Farming: towards a rigorous definition and efficient transputer implementation

Warren Day, Steve Hill
University of Kent

Abstract

The technique of the processor farm has become very widely used for parallelising applications, often being mentioned without reference to any source.

The goal of this work has been to put together a complete and rigorous understanding of what the technique can be used for and what is needed in order to arrive at an efficiently farmed application. This paper consists of these two parts.

We have shown, via the UNITY theory of programming, that the basic structure of the processor farm may be used to parallelise a much wider domain of applications than has generally been considered.

Second, we show by example, how to build efficient implementations for the first generation of INMOS Transputers. This work is new in that it is the first that has been able to test farming harnesses by taking an abstract view of the application.

This paper has been written in a semi-"instruction manual" style. Also it should serve as an introduction to the subject.

1 Introduction

The processor farm was proposed by May and Shepherd in [MS87] (also in [INM87]). Since this original piece of work there have been a large number of papers that have dealt with the technique in one context or another. These range from documenting an implementation that used farming at one end to performing a study of farming at the other. With there being many more of the former than of the latter. However, there are a few papers where the work done in optimising a farming harness have resulted in some interesting discoveries. Thus some of this work here is not new.

Formal methods because of their thoroughness are very good at clarifying and making precise our thinking. Here we have used UNITY [CM87] to explore the types of programs that can be parallelised using farming.

Our work with the UNITY theory of programming will be merely to look at the execution model and then to view our programs in this perspective. As the execution model does not use any mathematical logic, the reader not familiar with any formal methods may also use the UNITY perspective to view programs.

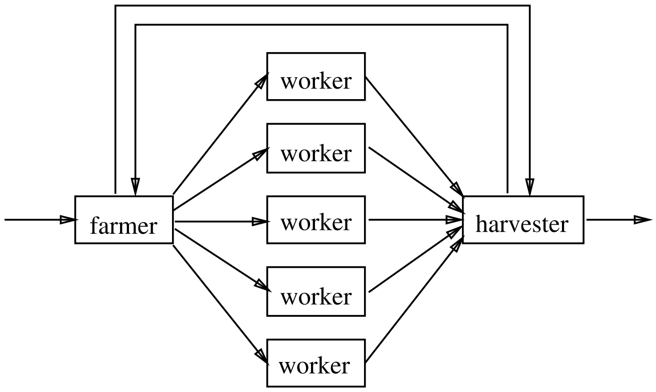
On a similar note, we have learned from the area of formal program construction that it is advantageous to view programs in terms of the properties they possess. When it is useful, we will use properties in our discussions.

We start by looking at what farming is, then we look at what it can be used for, and lastly how to implement it well.

2 Farming

The basic concept of farming consists of having a central controller that hands out *jobs* of work to be processed by the members of a pool of *workers*. It has become common practise to call the controller process a *farmer*. Jobs can take two types, the first is that of a job number and the second is that of a packet of data. We will refer to the overall computation to be performed as the *task*.

Sometimes it is desirable for the controller to exist as two processes, one for issuing the jobs of work and the other for receiving the results. If the destination process for the results is different from the source this second process is called the *harvester*. The diagram below shows this logical shape of a processor farm.



When implementing a processor farm, as the first series of transputers possesses a smaller degree of connectivity than the logical model. A communications system will be needed which achieves the same logical connectivity on the limited fanout of the hardware. This communications system is named the harness. This consists of some extra processes running on the various processors passing messages between some combination of the processes in the logical model and the other processes in the harness. As the second series of transputer possesses virtual channel routing and a hardware communications system this logical model can be implemented directly.

3 The uses of Processor Farming

UNITY is a theory that has been put forward to provide a foundation for programming. A program design can be developed and examined before it is implemented onto an architecture. The theory consists of an execution model and an associated program correctness proof system. Here we are only interested in the former.

One of the programming constructs in the UNITY programming notation is quantification. As with a `for` loop, quantification is used for generating a set of statements from a template. A block that also has some of the variables which are used as index variables.

