FARMING OUT: A STUDY

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Warren Godfrey Day
March 7, 1996

Dedication

To my Mum and Dad with all my love, for giving me life and everything they realised they possible could. Many thanks for it all.

To Tony Buzan, who's fascinating books and tuition on the brain, how to learn to learn, and on mental literacy in general, not only helped in this Ph.D., but also made it possible in the first place; for starting up the Use Your Head Club, for putting many things, including life, into context positively.

To Vanda North for her endless encouragement and enthusiasm.

Acknowledgements

The following people have either helped directly in the research in some way, or who just helped to make life wonderful.

I would like to thank the following for their general assistance during this research: Peter Welch, for his encouragement and help; Stephen Turner, for his input and thorough list of corrections; David Morse, for conversations, help and friendship on this long journey; Herman Roebbers, for his approach to optimisation; David Beckett, for his help on Kent's transputer setup; and Steve Hill for being my supervisor and for reading the many drafts.

Thanks also go to: Jayadev Misra, for his help in clarifying my understanding of UNITY; plus Edgar Knapp and Mark Staskaukas for their help and LaTeX markup for UNITY notation.

Further thanks to: Munna Lee, for her assistance with LaTeX; Ian Utting, for explainging document structure theory and how to implement it; Peter Barrett, for a copy of his Mind Maps Plus, on which this thesis was planned and written; the lab.'s operators for restoring corrupted files quickly. A general thanks to everyone in the laboratory who either helped directly or helped to make the lab. a nice place to work. Thanks also to Dominic Powell for getting me into Strunk and White's The Elements of Style.

Thanks also to my friends, fellow Hi-Fi enthusiasts and companions who have been there for me, and thus in some part helped get me through.

A very special thanks goes to: Jane Gower, for being the best of friends, and for always being at the other end of a phone; John Whyte, who has been a friend for so long, he's been a friend for almost as long before I started my Ph.D. as he has been a friend throughout it; Stuart Finch, for his friendship and similar passion about music; Nick Smith, for being a good friend, for help and also for introducing me to Shane; Ian Catchpole, for his friendship, conversations, interests and knowledge of life on the other side of the world's biggest pond; Matt Tanner, for being a friend, for his similar passion for Hi-Fi, for discussions about music and its reproduction, and for introducing me to Neil; Jackie Dann (Church), for long chats; Gary Ellis, another long term friend, who deserves not only many thanks, but a special thanks for inadvertently reminding me how much I liked occam! A large thank you to my Polish friends for their generousity: Bożena Bartoszek, for her friendship and for being a good colleague; Michal Wojtulewicz for his friendship, fascinating conversations and interest in Mind Mapping and the brain; Piotr, for my first place to stay in Poland and for introducing me to Kasia; Kasia and Tomek Warchulska, for their friendship.

Many thanks to Neil McBride, for his home-brew and fascinating discussions about music, its timing and its reproduction, electronics and space science. Thanks also go to Shane Sturrock, for talking so much about his work, for being so motivated, for introducing me to Old Peculiar and for everything else he does so well, including showing me what good Hi-Fi is and how to buy it.

Thanks also to: Stephen Thomas, for many long enjoyable and fascinating conversations, and especially for introducing me to Jane; Mike Ellis, for many conversations and interesting information about broadcasting technology; Heather Standring, for conversations, a common interest in playing the piano, cookery lessons and for motorbike pillion rides; Ian Buckner, for his many conversations and enthusiasm for life; Danielle Banyai, for her support over lunches; Paul Tysoe, for making so many meals so enjoyable; Scott Woodsell, Toby and Linda, for a very enjoyable social life recently; David Mitchem, for his great wisdom and conversations over lunches; and, Caroyln Piesse, for conversations and fun.

Lucie Tepla deserves a special thanks as her stunning example of success has been an inspiration.

Many thanks again to Mum and Dad, Tony and Vanda.

Thanks to various members of the Use Your Head Club for their encouragement and help.

Thanks to Ivor Tiefenbrun for the Sondek LP12 turntable and Linn Products Limited. Thanks for introducing me to some great music and musicians (especially the Palladian Ensemble) and for my Axis and my Sondek that have kept me so entertained during the construction of this thesis.

Rachel Podger deserves a special thanks for demonstrating that even the most difficult things should be performed effortlessly.

Abstract

Farming is one of severals ways of arranging for a group of individuals to perform work simultaneously.

Farming is attractive. It is a simple concept, and yet it allocates work dynamically, balancing the load automatically. This gives rise to potentially great efficiency; yet the range of applications that can be farmed efficiently and which implementation strategies are the most effective has not been classified.

This research has investigated the types of application, design and implementation that farm efficiently on computer systems constructed from a network of communicating parallel processors. This research shows that all applications can be farmed and identifies those concerns that dictate efficiency.

For the first generation of transputer hardware, extensive experiments have been performed using occam, independent of any specific application. This study identified the boundary conditions that dictate which design parameters farm efficiently. These boundary conditions are expressed in a general form that is directly amenable to other architectures. The specific quantitative results are of direct use to others who wish to implement farms on this architecture.

Because of farming's simplicity and potential for high efficiency, this work concludes that architects of parallel hardware should consider binding this paradigm into future systems so as to enable the dynamic allocation of processes to processors to take place automatically. As well as resulting in high levels of machine utilisation for all programs, this would also permanently remove the burden of allocation from the programmer.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Here we are interested in the technique of farming. This involves distributing the separate parts of a large task to individuals; these parts being distributed by communication.

Farming out is of interest as it can achieve very high levels of efficiency. Further, the number of workers involved in performing the work can be changed while the task is performed.

Farming is a very simple yet effective method of allocating work to workers. As well as being simple it has been in existence for some time.

The thesis of this research is that, as farming is highly scalable and has great potential for efficiency (due to the work load being balanced automatically via dynamic allocation), it should be used more. This includes adding its mechanism to parallel hardware to alleviate the need to continually reimplementing it in software.

1.1 Contents

Documented here is a two part study on farming: how to implement it and what can be farmed.

The first part looks at what makes an implementation efficient for both any architecture and the first generation of INMOS transputer. It also looks at some highly efficient farming harnesses and implementation practices.

The second part searches for and finds what domain of applications can be farmed out. This is performed by studying a number of applications (including implementations) and the mechanics of farmings itself. This has lead to a much clearer understanding of what can be implemented and how.

These studies constitute a scientific and rigorous exploration of the technique of farming, and to the knowledge of the author is first of such a study. The experimentation is the first known to be performed where the testing is independent of any one particular application.

1.2 Theories used

This work was conducted in the local environment that exists within the Parallel Processing Group at Kent and within the community of the World occam and Transputer User Group. Both of whom appreciate the advantages of a C.S.P. as a paradigm for modelling parallel interaction, the occam parallel programming language with its algebraic semantics and the INMOS transputers. The part of this work that deals with the development and testing of farming harnesses is influenced and in some respects may even be restricted by the particular knowledge, approaches and prejudices in this particular working environment.

Also used here is UNITY [CM87]. This is a foundation theory for programming that attempts to provide a framework in which to develop programs for all architectures.

1.2.1 On the use of UNITY

UNITY has only been used here to view the overall development of programs. It has not been used to develop programs formally. Thus, the only aspect of UNITY used here is that programs should be designed before implementation is attempted.

The results presented here fit into this philosophy and can be used to farm out program designs. This work is only related to how farm out program designs, not how to develop them in the first place. The design could have been developed in UNITY, but this is not mandatory.

Thus in essence, what is to be farmed out, should be known before using these results to implement an efficient farm.

1.3 Thesis structure

The material in this thesis is presented in the order in which it was worked on and thus discovered. Thus, all experiments are presented in the order they were performed. Similarly, the experimentation on farming harnesses is presenting before the exploration of farming's usability, as these two areas of the research was performed in this order. The work of others is invariably mentioned along side the work to which it is relevant.

The contents of this thesis's chapters are as follows.

Chapter 2 contains the background material needed to understand the work in this thesis. A reasonable knowledge and fluency of occam and transputers is assumed however.

Chapter 3 documents how this study came about.

Chapter 4 documents the experiments performed, the analysis of the results, the developing of understanding and the resulting conclusions.

Chapter 5 documents the exploration into how wide a range of applications can farming be used to execute. A much clearer understanding of farming and a general model of farming are developed.

Chapter 6 discusses some future work. It also looks at how farms will be built on some future generations of transputers and how these will be programmed in future version of occam.

Chapter 7 discusses the contribution and conclusions of this work.

Chapter 2

Background material

This chapter looks at the background material used in this research. This material consists of two theories. These take greatly different approaches to programming; this being due to their developed being independent. We will look at these two theories in the order in which they were learnt. The first of these two areas to be looked at was derived from Hoare's C.S.P. ([Hoa78] [Hoa85] [HJ89a]). The second is UNITY, a theory of parallel programming by Chandy, Misra [CM87] and others. UNITY was developed to be a foundation for programming.

This first area consists of a number of separate parts. These are the various aspects of occam program design and optimisation. Here we look at the following.

1. The method used to construct programs taught in a course by Welch.
2. An overview of the relevant parts of the parallel programming language occam (including its design principles, its semantics, the essences of its syntax and structure, and its transformation laws).
3. The transputer hardware on which the practical side of this work was performed.
4. The theory used for transforming programs into more efficient forms taught in a course by Roebbers.
5. Lastly, we look in more detail at the processor farm and what the general opinion was of the subject when this research started.

This work concentrated more on the philosophy behind the work, instead of the practical issues such as language. Thus, the philosophy behind the method of programming is mentioned before the programming language itself.

2.1 The process oriented method of programming

In this section the process oriented method of programming is looked at. This methodology is used to develop communicating parallel programs. In this work all programs are written in occam. The methodology introduced in this section is used as the norm both at Kent and in many establishments in both academia and industry.

The process oriented method of programming promotes the following,

1. a process oriented view of the application,
2. programs that are the same shape as the application,
3. one software component per single unit of application's functionality, and,
4. software reuse.

This process oriented methodology is presented on the "occam and Transputer Engineering Workshop" course [Wel]. The method is aimed at describing parallel systems with clarity. This was the driving concept in the design of occam and the transputer. As a result the occam language is ideal for describing parallel systems succinctly. Occam was also designed to be easy to compile. The transputer was designed to be efficient at executing such programs.

The model of parallelism used is Communicating Sequential Processes (C.S.P.). This model, due to Hoare [Hoa85], contains communication as a programming primitive. The design of occam and the transputer are based on the C.S.P. model.

As a method of programming the process oriented approach is simple. It consists of the functionality of the system being modeled as a number of independent communicating processes.

The course [Wel] mentioned three ways of organising work for a number of communicating parallel computers. Most of the course looks at the major problem of computer programming, the method itself and how it hopes to go some way towards solving this problem. Where this theory comes from is looked at here, as it what it contains, what are its advantages and how it is used in practise.

2.1.1 Parallel programming paradigms

It appears there are three elementary methods with which to organise communicating processors. The three techniques are as follows,

1. geometric distributions,
2. algorithmic distributions, such as pipelines, or,
3. processor farms.

The details of these are discussed below.

Geometric distributions

A geometric distribution can be used where a set of data repeatly needs the same process performed upon it. Here the domain of data is divided among the processors, all of which perform the same task. The processors communicate to exchange information across processor boundaries.

Solving the n-body problem (simulating planets or molecular particles) is an example of a geometric distribution, as is simulating (predicting) the weather. In the latter, all processors are programmed with the appropriate atmospheric model and are then arranged to look after one part of the atmosphere. Each processor is given either a strip or a volume of atmosphere. Then each processor starts with the initial data for its part of the atmosphere. The system is then left to run. In the n-body problem processors are programmed with the appropriate Newtonian mechanics and a number of bodies are then allocated to each processor.

Some method of performing data input or result output is also needed. This is either performed by having a processor observing the data traversing around, say, a ring, or by all of the processors in the network communicating results directly to another transputer.

It is believed a geometric distribution should give something like 90% of the maximum performance of which the hardware is capable [Wel]. The loss can be due to processors waiting for data from neighbouring processors. Losses are also caused by the lockstep nature of this approach, as some processors may be lying idle while waiting for other processors to finish the more time consuming calculations.

Algorithmic distributions

Algorithmic distributions consist of a functional breakdown of the steps taken to perform a task. One common example is that of a pipeline. Here data enters at one end of the pipeline and results emerge from the other. More elaborate algorithmic distributions are possible with data flowing around an intricate network of processes. One domain where this approach has been used is real time systems. Pipelines

are also not necessarily always one process or one processor wide at any one stage. If a great deal of processing is required at one point there could be several processors, each performing some suitable fraction of the work.

The drawback of algorithmic distributions is the whole system can only work as fast as the slowest component. To build pipelines that work to a high degree of efficiency involves precise knowledge and continual consideration of exactly what all of the processors will be doing. This includes knowing very precisely what the compiler will generate and what the execution characteristics of the processors are. Thus, to obtain more than 70% efficiency from an algorithmic distribution is believed to be difficult [Wel].

Processor farms

This method consists of having a worker process on each transputer. A farmer process hands out work to these workers. These items of work are then performed independently of each other. Upon completion the results are sent either back to the farmer or to a harvester process.

As a job of work is independent of any of the other jobs, the workers work independently of each other, and only communicating with the farmer and the harvester once for each job. The workers do not communicate with each other at any time. This gives rise to an important benefit. The different jobs can take different lengths of time to be executed, without there being any loss of performance. As soon as one job finishes another can be started immediately. There is no need to wait for any synchronisation with any of the other workers. Thus, there is no performance lost due to waiting for synchronisations. This is not the case with either an algorithm or a geometric distribution. With a farm it is in fact the case that it is better if the workers do finish at different times, as then the demand for work is staggered through time.

As transputers can only communicate directly with four other transputers, processor farms are implemented with the aid of a *farming harness*. This consists of some additional processes that are executed along with those already mentioned. These processes provide the logical interconnection indirectly that is not possible through direct physical means. The interconnection structure used is not relevant to the execution of the program itself. Nevertheless, different topologies may possess different communication overheads and thus the performance of the different topologies may vary.

It is believed that a farm can be 99% efficient [Wel].

2.1.2 Program organisation

What has probably been the greatest problem in computer science since its outset is that of how to develop correct and easily mantainable programs that are of a large size and complexity? The method of programming being discussed here uses a model of reality to aid in the solving of this problem.

This model consists of objects that interact with one another. For example, if an appropriate force is applied to a book, the book opens. The occam model restricts this C.S.P. model by saying that interactions take place through communication channels and that each channel only has two ends a sender and a receiver, and that communication is also synchronous. For example, light travels from the page of a book to the eye of the reader. This is synchronous as this communication can only happen when the book is prepared to be read, by being open, and the reader is prepared to look at the book.

The C.S.P. model of parallelism is modelled on how real interactions take place. Processes in reality communicate instead of having global memory. In our occam restriction of C.S.P. we have point-to-point communication as this has no multiple process contention problems. Communication is also unbuffered as well as synchronous and point-to-point.

A synchronous communication progresses in the following way. When one process wishes to communicate with another, the first process waits until the second is also ready to engage in the communication. Thus a communication is also a rendezvous between two processes.

Our approach to the designing and writing of parallel programs here consists simply of expressing the computation in terms of the parallel components inherent in what is being built. The behaviours of these processes are described from their own points of view. The interconnection between these processes coming from the shape of the system being modelled.

2.1.3 Advantages

A major advantage of this approach is it models reality directly. All the autonomous objects are modelled from their own point of view with their own independent behaviour and private data.

This clarity results in the design, implementation, validation and maintenance stages all being much easier to perform.

The other obvious advantage is that designs can easily be implemented to make use of parallel architectures. Indeed parallel hardware itself has been designed using this approach.

2.1.4 Design methodology

In this methodology programs are written so there is one logical function per process. This way each function is programmed from its own point-of-view. To obtain complex functionality such processes are composed in parallel, typically with these processes living for the whole length of the program's life. This method is termed "process-oriented" design.

This methodology is very simple. The first step is to draw a process diagram of the system being modelled. These processes are then written, each from their own point-of-view, with their own privately declared and retained internal state. The processes are then connected and run.

If simple changes need to be made to the system, we may only need to change the interconnection between the objects. If, for example, objects needed to be aware of each others existence, reprogramming would require changing many of these internal references, resulting in a great deal of unnecessary reprogramming.

Many processes terminate only when the program has completely finished. Some of these processes may be unable to determine when termination has occurred. For convenience such processes are written to execute indefinitely. In practise these can then be either terminated by resetting the transputer network (as was the case here) or by simply reusing the program's communication network. This network can allow for a shutdown message to be generated from a process that is able to directly determine when the program has terminated. The network then propagates these messages until all other processes have received a message. In farming for example, a shutdown message could be generated by the harvester and propagated through the harness and workers until the farmer receives a copy. This method of termination was suggested by Welch [Wel89].

2.1.5 On the transition from sequential to parallel programming

When writing parallel programs as experienced sequential programmers, who have developed a set of good sequential programming skills, care must be taken not to be set in these skills that are only appropriate for sequential programming. There is a natural human tendency to put into parallel programs all the small simple and local optimisations that are beneficial in sequential programs, but are not appropriate optimisations to make in parallel programs. Parallel programming is very different to the more conventional sequential programming. There is a different set of efficiency issues in parallel programming. For example, it is typically easier and more efficient to duplicate work than to communicate results.

2.1.6 Breadth of use

It was mentioned on the course that the process oriented method of programming can be used to write in any parallel programming language, not just occam. This has been performed on courses in Ada by Welch. On shared memory systems communication can be emulated.

In [LS90] Lin and Snyder have found that for shared memory architectures, the message-passing model is more efficient than the shared memory model for programs at an equivalent level of optimisation. The authors found this to be the case for two shared memory architectures for several versions of two applications. Simulated message-passing was between a few percent and 23 times faster. The reason for this seems to be down to the technique's exploitation of data locality and the large granularity of distributed memory programs. Lin and Snyder suggest a broader study with more machine types and programs. They are not sure whether the improvement is due to basic differences in their programming or due to actual advantages of the message-passing programming model. That said, the authors noticed

that the more optimised versions of the shared memory program possessed some of the characteristics of the message-passing programs.

2.2 The occam programming language

Here is an overview of version 2 of the occam programming language [Inm88]. As a language, occam is now well known. Thus, here most of the language will be covered quickly, spending more time emphasising the parts of the language that are often overlooked, but that are especially important to this work. For example, the program transformations made possible by the algebraic semantics are used reasonably extensively in this work.

Occam was designed so,

1. the concepts and constructs of parallelism are as equally central to the language as the sequential concepts and constructs,
2. programs would be efficient to implement,
3. the elements of the language would have a rigorously defined semantics,
4. the semantics would be algebraic, this giving programs an equivalence and thus allowing for programs to be transformed into others, and,
5. there would be facilities in the language to describe embedded systems just as easily as parallel systems.

Being an embedded systems language, occam has some restrictions. It should be easy to reason about the real time demands and the runtime performance of a program. Thus, the language's implementation should be efficient always, in all situations, and, for example, garbage collection cannot be allowed into the language. Equally the program should always be able to run. Thus there can be no parts that can fail, such as memory allocation or recursion. Further, with occam there is no memory management and the memory usage must be a determinable constant at compile-time. In fact in compiling generally, it is considered a good idea to not put off until run time what can be done at compile time [Gri71].

Being a parallel language, occam possesses the constructs to express several processes happening simultaneously. It also has the constructs for communication. The presence of parallelism in the language has restrictions in some of the sequential areas of the language. Recursion being one example, as growing an arbitrary number of stacks in parallel is not easy to implement without some form of garbage collection. The implementation would result in both long and nondeterministic runtimes for recursive calls; this is not appropriate for an embedded systems language. Similarly, although occam can syntactically express output guards, having them either exclusively or with input guards prevents an implementation from guaranteeing efficiency. Nevertheless, the fact that the syntax allows output guards means we can use them at the design stage.

The advantage of having a rigorous semantics is all programs have an exact meaning. Thus, all constructs in all situations and combinations have a defined unambiguous meaning not open to debate. This is not the case in most other languages.

2.2.1 Variable types

Occam has several integer types: `INT`, the size of the internal registers of the executing processor, `INT16`, `INT32` and `INT64` which are handled by the implementation appropriately. There is also `REAL32` and `REAL64` which fully implement the I.E.E.E. standard for computer floating point arithmetic. `BOOL` and `BYTE` are also implemented, but computation is not permitted on them directly. Arrays can be constructed of all of the built-in types.

2.2.2 Communication type

Communication in occam is synchronous, unbuffered and point-to-point. This form of communication is the simplest to have. Also, all other type of communication can be built from this: asynchronous, one-to-many, many-to-one, many-to-many.

Synchronous communications are also very easy to work with. Even asynchronous systems often have to perform synchronisations. As occam's communication is synchronised and unbuffered, the first process to arrive at the communication statement has to wait until the other process arrives at the communication.

The point to point communication in occam is where the occam is a semantic restriction of C.S.P. which may have any number of processes engage in an event.

In occam communications have a type. There is also the facility to group a number of types together into an ordered list. This is termed a `PROTOCOL`. These make it easier for the compiler to rigorously check the communication between two processes.

2.2.3 Processes

Occam programs are made up of processes. In turn these may be made up of constructs consisting of other processes. There are three basic processes: the assignment process, and two communication processes, input and output.

Assignment

The assignment process evaluates the expression on the right hand side of the process and checks any indexes on the left-hand side are valid, the result is then assigned to the left-hand side variable. The result of the expression and the variable must be of the same type.

```
total := 0
```

If the value is not computable, the error (division by zero for example) will cause a halt to occur. Multiple assignments are also allowed in occam. These consist of a set of assignments (written in list form). These follow the same rules as in above. The right-hand sides are evaluated, array indexes are computed and checked to be in range and then the results are assigned. Upon a valid completion, all of the values are assigned to the ordered list of variables which must also be of the same length.

```
x, y := y, x    -- swap
```

Expressions do not perform any side-effecting on variables and their values. This means assignments and boolean conditions do not perform side-effects. Thus, only assignment can change the state of a program, just the evaluation of an expression along can not.

Communication

The communication processes perform the two halfs of the assignment process. The output process performs the evaluation of expressions and if the computations are successful, communicates the values down the channel named in the process. The input process performs the reading of these values from the channel into its matched list of variables.

A process

A process in occam is either an assignment process, a communication process or a collection of these processes combined. Processes are grouped by `SEQ`, `PAR`, `WHILE`, `IF` and `ALT`.

The first two of these combine a list of processes into one process. `WHILE` takes a boolean expression and a process that it executes while the boolean expression is `TRUE`. `IF` takes a list of processes guarded by boolean expressions and executes the first process in the list whose guard evaluates to `TRUE`, otherwise the `IF` `STOP`s.

ALT is similar to IF, its processes are guarded by a communication, an optional boolean condition, or both. The process executing the ALT is suspended until one or more of the guarded processes become ready. One ready guards is then selected arbitrarily for execution. An empty ALT is equivalent to STOP.

PRI ALT is a variation of ALT. When more than one guard is ready PRI ALT always selects the guard with the highest priority for execution, i.e. the guard highest in the list of guarded processes.

For reasons of efficiency the implementation of occam only allows one type of communication process in an ALT. Only input guards are allowed it being more natural to stop waiting for a message to arrive than to decided to stop trying to send a message.

Over time there is no guarantee that ALTs are fair. Achieving fairness would require implementations maintain a state over time. Doing this would increase the time taken to execute this process that already takes a long time to execute. If such fairness is required it has to be implemented by hand.

Replicating

The method of repeating instructions is much more general in occam than in other languages. This is performed by replication of a process. The replicator in occam takes a process as a parameter. The advantage of replication is that any construct can take a single process may be replicated. Thus, as occam's replicator is a construct modifier, instead of a construct in itself, such as the for loop, this means constructs such as IF can be replicated as well as constructs like SEQ. A replicator consists of an index variable, this is declared automatically to be of type INT and an expression. This evaluates to a value of type INT and gives the number of times the process is to be replicated. This expression is evaluated once on entry to the replicator. One advantage of having replication is that constructs such as IF can be replicated on their own. For example,

```
IF
  IF i = 0 FOR SIZE value
    value[i] = entry
      found, index := TRUE, i
  TRUE
    found := FALSE
```

which is equivalent to,

```
IF
  value[0] = entry
    found, index := TRUE, 0
  value[1] = entry
    found, index := TRUE, 1
          .
          .
          .
  value[(SIZE value) - 1] = entry
    found, index := TRUE, (SIZE value) - 1
  TRUE
    found := FALSE
```

This alleviates the need for a for loop, such as in older languages. This is a good example of how, although occam is a smaller language, it is a great deal more expressive than other third generation programming languages.

Another common use for replicators, as well as being used to generate for-loops, is to access all of the elements in an array.

There is an equivalence between an occam WHILE loop,

```
INT index:
SEQ
  index := expr
  WHILE index < expression
    SEQ
      ... body
      index := index + 1
```

where index does not appear on the left-hand side in the loop's body, and a replicated SEQ,

```
SEQ index := expr FOR expression - expr
  ... body
```

This is independent of the body of the loop. Here this body is represented by an ellipsis (...). This notation comes from the use of folding editors. Folds can be used to perform textual abstractions on parts of a program.

2.2.4 Semantics

The semantics of occam have already been mentioned. These are rigorous due to being formally defined. One method in which this has been achieved is with equivalence laws. These have the advantage of giving a semantics that is algebraic in nature. This allows for any program to be changed mechanically into others that perform the same task (have the same semantic meaning), but have different performance characteristics.

In [RH86] Roscoe and Hoare uncover, for all of the occam constructs, both the laws that govern the constructs and present enough laws to translate finite programs (programs without WHILE loops) into normal form. These laws cover: declaration, assignment, SEQ, IF, PAR, ALT and $\perp$. The last are divergent processes, these are equivalent to,

```
WHILE TRUE
  SKIP
```

From these it is also possible to derive other laws.

Of these basic laws, we look at one here, *input*, as an example, as it shall be used later,

```
ALT
  c ? x  ≡  c ? x
    SKIP
```

It is also possible to give an equivalence between assignment, the most fundamental primitive of all programming, and the fundamental idea behind parallel programming, two parallel processes communicating,

```
                                  CHAN OF Type channel:
                                  PAR
variable := expression   ≡          channel ! expression
                                    channel ? variable
```

2.2.5 Compilation error modes

An occam program can be compiled in one of three compilation modes: HALT, STOP and REDUCED. The first two perform run-time checks such as checking that array indexes are in range. Should an error occur in HALT mode, a processor HALT instruction is compiled into the program and the processor that executes the instruction halts. When compiling in STOP mode the compiler inserts extra instructions throughout the code in order to stop only the process that is in error, instead of stopping the whole processor. In REDUCED mode no checking is performed.

2.3 Efficiency considerations

This section looks at the contents of the Advanced Transputer Engineering Workshop [Roe]. This work and course is due to Roebbers.

The course is about run-time efficiency and optimisation. The five parts it covered are listed below.

1. A study of the method used by the processor to execute instructions and how then next instruction is selected for execution.

2. The mechanics of how communications are performed across channels and across links.

3. How to write efficient code that process arrays of any size.

4. How to use the post-mortem analyser to explore a network of transputers, look at what code the compiler produces and several other tips that make debugging easier.

5. How to perform transformations on a piece of code in order to make it execute more efficiently. This consisted of transformating all the `PAR`s on each processor into `SEQ`s, attempting to minimise the amount of memory used by arrays, and finally adding buffering to the sequential process to engage the links in parallel. These optimisation transformations also included transformations similar to those used in sequential programming, such as loop unrolling. Due to the algebraic semantics of occam the transformations performed here are more reliable than equivalent transformations performed in other programming languages.

The last item has been performed by many others in occam, the essence of the work coming from the design of occam. But it is Roebbers's approach to this work of rigorous optimisation that is the most developed, complete and practical in real programming situations seen here.

The side of the course dealing with arrays involved several ideas. When writing library routines to manipulate arrays, instead of passing in the size of the array it is easier to use the unary operator `SIZE` instead. For example, `SIZE A` gives the size of the array `A` and `SIZE A[0]` gives the size of the second largest dimension of `A`. Abbreviations are of great use in cutting down the number of accesses to multidimensional arrays by retyping them to a one dimensional array and in abbreviating frequently used array elements into single variable constants. The optimum amount of loop unrolling for the transputer is also looked at on the course. The optimum number of unrolls being sixteen; due the transputer's instruction set encoding constants into four bits. It is this rigour and attention to detail, as used by Roebbers, that has not been seen elsewhere by this author either in the optimisation of programs in other languages or in other programming disciplines.

The largest part of the course consisted of fine-tuning a Fast Fourier Transform program. Most of the stages in this process involved performing some transformations on a program. We have already mentioned algebraic semantics are useful for reliably and rigorously transforming programs into others. But, mentioning such techniques in general terms does not show how they should be used. Here, we are interested in showing the use of these technique in practice, the transformation being towards programs that possess different and preferably better efficiency characteristics. For example a program that is faster, but may use just a little more memory.

As well as allowing performance enhancing changes to be made to programs, another beauty of the algebraic semantics is they are simple to use in practice. Indeed the rigorous transforming of programs into other more efficient programs can be performed mentally. Due to the simple nature of occam's semantics, any part of even very large programs can be reasoned about and transformed informally, though still thoroughly and rigorously. This is akin to transforming a mathematical expression into another, that is identical, but easier to calculate.

The Fast Fourier Transform program was initially written in the process oriented method. The "butterfly" processes of the F.F.T. was drawn and then described in occam directly. The calculation was then inserted into the butterfly process.

If this program is to run on multiple transputers with more than one butterfly on each processor, multiplexors must be added to multiplex the many channels over the four links of the transputer. As the transputer is entirely sequential and cannot execute multiple butterflies simultaneously, the program was

transformed so the internal parallelism and associated overheads were removed. The task is performed by observing the data flow within the parallel processes on the transputers. Communication is then changed into assignment and arrays of channels are changed into arrays of variables. This results in a completely sequential program that will run faster on a single transputer. Also, as is seen in the F.F.T. example, once the program has been sequentialised, some of the arrays are redundant. There is no longer the potential for any part of the calculation to happen at any time and in any order. Removing this redundancy saved a large amount of memory in this example.

A completely sequential program had now been achieved. This is suitable for running on any sequential processor. However, the transputer can also perform link communications in parallel with computation. Thus, the program can be further transformed. The transformation here being to turn code from its sequential form,

```
PROC process.seq (CHAN OF Type in, out)
  ... variables
  WHILE TRUE
    SEQ
      in ? data
      procedure (data, packet)
      out ! packet
:
```

into a parallel form,

```
PROC process.par (CHAN OF Type in, out)
  ... variables
  SEQ
    in ? data0
    PAR
      in ? data1
      procedure (data0, packet0)
    WHILE TRUE
      SEQ
        PAR
          in ? data0
          out ! packet0
          procedure (data1, packet1)
        PAR
          in ? data1
          out ! packet1
          procedure (data0, packet0)
:
```

Even the most elaborate array indexing optimisations have only as many as 3 or 4 PAR constructs in the WHILE loop once parallelised.

Notice all communications are set up before any computation is started. This works on the current compilers as these processes are executed in the order they are listed in the program. Nevertheless, in general one should write,

```
PRI PAR
  PAR
    in ? data1
    out ! packet1
  process (data0, packet0)
```

though this may result in a slightly larger program.

Another example used more in this work is the parallelisation of code that also involve some subtle changes to the program's behaviour. If both the channels in the following process were across links and

it was desired for this process to obtain maximum parallelism, the process could be transformed from the following,

```
PROC buffer.seq (CHAN OF PACKET in, out)
  ... variables
  WHILE TRUE
    SEQ
      in ? packet
      out ! packet
:
```

into a parallel form,

```
PROC buffer.par (CHAN OF PACKET in, out)
  ... variables
  SEQ
    in ? packet0
    WHILE TRUE
      SEQ
        PAR
          out ! packet0
          in ? packet1
        PAR
          out ! packet1
          in ? packet0
:
```

As another example, this result collecting process from a farming harness,

```
PROC collect.results (CHAN OF PACKET local, in, out)
  ... variables
  WHILE TRUE
    PRI ALT
      local ? packet
        out ! packet
      in ? packet
        out ! packet
:
```

could be parallelised into,

```
PROC collect.results.par (CHAN OF PACKET local, in, out)
  ... variables
  SEQ
   {{{ get packet0
    PRI ALT
      local ? packet0
        SKIP
      in ? packet0
        SKIP
    }}}
    WHILE TRUE
      SEQ
        PAR
          out ! packet0
          ... get packet1
        PAR
          out ! packet1
          ... get packet0
:
```

The parallel implementations here have similar, but not identical behaviour. In the parallel version there is the potential to perform a second input before the first output. In some situations this second process would prevent deadlocking behaviour the first process would not.

As mentioned earlier, this method of optimisation not only contained rigorous theory, but is also the most usable, methodical and practical for every day use. This usability and practicality reveals itself in the following two ways,

1. which particular transformation is appropriate in the particular situation at hand always seems obvious, and,
2. all of the transformations are very simple to perform.

Thus in general, it is simple to see where to apply the theory in any situation.

2.4 The processor farm

In this section what was considered to be good farming practice is looked at in more detail.

To recap, farming out consists of a farmer handing out work to a number of workers, these items of work are independent of one another and results are either passed to a harvester or are passed back to the farmer. The length of time it takes to process each job can vary without a loss of performance being incurred. The topology used to interconnect the processors is largely irrelevant, though it can affect the performance of the farm. Further to this, the processor farm possesses a number of other clear advantages.

As the jobs of work are completely independent of one another, and are performed on separate processors the work load is balanced out completely automatically by the nature of the design. On algorithmic and geometric distributions this balancing is achieved by hand. In essence the processor farm has a very simple design and is very effective at performing tasks. Further, it is also very easily scalable. The scalability of a processor farm is bound by the amount of parallelism in the decomposition of the application and the amount of bandwidth in the implementation. This again contrasts with the other two distributions where the number of processors used is likely to be very much tied into the shape of the application.

It is generally agreed in transputer folklore that only compute bound applications will farm well, and those that are communication bound will not. However, there appears to be no method that will determine whether an application is compute bound or communication bound. It is believed that application's are communication bound because of a communication bottleneck at the farmer and the harvester.

2.5 Naming conventions

In this thesis, processes are called after the real domain from which they come. Thus, in the processor farm, these names are taken from agriculture, giving us a farmer, some workers and a harvester. This agricultural analogy is used deliberately, not only for labelling, but also for thinking purposes. This reuse of nomenclature is done in the hope that with it we will also obtain all of the tried and tested efficient methods that have been developed in this field of work.

One of the reasons for not using the other names that are used in transputing community is these other types of thinking and labelling schemes are often hierarchical in nature (master slave etc.), and here an attempt is made to depart from hierarchical models and a move towards models in which the components are equals.

On a similar note, it is prefered to call the tasks in the farming harness "job distribution" and "result collecting" instead of demultiplexing and multiplexing as some others have. The sources of the result packets are not retained by the processes of the harness, as would be the case in multiplexing.

2.5.1 Diagrammatic conventions

A few diagrammatic conventions are used in this thesis. In simple diagrams no variation of line thickness is necessary. In more complex diagrams the following conventions are used. Communication channels are drawn using thin lines and low priority processes are drawn with boxes constructed from thin lines. Links and high priority processes are drawn with lines of medium thickness. Processors are drawn with boxes constructed from thick lines. See figure 1.

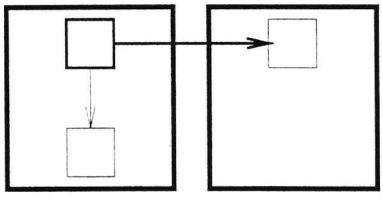


Figure 1: Drawing style used in this thesis

2.6 Hardware: Kent's transputer system

The transputer was designed as a communicating computer on a single silicon chip. Thus, transputers have a central processor, some memory (making it a very good embedded system processor) and a number of communication links. The name transputer comes from *trans*istor com*puter*. The transputers used for this research are Inmos T800s. In addition to the four kilobytes of on-chip memory, about half have four megabytes of external memory, a third have 256 kilobytes and the remainder possessing either one or two megabytes. The instruction set of the T800 was designed to execute occam efficiently.

So all of the later communication bound results can be put into context, it is worth looking in detail at what is known of the internal architecture of the system used here.

In Kent's MEiKO system the transputers are connected together using MEiKO's own general purpose message routing devices, commonly termed *switch-chips*. We use *wiring files* to connect our transputers into the topology of our choice. There are four switch-chips per board. These boards are in turn, connected to a forty-slot backplane. There being two rows of twenty boards on one backplane. The way these boards are connected is by both vertical and horizontal connection. The horizontal connections are between alternate boards, not adjacent boards.

Due to the synchronous method of communications used by the transputers' link engines, the sending of packets and acknowledgements is delayed, thus introducing latency into the formula for communications and thus the switch-chips reduce the bandwidth of communications. For every switch-chip through which a message passes there is a loss of bandwidth incurred. It's believed this loss of bandwidth is 5% per switch-chip [W+89].

2.7 UNITY

So far we have looked at the background theory behind occam and the transputer. These have been the standard theoretical areas have been looked at and used for several transputer research projects, especially at Kent. However, in this work programs have also used the UNITY theory of programming.

Despite the wide range of differences to be found in applications and more recently in architectures too, the UNITY philosophy is that there are more similarities to the task of programming than there are differences. Thus programming should be viewed as a single discipline. In UNITY, all programs are developed in a similar manner. Program designs are then mapped onto any architecture for execution.

Program development in UNITY is based on a small theoretical foundation. This foundation consists of a small computation model and an associated proof system. The UNITY proof system is rich and expressive and yet also flexible and abstract enough to enable the development of programs for all forms of parallel architecture. Thus the standpoint taken by UNITY is very much the complete opposite to the current general mainstream view of programming. One example is the existence of a foundation, opposed to many closely related fragmented disciplines. Here these differences are looked at as are the reasoning behind the choices made.

2.7.1 How UNITY is used here?

In this work UNITY's execution model and program notation is used. Using the execution model one can consider how programs are executed. Using the program notation one can construct and examine the logical structures found in programs. The execution of programs on the following architectures is considered: the UNITY execution model, a conventional Von-Neuman machine and a processor farm. As program development is not performed in this thesis the proof system is not mentioned in detail.

In this section the choices made by the theory are looked at, the notation it uses (in the situations when it has to have a program notation), the UNITY execution model and how programs are mapped onto architectures.

2.7.2 Philosophy

In [CM87] the goal is to show how programs can be developed systematically for a variety of architectures and applications.

As we are aware of the semantic preserving program transformations in occam, it is easier to envisage two computers could be performing the same application, even though they are running completely different programs. This is something not thought possible until UNITY was learnt. Why this was not considered before is interesting. This could be due to architectures shaping our programming decisions and even our bugs, i.e. the different architectures could influence the decisions we make when trying to implement the same application, thus resulting in different types of error on different systems.

The previous paragraph started discussing the application, progressed to discussing programs and finally discussed architectures. It is this order we consider things in UNITY. The application first, the program second and the architecture last. The application is our primary focus and goal.

2.7.3 Choices

Foundation over taxonomy

Language shapes our thoughts, and in science we have the choice of language and of paradigm. In Physics and Chemistry there are fundamental laws. However, in terms of plant life, so much exists, Botanists classify what exists through observation. In computing there are an increasingly large number of categories of both application programs (databases, word processors, operating systems), languages to write them in (procedural, object oriented, functional, parallel) and architectures to run them on (message-passing MIMD, shared memory MIMD, SISD and SIMD). Although the taxonomy approach has its merits and its uses, UNITY has many advantages in providing a theory with which to develop any program for any

architecture. The task of programming being identical for all the applications and architectures, not different in different situations. Thus UNITY attempts to be a small foundation theory for computing that transcends taxonomy.

Choice of foundation

In Physics the fundamental laws are those from which all others are derived. Here we are not just interested in studying computation and programming, for which a model such as the Turing machine is sufficient. Here we are interested in program development and desire a foundation theory that aids us in this.

Here Chandy and Misra have proposed a small rich and expressive theory that can develop any application program for any architecture. An alternative is to have a theory for each part of our taxonomy, a methodology ideal for building databases and for message-passing computer systems. Such specific theories might yield elegant solutions for their particular domain, but for those particular domains only. Such specific theories would not easily be applicable to other areas. At the risk of paying this price we give ourselves the advantage of having a single unified framework in which to build anything.

Design versus coding

The choice here is between proving programs correct or developing programs correct. The first consists of writing a program and then verifying that it meets its specification. The second is interested in developing a program from a specification.

The first is interested in programming and proofs of program texts. The second is interested in the stepwise development of designs for programs that are then implemented. As ultimately we are interested in performing some action in reality with the aid of our application, we choose the latter.

Formal and informal descriptions of programs

Informal reasoning is useful in helping us to reason about programs, for example philosophers sitting around a dining table. With such analogies there is no formal notation that one needs to learn and then learn to see beyond. Nevertheless, such reasoning cannot be checked in a rigorous way and there are several decades of evidence to show that programmers are fallible. In complete contrast to this, a mathematical notation is checkable. The small simple unifying framework of UNITY limits and thus restricts what and how much notation can be used. As both forms of reasoning have their uses, both forms are used when useful and appropriate.

Operational and non-operational reasoning about programs

It is possible to reason about programs in two ways. The first is reasoning about the computation as it unfolds. The second is reasoning about the static properties of a program, the things that are always true. In UNITY the static view is used. It is easier for us to deal with constants. Operational based reasoning has its value and provides insight in algorithm development. It often being based on operational and even anthropomorphic reasoning. Dijkstra is strongly against the use of such anthropomorphism [Dij89a]. Yet his work is full of anthropomorphic inspiration: railway analogies (semaphores [Gen65] and others [Dij89b]), Elephants built from Mosquitos humming in harmony [Dij89c] and a parallel partition (see quicksort [Hoa61]) inspired by the Dutch National Flag [Dij82]. Nevertheless, when using operational reasoning to prove programs correct,

1. we make many more mistakes, often by overlooking certain sequences,
2. the reasoning argument is much longer,
3. it is often harder to convince others of the correctness of an algorithm using operational reasoning.

Two other concerns that perhaps should be separate in programming are inspiration and perspiration. The inspiration of an idea for a design and the perspiration from constructing that design.

Separating proofs from program text

An early advocate of using assertions for proving that a program is correct was Alan Turing. At a conference in Cambridge (24 June 1950) Turing gave a short talk on "Checking a Large Routine" [Hoa80, HJ89b].

> How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.

Although this statement is based on sequential and operational thinking, performing checking with algebraic assertions does work. It has now been realised it is much quicker to work with the properties of the algorithm. Partly as these deal with all aspects of the algorithm, rather than just particular states at particular points. In UNITY we are also interested in abstracting away from the program text and proving the design of our program correct, not the actual text of a programming language itself.

Separating correctness from complexity

UNITY is interested in separating a program from its implementation. One program may be implemented on a number of different architectures. The correctness of a program is independent of its implementation. A program design is implemented via a mapping. Discussing complexity measures is only valid of a program and a mapping. UNITY has a logical proof system for discussing a program's correctness and the concept of mapping for implementing programs.

States, assignments and state-transition systems

State transition systems are used in many areas related to computing. State transition systems have a good methodology and thus are useful. Nevertheless, the semantics of the problem is lost once represented in terms of state transitions. Thus, in UNITY we choose to use the clean expressive form of assignment to represent the state changes of our programs.

We are not interested in architectures during program development. Thus, although the von-Neuman one-word-at-a-time bottleneck probably exists in all present computer architectures, it is not of interest here. UNITY allows us to perform complex assignments. Such an assignment could be performed as a sequence of assignments or possibly by several processors in unison.

Control flow and Determinism

Two concepts UNITY does not consider fundamental to programming are control flow and determinism.

Consider a program that consists of two operations working on a stream of data. The program could be fashioned in one of two ways: either as loop containing the two operations, or two co-routines both of which have a loop containing one operation. The execution of these co-routines may be interleaved in any arbitrary fashion. What is important is the flow of data, not the flow of control of execution. Control flow is not an concept essential to programming. Further as programs may be expressed much more elegantly without control flow it is not included in UNITY.

The same is true of determinism. A program executed twice on the same machine may consume different resources and may even produce different results. Some systems are inherently nondeterministic and we wish to be able to design such systems in our formalism. Thus, nondeterminism is included in UNITY. Nondeterminism is also useful as it allows us to not restrict ourselves by specifying excess detail.

With UNITY it has been found that a programming model based on assignment, but not control flow, possesses good properties.

Synchrony and asynchrony

Some computer systems are synchronous, for example a systolic array has a common clock. At the other extreme, computer networks spanning the globe are asynchronous. A complete theory of computing should be able to discuss both, and without bias. In trying to be a unified theory UNITY has both.

2.7.4 The name UNITY

The name UNITY stems from the way in which Chandy and Misra choose to view programs, that of Unbounded Nondeterministic Iterative Transformations.

2.7.5 Execution model

The UNITY execution model consists of selecting a statement in the program and executing it. This process is repeated indefinitely. The statements are selected for execution completely nondeterministically, though the selection process is fair over time — in the sense that every statement is selected infinitely often. Thus, after one assignment statement has been executed, another assignment statement is selected for execution. This second statement may be the same statement, it may not be, this is not relevant. What is important is that the execution is fair, if only over an infinite period of time.

Although our UNITY programs, once run, continue to execute indefinitely, they may reach a point from which no further change occurs, this is termed a fixed point.

This execution model is designed for program development and is clearly not practical as an efficient method for executing programs. The model is interesting as all programs only consist of statements that make progress to some degree or at worst keep the program in the same state. Any program containing a statement that undoes the progress made so far will prevent, potentially, the program from ever reaching a fixed point. It is precisely because the execution model is not restricted by implementation issues in anyway that UNITY and its execution model is an unrestrictive and thus powerful program development tool.

If R represents one of the possible *execution sequences* of a program and $R_0.state$ is the initial state. Variables that are not initialised in the *initially-section* (see subsection 2.7.6, page 22) may take on different initial arbitrary values on different runs of the program. $R_i.label$ is the i'th statement to be selected for execution. Thus, given any statement, $R_i.label$, and a state, $R_i.state$, to execute it in, $R_{i+1}.state$ is the uniquely defined state after execution of that statement. All assignments are deterministic and terminate in a finite amount of time.

2.7.6 Notation

In order to give examples of the fact that it is possible for all applications to be developed uniformly for all architectures, UNITY possesses a notation for programs. Nevertheless, UNITY is not about a programming language and this notation pays no attention to abstraction mechanisms or data structures.

The notation is based on the syntax for Pascal. UNITY programs have the concept of a logical block, variable declarations, even a name for the program, a "program" keyword and a list of sections.

Program structure

A UNITY program may consist of up to four sections. These are laid out as follows,

Program name
  declare *declarations*
  always *equations*
  initially *equations*
  assign *assignments*
end

The last three sections consist of the same type of constructs.

Assignment statement

UNITY programs are only about assignment. Thus UNITY naturally has the basic assignment statement,

x := expr

The type of the expression and the variable must be identical.

UNITY also has multiple assignment. UNITY has two forms for multiple assignment. The traditional syntax with a list of variables and expressions with the elements of the lists being separated by commas,

x, y := y, x

and a new notation consisting of composing other assignment statements with the lexical symbol for parallelism,

```
  i := 1
|| total := 0
```

The new syntax greatly increases the readability of assignments involving components which are unrelated logically, but where nothing prevents the different components from being executed at the same time. Where the components are related logically, as in the case of swapping, the traditional syntax is invariably the more readable.

A variable may appear more than once on the left side, however the value assigned must always be the same.

As UNITY is based solely around assignment there are no constructs for performing selection. That said, within the right-hand side of an assignment UNITY permits selection through the use of boolean expressions. This is often known as case analysis. The $\sim$ symbol is used in the syntax as a token to separate the alternative expressions from one another.

```
x := -1  if  y < 0  ~
      0  if  y = 0  ~
      1  if  y > 0
```

This formatting is arbitrary, this expression could equally have been written on one line.

So that assignments are deterministic, if more than one boolean expression is true, the appropriate expressions should all have the same value (this must be must guaranteed by the programmer). If none of the boolean expressions are true, the value is not changed.

Care should be taken when combining ifs with parallel assignment as the ifs binds tighter than a parallel composition of assignment-components, but bind looser than in a multiple assignment.

Composition

Composition of separate assignment statements into a block or list of assignments is denoted by a [] symbol.

Note that although the infix symbols [] and || are visually similar and they both perform statement composition, the operations performed are different. A [] is used to join statements together that are to be regarded as entirely separate. A || joins the two component statements into one statement that is to be performed as a single operation (although the statement is best read as two separate statements). The operation performed by || is synchronous, the operation performed by [] is asynchronous.

Quantification

UNITY possesses a powerful construct named *quantification*. As it is even more expressive and useful than replication in occam, quantification is used for generating an even wider range of objects. We introduce it here through its use in generating assignments.

Quantified assignment

A *quantified-assignment* consists of an assignment, prefixed by a quantification. This consisting of a list of variables and a boolean expression.

$i : 0 \leq i < N ::$

These bound or quantified variables are local and thus are only in scope within the quantification. The scope is delimited with a pair of obtusely angled brackets.

$\langle \| \, i : 0 \leq i < N :: A[i] := 0 \, \rangle$

This allows quantified-assignments to be used among and within other constructs.

The *instance* of a quantification is defined by Chandy and Misra as a set of values of the bound variables that satisfies the boolean expression in the quantification. An instance of a quantification must always be *finite* and may be *empty*. One reason for this finiteness is the fairness of execution rule. The execution model can not fairly execute an infinite number of statements infinitely often. For the same reason the number of statements must also constant throughout the running of a program.

Thus a quantified-assignment denotes zero or more assignment-components. These are obtained by replacing the bound variables in the assignment by their instances. Thus the above quantified-assignment is equivalent to,

$A[0] := 0 \parallel A[1] := 0 \parallel \ldots \parallel A[N\text{-}1] := 0$

As can be seen here the use of a boolean expression gives greater flexibility over the usual contiguous range of values obtained with a replicator or a `for` loop. For example,

$$\langle \| \, i, j : 0 \leq i, j < N :: Id[i, j] := \begin{array}{ll} 0 & \text{if } i \neg = j \\ 1 & \text{if } i = j \end{array} \sim \rangle$$

can be also written as,

$$\langle \| \, i : 0 \leq i < N :: Id[i, i] := 1 \parallel \langle \| \, j : 0 \leq j < N \wedge i \neg = j :: Id[i, j] := 0 \rangle \rangle$$

The symbol on the far left in a quantification dictates the style of assignment produced. If the parallel assignment symbol, ||, is used one multiple assignment statement is formed. If the composed assignment symbol, [], is used several separate assignment statements are formed.

Any undefined variables are considered to be free. For convenience we omit boolean expressions that would only re-state array bounds.

The other uses of quantification

In UNITY's programming notation the other uses of quantification include quantified expressions. These use binary, associative and commutative operators and functions that combine two elements of the same type. Thus, quantification can apply any operator or funcation to a whole array.

Quantified expressions are formed by supplying an operator or a function instead of a statement composing symbol. Thus, quantifications can be used in the following way to perform both expression evaluation and to denote specific values. Most operators can be used. Here are some examples, the first is our UNITY programming notation equivalent for,

$$\sum_{i=0}^{N-1} A_i$$

total := $\langle + \, i : 0 \leq i < N :: A[i] \rangle$ { Sum of array. }

biggest := $\langle \max \, i :: A[i] \rangle$ { Largest value in A. }

is.sorted := $\langle \wedge \, i : 0 \leq i < N :: A[i] < A[i+1] \rangle$ { Is sorted? }

count := $\langle + \, i : 0 \leq i < N \wedge A[i] > A[j] :: 1 \rangle$ { Number of elements larger than A[j]. }

$\langle [] \, i :: A[i] := 0 \rangle$ { Initialise array. }

Functions must have unit elements (min()'s is ∞), operators must be binary, associative and commutative, i.e. the order of application of operators must be irrelevant.

Quantification is also used within the UNITY logic to specify properties of a program by quantifying Hoare logics (see [Hoa69] and [HJ89c]) over the program's statements. This is how UNITY obtains the nonoperationality of program properties from an execution model that obeys a fairness rule defined in terms of the execution of statements.

The assign-section

The assign-section of a UNITY program consists of a *non-zero, static* and *finite* set of assignment statements. A requirement of the UNITY logic is that a program contains a minimum of one statement. As has been said, UNITY only deals with static programs, as programs with a dynamic number of statements would complicate the fairness of the execution model.

The initially-section

The *initially*-section specifies the initial conditions, the strongest predicate that holds prior to execution. This consists of a list of equations that for our convenience are written with all the expressive facilities available to us in the assign-section. Equations are written with an equals sign opposed to an assignment symbol. In the initially-section a [] denotes a sequential ordering of the equations. In this author's opinion a semi-colon should be used here.

Some initial values are expressed in terms of other initial values that must be already defined. Thus the set of equations must be *proper*. This constraint consists of three parts.

1. All variables acquire unique values.
2. The equations are compilable. Namely that all quantification variables are either bound or are initialised earlier on.
3. The equations are well-defined, i.e. after quantification expansion, any variable appearing on the right-hand side or in a subscript, appears on the left-hand side earlier on.

The always-section

The *always*-section consists of a set of equations that always hold (invariants). This section is useful for defining program variables in terms of others. The variables on the left-hand side of the equations are *transparent* variables, the name coming from them being referentially transparent. Transparent variables are functions of nontransparent variables. They are "read only", i.e. they may not be assigned to or initialised, though they may appear on the right hand side of any expression. Thus allowing transparent variables to be used as macros. Transparent variables must be proper and conform to the same constraints as the variables in the initially-section.

The always-section is not needed for writing programs, however it is advantageous in having a set of invariant equations that are easy to reason about.

2.7.7 Mapping

Once a UNITY program is developed, it is then mapped onto the particular architecture, or architectures, one is interested in running the program on. This is the second of the two stage process of program development in UNITY. It is this mapping stage that includes all the architecture specific aspects of implementation. Optimisation being an example. Developing a mapping should be a mechanical process, all the creative design work being performed in the development of the program design.

In [CM87] Chandy and Misra give some hints and ideas for the mapping of UNITY programs on to real architectures, including electronic circuits. For distributed systems they recommend that a variable be mapped either onto the local memory of one processor or onto a channel. The channel naming two processors. It is also assumed that all channels possess some buffering. Synchronous communication is not considered.

2.7.8 UNITY summary

UNITY is the work of Chandy and Misra. They suggest the act of programming is identical for all applications and architectures. With the proliferation of isolated studies in the different areas of research and applications: database programming, vision systems, object oriented, neural networks and operating systems; and especially with the different programming metaphors of the different parallel architectures it appears various programming methodologies have popped up. The thesis of their work is there is a fundamental common task to programming which is a foundation for all programming work; and that all applications can be developed for all architectures in a uniform manner from this foundation.

Comparing UNITY with other theoretical computer science theories, there have been several formal theories developed that try to develop large correct programs, a failing of intuitive methods. Where UNITY is unique is that it addresses the fact that programs out live the architectures they are run on.

2.8 Summary

Six areas have been looked at in this chapter.

1. There are at present three methods in which to organise parallel programs. In UNITY these methods are termed execution strategies or mappings.

2. The Process Oriented approach to programming consists of describing all the aspects of a program directly in separate communicating processes and running these in parallel. It is this directness of description that leads to all of the method's advantages of clarity and elegance.

3. Occam is a parallel programming language with an algebraic semantics designed to describe parallel and embedded systems succinctly and with precision.

4. The transputer is a communicating processor designed to execute occam efficiently. In the work here the transputers are connected via flexible, though slightly slow, electronic switches. It is understood through local folklore that the method in which these switch-chips are used to configure the machine could be greatly improved.

5. The algebraic semantics of occam can be used in a rigorous and practical discipline to transform occam programs into others. This being performed with a view to optimising performance.

6. UNITY is a theory that provides a foundation for programming. It contains an execution model and a proof system. In UNITY, applications are designed and then implemented, with formality aiding the first of these stages.

In the next chapter where these two theories agree and conflict is looked at, this is done respect to farming in particular.

Chapter 3

Origins of this study of farming

This chapter looks at programming from the perspective of UNITY's framework and considers how this impacts upon the purely process oriented approach used with occam. How to map UNITY programs onto transputer's execution strategies is also looked at. The additional insight provided by the first of these is used to look at farming. This includes looking at some of the work already performed in this area. The chapter closes with two questions to which there are no immediate answers.

3.1 The UNITY perspective and the process oriented model

This research was started by spending some time learning the various aspects of occam programming, transputer implementation and program optimisation. Then UNITY was discovered. This was then learnt over the period of the next few months. After having been engrossed in this theory for some time, attention returned to the programming of transputers, a discipline not practised while UNITY had been learnt. It was realised the UNITY approach had clarified understanding in a number of ways, and thus had brought to light a number of issues with respect to the way one should program transputers. Thus, the mapping of UNITY programs onto transputers execution strategies is looked at here.

Of the two approachs, UNITY is preferred. It focusing on application design before implementation issues. In contrast, the process oriented method focuses on constructing an implementation straight away.

In the remainder of this section looks at the insight the UNITY framework shed on four transputer programming issues.

3.1.1 Program efficiency

One aspect of the UNITY approach that looked appealing was its potential for arriving at implementations of a greater efficiency than the process oriented method. This being due to UNITY dealing with efficiency as an explicit part of the development process.

This contrasts with the process oriented method of programming where an application being built from a number of communicating processes. It is possible to use this method in a bottom up fashion, which may cause efficiency problems. If a program is constructed from components already in existence, there is a potential for inefficiency, as not all of these components would be designed with the requirements of the current application in mind. While the final program may have the correct emergent behaviour (i.e. it works correctly), it is not likely to be as efficient as a mechanism specifically derived for the purpose.

3.1.2 Execution strategies

The second stage of program development in UNITY is to map the program design onto an architecture. This is achieved by replacing UNITY's abstract execution model with a more concrete execution model or strategy. In the case of transputers these strategies would be: a processor farm, an algorithmic distribution, a geometric distribution or perhaps a combination of these strategies. From this a number of realisations were arrived at.

From this it became clear that what are called parallelisation techniques in the process oriented approach to programming are called execution strategies in UNITY terminology.

As in UNITY the execution strategy is chosen in the mapping stage, a single application may result in very different implementations on different architectures. Thus, it is possible for two very different programs to perform exactly the same task. Their differences being due to the programs running on different architectures.

Using UNITY's specification refinement approach one should build an application, then decide which paradigm will execute the application the most efficiently. What is obviously needed is a method by which to make this last decision. At present there does not appear to be a method to aid in the making of this decision.

Probably the most important realisation made in this work is that applications and execution strategies are independent of one another. As this is important, it is also worth restating this from both angles. An application is independent of the execution strategy used and an execution strategy is independent of any application that it may run. To a certain extent it seems both this author and the community in general were already aware of this, however using UNITY to study program execution has clarified this fact.

This independence implies that execution strategies may not be restricted to any particular range of applications.

3.1.3 Producer consumer model

One specific issue directly related to farming is that of how jobs should be distributed. Farming applications consist of supplying different data values to the same procedure. The best method of expressing a set of regular statements such as this in UNITY would use a quantification. Such a UNITY program could then be mapped onto any architecture. If it was mapped onto a conventional single processor machine, this repetition would probably be expressed with something like a `for` loop, the mechanics of which are simple: index values are generated by the top of the loop, one at a time, and are consumed by the body of the loop, one at a time. This is just the producer-consumer model. If the same application were to be mapped onto a processor farm, this should also be able to use the producer-consumer model. The work is generated by the farmer and executed by the workers. The regulation of the work is automatic. The farmer cannot give out any more work if there are no workers free to perform it and similarly the workers cannot perform work if it has not been given out.

That said, two farm designs seen in this work contain a mechanism to control the flow of work, instead of allowing the work to flow naturally as it is now realised we should. The first of these was the original harness used in the application that will be discussed in subsection 3.2.1 (page 30). This had a farmer that was aware of which workers had been given work and had not yet returned results and gave out work accordingly. This method was found to suffer from a bottleneck when the farm was large, the workers would sit dormant while waiting for their next job to be delivered by the harness. In the occam 2 reference manual [Inm88] a regulator process is also discussed. Some buffering systems have also been seen that hinder the efficiency, and are thus not suitable, for farms.

This is one example that shows, as the UNITY approach suggests, that one should build a farm from the point of view of getting the execution performed as efficiently as possible, as discussed in subsection 3.1.1, not from the point of view of just building up an implementation from a number of communicating processes that, when joined together, just happen to perform what is required.

The farming execution method of work distribution uses the client-server model of interaction. It is easy to analyse such a system to check that it is both deadlock and live-lock free. This is achieved by checking there are no client-server cycles and thus that a partial ordering of the processes exist (further details are in [WJW93]). For efficiency purposes we also use some link buffering in implementations of farms. These also use the client server model of interaction. Thus such buffering will not change the fact that a system is both free of deadlock and live-lock.

