
 100  CONTINUE
      DO 200 I=2,101,10
        DO 20 J=2,101,1
          OLD(I,J)=3
 20     CONTINUE
 200  CONTINUE
      DO 201 I=2,101,1
        DO 21 J=2,101,10
          OLD(I,J)=3
 21     CONTINUE
 201  CONTINUE
C
C     PLOT FIRST
C
      DO 500 I = 2,101
        DO 50 J = 2,101
          BUFFER(J-1)=OLD(I,J)
 50     CONTINUE
        TAG=(I+((X-1)*100))
        LEN=100
        CALL SENDMESS(0,1,TAG,LEN,BUFFER)
 500  CONTINUE
C
      DO 99 LOOP = 1,100
C
C     EDGE SWAP
C
```

```
      DX = X-1
      IF (DX.EQ.0) DX=XDIM
      DO 71 I =1,100
        BUFFER(I)=OLD(2,I+1)
 71   CONTINUE
      CALL SENDMESS(DX,Y,1,100,BUFFER)
      CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
      DO 72 I =1,100
        OLD(102,I+1)=BUFFER(I)
 72   CONTINUE
      DX = X+1
      IF (DX.GT.XDIM) DX=1
      DO 73 I =1,100
        BUFFER(I)=OLD(101,I+1)
 73   CONTINUE
      CALL SENDMESS(DX,Y,1,100,BUFFER)
      CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
      DO 74 I =1,100
        OLD(1,I+1)=BUFFER(I)
 74   CONTINUE
      DY = Y+1
      IF (DY.GT.YDIM) DY=1
      DO 75 I =1,101
        BUFFER(I)=OLD(I,101)
 75   CONTINUE
      CALL SENDMESS(DX,Y,1,101,BUFFER)
      CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
      DO 76 I =1,102
        OLD(I,1)=BUFFER(I)
 76   CONTINUE
      DY = Y-1
      IF (DY.EQ.0) DY=YDIM
      DO 77 I =1,102
        BUFFER(I)=OLD(I,2)
 77   CONTINUE
      CALL SENDMESS(DX,Y,1,102,BUFFER)
      CALL GETMESS(SX,SY,TAG,LEN,BUFFER)
      DO 78 I =1,102
        OLD(I,102)=BUFFER(I)
 78   CONTINUE
C
C     UPDATE
C
      DO 300 I=2,101
        DO 30 J=2,101
          COUNT = 0
          IF (OLD(I-1,J-1).EQ.3) COUNT = COUNT +1
          IF (OLD(I-1,J).EQ.3) COUNT = COUNT +1
```

```
            IF (OLD(I-1,J+1).EQ.3) COUNT = COUNT +1
            IF (OLD(I,J-1).EQ.3) COUNT = COUNT +1
            IF (OLD(I,J+1).EQ.3) COUNT = COUNT +1
            IF (OLD(I+1,J-1).EQ.3) COUNT = COUNT +1
            IF (OLD(I+1,J).EQ.3) COUNT = COUNT +1
            IF (OLD(I+1,J+1).EQ.3) COUNT = COUNT +1
            IF ((COUNT.LT.2).OR.(COUNT.GT.3)) NEW(I,J)=0
            IF (COUNT.EQ.3) NEW(I,J)=3
            IF (COUNT.EQ.2) NEW(I,J)=OLD(I,J)
 30       CONTINUE
 300    CONTINUE
        DO 400 I=2,101
          DO 40 J=2,101
            OLD(I,J)=NEW(I,J)
 40       CONTINUE
 400    CONTINUE
C
C       PLOT
C
        DO 600 I = 2,101
          DO 60 J = 2,101
            BUFFER(J-1)=NEW(I,J)
 60       CONTINUE
          TAG=(I+((X-1)*100))
          LEN=100
          CALL SENDMESS(0,1,TAG,LEN,BUFFER)
 600    CONTINUE
C
 99     CONTINUE
C
C   SEND FOR NO OUTPUT
C
C        DO 800 I = 2,101
C          DO 80 J = 2,101
C            BUFFER(J-1)=NEW(I,J)
C 80       CONTINUE
C          TAG=(I+((X-1)*100))
C          LEN=100
C          CALL SENDMESS(0,1,TAG,LEN,BUFFER)
C 800    CONTINUE
C
C       SEND A SIGNAL TO MASTER TO INDICATE END OF WORK
C
        CALL SENDMESS(0,1,5,1,BUFFER)
C
        CALL ENDWORK()
        END
```

E Migration Aids Review Code.

A complete set of the source codes used in chapter 4 is contained here. It was thought that it would be useful to provide working code examples for each of the systems used in the review.

E.1 The New Occam Toolkit.

E.1.1 The Coarse Farm Test.

The coarse farm test was implemented by passing the byte array forming the message and the processor identification number down the chain of processors as there is no direct screen access from each transputer in the new occam toolset system.

The configuration file.

```
VAL K IS 1024 :
VAL M IS K*K :

NODE rootp,worker1,worker2,worker3,worker4 :
ARC hostlink :
NETWORK simple.network
  DO
    SET rootp (type, memsize := "T800", 4*M)
    SET worker1 (type, memsize := "T800", 4*M)
    SET worker2 (type, memsize := "T800", 4*M)
    SET worker3 (type, memsize := "T800", 4*M)
    SET worker4 (type, memsize := "T800", 4*M)
    CONNECT rootp[link][0] TO HOST WITH hostlink
    CONNECT rootp[link][3] TO worker1[link][0]
    CONNECT worker1[link][2] TO worker2[link][0]
    CONNECT worker2[link][3] TO worker3[link][1]
    CONNECT worker3[link][2] TO worker4[link][0]
:

NODE root,worker.1,worker.2,worker.3,worker.4 :
MAPPING
  DO
    MAP root ONTO rootp
    MAP worker.1 ONTO worker1
    MAP worker.2 ONTO worker2
    MAP worker.3 ONTO worker3
    MAP worker.4 ONTO worker4
```

```
:

#INCLUDE "hostio.inc"
#USE "root.c8h"
#USE "worker.c8h"
CONFIG
  CHAN OF SP fs,ts :
  PLACE fs,ts ON hostlink :
  [4]CHAN OF ANY downchain :
  PLACED PAR
    PROCESSOR root
      root.proc(fs,ts,downchain[0],0,4)
    PROCESSOR worker.1
      worker.proc(downchain[0],downchain[1],1,4)
    PROCESSOR worker.2
      worker.proc(downchain[1],downchain[2],2,4)
    PROCESSOR worker.3
      worker.proc(downchain[2],downchain[3],3,4)
    CHAN OF ANY dummy :
    PROCESSOR worker.4
      worker.proc(downchain[3],dummy,4,4)
:
```

The root occam code for the coarse farm test with the new occam toolkit.