The UNITY execution model for "running" the program is a very simple continual process, a statement is selected at random and executed. This process continues, one statement at a time, indefinitely. Thus, there is no notion of control flow. The statement selection process has a notion of fairness in that "all statements are executed infinitely often".

A UNITY program may be "mapped" from being a UNITY program, to an equivalent program that implements the same specification. This program can be executed on a real architecture for execution in the real sense of the word. The abstract execution model of UNITY being replaced by a more concrete one such as the sequential method of conventional computers.

In light of the UNITY approach to program execution, a parallelisation technique, such as farming and pipelining, that distributes a computation over a number of communicating processors really amounts to an execution strategy.

Quantification was discussed above as it is the only method in UNITY by which sets of similar calculations can be generated. Thus any application that has been farmed would be expressed using quantification.

However, can all quantifications be farmed? Traditionally, the sets of similar calculations must be computationally independent of one another but with a bit of insight we can see that this restriction can be weakened.

For example, as the jobs return from being processed, the results could be farmed out again as an iterative computation. With the results being jobs with either the same or different types to the original job. When the jobs are of the same type iterations of the same task can be performed with any necessary synchronised operations being performed in between. Alternatively if the types are different then a second type of processing could be performed. This need not be performed by a different processor farm. By all of the work being performed by the same farm the entire work load becomes completely balanced. So there is no load balancing of the application to be performed in its development, the moving of a number of processors from one processing farm to another, as all load balancing is performed at run time. Of course all processes need to have the code to be able to execute all types of jobs, but this may not be a problem. Many farmed applications are not necessarily large at the job level it is just that there are so many jobs to be processed. This iterative approach may be useful if some form of synchronisation is required between computations. Often this synchronisation is be of a nearest-neighbour variety. An example of an iterative computation that could be farmed in this way is bubble sort. Pairs of elements are sent out as jobs to be processed. The processing in this case being comparison and swapping. Upon return the elements are paired up with their other adjacent neighbour and these are sent out to be potentially swapped. If this is repeated the appropriate number of times the data will be sorted. For a sufficiently long and also variable times of comparison this approach would actually be an ideal and efficient one, provided that the number of processors used was less than half the size of the data being sorted. In essence this example is farming out bubble sort's inner loop. Similarly one computation that can involve two stages is generating a three dimensional landscape plot of the Mandelbrot set [PR86]. Sections of the plane can be computed followed by pairs of scan lines could be sent out for the final rendering before display.

The jobs of the computations with non global relations can be sent out using one of two approaches. These are equivalent to depth-first and breath-first searches. In the first case as soon as the first handful of results are returned from the workers and all of the other results on which they rely are present then these jobs can be send out again for the second stage of processing immediately, taking preference over other primary jobs which have not been farmed out yet. The alternative is to complete one set of calculations and gather all of the results before starting the next set.

The advantages of the depth-first method are that as the results return there is no need for them to be stored anywhere, and thus there need be no or in some cases little provision made for them to be buffered or stored. However, this approach may bring up the problem that especially 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.

As has been already shown with the Mandelbrot set example, workers can be used to execute more than one type of job. This can be taken further. As opposed to just farming out one computation on a farm which takes several separate stages in one form or another. It is also possible to have two completely unrelated sets of jobs processed together on the same farm. As the two, or however many, tasks are completely unrelated the 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. It might be useful to have an application consisting of several parallel components that at any time may produce a request to the farmer, acting as a server, for some work to be performed. There is already one good example of a processor farm being used as one part of an application's implementation [BTU88]. Also there is a good example in [CU90] of a farm that computes different types of jobs and also uses the differences in characteristics to obtain a higher degree of speed-up.

As we can choose to have an extra process by having the harvester separate from the farmer, we can also have a number of harvesters, though this entails a more complicated collecting structure than is traditional. One use of this method could be for the processor farm to perform the generating of some computational result which could be displayed differently on different graphics monitors. Similarly, although it is highly advantageous to having a single central controller, there could be others; either as a hierarchy of processor farms [Inm88], or with workers generating fragments of work to be processed in a recursive manner.