In summary, a farm automatically performs any work that exists in a balanced way. The only thing one needs to arrange in an implementation is for work to be passed out simply and efficiently.

3.1.4 Mapping work

The example in [CM87] where architectural considerations are discussed talks about synchronous processor arrays and shared memory systems. The decompositions suggested for these architectures involving N^2, N^3 and even N^4 processors, where N is the size of the problem.

This approach suffers from being inflexible; the amount of resources required being directly proportional to the size of the application. This may result in either not enough processors being available, or conversely, too many processors being available and a portion of the machine lying idle.

Distributed systems consist of processors that have memory and channels. In [CM87] Chandy and Misra suggest that each variable should be mapped onto either of these. While this is of course possible, and we know communications are equivalent to assignment, see subsection 2.2.4 (page 10), this gives us little information as to how a program might be mapped and organised onto this architecture.

We can get more information from looking at the architecture of transputers itself. It is known the transputer has at least three execution strategies. Applications can be mapped onto these. Our job as programmer is just to decide which execution strategy is the most appropriate.

It was from looking at the subject of implementation from this point of view, that it was realised one can decide what execution strategy is the most efficient to use. Previously this author would use the execution strategy suggested by the internal structure of the algorithm application's main algorithm.

The processor farm is the most efficient execution strategy at best. It is also the most flexible, the number of processors used is unrestricted by the size of the application or other such details. This allows us to use however many processors are available. Thus it seems sensible, when highly efficient and flexible implementations are required, to try mapping applications onto a farm before other execution strategies.

3.1.5 A mapping example

At the time this research started no mapping work had been seen. Here is given a full example of the mapping of a UNITY program onto both a single processor implementation and a transputer farm. This second program is then optimised for efficiency.

Here is the design of the UNITY program. This colours the area surrounding the Mandelbrot set of,

$$z \mapsto z^2 + c$$

for complex z and c. This colouring is performed according to the distance each point on the complex plane is from the attractor infinity, or at least a suitable approximation to it. The Mandelbrot set itself, termed M, contains all of the other attractors and for simplicity all of these are traditionally just coloured black [Man82, PR86].

The program fills the array, *screen*, with the natural numbers up to *Imax* (the number of iterations we are prepared to do) or *Imax + 1* (the set). Normally black appears as zero in palettes, however using zero here would cause this design to loop infinitely, due to UNITY's non-terminating execution model.

A translation is required between the area of the complex plane one is interested in viewing and the screen. The program needs to perform this translation from screen coordinates to a complex number. This is performed by the function *map()*.

Program ***Mandelbrot***
declare screen : array[0..WIDTH-1, 0..HEIGHT-1] of integer;
z : array[0..WIDTH-1, 0..HEIGHT-1] of complex;
initially ⟨|| x, y : 0 ≤ x < WIDTH ∧ 0 ≤ y < HEIGHT :: screen[x,y], z[x,y] = 0, 0+0i ⟩
assign ⟨|| x, y ::
screen[x,y], z[x,y] := screen[x,y] + 1, z[x,y]^2 + map(x,y)
if $|z[x,y]| < A$ ∧ screen[x,y] < Imax
|| screen[x,y] := Imax + 1 if $|z[x,y]| < A$ ∧ screen[x,y] = Imax
⟩

Note that at the program's fixed point all the instances of the "if" conditions are false, preventing the assignment statement from making further progress.

To map this program design directly onto a single computer, `for` loops could be used to execute the program instead of the UNITY execution model. Here is such a program. This has been written in occam for a transputer using replicated PARallels. Adapting this program for a conventional processor simply consists of changing the four replicated PARs to replicated SEQs or the for loop of the language used.

Traditionally screen memory maps the pixels along the screen's rows into adjacent memory locations. Due to modern caching, varying this dimension quicker than the other results in better performance. Thus here the WIDTH dimension of the array is declared second dimension.

```
PROC mandelbrot ()
  VAL Imax IS 4000:
  VAL A IS 10.0(REAL32):
  [HEIGHT][WIDTH]INT screen:
  [HEIGHT][WIDTH]REAL32 zr, zi:
  PAR y = 0 FOR HEIGHT
    PAR x = 0 FOR WIDTH
      screen[y][x], zr[y][x], zi[y][x] := 0, 0.0(REAL32), 0.0(REAL32)
  PAR y = 0 FOR HEIGHT
    PAR x = 0 FOR WIDTH
      SEQ
        WHILE (modulus (zr, zi) < A) AND (screen[y][x] < Imax)
          SEQ
            screen[y][x], zr, zi := screen[y][x] + 1,
                                    complex.sq (zr, zi)
            REAL32 cr, ci:
            SEQ
              cr, ci := map (x, y)
              zr, zi := zr + cr, zi + ci
        IF
          modulus (zr, zi) < A
            screen := Imax + 1
          TRUE
            SKIP
:
```

This program terminates when all pixels in the plane have had their distance from the attractor calculated.

The UNITY program design can also be mapped onto a network of transputers. Again this is done by replacing the UNITY execution model by the architecture's model of execution. In this case this is the message passing method of execution, with which there are a number of execution strategies to choice from. Farming being the strategy of execution chosen here.

With this program design the instances of the quantified assignment can be farmed out. If again, a direct mapping is performed, the following set of processes are obtained,

```
PROC farmer ([]CHAN OF BOOL req, []CHAN OF JOB job)
  SEQ y = 0 FOR HEIGHT
    SEQ x = 0 FOR WIDTH
      ALT i = 0 FOR SIZE req
        BOOL any:
        req[i] ? any
          job[i] ! x; y
:
```

```
PROC worker (CHAN OF BOOL req, CHAN OF JOB job,
             CHAN OF PRODUCE result)
  VAL Imax IS 4000:
  VAL A IS 10.0(REAL32):
  INT x, y, screen:
  REAL32 zr, zi:
  WHILE TRUE
    SEQ
      req ! TRUE
      job ? x; y
      screen := 0
      WHILE (modulus (zr, zi) < A) AND (screen < Imax)
        SEQ
          screen, zr, zi := screen + 1, complex.sq (zr, zi)
          REAL32 cr, ci:
          SEQ
            cr, ci := map (x, y)
            zr, zi := zr + cr, zi + ci
      IF
        modulus (zr, zi) < A
          screen := Imax + 1
        TRUE
          SKIP
      result ! x; y; screen
:

PROC harvester ([]CHAN OF PRODUCE result)
  [HEIGHT][WIDTH]INT screen:
  INT x, y, c:
  SEQ t = 0 FOR WIDTH * HEIGHT
    ALT i = 0 FOR result
      result[i] ? x; y; c
        screen[y][x] := c
:
```

These processes are then configured together in order to perform the work. The below configuration, for the sake of simplicity, ignores the need for a harness due to the transputers limited fanout problem,

```
VAL workers IS 2:

[workers]CHAN OF JOB job:
[workers]CHAN OF PRODUCE result:
PLACED PAR
  farmer (job)
  PLACED PAR w = 0 FOR workers
    worker (job[w], result[w])
  harvester (result)
```

By increasing the number of worker processors used, the time taken to execute the task should decrease.

Another method of speeding this program up is optimisation. In the case of this program the code for the worker can be optimised by quite a large amount.

```
PROC worker (CHAN OF BOOL req, CHAN OF JOB job,
             CHAN OF PRODUCE result)
  VAL Imax IS 4000:
  VAL A IS 10.0(REAL32):
  INT x, y, screen:
  REAL32 zr, zi, zr2, zi2, cr, ci:
  WHILE TRUE
    SEQ
      req ! TRUE
      job ? x; y
      screen := 0
      cr, ci := map (x, y)
      zr, zi := 0.0(REAL32), 0.0(REAL32)
      zr2, zi2 : = zr * zr, zi * zi
      WHILE ((zr2 + zi2) < (A * A)) AND (screen < Imax)
        SEQ
          screen, zr, zi := screen + 1,
           (zr2 - zi2) + cr, (MULBY2(zr * zi)) + zi
          zr2, zi2 : = zr * zr, zi * zi
      IF
        (zr2 + zi2) < (A * A)
          screen := Imax + 1
        TRUE
          SKIP
      result ! x; y; screen
:
```

Discussion

This mapping has been performed by translating the UNITY program directly into an occam program. Initially this contained all the parallelism inherent in the original design. This was then optimised using Roebbers's techniques as discussed in section 2.3. This included the use of some parts of the program being implemented using equivalent but faster sequences of instructions. Also, some parts of the program that were previously coupled in parallel were altered to be executed into sequence, leaving just the amount of parallelism that is available in the implementation.

It is interesting to note that occam can cope with almost all of the parallelism present in the UNITY design.

This mapping was performed by choosing of execution strategy. In this case mapping the elements of a quantification onto a processor farm. This mapping was performed, easily, and as suggested by Chandy and Misra, directly in one stage (i.e. without the use of any intermediate language).

In [Bro94] Brown introduces *UNITY Communication Language*, *UCL*, which is just such an intermediate language. With this two mapping stages are required; the first being from UNITY to *UCL*, the second being from *UCL* to the implementation language. Brown's reason for proposing *UCL* is that it possesses a more concrete parallelism and communication. However, the direct mapping performed here, from the more general form of parallelism in UNITY to the executable form of parallelism in occam was found to be very easy. Further, occam was designed so programs would be very easy to read and reason about, however, the occam in [Bro94] is very obscure and difficult to understand.

Finally, it is worth noting once more there is no application design performed during the mapping. The process is mechanical and should only be performed after the application has been designed completed.

3.2 Other current work within the transputer community

At this point in time another research project at Kent was just coming to completion. This was presented at a conference that this author also attended, the 14th World occam and Transputer User Group conference in Loughborough, England. As a number of implementations of interest were discussed at this conference this work will be discussed in context to the rest of the conference.

A number of applications were of interest, falling into three categories. The first consisted of the successfully farmed application from Kent just mentioned. The second was a method of implementation similar to farming that wasn't performing as efficiently as the application in the first category. The third consisted of some applications that had not been farmed out, but might benefit through being implemented using this execution strategy.

3.2.1 An application successfully farmed

The project at Kent had implemented a computationally intensive biological protein searching and matching application [SS91, Stu91]). The program was farmed and the final version included a very efficient farming harness. This harness was designed by Welch and developed within the process oriented philosophy [Wel88] and was designed to be as efficient as possible; making full use of all the parallelism available within the hardware of the transputer. The links were engaged in parallel by separate buffer processes. The job distribution mechanism consisted of just a simple `ALT`, no attempt to make any decision in software was made.

This harness was implemented by Sturrock who found that there were also some parts of the harness's design that could be fine-tuned to improve performance further.

3.2.2 Inappropriate topology

Phillips and Capon have developed a system to load balance an arbitrary collection of processes on a network of transputers [PC91]. This system involves the use of a communication harness similar to that used in farming. Further to the work in the paper, where the transputers were arranged as a pipeline, the work presented at the conference had the processors arranged in a torus.

It was realised that constructing an efficient farm using a toroidal topology would be difficult. This is due to the number of job sources varying from worker to worker depending upon the worker's location in the torus relative to the farmer, see figure 2, Here workers 1 and 2 receive work down one link, however

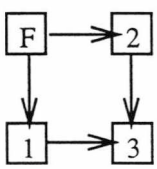


Figure 2: Message routes through a torus

worker 3 can receive work down two links. This processor would need a process that used an `ALT` just to obtain work, however the other two workers would not. Either the harness would need to consist of a number of similar processes or have one general process that in same cases would `ALT` over one channel. The first of these could be efficient, but would result in a large harness. The second would be less efficient but would require a smaller number of processes.

Thus it became clear a toroidally shaped farm is unlikely to work well. And ultimately that the shape of the topology and the complexity of the harness code ultimately influence the efficiency of the implementation.

In the case of toroidal topologies, the problems discussed above came about through using a topology not appropriate for the task in hand. Thus, toroidal topologies appeared to be inappropriate for farming. A toroidal topology has a completely uniform and symmetric shape. Nevertheless, the communication structure of a farm is not completely uniform and symmetric. It consisting of one to many to one communication structure, a shape a torus does not have.

So the problems encountered came about through using a topology not appropriate for the communication structure being implemented. This inspired an exploration of what topologies are appropriate for a farm.

It was realised there was a need for this clear understanding of what made topologies appropriate for farming, along with the other work performed here to be known about more widely.

3.2.3 Potentially farmable applications not farmed

There were a number of papers documenting applications that hadn't been farmed. From the nature of the applications, it occurred to me during the conference, they could have been implemented using a processor farm. From the figures mentioned in subsection 2.1.1 (pages 4–5) it appears farming has the potential to be more efficient in many situation than either algorithmic or geometric distributions. Thus farming out these applications, in a sensible way with a sensible granularity, might well result in a more efficient implementation for some of these applications.

This brought me to the conclusion that when one is implementing applications on transputers, farming should be the first execution strategy attempted out of the three when an efficient solution is desired.

3.2.4 Distant workers have priority

Jones and Goldsmith ran a workshop at the conference on the formal methods, developed at Oxford, that can be used with occam [JG91]. At this the subject of farming came up and whether the `ALT`s in the job distribution mechanism should give highest priority to the the on-chip worker or other workers. This issue generally seems to be considered important. At this workshop Jones said that when both the on-chip worker and other workers wanted more work at the same time, giving the work to the distant workers leads to a greater amount of overall parallelism.

3.2.5 Summary

From the work presented at the conference it was realised the process oriented approach needed to be disseminated further and also there were clarifications learnt through the use of UNITY that still needed to be clarified further.

By viewing farming as an execution strategy for UNITY programs, it should be possible to develop a coherent model of what makes a farm implementation efficient. This would allow quantitative predictions to be made about the performance of a particular application.

3.3 Questions

Looking at this other work raised two questions that were not immediately answerable. These two questions are looked at here.

3.3.1 What farming harness is the best?

A number of farming harnesses had been seen, all of which were designed to be good. With the harness designed by Welch being the more efficient so far. The argument for how it was the most efficient were based on how the transputer worked internally. Experimentation with an application had backed this up, finding only a few improvements that could be made. This led to a few questions:

1. Could this harness be improved upon further?
2. Was there an even more efficient harness that could be used? For example by using Roebbers's approach.
3. Could other applications be executed with the same level of efficiency as the protein sequence application?

So far Welch's harness had only been tested with one application. Before it could be reliably regarded as a highly efficient harness, it is natural that it should be used to farm out many more applications.

From studying the literature it was noticed all the papers documenting research on farming only used one application. This was true for both the majority of papers (these documented an implementation) and the most in-depth study of farming implementation details (based around a ray tracing program [PZ90]). Even though ray tracing and some other applications are flexible, and some are flexible enough to cover quite a wide range of both possible job run times and message sizes and even can allow for the grouping of jobs, any genuine application is not likely to be as flexible as we may well desire when exhaustively testing farming harnesses.

As execution strategies are independent of the applications they execute, it also follows that farming, and also farming harnesses, can be studied independently of any particular application. Thus such a study can be performed on a number of farming harnesses for any range of applications, simply by abstracting down to and focusing on the aspects of an application that the farming harness sees. Further, as a farm *is* independent of any application it executes, there should be an optimum way of implementing a processor farm and harness that should work as efficiently as possible. A good farming harness *should* be good at farming out any application that is of the type that farms out well. Whether this was true or not was something else to test for.

A farming harness is only aware of three aspects of an application,

1. the size of the job message,
2. the size of the result message, and,
3. the length of time it takes to process a job.

This last item can also be viewed as the length of time the harness sees in between each of the communications for each worker. Both sides of the harness see: communication, delay, communication, delay, communication and so on.

As these three sizes are the only aspects of the application of which a harness is aware, testing a number of farming harnesses with a suitable set of values for these parameters, it should be possible to discover which harness was the most efficient for any application.

Studying farming in this way will make it easier to arrive at a coherent model of farming, from which it is possible to decide if farming is a suitable execution strategy for an application and, if so, how to then implement it with a very high degree of efficiency.

From a coding point of view it was clearly possible to have a generalised worker process that accepted a message of any size, worked for a completely arbitrary length of time, as specified directly by the message and output a result the size of which was also specified in the original packet received.

As the farming harness is an execution strategy and not a part of the application, it has been realised here applications can be abstracted away from completely. Thus, a study of farming harnesses and their efficiencies can be performed without any restrictions being imposed by any particular application.

3.3.2 How much is farmable?

As has been said, all the applications that had been seen farmed out involved performing the same task for different values. The way such a structure would be expressed in UNITY would be with quantification.

If such UNITY programs of these application were mapped onto a conventional architecture, the quantification would be replaced by something like a `for` loop. In the case of transputers, these programs could be implemented on a farm. This begged the question: Is everything that is expressed with quantification farmable? This also led to the question: What other UNITY constructs are farmable? Or more generally: What other applications can be farmed out?

3.3.3 Summary

The fact that farming and the application are independent of one another provides two interesting results. Firstly, farming can be studied independent of any specific application (the performance of a farming harness being independent of the application). Secondly, farming might be able to execute a much wider range of applications than it has been currently. Both of these conclusions appeared to be original.

It was realised a model of farming could be put together, and it was decided to do this here. One of the harnesses at hand was initially highly efficient and some improvements had been made to this original design. Further, all of the previous work on farming had been performed in context to only one application. With the use of the UNITY execution model it had also been realised what applications could be sensibly mapped on a processor farm could be explored.

It was decided to look at the two areas in the order they have been mentioned here. First, finding the most efficient harness for the applications already known to be farmable. And second, looking at what range of applications is also farmable. It seemed sensible to perform the two studies in this order as the other applications that may be farmable may require a different type of implementation. And it may take some time before it's realised what type of harnesses (say) is the most appropriate for these types of application.

Thus these two issues are studied in the next two chapters. Chapter 4 evaluates a large number of farm implementations for their efficiency. Chapter 5 explores the range of applications that can be farmed out efficiently.

Chapter 4

Efficient farm implementation

This chapter documents the study to discover how to implement very efficient processor farms for the first series of INMOS transputers.

4.1 Overview

This chapter documents the development and performing of a number of experiments. This started out as a search to find the most efficient of six farming harnesses.

The following experiments were performed.

1. Measuring the throughput of the links.
2. Measuring the throughput of the six harnesses.
3. A study of the breakdown in efficiency of the six harnesses for the wide range of demands that applications can require.
4. A study of farm shut down.
5. A comparision of different topologies.
6. A look at the priority issues that arise in the coding of a harness.

Before looking at the experiments themselves we look at the harnesses to be tested and how this study was planned.

4.2 Which harness is the most efficient?

As discussed in subsection 3.1.2 (page 24) it has been realised that applications and farming are independent, and thus farming harnesses can be tested independently of an application. The fact a farming harness is independent of the application it executes implies that if a harness is efficient, it should be efficient for any compute bound implementation. As there is invariably a desire to want all applications to be as efficient as possible, here a search has been attempted to find the harness that is as efficient as possible for as many applications as possible.

4.2.1 Efficiency context

Processors have a set of instructions they can carry out. Each instruction taking the processor a known length of time to perform. In programming we are interested in getting processors to perform tasks that are not in this immediate vocabulary (instruction set). We are interested in more complex and intricate operations. We are used to building up these more complex tasks indirectly from a combination of the tasks

that can be performed directly. There are always many different ways of getting the processor to perform a given task, i.e. there are many different combinations of the processor's basic operations that result in it performing the desired task. We are interested, if not sometimes obsessed, in finding the combinations of operations that are the quickest for the processor to produce that result in the behaviour desired.

As farming is just a mechanism for execution, here we are interested in the harness performing a certain set of tasks in the minimum amount of time (overhead). Thus a good farming harness should be efficient and execute as quickly as possible, allowing the application to have the most access to the C.P.U. One way of looking at programming is to say it is about obtaining the desired emergent behaviour, with ideally the sum of the instruction run times being as small as possible. However, this obviously doesn't just mean using only the quickest instructions when writing a program. It may take many of the smallest instructions to perform the task in question. Where as using a few slow instructions, `ALT` for example, may perform the task quicker. Also, this is not the way in which applications are written, from the instruction timing sheet towards the application, but the other way round, from application towards the instructions. Although the run time of an implementation may be a parameter of the specification, unfortunately there is no apparent methodology to refine a specification into an implementation that has any particular run time or the quickest run time. The run time of an implementation is not a controllable parameter of the refinement process.

4.2.2 The breakdown of harness efficiency

As has been said, the area of interest here was finding the harness that was the most efficient for as many applications as possible. The question initially asked here being: is there a harness that farms out applications more efficiently than harness B? This question eventually became a more general one: which harness efficiently farms out the widest range of application mappings?

Having abstracted back to a parameterised view of applications, the approach to take, in order to find the most efficient harness, was to wind up or down the values of the parameters that are the only factors to affect farming harnesses. By studying the points at which the harnesses break down, the harness to break down the last will be the most efficient.

A harness's breakdown point is the point at which the farmer and the harness can no longer supply jobs at the rate needed to keep the workers constantly supplied with work. Clearly the more large jobs a harness can supply, the larger the number of mappings that harness will able to farm out efficiently due to it being compute bound rather than communication bound.

The breakdown of farming harness efficiency is the most important aspect of farming harness behaviour to study. It is easy to get caught into studying non-fundamental aspects of farming harness implementation. For example by just studying the code of the farming harness and considering simple alternatives, such as whether it should be the local or the distant worker that should have priority in a job distribution process. These issues are just fine tuning issues and should be dealt with last, once a good farming harness has been found. Thus this research set out to study the breakdowns of some well designed farming harnesses, before researching any other aspects that may constitute an efficient farm.

4.3 Harnesses

This section introduces the design details of the six harnesses studied. Two had already been used in applications and had shown to be efficient. The other four stem from ideas put forward by Roebbers and Welch that should improve these existing farming harnesses. For convenience throughout this thesis these will be referred to as harnesses A, B, C, D, E and F.

4.3.1 Harness A: this author's harness

The first harness was designed and written by this author for one particular application in which speed of execution was an issue. Being by the author, the design criteria and the ideas for why this harness was thought to be good are known.

This harness was designed with a minimalist approach. It was believed this would result in an application that executed the quickest. The harness should have as little communication buffering as possible

and contain as little code as possible. It was believed buffering would prevent a quick execution, jobs that filled up buffers far down the farm could not be performed by idle workers closer to the farmer. Being minimally buffered would prevent this. In order to have as little buffering as possible the harness would have to have as few processes as possible.

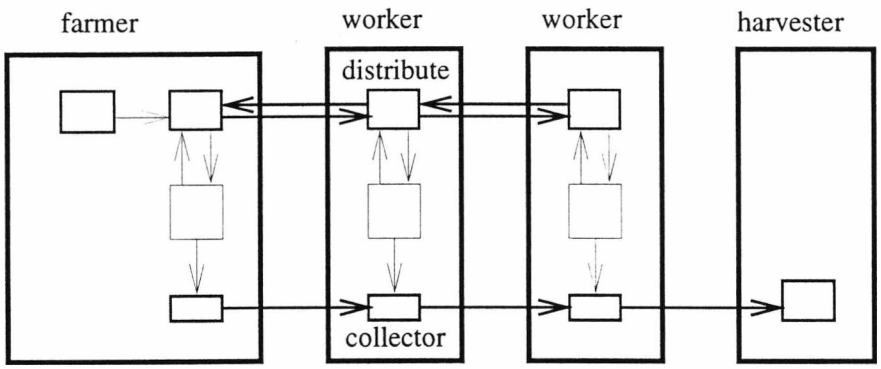


Figure 3: Structure of harness A

In terms of the higher level organisation of the farm results should come out of a different end to that of the jobs so as to keep the amount of communications performed by each processor balanced to a greater degree.

This harness had the distant workers with the highest priority in the `PRI ALT` of the job distributor process (`splitter`). It was believed this would be more efficient as the only time both workers would want work is upon initialisation. In this situation if the first job was passed to the local worker, the worker process upon receiving the job would be instantly context switched out so that the high priority job distributor could obtain the next job. Thus it was generally considered better to give out the first jobs to the most remote corners of the farm where they can be started immediately and the rest of the farm is initialised in a wavefront which propagates back towards the farmer.

```
PROC distribute (CHAN OF REQ req, CHAN OF JOB work,
                 CHAN OF REQ local.req, CHAN OF JOB local.job,
                 CHAN OF REQ distant.req, CHAN OF JOB distant.job)
  ...  variables
  WHILE TRUE
    SEQ
      req ! TRUE
      work ? job
      PRI ALT
        distant.req ? any
          distant.job ! job
        local.req ? any
          local.job ! job
:
```

This approach was considered a good idea as in the application this harness was developed for, jobs were smaller than results and the harvester was placed at the opposite end of the line of workers to the farmer. Thus it seemed sensible to make the smallest type of message travel the furthest distance, and the jobs were smaller than the results for this particular application. The order of the guarded processes above has been reversed for these experiments. This argument doesn't take into account that the overhead for communicating a job is just the set up times for the communication instructions, the length of the message, or more importantly the length of time to perform the transfer, is performed in parallel and may not be relevant. However, Roebbers's course had not yet been attended at when harness was designed. This illustrates nicely that by not knowing the rules they can be easily broken.

The code for the result collector was equally simple and minimalistic.

```
PROC collector (CHAN OF PACKET local.result, distant.result,
                                results)
  ... variables
  WHILE TRUE
    PRI ALT
      local.result ? result
        results ! result
      distant.result ? result
        results ! result
:
```

The `collector` process gives the local worker priority, thus allowing the worker to get on with the next job. As all combiners in the whole farm do this, all workers have some form of buffer to output to and don't have to spend long periods of time driving the slow links. Here it is preferred to call this operation collecting, instead of multiplexing as others do. This is because in this case we are collecting the results from a farm and giving them to a harvester. The task of multiplexing is about sending a number of separate communications through one physical channel.

These two processes are run at high priority with the worker process at low priority,

```
PROC worker (CHAN OF REQ req, CHAN OF JOB job,
             CHAN OF REQ distant.req, CHAN OF JOB distant.job,
             CHAN OF PACKET distant.result, results)
  CHAN OF REQ local.req:
  CHAN OF JOB local.job:
  CHAN OF PACKET local.result:
  PRI PAR
    PAR
      distribute (req, job, local.req, local.job, distant.req,
                  distant.job)
      collector (local.result, distant.result, results)
    application (local.req, local.job, local.result)
:
```

When a worker finishes a job it passes its result on to the on-chip collector process. The worker then gets its next job of work from the on-chip splitter process, there is no need for a special buffer process to hold this job for the worker, every process acts as a buffer for the information it holds. All that is needed is for the next job to be on-chip.

4.3.2 Harness B: Welch's harness

The second harness used here had been used very successful in an application mentioned in subsection 3.2.1 (page 30).

This harness was designed by Welch as a farming harness that exploited all of the internal parallelism of the transputer. This was designed using the process oriented model mentioned in section 2.1 (page 3) and thus consists of one process per function.

This harness had already been proved to give good performance and good linear speed up for one application. It also resulted in a quicker execution than harness A for the application that harness had been designed for.

The original design consisted of the workers arranged as a pipeline, see figure 4 (top of next page), with the results being returned back up towards the farmer. Thus, by expanding the number of adjacent workers, the farm's topology can be changed from a pipeline (an unary tree), to a binary or ternary tree, see figure 5 (also on next page for ease of comparison).

The harness was coded, tested and tuned in [SS91, Stu91]. For the protein sequencing application that this harness was use for, Sturrock found the harness performed better when the results were passed along to the last processor which is connected directly to the farmer chip. Here this is termed configuring the workers in a ring, see figure 6 (bottom of previous page), opposed to as a pipeline.

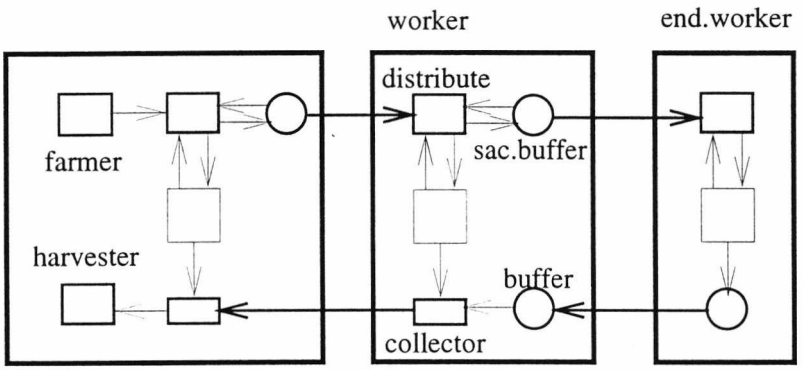


Figure 4: Structure of a pipeline

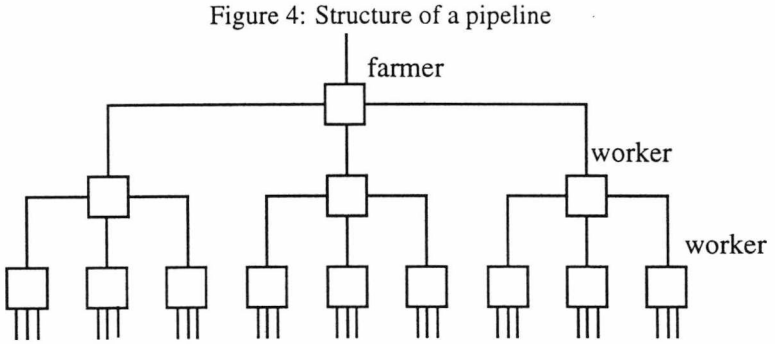


Figure 5: General structure of a ternary tree

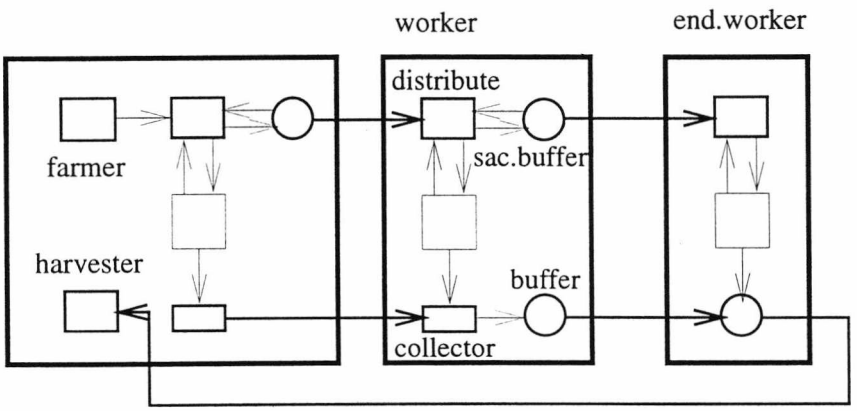


Figure 6: Structure of a ring

Transformation

The original design for this harness had the potential to perform an amount of buffering that was decided by the programmer. Sturrock found that the optimum amount of buffering to have was one item. For simplicity it has been presumed here this result carries over to all application mappings and for the rest of this work just a single buffer space is used in the other harnesses.

As only one item was needed Welch and Sturrock transformed the code of the process that performed this variable amount of buffering into a process that just buffered a single job. Unfortunately this code had an inefficiency in it.

```
SEQ
  work := FALSE
  WHILE TRUE
    PRI ALT
      (!work) & in ? job
        work := TRUE
      work & req1 ? any
        SEQ
          out1 ! job
          work := FALSE
      work & req2 ? any
        SEQ
          out2 ! job
          work := FALSE
```

By following the code through, we can see that when `work` is `FALSE`, the loop only engages the `in` channel of the `ALT`. Then after setting `work` to `TRUE` and going around the loop again, the other two channels are engaged and one of the output channels is given the job. This two stage process repeats continually.

We can express this inputting and outputting more clearly as the two separate processes they are. Using the techniques learnt from Roebbers's course the code was quickly transformed into,

```
WHILE TRUE
  SEQ
    in ? job
    PRI ALT
      req1 ? any
        out1 ! job
      req2 ? any
        out2 ! job
```

This piece of code is much more obvious, natural, readable and also more efficient as it engages only half as many `ALT`s.

This transformation can be performed explicitly in four stages: expansion, substitution and two stages of minimisation.

The first stage consists of writing out the loop with two copies of the body, one in sequence after the other,

```
SEQ
  work := FALSE
  WHILE TRUE
    SEQ
      PRI ALT
        (!work) & in ? job
          work := TRUE
        work & req1 ? any
          SEQ
            out1 ! job
            work := FALSE
        work & req2 ? any
          SEQ
            out2 ! job
            work := FALSE
      PRI ALT
        (!work) & in ? job
          work := TRUE
        work & req1 ? any
          SEQ
            out1 ! job
            work := FALSE
        work & req2 ? any
          SEQ
            out2 ! job
            work := FALSE
```

Then replacing the value of `work` directly into the code at every place where `work` is used,

```
WHILE TRUE
 SEQ
    PRI ALT
      !FALSE & in ? job
        SKIP
      FALSE & req1 ? any
        out1 ! job
      FALSE & req2 ? any
        out2 ! job
    PRI ALT
      !TRUE & in ? job
        SKIP
      TRUE & req1 ? any
        out1 ! job
      TRUE & req2 ? any
        out2 ! job
```

Here we can clearly see that the pre-conditioned guards are either constantly `TRUE` or constantly `FALSE`. For those that are `TRUE` the pre-condition can be removed. For those that are `FALSE` the entire guard and corresponding process can be omitted,

```
WHILE TRUE
  SEQ
    PRI ALT
      in ? job
        SKIP
    PRI ALT
      req1 ? any
        out1 ! job
      req2 ? any
        out2 ! job
```

This first ALT only has one channel to engage, clearly there is no alternative here and this can be written as a single communication,

```
WHILE TRUE
  SEQ
    in ? job
    PRI ALT
      req1 ? any
        out1 ! job
      req2 ? any
        out2 ! job
```

This last transformation is using the equivalence law *input* we mentioned in subsection 2.2.4 (page 10).

Code for harness B

The code used for harness B here was as follows,

```
PROC distribute (CHAN OF JOB work,
                 CHAN OF REQ local.req, CHAN OF JOB local.job,
                 CHAN OF REQ distant.req, CHAN OF JOB distant.job)
  ... variables
  WHILE TRUE
    SEQ
      work ? job
      PRI ALT
        distant.req ? any
          distant.job ! job
        local.req ? any
          local.job ! job
:

PROC job.buffer (CHAN OF JOB in,
                 CHAN OF REQ req, CHAN OF JOB out)
  ... variables
  WHILE TRUE
    SEQ
      in ? job
      out ! job
:
```

```
PROC sacrificial.buffer (CHAN OF REQ req,
                         CHAN OF JOB in, out)
  ... variables
  WHILE TRUE
    SEQ
      req ! TRUE
      in ? job
      out ! job
:

PROC collector (CHAN OF PACKET local.result, distant.result,
                               results)
  ... variables
  WHILE TRUE
    PRI ALT
      local.result ? result
        results ! result
      distant.result ? result
        results ! result
:

PROC result.buffer (CHAN OF PACKET in, out)
  ... variables
  WHILE TRUE
    SEQ
      in ? result
      out ! result
:
```

These processes were then connected together. The end worker needing two buffer processes instead of a full harness. The pipeline topology is used here.

```
PROC worker (CHAN OF JOB job.from.link, job.to.link,
             CHAN OF PACKET link.in, link.out,
             VAL INT id.number, worker)
  CHAN OF REQ local.req, distant.req:
  CHAN OF JOB local.job, distant.job:
  CHAN OF PACKET channel, local.result:
  PRI PAR
    IF
      id.number < worker
        PAR
          distribute (job.from.link, local.req, local.job,
                                     distant.req, distant.job)
          sacrificial.buffer (distant.req,
                              distant.job, job.to.link)
          result.buffer (link.in, channel)
          collector (local.result, channel, link.out)
      TRUE
        PAR
          job.buffer (job.from.link, local.req, local.job)
          result.buffer (local.result, link.out)
    application (local.req, local.job, local.result)
:
```

4.3.3 Harness C: a harness developed using Roebbers's transformations

Roebbers's course encouraged the use of single processes that, for the purpose of engaging links in parallel, repeatedly perform a sequence of PARs, each of which contain a small number of small processes. A third harness was developed from harness A that utilised these techniques. The job distribution process is as follows,

```
PROC distribute (CHAN OF REQ req, CHAN OF JOB work,
                 CHAN OF REQ local.req, CHAN OF JOB local.job,
                 CHAN OF REQ distant.req, CHAN OF JOB distant.job)
  ... variables
  SEQ
    req ! TRUE
    work ? job0
    WHILE TRUE
      SEQ
        PAR
          SEQ
            req ! TRUE
            work ? job1
          PRI ALT
            distant.req ? any
              distant.job ! job0
            local.req ? any
              local.job ! job0
        PAR
          SEQ
            req ! TRUE
            work ? job0
          PRI ALT
            distant.req ? any
              distant.job ! job1
            local.req ? any
              local.job ! job1
```

The behaviour of this job distribution mechanism is very different to that of harness B. This harness gets in work from the link and in parallel decides which worker should get the job. This is considered advantageous over the approach taken by harness B as if the on-chip worker is busy, harness B passes incoming jobs on to the sacrificial buffer automatically. If the on-chip worker then finishes its current task there is a job within the transputer's memory that the worker can not process.

The result collection mechanism is,

```
PROC merge (CHAN OF PACKET local.result, distant.result,
                           results)
  ... variables
  SEQ
    PRI ALT
      local.result ? result0
        SKIP
      distant.result ? result0
        SKIP
    WHILE TRUE
      SEQ
        PAR
          results ! result0
          PRI ALT
            local.result ? result1
              SKIP
            distant.result ? result1
              SKIP
        PAR
          results ! result1
          PRI ALT
            local.result ? result0
              SKIP
            distant.result ? result0
              SKIP
```

These two processes are tied together in the same way as harness A.

4.3.4 Harness D: pointer passing harness

This harness came out of an attempt to save passing large messages between the various processes of a harness and thus around the memory of the transputers. Welch believed it would be better to have a harness that exchanged indexes into an array of messages than to continually set up and shutdown several local PARs.

The strategy employed in this harness consists of declaring an array of message buffers and passing indexes to these between the different processes of the harness as pointers. This method still consists of having a number of separate processes, but doesn't consist of long messages being passed between them, just INTs.

Further, this method does not involve the continual setting up and shutting down of the expensive PAR construct used in harness C. The disadvantage of that strategy being each component of the parallel statement must terminate before the whole PAR can terminate and execution continue. Thus, if the message read on the output link is say much shorter than the message being read on the input link, the output link will sit idle until the longer communication has finished. This holds the harness back from progressing. Similarly, harness C can prevent work from being done by preventing results generated on-chip from being output to the harness as soon as possible. In this case, if a communication on the input link starts up just before a job finishes, the worker process is preventing from outputting the result to the harness until the incoming result is output. By having the links driven by separate processes, after a process has completed a short task, that process can continue on to its next task.

The main disadvantage of this pointer passing method is that the compiler's alias checking flag must be turned off in order for the code to compile. The checking that the different parts of the array are not accessed in parallel is removed from compilation and is given to the programmer to check or to prove.

This idea originates from a process that had two buffers, and a manager process, see figure 7 (next page). This manager is also a buffer process to decouple the other three processes from working in complete synchronisation, i.e. to introduce an extra process to create some parallel slackness.

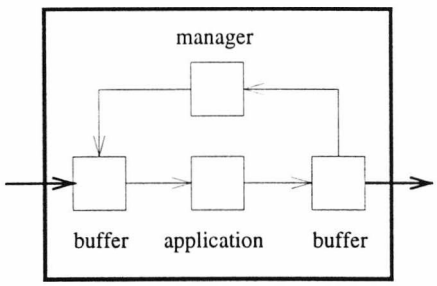


Figure 7: A pipeline with manager process

For a farming harness Welch realised a manager process was not required. The worker and the processes of the harness would always communicating in pairs. Thus, these processes just needed to exchange pointers, see figure 8 (next page).

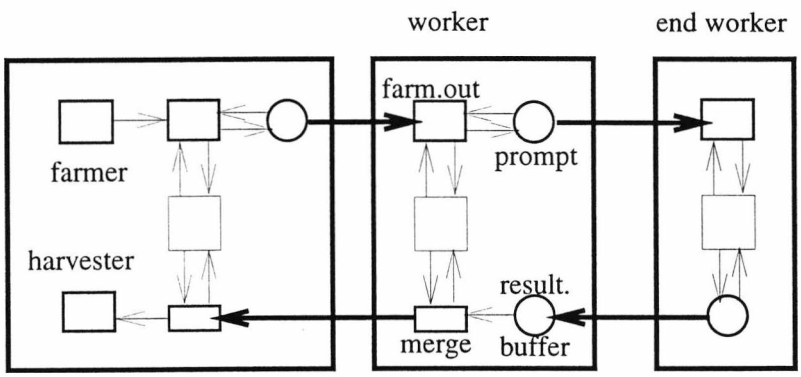


Figure 8: Structure of harness D

Naturally if the jobs and the results to be available to any process that required them, the buffer slots must be declared globally like so,

```
[3]INT len, n:
[3][max.job.msg]BYTE job.msg:
[3][max.result.msg]BYTE result.msg:
```

The harness processes Welch supplied were as follows,

```
PROC farm.out (CHAN OF JOB jobs, CHAN OF PTR local.req, local.job,
               workmate.req, workmate.job, VAL INT pointer)
  INT p, p2:
  SEQ
    p := pointer
    WHILE TRUE
      SEQ
        jobs ? len[p]::job.msg[p]
        PRI ALT
          local.req ? p2
            local.job ! p
          workmate.req ? p2
            workmate.job ! p
        p := p2
:
```

```
PROC prompt (CHAN OF PTR req, in, CHAN OF JOB out, VAL INT pointer)
  INT p:
  SEQ
    p := pointer
    WHILE TRUE
      SEQ
        req ! p
        in ? p
        out ! len[p]::job.msg[p]
:

PROC result.buffer (CHAN OF PACKET in, CHAN OF PTR out, new,
                    VAL INT pointer)
  INT p:
  SEQ
    p := pointer
    WHILE TRUE
      SEQ
        in ? n[p]::result.msg[p]
        out ! p
        new ? p
:

PROC merge (CHAN OF PTR local.result, local,
                        workmate.result, workmate,
            CHAN OF PACKET results, VAL INT pointer)
  INT p, p2:
  SEQ
    p := pointer
    WHILE TRUE
      SEQ
        PRI ALT
          local.result ? p2
            local ! p
          workmate.result ? p2
            workmate ! p
        results ! n[p2]::result.msg[p2]
        p := p2
:

PROC application (CHAN OF PTR req, in, out, new,
                  VAL INT pointer)
  INT ptr.j, ptr.r:
  SEQ
    ptr.j, ptr.r := pointer, pointer
    WHILE TRUE
      SEQ
        req ! ptr.j
        in ? ptr.j

        ... work from job.msg[ptr.j] to result.msg[ptr.r]

        out ! ptr.r
        new ? ptr.r
:
```

These four processes were tied together in the same way as harness B.

```
PROC worker (CHAN OF JOB jobs, distant.job,
             CHAN OF PACKET results, distant.result)
  [3]INT len, n:
  [3][max.job.msg]BYTE job.msg:
  [3][max.result.msg]BYTE result.msg:
  CHAN OF PTR onchip.req, onchip.job, onchip.result,
              onchip, req, job, result, new:
  PRI PAR
    PAR
      farm.out (jobs, req, job, onchip.req, onchip.job, 1)
      prompt (onchip.req, onchip.job, distant.job, 2)
      result.buffer (distant.result, onchip.result, onchip, 2)
      merge (result, new, onchip.result, onchip, results, 1)
    application (req, job, result, new, 0)
:
```

Again the worker at the end of the pipeline will need buffer processes to decouple the application process from the slow link communications. Thus, the configuration code used for harness D has the same overall structure of the worker process at the bottom of page 42.

Since this harness was designed and tested, it has been realised the message buffers are accessed here as global variables. This being due to the processes of the harness using the names of the buffers directly, without these variables being defined within the scope of the individual processes. By passing the buffers into the processes as parameters they would be accessed as local variables. This approach to writing code is not only more sound engineering practice, it is also faster to execute. The variables being accessed more directly by the transputer, there being no need to use an indirect addressing mode. Here we show how the `prompt` process would be written using this technique,

```
PROC prompt (CHAN OF PTR req, in, CHAN OF JOB out,
             []INT n, [][]BYTE job,
             VAL INT pointer)
  INT p:
  SEQ
    p := pointer
    WHILE TRUE
      SEQ
        req ! p
        in ? p
        out ! n[p]::job[p]
:
```

This network of processes would be set up with the following,

```
PROC worker (CHAN OF JOB jobs, distant.job,
             CHAN OF PACKET results, distant.result)
  [3]INT len, n:
  [3][max.job.msg]BYTE job.msg:
  [3][max.result.msg]BYTE result.msg:
  CHAN OF PTR onchip.req, onchip.job, onchip.result,
              onchip, req, job, result, new:
  PRI PAR
    PAR
      farm.out (jobs, req, job, onchip.req, onchip.job,
                len, job.msg, 1)
      prompt (onchip.req, onchip.job, distant.job,
              len, job.msg, 2)
      result.buffer (distant.result, onchip.result, onchip,
                     n, result.msg, 2)
      merge (result, new, onchip.result, onchip, results,
             n, result.msg, 1)
    application (req, job, result, new, 0)
:
```

Changing to this approach results in accesses to the messages being a cycle quicker to execute. However, this improvement is not likely to be large compared with the differences in performance obtained by the other harnesses in this study.

4.3.5 Harnesses E and F: bidirectional harnesses

Another idea Welch had was that harnesses like his, that only passed messages in one direction over the links, could be used to run messages over the links in both directions and thus could supply jobs to another worker process running in parallel with the first. The transputer has a very low context switch time so switching between the two workers evenly would not be a large overhead. This would increase the amount of bandwidth to the workers and the extra workers should also result in greater parallel slackness. Thus, should one of the workers on a processor be waiting for a job, the processor has another worker process to execute, instead of lying idle.

Using the links in both directions simultaneously doesn't give double the bandwidth, due to the transputers sending acknowledgement packets, but, Welch estimated, should give another fifty per cent. As the amount of computation performable hasn't increased it is possible to deliver more work in the same time to the same amount of computing power. This should prevent some implementations being communication bound.

The other harness that uses the links in one direction like this is harness B. For the sake of completeness it was decided to also double up harness B so there was a doubled up harness design that could be compiled with usage checking switched on. This might prove useful in performing comparisions between the different aspects of harness design, and not just individual implementations.

4.3.6 General note

As a general note on harnesses. The way chosen to implement the harnesses here is with a fixed amount of fan out and fan in. The reason for this is although the code for harnesses can be generalised to farm out work for an arbitrary number of channels and links, this generally requires replicators that take longer to set up than a normal ALT. As finding the best harness possible is what is of interest here, there was no interest in then making such a harness flexible and potentially slower, in order to allow the harness to be be configured for a poorer performance. Thus, when a harness for a ternary tree instead of a line of workers is needed, an appropriate harness will be developed for that situation.

4.4 Planning the study

We have established this study is interested in finding the most efficient harness. Also, we have looked at the harnesses around at the start of the experiments. Here we look at how the experimentation was planned.

4.4.1 The approach to testing

Most importantly, what was of interest was testing the design strategies that gave rise to these farming harnesses, not just the code itself. This approach was acquired from UNITY, where one is interested in the design of programs as opposed to the program itself.

The reason for this interest is the way in which a good program is written is much more important than the program itself, since a technique can be used in other situations. Thus, once this experimentation has arrived at a strategy that results in the most efficient harness known, we may be able to use that design philosophy to construct a strategy for dealing with other situations with similarly optimal efficiency.

4.4.2 This study's limited parameter space

In addition to the realisation that an application and an execution model are independent of one another, it was also realised there are three main aspects of an application that a farming harness is aware of. These are,

1. the average length of the job messages farmed out,
2. the average length of time to compute these jobs, and,
3. the average length of the result messages.

In UNITY terms, the actual values of these parameters would come from the way the application was mapped onto the hardware. For any application, many different mappings are possible. What was of interest here was finding which range of parameters are compute bound and thus efficient.

The natural scientific approach is to test a number of harnesses for a range of applications and then display these results on a graph. These graphs would not only need to display these three parameters on an axis each, but also another axis would be required for the purposes of comparision. This results in graphs containing four dimensions. Unfortunately, only three dimensional graphs can be display easily. This applying to the printed page especially. This led to a need for just having two parameters to vary.

Thus, the approach adopted here is tie together the two message related variables into one and evaluate the harnesses performance with both job and result message being the same size. Doing this gives the following variables,

1. the average length of job compute time, j, and,
2. the average length of message communicated, m.

This leaves us with a third axis free for comparisions as required. The method of comparision here being efficiency.

This reduces the scope of the experimentation by a small degree, but without invalidating the results obtained, as the two variables tied together are similar in nature. So the range of experimentation performed here has been restricted to a certain extent by our limitation to display results.

4.4.3 Varying the parameters

So this study was performed with two application parameters, j and m, to vary. There is also another parameter, that of the number of workers in the farm, w.

So, in order to find the breakdown of the harnesses, all that was needed was to wind up or down these three parameters. This generates a succession of mappings that are gradually more or less demanding depending on the particular influence of the parameter. Each parameter just needs to be wound up or

down far enough until all of the harnesses have broken down completely. The harness that can cope with the most demanding parameters will be the most efficient. Now a position had been reached where what should be tested was known.

It was also easy to see and predict what affect increasing each parameter would have on performance.

Increasing w, the number of workers, would result in more work being performed and also more communications in any time period due to more workers claiming work. This would increase the load on the farmer and would eventually result in the application becoming communication bound.

Increasing j, the average time it takes to process one job, would result in less jobs being processed in any particular time frame. This would result in the demand for jobs being less frequent. In turn this would decrease the load on the farmer and the application would become more compute bound, the same number of jobs taking more time to be processed.

Increasing m, the average length of the messages, would increase the communication load on the implementation and the farm will become communication bound.

An increase in B, the bandwidth of traffic coming out of the farmer, would results in more jobs going into the farm at any one time. This would make the farm become more compute bound.

4.5 Mathematical modelling

It was suggested there were two things one might wish to predict in advance about an implementation. The first is what a farm's efficiency would be,

$$\text{efficiency} = \text{eff}(j, m, w)$$

The second is, for a given application mapping j and m, what is the maximum number of workers a farm could have and still be efficient,

$$w = w_{max}(j, m)$$

The possibility of these were looked at.

4.5.1 Compute bound or communication bound

From thinking about what a programmer would want to use, it was realised one would be interested in knowing in rough terms whether an implementation is likely to be compute bound or communication bound. The actual efficiency may also be of use, though this is not as important.

It was realised that if a network of transputers was being utilised fully, both the links and the C.P.U.s would be in use all the time. In terms of farming this would mean all the links's bandwidth was required to keep all of the workers in work all of the time. This could be expressed as "the throughput of the links" would be the same as "the amount of work performed by the workers", i.e.,

$$\text{number of jobs through a link per second} = \text{number of jobs performed per second}$$

The number of jobs one can get through a link, in a second say, is simply the bandwidth of the link, B, divided by the sum of the number of bytes in the message, m, and the overhead of setting up the communication. Here this is expressed as the number of bytes that could be transferred in the time it would take to set up the communication, s,

$$\frac{B}{m+s}$$

The number of jobs performed in a second can be calculated similarly. This is the number of workers in the farm, w, divided by the number of seconds it takes to perform each job, j,

$$\frac{w}{j}$$

Putting these two together gives us a model in the form of a simple equation. This equation gives us an expression for a farm in which both the computational and communication parts of the implementation

are working to the full.

$$\frac{B}{m+s} = \frac{w}{j}$$

This equation indicates that, in a farm, the maximum number of jobs a farmer can output in any time period is inversely proportional to the length of those jobs, as well as being directly proportional to the amount of bandwidth there is available both out of the farmer and around the harness. Similarly, the maximum number of jobs that can be processed per second is proportional to the number of workers, and is inversely proportional to the time it takes for one worker to process a job.

In terms of farm mechanics, an implementation is compute bound and runs efficiently when the workers demand jobs at a rate lower than the supply. Or more simply, when the number of jobs that can be processed per second is less than the number of jobs that can be communicated per second. We can express this property of compute bound farms in terms of a model based on the equation developed above. In this model the relation between supply and demand for a compute bound farm is simply,

$$\frac{B}{m+s} \geq \frac{w}{j}$$

Similarly, an implementation is communication bound and not very efficient when the throughput is not enough to deliver the amount of work needed to keep all of the workers busy all of the time. Here, the above relation would contain a $\leq$.

The bandwidth, B, is likely to be constant for any particular piece of hardware, with there being slightly different values for each harness design, depending on its efficiency. On architectures like the transputer system used here, measures can be taken to increase B by reducing the number of switch chips used.

Communication setup time

In the above discussion, communication setup was measured in bytes, not time, as this hardware characteristic is usually measured. Here the communication's start up cost was expressed in how many bytes could the channel communicate in the time it takes to setup a communication. As we invariably know the number of bytes to be transferred in a communication, but not how long it will take, measuring the setup cost in this way is useful as it displays the start up overhead in the same metric that the length of the communication is measured in. This helps to provide some insight into the minimum size of a communication that will be effective and how efficiently the communication channel is being utilised.

The value of s is easily calculated with the equation,

$$s = B \times \text{set up time}$$

Thus, as the setup time for the T800 transputer is 3 microseconds, and 1.51 Megabytes per second has been obtained through a link here, s has the value of 4.75 bytes, for a standard communication and 13.5 bytes for a counted array that uses an `INT` for the count.

Presented here, for the first generation transputer is the raw bandwidth, B; the communication setup time expressed in microseconds; the communication setup time expressed as the number of bytes that can be transferred in that time, s; and the size of the communications needed to obtain 60%, 70%, 90% and 95% efficiency through the communication channel.

	B Mb/s	set up time microseconds	s bytes	m to get 60% bytes	to get 70% bytes	90% bytes	95% bytes
T800	1.5	3	4.75	7	11	42	90
T800 counted array	1.5	8.6	13.5	20	32	116	232

Table 1: Communication set up time's influence on performance of transputer

For large s, very large messages will need to be communicated if the channel is to be used reasonably effectively. As implementations should be driven by what will make an application efficient, a large s will

be detrimental to the performance obtainable for a fine grain mapping. Thus preventing such mappings from being an option. If s is very large, communication can ceases to be useful if parallel implementations are to be efficient. An example of this is in [JR92]. Here Jenson and Reed noticed that the communication set up time from their host to their Intel iPSC/2 hypercube was 720 microseconds (this included the operating system calls). As the bandwidth was 2.8 Megabytes per second, s is 2114 bytes here and messages need to be of 3 to 5 kilobytes before 60–70% of the bandwidth is used and 19 or 40 kilobytes if 90–95% of the bandwidth is to be used effectively. Thus, having such a large value of s for an architecture does not encourage the communicating of small messages, such as just a single integer.

As the value of s is small for the first generation of transputer links, it is ideal for performing fine grain communication and thus fine grain work.

When m is significantly greater than s, a situation we prefer to be in when implementing a farm, it should be possible to omit s in calculations for simplicity and the results still be highly accurate.

4.5.2 Estimating the maximum number of workers

As well as having a general model of efficiency, a method to work out the maximum number of workers a farm could cope with was also worked out.

Here we would like something that is calculated from the average time it would take to perform a job and the average length of a work packet (or the average length of time it would take to communication one). Thus we would like a function of the form,

$$w = w_{max}(j, m)$$

From the model of efficiency we already have,

$$\frac{B}{m+s} \geq \frac{w}{j}$$

obviously the maximum number of workers possible in any situation is going to be related to the values of j and m used,

$$w \approx \frac{j}{m}$$

This would seem to imply the way to proceed with an implementation, is to first measure the average job processing time and the average message length and the bandwidth out of the farmer. Then an estimate of the maximum number of workers such an implementation can have and it still be compute bound is,

$$w = \frac{jB}{m+s}$$

More generally, any farm with this or a smaller number of workers,

$$w \leq \frac{jB}{m+s}$$

will be compute bound. Any farm with a larger number of workers will be communication bound.

4.5.3 Rationale for models's simplicity

The models here are deliberately small and very simple. Initially, when considering potential mappings, all we would like to know is either if a mapping will be compute bound or not, or how many workers a mapping will allow. This second figure we only need to know to the nearest integer. It would be desirable to know either of these as quickly as possible. A small model will be quick to work with. A good small model will capture the essence of what is relevant, by giving all that is required, an estimate. A more precise and thus detailed model would give more precise values. There are two problems with this.

Firstly, performing the calculation for a more precise model will take longer due to the larger number of parameters involved in more complex models. All of these extra parameters have values that would

need to be found out. Some of these could obtained from data sheets. Others will have to be found by measuring an implementation which is very time consuming. If finding out the values to many parameters is time consuming, programmers will not bother to use the model.

Secondly, in order to be more accurate, a more detailed model would require much more complexity and would need to take into account the performance characteristics of the topology and harness being used. Modelling the harness would include issues such as the code's latency. Modelling the topology would include looking at the decrease in throughput from worker to worker. For example, if the throughput out of a farmer is,

$$\frac{B}{m+s}$$

Then if k is the amount of throughput lost through the execution of the harness on one transputer, the throughput out of the first worker is,

$$\frac{B}{m+s} - k$$

the throughput out the second worker is,

$$\frac{B}{m+s} - 2k$$

and so on.

One of the major problem with constructing modelling to this degree of accuracy is that if the topology or the harness is changed, the model needs to be changed appropriately too. Detailed models do have their use, but here they are not considered appropriate for use in the initial stages of implementation.

There may be the requirement for models that possess greater accuracy and thus are slightly more detailed in the appropriate ways. More detailed models exist in the literature, such as [TD90] by Tregidgo and Downton. However, the emphasis here is on smaller models that are both needed and useful.

The largest model here still has five parameters. Two of these, B and s can be looked up here or found by experimentation. One, w is selected by the programmer. The remaining two, m and j are a result of the mapping strategy selected by the programmer. The first of these can be counted in bytes directly from the code, the second is the only one that needs to be found by testing.

One problem with constructing models that attempt to obtain a certain degree of accuracy is that of knowing what issues make appropriately large differences and are thus worth including in the model.

4.5.4 Summary

A model has been developed here that can be used to study a farm. It can find out whether an implementation is compute bound or communication bound. By rearranging the equation it can also be used to estimate the maximum number of workers an implementation may have and still be compute bound.

The first is more useful in studying an existing implementation. The second is more useful in developing an implementation.

4.6 Raw link bandwidth

It was suggested that in the main experiments, the lengths of messages should be measured by the length of time it takes to communicate them, instead of the bytes as is usual. The advantage of this being that communication time and computation time can then be compared directly. This would provide insight into how the two variables affect one another more directly. For example, it would not be obvious that say, 1049 bytes of data would take 15.6 low resolution clock ticks to communicate. More importantly an application is likely to become communication bound, not when the job communication time is equal to the job process time, but when the job communication time is proportional to the product of the job process time and the number of workers. Such a relationship would be much easier to spot if all of the measurements being measured using the same metric.

The quickest way to turn a message length into a time is to measure how long it takes to be communicated. This conversion needs to be something that will be constant. It was decided to look at the bandwidth of both the links and the harnesses. Link bandwidth will be constant and thus it is looked at

in this section. The throughput of harnesses is going to be different from harness to harness, this is also of interest, and will be looked at in the next section.

Thus, this experiment was performed in order to discover the speed of communication through a transputer link.

This experiment is naturally all about a communication. The basis of the whole experiment is to discover how long it takes to perform the communication,

```
PLACED PAR
  link ! message
  link ? message
```

between two transputers.

Normally we are interested in communicating a counted array,

```
PLACED PAR
  link ! n::packet
  link ? n::packet
```

Thus, this experiment is very much about the amount of throughput obtainable for different lengths of message.

There is also the issue of how far apart these two transputers are. So in each test we are also interested in the number of switch-chips through which the communication takes place.

4.6.1 Developing the test program

In order to perform the timing of this communication it was decided to use the transputer's in-built high priority timer, as this gives a higher resolution of measurement.

That said, the length of time to communicate an individual byte or message is still very small so each message was sent a fixed number of times, 128. An average was then taken.

For the experiment to be informative, communications from one byte up to several kilobyte messages needed to be timed. Both extremes of this range being common in transputer implementations. It was decided to test packet lengths that are powers of two, starting at one byte and continuing up to 16 kilobytes. A message of this size should be more than enough to overshadow the communication's setup time. Sixteen kilobytes is also some way above the largest message size seen by this author in a genuine application (4 kilobytes in Sturrock's protein sequencer).

The timings were performed in the following way: an empty packet was sent across the link to act as a synchronisation, the receiving end then noted the start time, the packets would be sent through the link and the time difference was noted. The communication would actually get under way at the sending end while the start time is being read. However, the first byte of the message will not be acknowledged and thus the rest of the message not sent, until the receiving transputer executes the input statement.