```
#INCLUDE "hostio.inc"
PROC root.proc (CHAN OF SP fs,ts,CHAN OF ANY downchain,
                        VAL INT id,wnum)
  #USE "hostio.lib"

  VAL REAL32 ticks IS 15625.0(REAL32) :
  TIMER clock :
  INT x :
  INT t1,t2 :
  REAL32 rtime :
  [27]BYTE buffer :
  SEQ
    clock ? t1
    SEQ j=0 FOR 100
      SEQ i=0 FOR wnum
        SEQ
          downchain ? buffer
          downchain ? x
          so.write.string(fs,ts,buffer)
          so.write.int(fs,ts,x,0)
          so.write.string(fs,ts,"*c*n")
    clock ? t2
    t1 := t2-t1
    rtime := (REAL32 ROUND(t1))/ticks
    so.write.real32(fs,ts,rtime,0,0)
    so.exit(fs,ts,sps.success)
:
```

The worker code for the coarse farm test with the new occam toolkit.

```
#INCLUDE "hostio.inc"
PROC worker.proc (CHAN OF ANY outdownchain,indownchain,
                  VAL INT id,wnum)

  VAL message IS "Hello world from processor " :
  [27]BYTE buffer :
  INT x :
  SEQ j=0 FOR 100
    SEQ
      outdownchain ! message
      outdownchain ! id
      SEQ i=0 FOR (wnum-id)
        SEQ
          indownchain ? buffer
          indownchain ? x
          outdownchain ! buffer
          outdownchain ! x
:
```

E.1.2 The Geometric Test.

The geometric test configuration file.

```
VAL K IS 1024 :
VAL M IS K*K :

NODE rootp,worker1,worker2,worker3,worker4 :
ARC hostlink :
NETWORK simple.network
  DO
    SET rootp (type, memsize := "T800", 4*M)
    SET worker1 (type, memsize := "T800", 4*M)
    SET worker2 (type, memsize := "T800", 4*M)
    SET worker3 (type, memsize := "T800", 4*M)
    SET worker4 (type, memsize := "T800", 4*M)
    CONNECT rootp[link][0] TO HOST WITH hostlink
    CONNECT rootp[link][3] TO worker1[link][0]
    CONNECT rootp[link][1] TO worker2[link][2]
    CONNECT worker1[link][1] TO worker3[link][3]
    CONNECT worker1[link][2] TO worker2[link][0]
    CONNECT worker1[link][3] TO worker3[link][1]
    CONNECT worker2[link][1] TO worker4[link][3]
    CONNECT worker2[link][3] TO worker4[link][1]
    CONNECT worker3[link][0] TO worker4[link][2]
    CONNECT worker3[link][2] TO worker4[link][0]
:

NODE root,worker.1,worker.2,worker.3,worker.4 :
MAPPING
  DO
    MAP root ONTO rootp
    MAP worker.1 ONTO worker1
    MAP worker.2 ONTO worker2
    MAP worker.3 ONTO worker3
    MAP worker.4 ONTO worker4
:

#INCLUDE "hostio.inc"
#USE "groot.c8h"
#USE "gworker.c8h"
CONFIG
  CHAN OF SP fs,ts :
  CHAN OF ANY cc310,c10c3,c1133,c3311 :
  CHAN OF ANY c1220,c2012,c1331,c3113 :
  CHAN OF ANY c2143,c4321,c22c1,cc122 :
  CHAN OF ANY c2341,c4123,c3042,c4230 :
  CHAN OF ANY c3240,c4032 :
```

```
  PLACE fs,ts ON hostlink :
  PLACED PAR
    PROCESSOR root
      root.proc(fs,ts,c22c1,cc122,c10c3,cc310,0,4)
    PROCESSOR worker.1
      worker.proc(cc310,c10c3,c3311,c1133,
                  c2012,c1220,c3113,c1331,1,4)
    PROCESSOR worker.2
      worker.proc(c1220,c2012,c4321,c2143,
                  cc122,c22c1,c4123,c2341,2,4)
    PROCESSOR worker.3
      worker.proc(c4230,c3042,c1331,c3113,
                  c4032,c3240,c1133,c3311,3,4)
    PROCESSOR worker.4
      worker.proc(c3240,c4032,c2341,c4123,
                  c3042,c4230,c2143,c4321,4,4)
:
```

The root processor occam for the new occam toolset geometric test.

```
#INCLUDE "hostio.inc"
PROC root.proc (CHAN OF SP fs,ts,CHAN OF ANY l1in,l1out,
                    l3in,l3out,VAL INT id,wnum)
  #USE "hostio.lib"

  VAL REAL32 ticks IS 15625.0(REAL32) :
  TIMER clock :
  [128]INT x1,x3 :
  INT t1,t2 :
  REAL32 rtime :
  [27]BYTE buffer :
  SEQ
    clock ? t1
    SEQ i=0 FOR 10
      PAR
        SEQ
          l1in ? x1
          l3out ! x1
        SEQ
          l3in ? x3
          l1out ! x3
          so.write.string.nl(fs,ts,
                        "Hello world from processor")
    clock ? t2
    t1 := t2-t1
    rtime := (REAL32 ROUND(t1))/ticks
    so.write.real32(fs,ts,rtime,0,0)
    so.exit(fs,ts,sps.success)
:
```

The worker processor occam for the new occam toolset geometric test.

```
#INCLUDE "hostio.inc"
PROC worker.proc (CHAN OF ANY l0in,l0out,l1in,l1out,
                              l2in,l2out,l3in,l3out,
                        VAL INT id,wnum)

  VAL message IS "Hello world from processor " :
  [27]BYTE buffer :
  [128]INT x0,x1,x2,x3,x4,x5,x6,x7 :
  REAL32 rnum :
  SEQ i=0 FOR 10
    SEQ
      PAR
        l0in ? x4
        l1in ? x5
        l2in ? x6
        l3in ? x7
        l0out ! x0
        l1out ! x1
        l2out ! x2
        l3out ! x3
      rnum := 0.0(REAL32)
      SEQ j=0 FOR 1000000
        rnum := rnum + 1.0(REAL32)
:
```

E.2 3L Parallel FORTRAN.

E.2.1 The Coarse Farm Test.

The 3L configuration level allows not passing of parameters thus a piece of code was written for each processor in the system. Like the new occam toolkit 3L parallel fortran does not give access to the screen to all processors so the componant parts of the message to be output were passed in the system.

The configration level for the 3L parallel fortran coarse farm test.