Perhaps the largest insight this piece of work has provided us, is that the jobs could have some computational dependence between them. To clarify, the pixels of the screen for the Mandelbrot set can be computed individually. However, the iteration of the calculation on which the set is computed must be performed in order, the result of one calculation being the parameter for the next. The style of computation we are talking of here is where there is some form of middle group. If a task exists that has a non-cyclic directed graph of computational dependence, i.e. there must be some piece of work we can do first and know what we can do next and so on, then it is possible to farm out the work to obtain a speed-up. As always, the amount of parallelism possible is dictated by the application, and thus the number of processors usable. We may have to be careful of any buffering and the amount of it in our farming harness here and the way it behaves and effects the performance of the computation.

One obvious expansion from the domain of networking and remote procedure calls is that of setting the number of processors to be used to vary throughout the execution. However, due to the large overheads of, processor and program initialisation, and that of program storage and retrieval, it is unclear at present how the number of processors could be varied effectively and practically.

The processors we have considered here are of the message passing variety. This technique may be amenable to other multiprocessor architectures.

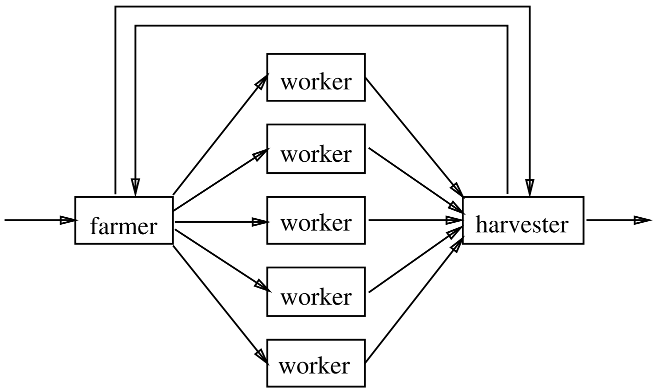
4 Efficient Processor Farm Implementation

Now we develop and examine the processes for a small variety of farming harnesses for the first series of INMOS transputers. We will arrive at some efficient processes that make use of the processor's built-in parallelism. The harnesses developed here are for classical processor farms, i.e. singlely executed completely independent jobs. Most of this knowledge on farming presented here is borne out of both theory and experiments. However, as we are attempting to bring together all of the necessary ideas in this one place, some of the ideas presented below are the work of others.

Our measure of efficiency here is a simple one, that of the quickest execution. Much of the theory here is to do with the basic characteristics of the transputer and how to use the processor's parallelism efficiently, and thus the approach here can be used to efficiently implement any type of system, including the more elaborate farming architectures described above.

From UNITY we know that the harness is separate from the application. However, the implementation of a farming harness is influenced by the data requirements of the application and shape of the network topology. In our tests we have abstracted away from the application by parameterising it by the aspects the farming harness sees: the size of the job packet, the time to execute a single job and the size of the result packet.

The author's work was performed in occam on our institution's MEiKO Computing Surface, but similarly efficient systems should be produced with any competent transputer system and compiler. Our tests have been of two varieties. The first has involved evaluating pieces of code running on a single processor. The second has been on full processor farms, of varying sizes, abstracted away from any application as described above. For all runs of our processor farms a task consisted of 100 jobs for each worker. This approach of issuing a number of jobs that is proportional to the size of all farms is so that the timings are to a small extent influenced by how well the farm starts and more importantly finishes. In order to make the processor's behaviour more realistic, the worker process polls the clock continually for the length of a job. The only thing artificial is that all jobs for any particular "application" took the same amount of time, thus it is not known how finely these harnesses load balance. All our times are in low priority clock ticks.



As we cannot directly implement the structure shown in our logical model above due to being restricted by limitations of the hardware, we need a harness. This comes in two parts: the topology, the shape of the interconnection at the placement level, and the communications processes at the software level.

4.1 Topology

First we will take a look at one aspect of implementing a processor farm that has not been given a great deal of serious consideration, the farm's topology.