The code used was,

```
[max.packet.len]BYTE packet:
SEQ size = 0 FOR 15
  SEQ
    n := 1 << size
    link ! 0::packet
    SEQ i = 0 FOR 128
      link ! 1<<n :: packet
```

on the sending transputer and,

```
INT len:
[max.packet.len]BYTE packet:
TIMER clock:
INT start, stop:
SEQ size = 0 FOR 15
  SEQ
    link ? len::packet
    clock ? start
    SEQ i = 0 FOR 128
      link ? len::packet
    clock ? stop
    performance ! stop - start
```

on the receiving end.

A bidirectional version of the code was also developed. Here the unidirectional communication, of the following basic form,

```
PLACED PAR
  SEQ i = 0 FOR 128
    link ! len::packet

  SEQ i = 0 FOR 128
    link ? len::packet
```

was replaced by a pair of parallel communications sent across the one link bidirectionally,

```
PLACED PAR
  PAR
    SEQ i = 0 FOR 64
      link1 ! size::packet
    SEQ i = 0 FOR 64
      link2 ? len::packet2

  PAR
    SEQ i = 0 FOR 64
      link1 ? len::packet
    SEQ i = 0 FOR 64
      link2 ! size::packet2
```

The both of the variables named `size` were initialised to the same value.

The two tests were performed by commenting folds in and out and toggling between the two tests.

The output from the `performance` channel was sent to a third transputer. Thus giving the final program, as shown in figure 9.

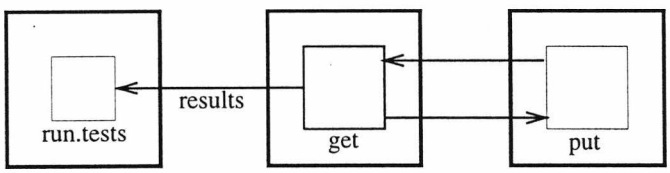


Figure 9: Test rig for link bandwidth experiment

The third processor handled the acquisition of the filenames, the conversion of the results into various units (bytes per second, seconds per byte etc.) and the filing of these values. This was done in order to keep the test code away from any external interferences. By having a separate processor perform these

tasks it prevents the setting up of the tests and storing of the results interfering with the running of the test.

This also meant more of the code was held in on-chip memory, not on the slower off-chip external memory. This would definitely not be the case if the outputting of the results to a file was performed by a processor involved with the testing. This is as the system's libraries involved for this system are large.

Having the test code in on-chip memory is ideal for this test. The bandwidth values arrived at is the maximum performance of the transputer and is also applicable to all transputers, regardless of the make of board being used. Thus these results are generic to all transputers.

The test program was compiled with all the usual occam flags switched on.

4.6.2 The testing performed

With the test program developed, there was then the issue of in which environment to perform the experiment. It seemed best to try and explore the extremes and the structure of the machine in use. Thus, the test program was run four times with the communication under test being between transputers of varying distances for each run of the program. The four positions used were between transputers on,

1. the same board (the smallest distance involving just one switch-chip),
2. adjacent boards within the machine (not directly connected on the machine's backplane),
3. alternate boards, namely board 4 and board 2 (alternate boards are directly connected on the machine's backplane via two switch-chips), and,
4. two boards that were as near to opposite ends of the machine as possible, namely boards 3 and 44, (the worst case, involving approximately 15 switch-chips).

There was one very minor restriction with allocating boards for this test. This arose when the two transputers performing the timed communication were on different boards. In this situation the two transputers had to be assigned to different board types (for example boards with different amounts of memory). This restriction was imposed by the `genrout` program through which the physical placement was performed. This limited the locations with which some tests could be performed within the machine.

The results were collected and analysed. There are three results: how long it takes to communicate a byte, how long it takes to communicate the packets of the various sizes and the bandwidth that we can obtain across a link.

4.6.3 Time to send a byte

The first thing discovered was how long it took to communicate a single byte down a link for each message size. This is shown in figure 10 (top of next page).

As expected for counted arrays, when the size of the message goes up, the higher is the throughput obtained. The overheads shrink and the graph's curve asymptotes down to an optimal value.

Reading off the most important figure from the graph, the time taken to transfer one byte from one transputer to another on the same board is 0.663 microseconds. Giving a bandwidth of 1.51 Megabytes per second. This level of performance is sustained for all packet sizes down to 512 bytes. Also a message travelling from one end of the machine to the other only travels at a speed 2.1 slower.

For bidirectional communication it appears that each byte is taking 0.456 microseconds to be communicated (2.19 Megabytes per second). In practice this consists of two bytes being communicated in opposite directions in 0.912 microseconds. A bandwidth of 1.1 Megabytes per second. This being due to the overheads of the acknowledgement packets used by the links, each byte actually takes longer to communicate when bidirectional communication is used, than with unidirectional communication.

4.6.4 Time to send a packet

It was also interesting to compare the time it took to communicate one packet against its size, see figure 11 (bottom of next page). The values on the y-axis of the graph are the actual timings for messages between two transputers on a single board in our machine. One thing to notice is that all of the figures are

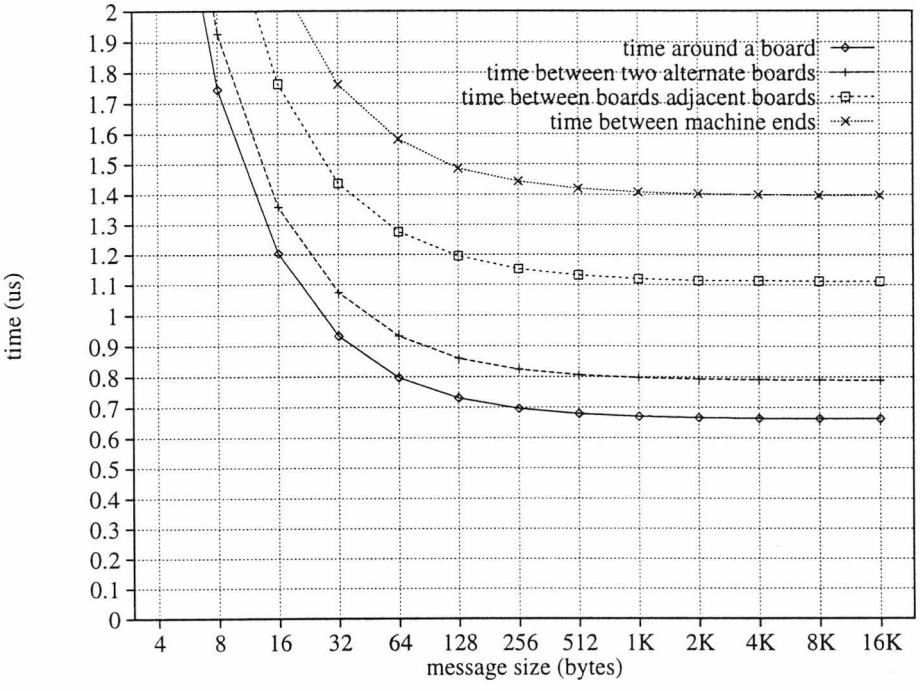


Figure 10: Time to communicate a byte (of a counted array) through a link

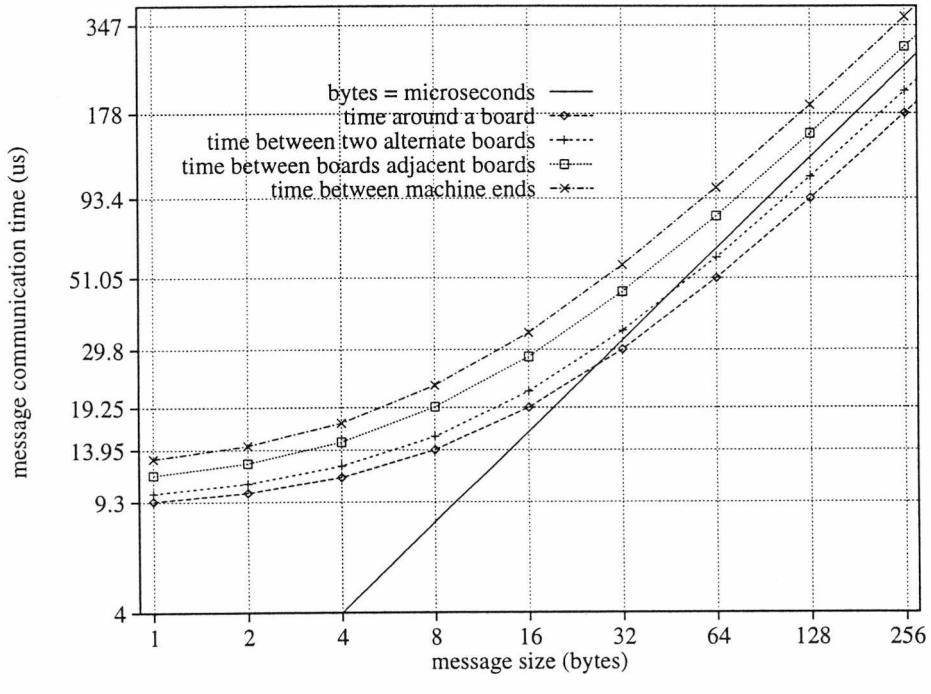


Figure 11: Average time to send a counted array through a link

reasonably close to the one byte per microsecond line. This figure is a nice average that is convenient to work with.

As can be seen, the graph consists of a near straight line that curves off towards the horizontal at the lower end. This curve is due to the set up time of the communication being of a large proportion for small messages. With large packet sizes the graph appears to be a straight line, the line only starting to curve at around 64 byte packets. At 32 bytes per packet the graph is clearly curved. This figure would appear to be a good minimum packet size whenever we have enough control over job and result packet sizes to be able to choose.

The shape of curve is due to the constant overhead of setting up a counted array communication (including communicating the count). Whenever possible we would like to avoid being at the top end of the curve. It is possible for an application to be just slightly communication bound and consisting of jobs that are smaller than 64 bytes each. In this case it would be worth ensuring that message sizes are larger than this. Larger messages would take less time on average to communicate per byte, due to the reduced set up time and could result in an implementation that is just compute bound.

4.6.5 Single component messages

These results show that an application is not inherently compute bound or communication bound necessarily. This matter also depends on both the way an application is decomposed into jobs and the size and the structure of the messages that are communicated around the farm. In short, the way an application is implemented affects the performance as well as the basic structure of the application itself.

The fact the curves are asymptotic in nature suggests a message will be communicated much more efficiently if communicated as a single contiguous sequence. On the transputer a sequential protocol of values is communicated as a sequence of separate messages and will take more time than a single large message. For example, the sequence, `8::[]BYTE; 8::[]BYTE; 8::[]BYTE; 8::[]BYTE` would take longer to communicate than just the single communication `32::[]BYTE`. This difference in performance could be enough to make an implementation compute bound instead of communication bound. In practice this can be achieved by packing all of the values for a job or a result into one array. The best way to achieve this in occam is to declare an array and then `RETYPE` segments of the array into the variables needed.

```
PROTOCOL Work IS INT::[]BYTE:
CHAN OF Work to.farm:
INT n:
[2*(SIZE INT)]BYTE packet:
VAL x.pos IS 0:
VAL y.pos IS SIZE INT:
SEQ
  INT x RETYPES [packet FROM x.pos FOR SIZE INT]:
  INT y RETYPES [packet FROM y.pos FOR SIZE INT]:
  SEQ
    x, y := generate ()
  to.farm ! n::packet
```

`RETYPE`ing parts of an array is still checkable by the compiler as before.

Packing a variant protocol into an array can be performed by declaring an array long enough for the longest message needed and then implementing the case tagging by hand.

Implementing variant protocol tags by hand also allows for as many tags as one desires. The version of occam used here currently only allows for a total 256 tags in a program [Wil91].

It is true that writing such code can be more prone to error. More code is being written so naturally there is more potential for error. In occam however, if one writes what one wants in an obvious way, then the original can be transformed into a more efficient approach, alleviating such error.

As well as communications being faster, another advantage that arises from only communicating a counted array is that all processes that use this protocol only need to know how to pass on counted arrays.

Thus, they are trivially easy to write compared with the amount of work needed to pass on a variant protocol. The body of the harness's code doesn't grow and become cluttered with all of the details of variant protocols. This saves memory and more importantly programmer time. Thus harnesses not only remain efficient and small, but can also be used for many more applications.

One disadvantage is that the processes at the other ends (the farmer and the worker) are more complicated to write. There is the issue of whether the extra code in the farmer (a bottleneck) results in a slower program. This would very much depend on how much preparation each job needed.

4.6.6 Bytes per second bandwidth

Finally here we look at the bandwidth obtained through a link.

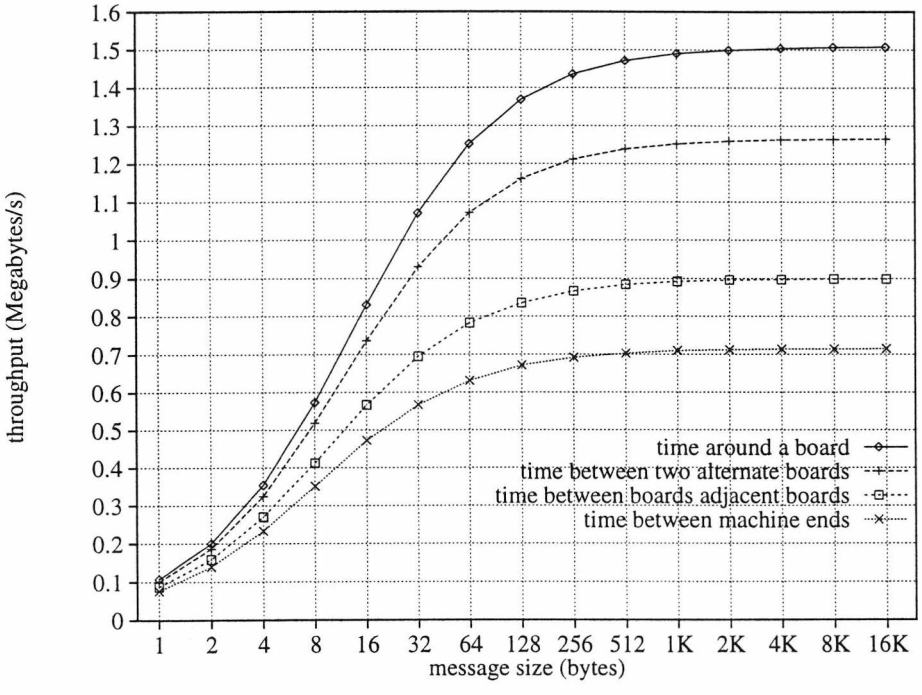


Figure 12: Throughput for a counted array through a link

If we look at these same figures in terms of the amount of data transferred per second, see figure 12, we can see that the time to communicate small messages is much longer than it should be due to the overhead of an `INT` length count.

One thing we would like to know is what is a sensible value for a smallest size of packet that is worth using. As communicating through alternate boards gives a maximum throughput of about one megabyte per second, we can use this graph as a "percentage of maximum throughput against message size" graph. So for instance, between 16 and 32 bytes we obtain between 60–70% of the maximum obtainable link bandwidth, very close to the maximum bandwidth obtainable, as predicted in table 1 (page 51).

4.6.7 Automatic processor allocation

Just to see how well the automatic domain allocation program allocated processors the test program was rerun on four domains. This was arranged by allocating a first domain. While this domain was still allocated a second domain was set up, and so on with a third and a fourth. Each domain was allocated by the automatic transputer allocation program. The program was run on each of these, one at a time. The throughput of all four domains is shown in figure 13 (top of next page).

It appears the automatic domain allocation program allocates transputers that will obtain a reasonable performance when compared with the optimum performance of the machine.

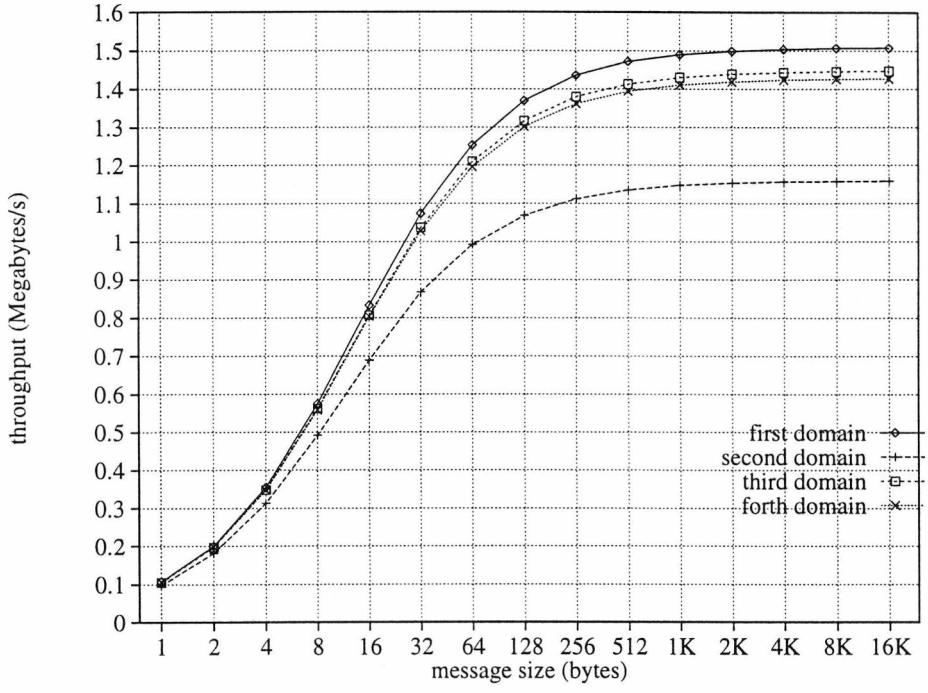


Figure 13: Throughput of counted arrays around automatically allocated domains of processors

Thus the systems software does quite a good job of allocating domains with the highest bandwidth possible to user programs.

4.6.8 Summary

The time to communicate each byte of a packet between two transputers on the same board is 0.665 microseconds. This is a bandwidth of 1.51 Megabytes per second. Communicating from one end of the Kent machine to the other (a route involving approximately 15 MEiKO switch-chips) a byte can take about twice as long to be communicated, here 1.2 microseconds. This is a factor of two difference in performance. When links are used bidirectionally, it takes 0.456 microseconds to communicate a byte. This is 2.19 Megabytes per second. Here both links are operating at 1.1 Megabytes per second.

From these graphs it can be concluded that when one has the choice of how large a counted array one can communicate, a sensible minimum message size is around 16 or 32 bytes. This will give between about 60% and 70% of the maximum bandwidth available. By the time we get to 512 byte packets there is not much more performance to obtain and much larger messages will start to eat up memory.

4.7 Results throughput obtained by harnesses

Having looked at the raw bandwidth of the transputer's links, the next thing to study was the throughput of the harnesses.

For this it was decided to use the same strategy as before, surrounding the object under test with a test rig that drives the object with a variety of messages. This experiment was then run in isolation, with again the addition of another separate processor to collect and store the results.

The collection of results involved obtaining the packets from the workers and then communicating them to the harvester. We are interested in finding out how quickly the six harnesses perform this. Thus the testing consisted of finding out how much data the merger processes could pass from the upstream link to the downstream link. This experiment set out to look at the "as much data as possible" or throughput aspect of this.

4.7.1 Test Rig Design and Implementation

The structure of this program is shown in figure 14.

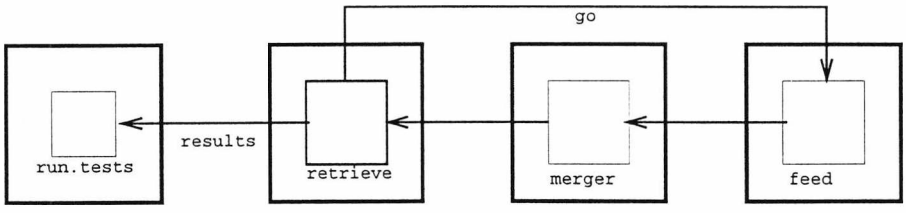


Figure 14: Test rig for harness throughput experiment

Here the third transputer holds the merge process under test. This process passed on to *retrieve* the result packets it obtained from *feed*.

The way this test program works is similar to the program used in the previous test. The *retrieve* process sent a start message along the *go* channel and would then start a timer. In this test rig this communication was sent through a separate channel. By directly connecting these two processes we can guarantee this communication is synchronous. On receiving the start message *feed* would send five groups of one hundred packets of the current size to the merger and these would eventually be received by *retrieve*. Once all the packets have been received the time was noted and was passed on to the fourth processor for computing an average and storing. As before the sizes tested were packet lengths that are the powers of two.

In these experiments a hundred packets were used, which again is adequate for obtaining timings of a detailed enough resolution with the high resolution clock.

The actual code used to implement the testing algorithm as discussed above was as follows,

```
SEQ i = 0 FOR sizes
  SEQ
    go ! TRUE
    clock ? start
    SEQ j = 0 FOR 5
      SEQ any = 0 FOR 100
        from.merger ? len::packet2
    clock ? stop
    figures ! stop - start
```

In the bidirectional version for harnesses E and F the PAR was set up after the timer was read.

```
SEQ i = 0 FOR sizes
  SEQ
    size := 1 << i
    go ! TRUE
    clock ? start
    PAR
      SEQ j = 0 FOR 5
        SEQ any = 0 FOR 100
          from.merger ? len::packet2
      SEQ j = 0 FOR 5
        SEQ any = 0 FOR 100
          to.merger ! size::packet
    clock ? stop
    figures ! (stop - start) / 2
```

This sends 1000 packets, 500 in each direction.

4.7.2 Settling

The test rig in this program is distributed across three processors, more than just the pair of transputers used in the first experiment. Because of this, there was a concern some parts of the program could be loaded and running onto a transputer before others parts that they communicated with had had a chance to initialise. This could be especially true for the code under test. The `feed` processor would have to be initialised as the timer on the `receive` processor would not be started until the `go` message had been acknowledged. If this was possible the first timing value would be incorrect as it would include the time it would take for one part of the program to initialise.

Thus, there was an obvious desire to make sure these timings would be correct and the whole domain was working. For this to happen would mean all three processors were loaded, initialised and running the test code at the same time. To achieve this it was decided to make sure all three important pieces of code should be allowed to settle. Thus, at the start of testing it would be known that all parts of the test rig would be in the correct state. To guarantee this the program performed some communication through the route under test just before the experiment got under way. For this communication to happen all three of the test rig processors must be loaded and initialised.

The whole application booted through the processor that dealt with the screen and filing, as everything was going through this, it was known this processor would be loaded correctly without any problem.

4.7.3 Compilation Flags

The test program was compiled with all the usual occam flags switched on except usage checking which was switched off. This was done as harnesses D and E, due to their design, only compile with usage checking turned off. For both convenience and fairness all the harnesses were compiled using the same set of compilation flags. Obviously if one is interested in performance all flags can be switched off for all the harnesses. What was of interest here was seeing how much of an improvement was made by moving between the different designs of the different harnesses alone, and not by giving some harnesses a performance advantage due to different compilation options.

4.7.4 Testing

The program was run with the three important transputers all on one board to obtain the near optimal performance. The program was run with all six harnesses.

4.7.5 Harness throughput

In figure 15 (top of next page) is the throughput of the harnesses. For reference the basic link bandwidths obtainable for both types of communication are also shown. Here we discuss the relative performance of the harnesses as shown.

The top line of the graph is the raw link bandwidth for bidirectional communication. Obtaining very nearly this raw performance is harness E, which clearly is the best harness here for any message of a reasonable size. The other bidirectional harness, harness F, is not as efficient, it's performance exhibiting some form of slightly erratic behaviour, even though the figures here are an average of five runs. The performance of harness F also rolls off for messages larger than 4 kilobytes. One possible cause of both of this is that block-copy instructions are not interruptable. This would interfere with other parts of the harness by delaying them from engaging in other communications. This situation is likely to get worse in a real farm as job distribution processes would be running as well as the result collecting processes, as these are likely to be using the same style of on-chip communication, they are also likely to suffer from the same problems, thus further adding to the poor performance.

Harness C's curve is very smooth and the best out of the unidirectional harnesses. Harness C also follows very closely the maximum possible unidirectional bandwidth performance across the whole range of packet sizes. This suggests that there is not much link bandwidth left to obtain and that there is always going to be a small performance cost for using a harness, which ever method of engaging both link engines in parallel is thought of.

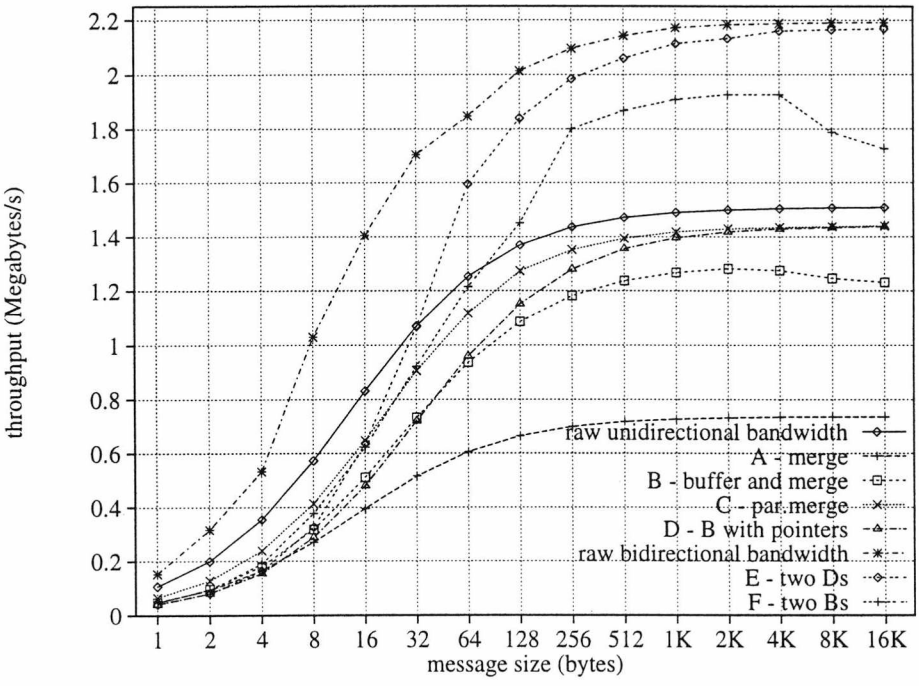


Figure 15: Throughput of result collecting processes

For the smaller message lengths harness C is the most efficient for all lengths up to about 17 bytes, followed by harness F, at about which point harness E becomes the most efficient. With the largest message lengths studied here, harness D also obtains the same level of performance as harness C, most notably at about 1 kilobytes and above.

Harness B does not perform as well as these last two unidirectional harness designs. Further, the performance of harness B, like its bidirectional equivalent harness F, tails off at just over 2 kilobytes. The most likely reason for this is the length of time it takes to perform the on-chip communication between the two processes of harness B for arrays of this size.

Harness A clearly produces the poorest performance here, giving just over half the performance of harness C. This is due to harness A only engaging one link at a time. The fact the performance is *just over* the half way mark is likely to be due to harness C needing to set up and close down PARs.

Harness E, which consists of two copies of harness D running in opposing directions through the links, possesses a much higher rate of throughput for all convenient sizes of packet than all of the other harnesses. It was thought by Welch, who designed harness E that it would probably only obtain about another 33% more throughput than harness D. In fact harness E obtains another 50% more throughput. Harness E also only provides less throughput than any other harness for packet sizes of just over 16 bytes. Below that harness E's performance deteriorates down to that of all of the other harnesses except for harness C.

4.7.6 Comparing harnesses against raw link performance

General conclusions to notice with respect to the previous test are that the raw link bandwidth can communicate smaller counted arrays at a higher speed than the harnesses. This is presumably due to the extra overheads incurred by the harness processes. Further to this, although 16–32 bytes is a good minimum message size to send between two transputers, if the message is to be routed through other processes, 32-64 bytes a slightly larger message size is recommended in order to keep the percentage throughput obtained between about 60 and 70%.

Comparing the performance of these harnesses with the raw link bandwidth results might be considered unfair. The current test does not just consist of messages travelling through a link, but messages

travelling through two links and a piece of code. Nevertheless, what is encouraging from this comparision is that the amount of bandwidth obtainable through a harness can still be very close to the maximum possible through a link.

4.7.7 Harness D versus harness C

As we have already seen, harness D is not as good as harness C for packet sizes up to 512 bytes. Further, harness D is even poorer in performance compared with harness B below about 40 bytes. This is interesting as harness D was supposed to be more efficient than harness C, it not involving the setting up and closing down of many PAR processes. As we have also seen, the only range of packet sizes where harness D is of comparable efficiency is for the largest packet sizes. Whereas for the most suitable sizes of packet, 64 bytes to 1K, harness D always obtains noticeably smaller throughput than harness C.

To review, harness C only uses on-chip communication for messages going to and from the local worker. There is no on-chip communication for messages passing through. Thus, the harness sets up and closes down two PARs per message. In contrast harness D performs two small on-chip communications and thus four or six context switches for messages that are passed on to the next worker, the same approach used for messages going to and from the local worker. As this is the case, there is the question of which harness of the two will be the most efficient in a real farm where actual work will be performed. This question comes down to the issue of what is the most important in terms of efficiency: the communication bandwidth obtainable by harness C, or harness D's potential to both, interleave differently sized messages and to utilise the transputer's computational resources better. This second point being due to harness D communicating pointers between the harness and the worker, not the actual messages themselves.

What is ultimately more efficient in a farm is an interesting question. Harness D provides less throughput here as the loss in bandwidth due to overheads happens for every message communicated on and off a transputer. In harness C, the bandwidth lost due to on-chip communication happens only for jobs that are performed by that worker.

So far we have only experimented and discussed the fact that harness D provides a lower throughput than harness C for a stream of messages that are all equally sized. When the length of message varies continuously harness D should be able to provide a much higher throughput. This is advantageous, as Cramp and Upstill reported in [CU90] that interweaving jobs of different sizes had load balancing advantages for their application. Harness D would probably be the better harness to use in such a situation.

This issue is looked at further in the next section.

4.7.8 Conclusions

Harness E is the best bidirectional harness here. Further, both bidirectional harnesses are better than the best unidirectional harness here, this being harness C. In decreasing order the best harnesses are: E, F, C, D, B and A. That said the choice of packet size can also reduce the performance of the best harnesses.

One general property these results show is that not using the in-built parallelism of the transputer results in poor performance. Using it in any shape or form reasonably sensibly results in a good performance. The more of it is used the better and better is the performance obtained.

In these results the harnesses form themselves into three very noticeable groups of similar levels of performance,

- group 1: harness A,
- group 2: harnesses B, C and D,
- group 3: harnesses E and F.

From looking at the design of the harnesses this is very much due to the three very noticeable levels of parallelism in these harnesses.

A minimum counted array message should be between about 32 and 64 bytes. A sequence of such messages obtains about 60–70% of the maximum link bandwidth available.

4.8 Harnesses D versus C for variable message sizes interlaced

In the previous section it was discovered that harness C provides a higher throughput of messages compared with harness D. This was a surprise as Welch believed harness D would have a higher throughput, it being designed so the actions of the processes were decoupled and if one communication completed before another, nothing was stopping the process re-engaging the link so another message could be read in.

That said, the previous experiment only used messages of all the same size, thus the above situation was not tested. Thus, it was decided to study these two harness designs further, to see if harness D was more efficient when alternate messages were of different sizes and if one communication could finish before the other, a situation Welch had realised harness C could not respond to.

4.8.1 The test program

The test program worked by communicating a sequence of messages through a harness. The messages were of two lengths interleaved, see figure 16.

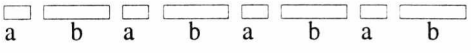


Figure 16: A sequence of interleaved messages

The lengths used were the powers of two, thus giving message lengths from 1 byte up to 16 kilobytes. For each test a thousand pairs of messages were passed through the harness.

The experiment here was performed using the test rig used previously, with a thousand pairs of messages of two different sizes being communicated through the four harnesses. The messages being received by the following,

```
SEQ i = 0 FOR sizes
  SEQ range = 0 FOR sizes
    SEQ
      sync ! TRUE
      clock ? start
      SEQ i = 0 FOR packets
        SEQ
          in ? n::p
          in ? n::p
      clock ? stop
      figures ! stop - start
```

and sent by,

```
SEQ i = 0 FOR sizes
  SEQ j = 0 FOR sizes
    SEQ
      x := 1 << i
      y := 1 << j
      sync ? any
      SEQ i = 0 FOR packets
        SEQ
          out ! x::p
          out ! y::p
```

4.8.2 Testing and results

This experiment was performed in a number of stages. Three versions of harness D were developed. Here we discuss these different versions. How each version of harness D was tested against harness C, and how

each new version was developed from the previous one.

Once the test program was written, the original versions of harnesses C and D were compared.

Looking at the results, unexpectedly harness D obtained a very similar level of performance to that of harness C, and for all of the combinations of message length tested. The results are so much the same that they appear directly on top of one another when plotted, and this graph is of no real use; there being no real 3D cues that helped in visualising the data.

It was only once a version of harness D had been developed that did produce different levels of performance to harness C that a way was found to plot a graph showing clearly that the first results of the first test were nearly identical.

The reason for no large overall improvement in the first test was due to harness D not having any spare buffer space. To illustrate, after a first message had been read, there was nowhere to read a second message into, unless the writing of the first message had been performed. Thus the two separate processes were still locked together synchronously due to the communications they engaged in.

This realised, a second version of harness D was developed. This had two extra buffers. These sat on the channels between the two main processes. Each buffer buffered one pointer.

```
PROC buffer (CHAN OF INT in, out)
  INT p:
  WHILE TRUE
    SEQ
      in ? p
      out ! p
:

CHAN OF INT result.a, result.b, return.a, return.b,
            local, return:
PAR
  result.buffer (in.link, result.a, return.b, 0)
  buffer (result.a, result.b)
  buffer (return.a, return.b)
  merge (local, return, result.b, return.a, out.link, 1)
```

This second version works in the following way. As soon as the result buffer finishes reading in a result, it can pass this on to the buffer and start immediately reading in the next packet from the link. Similarly, as soon as the merge process finishes outputting a message it can pass on the pointer to the buffer and start reading in the next packet from the link immediately. Now neither of the main processes are blocked immediately for output and the two main processes do not have to communicate in lockstep. The reason that was causing the hindrance in performance.

The test program was rerun with these two additional buffers. The performance of this harness was not better, but worse. This was due to a lack of buffer space. Although now the two link processes had had their communication behaviour decoupled, there were no extra buffer slots in which to place any extra messages.

This last version of harness D was modified so that three buffer slots were declared instead of just the two.

```
PAR
  result.buffer (in.link, result.a, return.b, 0)
  buffer (result.a, result.b)
  SEQ
    return.b ! 2
    buffer (return.a, return.b)
  merge (local, return, result.b, return.a, out.link, 1)
```

This extra buffer slot is initially given to the buffer that passes pointers from the merger back to the link buffer. This is so that as soon as the link buffer had a message, not only can it pass this on to a pointer buffer, but there is also a spare message slot into which another message can be read. This way the two

processes that sit on the links are decoupled both in terms of communication behaviour and in terms of buffer space allocation. This third version of harness D was much more efficient than harness C when adjacent messages were of different sizes.

All the results with first message size were plotted against second message size, see figure 17. However, it was only really for this last test was there any major difference in performance between the two types of harness, except for when both packet sizes were very small, an aspect of transputer behaviour not of interest due to the overhead of setting up a small communication.

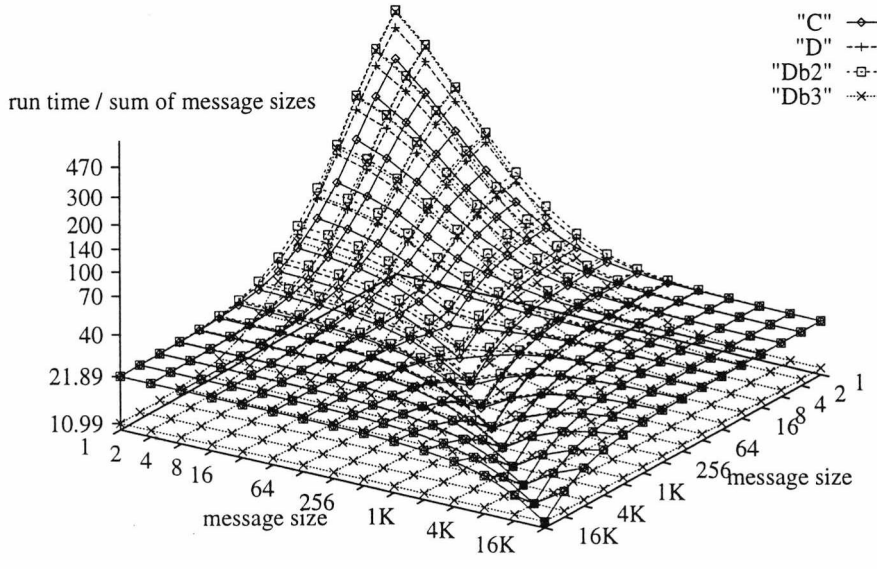


Figure 17: Time to interleave messages of different sizes

The definite result that comes through from this graph is there is a marked difference in performance for the two harnesses when alternate messages are of different sizes.

As we are only interested in the case when adjacent message sizes are either the same or different, it makes senses to take one message sizes as a constant and plot a 2-D graph. Indeed the behaviour is best seen from taking one particular message (here 256 bytes, which is a suitable size for messages) and comparing the performance of each version of harness D to the performance of harness C, see figure 18 (top of next page).

Here for completeness the results for the first two versions of harness D are plotted here. As can be seen the performance of both is worse than harness C throughout the range of message sizes.

4.8.3 Conclusion

Clearly adding three buffer slots to the design of harness D largely increases the throughput when messages vary in length by any large degree. Just having two processes that pass pointers between themselves is not enough. As was expected, having two extra pointer buffer processes results in a less efficient execution compared with not having them. Ultimately it is the use of additional buffer slots that gives the required parallel slackness, the buffer processes are only needed in order to pass these extra buffer processes around.

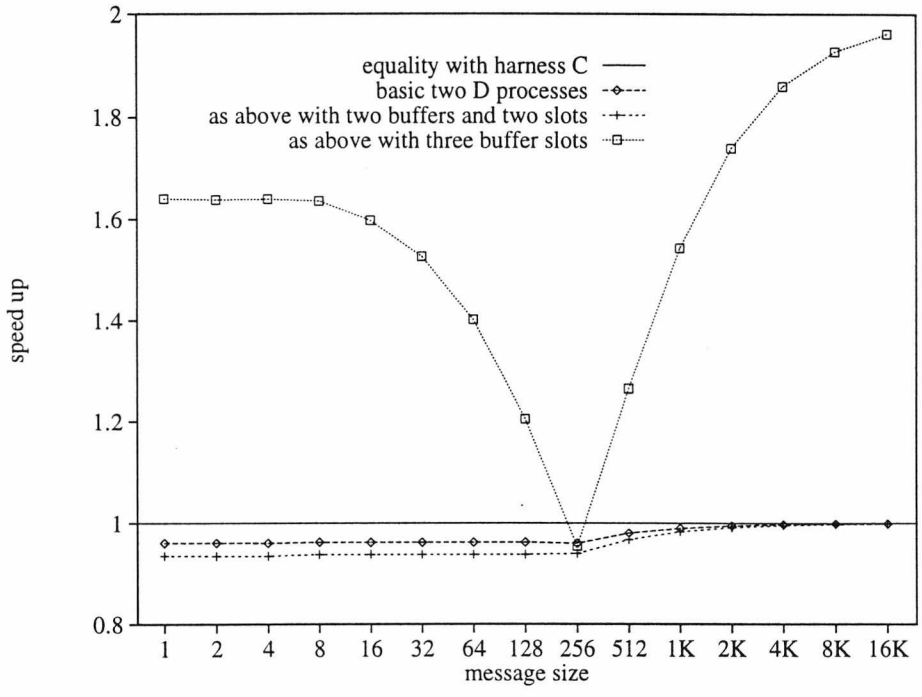


Figure 18: Speed-up of harnesses C and D for messages of 256 bytes interleaved with other sizes

4.8.4 Discussion

It has been established that harness D works well in this situation when two message sizes are interleaved. However, in real farms if message lengths vary they are likely to do so over a range. With a large range of message sizes, it is possible a large message may be followed by a number of shorter messages that altogether are smaller than the first. In this situation a number of buffer slots would be required so all the shorter messages could be buffered and thus the stream of messages kept moving. Two solutions to this are presented.

One approach would be to have enough buffer slots and enough single place buffer processes, one process for each extra message that needs to be held. There are two disadvantages to this. Firstly, the number of buffer slots and processes required must be known in advance. Secondly, all messages will be passed through these processes continually, resulting in dramatically increased amounts of on-chip communication and context switching, thus adding a constant additional execution overhead to the harness.

Another solution is to use a process that can buffer a variable number of messages. The disadvantage of this approach is that it needs to perform an `ALT` in order to discover whether it is to perform an input next or an output. This instruction is time consuming to execute and thus the level of performance obtainable would be impaired. In this case the `ALT` would need to be executed twice for each message communicated, once when buffered and once when output (also see 4.3.2 on page 39). The process wishing to receive a message from this buffer would have to issue a prompt.

If the range of message sizes varies greatly, the first method may be the most beneficial. If very large messages are only transmitted occasionally there will not be an advantage in streamlining the harness any further. It is more important that a harness is efficient most of the time, not in occasional situations. If faced with this situation we, as implementors, need to find the right balance.

4.9 Studying the breakdown of harness efficiency

As the length of time taken to communicate messages was now known, job compute times could now be compared with message communication times. Thus the point at which the efficiency of the harnesses broke down could be found.

The experimentation here was performed with a worker that could simulate the worker of any application and a farmer and a harvester that could simulate the farming out of any application. With this set up each harness could be studied completely independently from any application. This is performed through the use of a range of job compute times and messages lengths. All that now needs to be observed is which harness is the most efficient for all possible applications.

4.9.1 Processor Farm Test Rig Design

This experiment required a new program that would perform tests on full farms.

There are nine aspects to the development of this program. These are,

1. the construction of the artificial worker,
2. the variable distribution of job times,
3. how the job time and result message length were encoded into the job message,
4. the farmer and the harvester,
5. the test rig,
6. the measurement of the bandwidth to the end worker,
7. the overall shape of the program,
8. how multiple runs were performed, and,
9. the compilation configuration was used.

Worker simulation

It was important the worker process can accurately simulate all applications. As discussed in chapter 3 the simulation needs to be identical to that of a real worker from an external point of view. This means the length of time taken to execute a job and the length of a job message must be totally variable and also completely independent of one other.

As such a study naturally focuses around what length of time it takes to process a job, it was initial thought that using the transputer's in-built timer would be possible. The most simple and obvious method of waiting on a timer is,

```
SEQ
  timer ? now
  timer ? AFTER now PLUS run.time
```

It was obvious this approach is not appropriate as the `timer ? AFTER` construct is descheduled until the time is reached. Thus, if this algorithm was used the worker process would be descheduled from the C.P.U. throughout the length of the job. This approach would be far from being an accurate simulation of a real application worker.

The method considered for some time was to continually look at a timer in a `WHILE` loop,

```
SEQ
  ...  receive work

  timer ? now    -- get start time immediately
  ...  decode job execution time and result size
  finish.time := now PLUS job.length
  timer ? now    -- we've done something, get the time
  WHILE finish.time AFTER now
    timer ? now

  ...  send result
```

This piece of code would not be automatically descheduled and so the worker would be continually executing within the C.P.U. whenever possible. However, if in order to execute a part of the harness, the worker was descheduled for a short space of time, when the worker was rescheduled the timer would still have been advancing throughout the time the worker was descheduled. So it would appear that the worker had still been executing and effectively the harness had been executing for free. Further, it is possible that the worker could be rescheduled to find that the finish time had passed, thus not making the execution of the harness transparent all of the time. This approach of using the timer to simulate a real worker is unsuitable.

Thus, it seemed that the only way to accurately simulate "real work" was to actually perform some. By using an actual process, work would be performed when the process was executing within the C.P.U. and work would be stopped when the process was descheduled. This approach also has all the advantages of a real worker. It would take up memory, and, as with most real applications, it would contain instructions that take quite some time to execute and can not be immediately interrupted to switch contexts. This would delay the handing over of execution to the harness. Invariably the transputer can only switch from a low to a high priority process after the current instruction has finished executing. If this instruction takes a long time, such as in an integer or remainder division (the longest two) this can slow down the time it takes a transputer to respond to external events, such as link communications. Thus, this increases the time it can take before a context switch is performed. In terms of long calculations it is more common to use floating point numbers. The F.P.U. on the T800 also can stop the longer instructions at a number of suitable places. In this study the worker used a small floating point calculation, as many applications involve real numbers. Thus finally the following sequence of code was developed.

```
REAL32 p, x, x2:
SEQ
  p := -1.0(REAL32)
  x, x2 := 0.0(REAL32), 0.0(REAL32)
  SEQ k = 0 FOR 3
    SEQ
      x := x2 + p
      x2 := x * x
  x := x + x
  x := 0.0(REAL32)
  x := 0.0(REAL32)
  x := 0.0(REAL32)

  x := 0.0(REAL32)
  x := 0.0(REAL32)
  x := 0.0(REAL32)
```

This takes ten ticks of the transputer's high resolution clock to execute. This was executed in a loop to achieve any desired length of job run time.

The distribution of job times

The other issue to sort out was what jobs should be farmed out. It was decided to use a hundred jobs per worker; with this being scaled for the number of workers being used in any one test. This was done so that results for the different sized farms could be compared directly and linear speed up figures can be computed.

In order to make the results of the study as valid as possible, the times it took to process the jobs have been varied according to a distribution found in real applications. One advantage of using a distribution of job run times, instead of having all jobs take the same length of time to execute, is this can also reduce the number of times that more than one worker will want a job from a job distribution process simultaneously. This collision can reduce efficiency as workers will be starved of work for short periods of time.

The first distribution considered to be appropriate was the normal distribution. This being the most

common in statistics. However, three real farmed applications were looked at in order to see which distribution were found. All three applications had job distributions that were Poisson in nature. These applications are discussed below.

The corners of a picture of the Mandelbrot set take the least time to compute. The rest of the picture takes progressively more time the further into the centre of the picture one gets. Near the centre of the picture fewer and fewer points take more time.

In ray tracing, the sky parts of a scene (some times quite large) involve no reflections at all. Most of the scene invariably involves at least one reflected light ray. There being progressively less and less areas of the picture that involve a larger and larger number of reflections.

With the protein sequence database there is more of a lead in and a steeper trailing edge than in the Poisson distribution. There are a fair number of short sequences, but most are in the 200–300 residue region. There are a few sequences longer than 2000 residues.

From this it was decided to use a Poisson distribution. This was implementing in the following way. A sequence of 100 numbers was generated. The constant,

```
VAL njob IS 100:
```

is used throughout these tests. These numbers were produced from a routine in the NAG library [Gro] that produces numbers that fit into a Poisson distribution. The distribution used had a mean of one hundred. Thus for any particular average of job run time required for any application under test, this sequence of values could be scaled to produce an application with the appropriate Poisson distribution of job run times.

As what is of interest here is how balanced the farms are, all of the jobs needed to take roughly the same length of time to be processed. Thus the deviation of the distribution should be reasonably small. There should be just enough to add some realism in terms of different jobs taking different lengths of time to be processed. Thus the longest job run time was only a factor of one and a half longer from the shortest job run time (127 compared with 77). So, as this range of values is reasonably small, the average number of jobs performed by each worker should be at least reasonably equal numerically when a farm is well balanced.

We have already mentioned that jobs of different lengths can help prevent simultaneous requests for jobs from the same job distribution process. So as to reduce this collision further, an attempt was made to reorder the sequence of values so any two adjacent values were not close to each other numerically.

On the related note of message length, it was decided to have all messages the same length, i.e. no distribution of message length, and to just vary the length of time it took to process the jobs. This was to keep the test program easier to write.

Adding the Poisson distribution to the program generated a small problem. When 100 jobs were being farmed out to a single worker via a double harness, 50 jobs should be farmed out using both channels for job delivery. So the Poisson distribution would be used correctly in this situation, the code was modified for these runs. The first 50 jobs being sent into the farm from one end, the second 50 being sent from the other. No check was made to see if the length of time to process the first 50 jobs was the same as the second 50.

The contents of jobs and results

There are two parameters to encode,

1. m, the length of the average message, and,
2. j, the time to process an average job.

The length of the message is the length of both the job sent out and the result produced. This length is known by the farmer and can be used directly to indicate the length of the job being sent out,

```
INT msg.len:
[max.msg.len]BYTE job:
SEQ
  jobs ! msg.len::job
```

Getting a result message to be the same length as the job message is simply achieved in the worker by reusing the value received for the length of the job message. For instance,

```
SEQ
  job ? msg.len::job
  ...  process job
  result ! msg.len::job
```

The job run time was encoded into the first INT of the job message. This restricts our minimum message length to four bytes (the number of bytes in an INT on 32 bit processors). Again the farmer knows the value of this parameter and can place this value into each job message. The relevant part of the farmer is,

```
VAL njob IS 100:
VAL bytes.per.int IS 4:
VAL jobs IS workers * njob:  -- 100 workers per job
VAL job.lens IS [  98, 105,  93,  97, 107,  90,  97, 115,  98, 109,
                  119, 101, 112,  98, 110, 120,  89, 114, 100,  88,
                  113, 106, 108,  91,  87,  93,  86, 102,  91,  93,
                  102, 106,  88, 127, 104, 108, 106, 101, 109,  77,
                   90, 100,  99,  95,  93,  90, 101, 106,  89, 109,
                   86, 105,  89, 100, 106, 117, 105, 101,  98, 123,
                  101,  88,  95, 116,  88, 103,  92,  90,  98,  97,
                   94, 109,  85,  90,  88, 103,  88, 106,  93,  89,
                  114, 101, 115,  97,  89,  92, 115, 104,  80,  89,
                   98, 104,  90,  97, 105,  98, 104,  90, 102, 111 ]:
[njob]INT job.times:
INT msg.len:
[max.msg.len]BYTE job:
SEQ
  from.testrig ? msg.len; run.time.scale
  SEQ i = 0 FOR njob
    job.times[i] := (job.lens[i] * run.time.scale) / 100
  to.harv ! jobs  -- number of jobs to read in, also
                  -- prompts harvester to read timer
  SEQ j = 0 FOR workers
    SEQ i = 0 FOR njob
      SEQ
        INT count RETYPES [job FROM 0 FOR bytes.per.int ]:
        count := job.times[i]
        jobs ! msg.len::job
```

It was also decided to keep a tally of how many jobs were performed by each worker. This was done by having each result message contain the identification number of the worker that generated it. This number was created by the PLACED PAR replicator. The harvester then created a count for the number of results produced by each worker.

The storage of the identification number in the result message was performed in a similar manner to the above method of storing job run times, using the first INT in the array. Here the same abbreviation was used to gain access to the job run time value in order to perform some work and in order to store the worker's identification number into the result message,

```
PROC worker (CHAN OF REQ req, CHAN OF JOB job,
             CHAN OF PACKET result, VAL INT id.number)
  ... variables
  WHILE TRUE
    SEQ
      req ! TRUE
      job ? msg.len::job

      INT job.time RETYPES [job FROM 0 FOR bytes.per.int]:
      SEQ
        SEQ i = 0 FOR job.time
          ... do something for 10 ticks

        job.time := id.number

        result ! msg.len::job
:
```

The tally itself was performed by the harvester using these values as an index,

```
SEQ
  results ? msg.len::result
  INT id RETYPES [ result FROM 0 FOR bytes.per.int ]:
  work.done.by[id] := work.done.by[id] + 1
```

The contents of this tally array was cleared before each simulated application was run.

Designing the farmer and harvester

The structure of the two double harnesses E and F had a number of implications for the design of the front end of the farm. In total at least five links are needed in this situation. Two for giving out jobs, a further two for collecting results and one for communicating with the terminal and filing system. Obviously more than one transputer was going to be needed to provide this amount of interconnection. It was decided to organise the front end by having two processors, one for running the farmer and another for running the harvester. The farmer would use two links to give jobs to the workers. Similarly the harvester would use two links to collect results. These leaves two links each on both transputers. One pair can be used for communication between the farmer and the harvester. The link remaining on each processor can be connected to the test rig transputer.

It was decided to use this configuration for the testing of all six harnesses so that fair comparisions could be performed between both the single and the doubled up harnesses.

The method used by the farmer and harvester to drive these doubled up harnesses was simple, split the number of jobs in half and give each half to each end of the pipeline. The code from this version of the farmer looked as follows,

```
SEQ
  SEQ i = 0 FOR njob
    SEQ
      job.times1[i] := (job.lens[i] * run.time.scale) / 100
      job.times2[i] := job.times1[i]

  PAR
    SEQ j1 = 0 FOR workers
      SEQ i1 = 0 FOR jobs >> 1
        SEQ
          INT count RETYPES [job1 FROM 0 FOR bytes.per.int]:
          count := job.times1[i1]
          jobs1 ! msg.len1::job1
    SEQ j2 = 0 FOR workers
      SEQ i2 = 0 FOR jobs >> 1
        SEQ
          INT count RETYPES [job2 FROM 0 FOR bytes.per.int]:
          count := job.times2[i2]
          jobs2 ! msg.len2::job2
```

This will still produce a balanced implementation as both of these two sets of jobs are going to the same workers. Also both ends of the farm are identical in all major respects. One end will be connected through more switch chips than the other end, but this should only affect the bandwidth of communication but not the actual demand for work.

The above code will run much faster than an equivalent system that uses an ALT to distribute the work as needed. Consider,

```
SEQ i = 0 FOR jobs
  PRI ALT
    req1 ? any
      to.link.buffer1 ! msg.len::job
    req2 ? any
      to.link.buffer2 ! msg.len::job
```

As well as involving an ALT, this approach also requires two buffer processes to perform the appropriate requesting.

The fact the farmer and harvester are not only configured as separate processes, but are also on completely separate processors leads to a problem with performing timings accurately. It is only the farmer that knows when it gave out the first job and it is only the harvester that knows when the last result is received. Thus, only one can perform the timings, as being on different processors these processes do not share a common clock. In fact either process can perform the timing, as long as both the farmer and the harvester can be connected without intermediate buffering.

In this implementation the harvester performed the timings. The farmer sent a message to the harvester synchronously just before it started the replicated SEQ (equivalent to two assignments) to give out the jobs. The harvester made a note of the time twice, once directly after receiving this message and again after receiving the last result.

```
SEQ
  ...  bandwidth calculation
  WHILE TRUE
    SEQ
      SEQ i = 0 FOR workers    -- clear work.done.by
        work.done.by[i] := 0
      PRI PAR
        SEQ
          from.farmer ? njobs
          clock ? start
          SEQ i = 0 FOR njobs
            SEQ
              results ? msg.len::result
              INT id RETYPES [result FROM 0 FOR bytes.per.int]:
              work.done.by[id] := work.done.by[id] + 1
          clock ? stop
          to.farmer ! stop MINUS start; workers::work.done.by
        SKIP
```

When doubled up harnesses were used, some additional setting up within the harvester was needed to read in the two results streams in parallel.

```
SEQ
  from.farmer ? njobs
  clock ? start
  work.done.by1 IS [work.done.by FROM 0 FOR workers]:
  work.done.by2 IS [work.done.by FROM workers FOR workers]:
  PAR
    {{{  harvester 1
    SEQ i1 = 0 FOR njobs >> 1
      SEQ
        results1 ? msg.len1::result1
        INT id RETYPES [result1 FROM 0 FOR bytes.per.int]:
        work.done.by1[id] := work.done.by1[id] + 1
    }}}
    ...  harvester 2
  clock ? stop
  to.farmer ! stop MINUS start; (workers << 1)::work.done.by
```

The separate test rig

As with all the programs built previously, the first processor dealt with the filing of results, and in this case, the generation of the application parameters that are to be farmed as well. Again this code was placed on a separate processor so as to prevent the execution of this code interfering with the running of the experimentation. Also as before the system library code were on a separate processor so all parts of the code under test were in on-chip memory.

Measuring the throughput from the end worker

So far it was known the raw throughput of both a transputer's link and also the raw throughput of the harnesses when they were just passing traffic on. Here we could also obtain some measurements from a real farm. It seemed a good idea to find out what is the throughput to the harvester from the worker furthest away from it. This could easily be done as soon as the program was loaded and just before the application experiments were run.

Timing this accurately was easy to achieve. As the line of workers was set up so it could run the doubled up harnesses E and F, the harvester processor was connected to both ends of the line. Thus as well as the end workers at each end being the furthest away from the harvester, they are also directly connected to it via the link that deals with results travelling in the other direction. So after the harvester reads the clock, it then sends start messages to the end workers to start a transfer, the messages get there without any form of buffering to slow them down and corrupt the accuracy of the timings.

As before, it was considered important to make sure all parts of the program were initialised. The most effective and simplest method to achieve this is to send messages through the whole of the program. Only when all of the code was loaded and had started running the harness would the message get through to the other end. The code was written so when the program loaded only the end worker would be running special code, the rest of the farm would execute the harness immediately the program was loaded. Before the worse case throughput measurements were performed a synchronisation was sent from the end worker to its harvester. Thus this message would travel through the entire farm. The worse case throughput measurement was then performed as follows. The harvester would send a message to the worker processor next to it, then read the time. This message would be received by a process on the end worker that performed this test before becoming a normal harness buffer process. The throughput measurement transfer would be started and a hundred messages would be sent. The harvester would read the time immediately after these messages had been transferred.

This throughput measuring code in the harvester was,

```
PRI PAR
  SEQ size = 0 FOR sizes
    SEQ
      results ? n::message  -- settle synchronisation
      go ! 0::message    -- start message
      clock ? start
      SEQ i = 0 FOR 100
        results ? n::message
      clock ? stop
      bandwidth.out ! stop - start
  SKIP
```

The code in the end worker that this communicated with was,

```
SEQ size = 0 FOR sizes
  SEQ
    results ! 0::message  -- settle synchronisation
    workmate.results ? n::message
    n := 1 << size
    SEQ test = 0 FOR 100
      results ! n::message
```

For the doubled up harnesses the above piece of code was run on both end workers with a transfer size of 50 instead of 100. The code in the harvester used to communicate with both end workers was,

```
PRI PAR
  SEQ size = 0 FOR sizes
    SEQ
      PAR
        SEQ
          results1 ? n::message1    -- settle
          start1 ! 0::message1    -- start transfer
        SEQ
          results2 ? m::message2
          start2 ! 0::message2
      clock ? start
      PAR
        SEQ i = 0 FOR 50
          results1 ? n::message1
        SEQ i = 0 FOR 50
          results2 ? m::message2
      clock ? stop
      bandwidth.out ! stop - start
  SKIP
```

Overall program shape

Putting this altogether gave the overall structure of a processor farm hanging off a single processor that contained a test rig,

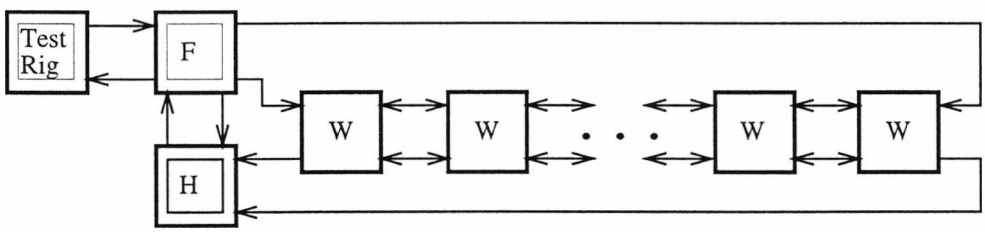


Figure 19: Full processor farm test rig

How multiple runs were performed

Multiple runs of each simulated application were performed and an average taken. The programming for this was in the test rig. Each set of application parameters were passed to the farmer five times. The results were collected and stored and an average overall run time was calculated.

Compilation

As had been said, all of the harnesses were compiled with usage checking switched off, in order to compare the true performance of the different approaches to harness design. Here this also provided convenience when recompiling the different harnesses, as there was no need to turn compilation flags on or off something that could also easily be forgotten invalidating the results.

In any implementation all of the flags can be turned off to add a speed up advantage.

4.9.2 The experimentation performed

Once the program was written the range over which to perform the study had to be decided upon.

Before the Poisson job run time distribution was developed some testing had been performed. This being in factors of two across a wide range of application parameters. From this it was found that the area

of interest lay in the area from 32 bytes to 4 kilobytes and from 0.06 milliseconds to 250 milliseconds (or 60 microseconds to a quarter of a second). Once the Poisson distribution had been developed the program was set to test across this range. The relevant part of the test rig is,

```
INT bytes, delay.loop, time, run, total.run, len:
SEQ b = 5 FOR 8
  SEQ
    SEQ t = 6 FOR 13
      SEQ
        bytes, delay.loop := 1 << b, 100000 >> (20 - t)
        time := 10 * delay.loop
        total.runs := 0
        SEQ i = 0 FOR 5
          SEQ
            to.farmer ! bytes; delay.loop
            from.farmer ? run; len:: work.done.by
            ...  output results
        ...  calculate and output averages
```

This program was then run for the six harnesses under test and for each application mapping under test. The sizes of farm were also selected as powers of two: 1, 2, 4, 8, 16 and 32. The tests for each farm size were all done in one session so both the same transputers and the same wiring set up were used for each size of farm.