```
processor host
processor zero
processor one
processor two
processor three
processor four

wire jumper host[0] zero[0]
wire ? zero[3] one[0]
wire ? one[2] two[0]
wire ? two[1] three[3]
wire ? three[2] four[0]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10k
task p0 ins=3 outs=3
task p1 ins=3 outs=3
task p2 ins=3 outs=3
task p3 ins=3 outs=3
task p4 ins=2 outs=2

place afserver host
place filter zero
place p0 zero
place p1 one
place p2 two
place p3 three
place p4 four

connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? p0[1] filter[1]
connect ? filter[1] p0[1]
connect ? p0[2] p1[1]
connect ? p1[1] p0[2]
connect ? p1[2] p2[1]
connect ? p2[1] p1[2]
connect ? p2[2] p3[1]
```

```
connect ? p3[1] p2[2]
connect ? p3[2] p4[1]
connect ? p4[1] p3[2]
```

The FORTRAN 77 code for the chain of processors 0..4 is shown below.

```
      PROGRAM ZERO

      IMPLICIT NONE

      INCLUDE 'TIMER.INC'

      INTEGER PROCID,TICKS
      PARAMETER (PROCID = 0)
      PARAMETER (TICKS = 15625)
      CHARACTER MESS(27)
      INTEGER PID,I,T1,T2
      REAL RTIME

      T1=F77_TIMER_NOW()
      DO 99  I=1,100
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        PRINT *,MESS,PID
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        PRINT *,MESS,PID
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        PRINT *,MESS,PID
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        PRINT *,MESS,PID
99    CONTINUE
      T2=F77_TIMER_NOW()
      T1=T2-T1
      RTIME = REAL(T1)/REAL(TICKS)
      PRINT *,RTIME

      END
```

```
      PROGRAM ONE

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 1)
      CHARACTER MESS(27)
      INTEGER PID,I

      DO 99 I=1,100
        CALL CHANOUTMESSAGE(1,
     *           'HELLO WORLD FROM PROCESSOR ',27)
        CALL CHANOUTMESSAGE(1,PROCID,4)
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        CALL CHANOUTMESSAGE(1,MESS,27)
        CALL CHANOUTMESSAGE(1,PID,4)
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        CALL CHANOUTMESSAGE(1,MESS,27)
        CALL CHANOUTMESSAGE(1,PID,4)
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        CALL CHANOUTMESSAGE(1,MESS,27)
        CALL CHANOUTMESSAGE(1,PID,4)
99    CONTINUE

      END
```

```
      PROGRAM TWO

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 2)
      CHARACTER MESS(27)
      INTEGER PID,I

      DO 99 I=1,100
      CALL CHANOUTMESSAGE(1,
     *          'HELLO WORLD FROM PROCESSOR ',27)
      CALL CHANOUTMESSAGE(1,PROCID,4)
      CALL CHANINMESSAGE(2,MESS,27)
      CALL CHANINMESSAGE(2,PID,4)
      CALL CHANOUTMESSAGE(1,MESS,27)
      CALL CHANOUTMESSAGE(1,PID,4)
      CALL CHANINMESSAGE(2,MESS,27)
      CALL CHANINMESSAGE(2,PID,4)
      CALL CHANOUTMESSAGE(1,MESS,27)
      CALL CHANOUTMESSAGE(1,PID,4)
99    CONTINUE

      END
```

```
      PROGRAM THREE

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 3)
      CHARACTER MESS(27)
      INTEGER PID,I

      DO 99 I=1,100
        CALL CHANOUTMESSAGE(1,
     *          'HELLO WORLD FROM PROCESSOR ',27)
        CALL CHANOUTMESSAGE(1,PROCID,4)
        CALL CHANINMESSAGE(2,MESS,27)
        CALL CHANINMESSAGE(2,PID,4)
        CALL CHANOUTMESSAGE(1,MESS,27)
        CALL CHANOUTMESSAGE(1,PID,4)
99    CONTINUE

      END
```

```
      PROGRAM FOUR

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 4)
      INTEGER I

      DO 99 I=1,100
        CALL CHANOUTMESSAGE(1,
     *           'HELLO WORLD FROM PROCESSOR ',27)
        CALL CHANOUTMESSAGE(1,PROCID,4)
99    CONTINUE

      END
```

E.2.2 The Geometric Test.

The configuration file for the geometric test. Note how the lack of replicators and the inablity to pass parameters causes inelegance.

```
processor host
processor zero
processor one
processor two
processor three
processor four

wire jumper host[0] zero[0]
wire ? zero[1] two[2]
wire ? zero[3] one[0]
wire ? one[1] three[3]
wire ? one[2] two[0]
wire ? one[3] three[1]
wire ? two[1] four[3]
wire ? two[3] four[1]
wire ? three[0] four[2]
wire ? three[2] four[0]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10k
task g0 ins=4 outs=4
task g1 ins=5 outs=5
task g2 ins=5 outs=5
task g3 ins=5 outs=5
task g4 ins=5 outs=5

place afserver host
place filter zero
place g0 zero
place g1 one
place g2 two
place g3 three
place g4 four

connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? g0[1] filter[1]
connect ? filter[1] g0[1]
connect ? g0[2] g1[1]
connect ? g1[1] g0[2]
connect ? g0[3] g2[3]
connect ? g2[3] g0[3]
connect ? g1[2] g3[4]
```

```
connect ? g3[4] g1[2]
connect ? g1[3] g2[1]
connect ? g2[1] g1[3]
connect ? g1[4] g3[2]
connect ? g3[2] g1[4]
connect ? g2[2] g4[4]
connect ? g4[4] g2[2]
connect ? g2[4] g4[2]
connect ? g4[2] g2[4]
connect ? g3[1] g4[3]
connect ? g4[3] g3[1]
connect ? g3[3] g4[1]
connect ? g4[1] g3[3]
```

The following code is the programs for the geometric test with 3L parallel FORTRAN.