Here is a list of properties we would like our farm to possess:

1. To keep communications to a minimum.

   In the ideal topology there is one channel between any two processes, therefore we would like to make any implementable topologies have an average interconnection as close to one as possible.

2. To keep all C. P. U. overheads to a minimum.

3. To use all of the communications bandwidth available.

4. Any buffering should only be present to aid performance, not to hinder it in any way.

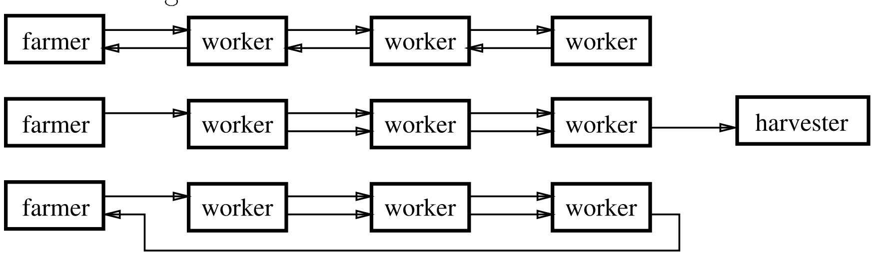
5. The harness should consist only of buffering that aids performance, and none that hinders performance in any way.

6. To have all processors in the farm working continually, so that no processors are starved of work.

7. A software harness that is as simple and easy to write as possible.

Solving most of these is quite easy. Keeping communications to a minimum can be achieved by having a high fan out of interconnection. Also having a good topology reduces the number of communications leaving the processors free for computation. Using the full bandwidth available is achievable by careful utilisation of the underlying hardware. As is shown latter very rarely does a sensible buffering mechanism get in the way. However any variables in the harness which hold data that is in transit act as a level of buffering. A parallel system only runs at less than full speed if important parts of the mechanism are prevented from proceeding by being forced to wait for resources.

With some topologies the last property is the most difficult 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 the last property, topologies should have the same harness code running on each transputer wherever the position in the topology. Thus what is ideally needed is a communications structure where one set of 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. This implies that from the point-of-view of the flow of the communications, the topology should look exactly the same from all places within it. The structure should be *self-similar* or fractal [Man82]. This is what we would ideally like to achieve, we will settle with being able to come close.



Pipes, rings and trees possess this property. However, this property does not apply to all even structures. For example grids, tori, hypercubes and others have a very uneven structure when viewed from the farmer's single viewpoint. Although it should be possible to develop a method 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 with one or two simple processes ([PC91] [CHvdV88]).

From the point of view of keeping communications down to a minimum, given four links a ternary tree is the best topology obtainable. The largest number of hops from the farmer to any worker is the base 3 log of the number of workers. However, a ternary tree does have a limitation and a disadvantage. The limitation of ternary trees is that the harvester and the farmer should reside on the same C. P. U. Although it is feasible to join two ternary trees together at the leaves forming a ternary diamond, with farmer and harvester at opposite ends. The disadvantage is that in configuration and placement languages that are evaluated by constant folding at compile time only, such as in occam, a fully scalable and balanced tree is very difficult if not impossible to describe. However, unbalanced trees are possible and it remains to be discovered how inefficient these are. Pipes and rings are very easy to scale linearly. However a large system will suffer from more communications problems than an equivalently sized tree.

There are of course some simple variations on the above. 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 one farmer.

The thing to keep in mind for simple topologies is that the shape should be self-similar from the point of view of the controller process or processes as it is this property which leads to an even flow of communication.