Thus the study set out to look at the breakdown in efficiency of the different harnesses across the range of different applications and farm sizes.

4.9.3 Results

When looking at the results obtained, the following questions were considered.

1. What is the efficiency of the harnesses for the varying application's parameters?
2. What are the causes of this behaviour?
3. What is the behaviour for different farm sizes?
4. How well balanced is the work load?
5. How good is the model of efficiency developed earlier?

Comparing harnesses A, B and C

Looked at first was the basic issue of which harness is the most efficient. Here, for simplicity of viewing, graphs are plotted containing three harnesses each. Efficiency is plotted against job computation time, j, and message length, m.

The first graph plotted, see figure 20 (top of next page), is of harnesses A, B and C running on a farm of 8 workers. Here we are interested in which harness breaks down last. The easiest way to examine the relative breakdowns of these harnesses is to look at the efficiency curves at the back of the surfaces where the 4 kilobyte jobs are. Also notice the slopes's different direction for smaller message sizes, this is due to the cost of starting up communications.

Harnesses B and C are very close in terms of performance, they both breakdown when jobs are 4 kilobytes long and take 31.3 milliseconds to process. Nevertheless, it can be seen that harness C is slightly more resilient to communication bound applications.

To a certain extent these results were expected. However, what was unexpected was that while harness A has the poorest performance in terms of being the first harness to breakdown, it is the most efficient of the three for comfortably compute bound mappings. This is of great significance as we are ultimately interested in our farms performing efficiently as well as being able to deal with demanding applications.

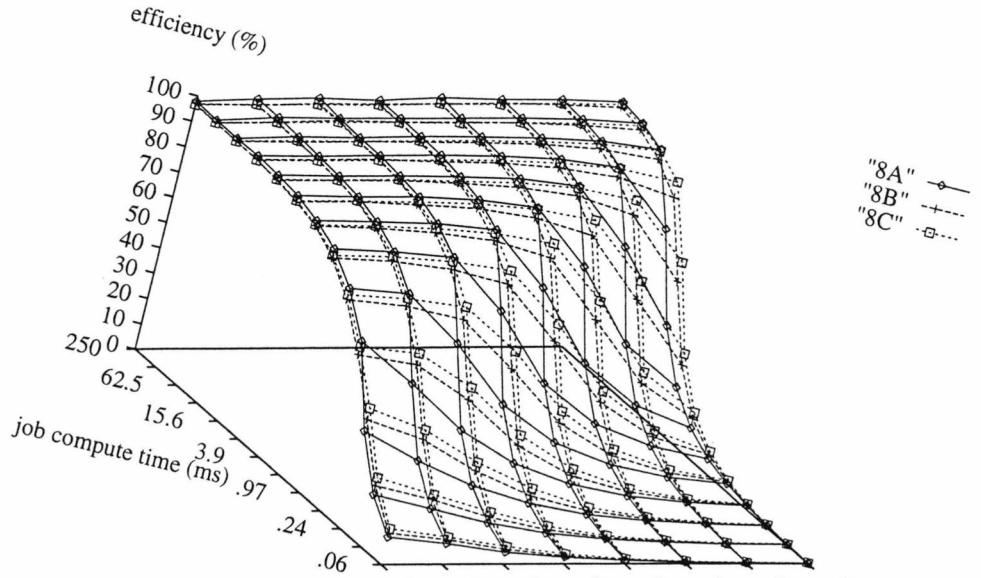


Figure 20: Breakdown of harnesses A, B and C on 8 workers

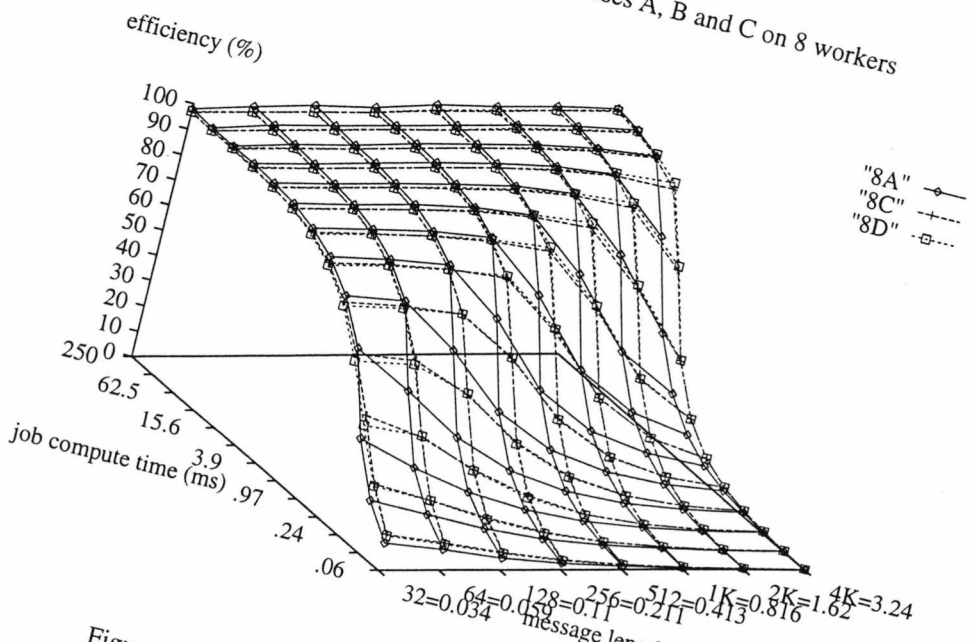


Figure 21: Breakdown of harnesses A, C and D on 8 workers

Comparing harnesses A, C and D

Here we compare the two harnesses that so far are more appropriate in one situation or another, harnesses A and C, against the performance of a third, harness D.

As we can see in figure 21 (bottom of previous page) harness D breaks down slightly later than harness C. What is interesting is that harness D performs better than harness C here even though all messages are the same length. Thus, on a real farm it is more efficient to pass pointers to messages between processes than to set up and shut down PARs in one process. It is assumed that even harnesses A and C would benefit from this approach.

Harness D is still only as efficient as B and C. Harness A is still the most efficient overall, at least for this part of the farm size application parameters looked at here.

The fact that harnesses B, C and D can all farm out more communication bound applications than harness A, again proves that having something in the way of a parallel strategy, makes better use of the parallelism of the transputer.

Comparing harnesses D, E and F

Here we compare harness D, the most resilient to breakdown so far, against the two doubled up harnesses, harnesses E and F.

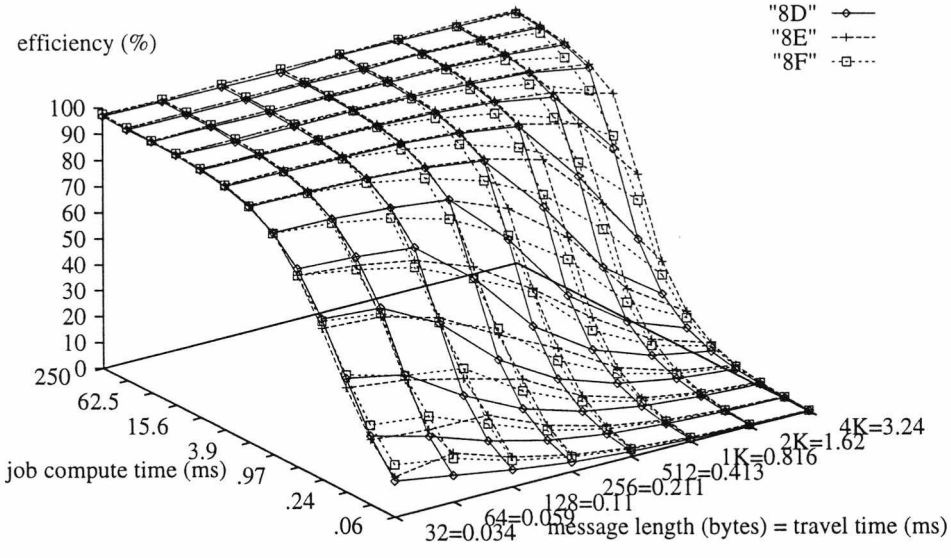


Figure 22: Breakdown of harnesses D, E and F on 8 workers

Looking at figure 22 harness E breaks down much later than harness D, as we might expect from the throughput results. The increased throughput results in more supply and consequently more resilience to breakdown. There are more jobs out of the farmer in any given period of time.

Harness F does not perform as well as harness E, although the harness is doubled up, the harness itself stems from a more primitive design strategy and as a result rolls off both earlier and more gradually. For example, examine the far trailing edge, in this case 4 kilobyte jobs break down when they take between a quarter of a second down to an eighth of a second to process. Harness F again ultimately can be more efficient than the unidirectional harnesses for compute bound applications, but breaks down before the best unidirectional harnesses. This is probably due to the overheads of communicating large arrays around on-chip memory.

From the results looked at so far, harness D is the unidirectional harness that is the most resilient to breakdown, and a doubled up version of harness D, harness E, is the bidirectional harness that is the most resilient to breakdown.

Summary — harnesses A, D and E

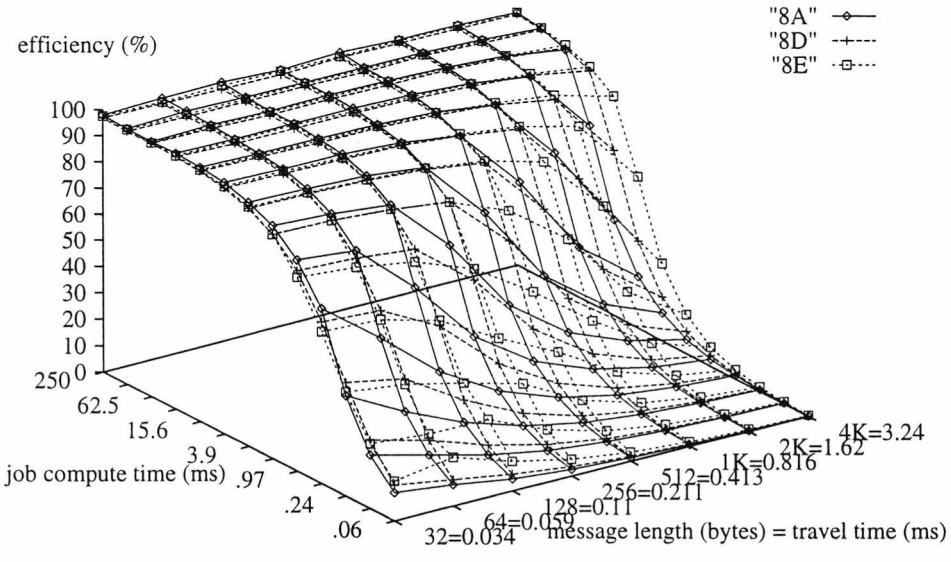


Figure 23: Breakdown of harnesses A, D and E on 8 workers

Summarising what we have so far, see figure 23. Harness A is the most efficient harness for compute bound applications. This seems sensible as when an application is compute bound only a small amount of bandwidth is needed from a harness, and thus it is perfectly adequate to use a harness that provides a small amount of bandwidth, but is efficient due to being small and quick to execute. Harness A has a very small amount of code. It also has a short message latency. After receiving a message, harness A can output it immediately, there is no ALT or PAR to close down.

Harness E breaks down last out of all of the harnesses studied here. Out of the unidirectional harnesses, harness D farms out the most demanding application with the greatest efficiency.

Farming is about supply and demand

When comparing these surfaces against one another two properties were noticed.

The first was the way in which the performance tails off in these efficiency breakdown graphs is the same no matter which parameter is being varied. This would indicate it is always the same type of behaviour that leads to harnesses breaking down. By looking at what happens when both of the parameters are changed we can see this is indeed the case. Increasing the length of messages results in it taking longer time to communicate each message, this in turn results in fewer messages travelling around the farm and thus in an implementation that is more communication bound. Similarly, reducing the time it takes to process a job results in more jobs being needed in a given time period thus requiring more bandwidth, this may not be available. Again this results in an implementation that is more communication bound.

The second point noticed was that when looking along one axis, the curves rolled off in a similar shape and in the same order as in the harness throughput test.

From these two points it was decided to plot a graph of increasing job compute time against efficiency for all six harness. This was done for farms of eight workers with all messages being 1 kilobyte in length.

Harness C has been omitted from this graph as it exhibits exactly the same performance of harness D and can not be seen on this scale.

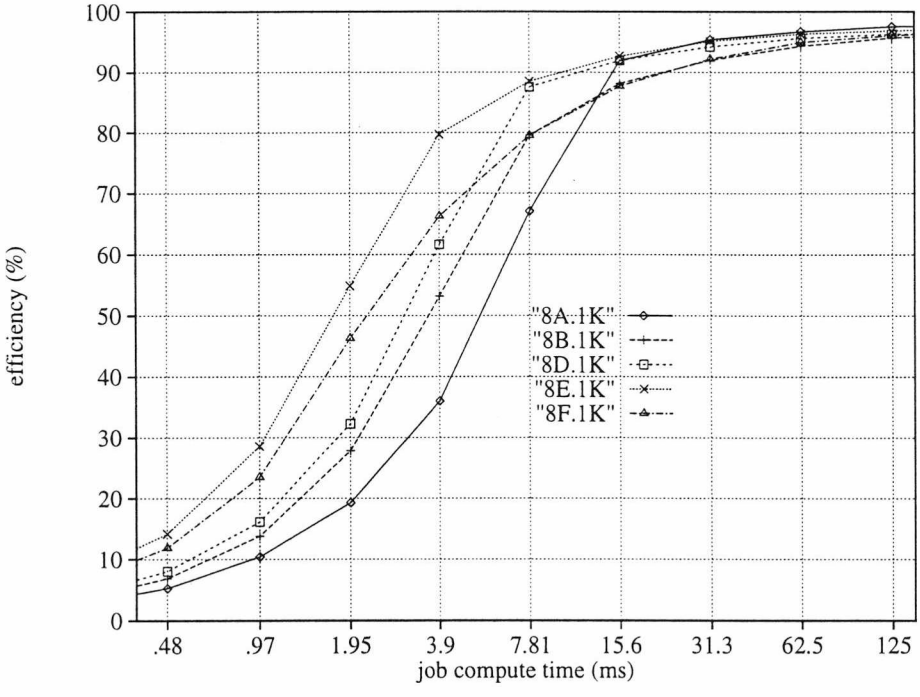


Figure 24: Breakdown of harnesses for 1K jobs on 8 workers

In figure 24 (top of next page) the shape of the curves we can see is the order in which the harnesses rise up and start to provide a high degree of efficiency. The shape of these curves and the order in which they rise up is also the same as in the graph of the harnesses throughput, see figure 15 (page 63). Thus it can be said the throughput of a harness and its general capacity to deliver jobs, and thus how the efficiency of that harness breaks down, is directly related. This seems logical. The higher the bandwidth there is through a harness, the greater the job supply rate will be and in turn the larger the number of demanding application mappings that harness can farm out efficiently. This leads to an important conclusion, efficient farm implementation comes down to supply and demand. Thus, if a farm is to be compute bound, the farmer and the communications system must be able to supply jobs at least at a quicker rate than they can be processed. This is also captured by the equation,

$$\frac{B}{m+s} \geq \frac{w}{j}$$

developed earlier in this chapter.

As the farmer only has a certain amount of supply, the amount of work a harness can only approach this maximum bandwidth. Thus, in conclusion, if we want a harness that can farm out demanding applications, we need a harness that has a high throughput. The unidirectional harness that is the last to break down here is harness D. The bidirectional harness whose efficiency breaks down last is harness E. These harnesses also have a very high throughput.

From looking at the top of the slope, see figure 25 (top of next page), we see again that where the implementation is compute bound, harness A is the most efficient harness, even more efficient than harness E which has the most parallel design.

Plotting efficiency logarithmically

Another point observed was that the breakdown of these harnesses are curves that roll off over a period of about eight doublings of either application parameter. This is about an order of magnitude and so it

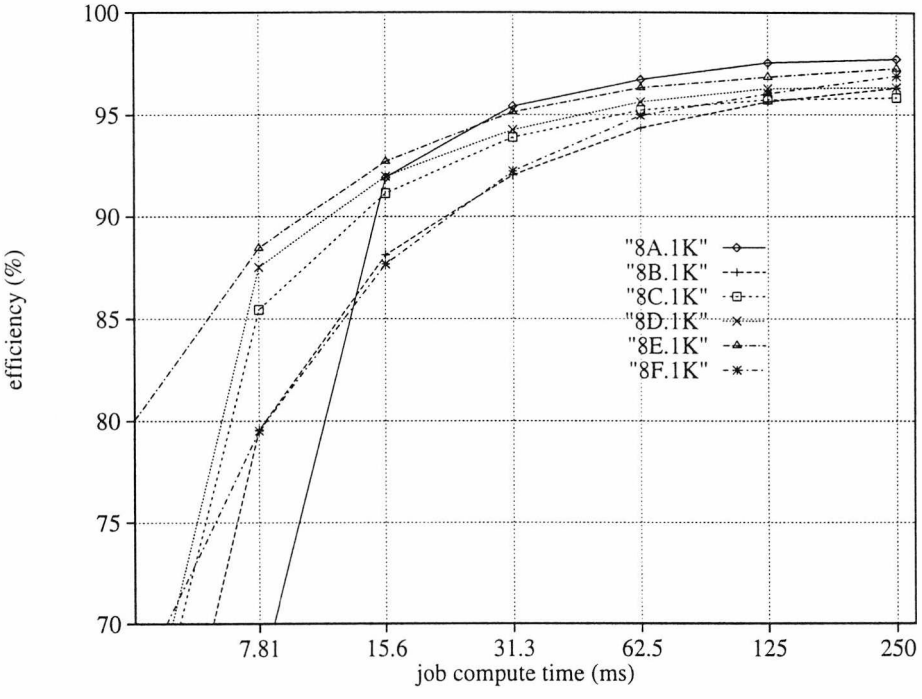


Figure 25: Top of breakdown of harnesses for 1K jobs on 8 workers

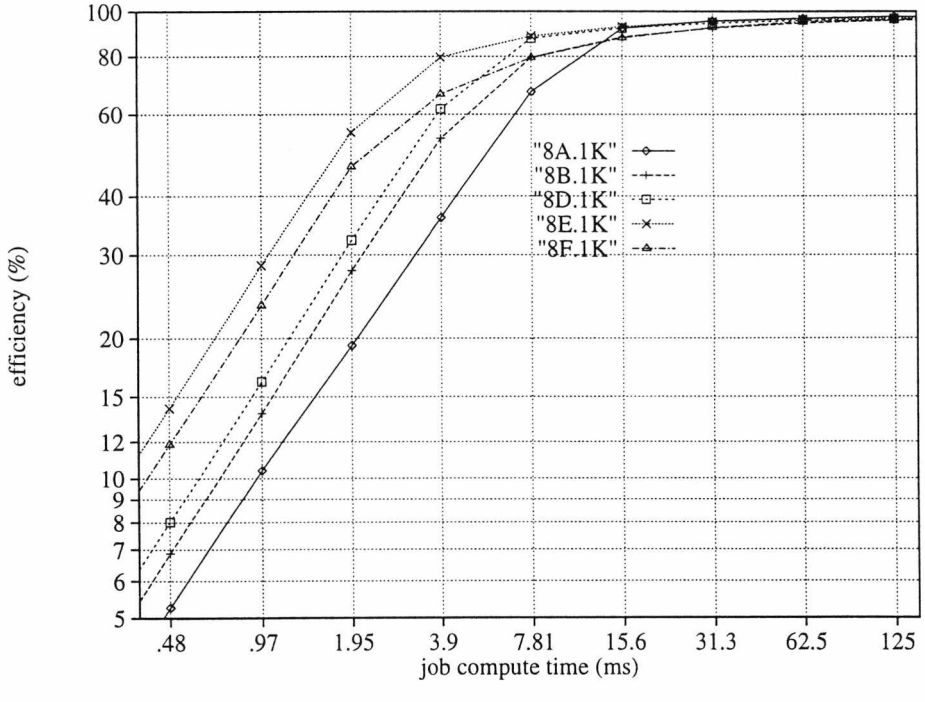


Figure 26: Log of breakdown of harnesses for 1K jobs on 8 workers

was decided to see whether these breakdowns were logarithmic. This seemed sensible as both of the application parameters were plotted logarithmically.

From looking at figure 26 (bottom of previous page) we can see this is indeed the case after the harness has broken down, the slight curve on this graph, it tails off in a straight line down, as we have reached the point where we are communication bound. Here the application is being performed as soon as it arrives, what we are timing here is the work being communicated, not the work being performed and the communications happening transparently.

Other sizes of farm

So far all of the results looked at have been for farms of 8 workers. Here we look at all the other farm sizes tested. To aid comparison, these five other figures have been grouped together on the next few pages. For clarity just the performance of harnesses A, D and E are shown, the other harnesses may be mentioned in discussion.

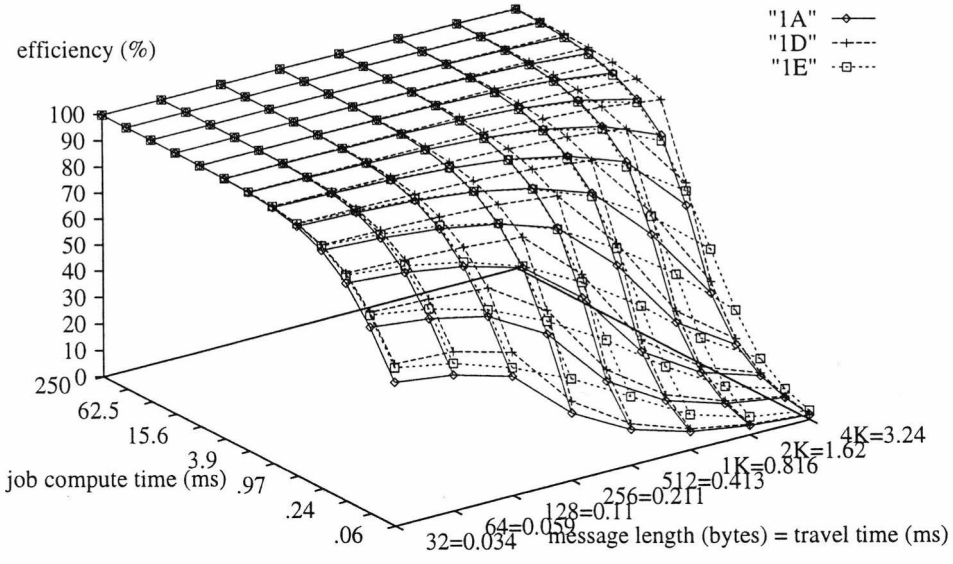


Figure 27: Breakdown of harnesses A, D and E on 1 workers

For farms with 1 worker, see figure 27, harness D is the best harness for all compute bound mappings, both those that are highly compute bound and those that are highly demanding. It is the most resilient to breakdown, even more so than harness E that here is only just as efficient as the other harnesses B, C and F. Harness D, and in fact all the harnesses, are also just as efficient as harness A for compute bound applications. This is presumably due to the fact that only link buffers are in use here as there is no fanning out to perform. It might be expected for harness D to be slightly more efficient as it is only communicating pointers not the actual messages themselves.

For farms with 2 workers, see figure 28 (top of next page), the harness most resilient to a breakdown of efficiency is harness D. For compute bound applications harness E is just slightly more efficient than harness A. This could be due to jobs being sent in directly to both workers. A virtue that comes from the design of harness E sending jobs to both ends of a pipeline. Also, although it is not of any real relevance, due to the slower roll-off characteristic of the breakdown curve of the doubled up harnesses, harness E is also the most efficient harness for some communication bound mappings here.

The results get more interesting for 4 and more workers, see figures 29 (bottom of next page), 30 (top of page 86) and 31 (bottom of page 86), as job distribution processes are placed on workers.

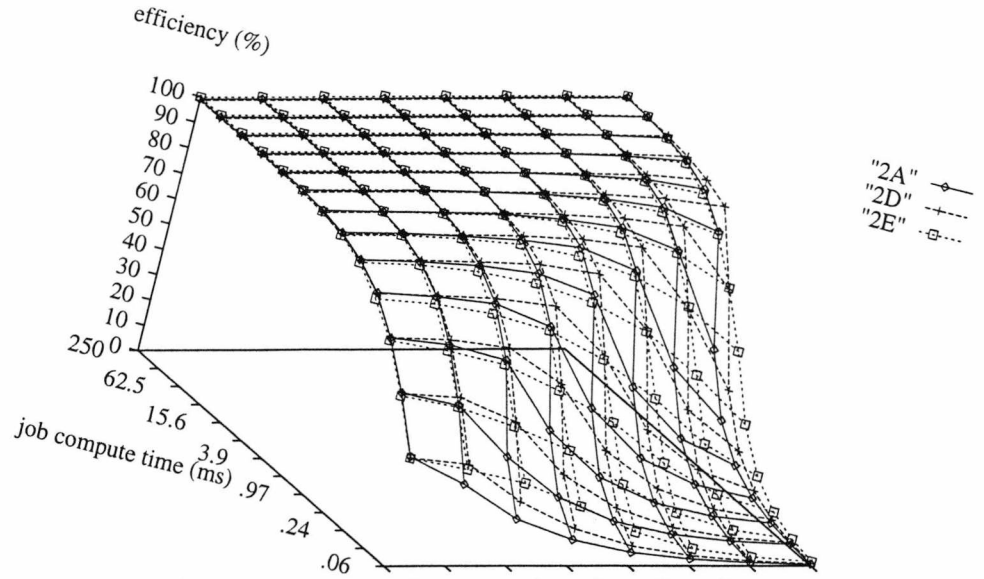


Figure 28: Breakdown of harnesses A, D and E on 2 workers

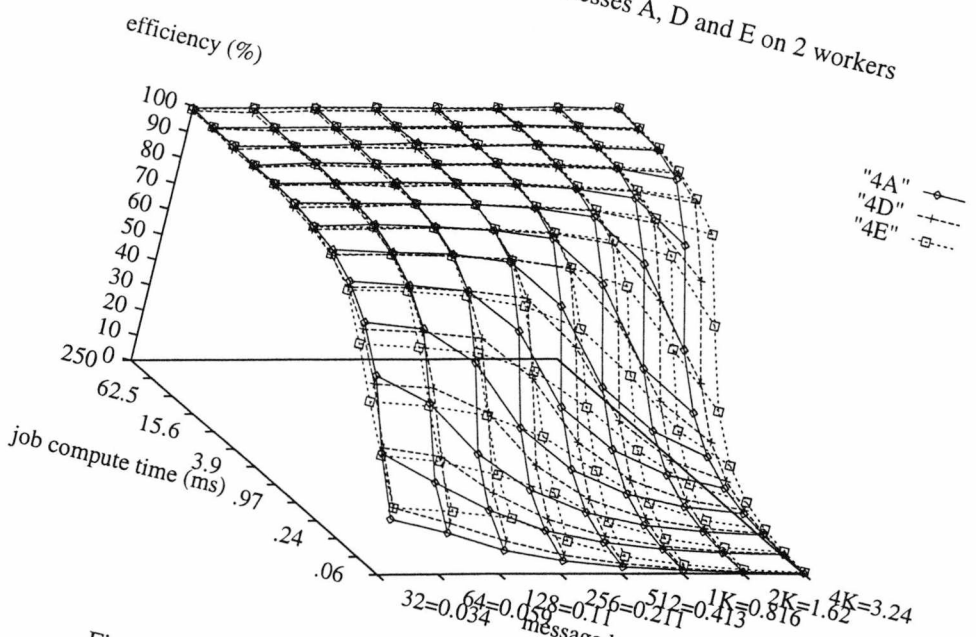


Figure 29: Breakdown of harnesses A, D and E on 4 workers

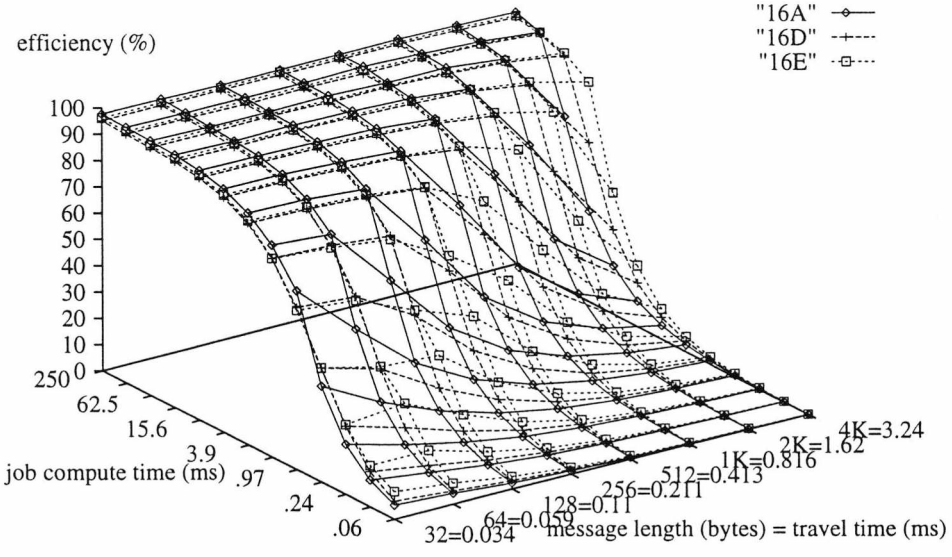


Figure 30: Breakdown of harnesses A, D and E on 16 workers

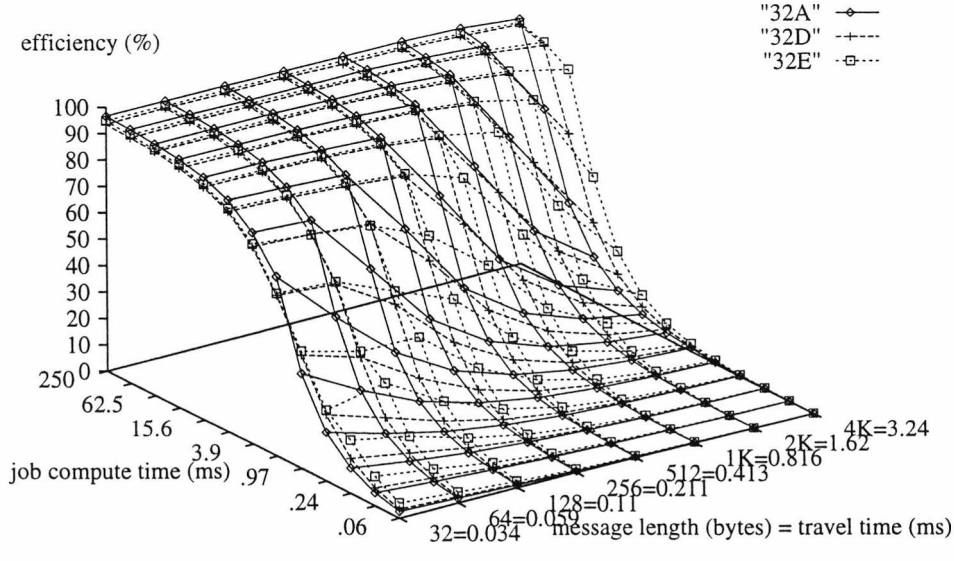


Figure 31: Breakdown of harnesses A, D and E on 32 workers

More of a gap opens out between harness E and the unidirectional harnesses. For 4 workers harnesses A and E are equally the most efficient then harness A takes over for larger sizes of farm. This is the case for a farm of up to 32 workers. Thus the performance of these harnesses is highly consistent for a wide range of farm sizes.

Even for a line of 32 workers harness A is still more efficient for compute bound applications than having a double harness that delivers jobs to both ends of the line. This would all seem to indicate that if a harness is good at providing throughput, it can farm out more demanding applications.

How well the harnesses speed up

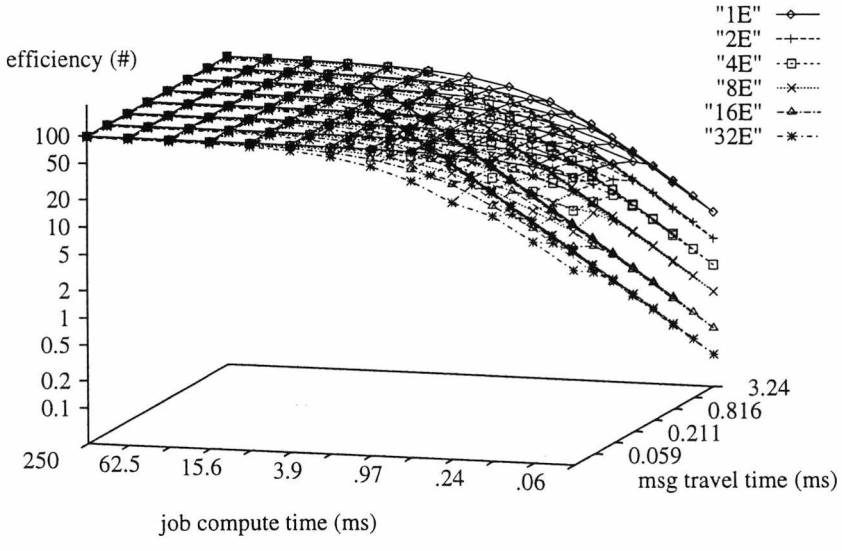


Figure 32: Breakdown of harness E on farms of 1, 2, 4, 8, 16 and 32 workers

So far the performance looked at has been of the harnesses relative to one another. Another aspect obvious from the graphs here is that from the point of view of the number of workers used, the more workers there are the smaller the number of mappings can be farmed out efficiently. This can also be looked at from the point of view of the application mapping. In figure 32 we are looking along the plane of the slopes. As can just be seen here, the more workers in the farm, the earlier the implementation breaks down and the lower the slope on the right-hand side of the graph. Thus, the harnesses cope less well with each application mapping as the number of workers increases. This is caused by there being an increase in demand but only a constant amount of supply. This results in a decreasing in number of application mappings being farmed efficiently when the number of workers is large.

Plotting j versus w

Here we plot the parameter j, the job run time against w, the number of workers in the farm.

In figure 33 (top of next page), as with figure 32, it can be seen very clearly that the breakdown of the larger farms is due to the supply being proportionally less per worker.

In subsection 4.9.3 (page 81) it was noticed the way the efficiency broke down was the same whether m had been increased or j had been decreased. The curves in figure 33 also break down in this same fashion. Thus increasing w, the number of workers, also results in exactly the same style of breakdown behaviour as if either of the two application parameters had been altered. Again this is due to demand being increased but the supply of work being constant.

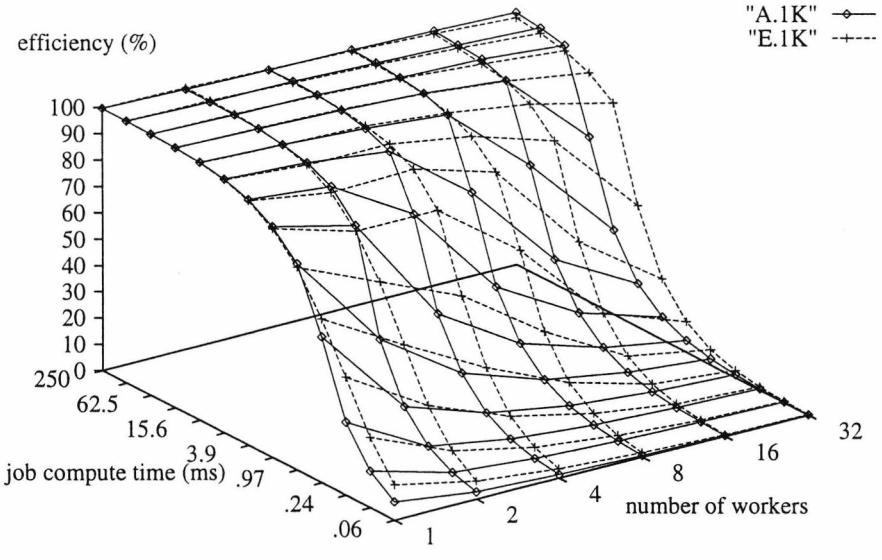


Figure 33: Breakdown of harnesses A and E for 1K jobs for increasing w

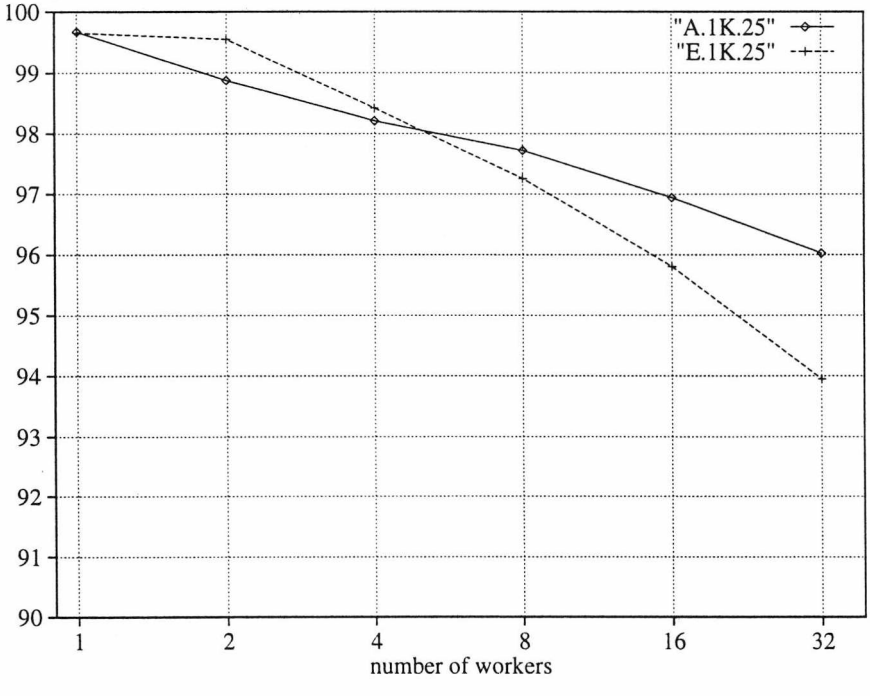


Figure 34: Breakdown of harnesses A and E for 1K 0.25 second jobs for increase w

From looking at the back edge of this surface, see figure 34 (bottom of previous page), we can see harness A is the most efficient for the compute bound farms with a large number of workers, despite all of the extra throughput engineering in harness E.

Execution imbalance

Here we look at how balanced the execution of the compute bound farms are. There is done by having a notion of maximum efficiency. This being a theoretical notion of efficiency, not a practical one. A farmed application consists of a number of items of work. The minimum possible time it would take to perform this work across a number of processors is, the time it would take one conventional processor to execute each item of work in turn divided by the number of processors used.

This does not take things like communication and distance into account, but as we are already aware transputers can perform communications in parallel at very low cost (there is the cost of executing the communication instruction and the links accessing memory during a communication slightly reduces the processor's bandwidth to memory). Thus the efficiency figures we present here are our observed values compared with a notion of maximum efficiency that is purely calculated, i.e. based purely on the maximum speed up we should be able to achieve if we could perform only work on all transputers all the time and no communications at all.

It should also be noted the values presented here are just for the efficiency of the workers. The execution overhead of the farmer and the harvester are not included. These processes are only being run on separate processors for the simplicity of running these experiments. In a real farm implementation it is best to run worker processes on all transputers including these. Thus, figures for efficiency that included the performance of the farmer and harvester processes would be slightly smaller than the figures presented here. However, this point is not of great concern. A generally impression of the efficiency of the workers due to harness overheads, via reasonably accurate figures was all that was needed here. Obtaining truly accurate figures is also not possible. Although the farmer and harvester can be run on the worker processors, there will always be the overhead of executing the statistics gathering code used here, and this would also affect the results obtained.

From looking at the figures on the graphs presented here it can be seen none of the farms obtain 100% efficiency. There are two reasons for this. The first is because of the overheads of running a harness. The second can be found by looking at the number of jobs performed by each worker. The distribution of the work is never completely balanced. This is due to the use of a Poisson distribute to vary the run times for the jobs. To illustrate this, here are the number of jobs performed by each of 16 workers. This was for an application that was just compute bound and harness A was used. The messages were 4096 bytes in length and they took 0.25 of a second to be processed, see figure 30 (page 86). The figures shown here are for the fastest, and thus the most efficient, of the five runs performed.

98 98 99 100 97 101 101 100 98 98 102 100 100 103 102 103

Whenever there is some variation in the amount of work performed by each worker, it is not possible to obtain an efficiency of 100%. Nevertheless, nearly 100% can be obtained, this indicates that implementing applications as farms using the highly efficient harnesses used here is highly effective, and any applications that can be implemented in this way should be.

How good is the performance model?

Here the model of efficiency developed early is compared with the results that have been obtained in this section.

If we look at a farm with 8 workers. This has a bandwidth to the end worker of 1.177 megabytes per second, presumably more to closer workers. If this farm is communicating 4 kilobyte jobs through harness D, see figure 23 (page 81),

$w = 8$
$m = 4096$ bytes
$B = 1.177$ Megabytes
$s = 12.9$ bytes

then the rate of supply is,

$$\frac{B}{m+s} = \frac{1.177\text{Mb/s}}{4096+12.9} = 300.4 \text{ jobs per second}$$

If s were to be ignored here we would get 308 jobs per second, instead of the 300.4 jobs per second as calculated above. This is about a 2% difference. This goes to show as was suggested on page 52, that for large m, s can be ignored.

Looking at some values of t found close to the breakdown of this mapping. For jobs that took 62.5 milliseconds to execute, the rate of demand is,

$$\frac{w}{t} = \frac{8}{0.0625} = 128 \text{ jobs per second}$$

This is clearly much less than the amount of supply available and indeed this mapping ran at 92.9% efficient.

For 31.25 millisecond jobs the rate of demand is,

$$\frac{w}{t} = \frac{8}{0.03125} = 256 \text{ jobs per second}$$

This is starting to approach what was measured to be the level of supply to the end worker, 287.473 messages per second. And indeed in these experiments this mapping only runs at 88.9% efficient here.

For $t = 15.62$ milliseconds the rate of demand is,

$$\frac{w}{t} = \frac{8}{0.01526} = 524.3 \text{ jobs per second}$$

This is clearly much greater than the rate of supply here and indeed this mapping ran at only 63.2% efficient. This is some way from maximum efficiency, although compared with an algorithmic mapping this still might be considered reasonably efficient.

What is interesting here is that this simple theoretical model predicts the performance and the maximum number of compute bound workers obtained in practice with quite a high degree of accuracy.

This model can also be used to calculate the largest number of workers an implementation can have and still be compute bound can be computed directly from the mapping and hardware parameters,

$$w_{max} = \frac{jB}{m+s}$$

It would appear the overheads of farming out work on a processor farm are highly minimal, on this architecture at least; the value of w_{max} is directly related to the value of the expression, and is not some fraction as it was thought it might be. Thus we really do get access to all of the transputer's potential.

In subsection 4.5.3 (page 52) it was considered if the throughput out of a farmer is,

$$\frac{B}{m+s}$$

and if k is the amount of throughput lost through the execution of the harness on one transputer, the throughput out of the first worker is,

$$\frac{B}{m+s} - k$$

and so on.

The value of B used in the calculations above is the bandwidth to the end worker, not the raw bandwidth out of the farmer, unfortunately this has not been measured. However, if we had this value, the value of k could be calculated. As,

$$\frac{B_{\text{farmer}}}{m+s} = \frac{B_{\text{endworker}}}{m+s} - wk$$

the value of k is simply,

$$k = \frac{\frac{B_{\text{farmer}}}{m+s} - \frac{B_{\text{endworker}}}{m+s}}{w}$$

4.9.4 Theory

So far we have a coherent idea that job compute times and message length are inversely proportional to one another. As has been discussed it is better to use a common metric, namely time, to view both computation and communication. Here this has been of use in realising the relationship between the parameters.

It was thought at the start of this research that the time to communicate a job must be much less in a compute bound farm than the time it takes to compute a job. There being many workers to communicate jobs to. This was not considered any further until the results had been looked at. Doing so made the breakthrough needed by considering what would happen when there was only 1 worker.

When $w = 1$, the time to communicate a message can be equal to the time it takes to process a job, so the next job has finished arriving on the worker as soon as the current job has finished being processed. This we can express as,

$$\frac{m+s}{B} \leq j$$

For 2 workers, $w = 2$ the communication time must be at most half the compute time,

$$\frac{m+s}{B} \leq \frac{1}{2}j$$

as 2 jobs must be performed in the time it takes to perform one job. Therefore generally,

$$\frac{m+s}{B} \leq \frac{j}{w}$$

By looking at values of j, w, m, s and B, one could find out how close to the optimum performance of the transputer these harnesses are, this should be high if in practice we are obtaining 90–99% of the theoretical maximum efficiency.

The main conclusion to draw here is that ultimately the maximum performance is equal to the maximum amount of supply one can generate. Therefore, when mapping an application onto a farm, once j, m, s and B have been found, the maximum value of w is dictated by,

$$\frac{jB}{m+s}$$

or just,

$$\frac{jB}{m}$$

If we wish our applications to be farmed out onto as large a farm as possible, then, from what has been discovered here, j should be as large as is sensibly possible (see next section) and m should be made as small as possible, with as few components as possible, as was discussed in subsection 4.6.5 (page 58).

4.9.5 Conclusions

Here we go over the four major conclusions arrived at in this section.

Most efficient harness

This experiment set out to find the harness that was the most efficient for as many applications as possible.

In general it transpires that no one harness is the most efficient. The harnesses that are the most efficient are so in one of two situations, never both. The first situation is when application mappings are reasonably compute bound. Here a harness is needed that has a very small execution overhead. Harness A is like that, but it is not the last to break down. The second situation is for the mappings that are the

most demanding. Here the implementation is compute bound and approaching the threshold where a farm becomes communication bound. The harnesses that are efficient here, harnesses D and E are generally slightly less efficient than the previous type.

When an application mapping needs a supply that is less than half of the maximum performance obtainable with the first generation of links, harness A is the most efficient harness for any number of workers (apart from two). What harness to use for mappings that are the most demanding is less simple. If one can construct a bidirectional harness and turn usage checking off, harness E is the most efficient, if usage checking must be kept on, harness F should be used instead. If only a unidirectional harness can be used, then harness D should be used if usage checking can be turned off, harness C should be used if it can't. This is summarised in the table 2.

number of workers	most efficient for compute bound	most efficient at breakdown
1	all	D, C
2	E	D, C
4	A and E	E, F, D, C
8	A	E, F, D, C
16	A	E, F, D, C
32	A	E, F, D, C

Table 2: Most efficient harnesses for farm sizes tested

Thus we can conclude so far that if an application mapping is highly compute bound then a simple harness is the most efficient and that only demanding implementations require sophisticated harness designs.

The farming execution strategy is independent of any application it may execute. That said, this does not imply that one harness that is the most efficient in one situation will necessarily be the most efficient for all applications.

Supply and demand

Farming comes down to supply and demand. The supply is how many jobs one can supply to or results one can retrieve from the farm, and thus how much communication bandwidth is available. So for example the larger the messages get, the fewer can be supplied in any given period of time.

It is always possible to create more demand, just simply by adding more workers. However, it is more difficult to provide a higher rate of supply, this can only be done by developing a better harness or a better farmer. For example, by reducing the use of `ALT`s.

Prediction theory

The maximum number of workers that will produce a compute bound application can be estimated for any application mapping. This is performed with the equation,

$$w = \frac{jB}{m+s}$$

Perhaps this should not be too surprising as this equation is just,

$$\frac{B}{m+s} = \frac{w}{j}$$

rearranged. If s is small, it can be ignored.

All that is needed is to measure the values of j and m from the mapping we intend to use, the value of B, the bandwidth of the implementation and s the cost of setting up a communication. These are reasonably easy to measure and the values of the last two figures can be obtained from the results presented here.

The normal procedure would be to write the code and then to run it. Here it is advocated that once the code is written it is tested to see if the mapping of the implementation results in a compute bound implementation. If not a better mapping and implementation can then be developed.

This results in only a small, but highly effective modification to the implementation development discipline used by many already. It is also one that informs the implementor as to the effectiveness of the implementation, without having to work out the efficiency or the speed-up of the implementation. A measurement that would involve the developing of a single processor implementation.

Model accuracy

It appears that this model captures the performance of a processor farm very accurately. This is probably helped by the minimal performance overheads of implementing a processor farm on the first generation of transputers. It also shows we are getting full access to the performance a transputer is capable of.

4.10 Influence of job compute time on finishing

So far this chapter has studied the general running of farms. This section looks at how well a farm finishes.

4.10.1 Introduction

Due to the nature of some applications, there is an extra decision that can be made when the design is being mapped onto the farming architecture. This extra decision involves how much work each individual job message is to contain, and thus how long an individual job will take to perform. In order for this decision to be well made the following farm mechanics must be considered. When a farm is finishing, the last few jobs will be in the buffers furthest from the farmer. At this point in the running of the program there will be an increasing number of idle workers. These are near the farmer and are unable to perform these buffered jobs as work can only flow away from the farmer. For jobs that take a long time to execute this slowing down will be noticeable by the user. With jobs that are quick to execute this is less noticeable. On the other hand, having lots of jobs will involve a larger number of communications overall, and thus again the farm could run potentially slower that it might. Somewhere between these two extremes there is balance to be struck.

The ideal situation is when the most time consuming jobs are the first to be performed (this also helps in the initialisation of the farm as we shall see in the next section) and the jobs that take the shortest time to execute are the last to be farmed out. Generally however, the jobs farmed out are in no particular order of compute length, as they haven't been here. Thus, normally how cleanly a farm finishes will depend on the average length of the jobs being farmed out.

A smooth finish is desired, with most workers completing their last job about the same time.

This experiment sets out to discover how much the job compute time influences an implementation's performance in practical situations. This is achieved through looking at one example application.

Here we are interested in how much of a difference this issue can make to the overall run time. This issue is likely to become of increasing interest as it is now being realised some applications can be implemented on a farm by farming out jobs not as one continual stream of jobs, but as a series of job sequences, where the beginning of a burst may not available until all of the previous set of results have been received and processed. Thus being aware of how much of difference this issue can make and also being aware of how to find the right balance is an important issue.

Here it was decided to take an application of a static amount of work and farm it out using different lengths of job, both in terms of the length of the job packet and the length of time it took to process each job.

Thus, here it was decided to study one application to see how much of a difference in run time this parameter could be responsible for.

4.10.2 Test Design

Studied here is an application consisting of jobs of data that can be easily varied in size. Here the length of the messages used would be scaled, as well as the length of time it would take to process them. Thus the amount of data communicated overall is always the same as the amount of work performed. In implementations of this nature changing the amount of work in a job changes both the number of communications performed and the amount communicated.

It was decided to study an application whose total run time in microseconds was exactly divisible by a wide range of numbers. Further the work was of a very fine grain, thus allowing for a great deal of scalability.

An application was wanted that would run on a farm of 32 workers, thus for convenience the amount of time the application should take was to be a multiple of 32 seconds.

The Poisson distribution was still going to be used to provide a suitable amount of variation in job computation time. Thus all runs of the farm would need to consist of a complete number of cycles through the Poisson distribution of job times. This was achieved by having a minimum of 100 jobs for each worker. As there were 32 workers this gives a minimum of 3200 jobs in any mapping. Each of these jobs would take a second to process.

```
SEQ w = 0 FOR 32
  SEQ i = 0 FOR 100
    -- give out 1 second jobs, varied by the Poisson distribution
```

Following on from this, if jobs took half a second to process there would 200 per worker and so on.

```
SEQ j = 0 FOR ?
  SEQ w = 0 FOR 32
    SEQ i = 0 FOR 100 * (2^j)
      -- set length of job to 1 second / (2^j)
      -- give out jobs varied by Poisson distribution
```

As mentioned before, a 1 second job is simulated by 100,000 iterations of the workers' time consuming loop. Unfortunately 100,000 doesn't divide perfectly by powers of two. As these results were going to be compared directly, it was important the amount of work performed was exactly the same. Thus the number of iterations done were powers of two. The range of iteration values arrived at were from 1024 to 131072. This latter value being around 1.3 seconds. Thus each time the application was farmed out the work would be grouped into different sized jobs, starting at jobs taking around 0.01 of a second and doubling all the way up to 1.3 of a second.

The only parameter left to decide upon was message size. It was desired that all the different mappings were to be compute bound. From the results obtained earlier we know if a job takes one second to process, a 32 worker farm could easily cope with 4 kilobyte messages, see figure 31 (page 86). Thus I set the test generator to scale this figure down linearly for jobs that took less time to perform, jobs taking 0.655 of a second to process having 2 kilobytes of data in the messages and so on. Thus the final test consisted of jobs ranged from 32 bytes taking 0.01 of a second to process up to 4 kilobytes taking 1.3 of a second to process. This test was run for a farm of 32 workers for all six harnesses.

4.10.3 Test Program

The essence of the test program was copied from the previous test program `farm1`. The worker and the harvester processes were the same. There was a similar farmer and a similar driving program.

In this test the driving program just generated a sequence of job processing times. The number of jobs to be used was then deduced from the processing time. Again each time was generated five times and again these five run times were averaged.

```
INT delay, time, run, total.run, len, njobs:
SEQ t = 10 FOR 8 -- 8 experiments
  SEQ
    delay := 1 << t    -- job len in 10s of u-seconds
    time := 10 * delay    -- job len in u-seconds
    -- the larger the size of the jobs - t
    -- the fewer jobs there should be
    njobs := workers * (njob << (17 - t))
    len := delay >> 5    -- calc message length
    total.runs := 0
    SEQ i = 0 FOR 5 -- reruns
      SEQ
        to.farmer ! njobs; delay
        ...  receive and output results from farmer
    ...  calculate and output averages
```

The farmer received a number of jobs and a processing time. From the processing time the farmer deduced the message length. For every 1024 iterations of the ten microsecond loop of the worker, the work packet had 32 bytes. This gives us the following job times and message sizes.

variable t	run time (seconds)	message length (bytes)
10	0.0102	32
11	0.0205	64
12	0.41	128
13	0.0819	256
14	0.164	512
15	0.328	1K
16	0.655	2K
17	1.31	4K

Table 3: Job processing times and their message lengths

The application was farmed out and the performances analysed.

4.10.4 Results and Conclusions

Figure 35 (top of next page) shows the run times of the six harnesses for each job compute time.

According to this graph the optimal length of time to be computing a job is at the curve's minimum, this is just above 0.164 of a second. This is with messages of 512 bytes.

The most important conclusion to draw from these results is that it is worth finding the best length of the average job. In this situation being an order of magnitude out, which in naivity is possible, reduces the farm's efficiency from the maximum efficiency possible here 98.5%, which is near perfect, down to 93%, see figure 36 (bottom of next page). In terms of run time this is a difference between 133 and 141 seconds, a saving of 8 seconds. Which in context to the size of the whole application is an extra 6% longer execution time. Thus finding the optimum job length is clearly worth doing in order to decrease the run time of the application.

What is also of interest is that the optimum length of job appears to be independent of the harness used, all of the curves change direction at the same point.

Here harness A is the most efficient. This is not too surprising as we deliberately arranged for this application to be highly compute bound. What is also interesting is to note is that the order of which harness is the most efficient, changes for different job compute times. For very small job compute times the order is: A, C, E, D, B and F. For the largest job compute times used here the order is: A, E, D, C, F and B.

Figure 35: Run times for varying job lengths for all 6 harnesses on 32 workers

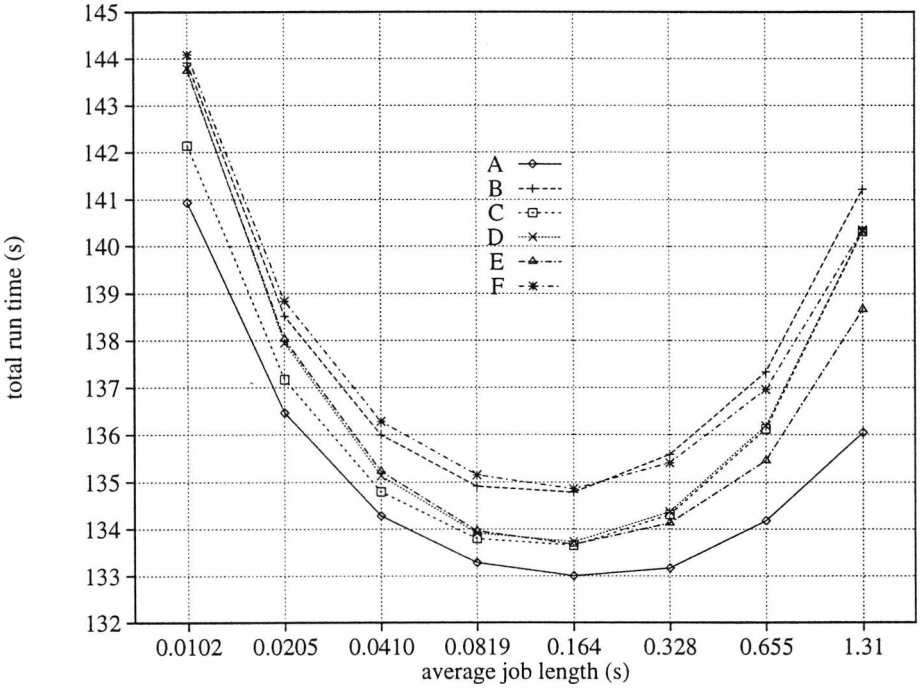
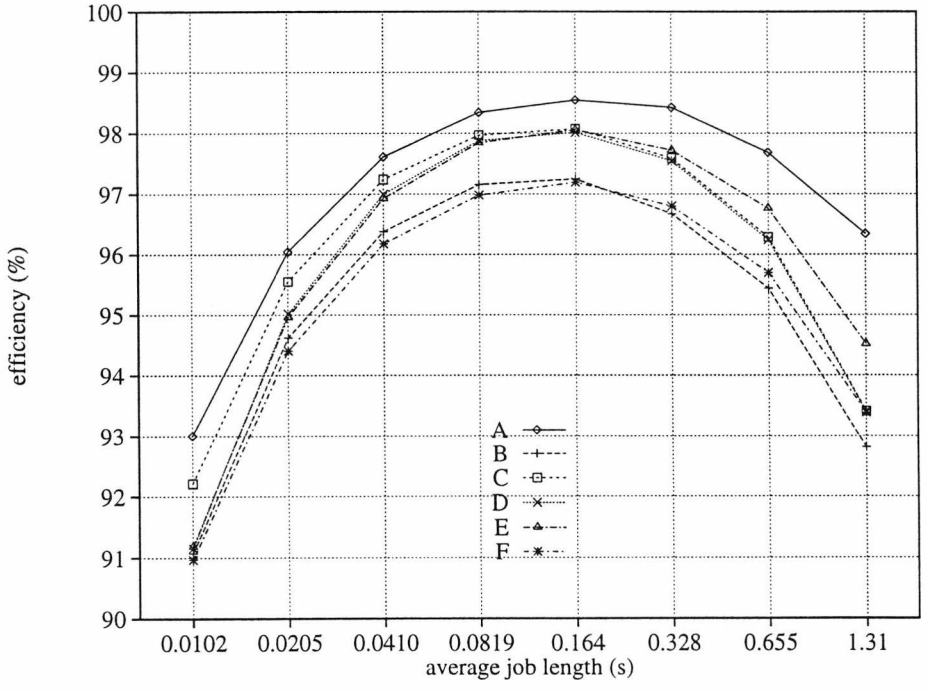


Figure 36: Efficiency for varying job lengths for all 6 harnesses on 32 workers



At the optimal job length, harness A is 0.65 of a second faster here than the next fastest harness, harness C. This is an extra $\frac{1}{2}\%$ more efficient. In this case, this is not very much and thus in this situation is not worth being concerned about. However, for farms that run for a long time period this degree of tuning may save minutes.

Here we have just looked at finding the balance for one particular application. This subject needs to be looked at in greater detail before a general theory can be developed.

This application only has a reasonably small amount of work to perform. For a larger amount of work the optimum size of job would be slightly larger. The smaller number of communications this would result in would absorb the extra time it would take for the farm to finish.

In this application the amount of work to be performed has deliberately been chosen to be small, this is so that this experiment would gain insight on implementations that farm out the series of job bursts discussed earlier. In such farms there would a considerable number of farm shutdowns throughout the running of the application. Thus, it has been shown here that if the sizes of jobs can be selected to any degree, it is even more important the optimum size of job be found.

4.11 Farm start up

Having looked at how much the average job length affects the efficiency and finishing time of a farm, there is also the question of how a farm starts up and becomes filled with jobs.

Initially it was thought it would be necessary to study this behaviour. However, after realising that farm start up just involved the harness becoming filled with spare jobs it was realised no experimentation would be necessary. Start up being something that just happens and in any good implementation it will happen quickly. We now look at why this is the case.