```
      PROGRAM ZERO

      IMPLICIT NONE

      INCLUDE 'TIMER.INC'

      INTEGER PROCID,TICKS
      PARAMETER (PROCID = 0)
      PARAMETER (TICKS = 15625)
      CHARACTER MESS(1024)
      INTEGER I,T1,T2
      REAL RTIME

      T1=F77_TIMER_NOW()
      DO 99  I=1,10
        CALL CHANINMESSAGE(2,MESS,1024)
        CALL CHANOUTMESSAGE(3,MESS,1024)
C        PRINT *,'HELLO 3'
        CALL CHANINMESSAGE(3,MESS,1024)
        CALL CHANOUTMESSAGE(2,MESS,1024)
C        PRINT *,'HELLO 1'
 99   CONTINUE
      T2=F77_TIMER_NOW()
      T1=T2-T1
      RTIME = REAL(T1)/REAL(TICKS)
      PRINT *,RTIME

      END
```

```
      PROGRAM ONE

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 1)
      CHARACTER MESS1(1024)
      CHARACTER MESS2(1024)
      CHARACTER MESS3(1024)
      CHARACTER MESS4(1024)
      CHARACTER MESS5(1024)
      CHARACTER MESS6(1024)
      CHARACTER MESS7(1024)
      CHARACTER MESS8(1024)
      INTEGER I,J,T1,T2
      REAL R1

      DO 99  I=1,10
        CALL CHANOUTMESSAGE(1,MESS1,1024)
        CALL CHANINMESSAGE(1,MESS5,1024)
        CALL CHANOUTMESSAGE(3,MESS3,1024)
        CALL CHANINMESSAGE(3,MESS7,1024)
        CALL CHANOUTMESSAGE(2,MESS2,1024)
        CALL CHANINMESSAGE(2,MESS6,1024)
        CALL CHANOUTMESSAGE(4,MESS4,1024)
        CALL CHANINMESSAGE(4,MESS8,1024)
        DO 999 J=1,1000000
          R1 = R1 + 1.0
999     CONTINUE
99    CONTINUE

      END
```

```
      PROGRAM TWO

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 2)
      CHARACTER MESS1(1024)
      CHARACTER MESS2(1024)
      CHARACTER MESS3(1024)
      CHARACTER MESS4(1024)
      CHARACTER MESS5(1024)
      CHARACTER MESS6(1024)
      CHARACTER MESS7(1024)
      CHARACTER MESS8(1024)
      INTEGER I,J,T1,T2
      REAL R1

      DO 99  I=1,10
        CALL CHANINMESSAGE(3,MESS7,1024)
        CALL CHANOUTMESSAGE(3,MESS3,1024)
        CALL CHANINMESSAGE(1,MESS5,1024)
        CALL CHANOUTMESSAGE(1,MESS1,1024)
        CALL CHANOUTMESSAGE(2,MESS2,1024)
        CALL CHANINMESSAGE(2,MESS6,1024)
        CALL CHANOUTMESSAGE(4,MESS4,1024)
        CALL CHANINMESSAGE(4,MESS8,1024)
        DO 999 J=1,1000000
          R1 = R1 + 1.0
999     CONTINUE
99    CONTINUE

      END
```

```
      PROGRAM THREE

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 3)
      CHARACTER MESS1(1024)
      CHARACTER MESS2(1024)
      CHARACTER MESS3(1024)
      CHARACTER MESS4(1024)
      CHARACTER MESS5(1024)
      CHARACTER MESS6(1024)
      CHARACTER MESS7(1024)
      CHARACTER MESS8(1024)
      INTEGER I,J,T1,T2
      REAL R1

      DO 99  I=1,10
        CALL CHANOUTMESSAGE(1,MESS1,1024)
        CALL CHANINMESSAGE(1,MESS5,1024)
        CALL CHANOUTMESSAGE(3,MESS3,1024)
        CALL CHANINMESSAGE(3,MESS7,1024)
        CALL CHANINMESSAGE(4,MESS8,1024)
        CALL CHANOUTMESSAGE(4,MESS4,1024)
        CALL CHANINMESSAGE(2,MESS6,1024)
        CALL CHANOUTMESSAGE(2,MESS2,1024)
        DO 999 J=1,1000000
          R1 = R1 + 1.0
999     CONTINUE
99    CONTINUE

      END
```

```
      PROGRAM FOUR

      IMPLICIT NONE

      INTEGER PROCID
      PARAMETER (PROCID = 4)
      CHARACTER MESS1(1024)
      CHARACTER MESS2(1024)
      CHARACTER MESS3(1024)
      CHARACTER MESS4(1024)
      CHARACTER MESS5(1024)
      CHARACTER MESS6(1024)
      CHARACTER MESS7(1024)
      CHARACTER MESS8(1024)
      INTEGER I,J,T1,T2
      REAL R1

      DO 99  I=1,10
        CALL CHANINMESSAGE(3,MESS7,1024)
        CALL CHANOUTMESSAGE(3,MESS3,1024)
        CALL CHANINMESSAGE(1,MESS5,1024)
        CALL CHANOUTMESSAGE(1,MESS1,1024)
        CALL CHANINMESSAGE(4,MESS8,1024)
        CALL CHANOUTMESSAGE(4,MESS4,1024)
        CALL CHANINMESSAGE(2,MESS6,1024)
        CALL CHANOUTMESSAGE(2,MESS2,1024)
        DO 999 J=1,1000000
          R1 = R1 + 1.0
999     CONTINUE
99    CONTINUE

      END
```

E.3 Par.C.

E.3.1 The Coarse Farm Test.

The Par.C language allows transparent access to the screen. Note how much more elegent and simple the code for the coarse farm test is compared to the two previous examples in this appendix

```
#include <stdio.h>
#include <system.h>
SYSTEM sys;

int main()
{
  int i;
  GetSysInfo( &sys );
  if (sys.Tn!=1){
    for(i=1;i<=50;++i)
    printf("Hello world from transputer %d\n",(sys.Tn-1));
    }
}
```

E.3.2 The Geometric Test.

The inclusion of language extansions and the single program multiple data model of Par.C produces compact and readable code. The geometric test code is shown below.

```
#include <stdio.h>
#include <system.h>
#pragma fpu
SYSTEM sys;

int main()
{
  int i,j,count;
  float rnum;
  char message0[1024];
  char message1[1024];
  char message2[1024];
  char message3[1024];
  char message4[1024];
  char message5[1024];
  char message6[1024];
  char message7[1024];
  GetSysInfo( &sys );
  if (sys.Tn!=1){
    for(i=1;i<=10;++i){
      par{
        SendLink(0,&message0,1024);
        RecvLink(0,&message4,1024);
        SendLink(1,&message1,1024);
        RecvLink(1,&message5,1024);
        SendLink(2,&message2,1024);
        RecvLink(2,&message6,1024);
        SendLink(3,&message3,1024);
        RecvLink(3,&message7,1024);
        }
      rnum = 0.0;
      for(count=1;count<=1000000;count++)
        rnum=rnum + 1.0;
      }
    }
  else{
    for(i=1;i<=10;++i)
      par{
          {
          /*printf("hello 1\n");*/
          RecvLink(1,&message0,1024);
          SendLink(3,&message0,1024);
          }
```

```
            {
            RecvLink(3,&message1,1024);
            SendLink(1,&message1,1024);
            }
        }
    }
  printf("hello\n");
}
```

E.4 ECCL

E.4.1 The Coarse Farm Test.

The ECCL configuration for the coarse farm test.