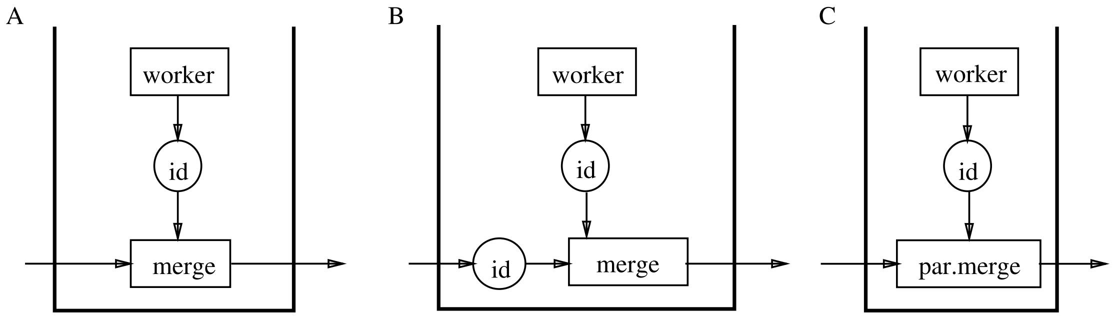
4.2 Harnesses

In essence harnesses consists of two extra processes, though for efficiency reasons we may use more. One is to distribute the jobs from the farmer to where it is needed. The other is needed to perform the equivalent collecting of results from the workers and passing them back up to the harvester. These run at high priority so that they may be executed in preference to the worker in order to service the links instantly.

This work originated out of a farming harness developed by Welch ([SS91] and [Stu91]), using the "lego-land" discipline [Wel88], to use fully the parallelism available on the transputer, for use in an application which had large data bandwidth and processing requirements. This basic design and variations upon it were implemented and tested by Sturrock. We have evaluated it further for the general case, and improved upon it while doing so. The basic strategy consists of passing out jobs from the farmer into the distribution part of the harness and filling the buffering capacity of the farm. The farmer becomes temporarily deadlocked until the workers obtain more work from the buffering in the harness. The collection mechanism returns finished jobs of work as quickly as possible to their destination.

We look at three methods of collecting results from the workers and two of distributing jobs to them. As the former involves simpler communications we look at this area first.

4.2.1 Merging results



The first method A, is a basic implementation of what needs to be performed, passing results back down stream to the harvester.

```
WHILE TRUE
  PRI ALT
    local.result ? result
      down.stream ! result
    up.stream ? result
      down.stream ! result
```

In the second method B, the reasoning behind having two processes is that now there is a process connected to each of the links, as these processes are running in parallel on the processor both links can be engaged at the same time, resulting in a quicker throughput of result traffic.

The third method C, is a conglomeration of the other two. It is a single process so there is less context switching and no on-chip copying of the data. This process is internally parallel and is an unwinding execution sequence of B.

```
SEQ
  {{{  load into result1   -- same as below
  PRI ALT
    local.result ? result1
      SKIP
    up.stream ? result1
      SKIP
  }}}
  WHILE TRUE
    SEQ
      PAR
        down.stream ! result1
        ...  load into result2   -- similar to above
      PAR
        down.stream ! result2
        ...  load into result1   -- same as above
```

Note that all of the `PRI ALT`s are written with the local worker process having priority. This ensures that at times of conflict the local worker is relieved of its result packet as soon as possible.

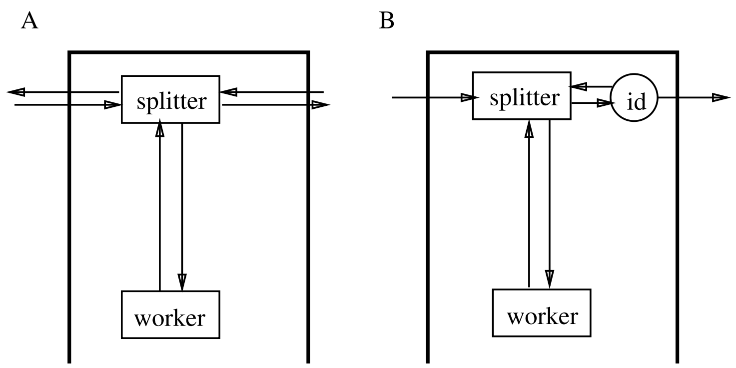
The worker has a buffer that relieves the process as soon as a job completes, by always being ready to complete the communications of the fresh result. The buffer then waits until the merge process is ready to interleave the result into the stream of results. In the worst case a job can finish just as the merger process has started to read from the upstream link. Any other processes waiting to complete a branch of the ALT have to wait for both that link to finish its transfer and for the down stream link to perform the same transfer. This buffer allows the worker to immediately proceed with the next job without having to wait for the slower link engines to complete transfers.