4.11.1 Opening discussion: what starting up a processor farm consists of

When a farm starts the harness and the workers are empty. The farmer outputs the first job into the farm, then the second and so on. In a compute bound implementation a point will be reached when all the workers are working on jobs and all the buffer spaces are filled with spare jobs. At this point we say that the processor farm is filled with work or initialised. If the farmer attempts to output a further job it is descheduled until the farm finishes a job and can receive more work. However, this is only likely to be the case for highly compute bound implementations. If an implementation was not highly compute bound the first job, or even the first few jobs, could finish before all of the farm was filled. Thus the farm would take a longer time to fill up with work and so would not be running at maximum efficiency for a short period of time, due to workers waiting for jobs to arrive down links. As was discussed in the previous section, it is desirable to have all of the long jobs farmed out at the beginning of a run.

Communication bound farms do not fill up with work. The same is also true of farms where the demand is equal to the supply. In such farms the farmer will be able to supply jobs to the workers, but these jobs are consumed at exactly the same rate. As a result the workers are likely to be idle after finishing an item of work as they wait for the next job to arrive down the link from the farmer. Also the buffers will never really be in a state where they are behaving as buffers, in possession of a job and descheduled until another process makes a claim for its job. The workers will be making a constant claim for work. Thus only compute bound farms actually use the buffering capacity in the harness.

In conclusion all that can be said so far is, if a farming harness does fill up with work, it will do so after some period of time.

4.11.2 Theory

An attempt to understand how a farm initialises via operational means was not successful. It could be seen that if an application was only just compute bound, the whole farm wouldn't be filled with work before the first job (or perhaps the first few jobs) finished. This approach led to problems when trying to crystalise this. The breakthrough came when an attempt was made to calculate the number of jobs that

would be output into the farm, in the time it took a worker to perform the first job. With this it was realised the speed of initialisation comes down to how much greater the rate of supply is than the rate of demand.

As a simple example lets say the harness we are using has w buffer slots, one for each worker. Then in this example we need an excess of $2w$ jobs to be supplied to the farm before it is filled. If the rate of supply is 6 jobs per second and the rate of demand is 4 jobs per second, that is an excess supply of 2 jobs per second, $6 - 4$. If $w = 8$ there are 16 slots to be filled, then initialisation will take the number of slots need to be filled divided by the excessive amount of supply. Here this is 16 slots, divided by a 2 extra jobs per second, giving 8 seconds. In general this is,

$$\frac{\text{buffer slots}}{\text{supply - demand met}}$$

where *demand met* is always less than or equal to the supply. If the demand is greater than the supply, the demand met will be equal to the supply. In this situation the implementation is communication bound and the buffers of the farm will never be filled.

From this model it is easy to see that the more compute bound a farm is, the quicker it will initialise.

This model can also be used to look at what was a problem to look at before: how many jobs are delivered to a farm before the first job is completed. As w is the basic number of jobs needed in the farm, and,

$$\frac{m+s}{B}$$

is how much time it takes to deliver a job. This multipled by w,

$$w\frac{m+s}{B}$$

is how long it should take to fill up the farm with jobs. If this figure is less than the time it takes to perform a job,

$$w\frac{m+s}{B} < j$$

all the workers will obtain a job before the first job finishes. Further, if h is the number of buffer slots in the harness, and,

$$(h+w)\frac{m+s}{B} < j$$

then all the workers and harness buffer slots will obtain a job before the first job finishes. Now, if the length of time to compute a job is less than this,

$$j < (h+w)\frac{m+s}{B}$$

but larger than the time take to fill up all the workers,

$$w\frac{m+s}{B} < j < (h+w)\frac{m+s}{B}$$

the farm will not be filled immediately, and thus will not be running at maximum efficiency for some time, as it will take a while for the empty buffer slots to be filled.

If part of the lack of supply is due to overheads within the farmer, several jobs could be prepared and farmed out in quick succession. Initially the farm sits empty for longer, but once the farm has been filled with work, the farmer only needs to top up the harness buffers with work, the workers always have work available on-chip and do not have to wait for work from the farmer. Thus, overall the farm should run slightly more efficiently. As this is not likely to be practical in most situations, it has not been looked at this here.

4.11.3 Priority

Which buffers are filled first depends upon, amongst other things, the design of the harness. This begs the question, should we then design a harness to fill up with work quickly? Here the answer is believed to be no. A harness should be designed to be as efficient as possible throughout the whole running of the farm. This being what a harness will be doing most of the time.

During initialisation the job distribution processes will simultaneously receive requests from all processes they are connected to. Changing which channel is given priority will change the way the farm comes filled. That said, it is unlikely that a farm will be able to fill up with work more rapidly.

4.12 A comparision of topologies

So far this chapter has looked at what approach to coding a harness for a line of workers is the most efficient. Looked at next are which topologies are suitable for farming and which of these are the most efficient.

4.12.1 What makes a topology appropriate for farming

The choice of topology is an important aspect of farm implementation. Both directly, as this choice affects the bandwidth around the farm, and indirectly, as different topologies will need different harnesses. Here it was decided to look in general at all the properties that would be appropriate for a farm and thus what topologies match these.

In subsection 3.2.2 (page 30) it was shown how a toroidal topology is not a very appropriate topology for farming. What is of interest here is why, so this can also be applied to find out which topologies are appropriate and inappropriate for farming.

Looked at here are what properties are desirable in a farm generally if it is to be efficient, and which of these affect the choice of topology and how.

Ideally all processors in the farm should be working continually. No processor should be starved of work. The properties considered important for this are listed here and discussed below.

1. Keep communications to a minimum.
2. Use all the communications bandwidth available.
3. Keep the harness's use of the C.P.U. to a minimum.
4. Any buffering within the harness should only aid performance, not hinder it in anyway.
5. Use a communication harness that is easy to write.

Keeping communications to a minimum can be achieved by not performing any unnecessary communications, for example by not passing messages back and forth continuously. Here we have used job distribution strategies that only give out work from the farmer to the workers in a client-server arrangement. Keeping communications to a minimum can also be achieved using a topology that possesses some degree of fanout.

Using the full communications bandwidth available is achievable by careful utilisation of the underlying hardware through the use of as many links as possible and link buffers.

Keeping the harness's use of the C.P.U. to a minimum is in fact reasonably easy. As we came to realise in subsection 3.1.3 (page 25), only work needs to be given out. Therefore the harness should just perform communications, inputs and outputs. Which direction work should be sent in should be decided by communication, via requests, rather than by some form of computation as these will require the C.P.U. Thus the code of the harness should consist of communications and as little of anything else as possible.

Any buffering within the harness should only aid performance and not hinder it in anyway is easily obtained through the use of any sensible buffering mechanism, as shown here in subsection 4.9 (page 68). As a general principle parallel system run at less than full speed if important parts of the mechanism are

prevented from proceeding if waiting for resources. It should be noted any variables in the harness that hold data in transit act as a level of buffering.

The mechanics of a topology that fans out needs to be discussed in detail. This is looked at next and leads naturally into a discussion on the last point of how to keep the harness easy to write.

Tree topologies

The performance of a compute bound implementation is limited by the rate at which work can be carried out. One method of enabling the workers to make the greatest progress with the application is to use a harness that uses as little of the C.P.U.'s resources as is possible. Another is to reduce the total number of communications performed. This can be done by reducing the number of jobs, as looked at in the previous section, and also by reducing the number of communications each message has to take to its destination. Naturally, the smallest number of hops possible is one, the farmer and workers being directly connected. In fact a logical model for a processor farm would also have this direct interconnection, see figure 37.

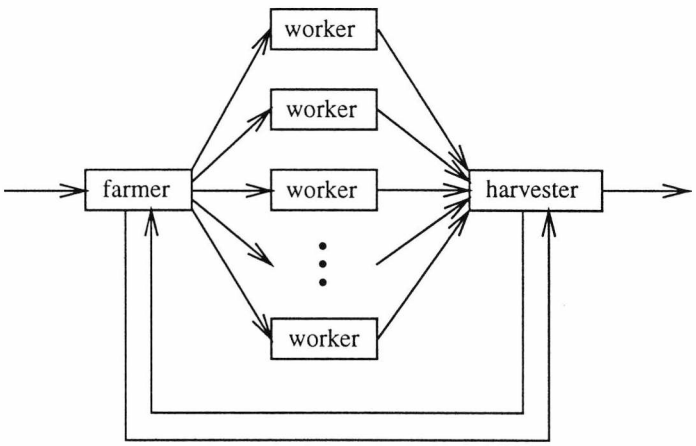


Figure 37: Logical structure of a processor farm

One method of getting close to this optimum is to fan out the work as much as is possible with the valency of the hardware. Doing this results in every job and result message passing through fewer processors than would be the case with a line of workers. Thus the total number of communications performed in the farm is reduced, as is the amount of time, on average, the harness is executed.

This additional fanout results in larger `ALT` constructs within the code of the harness. These take longer to set up and to shut down. They may also result in less work being performed by the workers close to the farmer, as these are executing expensive parts of the harness for a very large number of communications.

Looking at the reduction in communication in more detail. As a pipeline grows in length, the communication overheads grow in direct proportion, as can be seen in figure 34 (page 88). If there are n messages and w workers, each message is passed along an average of about

$$\frac{w}{2}$$

workers. So the total number of communications performed is proportional to

$$\frac{nw}{2}$$

If the number of workers is doubled, so does the overhead. With a tree the amount of communication is proportional to $n\lceil \log_f w \rceil$. Where f is the degree of fan out in the tree.

As a transputer possesses four links, the highest degree of fan out that can be obtained is three, giving the largest number of hops from the farmer to any worker is the base 3 log of the number of workers, greatly reducing the average number of links that need to be traversed between the farmer and the workers,

as can be seen in table 4 (top of next page). Also the amount of traffic at any stage is a third of what it

tree depth	number of workers	average distance to farmer from worker
1	3	1.0
2	12	1.75
3	39	2.615

Table 4: Average farmer-to-worker distances around a ternary tree

was at each previous stage.

This area has already been well looked at before [PZ90, TD90]. Both of these authors say ternary trees provide better performance than a single line of workers. The former also shows by how much.

Thus it is possible for an application that is not quite compute bound when run on a linear topology to be just compute bound when run on a more efficient topology such as a tree.

A ternary tree have one limitation. With a tree topology the farmer and the harvester must reside on the same C.P.U. This is unfortunate as the amount of work a farm can perform is related to the bandwidth available out of the farmer and more bandwidth can be obtained from a link that is only used unidirectionally rather than bidirectionally, and with a ternary tree results will be travelling along links in the opposite direction to jobs. It is therefore advantagous from a bandwidth point of view to consider having the farmer and harvester on separate processors. This could be possible by arranging two ternary trees together at the leaves forming a diamond, with the farmer and harvester at opposite ends.

For many sizes of farm a tree may not have every layer or level completely full of workers, as in figure 38 (top of next page). In order to obtain an evenly distributed communication load the best way to arrange for only the last level or layer of the tree to be partially empty, as in figure 39 (next page). There is a temptation to think that unbalanced trees must be placed as in figure 40 (bottom of next page). So, the communication load is distributed evenly over the first three workers. In fact it does not actually matter if the communication load is distributed evenly, the overall communication load on the whole farm will always be the same.

Unfortunately, trees that do not have all levels full are difficult to describe in occam. Slightly different versions of the harness may be needed on different workers depending on how many others workers each processor connects to. Thus, a different number of buffer processes may be required, especially in the case of harnesses like B or D.

Pipes and rings are very easy to scale linearly. However, a large linear topology will suffer from more communications problems than an equivalently sized tree.

In summary, on ternary trees, the average distance between farmer and worker is less, the average amount of traffic on a link is also less. The second of these two points has a further advantage as link communications consume memory bandwidth which slows the C.P.U. down and thus allows more processor resources for executing the application.

Keeping the harness to a minimum

With some topologies keeping the harness to a minimum is the most difficult property to achieve. Any topology is going to involve some processes being closer to the farmer or harvester than others. In order to remain true to this property, topologies should have the same harness code running on all of the workers in that topology, regardless of their position within it. Thus, what is ideally needed is a communications structure where one set of simple processes can be used through-out the network. For this to be achieved the same method of distribution should be performed at all places in the network, regardless of the position relative to the farmer. Thus, the whole topology should fan out from all of the nodes of the structure. All nodes should fan out by the same amount at all points. Also, no two communication paths should reconnect, thus each worker has only one communication path from the farmer and to the harvester. Pipes, rings, trees and combinations thereof possess this property. However, this property does not apply to all even structures. For example arrays, tori, hypercubes etc. have a very uneven structure when viewed from the farmer, because the separate branches reconnect. Although it should be possible to develop a method

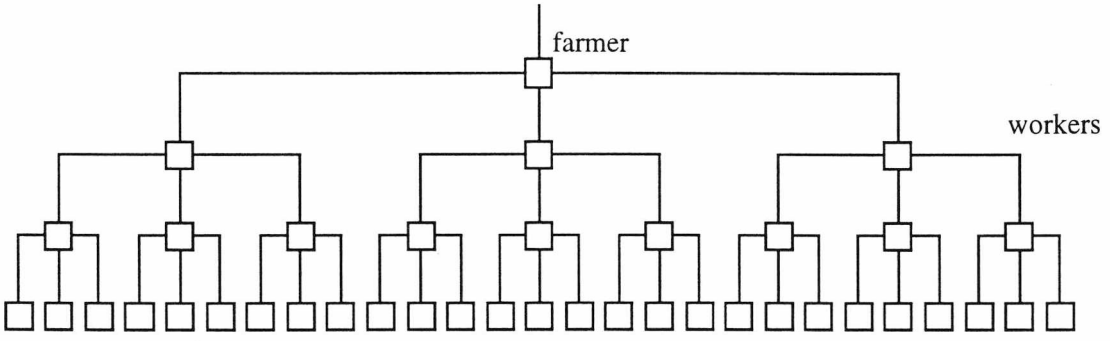


Figure 38: Fully populated three layer ternary tree

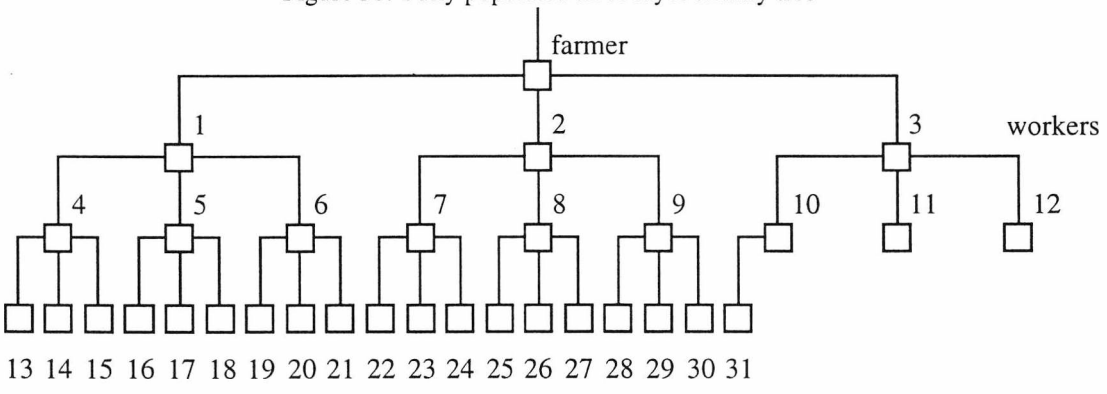


Figure 39: Partially populated ternary tree

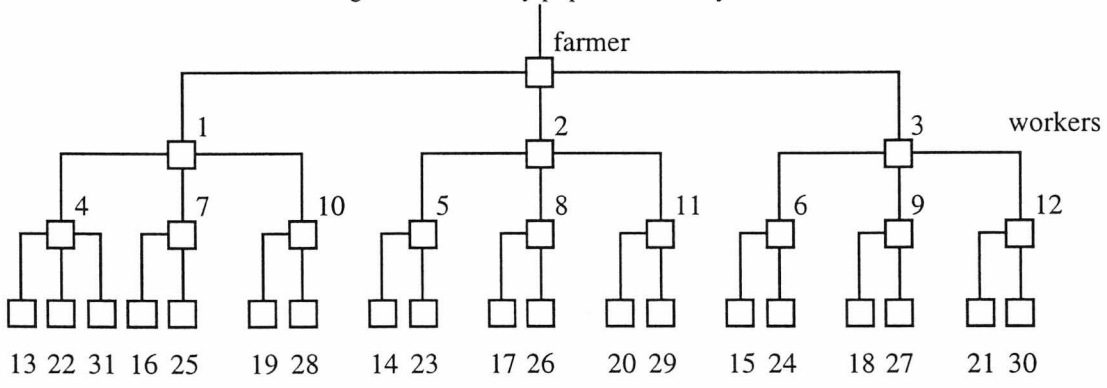


Figure 40: More balanced partially populated tree

of distribution that does supply jobs to all parts of the farm evenly when needed, as far as we are aware, this can not be done simply with one or two simple processes.

It is important to keep in mind that the topology should be self-similar from the point of view of the controller process or processes, the farmer and any harvester. If a topology doesn't adhere to this property any one single piece of code harness will not perform effectively or efficiently.

As hinted at above there are some variations on the basic suitable topologies. If a great deal of data is needed to pass in and out of the farm during run time then two pipes, rings or ternary trees could be used leaving two links to be used to link the farm to the outside world. A single binary tree could also achieve the same effect. Similarly, three pipes or rings could be hung off the farmer.

4.12.2 What was tested

As it was hoped ternary trees would be the most efficient topology possible, here the performance of ternary trees is compared against the line of workers used so far.

Why fully populated trees

As mentioned above this author is not aware of any method in occam of constructing mechanically fully balanced ternary trees of arbitrary size. Thus, for the purposes of these experiments it was decided to limit the testing to trees with a fully populated last layer. This is not a problem as a line of workers can be configured to be any length. Because of this restriction in testing the results here will not be totally general. Nevertheless, this was considered to be only a minor drawback, the results obtainable for unbalanced farms should be very close to figures extrapolated from the results here. The only problem being how to allocate the different versions of the harness required in different places in the topology.

Here trees consisting of 1, 2 and 3 layers are looked at. This gives farm sizes of 3, 12 and 39 workers. Each size of tree is compared against the same number of workers configured as a line. With three comparisions to perform there were three experiments.

For simplicity no worker was placed on the farmer and harvester processor.

Testing other topologies

Welch was of the opinion that trees have too much of an overhead to be more efficient than a line of workers. This was believed to be due to the number of communications the ALTs in the harness would have, each communication requiring to be set up and shut down every time the construct is executed. Further, from some experiments performed earlier (during a first attempt at this research) it appeared a three layer tree did not perform more efficiently than an equivalently sized line of workers. Although one and two layer trees were more efficient than the equivalent line of workers.

From this it was decided to also compare some topologies that were conglomerations of lines and trees. For example, as from the previous experiments, small trees could be more efficient than a line of workers and yet a three layer tree wasn't, perhaps it would be more efficient to arrange for such a large number of workers as a small tree that opens out into a number of pipelines.

We now talk through the three experiments and which topologies were compared in each.

Experiment 1: 1 layer tree, 3 workers

The first experiment consisted of comparing four topologies. There is only a small number of sensible topologies into which three workers can be arranged. The first three topologies are the three that are compared in all three experiments.

The first topology was the one layer tree itself. Here all three workers are connected to the farmer directly. The ternary tree was driven by a farmer that gave out jobs and received results on three links.

The second topology was a line of three workers connected together with the bidirectional harness E. The third topology was also a line of three workers, these were connected together with the most efficient harness we have so far for compute bound application mappings, harness A.

As these topologies consisted of a single line of workers with the links driven bidirectionally, it was decided to see what the performance would be if there were two independent and shorter lines of workers.

The placement code for this topology was written so that the two lines could be of different lengths. Thus, this topology was tested in all three experiments.

There was no point in building three lines of workers for three workers as this is just a one layer tree.

Experiment 2: 12 workers, 2 layer tree

The first four topologies tested here were the same as in the previous experiment: a two layer ternary tree, two lines of 6 workers, a line of 12 workers connected with harness E and a line of 12 workers connected with harness A.

A fifth topology was also built. This was a conglomerate of the tree and line topologies. This topology consisted of 3 lines of workers, four workers for each line. This was easily achieved by using the farmer used to drive a ternary tree. This topology might perform slightly more efficiently because of the reduced harness overheads for the workers within the lines of the topology.

Experiment 3: 39 workers, 3 layer tree

This experiment consisted of the four topologies tested in all of the previous experiments: a three layer ternary tree, two lines of workers, a line of 39 workers connected with harness E and a line of 39 workers connected with harness A.

Here it was also decided to look at reducing the farming overhead at various stages either by having 3 lines of 13 workers, still long but very much shorter than a line of 39 workers, or by having 3 workers fan out to 9 lines of 4 workers. This latter topology was achieved by having 3 workers that used a tree harness to feed 9 lines of workers.

4.12.3 Test program modifications

These experiments were performed on the same program as was used for the efficiency breakdown comparisions, `farm1`. Before the tests could be run additions and some changes were made to the program in its original form.

The workers of each topology required a slightly different variation of the harness code from any other. As we have said when introducing the harnesses, see subsection 4.3.6 (page 48), making a good harness more flexible is not of interest here as generalisations may result in a less efficient performance.

A different type of farmer was required for line, double line and tree topologies. The farmer for tree topologies was also used for the three line and nine line conglomeration topologies. A different farmer was also required for each harness used.

Most of the new topologies tested had no separate harvester processor, the harvester being on the same processor as the farmer. Only the topology consisting of two lines required a new harvester.

The farmer, harvester and the worker compilation modules contained many different versions for different types of farms. At any one time only one was compiled, the rest were commented out using comment folds.

With the bidirectional line of workers topology the worker furthest away from the harvester was also only one link away in the other direction. With the newer topologies here this was not the case and indeed in the case of trees there was no single worker further away from the harvester than any other. Thus none of the new topologies could practically perform any end worker throughput measurements. This was also one of the reasons why different versions of farmers and workers were created for each topology.

All of the new topologies required placement code to be added to the program, especially as most of the new topologies had no separate harvester processor.

There are three other detailed points to make about how the topologies were placed.

On the three and nine line topologies, in order to use as many of the links as possible, the body of the lines were connected up using two links: one for the jobs and one for the results. Doing this made the first worker in the line a slightly special case as it was connected to the rest of the line via two links and to the rest of the tree via just one link used bidirectionally. Thus this first worker needed to be `PLACE`d separately, see figure 41 (top of next page). Note this worker did not need to be a separately compiled module, but it did need its channels placed onto links in a different manner to that of the workers in the pipeline.

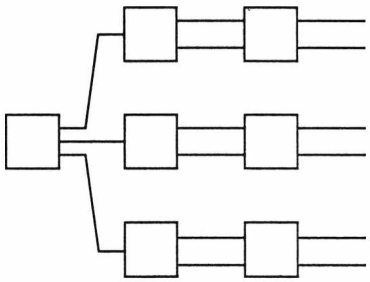


Figure 41: Detail of link usage for conglomerate topology

It was decided to place two, three and nine line topologies so that all of the lines were *grown* together in parallel, see figure 42 (next page). Thus the first worker of each line were all placed directly after

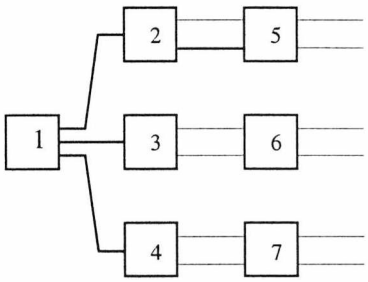


Figure 42: Order in which processors were placed

the farmer or tree element to which they were directly connected. This was done so that hopefully these processors would then all be equally close to the farmer and thus have a high bandwidth.

The nine lines topology needed an extra separately compiled module. As well as having workers that needed a line style harness this topology also had workers that needed a tree style harness.

4.12.4 Results

The above farms were run. Here we look at the results obtained. There are really two questions to ask here.

1. Do ternary trees breakdown later than a line of workers being run by bidirectional harness E? And thus, are they more efficient?
2. Are ternary trees more efficient than a line of workers being run by unidirectional harness A?

These questions are answered below for each of the three layers.

One layer tree versus equivalent topologies

Looking at these questions in order, we can see from figure 43 (top of next page) that for the three worker farms, the ternary tree is much more efficient than a line of doubled up workers (harness E). The tree also changes slope direction more slowly and at smaller message sizes, indicating lower communication start up costs. For larger message sizes (1K and above here) a doubled up line rolls off slightly slower and is around only 85% efficient for a small range of the most demanding mappings. The two lines of workers is the least efficient of the three topologies here.

Looking at the question of efficiency for when a farm is compute bound, see figure 44 (bottom of next page), we can see a ternary tree is also much more efficient than a single line of workers (harness A) on a line of workers, again, especially for more compute bound mappings when the messages are small. Thus, fanning out is worth it. The difference is only slight, however, we had found this before with small farms in the earlier experiments.

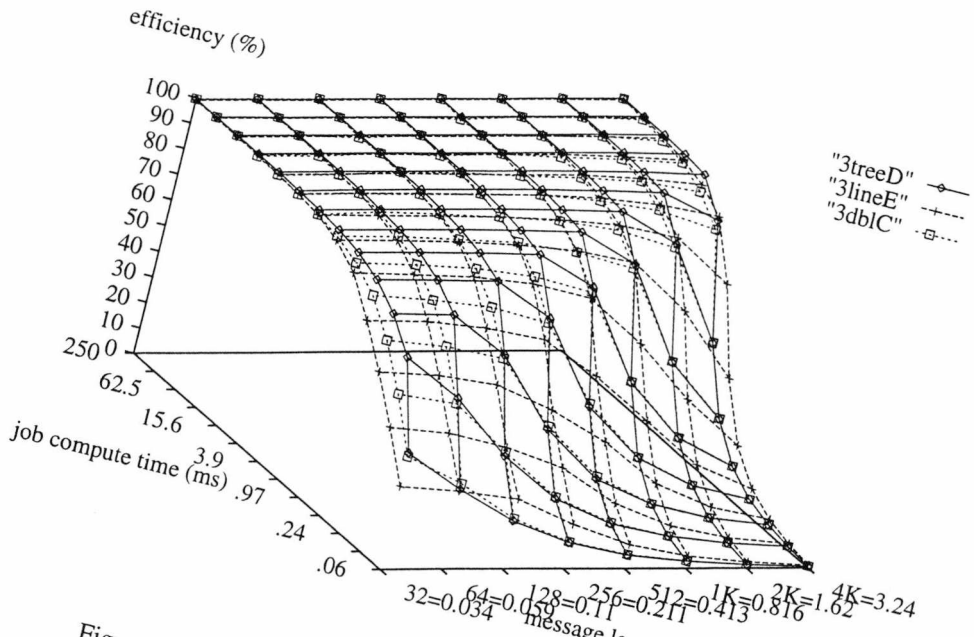


Figure 43: Breakdown for 1 and 2 lines and a tree of 3 workers

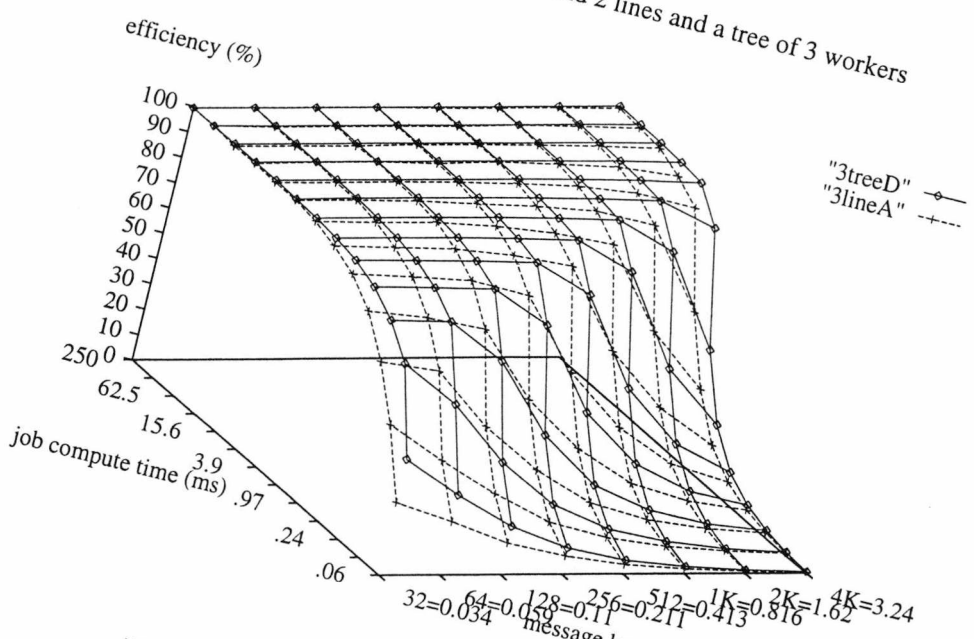


Figure 44: Efficiency for 1 line and a tree of 3 workers

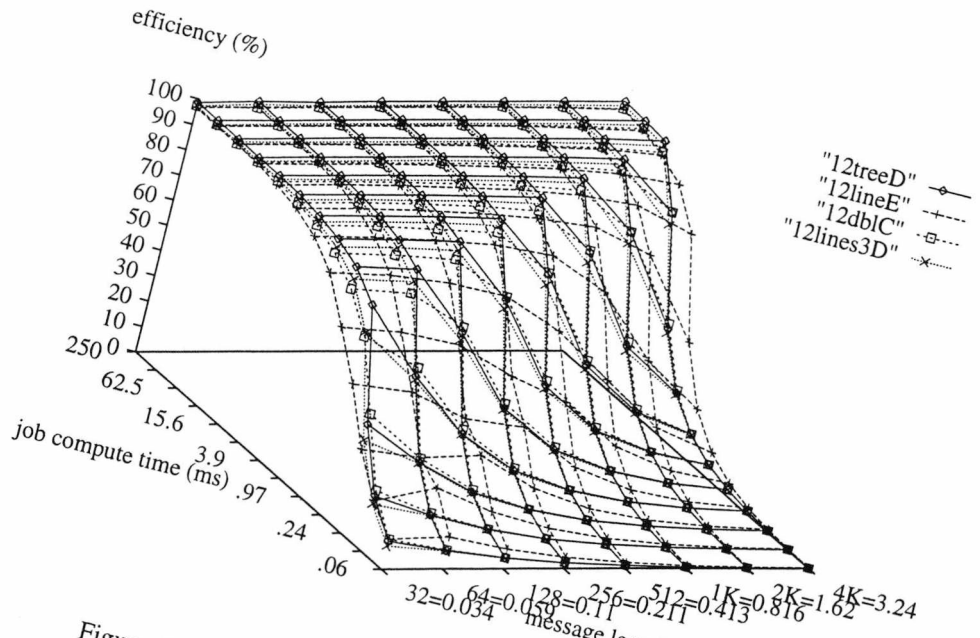


Figure 45: Breakdown for 1, 2, 3 lines and a tree of 12 workers

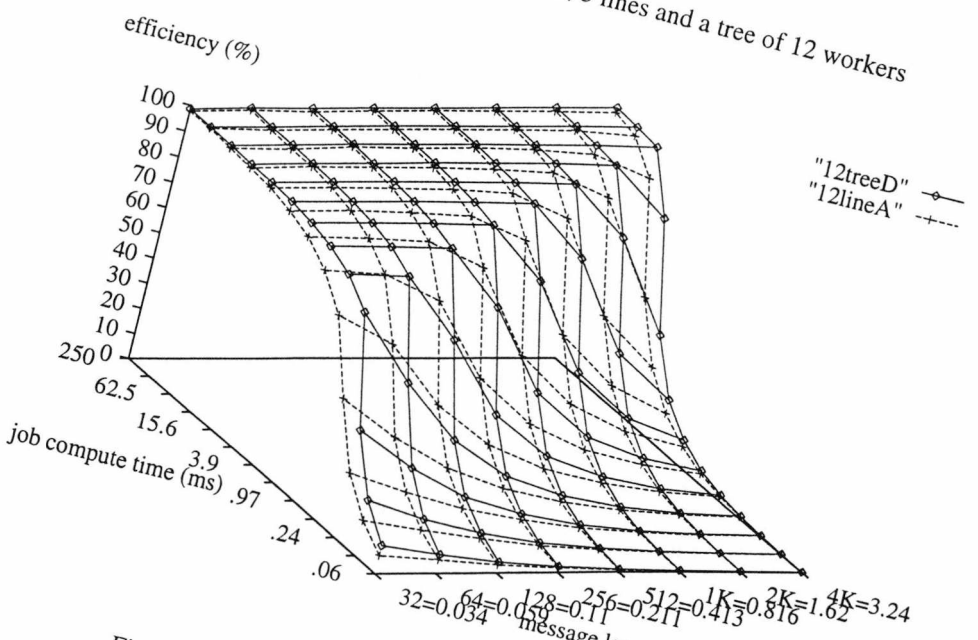


Figure 46: Efficiency for 1 line and a tree of 12 workers

Two layer tree and equivalent topologies

The results for 12 workers are more revealing due to the farm being larger, see figure 45 (top of previous page). As before a tree is the most efficient for compute bound mappings, for smallish messages (only 128 bytes here). Again a line of doubled up workers is the least efficient, however the roll off is much later for this larger farm and so is almost at its most efficient for highly demanding applications with large messages. The two and three line topologies to not exhibit anything of interest.

For all compute bound application mappings here, see figure 46 (bottom of previous page), a ternary tree is more efficient than a single line of workers. This is also much more clearly visible, this presumably because we have a two level farm and thus we are actually obtaining much more of the benefits of setting up a tree topology.

For all the application mappings here three lines of workers is less efficient than the full ternary tree, but is more efficient than two lines of workers; which for compute bound mappings is only as efficient as a single line of workers.

Another point to notice is that less mappings are compute bound here. This is to be expected. There are four times as many workers than a one layer farm and thus there is four times the amount of demand for work. The maximum amount of supply, however, is still the same as for the smaller farms.

Three layer tree and equivalent topologies

Figure 47 (top of next page) contains the three basic topologies that were compared against each other when three workers were looked at: a ternary tree, a single line of workers and two lines of workers. As the number of workers is even larger again here, there is even more of a spread in performance resulting in the differences between the different topologies being even clearer to see. As before the full ternary tree is the most efficient topology for highly compute bound mappings and also for highly demanding mappings with smallish messages (128 bytes). Again, a line of doubled up workers is still very efficient when a farm is on the verge of breaking down for non-small messages. Otherwise it is the least efficient topology. It is even just slightly less efficient than two shorter lines of workers. The reason for this is discussed next.

In figure 48 (bottom of next page) we compare the full ternary tree against the topologies that consist of many lines of workers radiating from the leafs of a small tree. This was done in the previous experiment with 3 lines of workers, but here there is also much more of a conglomeration of topologies as the 9 lines of workers required more code. In this comparision of three topologies we find the full tree is the most efficient topology of the three here for compute bound mappings. With nine lines of workers being very close, much closer than three lines of workers is. This reason for this topology being so efficient is because it is conglomerate topology that is most like a tree.

As can be seen from figure 49 (top of page 110) a ternary tree is much more efficient than a single line of workers. There being much more of a difference in efficiency. Also the nine lines of workers topology is also more efficient than the most efficient way of arranging workers into a single line. Again this is because nine lines of workers is so similar to a tree in structure. The single line of workers is more efficient than three lines of workers. For clarity the three lines of workers is only shown on figure 48 here.

Discussion

One thing that is clear from these graphs is that not only does the single line topology breakdown last, but that all the other harnesses, the two lines, the three lines and the ternary tree, all break down at more or less exactly the same time; the performance of these harnesses being practically identical after they have broken down. The reason for this must be to do with one aspect that is the same for all the topologies except the line of workers. Knowing that `ALT`s are expensive to execute I decided to look there and found the answer in the farmers used.

In the line of workers driven from both ends, to deliberately avoid the use of `ALT`s I had decided to give the work out evenly to both ends. As all of the work was given to the same workers, this would not cause a load balance problem. Nevertheless, with the farmers of the other topologies, I had used a number of sacrificial buffers, one on each link, these requested jobs from the farmer. Thus, the farmer executed an `ALT` for every job given out. Thus, it was this `ALT` that was the major bottleneck in the supply of jobs

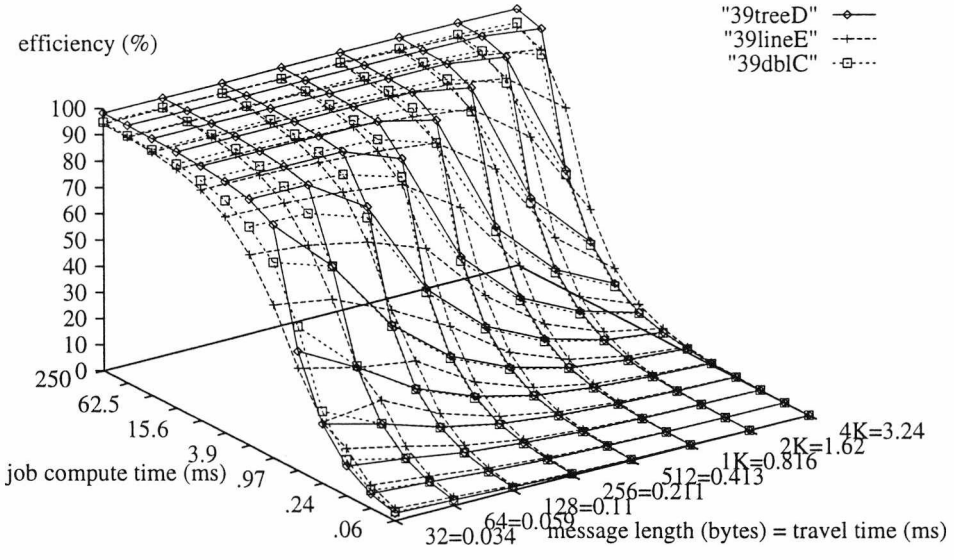


Figure 47: Breakdown for 1 and 2 lines and a tree of 39 workers

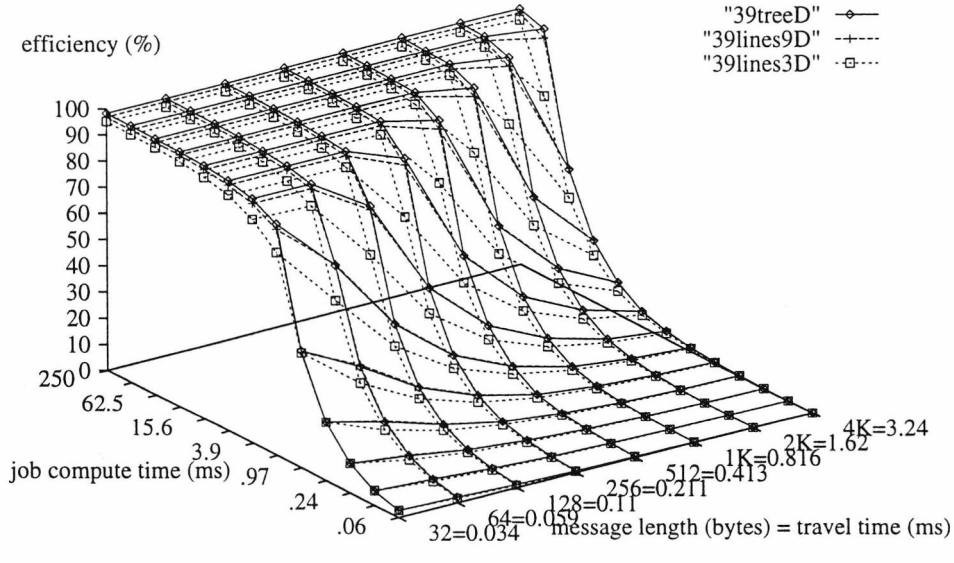


Figure 48: Breakdown for 3 and 9 lines and a tree of 39 workers

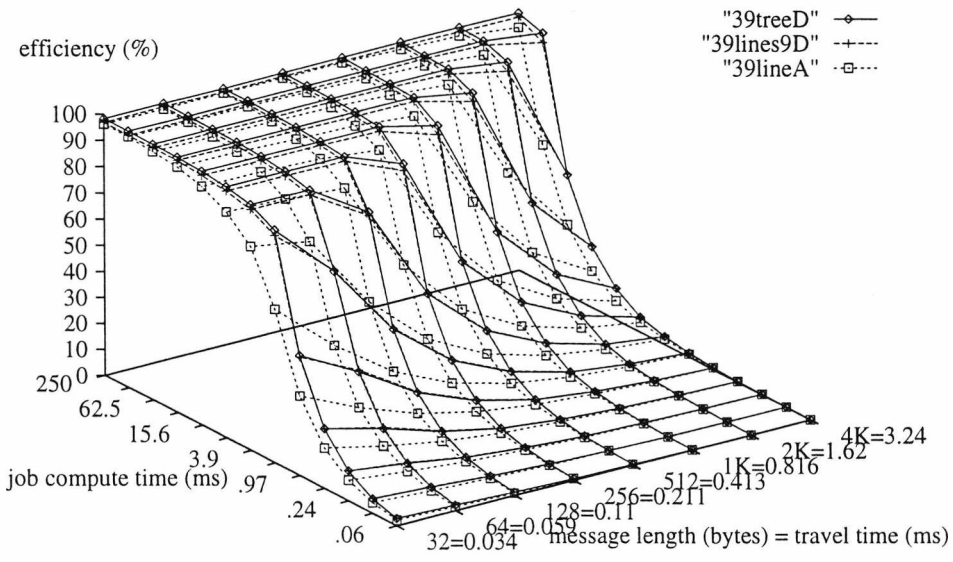


Figure 49: Efficiency for 1 and 9 lines and a tree of 39 workers

to the farm, not the number of links being used. This is interesting as although there is more potential bandwidth in the more elaborate topologies where a number of links is used, the major bottleneck is the total setup time for the job to be sent from the farmer.

4.12.5 Conclusions

In conclusion we can see that a ternary tree is the most efficient topology in which to arrange workers. More generally, fanning out over all available links is effective. That said, trees and related topologies break down just before the best linear topologies, due to the expensive ALTs and requesting buffers that are needed in the harness code and in the farmer especially to obtain the tree topology. Thus, it is also important that the highest rate of supply is used. This is especially the case when farming out over a number of links. Thus, when trying to implement a farm, it is advisable to avoid ALTs in the farmer process or near it.

This experiment has changed the theory of farm implementation developed so far. If an efficient mapping for an application can be arrived at, use a tree with harness D, instead of using harness A on a line of workers. If an efficient mapping can not be arrived at use a bidirectional line of workers using harness E. As trees perform fewer communications, smaller messages sizes are also more efficient.

4.13 Priority

When developing the code that distributes jobs, one realises it is possible for both the local and the distant worker to request a job simultaneously. Thus there is a decision to be made as to which worker should be given a job first. It is often though this decision can greatly affect farm performance and thus efficiency.

This issue is one where there are many different and often conflicting strategies. There appears to be no experimentation to back up any of these approaches.

4.13.1 Job requesting strategies

Both Welch and the work performed here with UNITY recommend the use a produce and consume strategy for distributing jobs, see subsection 3.1.3 (page 25). From UNITY it has also been reinforced that getting the work done is ultimately what is important. Spending time executing code to decide in which direction the next job should go depending on certain criteria can ultimately only slow a program down, not speed it up. Any attempt to make such a decision via algorithmic means at run time would require the C.P.U. for a period of time. This can only result in a longer execution time overall. The C.P.U.s executing decision making code as well as the application. It is also unlikely that the use of decisions would result in a smoother execution than is possible by just requesting. Here, not only is our method of evaluating which process the current job should be sent to simple, a single communication, but it also requires a very small amount of code, run time, memory and very little C.P.U. resources.

In this work it has been shown that some harnesses are highly effective. It is interesting to note that the processes of these harnesses possess some common properties,

1. if a process requires some information (in this case), it requests it when it is required,
2. if a process has some information to give away, it listens to the processes it is connected to that may require it.

As we want to execute the application as quickly as possible, it seems natural to concentrate programming effects on always executing the application when ever possible. This involves doing what we are here, trying to find out which worker process is requesting the current job that the distributor has.

This work has come across four different strategies to resolve this conflict,

1. always give priority to the local worker,
2. always give priority to the distant worker,
3. be fair over time to all whom you communicate with, and,
4. do what is appropriate for the deciding process's position in the farm's topology.

Here we only test the first three against each other, the fourth will always be specific to a topology.

4.13.2 Code used

Here are listed the distribution processes that were compared against one another.

The first process gave either the local worker or the distant worker priority simply depending on the order the folds were placed in,

```
PROC farm.out (CHAN OF JOB jobs, CHAN OF PTR local.req, local.job,
               distant.req1, distant.job1,
               distant.req2, distant.job2,
               distant.req3, distant.job3, VAL INT pointer)
  INT p, p2:
  SEQ
    p := pointer
    WHILE TRUE
      SEQ
        jobs ? len[p]::job.msg[p]
        PRI ALT
          {{{  local
          local.req ? p2
            local.job ! p
          }}}
          {{{  distant
          distant.req1 ? p2
            distant.job1 ! p
          distant.req2 ? p2
            distant.job2 ! p
          distant.req3 ? p2
            distant.job3 ! p
          }}}
        p := p2
:
```

The second process was written using the same folded code and the same interface as in the process above. This process alternatively gives workers priority,

```
WHILE TRUE
  SEQ
    -- local first
    jobs ? len[p]::job.msg[p]
    PRI ALT
      ...  local
      ...  distant
    p := p2

    -- distant first
    jobs ? len[p]::job.msg[p]
    PRI ALT
      ...  distant
      ...  local
    p := p2
```

4.13.3 The test

This test consisted of timing a full farm with each strategy in turn and seeing if there was any one that had a consistent increase in performance.

This test was run on a ternary tree as if there is any effect to be had on performance, this topology should show it up the greatest due to there being more fan out. There are most distant workers and thus more potential for parallelism, one of the arguments for always giving the distant worker priority.

4.13.4 Results and conclusions

Looking at compute bound applications, see figure 50, there are two points to notice.

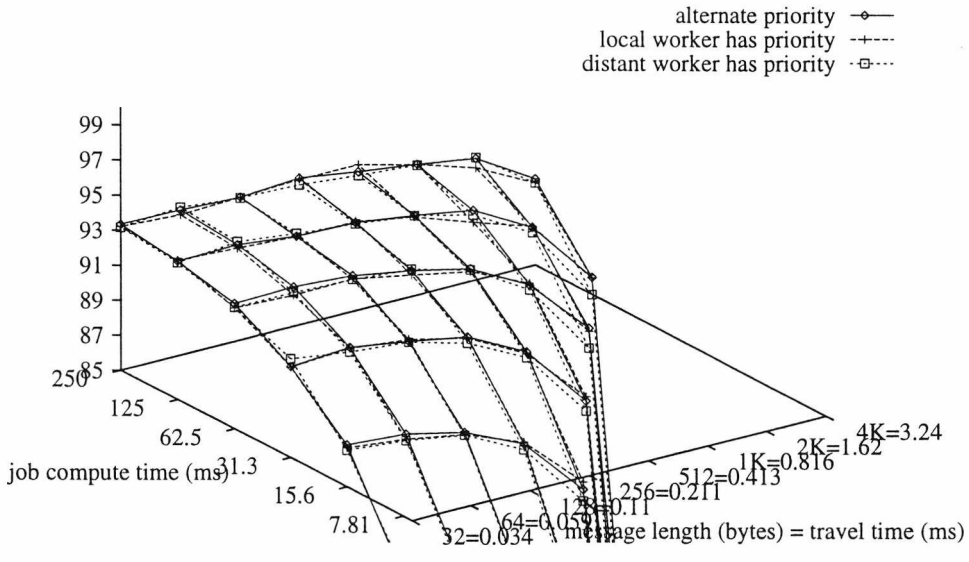


Figure 50: Breakdown for 39 workers of alternate, local and distant priority

There is no one method that is outstandingly better. This suggests there isn't much of a problem with a worker and the harness on an adjacent worker being starved at the same time. Thus other methods that involve executing code to decide with process has priority would slow the implementation down due to the execution of the decision making algorithm.

There is also very little difference between any of the methods, again suggesting that this situation doesn't occur often and thus it is not an area of implementation that is worthy of concern.

4.13.5 New fairness implementation

Another method of fairness is to give the channel used previously the lowest priority next time round.

Although this method was not compared here, precisely because there is no point in testing other aspects of an issue of little relevance. Nevertheless, documented here is an improvement to the approach used to implement this method.

The code developed in [Sar89] is,

```
PROC farm.out (CHAN OF JOB jobs, []CHAN OF PTR req, job,
               VAL INT pointer)
  INT p, p2, fav:
  SEQ
    p := pointer
    fav := 0
    WHILE TRUE
      SEQ
        jobs ? len[p]::job.msg[p]
        PRI ALT i = fav FOR 4
          req[i \ 4] ? p2
            SEQ
              job[i \ 4] ! p
              fav := (i \ 4) + 1
        p := p2
:
```

The major problem with this implementation is that the integer remainder operator is one of the slowest operations in the first generation's instruction set.

This author has realised a great deal of time can be saved by using a look-up table instead. This look-up table involves more assembly instructions, but this sequence of instructions is much quicker to execute, see discussion in subsection 4.2.1 (page 35). The length of the table needs to be twice the number of channels being alternated over. This restricts the processes generality to one size.

```
PROC farm.out (CHAN OF JOB jobs, []CHAN OF PTR req, job,
               VAL INT pointer)
  VAL lookup IS [ 0, 1, 2, 3, 0, 1, 2, 3 ]:
  INT p, p2, fav:
  SEQ
    p := pointer
    fav := 0
    WHILE TRUE
      SEQ
        jobs ? len[p]::job.msg[p]
        PRI ALT i = fav FOR 4
          req[lookup[i]] ? p2
            SEQ
              job[lookup[i]] ! p
              fav := lookup[i] + 1
        p := p2
:
```

Another suggestion due to Roebbers consisted of rounding the number of channels up to the nearest power of two and replacing the mod with a bitwise mask [SW90]. In the general case however, this will increase the number of input communications that are set up and shut down. Thus, the approach here may even be more optimal.

Unfortunately this improvement did not increase throughput.

4.14 Closing discussions and summary of conclusions

This chapter closes with, two discussions, a suggested methodology and a summary.

The first discussion is on how farming should be viewed. The second is on the fact that the parameters used to model applications in this study were appropriate.

A methodology for finding highly efficient implementations for farmed applications is recommended.

Finally, the summary reiterates the major conclusions found in this study.

4.14.1 How farming should be viewed

The current opinion held in the transputer community is that only compute bound programs can be farmed out. Through performing this work three ways of showing this point of view to be incorrect have been discovered.

The first of these being through the use of UNITY. As UNITY applications are designed and then mapped onto the hardware, one aspect of developing an effective implementation is finding a mapping that uses the hardware efficiently. For a farm this means a compute bound implementation. In conclusion, the way the application is mapped on to the architecture affects the final performance, not just the nature of the application alone.

The second argument has come through experimentation. The number of workers in a farm dictates the amount of demand there is for work. Although this is easily variable, the amount of supply a farmer can generate is fixed. Thus again it is not just the application that is responsible for an implementation being compute or communication bound, increasing the number of workers will also make a farm less compute bound and more communication bound.

The third argument has also been arrived at through the study performed here. The different topologies and harnesses result in different levels of performance. For a fixed mapping and a fixed number of workers, the choice of topology and harness can make or break whether an implementation is efficient in some situations.

The conclusion arrived at here is there are three issues affecting whether a farm will be compute bound or not. These are,

1. the amount of time the implementation takes to perform jobs and to send messages,
2. the number of workers used in the farm, and,
3. the efficiency of the topology and harness used.

4.14.2 Parameters used

This work has been made possible by being able to abstract an application down to a small number of parameters. Initially it was seen that three parameters could be used: compute time, job message length and result message length. However, it was only possible to study the data if two parameters were used: the final simplicity of testing only two were used: compute time, j and message size, m.

As has been discovered here, the way these two parameters affect the performance of a farm is identical. Whether a farm is efficient or not comes down to just supply and demand. Thus working, with only two variables has been a sensible restriction to make here. Working with both job message length and result message length would have unnecessarily complicated the study.

In terms of the practicality of real implementations the two messages may be of different sizes. In this case, the largest of the two messages should be used in calculations. Or, it may also be of use to consider each message separately, if, for example, a topology is constructed that has a large harvester bandwidth to cope with the large message size.

4.14.3 Method for finding efficient mappings

Here a method is presented for finding efficient mapping. We have yet to look at what applications are farmable, thus here we restrict ourselves for the time being to applications that consist of independent jobs.

Mapping a UNITY program design onto an architecture is independent of any particular architecture, as is this method. Thus, the method presented here is fleshed out with the results obtained here, the architecture providing the thresholds above which implementations are compute bound. This has been done for two reasons. Firstly, to show this method complete with the details of one particular architecture. Secondly, so that compute bound farms for the first generation of INMOS transputer can be developed.

UNITY has been used in this work and this chapter has talked in terms of farming out an application mapping. It is better to design an application and then map this onto an architecture. Thus, it does not matter if UNITY is not being used to develop programs here, what is important is that the program is designed, then any number of job and message sizes can be tried out. All that is needed is to be able to generate mapping parameters, these coming from the application. Thus, one needs something in the way of a design.

The method consists of looking at how to partition an application for a particular set of hardware characteristics. The best way to implement a farm is to try and find a set of mapping parameters, as the application allows, that are appropriate for transputers. This process consists of the following stages,

1. develop a mapping,
2. check to see if it is compute bound,
3. developing other mappings if required,
4. implementing that mapping with a farm that is the most appropriate.

We now look at these four stages in detail.

Develop a mapping

The first stage is to develop a mapping of the application for a farm. In the case of farming this consists of realising how the majority of the application can be performed in the workers and which parts in the farmer and the harvester. This should be done with the aim of making the mapping as compute bound as possible, as this will allow for a larger number of workers. The relevant factors to which are: the length of time to compute the job, the lengths of the job messages and the lengths of the result messages.

Check if compute bound

The next stage is then to check if this mapping is indeed compute bound. This is performed as follows using,

$$w_{max} = \frac{j}{c}$$

To measure j, for the mapping proposed, implement the algorithm to be performed by the worker processes on a single processor and measure how long it takes on average to perform a job.

Next, work out how long it takes to communicate the largest of the two message types into this worker, c. This value could also be calculated here from the bandwidth, the size of the message and the length of time it takes to start up such a message,

$$c = \frac{B}{m + s}$$

however as an implementation has been started here, performing the additional work in measuring the time taken to perform a communication is reasonably small.

Dividing the first figure by the second gives an accurate estimate of the largest number of workers this farm can cope with and still be compute bound. This method of estimating the largest number of workers is considered to be a very useful tool.

Try other mappings as appropriate

It may be worth trying again and looking at some other possible mappings. There are three situations in which one might do this: one might have a larger number of workers in mind, the mapping tried was not

compute bound, or perhaps the mapping was only just compute bound and would like to find a mapping that is more compute bound.

Doing this could be starting from scratch and coming up with a completely new mapping, optimising the code of the farmer and the harvester, or by just adjusting the values of j and m in the mapping already developed so that $\frac{j}{m}$ is larger. This could consist of configuring each job to be a group of individual jobs. This was also recommended in [PZ90] by Purgathofer and Zeiller. In terms of communication, normally we only look at the size and structure of the message. We do not normally look at how long it takes to communicate a message of any particular size. Also, we don't generally look to see how much processor resources a communication will take up in terms of the number of separate messages that are communicated. As was discussed in subsection 4.6.5 (page 58) it is much more efficient to communicate a single array than a sequential protocol consisting of the same number of bytes.

One thing to consider when tuning a mapping is the details of the architecture. Here there is a range of parameters in which the hardware works well. Working within these will help in the implementation of efficient farms. In this chapter we have looked at counted arrays and here small messages had a large set up time. Thus messages containing data achieving a minimum message size of 32 bytes will get the links performing at half maximum bandwidth. This is where grouping jobs could be useful. Instead of communicating a job that consists of say just a `REAL32`, jobs could consist of a small array of `REAL32`s. This is the same amount of work for the same amount of communication, but the links are being used much more efficiently, this could be the difference between being compute bound and communication bound. On a similar note, by about 512 bytes most of the links bandwidth is being used and larger messages will only use up more memory which could be disadvantagous. Obviously if jobs consist of simple job numbers these messages could just be individual bytes.

It is also possible that a compute bound mapping can not be found, due to the nature of the application. In this case a geometric or an algorithmic mapping should be tried.

In summary, the only thing to say is that finding a good mapping may just consist of persistence, and trying different ways of mapping the design so that as much as possible of the implementation fits into these hardware parameters.

Implement best

Once a suitable compute bound mapping has been arrived at, the best supply should be developed. This should be performed by selecting the most efficient topology and the best design of harness for that topology.

If the mapping used is reasonably compute bound then a ternary tree should be used, either with harness D or harness B performing the communication. Depending on whether usage checking can be turned off.

If one can only come up with a nearly compute bound mapping one should use a line topology with a high performance harness, such as harnesses E or F, again depending on whether usage checking can be turned off.

If flexibility of farm size is required a line topology with harness A, the harness with the most optimal performance should be used.

Finally the implementation can be fine-tuned. This includes fine-tuning both the application and the harness.

4.14.4 Summary

It appears the best way to implement a farm is likely to be a ternary tree topology and with either a harness D or if usage checking can not be turned off, harness B. These harnesses both have a process on each hardware channel, with harness D these processes pass message pointers. If the messages are of different sizes buffers should be added to decompose the link processes and allow them to more readily engage in further external communications.

There are some situations where a greater amount of supply is needed, for example to allow for a greater number of workers in the farm. Here, a line topology should be used with a high performance harness, such as harnesses E or F, again depending on whether usage checking can be turned off. These

bidirectional harnesses consists of two copies of harnesses D or B respectively and thus has a process on each hardware channel, two per link. Again harness E, stemming from harness D, has processes that between them pass pointers to messages. There is a worker process for each copy of the harness. These two separate system do not communicate with one another directly. On line topologies, sending messages involve a larger number of communications to be started, as a result, smaller messages breakdown more rapidly.

Only trees with each layer being fully populated have been used here. How to easily scale the harness is not known. This prevents easy scaling of the number of workers. However, scaling a line of workers is easy. If flexibility of farm size is required a line topology with the most efficient harness, harness A, should be used. This harness being probably the most simple possible, just a single simple process.

Here we have developed a model to calculate the maximum number of works a processor farm can have and still be efficient,

$$w = \frac{j}{c}$$

where j is the time to perform a job and c is the time to communicate a job or result message, which ever is the largest. This can either be measured or calculated by,

$$c = \frac{m + s}{B}$$

Ideally s should be suitably smaller than m, i.e. the overhead of setting up communication is suitably smaller than the message size. If this is the case, s (and its influence) can be ignored.

Farming is about supply and demand, and in particular having enough supply to meet the demand for work.

There are a few other important conclusions that have been arrived at here.

- In an implementation it is important that jobs are allocated to workers quickly.
- The overheads of farming out are minimal.
- The measured bandwidth of a transputer link here is 1.51 Megabytes per second (one byte every 0.663 microseconds) and 2.19 Megabytes per second when the links are used bidirectionally.
- It doesn't matter which worker the `ALT` in the harness gives a job to when all workers want work.
- The transputer's s is small, allowing for mappings of a fine granularity.
- Farm performance can be improved if the size of jobs is tuned so the farm finishes as quickly as possible.

Chapter 5

Farming's Range of Use

This chapter documents a search to find the limits for which farming out can be used. It also contains some examples of applications that are not usually farmed.

5.1 Overview

As discussed in chapter 3, it is possible for a wider range of applications to be farmed out than has been generally considered.

In [TD90] Tregidgo and Downton suggest that farming should be attempted first, due to its generality. However, there did not appear to be anyone saying what range of applications farming could be used to execute. Thus this study looks at this issue.

The search to find other applications for farming was performed via a number of different studies.

1. The extensions that could be made to the processor farm's mechanics were studied. If these mechanics could be extended, hopefully the range of application would be extended too.
2. Farmed implementations for two recursive applications were developed.

   The goal here was to farm out an application that possessed an internal structure not normally considered suitable for implementation on a processor farm.
3. A study of some other types of application and how they would be farmed. This led to a model of farming being developed.
4. A look at how a geometric decomposition of a set of data could be farmed.
5. A detailed look at some examples of farming. These are taken from both the work by others in computing and from some other areas of human society.
6. A more useful method of describing and viewing three parallel execution strategies was also developed.

5.2 Extending the processor farm's mechanics

This section looks at extending the mechanics through which the processor farm works. This is done through looking at the basic model of a farm and looking at the extra processes and behaviour that can be changed or added.