```
VAL network.size IS 5 :
VAL host.id IS 0 :

--  SC host goes here
--  SC worker goes here

PROGRAM
  NETWORK SIZE = network.size
    PAR
      CONNECT PROCESSOR 0 LINK 3 TO PROCESSOR 1 LINK 0
      CONNECT PROCESSOR 0 LINK 1 TO PROCESSOR 2 LINK 2
      CONNECT PROCESSOR 1 LINK 1 TO PROCESSOR 3 LINK 3
      CONNECT PROCESSOR 1 LINK 2 TO PROCESSOR 2 LINK 0
      CONNECT PROCESSOR 1 LINK 3 TO PROCESSOR 3 LINK 1
      CONNECT PROCESSOR 2 LINK 1 TO PROCESSOR 4 LINK 3
      CONNECT PROCESSOR 2 LINK 3 TO PROCESSOR 4 LINK 1
      CONNECT PROCESSOR 3 LINK 0 TO PROCESSOR 4 LINK 2
      CONNECT PROCESSOR 3 LINK 2 TO PROCESSOR 4 LINK 0
  HARNESS
    PAR i = 0 FOR network.size -1
      PAR
        CONNECT PROCESSOR (i+1) OUTPUT 0 TO
                PROCESSOR 0 INPUT i
        CONNECT PROCESSOR 0 OUTPUT i TO
                PROCESSOR (i+1) INPUT 0
  PLACED PAR
    PROCESSOR 0 T8
      PREDEF   CHAN OF INT keyboard :
      PREDEF   CHAN OF ANY screen :
      host (keyboard, screen)
    PLACED PAR i = 0 FOR network.size - 1
      PROCESSOR i + 1 T8
        worker (i+1)
```

The root processor occam for the coarse farm test with ECCL.

```
PROC host ([4]CHAN OF ANY OUTPUT, [4]CHAN OF ANY INPUT,
           VAL INT PROC.ID, VAL []INT FLAG,
           CHAN OF INT keyboard, CHAN OF ANY screen)
  #USE euser
  #USE userio
  VAL REAL32 ticks IS 15625.0(REAL32) :
  TIMER clock :
  INT pid,error,bytes.unsent,bytes.got,t1,t2 :
  [27]BYTE mess :
  REAL32 rtime :
  SEQ
    clock ? t1
    SEQ j=0 FOR 100
      SEQ i=0 FOR 4
        SEQ
          input.message(i,error,bytes.unsent,
                                bytes.got,mess,FLAG)
          [4]BYTE pnum RETYPES pid :
          SEQ
            input.message(i,error,bytes.unsent,
                                  bytes.got,pnum,FLAG)
          write.full.string (screen, mess )
          write.int (screen,pid,0)
          newline (screen)
    clock ? t2
    t1 := t2 -t1
    rtime := (REAL32 ROUND(t1)) / ticks
    write.real32 (screen,rtime,0,0)
    newline (screen)
    SEQ
      newline (screen)
      write.full.string (screen, "Terminate harness...")
      newline (screen)
      terminate.harness (0, error, FLAG)
    INT any :
    SEQ
      newline (screen)
      newline (screen)
      write.full.string (screen,
                         "Press <any> to return to TDS")
      keyboard ? any
:
```

The worker code for the coarse farm test with ECCL.

```
PROC worker ([1]CHAN OF ANY OUTPUT, [1]CHAN OF ANY INPUT,
             VAL INT PROC.ID, VAL []INT FLAG,
             VAL INT pid)
  #USE euser
  INT error,bytes.unsent,pn :
  VAL message IS "Hello world from processor " :
  SEQ
    pn := pid
    SEQ j=0 FOR 100
      SEQ
        output.message (0,error,bytes.unsent,message,FLAG)
        [4]BYTE pnum RETYPES pn :
        output.message (0,error,bytes.unsent,pnum,FLAG)
:
```

E.4.2 The Geometric Test.

The configuration level for the geometric test with ECCL.

```
VAL network.size IS 5 :
VAL host.id IS 0 :
--  SC host goes here
--  SC worker goes here
PROGRAM
  NETWORK SIZE = network.size
    PAR
      CONNECT PROCESSOR 0 LINK 3 TO PROCESSOR 1 LINK 0
      CONNECT PROCESSOR 0 LINK 1 TO PROCESSOR 2 LINK 2
      CONNECT PROCESSOR 1 LINK 1 TO PROCESSOR 3 LINK 3
      CONNECT PROCESSOR 1 LINK 2 TO PROCESSOR 2 LINK 0
      CONNECT PROCESSOR 1 LINK 3 TO PROCESSOR 3 LINK 1
      CONNECT PROCESSOR 2 LINK 1 TO PROCESSOR 4 LINK 3
      CONNECT PROCESSOR 2 LINK 3 TO PROCESSOR 4 LINK 1
      CONNECT PROCESSOR 3 LINK 0 TO PROCESSOR 4 LINK 2
      CONNECT PROCESSOR 3 LINK 2 TO PROCESSOR 4 LINK 0
  HARNESS
    PAR i = 0 FOR network.size -1
      PAR
        CONNECT PROCESSOR (i+1) OUTPUT 0 TO
                PROCESSOR 0 INPUT i
        CONNECT PROCESSOR 0 OUTPUT i TO
                PROCESSOR (i+1) INPUT 0
    PAR
      CONNECT PROCESSOR 1 OUTPUT 1 TO PROCESSOR 2 INPUT 1
      CONNECT PROCESSOR 2 OUTPUT 1 TO PROCESSOR 1 INPUT 1
      CONNECT PROCESSOR 3 OUTPUT 1 TO PROCESSOR 4 INPUT 1
      CONNECT PROCESSOR 4 OUTPUT 1 TO PROCESSOR 3 INPUT 1
      CONNECT PROCESSOR 1 OUTPUT 2 TO PROCESSOR 3 INPUT 2
      CONNECT PROCESSOR 3 OUTPUT 2 TO PROCESSOR 1 INPUT 2
      CONNECT PROCESSOR 2 OUTPUT 2 TO PROCESSOR 4 INPUT 2
      CONNECT PROCESSOR 4 OUTPUT 2 TO PROCESSOR 2 INPUT 2
  PLACED PAR
    PROCESSOR 0 T8
      PREDEF   CHAN OF INT keyboard :
      PREDEF   CHAN OF ANY screen :
      host (keyboard, screen)
    PLACED PAR i = 0 FOR network.size - 1
      PROCESSOR i + 1 T8
        worker (i+1)
```

The host code for the geometric test with ECCL.