The results below show how long it took to transfer 100 counted array of bytes (INT::[]BYTE) through the three mergers. Note for transfers of less than 6 bytes

bytes	A - merge	B - merge + buffer	C - par.merge
1	32	34	22
2	35	35	23
4	40	37	25
8	50	43	31
16	69	56	40
32	109	80	64
64	187	130	110
128	345	229	202
256	659	427	388
512	1,288	824	760
1024	2,545	1,617	1,502
2048	5,062	3,204	2,989
4096	10,090	6,428	5,960
8192	20,153	13,020	11,900
16384	40,266	26,212	23,785
32768	80,509	52,584	47,552

(including the 4 byte INT) B performs slower than A, although clearly for larger packets there are considerable performance benefits. However, C clearly obtains the best all round performance, as it is when on-chip communications are removed that the largest obtainable performance is achieved from the hardware. Any of the three processes may be selected for a farming harness, depending on the amount of throughput required and how much code is desired.

4.2.2 Distributing jobs



Here we are not just collecting packets of data and sending them to one point, but rather the complete opposite, sending packets of data, in this case jobs of work, out from one place to where they are needed. This makes this part of the harness the more important. The selecting of where a job is needed is performed by requesting, with all harnesses performing this to varying degrees. In the past, some strategies have just performed requests at a local level, others involve requests travelling all the way between a worker and the farmer. From UNITY we have seen that a set of computations can be performed with a `for` loop sequentially may also be executed on a processor farm, assuming no interdependencies between each computation in the set. In the traditional model of execution the `for` loop generates the index values and the body of the loop consumes them directly and performs the computations. Thus in the equivalent of farming there is no need for requesting as part of the basic execution strategy, only production and consumption. Thus requesting should not be in a farming harness as part of the main strategy of job distribution. Rather they should only be in at the low level in order to prompt any `ALT` statements used.

The two methods of job distribution here are written with the same philosophies as the processes involved with collection and as a result use similar processor resources and produce similar performance.

The first, A is that of a single process that performs both parts of the job requesting protocol as follows,

```
WHILE TRUE
  SEQ
    req ! TRUE
    in ? job
    PRI ALT
      req.distant ? any
        give.distant ! job
      req.local ? any
        give.local ! job
```

This harness exhibits minimum buffering and is ideal for when there is only a slow demand for jobs.

The second, B uses two processes, the code of the first is identical to that of the above except that the instruction which makes the initial request for work isn't needed. Request being sent over links are avoided here by use of a buffer that performs the necessary request and then passes the job over the link. The advantage here again is that both links may be engaged at the same time allowing jobs to be distributed through the network at a higher rate. One interesting feature of B is that the job held in the buffer is only destined for other workers. The local worker can not access or claim it. This property of approach B may be undesirable for use with trees as the large number of end workers will be forced to finish off more of the final jobs of the application.

If it is known that the results bandwidth is equal to that of the jobs bandwidth, then the results should always be retrieved slightly faster than the jobs can be delivered.

In the distribution process, the local worker is at the lower priority in the `ALT`, in contrast to the merging of results. This shouldn't matter too much as most of the time the farm should be working flat out. However, if both guards are ready at the same time, which is the case when a farm starts up, the `ALT` is executed then either the jobs can be performed locally (less communication) or by a distant worker. As this code is run many times on many processors this latter option may lead to more parallelism and is therefore desirable. This point is more important in branched topologies. In our timings, where jobs could be processed faster than the harness could supply them (something that may happen in localised places in some farm topologies) having the `ALT` this way round did indeed lead to more processors working at any one time.