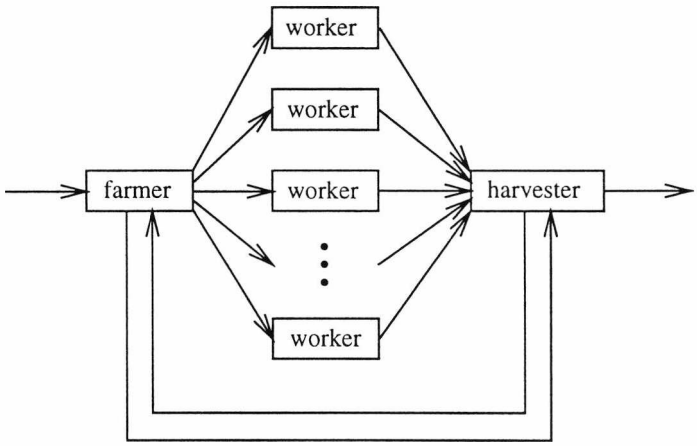


Figure 51: Structure of a farm

5.2.1 Multiple result and job types

The farms in chapter 4 all possessed the logical structure shown in figure 51 (top of next page). Here, the farmer farms out jobs to the workers who communicate results to the harvester. The farmer and the harvester may also communicate.

While keeping the same process structure, it is possible for results to be sent to the farmer, either directly or via the harvester. Thus it is possible for results to be farmed out again as other jobs or as part of future jobs.

Workers can perform different types of job, instead of just a single job type. Thus, work can be performed in a number of stages. It is also possible that some form of intermediate processing could be performed on these results by the farmer, to sort out any interdependencies between them for example, before they are farmed out again. The interdependencies found in application are often just of a nearest-neighbour variety. Thus in such situations these interdependencies can be dealt with by the farmer while other jobs are being processed within the workers of the farm.

The different jobs can be either of the same job type or of a different type. When the jobs are of the same type, this opens up the possibility for computations that are iterative or recursive in nature to be performed. When the different sets of jobs are of a different type, this allows for the various parts of a several stage calculation to be farmed out. There is an example in [CU90] of a farm that computes different types of jobs. This implementation also interleaved the differing job types to aid obtaining a higher degree of speed-up.

5.2.2 Automatic load balancing

There is no need for different job types to be performed by a completely separate processor farm. All work can be executed by the same set of workers. This automatically balances the work load at runtime, an advantage farming has always had. Doing this alleviates the need for any load balancing to be performed during the development of the implementation, through the moving of processors back and forth between different processor farms. The different jobs will have different average processing times. It is unlikely that the ratio between these will be obtainable exactly with the number of transputers within a network.

All workers performing all job types will naturally affect memory usage; as all processors will require the code to compute each job type. It is possible to have only a few workers that can execute more than one type of job. However, such an arrangement is again likely to increase the amount of design work that will need to be performed. Here jobs would need to be directed to particular workers and memory is considered to be cheaper than programmer time. This extra memory is not likely to be a problem as most large applications do not necessarily consist of large jobs, but instead consist of a very great number of small jobs.

5.2.3 Performing work depth and breadth first

When results are returning to the farmer and then being farmed out as more jobs, a decision can be made whether the farmer should complete one thread of the application before another (a depth-first computation) or to complete one set of jobs before another (a breadth-first computation).

The advantage of the depth-first method, as always, is as the results return to the farmer there is no need for previous leafs of the computation to be stored, thus there is no need, or in some cases very little need, to make provision for storage of these previous results.

The advantage of the breadth-first method is that all stages of the application progress forwards together.

5.2.4 When there is an ordering with some work

It should be possible to farm out applications whose computation consists of a directed graph of dependency, i.e. there are some parts of the application that can only be performed after others. As always, the amount of parallelism possible is dictated by the application, and thus the number of processors usable.

5.2.5 Multiple harvesters

Just as implementors can choose to have the harvester separate from the farmer, it is also possible for there to be a number of harvesters, though this will entail a more elaborate result collecting mechanism. An example of where this might be useful is a processor farm that generates different results for different graphics monitors.

5.2.6 Multiple farmers

Similarly, farms might be constructed with a number of farmers. This could be advantageous due to the greater bandwidth obtainable out of a greater number of processors. There are two strategies that have the concept of multiple farmers.

Firstly, workers could generate work for other workers downstream. This could be performed repeatedly, and thus a farm would consist of a hierarchy of farmers. This has also been described in the occam 2 reference manual [Inm88].

It might be possible to have a number of farmers that are independent, i.e. they do not communicate with one another. Each of these would supply work to its own set of workers. With some applications such an approach might achieve a high degree of balance. It might be possible to have a main farmer existing only in the program's proof of correctness. The real farmers obtain their jobs from this. In practice the farmers respond differently to the same situation in order to keep the farm running harmoniously. Results from all the workers could still be collected by one harvester.

Whether either of these strategies are worth implementing remains to be seen. One obvious advantage is that the bandwidth of jobs out of a number of different processors is going to be higher than out of just one. However, whether having multiple farmers can lead to implementations that maintain load balancing remains to be seen. It is the fact that a farm is simply automatically load balancing that helps to lead to its potential efficiency. It would seem that a multi-farmer farm would be less likely to be as balanced than a single farmer farm. A single pool of work is definitely easy to manage.

5.2.7 The farmer-harvester bottleneck

For very large farms, there can be the question that the speed of the farmer or the harvester process can be the bottleneck of the entire farm. If for the application in hand such synchronised processing is creating a bottleneck the dependent processing could be pipelined over a few processors. So far it has always been that the farmer and the harvester have been a single process on a single processor each. There may be situations however, where the tasks to be performed by some farmers and harvesters can be decomposed into a number of processes and run across a number of processors.

5.2.8 Varying the size of the farm

It is possible to vary the number of workers throughout the execution of the program. Adding more workers to a processor farm involves two stages. First, the workers must be connected to the farmer or into the harness. Second, the workers must then be initialised with the code with which they can execute jobs. If processors are removed from the farm care must be taken to ensure that no jobs are lost.

5.2.9 Farming out all farmable applications

As has been discussed at the beginning of this section, workers can be used to execute more than one type of job. This can be taken further. Just as it is possible to farm out one application that consists of several separate stages, it is also possible to have any number of unrelated applications executing on the same farm. As the applications are completely unrelated their jobs can be computed side by side on the same processor farm at the same time, with all of the advantages of dynamic load balancing to be gained.

5.2.10 The processor farm is one part of an application

It is possible to have an application consisting of several parts or components, one of these being a farm. The other components could produce tasks for the farm to compute, the farmer acting as a server. There is already one example of a farm being used as one part of a parallel implementation is [BTU88].

5.2.11 Summary

The largest pieces of insight this work has provided us with so far is that,

1. the computation may proceed in several stages and thus consist of jobs that produce only intermediate stages of results,
2. the jobs may be of different types, and,
3. jobs may have some computational dependence between them.

The many stages of work can be either performed in a depth first or in a breadth first arrangement, depending on the requirements of the application or the amount of memory available. A breadth-first strategy requiring all intermediate results to be stored.

It is easier to have all of the processors being able to execute all of the parts of an application, though again this may be restricted by the amount of memory that is available on any one particular hardware configuration.

It is possible to have more than one harvester present in one farm for different types of result. It is also possible to have more than one farmer, although this is likely to lead to a lack of coherency of information about what work has been performed and the lack of a single central pool of work. This could result in one part of a farm being starved of work while another is busy. Multiple farmers may be best implemented by having a hierarchy of farmers.

A variable number of workers may be easy to arrange on particular hardware. To be done efficiently, and easily from a programming point of view, it requires fast initialisation and the concept of dynamic process creation.

The processors considered here are of the message-passing variety. This technique may be amenable to other multiprocessor architectures.

5.3 Farming out the towers of Hanoi

At the time this work started no examples of recursive applications being implemented on a farm in the literature had been seen. Thus it was decided to show by example that such implementations were possible. Presented here are two examples that show such applications can be implemented on a farm. The first of these is documented in this section, the second is shown in the next section.

This first example application is the towers of Hanoi. This example is a simple one, being often used to introduce the concept of recursion.

The towers of Hanoi has an expansion factor of two, i.e. for every recursive call, another two calls are generated. A common example of a recursive problem with an expansion factor of one is the factorial function.

5.3.1 Building a farmed implementation

Traditionally the towers of Hanoi is expressed using recursion,

```
PROC hanoi (VAL INT n, VAL BYTE from, to, via)
  IF
    n > 0
      SEQ
        hanoi (n - 1, from, via, to)
        move (n, from, to)
        hanoi (n - 1, via, to, from)
    TRUE
      SKIP
:
```

However, recursion isn't present in occam.

This problem is easily solved. Recursion is implemented using a stack. All that needed to be done is to implement a stack and the recursion by hand. As soon as it was realised this was the way to proceed, it became obvious that the contents of this stack could be farmed out. There can be many workers in a farm and thus any parts of the problem that can be solved in parallel will be.

Thus, instead of taking a problem from the top of the stack and processing it directly, the contents of the stack were farmed out to workers. Thus many parts of the solution would be in the process of being unwrapped by workers simultaneously.

One problem with going parallel

Going parallel leads to a problem: we lose the context of where we are within the problem. Previously the solution was unwrapped sequentially, a sub-problem would be removed from the stack, unwrapped and the new sub-problem would be pushed onto the stack. Where the computer was in solving the program would be known, the program always working within the same part or context of the problem at any one time. By going parallel we lose this.

This problem lies in the fact that at the farmer different parts of the problem and the solution will be going out and arriving simultaneously. We must know which peg movement instructions goes where within the whole solution and where the different sub-problems fit into the whole of the problem. From the normal information we have, the disc number and the order of the pegs, we do not have enough information to distinguish any one part of the solution from another, especially as the smallest disc will be moved many times. In fact there are only six ways in that three pegs can be arranged. How this can be solved is looked at next.

Labelling

This problem's solution is generated by expanding the problem,

```
hanoi 64, from, to, via
```

into sub-problems,

```
hanoi 63, from, via, to
move 64, from, to
hanoi 63, via, to, from
```

as normal. Here this will be expanded in parallel to,

```
hanoi 62 from, to, via
move 63, from, via
hanoi 62, to, via, from
move 64, from, to
hanoi 62, via, from, to
move 63, via, to
hanoi 62, from, to, via
```

A labelling scheme is needed to identify each part of the solution uniquely from all of the others. A technique is also needed for generating each new label from its ancestors. Part of the problem here lies in the move instructions being generated at all levels of the problem.

The method used here to generate a new label from its predecessor was developed as follows. Each move instruction can be given the same label as the sub-problem that generated it. Further, a smaller label should be given to the sub-problem that comes before the move instruction and a larger label should be given to the sub-problem that comes after it.

This method needs to be arranged so it works recursively from any sub-problem. Not only should the two sub-problems generated have labels respectively smaller and larger than it, but all of the labels then generated from them are also respectively smaller and larger than the original label.

In order to do this, how much the problem expands at each level, and how far the recursion will go, needs to be taken into consideration in the design of the labelling system. Here the number of sub-problems generated by each sub-problem is two. The number of discs the program should be able to deal with is 64, the largest size associated with the problem.

By starting with a label for the original problem, this first problem will generate two sub-problems and a single move instruction. As this first expansion takes us from the original problem of size 64, say, to two problems of size 63, and a move instruction, the move instruction can be given a number that is simply based on a number like 63 or 64.

As this problem has an expansion factor of two, each level of the problem can be encoded by a single bit for whether or not the sub-problem goes before or after the move instruction of the same level.

	01 prefix sub-problem
10 original sub-problem becomes	10 move instruction
	11 suffix sub-problem

Here this algorithm is checked to see if labels are generated correctly. As will become clear, care must be taken here. The first job number expands in the following way,

```
      0100
1000  1000
      1100
```

This is what is desired. The prefix job generated by this expands in the same manner,

```
      0010
0100  0100
      0110
```

However, looking at the suffix job of this,

```
      0101
0110  0110
      0111
```

is wanted. If the numbers were to just be shifted down and added, the following would be produced,

```
      0011
0110  0110
      1001
```

Thus, only the current 2 bits of the working window should be shifted down. The upper bits have already been set and these should be left in place. This operation can be performed by using the bit pattern `01`. Taking this away from a sub-problem label number would give the label of the sub-problem to prefix this. Similarly this value can be added to a sub-problem label number to give the label of the sub-problem to suffix this.

`10` - original problem label
`01` - prefix

prefix sub-problem `10 - 01 = 01`
suffix sub-problem `10 + 01 = 11`

This method gives the correct numbers for the two new sub-problems generated and correctly preserves the history of these sub-problems as required.

This leaves the question of what label should the program initially start with. This is done by taking the size of the problem to be solved and setting that bit (counting from one) of an otherwise clear label. Thus if we were to have 64 discs, the label of the problem, the first problem, would be `1 << (64-1)`.

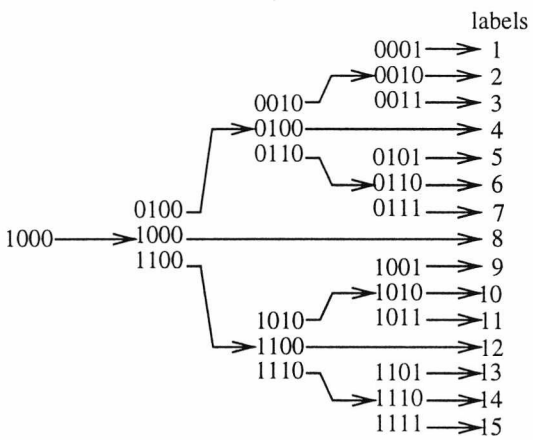


Figure 52: Expansion of the towers of Hanoi problem

Figure 52 shows how this works as the labels expand. As can been seen here the level of the recursion becomes encoded into the labels.

Developing the farmer and the workers

Now each part of the solution can be suitably labelled, the essence of the stack management code in the farmer can be separated from the application in the worker.

The first thing that needed to be performed in order to separate the application into two processes was to develop the protocols so the two processes could communicate intermediate stages of the application. In a normal Hanoi implementation the sub-problems would consist of a disc number and the three pegs, and a move instruction would consist of a disc number and the two pegs. Both messages contain a label.

Algorithm optimisation

One optimisation has been made here. For simplicity the towers of Hanoi are usually written,

```
PROC hanoi (VAL INT n, VAL BYTE from, to, via)
  IF
    n > 0
      SEQ
        hanoi (n-1, from, via, to)
        screen ! from, to
        hanoi (n-1, via, to, from)
    TRUE
      SKIP
:
```

When $n = 0$ however, this generates two unnecessary sub-problems that result in nothing being generated. This is more relevant than normal as these extra messages take up extra communication time and bandwidth through being sent back to the farmer and sent out. In between this these messages take up stack memory and processor time through being ordered. Normally this would waste one position on the stack and some time. Here, there are many $n = 0$ jobs to be stored on the stack and passing them around takes a lot of time. Thus, removing this overhead will be quite a saving, there are 2^N extra of these $n = 0$ sub-problems in total. This is one level of recursion more than is needed. This would create a redundant sub-problem for each valid part of the solution. This is even more relevant here as instead of the parameters of the recursive calls being passed, they are communicated, which has a higher overhead. Thus, in order to be more efficient and save on the amount of memory used, and the number of communications performed, $n = 1$ is dealt with as a special case and only one result is generated.

```
PROC hanoi (VAL INT n, VAL BYTE from, to, via)
  IF
    n > 1
      SEQ
        hanoi (n-1, from, via, to)
        screen ! from, to
        hanoi (n-1, via, to, from)
    TRUE
      screen ! from, to
:
```

Also the way the labelling system has been developed does not allow for n of size zero. Part of the reason for this is that $n = 0$ would require an extra bit in the label. Here, given a maximum label size of 64 bits, this would prevent the solving of $N = 64$.

Stacks

As in this application there are sub-problems and peg move instructions, the farmer needs to manage two stacks. The sub-problems consist of the same disc number and three pegs that a conventional Hanoi program would contain and here we also need our label. The peg move instructions also consist of the label and the 'from' and 'to' pegs.

Naturally we are interested in designing a program to finish as soon as possible. In this case the sooner the program can start printing the solution the quicker it can finish. There are two aspects to this. Getting to the start of the solution as quickly as possible and making sure we can then print the rest of the solution as quickly as possible.

Thus we are interested in progressing as quickly as possible from the start of the problem to the start of the solution, in this case the first move instruction at the top right of the solution as it is drawn in figure 52 (on previous page). In general this means we should always try to be solving the sub-problems that are on the leading edge of the problem. Those closest to the top of the problem and also closest to the right

From looking at the runtimes for the eight peg problem, as it takes 0.062 of a second to solve on one worker, the optimum possible run time for eight workers should be 0.0077 of a second. However, the actual time it takes is nearly half a second. This is a speed-up of 0.128, or to put it another way a slow down of 7.8.

If more workers were to be added to a communication bound implementation, the time the farm would take to finish would remain about the same. This is because the amount of work being performed is the same. It is just there are some extra workers that are not being supplied work. So, as a dramatic slow down is being obtained here, there must be some form of new behaviour related to increasing the number of workers that is hindering the performance of the farm in this dramatic way.

The reason why the program runs so much slower when more workers are added is likely to be caused by something major. The first reason considered was an issue had been looked at in subsection 5.3.1 (page 126). Although the farmer is always trying to give out the leading edge, as soon as it does, the leading edge is in the farm being communicated out, expanded and communicated back. Thus, the farmer doesn't have that leading edge job any more and won't again until the results of that job come back. However, as there are still many requests for jobs and the farmer is programmed to give out more work, what happens instead of just the pure leading edge of the calculation being expanded upon, the area just behind the leading edge is also expanded upon at the same time. This area of expansion has a size and the more workers there are in the farm the larger this area of progression is. The size of this area of expansion is directly proportional to the capacity of the farm. By thinking in terms of farm mechanics we can see that the area of expansion's size is the number of jobs the farmer can output into the farm before receiving the results of the previous leading edge back and thus what is the very tip of the leading edge back again. So, the greater the capacity in the farm the larger this area being expanded will be.

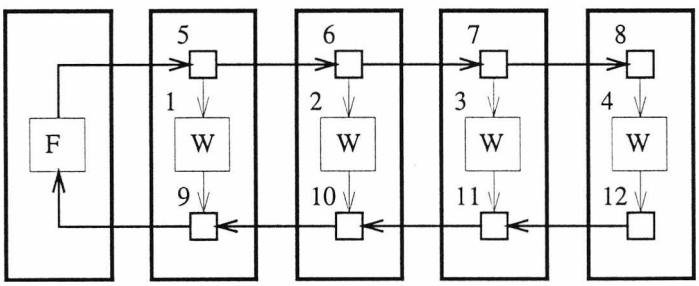


Figure 54: Capacity of a four worker farm

As can be seen from figure 54, a farm's capacity is made up of the number of workers, four here, plus the amount of buffering in the harness, eight. As there is some form of buffering capacity in this harness (it is difficult and also inefficient to have a harness with no buffering capacity) by definition the number of workers working on the leading edge can only make up one part of this edge, a third in this case. Thus, as only a third of this is being expanded at any one time and then there is a latency, due to this buffering, before any more work can be performed on these sub-problems.

It is worth mentioning here that at the beginning of the program running the work is performed in a breadth first manner.

When watching a farm of eight workers generating the move instructions, it can be seen that there seems to be a long delay after printing the first dozen move instruction.

Another more practical reason that can be seen for this implementation slowing down is due to, as is discussed below, the few hundred move instructions that are sorted into order on the instruction stack. As this implementation uses just a simple insert sort type algorithm to perform this operation this will perform badly when trying to ripple move instructions down to the bottom of the stack when this stack becomes as large as it does. This is very time consuming and thus when there are a large number of workers the farmer probably spends more time performing this slow sort than doing anything else. Using a better algorithm here, such as a binary chop search to find the insert point and then a block move to create the gap would have been more efficient.

Another point to note is that the towers of Hanoi has a fine granularity and this has been implemented

directly here, resulting in a communication bound farm. Thus the performance of the application could be better if the grain was to be coarsened. This could be done by performing a number of recursions on one transputer. However, although this may improve things, due to the problem discussed above, run-times would still not likely to be anywhere near optimal just simply because of the application's shape and nature. On its own however, this idea is a good one and usually it can be used to good effect.

Maximum sizes

As well as finding out the runtime of the program, there are another three aspects that are of interest: the largest number of items on the two stacks and largest number of items in the farm at any one time. We are interested in the stack sizes here as they are implemented by hand and in occam we need to know how much space will be required for the sizes of problems we wish to solve.

The figures shown here are for the same four runs as above and two other runs performed where the eight workers were configured into a pipeline topology as well as the ring topology. As we shall see this can have quite an effect on the performance of the program as results produced by workers near to the farmer can travel straight back and be dealt with immediately. At the low level this change makes the largest improvements when the jobs are on very tip of the leading edge.

It was expected that the maximum number of jobs in the farm at any one time would just be equal to the farm's capacity for messages, i.e. the number of workers plus the number of buffer spaces in the harness.

	workers		
N	1	8 (ring)	8 (pipe)
4	2	7	6
8	2	16	14

Table 6: Maximum number of jobs in farm at one time for Towers of Hanoi program

This turned out to be the case. However, the largest sizes the two stacks obtained were completely different to what was expected.

	workers		
N	1	8 ring	8 pipe
4	5	1	1
8	13	16	1

Table 7: Maximum sizes of sub-problem stack for Towers of Hanoi program

It had been thought that the maximum size of the sub-problem stack would just be proportional to the size of the problem being solved. However, as can be seen when a large farm is used to solve a small problem, the sub-problem stack only ever stores at most one job. A likely reason for this is that although there are a quite a few sub-problems generated, they will get stored in buffering capacity of the harness. Thus as soon as a sub-problem returns from the farm it is farmed out again immediately. This however is not the case for when a large problem is farmed out on a larger farm, here the largest size the sub-problem reaches is much larger. This could be due to there not being a linear but some form of exponential relationship.

The move instruction stack had to much larger than was anticipated. It has to be almost as big as the problem size when a large number of workers is used. This is brought about by the breadth first nature in which the problem is solved on the farm.

This behaviour is brought about because as soon as the leading edge of the problem is given out, all the jobs that are then given out after that are not a part of the leading edge of the solution.

As has been discussed before, as soon as the farmer gives out the job that is on the leading edge of the problem, until it gets this sub-problem's prefix sub-problem result, all the jobs the farmer will give out in

	workers		
N	1	8 (ring)	8 (pipe)
4	5	10	8
8	13	234	247

Table 8: Maximum sizes of move instructions stack for Towers of Hanoi program

the future will not be a part of the leading edge. The sub-problems given out in this way will then generate more sub-problems that will just fill up the lower regions of the stack more and move instructions that will fill up the bottom of the move instruction stack until these are needed near the end of the program. In this instance the performance of the application is hindered by the often large amounts of buffering that are often used to engage the links in parallel. All this extra buffering is filled up with even more sub-problems widening the leading edge and resulting in the memory problems discussed above.

5.3.3 Improving the performance of Hanoi

At some stage the implementation should be arranged to be compute bound. This is could be achieved by grouping jobs together, perhaps implementing some recursion locally within the workers.

Solving this imbalance in execution like this would involve studying and thinking about how the problem expands upon execution. By doing this one can give the workers a strategy for expanding the problem carefully in the various stages of execution. In essence this consists of carefully arranging for the program to progress well.

The disadvantage with this tuning is it needs to be performed for each size of problem. This requires a great deal of specific knowledge about how the architecture would go about executing the problem as it unravels. Such knowledge of what will happen at run time is only really obtainable by spending time observing or modelling how a program runs.

All this would mean polluting the natural structure by which the program works. Although arranging the program to perform well might result in a speed up, we are not interested in this here. This program is just an example. Thus it was decided here to find an application that was more appropriate to farm out, due to it not needing to both expand and output results throughout the computation.

5.4 Farming out quicksort

Although the previous section shows that a recursive application can be implemented on a farm, it doesn't show doing so is worth while from a performance point of view. Thus it was decided to look at a more suitable example. Namely one that only needed to output its results at the end of the computation, not all the way throughout it. Thus quicksort was chosen as a second example. Quicksort has two advantages. Firstly, the size of the data throughout the running of the program is constant, not an exponential expansion as with the towers of Hanoi. Thus making quicksort a much more practical application to farm out. Secondly, more of the internal parallelism within the design of the transputer tends to be used with real (non toy example) applications.

The implementations and performance of two programs are presented and compared in this section. The first is a sequential quicksort program, very close to a conventional implementation. The second farms out quicksort similar to the way the towers of Hanoi was farmed out, through storing partitions on a stack and by simultaneously farming out as many partitions as possible.

It was realised this farm would start with only a few large jobs at the beginning of the programs execution and could not obtain a high degree of speed up. Thus a second mapping was conceived. This consists of decomposing large partitions into equally sizes segments. Once partitioned the resulting *le* and *ge* segments could then be inserted into the appropriate ends of an array. The two result partitions could then be grown towards each other and eventually would meet when all the results have been received. These two partitions could then be simultaneously farmed out in the same way. This mapping has not been successfully implemented, but is documented here both for completeness and as some of the

development decisions for the other two programs were made so that the implementation of this mapping would be much easier.

In the early stages of planning it was realised quicksort could be implemented more directly, without performing any major restructuring of the algorithm's workings, as was needed in the implementation of the towers of Hanoi. This is possible as quicksort is tail recursive. As will become clear, quicksort can be performed using a stack of partitions by repeatedly removing a partition from the stack, partitioning it and placing the two resulting partitions directly on the stack.

The conventional sequential program was developed first. The program performing the recursion by hand on a stack. As well as allowing us to compare a farmed version of quicksort with a conventional implementation, something we were not able to do with the towers of Hanoi, the development of this conventional program also served as a useful stepping stone towards the farmed versions.

This seemed the best and most natural way to progress. Since this work started, Barrett has also written about developing parallel programs from sequential one using the semantic preserving transformations possible in occam [Bar93]. The advantage there being sequential programs and thus also equivalent parallel programs do not deadlock.

Finally the performance of the programs developed here were compared.

5.4.1 Conventional version of quicksort

Here how the sequential version was developed is looked at. This was developed by obtaining the original algorithm from [Hoa61], converting this to occam and finally by checking thoroughly that the implementation was working correctly.

Quicksort: recursive partition

Quicksort [Hoa61, Hoa62, HJ89d] consists of recursively performing the operation of partitioning on an array,

```
PROC quicksort ([]TYPE a)
  INT i, j:
  SEQ
    i, j := partition (a)
    quicksort ([a FROM 0 FOR i])
    quicksort ([a FROM j FOR ((SIZE a) - j) + 1])
:
```

It is simple to see that this could be written in a parallel language with recursion as,

```
PROC quicksort ([]INT a)
  INT i, j:
  SEQ
    i, j := partition (a)
    PAR
      quicksort ([a FROM 0 FOR i])
      quicksort ([a FROM j FOR ((SIZE a) - j) + 1])
:
```

As with the towers of Hanoi, the recursive element of the program can be separated from the partition algorithm. Also, as with Hanoi, the recursion can then be written in occam by using a hand built stack. Thus, partitions can be taken off a stack and farmed out. The results of this are pairs of partitions that can be placed on the stack ready to be farmed out as smaller partitions.

Recursion

Here the recursion in,

```
PROC quicksort ([]INT a)
  [max.size]INT front, end:
  SEQ
    front, end := partition (a)
    quicksort (front)
    quicksort (end)
:
```

needed to be re-implemented by taking a partition from a stack, dividing it into two and then putting these on the stack. The pseudo-code for this is,

```
SEQ
  push (a)
  WHILE stack has some contents
    SEQ
      pop (a)
      front, end := partition (a)
      push (front)
      push (end)
```

Having developed the basis of the recursion that was to be performed by the program, work progressed on to implementing the partition algorithm.

Partition algorithm

One reason for wanting to develop a parallel implementation such as a processor farm, is that such implementations are very fast. If one is already interested in high performance, a highly optimised algorithm should also be used. Thus a suitable optimised version of this partition was found.

The first implementation of partition looked at here was not particularly efficient [Kru87]. The second implementation of partition looked at was Hoare's original 1961 algorithm [Hoa61]. This was written to be very efficient. Thus it has been used here. Being highly optimised this version contains goto statements and is not as clearly structured as is possible. The original Algol-60 reads as follows,

```
        key.index := random(front, end);
        key := a[key.index];
        i := front;
        j := end;
up:     for i := i step 1 until end do
          if key < data[i]
            goto down
        i := end
down:   for j := j step -1 until front do
        if data[j] < key
          goto change
        j := front
change:if i < j then
          begin
            exchange(data[i], data[j])
            i := i + 1
            j := j - 1
            goto up
          end
        else if i < key.index then
          begin
            exchange(data[i], data[key.index])
            i := i + 1
          end
        else if key.index < j then
          begin
            exchange(data[key.index], data[j])
            j := j - 1
          end
```

The next step was to convert this algorithm into occam.

Key selection

The first important aspect of implementing a good version of quicksort is making sure the algorithm selects a good key from which to partition the data. Selecting a bad key in quicksort can result in the algorithm sorting in n^2 time. The key is thus naturally best selected from the data by choosing a value that is medium to the value space of which the data consists.

Hoare simply selected a random key,

```
key.index := random(front, end);
key := a[key.index];
```

Here it was decided to select the medium value from the first, last and centre element of the array. This is reasonably quick to perform and avoids the n^2 worse case performance. Thus the following was developed,

```
INT middle:
BOOL front.end, middle.end:
SEQ
  middle := (front + end) >> 1
  IF
    data[front] < data[middle]
      IF                          -- front < middle
        data[middle] < data[end]
          key.index := middle     -- front < middle < end
        TRUE
          IF                      -- front < middle >= end
            data[front] < data[end]
              key.index := end    -- front < end < middle
            TRUE
              key.index := front  -- end <= front < middle
    TRUE
      IF                          -- middle <= front
        data[middle] < data[end]
          IF                      -- middle <= front & middle < end
            data[front] < data[end]
              key.index := front -- middle <= front < end
            TRUE
              key.index := end    -- middle < end <= front
        TRUE
          key.index := middle     -- end <= middle <= front
  key := data[key.index]
```

Forward search fragment transformation

The partition algorithm was implemented by carefully working out what was meant by the Algol-60, a language not known, and then turning the code into efficient occam.

There are only two parts to the algorithm where the transformation from Algol to occam was intricate. Both of these perform the same type of task, searching for the next value that is not in place. As both transformations are identical, just the first is shown here.

The original Algol-60 for this search loop is,

```
up:     for i := i until end do
          if data[i] > key then
            goto down;
        i := end;
down:
```

It was decided the first step should be to replace the for loop into a type of loop that occam had and that was appropriately fast.

With loops the most important thing is that they start and terminate with the correct values. This loop is looking for a value that is out of place, it looks until it finds the end of the list, all values being available for swapping. In contrast to many other languages, occam's "for" loop declares its own variable and in the original algorithm the variable `i` is used outside the loop. This suggested a `WHILE` loop, giving,

```
SEQ
  GUY
    :up
  WHILE i <= end
    SEQ
      ...  IF statement
      i := i + 1
  i := end
  GUY
    :down
```

Here the IF statement has been translated into occam using the assembler directive GUY,

```
IF
  data[i] > key
    GUY
      J .down
  TRUE
    SKIP
```

After putting these two parts together,

```
SEQ
  GUY
    :up
  WHILE i <= end
    SEQ
      IF
        data[i] > key
          GUY
            J .down
        TRUE
          SKIP
      i := i + 1
  i := end
  GUY
    :down
```

It was realised the loop could undergo some transformation, as the first branch of the IF statement terminates the loop. Thus the result of the incrementation is only preserved if the second branch of the IF statement is executed. Thus the loop can be transformed into,

```
SEQ
  GUY
    :up
  WHILE i <= end
    IF
      data[i] > key
        GUY
          J .down
      TRUE
        i := i + 1
  i := end
  GUY
    :down
```

Further, if this loop is followed through, it can be seen that when `data[end]` is larger than `key` the program goes to the `down` label and on to the next loop with `i` still equal to `end`. However, if `data[end]` is less than or equal to `key` the variable `i` will be incremented to `end + 1`, the program drops out of the end of the loop and instantly resets the value of `i` back to `end`, the value before the incrementation was performed (this is left over from the way for loops tend to leave their index variable advanced by one beyond the finishing value). Thus when the loop terminates naturally, there were no out of place values to be swapped (the program has got to the end of the array and has found all the values should be in the first partition). Thus, the actual value of `data[end]` is irrelevant, and this loop can be further optimised to,

```
SEQ
  GUY
    :up
  WHILE i < end
    IF
      data[i] > key
        GUY
          J .down
      TRUE
        i := i + 1
  GUY
    :down
```

Now the loop performs two comparisons, if the first one fails, the loop terminates, if the second one success, the loop terminates. Thus the loop can be rewritten as,

```
WHILE (i < end) AND (data[i] <= key)
  i := i + 1
```

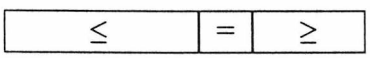
As was stated above, the second key searching loop was modified in a similar fashion.

The final code arrived at was,

```
SEQ
  key.index := index.of.median (front, middle, end)
  key := data[key.index]
  i, j := front, end
  -- start of loop
  GUY
    :up
  WHILE (i < end) AND (data[i] <= key)
    i := i + 1
  WHILE (j > front) AND (key <= data[j])
    j := j - 1
  IF
    i < j
      SEQ
        swap (data[i], data[j])
        i := i + 1
        j := j - 1
        -- goto up
        GUY
          J .up
    -- end of loop
    i < key.index
      SEQ
        swap (data[i], data[key.index])
        i := i + 1
    key.index < j
      SEQ
        swap (data[key.index], data[j])
        j := j - 1
    TRUE
      SKIP
```

The three results of partition

After dry running through the algorithm by hand once to see how it worked and what it did, it was realised the second and third conditional processes of the `IF` were responsible for moving the key element in between the two partitions. Thus, the array was partitioned into three parts: less than or equal to the key, equal to the key and greater than or equal to the key.



The way the algorithm worked, this equals partition would contain at least the key and potentially some other data elements that were equal to the key and had started out being near the final boundary. Once the algorithm was up and running it was noticed the indexes generated by the algorithm for further partitioning didn't include this area that was equal to the key, due to this part of the array clearly being sorted and already in place.

When running a small test sort of 79 characters it was found partitioning the array into two instead of three partitions resulted in a massive 38% loss in performance. By ignoring parts of the array already sorted and in place, the algorithm has a smaller number of elements to sort and these will be partitioned into a smaller number of partitions.

If this highly efficient aspect of the algorithm was to be kept, it would involve a storage problem when farming out segments of partitions. As result segments could arrive from the workers in an arbitrary order, the = segments would have to be stored by the farmer until their final position between the $\leq$ and

the $\geq$ partitions was known. This location would only be finalised after all the other segments had arrived. As occam does not have memory allocation, it was thought developing one would involve a great deal of work, considering the small aspect of the algorithm for which the allocator was needed. It was decided to avoid this work as the implementation of a memory allocator was not of any great direct benefit to this research. This work was attempting to show that different mapping strategies would result in different levels of performance and speed-up, regardless of the particular details of the application's mechanics. Thus, as it was desired to perform comparisons on just the different performances obtained by the different mappings, it was decided to compare different mappings of the same programs, not different mappings for different programs. Thus the inclusion of this aspect of the algorithm was not used in these implementations of quicksort.

Push ordering

There is also another subtle behavioural aspect of the algorithm that is discussed by Hoare in the original paper. If the larger of the two partitions is always pushed onto the stack before the smaller, the stack space behaviour is log_2 of the problem size.

This was made use of here. The algorithm used here has two variables, `lt.rng` and `ge.rng` and are the ranges (one less than the length) of the *l*ess *t*han and *g*reater than or *e*qual to partitions.

```
{{{  push arrays >= 2 onto stack, largest first
SEQ
  lt.rng := j - front
  ge.rng := end - i
  IF  -- 1 is the shortest length we have to deal with
    ge.rng > lt.rng  -- Only push partitions >= 2 in
      -- length, as the above ge.rng is >= 2
      -- (ge.rng > lt.rng and lt.rng >= 1) it should
      -- be pushed, and first as it is the larger
      {{{  push ge.rng; and lt.rng if big enough
      SEQ
        ...  push ge.rng
        IF
          lt.rng >= 1
            ...  push lt.rng
          TRUE
            SKIP
      }}}
    ge.rng < lt.rng
      ...  push lt.rng; and ge.rng if big enough
    -- lt.rng = ge.rng here so are they both >= 2
    lt.rng >= 1
      SEQ
        ...  push ge.rng
        ...  push lt.rng
    TRUE
      SKIP
}}}
```

The partitions are pushed on the stack as follows,

```
{{{  push ge.rng
SEQ
  fronts[stack.ptr] := i
  ends[stack.ptr] := end
  stack.ptr := stack.ptr + 1
}}}
```

```
{{{  push lt.rng
SEQ
  fronts[stack.ptr] := front
  ends[stack.ptr] := j
  stack.ptr := stack.ptr + 1
}}}
```

Insert sort

In quicksort implementations insertion sort is sometimes used as the sorting algorithm once the number of items to sort is small. Insert sort being quicker for small partitions. For this reason insertion sort has been used here when partitions reach a suitably small size.

The only aspect of this algorithm that needs to be explained is the variables used to drive the replicator. The variable `lt.rng` has just been discussed. To save on variables, `lt.rng` was used here to refer to the length of the partition being worked on before it is partitioned. The variable `front` is used to store the offset of the partition being worked on from the beginning of the array.

```
SEQ i = front + 1 FOR lt.rng
  IF
    data[i] < data[i - 1]
      SEQ
        j := i
        element := data[i]
        WHILE (j > front) AND (element < data[j - 1])
          SEQ
            j := j - 1
            data[j + 1] := data[j]
        data[j] := element
    TRUE
      SKIP
```

Program

Putting all this together gave the sequential program. All the variables here are INTs or arrays of INTs.

```
SEQ
  stack.ptr := 0
  ...  init array with some values
  fronts[stack.ptr], ends[stack.ptr] := 0, max.size - 1
  stack.ptr := stack.ptr + 1
  max.stack.ptr := stack.ptr
  WHILE stack.ptr > 0
    SEQ
      {{{  get job from stack
      stack.ptr := stack.ptr - 1
      front := fronts[stack.ptr]
      end := ends[stack.ptr]
      }}}
      lt.rng := end - front
      IF
        lt.rng >= THRESHOLD
          SEQ
            ...  key selection
            ...  partition
            ...  push arrays >= 2 onto stack, largest first
        TRUE
          ...  insert sort
```

5.4.2 Farmed version

Developing the farmed version from this stage is easy. The following basic structure has already been mentioned,

```
SEQ
  push (a)
  WHILE stack has some contents
    SEQ
      pop (a)
      front, end := partition (a)
      push (front)
      push (end)
```

Farming this out consists of putting the partition process into the worker. This makes the code of what will be the farmer something like,

```
SEQ
  push (a)
  WHILE stack.ptr > 0
    SEQ
      pop (a)
      to.worker ! a
      from.worker ? front; end
      push (front)
      push (end)
```

However, this code just performs a remote procedure call and is not a farmer. What is eventually needed, as was performed with the towers of Hanoi, is for this process to be able to give out more than one job

at a time. As before this involves removing from the farmer any sense of a current context. In a sense this has been done already here as the `first` and `last` values read back in from the farm. After some thought it was realised this could be performed by having a farmer that used exactly the same interface and set of interactions to the farm, as was used with Hanoi. Something along the lines of,

```
SEQ
  push (front, end)
  WHILE stack.ptr > 0
    PRI ALT
      req ? work
        SEQ
          pop (front, end)
          to.worker ! front, end
      from.worker ? front; j; i; end
        SEQ
          push (front, j)
          push (i, end)
```

Quicksort is less of a problem than Hanoi as if a partition is in the farm, there are no smaller sub-partitions of it that might also be in the farm at the same time.

As there will be a number of jobs in the farm at any one time, when a result arrives at the farmer, it must be known where that sorted partition fits within the whole of the array. What needs to be done here is to store the offsets of the partitions from the front of the array. It was decided to do this by having an array of these offsets in the farmer. This array then stores the offset of the partitions that are in the farm.

The practicalities of this were implemented in the following way. There were two parallel arrays, the first contained the offsets as described above and the second contained booleans that marked which locations in the first array were in use. When the farmer went to give out a job, it searched for an index in the array that was free. The partition's offset was then stored in this location.

This index was then put into the job. The worker would leave this value unaltered and return it in the result. The farmer would then use this index to look up the offset of the result partition. This technique could also be used to store more information about the jobs in the farm.

Protocols

The farm could now be implemented. The following protocols were developed.

```
PROTOCOL JOB IS BOOL; INT; INT::[]INT:
```

The job protocol contains the sort type, the job number and the data itself. The sort type is whether the worker is to partition or insert sort the data. This was implemented using the following constants,

```
VAL partition IS TRUE:
VAL insertion IS FALSE:
```

The job number is the array index of this job's partition offset in the farmer's partition offset array just discussed. The worker leaves this value unchanged and passes it back to the farmer. Thus the details of the result protocol are similar,

```
PROTOCOL PACKET IS BOOL; INT; INT; INT::[]INT:
```

This contains of the sort type, the job number, the partition offset and the data. The sort type is a record of what algorithm was used to sort the data. If the job has been partitioned further work is required. If insert sort has been used then this part of the array is in order and no further work needs to be performed. Next is the job number for the farmer's own reference. The partition offset is how far into this message is the start of the greater than and equal to partition if this job has been partitioned.

Farmer

The complete top level of the farmer had the following declarations,

```
VAL stack.size IS 100:
VAL list.size IS 50:
TIMER clock:
INT start, stop:
INT max.stack.ptr, max.index.used, max.jobs.in.farm:
[max.size]INT data:
[stack.size]INT fronts, lens:    -- partition stack
INT stack.ptr:
[list.size]BOOL index.used:  -- which slots are used
[list.size]INT job.front:
INT jobs.in.farm:
BOOL job.type:
INT index, len, lt.len, ge.len:
INT front, end:
BOOL any:
```

and the following code,

```
SEQ
  {{{  init
  stack.ptr := 0
  jobs.in.farm, max.jobs.in.farm := 0, 0
  max.index.used := 0
  SEQ i = 0 FOR SIZE index.used
    index.used[i] := FALSE
  }}}
  ...  init array with some values
  ...  place first job on stack
  clock ? start
  WHILE (stack.ptr > 0) OR (jobs.in.farm > 0)
    PRI ALT
      ...  get in results
      ...  give out work
  clock ? stop
```

As before priority has been given to the receiving of results over the giving out of more work.

How work was given out will be discussed first here. The storing of offsets was performed as simply as described above. Here a sequential search through the list of indexes was performed until a free slot in the array was found. The array was declared to be sufficiently large to comfortably cope with the sizes of farm used.

```
{{{  give out work
stack.ptr > 0 & reqs ? any
  SEQ
    {{{  get job from top of stack
    stack.ptr := stack.ptr - 1
    front := fronts[stack.ptr]
    len := lens[stack.ptr]
    }}}
    end := front + (len - 1)
    {{{  get a job index number
    index := 0
    WHILE index.used[index]
      index := index + 1
    }}}
    index.used[index] := TRUE
    job.front[index] := front
    ...  update max.index.used
    IF
      len >= THRESHOLD
        jobs ! partition; index; len::[data FROM front FOR len]
      TRUE
        jobs ! insertion; index; len::[data FROM front FOR len]
    jobs.in.farm := jobs.in.farm + 1
    ...  update.max.jobs.in.farm
}}}
```

Getting the results back in is reasonably easy, though subtle. The process consists of loading the data back into the array from where it came. This is achieved through using the index value that we gave out with the job. If the data was sorted using insert sort, the data is in order and so nothing further is necessary.

```
{{{  get in results
results ? sort.type; index; lt.len;
          len::[data FROM job.front[index] FOR len]
  SEQ
    IF
      sort.type = partition
        SEQ
          ge.len := len - lt.len
          ...  push arrays >= 2 onto stack, largest first
          ...  update max.stack.ptr if necessary
      TRUE
        SKIP
    index.used[index] := FALSE
    jobs.in.farm := jobs.in.farm - 1
}}}
```

Worker

In [DH90] the workers have a number of different worker processes, each of which performs a different type of job. Which worker a job message is sent to is dealt with by the farming harness.

As has already been discussed in this work, decision making in the farming harness prevents it from getting on with the task of obtaining further work and could lead to inefficiency. The approach used here is to let the worker analyse the job and decide which procedure should be called. In this farm the workers can perform two types of job: a partition and an insertion sort. Here the worker receives a job and processes it directly, according to its type.

```
{{{  declare variables
[max.size]INT data:
INT job.number:
BOOL job.type:
INT middle:
INT len, end, i, j:
INT key:
INT element:
}}}

SEQ
  request.work ! TRUE
  receive.work ? sort.type; job.number; key; len::data
  end := len - 1
  IF
    sort.type = partition
      ...  key := data[median(front, middle, end)]
      ...  partition Tony Hoare CACM 1961 v4 n9 p321 two way split
    TRUE
      ...  insert sort
  result ! sort.type; job.number; i: len::data
```

The insert sort algorithm now runs `FROM 0 FOR end`, not `FROM front FOR end` as was written in the sequential version previously. The partition algorithm also needs similar modification from the form written in the sequential version. The variables `i` and `j` are initialised differently.

```
{{{  partition Tony Hoare CACM 1961 v4 n9 p321 two way partition
SEQ
  i, j := 0, end
  ...  up: LOOP START
  ...  check lower partition
  ...  check upper partition
  IF
    i < j
      ...  swap and goto up
    ...  COMMENT move key
    TRUE
      SKIP
  i := j + 1
```

This algorithm generates three partitions, as has been discussed, the values between `i` and `j` being equal to the key. This implementation only partitions the array into two sub-partitions. This last line sets `i` to point to the beginning of this middle partition. This is valid here as the key element is contained within the partition.

The farm was arranged as a line of workers, with the first worker placed on the farmer's transputer.

5.4.3 Tests

The programs were then run to sort 10,000 numbers. These numbers were picked using the following simple algorithm,

```
{{{ init array with some values
SEQ i = 0 FOR 10,000
  data[i] := (314 * i) \ 19950
}}}
```

As before program run was performed five times and an average taken.

5.4.4 Results

For the sequential program, the optimum value of THRESHOLD was found to be 15. For the farm larger values of THRESHOLD resulted in quicker run times. The partition algorithm is quick to execute and 15 is a small number of values to partition. As the overhead of communicating a job is large due to the packet's header size this farm will easily become communication bound. As a result of this the farm was run with THRESHOLD equal to multiples of 15. Farms were run for sizes of 1 through to 6. The run times are shown in table 9.

workers	run time (THRESHOLD = 15)	farm run time (THRESHOLD)
1	1.0805	
1	1.2678	1.1371 (30)
2	0.8910	0.7352 (45)
3	0.8456	0.6094 (75)
4	0.8299	0.5531 (90)
5	0.8211	0.5238 (90)
6	0.8250	0.5147 (105)
7	0.8217	0.5100 (135)
8	0.8286	0.5184 (135)

Table 9: Quicksort run times

Below the sequential program is compared against the farms with THRESHOLD equal both 15 and whatever multiple of 15 obtained the greatest speed up. These values are shown in brackets along the x axis of the graphs.

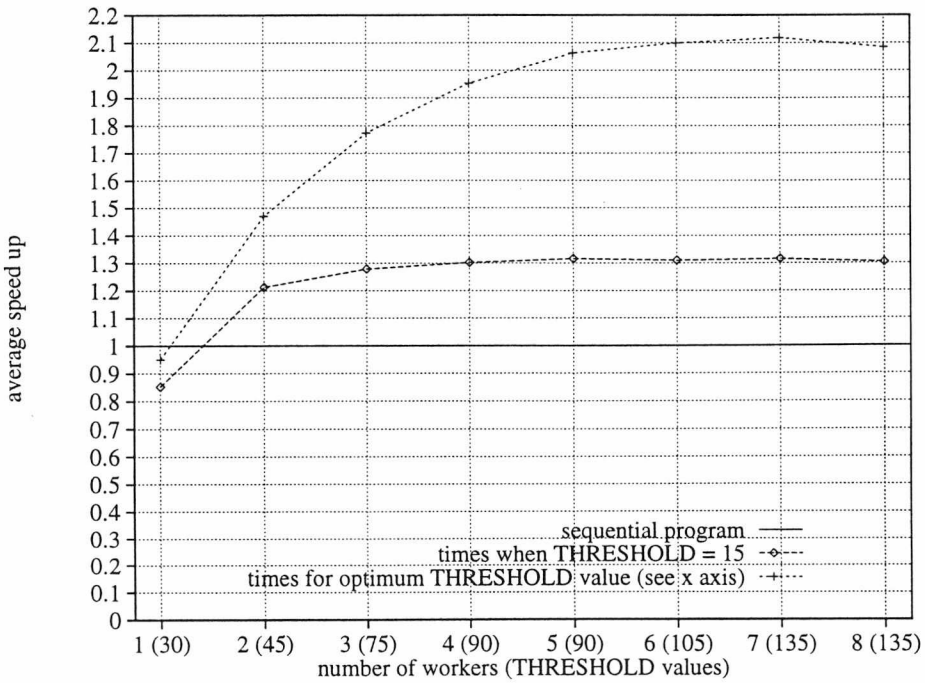


Figure 55: Speed up of quicksort programs

As can be seen from the reduced run times, farming out quicksort in this way does result in some speed up, see figure 55. On completely equal terms a speed up of 1.31 is obtained. This was on 5 workers giving an efficiency of 26%. By tuning the algorithm selection threshold value to its optimum for a farm, a speed up of 2.12 is obtained. This was with 7 workers, resulting in an efficiency of 30%.

A speed up was only obtained for farms sizes greater than 1 worker here. A single worker farm is slower due to the overheads a farm, as well as the monitoring mechanism implemented here. Also note that 15 was not an optimal value of THRESHOLD for any of the sizes of farm.

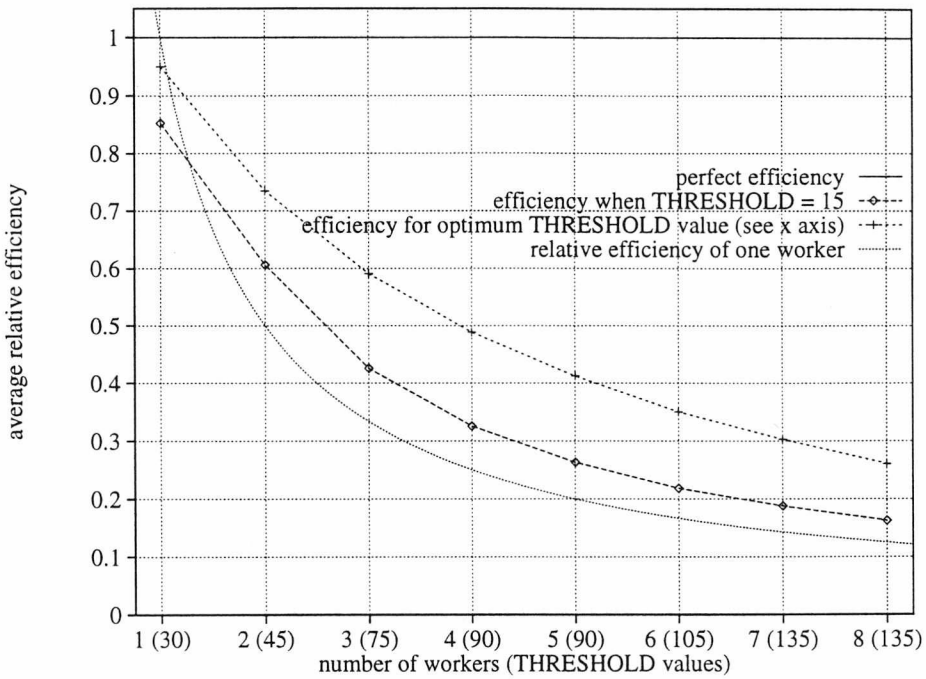


Figure 56: Relative efficiencies of quicksort programs

From looking at the relative efficiencies, figure 56, it can be seen that the usage of the transputers tails off very rapidly. As more workers are added they perform very little extra work. Also, the more workers in the farm, the more inefficient the program is and for longer at the beginning of the run. Although more work was always performed by the farm than would be by just one processor.

In summary, although some speed up is achieved by using this strategy of mapping quicksort onto a farm, the speed up and the efficiency this implementation obtains was only small.

The reason for only obtaining a small amount of speed up when farming out partitions was due to the farm just having one very large job perform at the start of the run. Thus initially the most of the workers are starved of work, and for a considerable length of time. If a farm is to be properly utilised, at any one time there needs to be at least as many jobs being supplied to farm as there are workers. This was not the case here and thus the full potential of a farm can not be realised. Thus the performance here is mediocre compared with say the efficiencies obtained in chapter 4.

Also, in [MS87] McBurney and Sleep implement a matrix multiplication on a architecture for any application that can be expressed using recursive divide-and-conquer. On 1 processor their system obtains an efficiency of 94% compared with a tight sequential loop performing the same problem. For quicksort a 1 processor farm obtains an efficiency of 85% (or 95% once tuned). On 12 transputers the matrix multiplication obtains an efficiency of 73%. This is quite good and is clearly quite a bit better than the 30% efficiency obtained here on 7 transputers farm. The quicksort farm loses such a large degree of efficiency due to insufficient parallelism in the mapping used. It is also worth noting that McBurney and Sleep's architecture also loses some efficiency due to only communicating jobs over at most one link from their origin. Thus this architecture is not very efficient for problem sizes not significantly larger than the size of the transputer domain used. This quicksort farm is a good example that an appropriate mapping is required for an efficient implementation. McBurney and Sleep's ZAPP architecture is a good example that the correct communication strategy is important if an implementation is to be efficient.

That this farmed implementation of quicksort obtains some degree of speed up when implemented on a farm is satisfying. Nevertheless, the fact this level of performance is poor was expected, due to the

initial insufficient amount of parallelism in this mapping. This was why a second more suitable mapping was attempted.

Maximum behaviour

What is also of interest here is the stack behaviour of the two quicksort programs. The sequential program had a maximum stack size of 8. After running the farm program it was found the `max.jobs.in.farm` and `max.index.used` values were the same for all runs of the program. Thus the values obtained here are only listed under the name `max.jobs.in.farm` in table 10.

workers	max.jobs.in.farm	max.stack.size (15)	max.stack.ptr (optimum)
1	2	16	15
2	5	33	21
3	8	27	20
4	11	30	18
5	14	35	17
6	17	29	18
7	20	22	20
8	23	51	12

Table 10: Maximum sizes in quicksort farms

The `max.stack.size` was larger for farms, growing in size with the size of the farm. The most interesting result here is that the maximum size of the stack is reasonably constant for optimised farms, regardless of the number of workers in the farm.

5.4.5 Farming out segments

Suggested here is an approach in which the task decomposing the work into jobs that are of effectively all the same size throughout the running of the farm.

As before, this mapping revolves around a stack of partitions. Here however, these are not just farmed out as is, but as a series of jobs. A partition is removed from the stack and is then stored in a variable (or variables) until all of it has been farmed out. These variables should not be used by the code that reads in results. Every time the farmer goes to give out a job, it sees if it *has a partition*, getting a new partition from the stack if not. Just a segment of this current partition is then farmed out. The `lt` and `ge` results of these segments are then grown from the ends of the partition towards each other. These meet when all of the results of the partition have been received.

The top level of the farmer is,

```
WHILE (stack.ptr > 0) OR have.a.partition OR (jobs.in.farm > 0)
  PRI ALT
    ...  get in results
    ...  give out segment jobs
```

Segment jobs should be given out as follows,

```
(stack.ptr > 0) OR have.a.partition & req ? any
  SEQ
    IF
      NOT have.a.partition
        SEQ
          ...  get a partition and prepare to farm it
      TRUE
        SKIP
    IF
      partitioning  -- if partitioning this job
        IF
          length > SEGMENT  -- give out next segment
            SEQ
              jobs ! partition; job.index; key;
                    SEGMENT::[data FROM front FOR SEGMENT]
              front := front + SEGMENT
              length := length - SEGMENT
          TRUE
            SEQ   -- give out last segment
              jobs ! partition; job.index; key;
                    length::[data FROM front FOR length]
              have.a.partition := FALSE
      TRUE
        SEQ  -- if insertion sorting this job
          jobs ! insertion; job.index; 0;
                    length::[data FROM front FOR length]
          have.a.partition := FALSE
    jobs.in.farm := jobs.in.farm + 1
```

The fold `... get a partition and prepare to farm it` receives a job from the stack, and then similar to before, the farmer remembers where the ends of the partition are and the positions where the result segments are to be inserted. The key also needs to be selected at this stage.

```
SEQ
  stack.ptr := stack.ptr - 1
  front, length := fronts[stack.ptr], lengths[stack.ptr]
  end :- front + (length - 1)
  ...  get a job number
  partition.front[job.index] := front
  partition.len[job.index] := len
  IF
    length > THRESHOLD
      SEQ
        insert[job.index][lt] := front
        insert[job.index][ge] := end + 1
    TRUE
      SKIP
  have.a.partition := TRUE
  ...  select key
```

The `insert` array contains for each job the address the next `lt` result segment should be loaded into. The `ge` result segment value is set to be `end + 1`, just beyond the end of the array. The length of the segment should be subtracted from this value to give the address the `ge` result should be loaded into. If the result message has been loaded into `buffer`, this would be done as follows,

```
IF
  partition.type = lt
    SEQ
      [tmp FROM insert[result.index][lt] FOR len] :=
         [buffer FROM 0 FOR len]
      insert[result.index][lt] := insert[result.index][lt] + len
  TRUE  -- partition.type = ge
    SEQ
      insert[result.index][ge] := insert[result.index][ge] - len
      [tmp FROM insert[result.index][ge] FOR len] :=
         [buffer FROM 0 FOR len]
```

All the partition has been received when,

```
insert[result.index][lt] = insert[result.index][ge]
```

the contents of the `tmp` array can be copied back into the main `data` array and the two partitions can be placed on the stack for further partitioning.

As before the workers are given an array and partition this into two. As these are loaded separately into the farmer, these two partitions should be communicated in two separate result messages.

As the work is being segmented here, the workers may get a segment than contains values all of which are less than the key or greater than the key. In these cases only one result message should be generated.

5.4.6 Recursion

Recursion isn't present in occam, as was mentioned in the section on the towers of Hanoi. It is also not present in UNITY either.

Many programmers, on their first contact with occam, complain about this and often suggest it should be added to future versions of the language.

Further, a common opinion expressed by some members of the transputer community (in the Usenet newsgroup `comp.sys.transputer`) a few years ago is that recursion would be very difficult to implement by hand. The experience obtained here however, is completely the opposite. Developing the stack, and the operations that manipulate it, was one of the easiest parts of both the Hanoi and the quicksort programs. Further, translating the quicksort algorithm from another very early programming language was also found to be quite easy. This conclusion has also been arrived at by other work [MMF87].

This research has come to the conclusion that, if anything, recursion does not enhance our thinking, but restricts it. As programmers we are interested in developing parallel implementations of applications.

When setting out to develop a towers of Hanoi farm, no way to parallelise the application could be seen when it was expressed in recursive form. As soon as the application was thought of in terms of jobs on a stack, the way to proceed was obvious. Thus, it is the belief of this work that recursion is not wanted in parallel languages: as when trying to implement a recursive algorithm in parallel, recursion restricts our thinking and makes the parallelism in the application harder to find, not easier. Thus recursion is a form of expression not easily amenable to parallelisation. Thus making recursion of little use when implementing algorithms for parallel machines.

Language shapes our thoughts and recursion shapes our thoughts in a way that is not appropriate for parallel programming. Although it is a useful shorthand for expressing sequential algorithms, it is not useful for expressing algorithms in a way that is appropriate when wishing to find a way of parallelising a sequential algorithm. Further, when implementing a recursive application on a farm, the execution of the recursion will always be implemented by hand. Thus, the problem will be implemented completely differently and so recursion is not useful for expressing the structure of the algorithm.

5.4.7 Conclusions

The main conclusion to draw from this, and the previous section, is that recursive applications can be farmed out.

In this section it has been shown that if a suitable mapping strategy can be found recursive applications can be farmed out efficiently and speed up can be obtained.

That the same basic farmer could be used for both programs could suggest that all applications that can be farmed out in stages should use this type of farmer.

5.5 Some other applications and a model of farming

Having developed an efficient farm of a recursive application, focus returned to what range of applications could be farmed out.

The work here took place in a number of forms. This started by looking at how two applications would be farmed out. Next two ways in which the jobs may have some form of dependency between them was looked at. Lastly it was decided to look at the mechanics of the processor farm itself and see what its limitations were. From this a model of farming and what types of program it can execute was developed.

5.5.1 Farming out functional applications

It had been suggested that the evaluation of functional programs could be farmed out. This being achieved by giving out the various parts of the functional graph. It was realised here that if this were possible, the job messages's would contain the processes to be executed by the workers.

This contrasts with all the processor farms seen here so far where the job messages contained just data. This approach is easy to implement, and further, is all that is possible in fully checked occam.