```
PROC host ([4]CHAN OF ANY OUTPUT, [4]CHAN OF ANY INPUT,
           VAL INT PROC.ID, VAL []INT FLAG,
           CHAN OF INT keyboard, CHAN OF ANY screen)
  #USE euser
  #USE userio
  VAL REAL32 ticks IS 15625.0(REAL32) :
  TIMER clock :
  INT pid,error,bytes.unsent,bytes.got,t1,t2 :
  [27]BYTE mess :
  REAL32 rtime :
  SEQ
    clock ? t1
    SEQ j=0 FOR 10
      SEQ i=0 FOR 4
        SEQ
          input.message(i,error,bytes.unsent,
                                bytes.got,mess,FLAG)
          [4]BYTE pnum RETYPES pid :
          SEQ
            input.message(i,error,bytes.unsent,
                                  bytes.got,pnum,FLAG)
          write.full.string (screen, mess )
          write.int (screen,pid,0)
          newline (screen)
    SEQ i=0 FOR 4
      SEQ
        input.message(i,error,bytes.unsent,
                              bytes.got,mess,FLAG)
        [4]BYTE pnum RETYPES pid :
        SEQ
          input.message(i,error,bytes.unsent,
                                bytes.got,pnum,FLAG)
        write.full.string (screen, mess )
        write.int (screen,pid,0)
        newline (screen)
    clock ? t2
    t1 := t2 -t1
    rtime := (REAL32 ROUND(t1)) / ticks
    write.real32 (screen,rtime,0,0)
    newline (screen)
    SEQ
      newline (screen)
      write.full.string (screen, "Terminate harness...")
      newline (screen)
```

```
      terminate.harness (0, error, FLAG)
    INT any :
    SEQ
      newline (screen)
      newline (screen)
      write.full.string (screen,
                         "Press <any> to return to TDS")
      keyboard ? any
:
```

The worker code for the ECCL geometric test.

```
PROC worker ([3]CHAN OF ANY OUTPUT, [3]CHAN OF ANY INPUT,
             VAL INT PROC.ID, VAL []INT FLAG,
             VAL INT pid)
  #USE euser
  INT error,bytes.got,bytes.unsent,pn :
  VAL message IS "Hello world from processor " :
  [1024]BYTE mess0,mess1,mess2,mess3,mess4,mess5,mess6,mess7 :
  REAL32 rnum :
  SEQ
    pn := pid
    SEQ i = 0 FOR 10
      SEQ
        IF
          ((pid = 1) OR (pid = 3))
            SEQ
              output.message (1,error,bytes.unsent,mess0,FLAG)
              output.message (1,error,bytes.unsent,mess1,FLAG)
              input.message (1,error,bytes.unsent,
                                     bytes.got,mess4,FLAG)
              input.message (1,error,bytes.unsent,
                                     bytes.got,mess5,FLAG)
          TRUE
            SEQ
              input.message (1,error,bytes.unsent,
                                     bytes.got,mess4,FLAG)
              input.message (1,error,bytes.unsent,
                                     bytes.got,mess5,FLAG)
              output.message (1,error,bytes.unsent,mess0,FLAG)
              output.message (1,error,bytes.unsent,mess1,FLAG)
        IF
          (pid < 3)
            SEQ
              output.message (2,error,bytes.unsent,mess2,FLAG)
              output.message (2,error,bytes.unsent,mess3,FLAG)
              input.message (2,error,bytes.unsent,
                                     bytes.got,mess6,FLAG)
              input.message (2,error,bytes.unsent,
                                     bytes.got,mess7,FLAG)
          TRUE
            SEQ
              input.message (2,error,bytes.unsent,
                                     bytes.got,mess6,FLAG)
              input.message (2,error,bytes.unsent,
                                     bytes.got,mess7,FLAG)
```

```
            output.message (2,error,bytes.unsent,mess2,FLAG)
            output.message (2,error,bytes.unsent,mess3,FLAG)
        rnum := 0.0(REAL32)
        SEQ j = 0 FOR 1000000
          rnum := rnum + 1.0(REAL32)
        output.message (0,error,bytes.unsent,message,FLAG)
        [4]BYTE pnum RETYPES pn :
        output.message (0,error,bytes.unsent,pnum,FLAG)
    output.message (0,error,bytes.unsent,message,FLAG)
    [4]BYTE pnum RETYPES pn :
    output.message (0,error,bytes.unsent,pnum,FLAG)
:
```

E.5 VCR.

E.5.1 The Coarse Farm Test.

The VCR configuration for the coarse farm test.

```
#INCLUDE "hostio.inc"
VAL wnum IS 4 :
[wnum]CHAN OF ANY to.root :
[wnum+1]CHAN OF SP fs,ts :
[wnum+1]CHAN OF BOOL stopper :
PLACED PAR
  PROCESSOR 0 T8
    #USE "hosthook.c8x"
    #USE "root.c8h"
    PAR
      root.proc (fs[0],ts[0],to.root[0],to.root[1],
                 to.root[2],to.root[3],0,wnum)
      []CHAN OF SP fs IS fs :
      []CHAN OF SP ts IS ts :
      []CHAN OF BOOL stopper IS stopper :
      PAR i = 0 FOR (wnum+1)
        hosthook (fs[i],ts[i],stopper[i])
  PLACED PAR i=1 FOR wnum
    PROCESSOR i T8
      #USE "worker.cah"
      worker.proc (fs[i],ts[i],to.root[i-1],i,wnum)
```

The root process for the VCR coarse farm test.