4.3 Other aspects

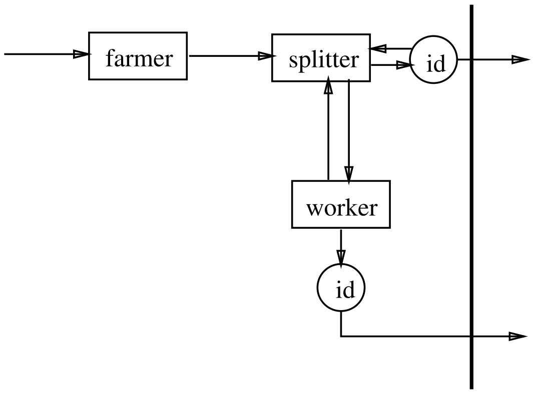
One common way to cut down communications is to put the jobs into groups. The only advantage to grouping up jobs that contain data is that of reducing the communications set up time. Further, jobs that consists solely of sequence numbers can be grouped to reduce the amount of communications bandwidth needed, by simply redefining the sequencing. If possible only have one part to the sequence number reducing the work load the farmer has to perform. For example, areas of Mandelbrot screens are best farmed by generating a single sequence of numbers with a simple loop. These numbers are then decoded by the workers in parallel into the y coordinate and the appropriate segment of the scan line. In general it is a good idea to perform as much work as is possible in the workers so that the farmer and harvester will have a very low cycle time, be able to process more and thus allow for a larger maximum possible size of farm.

Grouping jobs results in a reduced total set up time for communications. Though this grouping should be done to keep the total communications in balance overall. If jobs take a noticeably long period of time to process, the completion of whole task will be an unnecessarily drawn out process. In such cases the last job to finish will have been stuck behind C. P. U. intensive jobs in a queue of buffers that probably may only be executed by a single worker. The two parallel job distribution processes both contain a buffering capacity of two jobs per worker. Having a shorter job time greatly reduces the finishing latency. This can be especially noticeable when users can see the results coming in. For graphical demonstrations a suitable upper limit for the longest job could be of the order of a second.

The topologies that have some form of an end need to be terminated with a little care. For example, be careful not to engage `ALT` statements that consist of dummy channels as jobs will be lost. The preferred mechanism is to have a separate process that is a cut down version of the splitter process that performs exactly what is required at the end position. Having a process is recommended so a buffer has more work ready for immediate processing.

In general `ALT`s are expensive at run time so they should not be used to generalise an input. To illustrate, if writing the farming harness for a ternary diamond that fans out from the farmer and then in to the harvester; two processes should be written, one for each section of the diamond. A general process with two `ALT`s would increase the latency of the harness a great deal.

On many farms, the farmer process will not consume all of the available computing resources of the processor on which it executes. In such cases it is advantageous to have a worker process running on the same processor at low priority using the remaining processing capacity. Naturally the farmer should be run at high priority.



Thus it can be a good idea to keep the jobs of a reasonably similar size of small grain, as any large variation of execution time present in the job sequence can hinder a farm from finishing the task smoothly. When a farm, consisting of splitter B, supplied jobs that, in the middle of the task were a few orders of magnitude more computationally intensive than the rest, some workers in the farm ran out of jobs while the rest of the farm were still computing most of the central region of the task. Though to a certain extent a part of this problem is also due the way B passes around jobs as mentioned before. This hindrance would be reduced with splitter A in this situation due to the much smaller amount of buffering in the harness, but the speed of hand out of A is also slower.

For linear farms tests show it is better to send the results in the same direction as the jobs. This puts the harvester at the other end of a pipeline from the farmer, which allows for when it is desirable for the harvester to be a separate processor. If it is desirable to have both farmer and harvester on the same processor a ring topology can be used.

In a ring all of the data is following in one direction so it might be advantageous to multiplex jobs and results together across both links and then demultiplex the stream of data at the other end. This approach balances out the high proportion of jobs at the start of the ring and the high proportion of results as the end.