This is why farming out and geometric decompositions have seemed similar, both involve giving different items of data to be computed to different processors executing the same process.

However, now it was realised workers may execute the contents of the job messages they receive. In fact, this is possible even if functional programs can not be farmed out; nothing about the mechanics of a processor farm prevents messages containing executable objects or the workers executing the contents of job messages they receive. Thus, processes themselves may also be farmed out. As before messages are simply communicated from the farmer to the workers as jobs of work.

Although job (and result) messages may contain instruction sequences of the processors' assembly language, this is not possible in (pure) occam; there is no executable object type so only jobs of data can be fully checked by the compiler.

It is important to note that all jobs must still terminate. This includes executable jobs.

In summary, job messages may be of any type: data, the executable code of any process that terminates, or the two combined.

5.5.2 Farming out of sequential applications

Considered next was whether a sequential application could be farmed out. The application considered was the classic program that printed "Hello world".

This application consists of outputting a sequence of bytes, and thus is just a sequential application as the task can not be decomposed into a number of separate parallel jobs or processes. This is a feature of the application, and is not affected by any approach taken to implement it.

It was further realised that this does not prevent the application from being implemented on a farm, it is just there is no parallelism inherent within the application to make use of the parallelism of a farm, or any other parallel implementation technique. There being only one thing to be performed at any one time. Naturally such a program is non-scalable. Thus in general, an implementation is dependent upon both the amount of parallelism within the application and the way that parallelism is decomposed. Also, with sequential applications it is important that jobs are executed in order. One simple way to guarantee this is by having a one worker farm.

It was realised that any sequential application could be farmed out by the program's machine instructions being given out one at a time. This also showing again that job messages may contain processes as well as data.

Farming out sequential programs would be less efficient than executing the program in the conventional way on a single processor. This being due to the overheads of setting up and running a processor farm. However, the important result here is that farming out sequential programs is logically possible. In fact, it was realised the structures of a one worker farm and a Von-Neuman machine are identical, as can be seen in figure 57.

Figure 57: Structure of a one worker farm and a Von-Neuman machine

Similarly, it is also the case that shared memory machines (see figure 58) and multi-worker farms (figure 59, page 158) are also identical in structure.

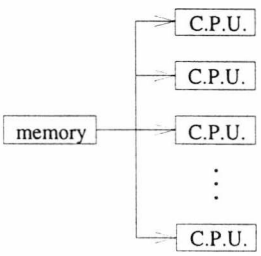


Figure 58: General structure of multiprocessor shared memory architectures

Such parallel architectures are ideal for farming out a queue of processes (or 'threads'). In fact, a number of machines have started to appear on the market that farm out the process queue although they provide a "conventional" programming interface to the users. Thus, there is a greatly increased potential for the use of parallel programs in the future. This being due to both of the main advantages of parallel programming: scalable linear speed up and the naturalness achieved through describing the behaviour of naturally parallel systems in a parallel language.

The fact we can farm out sequential applications is not very useful; generally we are interested in farming out applications containing parallelism on a number of workers in an efficient manner. This being with scalability and linear speed up. This involves utilising efficiently (and thus effectively) the power of hardware contained in a multiple number of processors.

Thus in short, we need to concentrate on and be interested in parallelism. This being both parallelism within the implementation as well as within the application. As we have see with the "Hello world" application, in order to have parallelism within the implementation there must be parallelism within the application. Thus we can only gain better performance through parallelism if applications have a reasonable degree of parallelism within them.

In order for an implementation to be efficient, the parallel decomposition should be of a sensible grain for the hardware. Experience from chapter 4 tells us a sensible grain for farming is slightly bigger than that of the grain of the hardware's primitive operations, for example see subsection 4.7.6 (page 63).

Thus this study of farming out a sequential program has brought out an important distinction. This is between whether it is logically possible for a program to be farmed out, and whether an efficient implementation can be developed. These issues are completely separate: the issue of efficiency is dependent upon both the mapping and and the architecture used, the issue of possibility is dependent upon the nature of the application.

For any implementation developed we should ask the question: is the program not only efficient, but more efficient than would be obtained through using another execution strategy?

From looking at the farming out of sequential applications, three things have been learnt,

1. the amount of parallelism is limited to the application,
2. if it is possible to farm out an application, this does not guarantee that an efficient implementation will be found, and,
3. a processor farm has the same structure as both Von-Neuman machine and shared memory machines.

5.5.3 Farming out processes with communication dependencies between them

It was realised that processes with communication dependencies between them can be farmed out. How this can be done is demonstrated with the following example,

```
CHAN OF Type chan:
PAR
  SEQ
    ... a
    chan ! intermediate-result
    ... b
  SEQ
    ... C
    chan ? parameter
    ... D
```

These two processes can be farmed out if they are divided into processes that contain just the independent computation, processes `a`, `b`, `C` and `D`, and the communication dependency. The independent processes can then be farmed out as jobs and the dependency can be performed via the farmer; this being done using the existing result and job messages. The intermediate result produced by job `a` can be communicated, as a part of the job's result, to the farmer. This intermediate result can be supplied as a part of the job when process `D` is farmed out.

This is equivalent to transforming the above processes into,

```
CHAN OF Type result, job:
PAR
  SEQ
    result ? value
    job ! value
  SEQ
    ... a
    result ! intermediate-result
  SEQ
    job ? parameter
    ... D
```

This performing of work in a number of stages is the same thing as was talked about in subsection 5.2.1 (page 120).

5.5.4 Processes with loose computational dependencies between them

Work with a loose amount of computational dependency can also be farmed out. In such situations there is a (possibly cyclic) graph of interdependencies between the items of work, each edge representing where a process needs results from another process. If at any one time a number of processes can proceed simultaneously, these processes could be farmed out. When the results of these jobs return they will contain values required for the next set of jobs.

5.5.5 The phrase "farm out"

In section 2.5 (page 15) it was decided to use words like farming and farmer, not master and slave etc. More recently it was discovered that the verb farming has been in the language for some time.

Dictionaries have a few definitions for the verb *to farm*. One, which is frequently followed by *out*, is defined along the lines of,

> to delegate, subcontract, send or give (work) to (be done by) others.

This is what is performed in a processor farm. This usage dates from the mid 17th Century [Uni93].

Further, [Uni33] dates to 1666 a definition with similar meaning,

> contract to maintain and care for (persons, esp. children)

This phrase is also still in use in society today, "children are farmed out to neighbours or relatives".

Using the verb farming out refers to what is going on directly and thus shifts the emphasis away from an object and towards the action. As it is this action and mechanism that is more important, we use this term and view farming out as an action more than than a physical entity. Thus, where possible, the verb "farm" or "farm out" is used in the rest of this thesis opposed to the nouns "farm" and "processor farm".

5.5.6 A model of farming out: what it is and what it can do

Here we present a model of the mechanics of farming out. Using this model we go to discuss what jobs and what applications it can perform in the next two subsections.

Farming out is an execution strategy, one way of arranging for work to be performed, the allocation of the jobs being performed at run-time.

To aid in the definition of a model for farming out, three items need to be introduced. These are,

- a job, an item of computational work,
- a bag of jobs (or a central pool of work), and,
- a number of workers, each of which is capable of performing work.

In farming out, a bag and a number of workers are arranged as in figure 59 below.

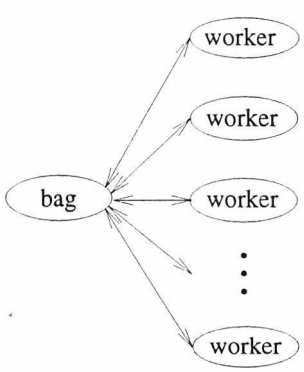


Figure 59: A model of farming out

Farming out consists of two actions,

1. the allocating of a job (via some form of interaction) between the bag and a (free) worker that does not have a job, and,
2. the performing of those jobs by the workers.

The interaction may consist of transferring something like a request and a message containing either a job number to be interpreted by the worker, the computation to be performed, or the data to be processed by the worker which already has the algorithm to be used.

Thus farming out consists of the continual scheduling and performing of work. The next job to be performed is allocated from the bag to a worker as soon as that worker becomes free, or until any buffer process responsible for that worker becomes empty. This worker executes the job and again is issued with another as soon as it becomes free once more.

Each job may produce any number of results and further jobs. Any jobs generated are returned to the bag. This keeps the mechanics of job distribution simple: work always travels away from one place (the bag) to where it can be worked on (the workers). Results should be dealt with as appropriate, invariably they are collated. An implementation may require a process or a number of processes to perform the collating of results. This action can either be performed by the process responsible for the bag of jobs (the farmer process) or a separate process (a harvester process).

As a method of organising work, farming has a number of advantages,

1. the work load is automatically balanced,
2. the overheads are reasonably light, and,
3. the performance can be scaled by just adding or removing workers (this being done with practically linearly scalable performance).

These last two points depend on there being more jobs than workers, this amount of parallelism being constant throughout the running of the farm, and the amount of supply being greater than the amount of demand.

The above model represents the basic structure of farming out. An implementation may extend upon this. For example, the model doesn't mention a farmer process to manage the bag, however many architectures will require such a process. Also, the model doesn't mention that results are generated and collected. In practice they are and any number of harvester processors may be used for a particular application, if appropriate. Similar the sorting out of job interdependencies can either by done directly by the processor maintaining the bag or by other processors, see figure 60.

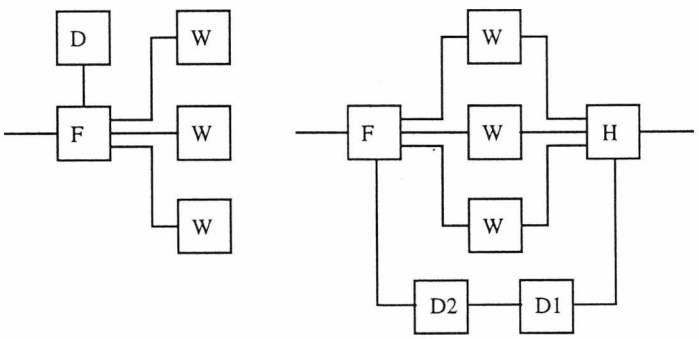


Figure 60: Two ways of dealing with dependencies between jobs on a processor farm

At the beginning of this chapter a number of extensions to the mechanics of farming out were discussed. Really, there are just three extensions made here to what was generally understood to be farming out before this research was performed. These three extensions are that,

1. workers may perform more than one type of job,
2. workers can generate jobs as well as results, and,
3. any dependencies their may be between jobs can be dealt with by the farmer.

5.5.7 What type of jobs can be farmed out

Conventional understanding tells us that, in farming, the jobs being farmed out must be independent of one another. Though, as we have become aware of here, it is possible to farm out a set of jobs that have dependencies between them. These dependencies being sorted out via communication.

Whether jobs can contain communications as well as communications depends on the mechanics of the architecture the farm is being implemented on.

Jobs could contain communications if they could be descheduled and returned to the bag ready to be continued, until both processes were ready to communicate. Such execution mechanics could be either a part of the architecture, or constructed. Constructing such mechanics however, might pollute farming's simple scheduling mechanism, introducing overheads and thus reducing the efficiency at which all applications would be executed.

As this is not always possible, the alternative is for jobs to be decomposed into their independent parts and communications. The former can proceed independently of one another and the dependencies can be communicated as a part of the results and jobs, as was looked in subsection 5.5.3 (page 157).

From a programming point of view, when decomposing applications for farming, the extreme of having an application fragmented into too high a granularity should also be avoided.

5.5.8 What applications can be farmed out

Applications consist of two main building blocks. Here we look at how and where these can be executed on a processor farm. From this it was been realised that all programs can be farmed.

Jobs can be farmed out in occam to transputers and among people in the real world. These two circumstances share a common model of parallelism; the jobs are independent of one another, i.e. their internal workings are private.

The tasks that are to be performed within this way of having things are made up of processes that either just perform computations or that just perform communication. Such a computation consists primarily of expression evaluation and assignment, it also consists of selection and other constructs, such as looping. Communication consists of parallel processes exchanging items of information. These two can be combined to form parallel computations.

A computation contains no communication. Thus, it is independent of (or irrelevant to) any other processes being executed at the same time. This is due to the internals of objects being private, thus processes are independent and can not interfere with one another (in occam they have only local variables). Thus, computations can be executed by workers working completely independently of one another.

A communication is an exchange of information between two processes. Communication can be executed on a farm in one of at least two ways. While being executed by workers, jobs can communicate either directly (as discussed in subsection 5.5.7, page 160), or via the farmer (subsection 5.5.3, page 157). Thus, communication can be executed on a processor farm, the exact details of how depend on: what is possible with the architecture and what is efficient.

In fact, with the first generation of transputer, the farmer would be involved in all forms of communication. For example, any ALTs in the program and any communications with the user.

So, all tasks that are expressed in this real world model consist of a collection of independent computations, and communications. As both of these can be farmed out, all tasks expressed in this way can be farmed out.

At present farming can perform a number independent computations in parallel and deal with only one process interdependency, future architectures may also be able to perform a number of communications to be performed in parallel too.

So, all tasks that can be expressed as independent computations can be farmed out. We have not looked at any other models here. Nevertheless, it is likely some of these other models may allow for programs to be transformed into this form. We have also not looked to see if farming can be performed directly within any of these other models of task expression.

Whether an implementation will be efficient or not is still constrained by two factors.

Firstly, parallel slackness is needed, i.e. there are more parallel processes available for execution than there are workers in the farm.

Secondly, (as discussed in chapter 4) the mapping onto the architecture should be compute bound, i.e. there needs to be a greater supply of work than there is demand for it.

As communication is not performed in parallel, but only one communication done at a time, this approach is not efficient. This only affects efficiency, it is still logically possible to execute communication on a processor farm.

We conclude by saying that all tasks expressed in this real world model of independent computations can be farmed out.

5.5.9 How farming out differs from algorithmic and geometric decompositions

A difference has been noticed between farming out and the other two execution strategies: geometric and algorithmic distributions.

With farming out, where a particular item of work is performed is decided at run time. The decision being dependent upon what processors are free at the time the job is sent into the processor farm. Which processors are free is dependent on how long the previous jobs took to perform. This method of allocation of work contrasts with the approach used by the other two execution strategies. With those, where a particular item of work is performed is decided by the programmer when the mapping is designed, according to the internal structure of program's parallelism. Thus, farming out performs the allocation of work dynamically instead of performing it in a static and predetermined way.

This is why farming out can be potentially more efficient than algorithmic and geometric decompositions, the decisions are performed dynamically at run time instead of statically at design time. Thus, farming out is an execution strategy that is flexible and load balances automatically, instead of an execution strategy that is fixed and rigid in nature. However, although farming out may provide greater efficiency through flexibility, these allocating or scheduling decisions should only be left until run time if the cost of making them can be made to be low.

This suggests applications should be developed by taking advantage of the parallelism within the application's data or instructions and then this set of processes is executed. This being done with the aid of an execution strategy. How the items of work (processes) are allocated to the different processors may be either predetermined by the programmer or performed automatically by the order of interactions of the workers with the farming harness. Farming out's use of dynamic allocation involves some overheads, but copes automatically with different work items taking different lengths of time to be performed.

Dynamic process allocation is an execution strategy not mentioned in Welch's course and thus it was not mentioned in subsection 2.1.1 (pages 4–5). This method of execution continually reallocates processes from processors with a heavy work load to adjacent processors with a lighter work load. This involves implementing a software kernel that is run on each processor. These kernels attempt to ensure that all processors have an equal amount of work to perform by having a near equal number of processes on their process queues. This continual reviewing by the kernels of how much work each processor has incurs some overhead. Also if there is an imbalance, processes and their workspace will be communicated across links and reinstated on other processors. This continual migration has very large overheads. As farming out is more likely to have lighter overheads, it is likely to be much more efficient.

5.5.10 Summary

Farming out is a method by which work can be allocated (scheduled) and performed by workers. The allocation of work is from a bag of jobs. This allocation is performed in real-time. The contents of the job messages farmed out may be data, processes that terminate or both. These jobs are performed by workers. The number of workers can be scaled independently of the application and its internal structure.

The most important conclusion here is that it is possible to farm out all applications that can be expressed as a collection of independent computations. This applies to a collection of actions in the reality and occam programs. Any information or data communicated between processes can be dealt by communication via the bag of work.

An implementation's degree of scalability is dependent on both the total amount of parallelism in the application and the design.

Whether a farmed implementation is efficient or not is dependent upon three things: how much parallelism there is in the application, how this has been designed and mapped onto the hardware and the details of the architecture.

We can only gain better performance through parallelism if applications have a reasonable degree of parallelism within them, can be decomposed into a reasonably continuous supply jobs, and each job is as independent of the others as possible.

5.5.11 Rest of chapter

Having reached the goal of knowing which programs are farmable, the rest of this chapter looks at some examples of interest that illustrate some of the possibilities.

Next are some pointers for how to farm out applications that contain geometric data sets, and thus are usually implemented using geometric distributions.

Following this, some work involving farming performed by others is studied. This is looked at in two sections: the first looks at some examples by others of farming out on various parallel computing architectures, the other is of some examples taken from other aspects of society.

Before the conclusions some issues relating to how applications should be decomposed, allocated and executed are considered.

5.6 Implementing geometric data sets

It has been realised that applications containing a geometrically shaped set of data, as well as being implemented using a geometric distribution, can also be farmed out. This section looks at this type of application, how these have usually been implemented and how they can be implemented on a processor farm.

A typical example of this type of application consists of repeatedly performing a computation over a 2D array of data. The new value for each element being derived from the old value and its nearest neighbours.

```
data[i][j] := f (data[i-1][j-1], data[i-1][ j ], data[i-1][j+1],
                 data[ i ][j-1], data[ i ][ j ], data[ i ][j+1],
                 data[i+1][j-1], data[i+1][ j ], data[i+1][j+1])
```

In one such application, due to Morse, two 2D arrays represent the populations, at the various points across an area of land, of two species of insect, one of which is a host, the other is a parasite. The program computes the on-going size of the two populations [Mor93].

5.6.1 Geometric implementation

This application has been parallelised by mapping the array onto a line of processors. The array is divided up into segments, each segment being the full width of the array in one dimension, see figure 61. Each processor is then allocated one data segment. All the processors are then responsible for computing

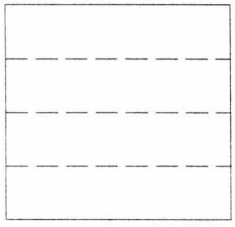


Figure 61: A 2D array divided up into full width segments

the new values for their part of the array. Nearest neighbour values are communicated with adjacent processors. One pair of links on each transputer is required for this communication. The other pair is used

to communicate the results at the end of each computational stage so they can be displayed or stored as appropriate.

Such a program is best implemented as outlined below. Here the non-edge computations are performed in parallel with the nearest neighbour communications. The boundary computations are then performed last after the nearest neighbour information has been received.

```
PLACED PAR p = 0 FOR P
  [n+2][width]TYPE data:
  WHILE TRUE
    SEQ
      PRI PAR
        ... exchange edges
        ... compute centre of dataset
      ... compute edges
      ... output view of data
```

As this code is an example, the amount of data is assumed to be a multiple of the number of processors.

It is worth pointing out one advantage of this approach to implementation is that each processor only performs two pairs of communications to exchange information with neighbouring processors. Both communications are simple, consisting of just a row of values from each processor. If this 2D array of data was mapped onto a 2D array of processors, each processor would contain a square patch of data. Here the nearest neighbour communications consist of (at least) eight pairs of communications with the eight nearest neighbours, see figure 62. This would require multiplexing, due to the transputer having four

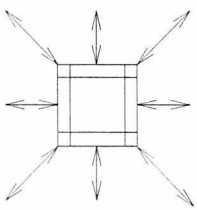


Figure 62: Number of communications that need to be performed with a 2-D geometric distribution

links. This would also involve extracting the single values down each side of the array for communication with horizontal neighbours. Doing this is not very efficient (as well as being laborious to program) as these values are not adjadcent to one another.

5.6.2 Farming out

These applications can also be farmed out. Here the application is again divided into segments as before. These are then farmed out complete with the nearest neighbour values needed to compute the result, see figure 63. Each worker then returns the result for the set of values it was given. These result messages

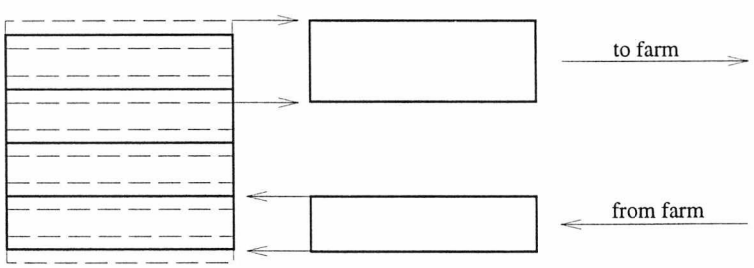


Figure 63: How nearest neighbour relationships could be farmed out

will be smaller than the job messages by two or more lines of data values. The farmer can then output these results to the user and the next set of jobs can be sent out again as before.

It would be advantageous to farm out applications containing geometric data sets if the different computations took different lengths of time to perform (for example in image processing, where the time taken to analyse each strip depends on the complexity of the image in that strip). On a geometric distribution this imbalance in workload would result in some processors lying idle waiting on communications with the other processors that had yet to finish their computations. It is believed this is unavoidable for a geometric distribution, however for a processor farm it is potentially possible to avoid this behaviour, via farming out's self-balancing nature. Next we look at some possible mapping strategies that could be used on a processor farm.

5.6.3 A simple farmer

Here is a sketch of a simple farmer. This gives out work from one array and receiving results into another. The two arrays are then exchanged and the process repeats. To keep this example simple the data wraps around at the ends. The simplest way to implement this is to duplicate the first and last rows of the data at the opposite ends of the array. Hence the `height+2` in the below declaration.

```
[2][height+2][width]TYPE data:
INT in, out:
SEQ
  in, out := 0, 1
  ...  initialise data[out]
  WHILE TRUE
    SEQ
      in IS data[in]:
      out IS data[out]:
      SEQ
        -- wrap around, duplicate first and last rows
        out[0], out[height+1] := out[height], out[1]
        PAR
          -- farm out jobs
          SEQ i = 0 FOR jobs
            to.farm ! i; [out FROM i*n FOR n+2]

          -- accept results
          INT i:
          SEQ j = 0 FOR jobs
            SEQ
              results ? i; [in FROM (i*n)+1 FOR n]
              output ! i; [in FROM (i*n)+1 FOR n]
      in, out := out, in
```

On a geometric distribution the work is divided among the processors. With a processor farm there needs to be a greater number of jobs than there are workers. Thus giving some parallel slackness. This is needed so that there is some work in both the farmer and the buffers of the harness and thus the processor farm can function at its best. The farmer needs spare jobs so it can be preparing future work (through dealing with the interdependencies between items of work) and thus there are spare jobs ready to be given out. Another advantage of parallel slackness is that the finer the granularity of the work, the quicker the processor farm will complete the work.

Unfortunately, the way this strategy works, the farmer gives out all of one set of jobs and has to wait for all the results to come back before it can start to give out any of the next set of jobs. Thus, as all of one set of jobs finishes all the workers gradually fall idle. Although a processor farm is self-balancing, as the jobs start at different times the workers are likely to finish at different times at the end of a set of work. This all results in some inefficiency. However, a highly efficient implementation may still be possible if these losses can be kept very low.

So although this is the same problem geometric distributions suffer from, with a processor farm it is worse as here each message is larger and also needs to be communicated across several links to and from the farmer. Thus potentially this processor farm could be less efficient than a geometric distribution. There are a number of solutions to this.

5.6.4 Using a finer grain

An obvious first solution is to use an even finer grain of work. This reduces the range of time over which the workers finish their last job. Hence, leading to a small improvement in performance. However, the finer the grain of work, the more communications will need to be set up and the more information will be communicated overall (due to the extra number of nearest neighbours there would be).

The size of the jobs can be reduced by either narrowing the strips made or by farming out rectangular patches. The latter should be easier to implement on a processor farm than on a geometric distribution; all interdependencies are dealt with within the farmer. On a geometric distribution nearest neighbour information needs to be communicated in eight directions which is troublesome to implement, as discussed above.

One problem with decomposing the data into patches is obtaining the data from each line of the array. One possible way of obtaining the data for such a job, is to communicate the lines of the array separately. This saves memory copies in the farmer, but results in communications taking longer (as discussed in subsection 4.6.5, page 58).

```
to.farm ! [data[i*n]   FROM j*n FOR patch.width];
          [data[i*n+1] FROM j*n FOR patch.width];
          [data[i*n+2] FROM j*n FOR patch.width];
          [data[i*n+3] FROM j*n FOR patch.width];
          [data[i*n+4] FROM j*n FOR patch.width]
```

Extracting a job from the middle of the data can also be performed by the 2 dimensional block move instruction. This is present on the T800 series of transputers. This allows rectangular blocks of data to be copied between windows of different widths, see figure 64. This instruction can be used to obtain a job

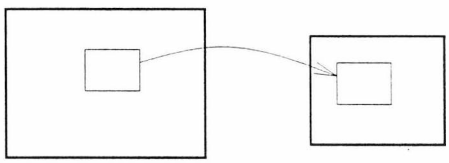


Figure 64: The general form of a 2-D block move

by copying the rectangular patch into an array the exact size of the job, see figure 65. Once this overhead

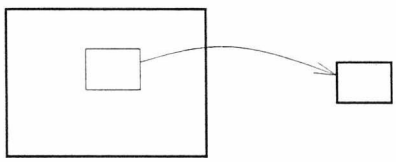


Figure 65: Extracting a rectangular patch of data using a 2-D block move

of a memory copy has been performed, this job can be farmed out by simply communicating the array,

```
to.farm ! array
```

Ultimately, using a finer grain of parallelism does not alleviate the problem, it only reduces its impact on performance. Getting the workers to perform work all of the time is a better solution.

5.6.5 Decoupling computations from the farmer's thread of control

By looking at what jobs can be given out, it can be seen that by the time the last few jobs have been given out, it is likely that in many implementations some of the first results will have been received (assuming the data has been decomposed into a number of jobs suitable for the size of the processor farm). Thus, performance could be improved through the use of a farmer that can start to give out some jobs from the next set of work before all the previous set has finished.

To do this we need to be explicitly aware of the relationships between the various jobs and get the farmer to maintain a list of what results have been received so that it can be aware of which of the next set of jobs can be farmed out. To do this the farmer would need to be aware of the relationships between the various jobs and thus must remember what results have been received so what jobs can be farmed out can be computed. In this type application the jobs have multiple interdependencies, unlike the towers of Hanoi and quicksort. These two applications have job interdependencies that were expansive: a job generated a number of other jobs and any job was only ever generated from one job. Applications with nearest neighbour interdependencies are, in a sense, the opposite to this: the next generation of jobs are generated from *a number* of results from the previous set of jobs, see figure 66.

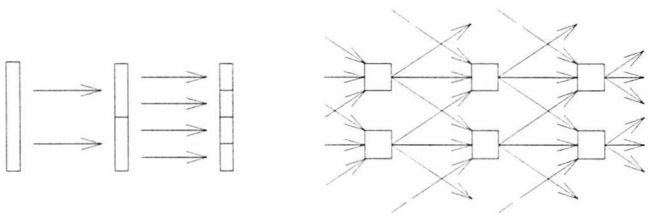


Figure 66: Job dependencies for quicksort and applications with nearest neighbour dependencies

Sorting out the interdependencies between the jobs in applications with nearest neighbour interrelations is simple; before a result can be given out as a job, the result all around it also need to have been received. By being aware of what results have been received in this way, work could be given out continuously and workers would be active throughout the execution of the program.

In order to do this the computations must be decoupled from the farmer's thread of control. What is needed here is a farmer that is centered around a `PRI ALT`, as were the farmers in the towers of Hanoi and quicksort. In those farmers the farming out of the work and the receiving of the results were not coupled to the structure of the problem. The advantage of this strategy is that the resultant farm only finishes once at the end of the program. The disadvantage of this strategy is that implementing this decoupling involves developing a method of storing which jobs have been processed and that this introduces more overheads that are likely to slow down the farmer.

5.6.6 An intermediate strategy

There may also be some other strategies that are intermediate between these two extremes. It is possible some of these could be easier to implement than a processor farm whose execution is not coupled to the structure of the problem.

It is important to remember that the first jobs to be given out are those that are the most likely to be the results first received. So, for example, the following very lightweight strategy is possible. As the results come in, a counter could be incremented for each result that has a small index number.

```
SEQ
  results ? i; data
  IF
    i  <  (N / 2)
      count := count + 1
    TRUE
      SKIP
```

When this counter gets to a certain value it is known that all of a certain fraction of the next set of jobs, 1 through to $(N/2) - 1$ in this case, can be farmed out while the rest of the results come in.

5.6.7 Discussion

If an efficient implementation can be arrived at, there remains the question of whether it would be more efficient than a geometric distribution? The disadvantages of farming here is that more messages (and larger messages) are communicated and they are communicated further.

It is hoped it is worth implementing applications on a farm if the automatic load balancing performance benefits gained are greater than the overheads of allocating location of execution at run time. That is the efficiency point of view. By farming out the work there is also the advantage of flexibility as the size of the implementation is independent of the application size.

5.7 Other applications that have been farmed out

Here we look at some other types of application that have been farmed out.

The first has some similarities to the towers of Hanoi and quicksort implementations developed here. From the design strategy used, it was thought the implementation would be inefficient, in fact it is very efficient.

The other two applications have jobs that are not related to one another, unlike the towers of Hanoi and quicksort, and also are more than just a two dimensional array of calculations, unlike the Mandelbrot set and ray-tracing. Thus these two applications may have a structure probably more akin to many other applications that have yet to be farmed out on parallel hardware.

5.7.1 Farming out the search for a minimal perfect hash function

In this work Bartoszek, Czech and Konopka had implemented an application that required jobs to be farmed out in an iterative manner [BCK93], as was performed here with the towers of Hanoi and quicksort. Of the two implementations were developed, one was developed at Kent by Bartoszek in occam. The approach used in this implementation resulted in an efficient performance. Although some of the aspects of the design were thought by this author to not be efficient. Thus this work is of interest here.

The application was to find a *minimal perfect hash function* (MPHF) for a given set of data. MPHFs are useful for efficient use of memory and fast retrieval of items from a static set, reserved words in programming languages for example. Finding a MPHF is performed by searching a graph that is viewed as a large n-ary tree. The bottom of the tree consists of hash functions, only some of these are minimal and perfect. Only the first (left most solution) need be found. Thus the search is from top to bottom and left to right.

Implementation strategy

From the current starting point each worker advances a number of vertices down the graph. The number of vertices traversed is selectable and dictated by the grain of the implementation. Once the appropriate number of edges have been traversed, the suitability of the destination vertices are reported back to the farmer. The vertex that will lead to the first minimal perfect hash function is then farmed out again and the workers go down another few levels. This continues until the first minimal perfect hash function is obtained. This again is farming out an application in a number of stages.

This application is of a similar structure to the geometric n-body problem, each calculation in the sequence is followed by a sharing of intermediate results. Normally this would be performed by a geometric distribution. However, as there is a need to perform a comparison on all the intermediate results, this is best performed by a single processor, and so farming is an appropriate way to implement this application.

Points of interest in implementation

There are three points of interest about Bartoszek's implementation.

The workers had the first job to be performed compiled into them. Thus, as soon as the program started all the workers began solving the problem immediately. This saved time as the first set of jobs did not have to be distributed.

The harness sent specific jobs to specific workers. This is an upshot of the fact that the workers maintain their state between jobs and a job may simply be to continue. This detail is of use as these jobs take some time to initialise. Thus, at least one worker would be able start its job sooner.

After giving out a set of jobs, the farmer had some internal housekeeping to perform. Thus in this implementation, the farmer was placed on a separate processor so it could proceed with this internal housekeeping unhindered. Having the farmer on the same transputer as a worker could result in the application being slowed down. This directly contrasts with the experience here of conventional farms, where having a worker on the same transputer as the farmer resulted in a faster implementation, due to that transputer also being able to assist in the computing of work.

Performance of implementation

It was thought the approach used would lead to inefficient implementations for two reasons. Directing jobs to specific workers had been attempted in Sturrock's first implementation of a farming harness [SS91, Stu91]. This lost linear speed up for large numbers of workers. It was also thought that having the workers idle while the farmer developed the next set of jobs would be inefficient. It being thought here that in an efficient implementation all processors would be busy performing constructive work almost all the time.

Bartoszek's occam farm obtains constant linear speed up for up to 9 workers. When run the program was run with a search granularity of 1, 9 workers obtained a speed up of 7.0 over the performance obtained with 1 worker, an efficiency of 77.7%. With a search granularity of 5, a speed up of 8.5 (an efficiency of 94.4%) was obtained. Thus on the whole, this implementation is very efficient and thus makes effective use of the hardware.

Thus this shows that the two prejudices above would have restricted us from developing this type of efficient implementation. That the workers are temporarily idle doesn't matter too much if the granularity of work is large enough to reduce the impact this has on performance.

One reason for this implementation being efficient is that an appropriate strategy of organising the work has been used. This is more important for efficiency, in contrast to the towers of Hanoi farm developed here where an inappropriate strategy was used and the resulting implementation was inefficient.

There are three things to say with regard to directing jobs to specific workers.

There will be small additional overheads in continually checking jobs in order to discover the directions they should be sent in, and when there are many workers the accumulation of these overheads will affect the rate of supply and thus performance. Nevertheless, this implementation is obviously efficient for the number of workers used. Bartoszek uses nine workers, this is not as many as the 64 worker used by Sturrock.

Normally in a farm jobs would only be allocated to workers that were free. However in this case all the jobs being farmed out are going to workers that are free. Also, directing specific jobs to specific workers can allow for a faster program here (one worker being allowed to simply continue where it left off).

In general, restricting harnesses to only allocating jobs in an arbitrary fashion is artificial. Jobs can be placed specifically if appropriate for the situation. The only property of a good implementation is that it makes effective use of the hardware resources and runs quickly. As was said in chapter 4, what is important in a harness is that all workers are allocated jobs quickly. Also, as programmers, we must be careful of our prejudices preventing us from doing things that are efficient.

What is also of interest about Bartoszek's implementation is that a high degree of efficiency is obtained even though only one job is farmed out to each worker. Thus not using one of the major advantages of this method of farming out. Effectively this implementation behaves more like a geometric distribution, synchronisations being performed after every item of work. Indeed these communications are made with a distant farmer not with neighbouring processors, making the implementation potentially even more inefficient.

Conclusion

Here the workers are given their first job at compile time. Workers also maintain their state and the harness directs jobs to specific workers. Only one job per worker is given out at a time. The next group of jobs is generated from the best result of the previous group. A job may be a continue job. The farmer is placed on a separate processor as it does some housekeeping in between jobs.

The only property of a good implementation is that it makes effective use of the hardware resources and runs quickly. With parallel hardware this means always having a number of things to do in parallel. Something that was not the case with the towers of Hanoi and quicksort.

In a harness it is important that all workers are allocated jobs quickly.

5.7.2 Some other examples of farming out

In [And91] Andrews gives two parallel designs that make us of the replicated workers paradigm (farming out). There are also many references to other examples.

The two parallel designs discussed are computing the area under a curve and the travelling salesman problem. Both of these applications possess jobs that are independent of the other jobs within the application. Thus these applications are akin to the ray tracing and Mandelbrot set applications in this respect, opposed to the towers of Hanoi and quicksort applications where the jobs are interrelated. In computing area under curve, as with the towers of Hanoi, messages are of a constant size. Also like both Hanoi and quicksort, the initial number of jobs is one, and this continually doubles throughout the running of the program. In the travelling salesman problem there is constant number of jobs, all of which have a constant message size.

These have designs developed using different communication mechanisms to both each other and to the implementations here.

Thus these two applications and their designs are of interest. Here we looked at the two examples, then at the view of programming used in the work.

Computing the area under a curve

This application consists of computing the area under the curve of a continuous non-negative function. An approximation of the true area is given by dividing the interval in two and summing the areas of the trapezoids formed. The interval can be repeatedly subdivided until two successive approximations are close enough. This decision can be done dynamically at the low level end of the algorithm, by only further partitioning subintervals whose estimates are not close to the estimate of the interval which they're derived from. Performing this decision is quicker when parts of the curve are flat.

There are some minor differences between Andrews's design and those developed by this author. This is a result of the different communication mechanisms used. The design used here is written using channels that use asynchronous message passing with unlimited buffering and a first-in-first-out (FIFO) semantics. These channels are used for carrying the results to the administrator (farmer) process and to implement the bag.

Andrews's design consists of a bag of problems (jobs), each of which consists if an interval and the current estimate for its area. The farmer process outputs the whole interval into the bag channel and then receives any results on the result channel. The workers take jobs from this bag directly. The interval is subdivided, the two subareas are then computed and their sum is compared with the area's original estimate. If the difference is small the old estimate is produced as a result for that subinterval, if the difference is not small the two subintervals are returned to the bag with their estimated areas for further subdivision. This method of organising work is clearly farming out and benefits from all of its usual advantages.

The program terminates when the farmer detects the area has been computed accurately enough. Although no implementation is mentioned, Andrews says it is difficult to detect for an empty bag and for idle workers. This has not found to be the case in the two examples developed on the synchronous message passing hardware used here. This is because here the bag had to be implemented by hand and as a result only a simple condition was required in the farmer. This condition checked to see if the "number of jobs in farm" counter or if any of the appropriate stack pointers were greater than zero. In the towers of Hanoi farm this was,

```
WHILE (sp.top > 0) OR (mi.top > 0) OR (jobs.in.farm > 0)
```

Using the unlimited buffering capacity of the asynchronous communication channels to implement the bag of work removes control from the programmer. For example, due to the FIFO nature of the communication channels there can be no reordering of the jobs. The towers of Hanoi farmer gave out the job that was the closest to the leading edge of the calculation so as to start printing the answer as quickly as possible. The quicksort farmer gave out the smallest jobs first so as to save stack space.

In the programming notation used here, like in occam, a process only mentions the names of the channels it communicates with, not the name of the other processes.

The travelling salesman problem

In the travelling salesman problem there are n cities ($n \geq 2$) and a n^2 matrix of distances from city i to city j. There is a (finite) path from every city to every other city,

$$\forall\, i, j.dist[i, j] < \infty$$

but not to itself,

$$\forall\, i.dist[i, i] = 0$$

This matrix is also symmetric,

$$\forall\, i, j.dist[i, j] = dist[j, i]$$

Each city should be visited once (the array *shortestpath*[] will contain a permutation of the integers 1 to n) via the shortest path,

$$dist[shortestpath[1], shortestpath[n]] + \sum_{i=1}^{i=n-1} dist[shortestpath[i], shortestpath[i+1]]$$

this includes returning to the first city.

The strategy suggested is to have a farm with a bag of $n-1$ jobs, each job being a path that starts at city 1. The Manager takes a path from the bag and gives it to a worker. Workers receive a path and add an unvisited city to it. If this path is longer than the current shortest path, the path is discarded, cutting down the searching space from $(n-1)!$ paths. If the short path is incomplete, it is returned to the bag with its new length. If the short path is complete, these values are sent to the farmer who updates its variables *shortestpath*[] and *shortestlength*. Again this strategy is clearly farming out.

The communication mechanisms used in this example are Remote Procedure Call and rendezvous. The notation used for communication in this example is slightly different to that used in the previous example. Here the communications mention explicitly the name of the process being communicated with. Thus processes would need to be rewritten in order to be reused elsewhere. Further, the operations used can only be serviced by the modules that declare them. Thus in order for the workers in this example to put jobs into the bag and to send a new shortest path length, the name of the Manager has to be referenced explicitly. This again does not aid code reuse.

When a new shortest path is found, its length is broadcast to all workers. As this event may occur reasonably frequently, some workers may receive a number of broadcasts before starting a new job. This is potentially wasteful of communication resources, especially as the granularity of the application is reasonably fine. This may result in a communication bound implementation. As each broadcast creates w messages. If the product of both w and the number of broadcasts performed by a program is large, it is possible the number of path length messages sent could be larger than the total number of jobs farmed out. An better approach might be to send the length of the new shortest path as a part of each job. This could result in less data sent overall as fewer copies of the shortest path's current length will be sent out when many new shorter and shorter paths are found in quick succession. Further, this communication strategy would definitely result in less messages being setup and communicated overall, as only jobs are sent out. These jobs are larger than before, but the number of communication that need setting up is the same.

View of programming

In the introduction to the message passing section of [And91] on page 342, Andrews says there are "several process-interaction patterns that occur in distributed programs. Each *interaction paradigm* is an example or model of a communication pattern and associated programming technique that can be used to solve a variety of interesting distributed programming problems. Because each paradigm is different, there is no single derivation method for distributed programs."

This view of how parallel programming should be performed is not as useful as the view used in UNITY, especially as Andrews lists eight interaction paradigms. In UNITY, all program designs can be developed in a single uniform manner that initially ignores architectural concerns. Once developed, a parallel design can then be mapped onto an architecture using any number of appropriate execution strategies. Thus parallel program designs can be developed to solve problems regardless of the interaction pattern between the processes by developing the design before it is decided what architectures (whether distributed or not) the program will be run on. For example, Bartoszek's farmed out search program discussed previous used a parallel equivalent of a sequential depth-first search. Andrews also mentions an interaction paradigm called probe/echo as a parallel equivalent to the depth-first search.

It is likely at least some of the other applications that Andrews references under his other interaction paradigms could also be farmed out efficiently.

One of the other interaction paradigms mentioned is the client-server paradigm. In this one server processes requests for its many clients. Farming out is the opposite to this, a number of servers process requests for the one client.

5.8 Other man-made examples of farming out

Here we look at the use of farming out in some other current man-made systems. This discovery started when it was realised a manufacturing company farmed out the assembly of its products.

5.8.1 Farming out in manufacturing

Here we look at the manufacturing company Linn Products Limited. The company is organised so one person assembles, tests and packs each product. It was noticed this is farming out, not the production line approach normally used for product assembly. Thus Linn is of interest from an algorithm decomposition and parallel execution standpoint.

Linn strive to manufacture equipment that reproduces music in the home with the same involvement, intelligibility and emotional impact of a live performance. Achieving this requires a high standard of accuracy. The quality of both product design and assembly are important to the level of performance achieved by a product.

Original approach

In [Tie93] Tiefenbrun, the company's founder and managing director, says Linn's original 1973 structure was copied from General Motors, the world's largest manufacturing company at the time. This structure was the conventional production line (algorithmic decomposition). Using this structure Linn quoted a seven month delivery time and seldom met it.

New approach

When computerising the business no rational way could be found to program the production. At one point the managing director asked someone from the assembly line to collect all the parts required to build a record player, assemble them and bring the finished turntable back to his office. This task took about 17 minutes, instead of the 27 minutes of labour it took to build the same product on the production line. This lead to the company being completely restructured, abandoning many traditional techniques in the process.

The whole factory was reorganised so that wherever possible, one person assembles and tests a product in its entirity. This led to a saving of space as a total of 47 main buffer and storage areas were eliminated. Within three months real-time production had been obtained and products were being dispatched to the customer the same day they were ordered.

The assembly section is organised into three teams, one for each type of product (sources, control and playback) the customer sees. During the quieter period of trading during the summer, the members of these teams have the option to train to become members of the other teams. Thus, the three team assembly stage is even closer to being just one farm and thus having the potential to fully load balance, as nearly any product can be assembled by any worker.

In subsection 5.2.2 (page 120) it was suggested all job types should be performed by all workers. While although it is obviously possible to get extremely close to this optimum by having workers that can be moved from one team to another at a moments notice, getting even closer is possible. The loss in performance would only be slight compared with using a different method of organisation.

Advantages of the new approach

It was found that farming out product assembly results in only necessary dependencies. For instance there are still dependencies between the components in a product, but not between different instances of the same product, a dependency present with a production line. Previously an machinery failure or the non-appearance of any individual could result in the stoppage of the entire production line. Now, the non-appearance of an individual or machinery failure only reduces the rate of production, instead of stopping it completely. This issue is not covered in this research, but it is another advantage to farming. It is equivalent to a worker processor failing.

Linn have found the largest advantage of this approach is that of much greater workforce flexibility. One obvious example is that a production (pipe) line takes a long time to be started and to be brought to a stop.

Another advantage that Linn discovered quickly is one not seen in the transputer community. That farming out work led to products of a much higher quality, even though there was no direct supervision of assembly.

With the single-stage build (as Linn call it) each employee has a task that is intelligible, complete with responsibility and control over what he or she does, instead of just repeatedly performing the same trivial task without obtaining any quality control feedback. The same person who builds a product, is the same person who tests it, listens to it and thus sees a connection between what they can do and how the product performs. The assembly teams talk directly to the designers on how products can be improved. There is no service department, a product goes back to the team that built it.

Tiefenbrun believes these more flexible methods are the reason why Linn Products out performed all the U.K. competition, and puts forward that manufacturing industry should adopt this more flexible method of organisation if it is to make any advance. The Industrial Revolution is given as showing the scale of the advantage that is to be gained through using a superior methodology. Tiefenbrun concludes by saying that using people (workers) to their highest potential is what is the most important.

Overall company structure

Farming out is used within Linn's assembly teams. It is worth looking at how the overall tasks are decomposed into teams and how these teams are interconnected.

The technique of organisation developed and used by Linn, the single-stage build, is more general than farming out. The company's approach is to use every individual to their highest level of potential. In the single-stage approach as much as one person can control and be responsible for is gathered at every stage (node) and is taken as far as is possible towards the final objective. Thus their communication structures resembles a tree. This approach is used in everything they do: design, ordering materials, sales and accounts, as well as assembly. The whole company is structured into teams. Each team is connected to others.

Linn's internal structure could be described as consisting of algorithmically different teams, some of whom farm out data jobs internally. The jobs only consisting of items or materials to be processed; they

do not come with what should be performed. The teams form an algorithmic decomposition as each team performs a different job.

Thus Linn is not one big farm, but a tree of farms. Any job can only be performed by a subset of the workforce. This is not the most efficient arrangement possible, however, it is still highly efficient. Having one large general pool of work and all workers capable of performing any type of job would be impractical, and if possible, would not lead to a much greater degree of flexibility.

In large organisations many skills are required and these are difficult to communicate simply. Thus, having tree structures with as much gathered at every stage as one worker can control is a much more sensible method of organisation, opposed to always attempting to farm out every single job to any worker. As has been said, the loss in generality, flexibility and efficiency can be much less than the performance obtained through using another method of organisation.

Thus, Linn Products is an algorithmic decomposition. Decomposing an algorithm into steps that take equal lengths of time is very difficult, if not impossible generally. By having a algorithmic decomposition of teams, each of which can be varied in size, the differences can be taken into account by simply adjusting the size of the teams.

Comparing efficiency results

On the course [Wel] Welch mentioned that 70% was a good level of efficiency to obtain from an algorithmic distribution and 99% was a good level of efficiency to obtain from a farm.

It took about 27 minutes for a turntable to assembled on the production line and it took only about 17 minutes for one to be assembled by a single person. As it can be assumed that the single person was working continually, and thus at 100% efficiency, the production line can only be working at 62.9% efficiency. A figure that is reasonably close to the 70% efficiency mentioned by Welch.

Nomenclature

Tiefenbrun calls this approach female manufacturing. Management theorists and analysts are now also arriving at the conclusion that real-time approaches to organising work are greatly superior to the static approaches used in manufacturing today. Management theorists term this approach Fordism, as the technique was exploited highly in the manufacturing industry by Henry Ford. The modern approach is called post-Fordism.

Conclusions

At Linn, farming out orders among a team of single-stage building stations requires a higher skill level (more memory), and as well as the usual advantages also results in a superior build quality, lower costs, increased output per employee (worker or processor) and dramatically improved labour flexibility.

Decomposing large tasks algorithmically into a tree with a very small number of nodes (teams), and the work in each node being farmed out to a number of workers is an efficient. The number of workers in each node can be selected to be proportional to the processing time for that node. To increase the balance of the system further, some workers can be trained to work in a number of nodes.

Instructions and especially skills may be difficult to communicate quickly. However, it is easy to just farm out the data or materials to be worked on. This loses generality a little bit and some flexibility and thus potentially some efficiency. Nevertheless, the amount of performance lost is likely to be less than using another method of organisation.

5.8.2 Other man-made examples of farming out

The methods of organising work used in parallel programming are also used to organise other man-made processing structures.

Farming out is employed in post offices and banks. Customers (who possess jobs) are in a single queue, the customer at the head of queue goes to the first free serving window (worker). As here there is only one queue, we are truly farming out this queue of jobs. Thus, the allocation of customers to windows

is performed dynamically and windows opening and closing is dealt with automatically. This system is fair in that customers are served in the order they arrive.

Farming out is also employed in railway stations and supermarkets. However, here the bag of work is not kept in one queue, but there is a queue for each window or checkout and customers join the back of any queue. As different customers take different lengths of time to be served, the rate at which each queue moves is also different, the shortest queue can be the slowest to be served, an observation to which society is commonly aware, and the customers are not served in the order they arrived. Thus just having a queue for each worker is results in unfairness as the allocation of people to queues is performed according to the lengths of the queues not which workers are free.

Also changing the windows or checkouts open causes problems as queues will need to be reshuffled or merged. This ordering of the new queues tend to take place according to previous geographic location and not according to the time of arrival in a queue, and thus changing the windows open to meet increased demand can turn out to be unfair to some customers.

This shows that it is better having just one queue representing the bag of jobs, as this is fair. This example also highlights the problems that may also be found if multiple farmers (discussed in subsection 5.2.6, page 121) were to be implemented.

That said having a queue for each worker that is local to that worker is efficient. With pure farming out the worker may be idle while waiting for the next job. As before this can be alleviated by having a single item queue or buffer for each worker. Here, this entails having one person stand behind each person being served. This will lead to a slightly unfairness as some people will then get served out of order. Nevertheless, this will be more efficient as this strategy will result in a better throughput of customers as each customer will get seen quicker than in the completely fair approach.

A car production line has already been mentioned. Here both algorithmic and geometric decomposition are used. Each production line is an algorithm distribution. Often many of these production lines are set up side-by-side.

Paper work and administrative tasks are also invariably dealt with in an algorithmic way, in council offices for example. A receptionist will accept a form. This is passed on to the relevant department. There it is examined by someone who may generate some internal memo for someone else who is responsible for a letter to be written.

In summary, it can be seen here that by farming out work, it is organised more fairly as jobs are allocated in the order they were placed in the bag or queue and are allocating to workers that are free when the jobs are about to be performed.

5.9 A clearer understanding of the stages of implementation

Presented here is an approach to viewing the stages of implementation development that provides a much greater understanding of the issues that have the greatest impact on an implementation's performance. This includes a clearer view of the choices open to the programmer when developing an implementation.

5.9.1 Parallelisation

A number of points of interest were noticed in two of the previous examples.

In Bartoszek's implementation the workers only performed one job in each stage of the calculation. Thus there was no opportunity for the work load to be balanced in any way. Further, each communication the workers performed was over a long range with the farmer via a harness, not over a short range with nearest neighbours. Nevertheless, despite suffering from all the disadvantages of both farming out and geometric distributions, and not obtaining any of the generally quoted advantages of either execution strategy, this implementation is still very efficient.

At Linn Products, product assembly is much more efficient when farmed out than when performed by an algorithmic production line.

Similarly, on Welch's course, the levels of efficiency expected from a good implementation is much higher for farming and geometric distributions (99% and 90%) than it is for algorithmic distributions (70%) [Wel].

It was realised that parallelism in farming and geometric decompositions is between independently processed items of data, whereas the parallelism in algorithmic decompositions is between both the separate parts of the computation and the separate items of data. The ability to perform data items or algorithmic components in parallel comes out of those parts being independent of one another.

Which independencies are selected for use in both program designs and mappings have a significant influence on the efficiency of the final implementation. In particular, it is whether computational independencies are made use of that has the significant influence on efficiency.

It is usually more efficient to perform a computation all in one go. Decomposing a computation into many interlinked parallel processes has two disadvantages. First, the overhead of communication is introduced. Communication is not necessarily expensive, but it does have its costs. Second, if the processes are executed on different processors, and if each process takes a different length of time to execute (which is invariably the case), some processes will be become idle while waiting for synchronisations to happen.

Thus, dividing up algorithms should be avoided as much as possible, for efficiency reasons, even when it is necessary it should be done as little as possible. How we use the independencies within the application to arrive at parallel processes and how those parallel processes are then allocated are separate concerns.

This looks like a much more useful way of view how to go about implementation than trying to select one of the three execution strategies.

5.9.2 Allocation

Allocation is separate to and happens before execution. Here we look at the two types of allocation seen here: static (or predetermined) allocation and dynamic (or run-time) allocation.

With static or predetermined allocation the processes are allocated to the hardware as is, according to the shape of the parallel design. This is performed before the program is run.

With dynamic or run-time allocation the processes can be allocated to the hardware according to what processors are free at the time of allocation.

5.9.3 Performing allocation in hardware

Current hardware only executes pre-allocated parallel processes. Thus, if dynamic allocation is desired, it must be performed by a harness of nondeterministic ALT processes, such as those in chapter 4. With such architectures the overall efficiency of a program is not just limited by how the program has been written, but also by the efficiency of the allocation software used.

By performing process allocation in hardware however, all the performance advantages of dynamic allocation are obtained without the overhead of writing or executing a software allocation system. The parallel processes can be farmed out at the hardware level regardless of the method of process allocation used at the software level.

In [Wel95] Welch describes the design of a system that could perform process allocation in hardware. This architecture consists of a number of processing nodes and memory nodes, these are connected via links to switching nodes.

When a process reaches a synchronisation point, it is automatically descheduled. It is this mechanism of automatically descheduling at synchronisations that allows such architectures to farm out non-independent processes. This is how the multiprocessor shared memory systems (mentioned in subsection 5.5.2, page 156) can farm out their process queue. Thus, processes like the two seen in subsection 5.5.3,

```
CHAN OF Type chan:
PAR
  SEQ
    ...  a
    chan ! intermediate-result
    ...  b
  SEQ
    ...  C
    chan ? parameter
    ...  D
```

can be farmed out directly without them needing to be decomposed into processes that do not communicate. Thus on this architecture, a process is executed by a processor until it reaches a communication. The communication would be set up and the process descheduled. After the communication completes that process is returned to the back of the process queue, a structure that is (securely) shared between all processors. When it reaches the front of that queue, it is rescheduled to any demanding processor.

On this architecture processors execute instructions and just instruct the switching fabric to perform the transfers between: memory and processor, and memory and memory. This allows for communications and assignments to be performed using the same architectural mechanics. In turn, this allows direct worker to worker communication, and thus process to process communication. This design also allows for many communications to be performed in parallel, thus it is more likely that communication rich programs will execute more efficiently on this architecture.

If an architecture does have the mechanics to deschedule a process and return it with its state to the bag ready to be continued, then the program should have any communicating processes decomposed into computation-only processes, as was prescribed in subsection 5.5.3 (page 157).

As it is unlikely that mutual dependencies will be executed simultaneously on any architecture that uses dynamic allocation, it is likely that such a set of communicating processes will be descheduled while waiting for communications to be performed.

Worker to worker communication is not mentioned in the farming model. Farming is only concerned about the mechanics of allocation, and not about the details of what is allocated, such as any dependency between jobs. Thus, the precise details of how communication is performed when farming is dependent upon architecture.

It is interesting to note that the abstract of [Wel95] gives the same specification for a method of parallel program development that was made by the work that developed UNITY: simple, mathematically sound, can express the parallelism within applications and yet is architectural neutral.

5.9.4 What the execution strategies really are

An algorithmic distribution is where both the data and computational independencies have been parallelised and the processes are allocated onto the hardware in a predetermined way.

A geometric distribution is where the data independencies have been parallelised, again the processes are allocated onto the hardware in a predetermined way. To date geometric distributions have been constructed when the application consists of data in a regular geometric shape. This shape is reflected in the implementation's topology. Hence the name geometric distributions.

Farming out has also been used where the data independencies have been parallelised. The difference with farming and the other two, as discussed in subsection 5.5.9 (page 161), is that the jobs are allocated to the worker processes dynamically (or at run-time).

As it has now been realised, jobs can also consists of computational processes as well as data, subsection 5.5.1 (page 155), thus farming out can also be used when both the data and computational independencies have been parallelised. Again with the allocation being performed at run time. However, as parallelising both data and computational independencies is less efficient than just parallelising data independencies, this is not to be recommended. This being due to an application running slower if farmed out in stages, as was suggested in subsection 5.2.1 (page 120), due to the extra overheads of communicating intermediate jobs and results.

In dynamic process allocation the processes are continually reallocated to the various processors at run-time in order to try keeping the work load balanced.

5.9.5 How these stages affect programming

How a parallel implementation is developed can now be separated into two concerns: the designing and mapping parallel processes, and allocation.

Developing a parallel design and mapping

Looking at the theoretical level, applications can possess independencies between the various parts of the computation and between the various items of data. When developing a program design and a mapping onto an architecture, we might be able to choose some of these independent things to perform independently of one another in parallel. We might also choose to use none (sequential programming).

If we can select which independent parts of the application are to be performed in parallel, we can either make use of the the independencies between the data items, computing these items of data in parallel and have a copy of the algorithm for each item of data (figure 67),

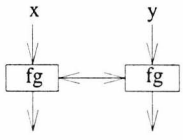


Figure 67: Making use of the independencies within the data

or, if there are no interdependencies between items of data for the computation to be performed, we can choose to make use of both this (in the usual way we make use of independencies for sequential purpose) and the independencies between the stages of the computation, and have each stage on a processing node to itself (figure 68).

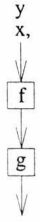


Figure 68: Making use of the independencies within the data and the computation

We find writing this type of parallelism very easy as it is so close to sequential programming. It is also possible to use both sets of independencies further, by making full use of the independencies between the items of data (figure 69).

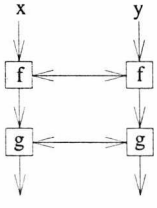


Figure 69: Making more use of the independencies within the data and the computation

So, a program design, before or after being mapped onto an architecture, consists of a number of parallel processes. These interact, or communicate, in some structure. This structure is based either on the shape inherent in the data or on the shape inherent in structure of the data and computation.

When the shape of the design or mapping is taken from just the shape of the data, a number of processes are arranged according to the geometric shape of the data. This is easy to perform when the data is in some simple geometric shape, such as a rectangular grid. When the shape of the design or mapping is also taken from the shape of the computation, a number of processes are arranged according to the shape of the computational system.

So, those are the various options open to us. In terms of efficiency, implementations that use just the independencies between the items of data are much more likely to be efficient (as less communications are added). Thus, it is best to see if the application can be parallelised by just using data parallelism. Similarly, if the computation needs to be split up, the number of stages it is decomposed into should be as small as is sensibly possible so that as few additional communications are introduced. There are also likely to be some advantages in balancing the amount of work between each of these stages.

Thus, although algorithmic parallelism seems to be a favoured method to break up a large algorithm, and it is easy to think that this is an easy way to make things parallel with lots of advantages, in fact this is not the case. While this approach does add parallelism it does so a way that is not efficient. Geometric parallelism is just as easy, if not easier, to develop and is efficient, as well as adding parallelism.

Thus in conclusion, data parallelism should be attempted before computational (or algorithmic) parallelism. Both are trivial, but the former does not introduce so many overheads and is therefore more likely to be efficient, thus should be attempted first.

Once a design is developed and mapped, these processes can then be allocated to processors and executed by the hardware.

Allocation

If the hardware does not have a dynamic (run-time) allocation, it should be considered whether it is worth implementing, and if so, if it would be more efficient than static allocation. To find out the most efficient would involve trying both. Nevertheless, dynamic allocation has flexibility advantages over static allocation. A dynamic allocation method will be efficient if its overheads are small and lightweight, and if the rate at which work can be supplied can be greater than the rate at which work is demanded.

As has been said, decomposing computations into a large number of stages will probably slow down an implementation, due to the communications introduced. In order to be farmed on architectures that can not perform worker to worker communication, processes with dependencies between them will need to be decomposed into processes that contain only computation. Unfortunately, due to the communication overheads of dynamic allocation, farming out is likely to be less efficient for computations that consist largely of fine grain parallelism.

5.9.6 Static and dynamic allocation compared

In [JR92] Jenson and Reed performed a few comparisons between two geometric decompositions and farming. This was performed via a ray tracing application on an Intel iPSC/2 hypercube. Two geometric decompositions were used. One used the standard tiled approach (as in figure 62, page 163). The other was a scattered approach in which row i was allocated to processor $i \bmod P$. It was found that farming and the scattered geometric approach were found to be about equal in performance terms, give or take 15 seconds either way on a 10 minute run, or about 2.5% difference either way.

Jenson and Reed concluded that the dynamic approach of allocating work was slightly slower overall. The reason there being due to the large cost of setting up a communication. Thus, the overall performance is related to the number of communications performed overall. Here, a statically allocated implementation could perform either no or P job allocations at the beginning of the program, depending on how it is implemented. Whereas the number of allocations a farmed implementation would have to perform would be many times larger than P.