```
#INCLUDE "hostio.inc"
PROC root.proc (CHAN OF SP fs,ts,
                CHAN OF ANY from0,from1,from2,from3,
                   VAL INT id,wnum)
  #USE "vhostio.lib"

  VAL REAL32 ticks IS 15625.0(REAL32) :
  TIMER clock :
  INT x0,x1,x2,x3 :
  INT t1,t2 :
  REAL32 rtime :
  SEQ
    clock ? t1
    PAR
      from0 ? x0
      from1 ? x1
      from2 ? x2
      from3 ? x3
    clock ? t2
    t1 := t2-t1
    rtime := (REAL32 ROUND(t1))/ticks
    so.write.real32(fs,ts,rtime,0,0)
    so.exit(fs,ts,sps.success)
:
```

The worker code for the coarse farm test with VCR.

```
#INCLUDE "hostio.inc"
PROC worker.proc (CHAN OF SP fs,ts,CHAN OF ANY to.root,
                  VAL INT id,wnum)

  #USE "vhostio.lib"
  SEQ
    SEQ i=0 FOR 100
      IF
        id=1
          so.write.string.nl (fs,ts,
                  "Hello world from processor 1")
        id=2
          so.write.string.nl (fs,ts,
                  "Hello world from processor 2")
        id=3
          so.write.string.nl (fs,ts,
                  "Hello world from processor 3")
        id=4
          so.write.string.nl (fs,ts,
                  "Hello world from processor 4")
    to.root ! id
:
```

E.5.2 The Geometric Test

The VCR configuration for the geometric test.

```
#INCLUDE "hostio.inc"
VAL wnum IS 4 :
VAL index IS [[0,3],[1,0],[2,1],[3,2]] :
[wnum]CHAN OF ANY to.root :
[wnum]CHAN OF ANY loop.up :
[wnum]CHAN OF ANY loop.down :
[wnum+1]CHAN OF SP fs,ts :
[wnum+1]CHAN OF BOOL stopper :
PLACED PAR
  PROCESSOR 0 T8
    #USE "hosthook.c8x"
    #USE "groot.c8h"
    PAR
      root.proc (fs[0],ts[0],to.root[0],to.root[1],
                 to.root[2],to.root[3],0,wnum)
      []CHAN OF SP fs IS fs :
      []CHAN OF SP ts IS ts :
      []CHAN OF BOOL stopper IS stopper :
      PAR i = 0 FOR (wnum+1)
        hosthook (fs[i],ts[i],stopper[i])
  PLACED PAR i=1 FOR wnum
    PROCESSOR i T8
      #USE "gworker.c8h"
      VAL xi IS index[i-1][0] :
      VAL yi IS index[i-1][1] :
      worker.proc (fs[i],ts[i],loop.up[xi],loop.down[xi],
                   loop.down[yi],loop.up[yi],
                   to.root[i-1],i,wnum)
```

The root occam for the VCR geometric test.

```
#INCLUDE "hostio.inc"
PROC root.proc (CHAN OF SP fs,ts,
                  CHAN OF ANY from0,from1,from2,from3,
                     VAL INT id,wnum)
  #USE "vhostio.lib"

  VAL REAL32 ticks IS 15625.0(REAL32) :
  TIMER clock :
  [256]INT x0,x1,x2,x3 :
  INT t1,t2 :
  REAL32 rtime :
  SEQ
    clock ? t1
    PAR
      from0 ? x0
      from1 ? x1
      from2 ? x2
      from3 ? x3
    clock ? t2
    t1 := t2-t1
    rtime := (REAL32 ROUND(t1))/ticks
    so.write.real32(fs,ts,rtime,0,0)
    so.exit(fs,ts,sps.success)
:
```

The worker code for the geometric test with VCR.

```
#INCLUDE "hostio.inc"
PROC worker.proc (CHAN OF SP fs,ts,
                  CHAN OF ANY xout,xin,yout,yin,to.root,
                     VAL INT id,wnum)

  #USE "vhostio.lib"
  [256]INT b0,b1,b2,b3,b4,b5,b6,b7,b8 :
  REAL32 rnum :
  SEQ
    SEQ i=0 FOR 100
      SEQ
        PAR
          SEQ
            xout ! b0
            xout ! b1
          SEQ
            yout ! b2
            yout ! b3
          SEQ
            xin ? b4
            xin ? b5
          SEQ
            yin ? b6
            yin ? b7
        rnum := 0.0(REAL32)
        SEQ j=0 FOR 1000000
          rnum := rnum + 1.0(REAL32)
    to.root ! b8
    so.write.string.nl (fs,ts,"Done")
:
```

E.6 Express.

E.6.1 The Coarse Farm Test.

The coarse farm C for the Express system.

```
#include "express.h"
#include <stdio.h>

struct nodenv env;

main()
{
  long t1,t2;
  int i;
  t1=extime();
  exparam(&env);
  fmulti(stdout);
  for(i=1;i<=1;++i)
  printf("Hello world from processor %d\n",env.procnum);
  fflush(stdout);
  t2=extime();
  t1=t2-t1;
  printf("%d %d\n",env.procnum,t1);
  exit(0);
}
```

E.6.2 The Geometric Test.

The geometric test code for the Express system.

```
#include "express.h"
#include <stdio.h>

main()
{
  struct nodenv env;
  long t1,t2;
  int i,j,count,loop;
  float rnum;
  int length,psrc,ptype;
  int dest,type;
  char message0[1024];
  char message1[1024];
  char message2[1024];
  char message3[1024];
  char message4[1024];
  char message5[1024];
  char message6[1024];
  char message7[1024];
  type = 5;
  ptype = 5;
  t1=extime();
  exparam(&env);
  fmulti(stdout);
  for(loop=1;loop<=100;++loop){
  if (env.procnum==0){
      psrc = DONTCARE ;
      dest = 1;
      exwrite(message0,1024,&dest,&type);
      exread(message4,1024,&psrc,&ptype);
      exwrite(message1,1024,&dest,&type);
      exread(message5,1024,&psrc,&ptype);
      dest = 2 ;
      psrc = DONTCARE ;
      exwrite(message2,1024,&dest,&type);
      exread(message6,1024,&psrc,&ptype);
      exwrite(message3,1024,&dest,&type);
      exread(message7,1024,&psrc,&ptype);
    }
    else if (env.procnum==1){
      dest = 0;
      psrc = DONTCARE ;
      exwrite(message0,1024,&dest,&type);
      exread(message4,1024,&psrc,&ptype);
```