To give an indication of the fall off in performance due to communications overheads the following results are presented. These were obtained on a ring topology using splitter B and merge C, for computationally bound work of small jobs (100 for each worker) generating in just over a second of work to create a 4K result. No worker was present on the farmer's processor. Even with a large ring the performance obtained is

processors	%	effective speed-up
1	100.000	1.000
2	99.025	1.981
4	98.036	3.921
8	97.089	7.767
16	96.158	15.385
32	95.236	30.476

encouraging. For applications where the communications demand starts to exceed the computational demand this starts to drop off and a non-linear topology needs to be used. For processor farms containing anything of the order of 16 or above workers one should start to consider a tree or perhaps diamond topologies in order to reduce the amount of unnecessary communications traffic.

So far we have only discussed implementation for a linear topology, namely a pipe or ring. Fortunately all of the processes above can be easily expanded to implement spanning topologies such as trees. However, this raises some questions as regards how to extend the harnesses. Fortunately for the harnesses developed here the body of the `ALT` statements can just be expanded for the extra channels. However, replication makes `ALT`s more expensive at run time. Losing up to around 8% of the communications bandwidth. Thus it is better to write out the three channels by hand rather than to use replication.

The number of jobs should always outweigh the number of processors or the performance of the farm greatly deteriorates. Should such performance be desired other parallelisation techniques should suggest themselves to be more appropriate.

5 Future Work

We need to test to see if for some application parameters that the time lost while workers are blocked by link engines are less than the overheads of having a buffer.

Trees and diamonds need to be evaluated in the same manner. Trees may exhibit adequate performance without being fully balanced.

Ideally it should be possible to recommend a suitable topology and harness for any given job size, job run time, size of result and desired performance requirement.

Various works have been published that use a discipline which looks carefully at the times of any particular transputer operation, say C. P. U., and then considers what else may be done in that length of time by the other parts of the chip, say link engines. This discipline needs to be applied here to define precise points at which one technique has advantages over another.

6 Conclusions

This work has reached six conclusions.

1. The idea of a single central controller that passes out work to a number of processors is a very flexible method of execution. Also as the technique consists of very little synchronisation, it is likely to be very efficient.

2. A parallelisation technique is really a strategy for the execution of the statements of a program.

3. Thus as farming is such an execution strategy, it should perform equally well for all applications, subject to hardware constraints.

4. In a parallel implementation, which topology is used is just as important to the success of the implementation as such issues as algorithm selection and the quality (efficiency etc.) of the communications harness code.

5. We can implement a processor farm on a network of transputers to obtain close to the maximum possible computational performance.

6. We believe that farming is as fully understandable as the "fetch-and-execute" cycle of a conventional C. P. U., and that it is worth studying in its own right, thoroughly, rigorously and completely.

In this paper we have attempted a first significant step in this direction.

7 Acknowledgements

This work was funded by a Science Engineering Research Council (SERC) Research Studentship Award. Many thanks to my supervisor Steve Hill and David Morse for enduring the earlier work. Thanks also go to the anonymous referees for their comments and Shane Sturrock for talking so much about his work. Those who know the works of Peter Welch and of Herman Roebbers will find their influences in the latter half of this paper.

References

[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.

[CHvdV88] N. Carmichael, D. Hewson, and J. van der Vorst. A prototype simulator output movie system based on parallel processing technology. In Charlie Askew, editor, *Occam and the Transputer—Research and Applications*, pages 169–175. 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, *Occam and the Transputer—Research and Applications*, pages 50–58. IOS Press, 1990.

[INM87] INMOS. *Communicating Process Architecture*, pages 31–44. Prentice Hall, 1987.

[Inm88] Inmos. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[Man82] Benoit Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co., 1982.

[MS87] David May and Roger Shepherd. Communicating Process Computers. Technical Report 22, INMOS, 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.

[SS91] Shane S. Sturrock and Ian Salmon. Application of Occam to Biological Sequence Comparisions. In Janet Edwards, editor, *Occam and the Transputer—Current Developments*, pages 181–190. IOS Press, 1991.

[Stu91] Shane S. Sturrock. Biological Sequence Comparisons on a Transputer Network. Master's thesis, University of Kent at Canterbury, October 1991.

[Wel88] Peter H. Welch. The Occam Approach to Transputer Engineering. In *Third Conference on Hypercube Concurrent Computers and Applications*. ACM, 1988.