Jenson and Reed also say the dynamic approach (farming out) can only be efficient if j can be offset against the latency of the communication setup time s. More precisely,

$$j > \frac{w(m+s)}{B}$$

As we looked at in chapter 4 (page 52), it is very expensive to set up a communication from the UNIX host Jenson and Reed used to a Intel iPSC/2 hypercube. Therefore it is not surprising this is the only factor they noticed.

5.9.7 Summary and Conclusions

An application has two types of independency. Those between its various items of data and those between its various computation stages. The process of implementation consists of using these independencies to develop a parallel design consisting of processes. This design can then be mapped onto an architecture. This may include transforming the processes of the design in some way (granularity adjustment for example) into another set of processes. These processes can then be allocated for execution in either a predetermined or a dynamic manner.

It appears to be more efficient to use just the independencies between the items of data, than to use the independencies between the various parts of the computation as well. It also appears to be more efficient to allocate parallel processes for execution dynamically as well.

Thus, for efficiency purposes, decomposing computations should be avoided if at all possible. Independent jobs should be kept intact by farming out a single sequence of jobs as was done in chapter 4. Nevertheless, decomposing computations is necessary when there are dependencies between different parts of the computation. This introduces the overhead of communication, if it was not there already. Therefore the smaller the number of stages the better.

Breaking a computation up into stages can be worth performing if there is not enough memory for each worker to be able to perform all of a complex task. This can be achieved by having a tree of separate farms and gathering the results of each section towards the final objective, as performed at Linn Products.

The processes in a program design or a mapping have a shape. This shape is the same shape as the data or the computational components.

One possible further conclusion is that the more types of independencies are parallelised together, the lower the final efficiency of the implementation. However, as we are only aware of two types of parallelism inherent within applications generalising further like this does not appear to have any merit.

In programming we need to be able to find processes that are independent of one another and that are of a grain suitable for both the grain of the hardware and any overheads the method of execution may also impose.

5.10 Closing discussions and summary of conclusions

This study set out to find what range of applications could be farmed out. It has been discovered that it is possible to farm out all application that are described as sets of independent computations. How dependencies between these computations are dealt with is dependent upon the architecture used.

This chapter closes with two discussions and a summary.

The first discussion is on the factors that affects efficiency. The second discussion is on one aspect of how transputers are programmed, how this aspect should be viewed and thus how transputers should be programmed.

The summary reiterates the conclusions arrived at in this chapter.

5.10.1 Factors affecting efficiency

Generally we are interested in whether an efficient implementation can be developed. In chapter 4 it was realised that an implementation would be efficient if a compute bound mapping could be found for the architecture. This involving the granularity being appropriate etc.

In addition to the two factors that were realised to affect farm efficiency in chapter 4, it has been realised here that there are another two factors related directly to the internal structure of the application's parallelism. Thus we now realise that farming can be efficient only if,

1. there is more work within the application than there are workers,
2. this parallelism can be decomposed so there is a reasonably continuous supply of jobs,
3. the supply of work is greater than the demand generated by the number of workers used, and,
4. this decomposition is of a granularity that is above that of the architecture.

An example of a parallel application that was well designed and mapped was seen in the minimal perfect hash function finding program in subsection 5.7.1 (page 167). The Hanoi and quicksort farms are bad examples. For example they both initially had only one job. In quicksort the job size also diminished throughout the run of the program.

The above three points probably apply to all implementations not just farms. So generally, if we are to fully utilise any (parallel) hardware, work must be organised so that all processors are performing work for as much of the time overall as possible.

5.10.2 One aspect of how transputer programming is and should be performed

Currently there is no known rigorous method with which to select an execution strategy. A lot of applications appear to be implemented by simply using the method of execution that is closest to the internal structure of the application, as this is the easiest to implement. Thus, a set of independent jobs, like pixel-wise operations and the protein sequencing program are being farmed; a set of data that needs to be processed repeatedly are being implemented geometrically; and everything else is just being implemented algorithmically.

There does not appear to be any method to find what the most efficient and flexible method of execution is for that application.

Part of this problem stems from applications being designed and implemented at the same time, not designing the application first and then implementing it second as UNITY suggests.

Here it has been realised that when implementing an application, there may be many possible mappings. So, in every programming situation it is worth looking to see what alternatives there are. Doing this will result in a better solution as then one can choose an alternative that is the most appropriate in terms of flexibility and/or efficiency.

5.10.3 Summary of conclusions

Here we list the fundamental conclusions arrived at in this chapter. This includes the model of farming developed. Some conclusions related to farming are listed. Presented last are conclusions not directly related to farming.

Main conclusion

The most important realisations of this chapter are that it is possible to farm any application that is expressed as independent computations, and that it is likely to be efficient if there is a constant supply of work that is quicker to allocate than to perform.

There are four parts that this second point of farming efficiency is related to: the application, the program design, its mapping and the architecture (these were discussed more fully at the top of this page). Thus, only applications that are parallel in nature should be farmed and they should be implemented in a way such that this parallelism is preserved in the program design and the mapping used for the architecture.

Farming model

Farming out is one way of arranging for work to be performed, the allocation of the jobs being performed dynamically. It consists of continually communicating the next job of work (from a bag) to a free worker that performs that job. The bag and the workers are arranged as in figure 70 (top of next page).

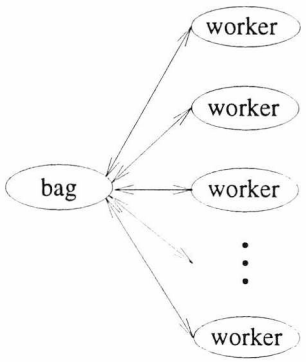


Figure 70: A model of farming out

Jobs may produce results and further jobs. The results produced should be dealt with as appropriate, invariably they are collected and output. Any jobs generated are returned to the bag. Thus keeping the mechanics of job distribution simple. This model's basic structure allows for variations *within* this structure for the situation in hand. A more detailed description of the model can be found in subsection 5.5.6 (page 158), which includes how the model can be varied.

Farming conclusions

Implementations consist of four stages: parallel design, mapping, allocation and execution. Applications can be parallelised using the independencies between the items of data and the stages of the computations. Mapping may reconfigure the processes into a different set of process that the architecture can execute. For example, by making the granularity of the processes coarser or finer. Allocation can be static and predetermined or performed dynamically in real-time. Farming out is one method by which processes can be allocated dynamically for execution.

Applications possess independencies between two the various parts of the computation. Which of these are used to develop a set of parallel processes is a separate concern from how those processes are then allocated to the processors of a parallel architecture. Nevertheless, the two are related. For instance, some mappings will be more efficient when allocated statically than when allocated dynamically, other mappings will be more efficient when allocated dynamically. Thus, it is easier to consider the development of an efficient implementation in detail if the issues of mapping and allocation can be considered separately as well as jointly.

Scalability is dependent upon the total amount of parallelism inherent within the application and the design.

If communicational dependencies between the processes can not be dealt with directly by the architecture, they can be dealt with by the process that manages the jobs.

The largest two additions to farming mechanics made here are that: one, applications can be farmed out as a sequence of stages, with each group of jobs providing the jobs for the next stage; two, workers may perform more than one type of job.

Another possibility that we are now aware of is that job messages can contain processes instead of, or as well as, data.

Whether or not all workers will be able to process all job types is just a question of the amount of memory the processors possess. If there is not enough memory on each worker to perform every type of work within the application, the best way to proceed may be to decompose the processes into stages and then to construct a suitably sized farm for each stage. These stages could be in the shape of a tree with each stage performing as much as possible and taking it as far as possible towards the final objective. This results in a series of farms running in parallel each of which performs a different type of work.

Farming related conclusions

A large amount of buffering can hinder the performance of applications that have been farmed out in a multi-stage way. For example, quicksort, the towers of Hanoi and Bartoszek's MPHF search. Buffering is not appropriate in the last example as there is no other job to be started immediately after a worker has finished one job.

As multiple processor shared memory systems possess the same structure of a farm, these systems can farm out their process queue. So far this has been used to speed up multi-user machines, but as yet programs have not been written in parallel to take advantage of the speed up available with these machines.

Miscellaneous conclusions

The experience gained here is that converting algorithms expressed sequential into a parallel form, such as occam is easy. The opposite is generally believed to be true.

Recursion restricts our thinking when attempting to redesign an algorithm into a parallel form.

Chapter 6

Future research and architectures

This chapter recommends both future areas of research and in what ways farming should be implemented on future architectures.

The following issues are considered,

1. extending the vocabulary of occam program transformations,
2. recommended future work for the first generation of transputer,
3. how one could implement farms on the second generation of transputer,
4. architectures that farm out work automatically, and,
5. developing the programming methodology further.

A summary is provided.

6.1 Program Transformation

In subsection 4.3.2 (page 39) one process was transformed into another. It is interesting to note that only one of occam's equivalence laws was used in this transformation. This is perhaps to be expected as the equivalence laws are of a low level and were not intended to be used in practical situations. Nevertheless, as such transformations are clearly of great use in the task of programming, a set of equivalence laws for practical use is desirable.

There are two ways seen here to approach the providing of equivalence laws. The approach taken by Roebbers has been to supply a handy set of tools that can be applied intuitively and practically as required by the situation. As an example, here is one transformation that was used in this research a number of times. It is expressed here as an equivalence,

```
                                WHILE TRUE
WHILE TRUE          ≡             SEQ
  P                                 P
                                    P
```

The approach taken by Roscoe and Hoare [RH86] is to provide a number of laws to transform between standard forms. Below is one example equivalence from [RH86]. These two processes are similar in nature to the two harness B job distribution processes transformed between in section 4.3.2 (page 39). In [RH86] the equivalence is between,

```
WHILE b
  SEQ
    P
    Q
```

and the state-machine like program,

```
BOOL x:
SEQ
  x := FALSE
  WHILE x OR b
  IF
    x
      SEQ
        Q
        x := FALSE
    TRUE
      SEQ
        P
        x := TRUE
```

6.1.1 Types of transformation

Three types of equivalences are needed: efficiency improvement, parallel introduction and parallel removal.

Efficiency improvement should involve equivalences between forms that are easy to write and forms that are efficient for a particular architecture. These can then be used to make obvious methods of expressing algorithms instantly efficient.

Parallel introduction is of use in transforming old sequential algorithms into those exhibiting parallelism (also see [Bar93]) and in introducing buffering, as mentioned in Roebbers's course, see section 2.3 (page 11).

Roebbers's course also looked at sequentialising a highly parallel algorithm. This is so the code sitting on each transputer was only sequential in nature and was thus more efficient. Equivalence laws to assist in this process are also required. The latter stages of this process involving reducing the amount memory used after the basic essential parallelism has been removed. This includes a variety of other transformations such as reducing the size of the arrays used.

In [RH86] Roscoe and Hoare also expressed a need for the first two of the three mentioned here.

6.1.2 The first of the Four Disciplines

In the forward to [CM87] Hoare states that a complete theory of programming includes four disciplines,

1. one for specifying programs that permit individual requirements to be clearly stated and combined.
2. one for reasoning about specifications and aids in elucidation and evalution of alternative designs,
3. one for developing programs with a proof their specification is met, and,
4. one for transforming programs to achieve high efficiency on the machines available for their execution.

Due to the rigorousness of the Roebbers's discipline, it is believed here this last item on the above list has been obtained for optimising occam programs for transputers and transputer networks.

Transforming programs until they are highly streamlined is usually referred to as *hacking* by the computing community in general. This being due to the informal and intuitive manner with which such program optimisations are usually performed. Nevertheless, here it has been seen how it is possible to perform such correctness preserving optimisations with more rigour and a stricter discipline than the rest of program construction process.

Roebbers's course provides a convenient, intuitive and practical set of tools that can be applied quickly and easily to any occam program as appropriate. Another advantage is that transformations can be performed mentally, without the need for any representation. Often several transformation steps can be performed at once in this way.

As the disciplines in Hoare's list are in decreasing order of generality, perhaps the other three disciplines will be arrived at in reverse order.

6.2 Future work

This section looks at how the work of chapters 4 and 5 may be extended.

6.2.1 Harness efficiency refinement

The most important piece of extra work to perform is that of how to scale ternary trees so the last layer of workers is not fully populated. What is needed here is to know how to setup the connections in the harness from the penultimate layer of workers to the workers in the partially populated last layer. This needs to be done in a regular fashion so the number of workers can still be simply varied linearly as before.

Although harness D had slightly poorer throughput over harness C, it did perform better when in a full farm. This presumably being due to it passing of pointers when communicating with the workers. Thus it would be worth studying the performance of versions of harnesses A and C that also passed pointers in this way.

Harnesses A and C could also be used on ternary trees if result messages could serve as requests for work. Once up and running, this is very easy to perform, and means that requests for work would be performed for free. Nevertheless, the initialisation case is a special case as a null result message will need to be sent from the workers to make the initial requests for work. The farmer would have to be able to recognise these null results and ignore them. In order for the communication protocol to be lightweight, it is best if this message is not of any particular special type from an external point of view, such as a tagged protocol. That said, a result message that is just a request-only result message must be computationally cheap for the farmer to recognise. The harness would also have to issue null result-requests in order to obtain work for its buffers.

There are a few extra minor tests that could be performed on the harnesses with the existing programs. The first of these would mean the recommendations finally arrived at in chapter 4 could be made more complete by seeing which harness is the best to use if one cannot turn off usage checking. Some of the most efficient harnesses found here were a tree run by harness D for compute bound mappings and a line of workers run by harness E for mappings that are approaching the boundary when a farm becomes communication bound. Both of these harnesses require usage checking to be turned off. It is desirable to know how well a tree running harness B and line of workers running harness F would perform.

Although even more efficient or lightweight harnesses may be possible, some further fine tuning of the harnesses used here may also be possible. For example, efficiency may be increased by introducing processes to buffer the job and result in and out of the worker process. Potential places for these are between the worker and either the job distributor or the result merger.

As mentioned in section 4.10 (page 93) a method for finding the job processing time that produces the optimum farm run time would be of use.

Automatically farming out on this generation of transputer is what is ultimately desirable.

6.2.2 Model parameters

A number of calculations have been performed here using the models developed. The parameters: job processing time, j, message size, m, and number of workers, w, are integers arrived at by the programmer and thus are accurate by definition. However, the bandwidth, B, and the communication set up overhead, s, are dictated by the implementational characteristics and need be measured.

The value used for B should be the average bandwidth between the farmer and the workers. This value is bounded closely by the two values that were used here, the bandwidth out of the farmer and between the harvester and the last worker.

Similarly, the value s should be the average number of bytes that could have been communicated in the time lost during an average farmer to worker communication due to the setting up of all the communications performed including all of those performed by the harness. The figure used here was the communication set up time for sending a message over one link.

Obviously both of these figures are affected by the different topologies and harnesses that can be used. For example, on a line of workers there are on average approximately $w/2$ individual communications to be set up, whereas on a ternary tree an average of just $\lceil \log_3 w \rceil$ communications need to be set up. This is just three individual communications for the largest farms used here.

6.2.3 Efficient mappings for farms that don't just produce final results

Until now many types of applications have not been farmed out. These applications can be implemented by farming out jobs that don't just produce final results, but also produce intermediate results and even further jobs. Now it has been realised these applications can be farmed out, we need to know,

- which applications,
- which parallel designs, and,
- which topology-harness combinations

will produce efficient implementations of these applications.

So far, in situations where minimal buffering is required, it seems that some of the harnesses tested in chapter 4 can still be of use. For instance, harness A has very little buffering and is highly efficient. Nevertheless, these harnesses are only known to be efficient when a *keen* job release mechanism is in use. Thus, harness strategies that are more appropriate in other situations may exist.

The parallel designs that have worked here have not been very efficient, due to lacking a constant supply of medium grain work. It is here where most of the further work needs to be done.

We are aware of three instances where study would provide a great deal more insight into how to farm out such applications efficiently,

1. when the farmer may give out jobs in a sequence of bursts, with no work being given out in the pauses in between,
2. when the leading edge of the computation or some other small area of the computational graph needs to be concentrated on, and,
3. farming out work that is then executed directly by the workers.

An example of an implementation that farmed out bursts of jobs was the minimal perfect hash function finding application [BCK93] discussed in subsection 5.7.1 (page 167).

An example of a farm that need to continually expand the leading edge of the computation was the towers of Hanoi application developed in section 5.3 (page 122). Here it was important to continue to expand the leading edge of the calculation so the first results could be output as soon as possible. Thus, as working on the new leading edge as soon after it was generated as possible had more of an impact on performance than the general efficiency of the harness, it was more important to have a harness that performs as small an amount of efficiency enhancing buffering as possible. Thus, harness A was used in this and the quicksort farms, due to it possessing the smallest amount of buffering out of the six harnesses, one job and one result per worker. A tree topology and a more elaborate harness would buffer four of each type of message per worker.

Another avenue that might be worth exploring is that of getting the workers to execute directly the contents of the jobs they receive. In order for this to be achieved the worker needs to be an execution harness that can add the process it receives onto the process queue of the C.P.U. This is the same as some parts of the mechanics used in dynamic process allocation, which is effectively what is going on, however we only call it farming out if we are allocating jobs from a single bag. Dynamic process allocation tends to involve moving processes around while they are executing so as to distribute the work load as effectively as possible.

Thus in general, it is desirable to know what types of parallel design will result in efficient implementations of applications where the computation needs to be decomposed into a number of stages. This study is large. It was avoided here by studying the harnesses before studying what applications could be farmed.

6.2.4 Understanding when to use each execution strategy

One aspect of this study of farming has been looking at what type of applications will farm efficiently.

The work has revealed that in order to develop an implementation of the appropriate level of flexibility, scalability and efficiency, a programmer needs to be able to evaluate which execution strategy is the most appropriate for the application and situation.

It is desirable that this best way to proceed with an implementation should be obvious from the internal nature of the application. From the application one should be able to determine whether the most appropriate mapping consists of data or computation decomposition, and dynamic or static allocation.

6.2.5 Farms with different types of worker

It may also be of use to know how to best arrange farms where the different workers specialise in different kinds of work.

For example, in banks we have started to see some windows that deal only with small transactions in addition to the usual windows that deal with the full range of transaction services.

When customers with large transactions get to the front of the queue,

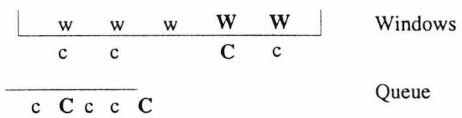


Figure 71: A multiple workers queuing problem

these customers should progress to a second queue that is for customers still waiting for a full transaction window to become free.

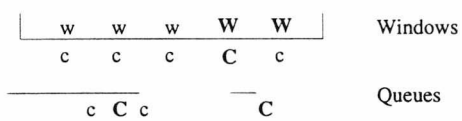


Figure 72: Solution to multiple workers queuing problem

This prevents customers with smaller transactions from being delayed. All customers should initially join the main queue to preserve ordering.

Further work in this general area is required.

6.3 The second generation of Inmos transputer

The second generation of transputer products implement two new facilities. These are extra communication mechanisms that affect the way work can be farmed out. Here we looked at these two new mechanisms and then at the various issues of how these should be used to implement farms.

6.3.1 The new computation and communication performance ratio

As the speed of the processors and the links have been improved, the performance ratio between computation and communication is also different. What needs to be found are the thresholds of farm efficiency for this second generation of transputer, just as was done here in sections 4.9 and 4.12 (pages 68 and 99).

6.3.2 Virtual channels

The first of these new mechanisms is a communication sub-system. The new transputer, the T9000, has a virtual channel processor which can multiplex 65,536 channels over the four hardware links. In addition to the T9000 there is also a packet-switching router device, the C104. Together these alleviate the need to write multiplexing software when non-adjacent processors need to communicate.

This obviously leads to a much greater freedom of interconnection. In a farm this allows a farmer to communicate directly with all of the worker processors and alleviates the need for a harness. This allows for the fan out to be expressed directly in the farmer.

```
PROC farmer ([]CHAN OF REQ req, []CHAN OF JOB job)
  SEQ j = 0 FOR Jobs
    ALT i = 0 FOR SIZE req
      req[i] ? any
        job[i] ! work.item[j]
:
```

The C104 routing device

Further to the virtual channel routers, message routing devices have also been developed. These route messages directly. Being devices separate from the processing elements, these allow for the communication requirements to be scaleable separately from the amount of computing requirements of an application. This is a great enhancement as the amount of communication bandwidth needed can grow more rapidly than the amount of computation being performed.

These routing devices, C104s, are programmable 32 way routing switches. These work at the same high speed as the T9000 and thus have very high rates of packet throughput and introduce only minimal latency. C104s may be cascaded to any depth, providing whatever amount of interconnectivity is required. C104s provide both wormhole and grouped adaptive routing.

6.3.3 Resource channels

The second added communication facility is the provision of many-to-one communication channels. Here one end of a channel is shared between a number of processes.

The mechanism for resource channels works as follows. The processes on the channel's shared end make *claims* for the shared end of the channel. As there is obviously a number of these processes the underlying system queues these claims. Each claim for the channel must therefore be *granted* when the process on the fixed end of the channel is prepared to interact with a channel. Thus the programmer makes the grant explicitly for the next channel in the queue and the two processes can then communicate.

For a farm this clashes slightly with our intuition. In a farm the farmer is a client sending out data to be processed and the workers are the servers taking in data, processing it and producing results. The view of the resource channel's mechanism is the other way round to this — the workers are clients demanding work from the farmer, which is the server providing work. The workers also act as clients pushing results to the harvester, which acts as the server to absorb them.

6.3.4 Implementation

These two additional mechanics open up new approachs to implementing farming. While the existence of virtual channels alleviates the need for farming harnesses, and thus generally less code to be written, the addition of shared channels will require the code that is written to be of a form different from that used for the first generation of transputer. The extra support, in terms of language constructs, for resource channels as implemented on the T9000 will be provided by shared channels in occam 3 [Bar92].

These are shared as a record, opposed to individually, and are declared as follows,

```
CHAN TYPE WORK
  RECORD
    CHAN OF INT job:
:
CHAN TYPE REPORT
  RECORD
    CHAN OF INT result:
:
```

Then, just as before, channels are declared of the appropriate type over the scope of the parallel processes that communicate using the channels.

```
PROC farmer (SHARED WORK work)
  SEQ j = 0 FOR Jobs
    GRANT work
      work[job] ! generate.job(j)
:

PROC worker (SHARED WORK work, SHARED REPORT report)
  WHILE TRUE
    SEQ
      CLAIM work
        work[job] ? j
      application (j, r)
      CLAIM report
        report[result] ! r
:

PROC harvester (SHARED REPORT report)
  SEQ r = 0 FOR Jobs
    GRANT report
      report[result] ? result[r]
:

SHARED WORK work:
SHARED REPORT report:

PAR
  farmer (work)
  PAR w = 0 FOR Workers
    worker (work, report)
  harvester (report)
```

As the mechanics of the claiming and the granting provides the necessary multi-way selection needed, there is no need to construct this claiming of the work, by the programmer, through requesting as is performed on the first generation of transputers.

Parallel version

The above is the structure of a basic processor farm written using shared channels. However, although a farming harness is no longer needed, buffering messages will still improve the efficiency of an implementation, as buffering both prevents the workers from dealing directly with links and in the case of job supply also ensures that the next job is always in the processor's local memory.

```
PROC worker (SHARED WORK work, SHARED REPORT report)
  ... variables
  SEQ
    CLAIM work
      work[job] ? j0
    PAR
      CLAIM work
        work[job] ? j1
      application (j0, r0)
    WHILE TRUE
      SEQ
        PAR   -- get work0, give result0, process job1
          CLAIM work
            work[job] ? j0
          CLAIM report
            report[result] ! r0
          application (j1, r1)
        ... get work1, give report1, process job0
:
```

If the time to generate a job is longer than the time to communicate the job across a link it is worth performing these two actions in parallel; the same applying to result collection. This is probably best performed with a separate buffer sitting on the link.

```
PROC farmer (SHARED WORK work)
  PROC generate (WORK out)
    SEQ i = 0 FOR n
      out ! generate.work.packet(i)
  :
  PROC farm.out (CHAN OF WORK in, SHARED WORK out)
    WHILE TRUE
      SEQ
        in ? j
        GRANT out
          out[job] ! j
  :
  CHAN OF WORK raw.work:
  PAR
    generate (raw.work)
    farm.out (raw.work, work)
:
```

6.3.5 Will these new mechanisms result in faster farms?

There is the question of whether these new mechanisms can result in faster implementations than building the farm mechanics by hand.

These extra mechanisms are implemented in hardware, which is invariably can do things faster than software. Further, both the virtual channel processor and the C104 router devices are separate from and operate in parallel with the main processor, and thus should result in quicker execution: the main C.P.U.s are free to execute work.

The resource channel mechanism is implemented in hardware by instructions in the processor. Thus, the argument about hardware being potentially faster than software applies (the argument about the mechanism resulting in faster implementations through being separate does not).

There is one more aspect of the resource channel mechanism. As already mentioned earlier, it is implemented by queuing the claims. This queue is managed by the processor on which the fixed end of the

channel resides. This centralises the processing of this resource. With the first generation this load (of allocating jobs to workers) was distributed over most of the transputers (through the use of a network of job distribution processes) and the farmer only gave out work on one channel. Thus, here the processor that is potentially the largest bottleneck of the implementation (performing any necessary management of the jobs) also has to perform all the details of job allocation. Although this only involves managing a queue, this bottleneck will reduce the maximum number of workers a farmer can farm to for any given granularity. So, although the allocation of jobs may be performed quicker than with the previous generation, the implementation used is not ideal as it can restrict the maximum size of a farm used.

6.3.6 No automatic link demultiplexing

Unfortunately, the T9000's virtual channel processor does not automatically demultiplex communications across more than one link, even if a number of links are used in an implementation.

Thus if a particular implementation is communication bound utilising multiple links will have to be performed by hand. Just as with the first architecture, this is possible by doubling, tripling or quadrupling the number of workers per processor. The extra set of channels being placed on different links. This increases the communication bandwidth of the farm while the maximum potential processing capacity remains the same.

This also alleviates the need for link buffers on T9000 farms. As there is more than one worker process per processor, the four worker processes are all that is needed. As two link communications are in progress for two workers, one for input and one for output, another worker can be computing a job. Thus the C.P.U. is always active, as long as there is work to be performed. As well as increasing the compute/communication ratio of the implementation this also results in a more efficient execution overall. There are just worker processes to execute, no buffers. There are also no memory requirements for the pairs of buffers (one in, one out) that otherwise would be needed.

```
PROC farmer ([4]SHARED WORK work)
  SEQ j = 0 FOR Jobs
    PRI ALT i = 0 FOR 4
      GRANT work[i]
        work[i][job] ! generate.job(j)
:

PROC quad.worker ([4]SHARED WORK work, [4]SHARED REPORT report)
  PAR w = 0 FOR 4
    worker (work[i], report[i])
:
```

When these processes are placed onto processors, each shared channel will need to be placed onto a separate link of each transputer.

6.3.7 Updating the workers' state

Some computations consist of a number of sets of work, where each set needs to be computed in a different state. For example, when ray tracing an animation the contents of the scene for each frame will be slightly different as the characters move about, including in and out of frame.

In order to farm out such computations a mechanism is needed so the different states can be distributed to the workers. Here we look at how this has been implemented on the first generation of transputer and how it is best implemented on the second generation.

First generation of transputer

The method implemented by this author has been to broadcast the state change via the job distribution harness. This approach avoids sending the state information with every job. This method works by using the tagged protocol,

```
PROTOCOL WORK
  CASE
    job; ...
    state.update; ...
:
```

The farmer can then send new state information to the workers before farming out each set of jobs,

```
PROC farmer ([]CHAN OF REQ req, []CHAN OF WORK work)
  ... variables
  WHILE TRUE
    SEQ
      PAR i = 0 FOR SIZE req
        SEQ
          BOOL any:
          req[i] ? any
          work[i] ! state.update; state.info
      SEQ j = 0 FOR Jobs
        PRI ALT i = 0 FOR SIZE req
          req[i] ? any
            work[i] ! job; generate.job(j)
:
```

the harness distributing these state update messages to all the workers,

```
PROC job.distributor (CHAN OF REQ req, CHAN OF WORK in,
                      []CHAN OF REQ worker.req,
                      []CHAN OF WORK worker.job)
  ... variables
  WHILE TRUE
    SEQ
      req ! TRUE
      in ? CASE
        job; j
          PRI ALT i = 0 FOR SIZE worker.req
            worker.req[i] ? any
              worker.job[i] ! job; j
        state.update; state.info
          PAR i = 0 FOR SIZE worker.req
            SEQ
              BOOL any:
              worker.req[i] ? any
              worker.job[i] ! state.update; state.info
:
```

There are two main advantages to communicating state information along the same channels used to distribute the jobs. First, messages cannot overtake one another. This preserves the order of the messages as they travel through the farm and consequently jobs are executed in the correct context and correct results are produced. Second, the level of performance is higher: job messages are smaller as they consist of only the job, also, on farms where some workers are not directly connected to the farmer (on long linear topologies for example) jobs from a number of sets of work are executed simultaneously. There is only a minor reduction in performance while state update messages propagate down the harness and the farm gradually changes from computing one set of jobs to computing the next. This approach requires that the harvester can tell the different between the different types of result, but the performance advantages are worth it.

The disadvantage of this method is all processes that communicate jobs must contain the code to deal with all parts of the protocol. This consumes programmer time and memory, especially when many different parts of the workers' state can be updated separately.

Second generation of transputer

The introduction of shared channels changes the way work is farmed out. In particular shared channels cannot be used for broadcasting and a new method for distributing state information will need to be found.

With Welch a new method has been developed. Instead of sending out the new state when the next set of work is to be started, this method works by only sending out the new state when workers with an older state request work.

This can be implemented as follows. When each worker requests a new job, it informs the farmer of the state it is using. If this state is not the current one, the farmer sends the latest state as well as the job to be performed.

The worker will not know whether it will be expecting a state or a job first and it is more efficient to execute a CASE-input on a tagged protocol than to ALT over two or more channels. Thus, we will have a tagged protocol that can communicate either a job or both a state and job,

```
PROTOCOL WORK
  CASE
    job; ...
    state.job; ...
:
```

We now need a channel type to communicate both the above (from the farmer to the worker) and some form of state identifier (in the opposite direction),

```
CHAN TYPE STATE.WORK
  RECORD
    CHAN OF INT state.label:
    CHAN OF WORK send:
:
```

We can then write the farmer and the worker,

```
PROC farmer (SHARED STATE.WORK work)
  ... variables
  SEQ
    current.state := 0
    WHILE TRUE
      SEQ
        ... initialise next state
        current.state := current.state + 1
        SEQ j = 0 FOR Jobs
          GRANT work
            SEQ
              work[state.label] ? worker.state
              IF
                worker.state = current.state
                  work[send] ! job; generate.job(j)
                TRUE
                  work[send] ! state.job; state.details;
                                        generate.job(j)
:
```

```
PROC worker (SHARED STATE.WORK work)
  ... variables
  SEQ
    state.number := 0
    WHILE TRUE
      SEQ
        CLAIM work
          SEQ
            work[state.label] ! state.number
            work[send] ? CASE
              state.job ? ...
                ... perform any initialisation
              job ? ...
                SKIP
        ... process job
        CLAIM report
          report[result] ! r
:
```

There are a number of advantages to this method and the approach it takes to communication. First, workers are only sent the most recent state. This way workers that have been working a long time on their previous job only receive the most recent state and no time is spent receiving state update information that is now out-of-date. With the solution used with the previous generation of transputer, every worker received every state communication, even if a worker did not work on jobs to be processed in that state. Second, it easily allows for a dynamically variable farm size as the farmer does not actually need to know the number of workers in the farm.

Perhaps the two most important advantages of this method are that, first, it uses the mechanics of the shared channel and thus does not introduce any further channels or arrays of channels. Second, this method allows different workers to work on jobs from different sets of work at the same time, as was achieved on the previous generation of transputer. The farm does not have to wait for the whole farm to stop processing the previous set of work or send the state information with every job.

This method allows for a process to make a decision about what it should send another remote process. This is achieved by communicating information that either is directly from a process's state, or, as in this case, is an identifier that represents the state.

What is most interesting about this method is the approach to communication it uses. Instead of using a single brute-force broadcast to communicate state information as soon as it becomes current, the updating is performed on an individual basis. It is because the farmer is now communicating with workers on an individual basis that allows for not just one way, but interactive communication. It is this that gives rise to all the advantages such as updates only being communicated when they are required.

In conclusion, this interactive approach to communication, aided by a shared (and virtual) channel, results in a farmer that is sensitive to the individual requirements of each worker and thus can respond to these requirements accordingly. By using this interactive approach to communication it is hoped that other solutions can be developed that will provide similarly flexible, subtle and responsive designs that are thus equally effective and efficient implementations.

6.4 Automatically farming out processes

Farming out an arbitrary set of communicating processes on the first generation of transputer would involve considerable overheads and as a result static allocation will be more efficient.

On the architecture mentioned by Welch [Wel95] (also see subsection 5.9.3, page 175), although this architecture greatly widens the Von-Neuman bottleneck by sharing many memory nodes simultaneously, it also farms out the process queue (as is the case with any share memory multiprocessor machine). On architectures such as this it will always be just as efficient to execute an arbitrary set of communicating processes as it would be to execute a data decomposition.

In general, the best approach for the future, is to develop architectures that have dynamic allocate processes built into them. This is obviously best done by having (multiple processor) architectures that farm out the process queue automatically.

Here we look at how we should write programs for such architectures.

6.4.1 Programming

The greatest advantage in having application's farmed out automatically is we only need to develop a program as far as the set of parallel process to be executed, there being no longer the need to implement any methods of dynamic allocation. This removes both the need to separate out the application into farmer and worker and the need to implement a farming harness.

Thus, the ideal way to express all programs would be with as much as possible going on in parallel,

```
PAR
  .
  .
  .
  PAR i = 0 FOR n
    ...
  .
  .
  .
```

Many programs possess some dependencies between processes. Though as long as there is enough parallel slackness for the architecture, this being dictated by the architecture's design as well as the number of processes, the program will execute efficiently.

6.4.2 Parallel slackness and program granularity

One concern that is important, is whether the amount of parallel slackness required for efficient implementations will be greater than the amount of parallelism inherent in applications.

When farms on the first generation of transputers had more than one worker process per processor (harnesses E and F), it was likely that if a worker was without a job to process the other process would be able to continue. This illustrates that in order to achieve optimum efficiency some parallel slackness is required — the number of processes required will be in excess of the number of processors.

That said, expressing the application's parallelism should still be performed with some sensible degree of grain that is appropriate for the hardware in question; decomposing the two sub-expressions in,

```
a := (b + c) + (d + e)
```

across two processors, thus,

```
SEQ
  PAR
    bc := b + c
    de := d + e
  a := bc + de
```

will only gain any benefit if the setting up and closing down of the parallelism plus any communications have a lower cost than performing one of the sub-expressions.

6.5 Summary

In this chapter we have looked at some preliminary designs for implementing farms efficiently on the second generation of transputer and one type of future architecture.

Semantic preserving algebraic transformation laws have been very useful in this thesis. There is need to extend this work, this includes proving correct some transformations that have been used already.

Although the first generation is now well understood, there is still some work to be done. This consists mainly of fine tuning and the discovery of the few unknown details.

The second generation of transputer is now on the market. There are two new hardware communication mechanisms, thus some frequently used communications are now easier to implement. One provides a general message routing fabric alleviate the need for explicit communication harnesses. The other provides a new communication mechanism for two-way communication of the one-to-many variety. Interactive communication appears to have much greater design and performance advantages over brute-force broadcasting and this new one-to-many mechanism seems to encourage interactive communication and prevents that later. Testing also needs to be performed to find out the threshold above which farming out work is efficient.

Armed with this last piece of knowledge, implementation will again come down to knowing how to choose between the execution strategies.

Future architectures should use farming to automatically perform the allocation of processes to processors.

Chapter 7

Conclusions

This chapter contains some discussions on UNITY, the practical use of the work, and the order in which the two studies were performed. Finally, this thesis's contributions are discussed and its conclusions listed.

7.1 UNITY

Here we comment on how useful UNITY was in this work.

UNITY is a foundational theory with which to perform program development. This is independent of the fact that UNITY also suggests a formal approach to program development.

This author's opinion is that the most useful aspect of UNITY is that it separates program design from the implementation, and thus that programs should be designed before implementation issues are considered. This has been used here to realise that execution strategies are separate from applications.

It is interesting and reassuring that working with UNITY gave the same good design properties that C.S.P. also gave, namely that the farming out of work is best organised as a producer-consumer strategy.

In UNITY a design may be implemented on many different architectures. This highlighted the fact that if a program can be farmed, programs of a similar internal structure could be farmed as well.

7.2 Is this work of practical use?

Many of the first generation transputers have been sold and sales continue to increase — the T9000 having a different market. Thus, this work is increasing in its usefulness generally and the quantitative results presented are of practical use to the expanding domain of first generation transputer systems.

7.3 The order of the two studies

As discussed in chapter 3, it was decided to study implementation before the theoretical issues. This decision was made as it was realised that these, as yet unfarmed applications, may need to be executed on more efficient harnesses and topologies that would not be discovered yet for some time. This was the correct decision to make, as these additional applications will become farmable when program designs and mappings that produce non-final results are developed.

7.4 Contributions

Here we discuss the contributions of this research.

7.4.1 Major thesis contributions

The major contributions of the two studies performed in this research are listed below.

The major contributions of chapter 4 are that this research has identified the circumstance in which farming is efficient: the demand for work is outweighed by the supply. An equation has been developed that encapsulates the above. It has been showed that picking the most obvious execution strategy is not the only approach possible to implementation. It has also showed that it is possible to study extensively the breakdown of farming harness efficiency independently of any specific application. Using this it has showed the efficiency of a number of very efficient farming topology and harness combinations.

The major contributions of chapter 5 are that this research has developed a definition for the basic structure of the farming mechanism and some useful extensions. It has showed it is logically possible to farm out all applications expressed as independent processes. It has also clarified that farming out is just one way of dynamically allocating any set of parallel processes, whether data or computational independencies have been parallelised.

As a result this research provides a clearer and detailed understanding of: the farming mechanism, its range of use, and how to implement it efficiently.

The combination of the last two suggest that we should use farming more, as it is likely we can implement it efficiently for many applications.

This work is not the first to say that farming should be used more, however, it is original in both showing that farming can be used more widely and in indicating the limits of this.

We now look at the other significant contributions of each chapter.

7.4.2 Other contributions from chapter 4

Chapter 4 documents a study of some highly efficient harnesses and topologies. This includes finding the boundary conditions of this efficiency.

The other contributions made in this study have included the development of an equation that relates the number of workers, w, with the average job compute time, j, and message communication time, c,

$$w \leq \frac{j}{c}$$

where c may either be measured or calculated from the number of bytes in a message, m, the bandwidth, B, and a new method introduced here for measuring communication set up costs by using the number of bytes that would be communicated in the time it takes to set up the communication, s,

$$c = \frac{m + s}{B}$$

These can be combined to give,

$$w \leq \frac{jB}{m + s}$$

Other contributions also included recommending minimum values for m, so that as much as of the link bandwidth as possible would be utilised. Some values for B, the bandwidth around a processor farm, have also been found.

This research has also found that initialisation is not a special case and has showed tuning job size affects overall efficiency due to how well the farm finishing.

On a related note, chapter 6 discussed some farming harness designs for the T9000, the advantages of interactive communication over brute-force broadcasting (as encouraged by shared channels), and how processes can be dynamically allocated automatically using farming on future architectures.

7.4.3 Other contributions from chapter 5

Chapter 5 looked at the question: what applications can be farmed? This being both in terms of what range applications it is possible to farm and what range of applications are likely to be efficient.

By answering this question, the chapter raises yet more questions. As applications don't have to be implemented using the most obvious execution strategy, we can select the mapping that is the most appropriate (flexible, scalable, efficient). Thus, in order to implement these other applications (with dependencies between jobs for example) efficiently for whatever architecture is in hand, we need to develop more sophisticated mappings.

This study also made some other contributions as this part of the research has found that jobs can contain anything: data, or computational processes or both. It has showed some original mapping designs that could be used to implement computations on geometric data sets and recursive applications such as Hanoi and quicksort. It has also realised more mapping designs are needed. It also found that recursion was not use useful when attempting parallel program development.

7.5 Conclusions

Drawing these conclusions together, we are now armed with a much greater understanding of farming: what it is, what it can be used for, and how to use it efficiently.

We have also come across some good general implementation properties. We look at these first. Discussed next is how we should change the way we go about selecting an execution strategy. Lastly we discuss in depth the following aspects of farming: its structure, its advantages, where it can be used, and what makes a design, a mapping and an implementation efficient. We also look at the designs for the most efficient farming harnesses.

7.5.1 Implementation

Three conclusions apply to implementation in general.

Match Program Shape to Hardware Shape

The efficiency of the harnesses here are due to the accuracy with which their shape matches the design of the hardware on which they are executed. This architecture is not just a single processor that executes load, arithmetic and store operations; here there are four communication links as well as the C.P.U. Therefore the most efficient results are often obtained when the transputers are programmed to run the link engines separately in parallel.

```
PAR
  PAR i = 0 FOR 4
    link(i)
  cpu()
```

This is a general point that applies to all computer implementations not just processor farms implemented on transputers.

Communication structures should only communicate

Communication harnesses should only perform communication, i.e. they should perform no computation. Any selection a communication harnesses performs should be achieved solely through communication (i.e. `ALT`) and not via computation (e.g. `IF`). This excludes communication harness from deciding which worker a job should be sent to, such a decision should be performed within a master worker process while the communication harness is executing concurrently obtaining further work.

Miscellaneous conclusions

The experience gained here is that converting algorithms expressed sequential into a parallel form, such as occam is easier than this author was lead to believe. This includes implementing recursion on a stack by hand. This is generally believed to be difficult.

Recursion restricts our thinking when attempting to redesign an algorithm into a parallel form.

7.5.2 Selecting an execution strategy

For many applications, the execution strategy selected is one with an internal structure that matches the internal structure of the application. The usual motivations behind this decision are efficiency and ease of implementation.

This approach, however, fails to take into account the fact that an execution strategy may need to be appropriate for the architecture as well as for the application if it is to be efficient.

For any application it is possible for several executions strategies to execute it, even if only the type of allocation used can be varied. Thus, if it is possible to select the strategy with which the execution is arranged, it may be better to pick one that is known to be highly flexible and efficient.

There are two separate areas of implementation development: developing a parallel design and allocation. Implementation involves selecting the independencies between parts of the application and then allocating the resulting parallel processes. Which set of independencies are parallelised and which allocation method is used can be selected so as to be the most appropriate for the application, whether the most flexible, the scalable, or the most efficient implementation. Farming out is a dynamic method allocation.

For example, take an application that sends lots of data through a few stages of a graphics pipeline. This would normally be implemented by using the independencies between the algorithmic stages to create a set of parallel processes (that need to communicate intermediate results to one another) and then allocating them statically. However, this application could be parallelised by using the independencies between the different pieces of data and then allocating these processes dynamically, by farming out each graphical object.

7.5.3 Farming model

Farming consists of continually communicating the next job of work (from a bag) to a free worker that then performs that job. The bag and the workers are arranged as in figure 73 below.

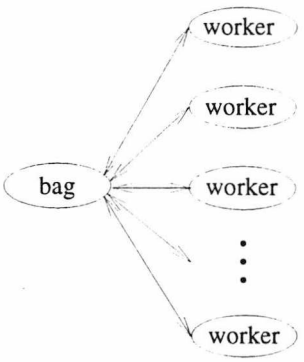


Figure 73: A model of farming out

Jobs may produce results and further jobs. Results produced should be dealt with as appropriate, invariably they are collected and output. Any jobs generated are returned to the bag. Thus keeping the mechanics of job distribution simple. This model's basic structure allows for variations *within* this structure for the situation in hand. A more detailed description of the model can be found in subsection 5.5.6 (page 158), which includes how the model can be varied.

7.5.4 Advantages of farming out

Farming out is a dynamic method of allocation that load balances work automatically. It is very simple and can also be reasonably easy to implement.

Farming's structure is independent of the structure of the application it executes. It can also be implemented easily on any architectural topology where there is a single path from the bag to each worker, thus making it very flexible. Even the number of workers can be varied dynamically.

Farming's only overhead is the communicating of jobs from the bag to the workers (this includes the overheads of starting up the communications). If these communications can be performed in parallel with the work, and can be made to be small, farming can be very efficient. With a good mapping of a constantly parallel design and suitable grain, up to 99% efficiency has been obtained here. The lowest efficiency obtained in chapter 4 for suitably mapped applications was 90%.

7.5.5 What applications can be farmed

It is possible to farm all computations that can proceed independently (i.e. without interfering or sharing memory with one another).

It is also possible for these computations to communicate with one another by exchanging messages. How this is achieved is dependent upon the mechanics available on the architecture. Processes may just need to be descheduled while waiting for the synchronisation to be performed, or they may be decomposed into their non-communicating components and the interactions are then performed via the process managing the bag.

Two major consequences of this are: that the workers may perform more than one type of job, and that the jobs farmed out may produce intermediate results and further jobs, not just final results.

Despite it being logically possible to farm out all these types of work, not all implementations will be highly efficient. All farmed computations should be efficient if,

1. there is more parallel work than there are workers,
2. this parallelism can be decomposed so there is a reasonably continuous supply of jobs,
3. the supply of work is greater than the demand generated by the number of workers used, and,
4. this decomposition is of a granularity that is above that of the architecture.

In short, farming is efficient if there are things to do in parallel, there needs to be parallelism within the problem, the architecture and this parallelism needs to be preserved in the design and mapping stages in between. As which architecture used will affect efficiency some applications will only be efficient on some architectures. The amount of parallelism within the application also limits the amount of scalability within the implementation.

7.5.6 Developing a mapping

Mapping a program design onto an architecture consists of two stages,

1. selecting some of the independencies between some of the separate parts of the design, and,
2. selecting a method with which to allocate this parallel processes.

The first stage consists of selecting either the separate items of data or the separate stages of the computation, and making sure the granularity of the processes is appropriate for the architecture, so that the application will be executed efficiently by the architecture.

The second stage will involve looking at the communication performance and start up time of the architecture to see whether a dynamic method of allocation will be more efficient.

These two stages can be thought of separately. However, when considering efficiency, the decisions made at both stages may need to be considered together, as some mappings will be more efficient when allocated dynamically, others will be more efficient when allocated statically.

Dynamic allocation may be less efficient if the work has to be organised with a fine level granularity. When there are many dependencies between the stages of the computation, if the work is to be allocated dynamically, many communications will need to be introduced and the computation may need to be decomposed into a number of separate independent jobs. This fine level of granularity will introduce a large number of communication overheads and thus allocating the processes statically may be more efficient.

If a computation is larger than the amount of memory available per worker, it remains to be seen whether it is better to decompose the computation and farm out these processes, or decompose the computation and have a network of farms.

The messages communicated may consist of either data, a process, or a combination of the two.

7.5.7 The factors that affect an implementations efficiency

Implementations should be efficient. This is achieved by ensuring the workers can and are constantly advancing the computation. Arriving at a mapping that is efficient may take a few attempts.

Farming out is about supply and demand. Thus, a compute bound mapping is one where the work supply is greater than the capacity for the farm to perform work. This is achieved by arranging for the jobs to take less time to communicate than they do to compute. With the average length of time it takes to perform a job, j, and the average length of time to communicate a job or result message, c, should be selected so that the ratio between them is as large as possible.

$$c \ll j$$

This allows the farmer process to farm out as many jobs as possible in the time it takes to perform a job. This in turn allows for as many workers as possible. In terms of efficiency, this means that the maximum number of workers, w, that an application can use is dictated by the number of jobs an implementation can supply and the length of time they take,

$$w \leq \frac{j}{c}$$

not just the amount of parallelism within the application, the program design and the mapping.

The overhead of farming is that of having to communicate jobs. Thus, the time to compute jobs should be larger than the time to communicate them.

Further, the overhead of communication is that of having to start communications up. Thus, the time to communicate the body of a message, m/B, should be much larger than the communication set up time, s/B,

$$\frac{m}{B} \gg \frac{s}{B}$$

This is the average bandwidth to a worker. This value will be somewhere in between the bandwidth out of the farmer and to the last worker. Naturally the values of s and B are affected by the harness used. On the first generation of transputer, the raw bandwidth of a link used unidirectionally is 1.51 Megabytes per second, bidirectionally it is 2.19 Megabytes per second.

For the first generation of transputer, the communication set up time, s, is very small. This allows for mappings of a reasonably fine granularity. With large messages here, it has been possible to ignore s. That said, when using a harness the start up costs of communication go up. Thus, if the size of the message can be selected, it should be at least 32 bytes, and preferably 64 bytes or more. Doing this makes a mapping more likely to be compute bound.

Another way of keeping the start up costs of communications down is to always transmit contiguous messages, `[2]INT`, rather than `INT; INT`.

If the size of the jobs is highly selectable (such as when farming out many jobs in one message) then, once a compute bound mapping has been found, adjusting the size of the jobs can be used to find the optimum run time by balancing between larger jobs that result in fewer jobs farmed out overall and smaller jobs that result in a smoother finish.

To initialise a farm requires filling all the workers and all the harness job buffer spaces with jobs. If the farmer can supply a greater supply than there is demand, it takes,

$$\frac{\text{number of slots}}{\text{excess supply}}$$

units of time to initialise a farm. If there is not much excess supply, it could be slightly beneficial if some jobs are output to the farm at an increased rate.

7.5.8 Farming harnesses

The most efficient farming harness for the first generation of transputer consists of the workers in a tree topology. One link is connected to the farmer and the other workers above it in the tree, the remaining

three links (thus making a ternary tree) are connected to the workers further down. The processors have a communication process on each hardware channel (i.e. two per link) that pass pointers to messages between themselves and the worker process. If the messages vary in size, buffers should be added to decouple the link processes from one another and allowing each process to respond to external communications more rapidly.

As it is not currently known how to easily set up a ternary tree with the last layer of workers only partially populated, if the number of workers to be used is to be varied, it may be more appropriate to use an efficient line topology.

The most efficient line topology has a harness consisting of a simple process to pass on jobs and the same to pass on results. For compute bound mappings, this harness is also only about 2% less efficient than the ternary tree. However, due to not greatly using the parallelism of the transputer, this harness can only supply about half or a third of the number of jobs as the other harnesses. Thus, a lower value of B will need to be used when performing calculations with the model.

The second most efficient line topology has two (separate) harnesses, each with its own worker process. Each harness passes messages in each direction along the line of processors. Each harness has a process on each hardware channel as with the ternary tree (eight harness processes in all). Also like the tree harness, these processes pass pointers to messages between themselves and their worker process. For compute bound mappings, this design is only about 4% less efficient than the ternary tree topology, however its level of supply is much higher and thus it breaks down even slightly later than the ternary tree.

One disadvantage of a line topology over that of a tree topology is that, due to performing $w/2$ communications instead of $\lceil \log_3 w \rceil$, the cumulative communication set up costs are much larger and small messages obtain a smaller percentage of the available maximum throughput.

Which worker a job is given to in times of choice is not a matter that affects performance in one way or the other.

7.5.9 Summary

There are a number of situations where a computational task is too large to be performed by a single individual. If this is the case the task can be performed by a number of workers (people or computers) can work on the individual parts of the task. Whether these parts be separate data items or the separate parts of the task's stages.

There is a strong likelihood that the task will be performed more efficiently, and thus more rapidly, if the work is allocated dynamically as the workers become free. Farming out is one method of dynamic allocation that is worth considering, due to it having only very light overheads (only a bag for the work and a distributing communication mechanism need to be implemented) and through being very simple (work is dynamically allocated from the bag to the workers, as a result the work load is balanced automatically).

Farming should be able to execute any application efficiently if the application contains a reasonably continuous amount of internal parallelism, and if it takes longer to process a job (j) than it does for all the workers (w) to have jobs allocated to them (c),

$$j \geq wc$$

Using a granularity larger than that of the architecture's set up costs can aid this.

When a large number of workers is required an interconnection topology is needed to connect all the workers to the bag. A communication harness is then needed to provide the distributed interconnection. Some of the most efficient combinations for the first generation of transputer are included here.

Farming is a simple and efficient method of process allocation. Thus, it should be used whenever an application's sizeable and constant amount of parallelism can be mapped onto an architecture such that the overheads of allocation are small. One method of reducing these overheads would be to develop parallel architectures where the hardware performed the dynamic allocation. This would significantly extend the range of (especially imbalanced algorithmically) parallel applications that could be executed more efficiently, it would also remove the burden of writing allocation software from the programmer. The most appropriate method of dynamic allocation that would be simple for hardware to perform, and yet be powerful and effective, is likely to be farming.

Bibliography

[And91] Gregory R. Andrews. *Concurrent Programming*. Benjamin/Commings, 1991.

[Bar92] Geoff Barrett. *occam 3 reference manual (draft)*. Inmos, March 1992.

[Bar93] Geoff Barrett. How to write a highly parallel program. In Jon Kerridge, editor, *Transputer and occam Research: New Directions*, pages 209–217. IOS Press, 1993.

[BCK93] Bożena Bartoszek, Zbigniew J. Czech, and Marek Konopka. Parallel searching for a first solution. Technical Report 8/93, Computing Lab., University of Kent, Canterbury, CT2 7NF, England, September 1993.

[Bro94] N. Brown. A sound mapping from abstract algorithms to occam programs. In H. R. Arabnia, editor, *Transputer Research and Applications 7*, pages 218–231. IOS Press, 1994.

[BTU88] R. D. Beton, S. P. Turner, and C. Upstill. A state-of-the-art radar pulse deinterleaver—a commercial application of occam and the transputer. In Charlie Askew, editor, *occam and the Transputer—Research and Applications*, pages 145–152. IOS Press, 1988.

[CM87] K. Mani Chandy and Jayadev Misra. *Parallel Program Design—A Foundation*. Addison Wesley, 1987.

[CU90] I. Cramb and C. Upstill. Using transputers to simulate optoelectronic computers. In Stephen J. Turner, editor, *Tools and Techniques for Transputer Applications*, pages 50–58. IOS Press, 1990.

[DH90] Keith R. Dimond and Samir Hassan. Incremental behavioural simulation on a network of transputers. In Stephen J. Turner, editor, *Tools and Techniques for Transputer Applications*, pages 223–231. IOS Press, 1990.

[Dij82] Edsgar W. Dijkstra. *Selected Writings On Computing: A Personal Perspective*, chapter EWD608, pages 264–267. Springer Verlag, 1982.

[Dij89a] Edsgar W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1397–1414, December 1989.

[Dij89b] Edsgar W. Dijkstra. *Selected Writings On Computing: A Personal Perspective*, chapter EWD501, pages 132–140. Springer Verlag, 1989.

[Dij89c] Edsgar W. Dijkstra. *Selected Writings On Computing: A Personal Perspective*, chapter EWD464, pages 79–83. Springer Verlag, 1989.

[Gen65] F. Genuys, editor. *Programming Languages*, pages 43–112. Academic Press, 1965.

[Gri71] David Gries. *Compiler Construction for Digital Computers*. Wiley, 1971.

[Gro] National Algorithms Group. N.a.g. fortran library manual, mark 15, np2136/15.

[HJ89a] C. A. R. Hoare and Cliff Jones, editors. *Essays in Computing Science*, chapter 16, pages 259–288. Prentice Hall, 1989.

[HJ89b] C. A. R. Hoare and Cliff Jones, editors. *Essays in Computing Science*, chapter 1, pages 1–18. Prentice Hall, 1989.

[HJ89c] C. A. R. Hoare and Cliff Jones, editors. *Essays in Computing Science*, chapter 4, pages 45–58. Prentice Hall, 1989.

[HJ89d] C. A. R. Hoare and Cliff Jones, editors. *Essays in Computing Science*, chapter 2, pages 19–30. Prentice Hall, 1989.

[Hoa61] C. A. R. Hoare. Partition, quicksort and find. *Communications of the ACM*, 4(7):321–322, July 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *BCS Computer Journal*, 5(1):10–15, January 1962.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–80, October 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, August 1978.

[Hoa80] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1980.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Inm88] Inmos. *occam 2 reference manual*. Prentice Hall, 1988.

[JG91] Gerraint Jones and Michael Goldsmith. Formal methods applied to occam workshop. 14th World occam and Transputer User Group, Loughborough, Programming Research Group, 33 Keble Road, Oxford., September 1991.

[JR92] D. W. Jenson and D. A. Reed. A performance analysis exemplar: Parallel ray tracing. *Concurrency: Practice and Experience*, 4(2):119–141, April 1992.

[Kru87] Robert L. Kruse. *Data Structures and Program Design (second edition)*. Prentice-Hall International, 1987.

[LS90] Calvin Lin and Lawrence Snyder. A comparison of programming models for shared memory multiprocessors. In *International Conference on Parallel Processing*, volume 2, pages 163–170. Addison Wesley, 1990.

[Man82] Benoit Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co., 1982.

[MMF87] V. Martorana, M. Migliore, and S. L. Fornili. Molecular dynamics simulation of lennard-jones systems: parallel implementations on transputer arrays. In Traian Muntean, editor, *7th occam Users Group and International Workshop on Parallel Programming of Transputer based Machines*. Addison Wesley, 1987.

[Mor93] David Morse. Spatial simulation modelling of insect population dynamics on a transputer network. In Jon Kerridge, editor, *Transputer and occam Research: New Directions*, pages 66–75. IOS Press, 1993.

[MS87] D. L. McBurney and M. R. Sleep. Experiments with a transputer-based diffusion architecture. In Traian Muntean, editor, *7th occam Users Group and International Workshop on Parallel Programming of Transputer based Machines*. Addison Wesley, 1987.

[PC91] Iain Phillips and Peter Capon. Strategies for workload distribution. In Janet Edwards, editor, *occam and the Transputer—Current Developments*, pages 39–51. IOS Press, 1991.

[PR86] Heinz-Otto Peitgen and Peter Richter. *The Beauty of Fractals*. Springer Verlag, 1986.

[PZ90] Werner Purgathofer and Michael Zeiller. Configuring transputers for ray-tracing. In Len Freeman and Chris Phillips, editors, *Applications of Transputers 1*. IOS Press, 1990.

[RH86] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. Technical Report PRG-53, Oxford University Programming Research Group, February 1986.

[Roe] Herman H. Roebbers. *Advanced Transputer Engineering Workshop*. University of Twente and University of Kent, Computing Lab., The University, Canterbury, Kent. CT2 7NF.

[Sar89] Amir Mansour Sarrafan. *Transputer Models for High-Performance Bridges in Local Area Networks*. PhD thesis, University of Kent at Canterbury, November 1989.

[SS91] Shane S. Sturrock and Ian Salmon. Application of occam to biological sequence comparisons. In Janet Edwards, editor, *occam and the Transputer—Current Developments*, pages 181–190. IOS Press, September 1991.

[Stu91] Shane S. Sturrock. Biological sequence comparisons on a transputer network. Master's thesis, University of Kent at Canterbury, October 1991.

[SW90] A. M. Sarrafan and P. H. Welch. Transputers models for a high-performance local area network bridge. In Stephen J. Turner, editor, *Tools and Techniques for Transputer Applications*, pages 111–121. IOS Press, 1990.

[TD90] R. W. S. Tregidgo and A. C. Downton. Processor farm analysis and simulation for embedded parallel processing systems. In Stephen Turner, editor, *Tools and Techniques for Transputer Applications*, pages 179–189. IOS Press, 1990.

[Tie93] Ivor Tiefenbrun. Manufacturing in the future. *RSA Journal*, CXLI(5441):549–557, July 1993.

[Uni33] Oxford University. Shorter oxford english dictionary (7th edition), 1933.

[Uni93] Oxford University. Shorter oxford english dictionary (8th edition), 1993.

[W+89] Peter H. Welch et al. Evaluation of a multi-user transputer environment. Technical report, Computing Lab., University of Kent at Canterbury, March 1989.

[Wel] Peter H. Welch. *occam and Transputer Engineering Workshop*. University of Kent, Computing Lab., The University, Canterbury, Kent. CT2 7NF.

[Wel88] Peter H. Welch. The occam approach to transputer engineering. In *Third Conference on Hypercube Concurrent Computers and Applications*. ACM, 1988.

[Wel89] Peter H. Welch. Graceful termination — graceful resetting. In *Applying Transputer-Based Parallel Machines*. Addison Wesley, 1989.

[Wel95] Peter H. Welch. Parallel hardware and parallel software: a reconciliation. In Peter Fritzson and Leif Finmo, editors, *Proceedings of the ZEUS'95 & NTUG'95 Conference, Linkoping, Sweden*, pages 287–301. IOS Press, May 1995.

[Wil91] Colin Willcock. *X-Windows Programming In The Large*. PhD thesis, University of Kent at Canterbury, November 1991.

[WJW93] Peter H. Welch, George Justo, and Colin Willcock. High-level paradigms for deadlock-free high-performance systems. In Grebe et al., editor, *Transputer Applications and Systems '93*, pages 981–1004. IOS Press, 1993.