```
        exwrite(message1,1024,&dest,&type);
        exread(message5,1024,&psrc,&ptype);
        dest = 3;
        psrc = DONTCARE ;
        exwrite(message2,1024,&dest,&type);
        exread(message6,1024,&psrc,&ptype);
        exwrite(message3,1024,&dest,&type);
        exread(message7,1024,&psrc,&ptype);
      }
      else if (env.procnum==2){
        dest = 3;
        psrc = DONTCARE ;
        exwrite(message0,1024,&dest,&type);
        exread(message4,1024,&psrc,&ptype);
        exwrite(message1,1024,&dest,&type);
        exread(message5,1024,&psrc,&ptype);
        dest = 0;
        psrc = DONTCARE ;
        exwrite(message2,1024,&dest,&type);
        exread(message6,1024,&psrc,&ptype);
        exwrite(message3,1024,&dest,&type);
        exread(message7,1024,&psrc,&ptype);
      }
      else if (env.procnum==3){
        dest = 2;
        psrc = DONTCARE ;
        exwrite(message0,1024,&dest,&type);
        exread(message4,1024,&psrc,&ptype);
        exwrite(message1,1024,&dest,&type);
        exread(message5,1024,&psrc,&ptype);
        dest = 1;
        psrc = DONTCARE ;
        exwrite(message2,1024,&dest,&type);
        exread(message6,1024,&psrc,&ptype);
        exwrite(message3,1024,&dest,&type);
        exread(message7,1024,&psrc,&ptype);
      }
      rnum = 0.0;
      for(count=1;count<=1000000;++count)
        rnum = rnum + 1.0;
    }
    fflush(stdout);
    t2=extime();
    t1=t2-t1;
    printf("finished %d %d\n",env.procnum,t1);
    exit(0);
}
```

E.7 Helios.

E.7.1 The Coarse Farm Test.

The following single line configures the processors for the coarse farm test in helios using the CDL.

```
hello [3] ^^ hello
```

The code run under helios for the hello test follows :

```
#include <stdio.h>
#include <nonansi.h>

int main()
{
int i,t1,t2;
t1=_cputime();
for(i=1;i<=100;++i)
fprintf(stderr,"Hello world from processor %d\n",0);
t2=_cputime();
t1=t2-t1;
fprintf(stderr,"%d\n",t1);
}
```

E.7.2 The Geometric Test.

The geometric test was configured with the following CDL :

```
master ( , [4] <> worker)
```

The code was split into two parts. The master code :

```
#include <stdio.h>
#include <syslib.h>
#include <helios.h>
#include <nonansi.h>
#include <POSIX.h>

int main ()
{
int t1,t2,n,result;
char buff1[1024],buff2[1024],buff3[1024],buff4[1024];
char buff5[1024],buff6[1024],buff7[1024],buff8[1024];
t1=_cputime();
for(n=1;n<=10;++n){
result = read(4,buff1,1024);
result = read(4,buff2,1024);
result = read(6,buff3,1024);
result = read(6,buff4,1024);
result = read(8,buff5,1024);
result = read(8,buff6,1024);
result = read(10,buff7,1024);
result = read(10,buff8,1024);
result = write(5,buff3,1024);
result = write(5,buff4,1024);
result = write(7,buff1,1024);
result = write(7,buff2,1024);
result = write(9,buff7,1024);
result = write(9,buff8,1024);
result = write(11,buff5,1024);
result = write(11,buff6,1024);
result = read(4,buff1,1024);
result = read(4,buff2,1024);
result = read(6,buff3,1024);
result = read(6,buff4,1024);
result = read(8,buff5,1024);
result = read(8,buff6,1024);
result = read(10,buff7,1024);
result = read(10,buff8,1024);
result = write(5,buff5,1024);
```

```
result = write(5,buff6,1024);
result = write(7,buff7,1024);
result = write(7,buff8,1024);
result = write(9,buff1,1024);
result = write(9,buff2,1024);
result = write(11,buff3,1024);
result = write(11,buff4,1024);
printf("Hello %d\n",n);
}
result = read(4,buff1,1);
result = read(6,buff1,1);
result = read(8,buff1,1);
result = read(10,buff1,1);
t2=_cputime();
t1=t2-t1;
printf ("Hello from master %d\n",t1);
}
```

The worker code :

```
#include <stdio.h>
#include <syslib.h>
#include <helios.h>
#include <POSIX.h>

int main ()
{
int i,n,result;
char mess1[1024],mess2[1024],mess3[1024],mess4[1024];
char mess5[1024],mess6[1024],mess7[1024],mess8[1024];
float rnum;
for(n=1;n<=10;++n){
result = write(1,mess1,1024);
result = write(1,mess2,1024);
result = read(0,mess5,1024);
result = read(0,mess6,1024);
result = write(1,mess3,1024);
result = write(1,mess4,1024);
result = read(0,mess7,1024);
result = read(0,mess8,1024);
rnum = 0.0;
for(i=1;i<=1000000;++i)
rnum = rnum + 1.0;
}
result = write(1,mess1,1);
}
```

F List of Company Addresses.

The company addresses here are the sources for all the software used or mentioned in this thesis and the various employers of the author during the work.

The software was from :

(Hardware, occam toolsets)

INMOS Limited
1000 Aztec West
Almondsbury
Bristol
BS12 4SQ
UK.

Tel. 0454 616616

(Transputer Ada)

Alsys Limited
Partridge House
Newtown Road
Henley-on-Thames
RG9 1EN
UK.

Tel. 0491 579090

(Transputer FORTRAN, C and Pascal compilers)

3L Limited
Peel House
Ladywell
Livingston
EH54 6AG
UK.

Tel. 0506 415959

(Transputer C compiler)

Parsec Developments
PO Box 782
2300 AT
Leiden
The Netherlands

Tel. +3171 142142

(Transputer Express system, C and FORTRAN compilers)

ParaSoft Corporation
2500, E.Foothill Blvd.
Pasadena
CA 91107
USA

(Helios transputer operating system and C compiler)

Distributed Software Limited
OR
Perihelion Software Limited
24 Brewmaster Buildings
Charlton Trading Estate
Shepton Mallet
Somerset
BA4 5QE
UK

Tel. 0749 4203

The author was employed by the following organisations during this work.

PARSYS Limited
Boundary House
Boston Manor
London
W7 2QE
UK.

Tel. 081 579 8683

based at
University of Southampton
Highfield
Southampton
SO9 5NH
UK.

Tel. 0703 590000

then at what is now
The Parallel Applications Centre
Unit 2
Chilworth Research Centre
Chilworth
Hampshire
SO1 7NP
UK

Tel. 0703 760834
Tel. 0703 760835

Southampton Novel Architecture Research Centre
Department of Electronics and Computer Science
University of Southampton
Highfield
Southampton
SO9 5NH
UK.

Tel. 0703 592069