Occam on T8 and T9 machines: coding and conversion issues

Gabriel Howard *Tim Oliver*
Centre for Mathematical Software Research
University of Liverpool

August 1991

1 Introduction

The T9/C104 transputer system allows programs to be written which reflect the problem rather than the details of the hardware. We no longer need to route every piece of data through intermediate processors to its destination. Just declare a suitable channel and the system will ensure that the message is delivered. The conceptual networks around which the program messages flow and the actual hardware switching networks which deliver the messages are uncoupled. However this freedom needs to be handled with care if the resulting programs are to be efficient. In this paper we discuss parallel programming of such transputer systems, to help programmers familiar with writing T8 `occam` code appreciate the issues involved when writing for T9 systems.

We look at these issues from two points of view

- Writing code that is efficient on T9 systems.
- Converting existing T8 code to take advantage of the flexibility of T9s and developing code on T8s targetted at T9s.

Section 2 gives a short description, from the software developer's point of view, of the hardware and the facilities of the T9 which are not available on the T8. The T9 has pipelining and cacheing to improve performance but we discus techniques for efficient use of the processor only briefly. Our main concern is the effective use of the enhanced communications network. T8s can have 4 2-way links and this is reflected in the software. Any messages for different destinations are multiplexed down an appropriate link and distinguished by software protocols or tags. The T9 liberates the programmer by doing the multiplexing and routing invisibly to provide so called virtual channels, direct paths between two processes on any two processors. A suitable switching network is required but the application programmer should assume that the network exists and concentrate on where the data will end up and not on how it gets there. Section 3 discusses the need for and use of libraries of routines to do higher level communication operations such as broadcasting or data gathering.

We also look at developing programs in `occam` under the Southampton Virtual Channel Router, VCR [2]. This system allows us to write T9 like communications independent code on existing T8 machines. It incorporates a new, experimental, `occam` compiler from INMOS, the VOC, with support for virtual channels and extra facilities at the configuration level. Our code examples are based on this system and the final constructs may be different for the T9 product compilers. For T9 code being developed on current T8s we can also code in pseudo-parallel on a single T8. By pseudo-parallel we mean that the code is a number of independent procedures communicating only via channels. These procedures are then combined at a top level of code which defines the channels and passes them into the processes and the entire program runs on one processor. Using this method there are problems about the way in which channels are defined and passed to processes. Pseudo-parallelism has the advantage that the standard debugger works but is otherwise very much a second best option. The VCR is a much more attractive development platform as

virtual channels have been simulated in software and the program can run on current transputer arrays. However there is a considerable overhead and time critical code is difficult to test. We do not discuss Fortran or C programming; the same principles for obtaining efficiency apply. However the VCR system does allow Fortran or C code to be incorporated within a `occam` harness.

In later sections we consider solving problems on distributed processors and examine three approaches, Geometric Parallelism - physical or over a data space, Divide and Conquer, and Farming and illustrate points made earlier with code fragments. Most of the examples use geometric parallelism and come from numerical linear algebra problems.

There are no pipeline or conveyor belt examples. Simple pipelining with data being transformed in a series of steps performed in different processes does not need the extra facilities of the T9. If the transformations are dissimilar this type of functional pipeline is usually short simply because each process has to be coded individually and loaded onto a specific processor. An example of a pipeline of similar operations is a prime number sieve. A sieve could be coded as a pipeline to process a stream of odd numbers with a new process being spawned at the end of the pipe every time a new prime was detected. Mapping the pipe onto the available processors to achieve load balancing and effective use of the communications network is a challenging problem.

Farming and divide and conquer are now easier to implement. However they are limited by the speed at which data can be transferred from the master processor. If the individual work segments can be generated from a small amount of data then this is an attractive approach but it needs to be implemented with care. We give an example of naive farming code for a master transputer and the more sophisticated code required by an efficient implementation. Divide and conquer can also be done by spawning processes; the technique is usable under the VCR. Again the the amount of data required to describe the problem at each level is a limiting factor.

Data distribution is important in all cases. The speed of communications has increased but so has the speed of computation so we have to avoid repeatedly redistributing large quantities of data. This is obvious with divide and conquer algorithms. There is no point in routing data through a number of processors for processing by processors at the bottom of a tree. The data should be sent direct to the processor on which it is going to be handled.

2 Hardware

At the simplest level, the T9 can be considered as a faster T8. In raw computing terms, floating point operations on the T9 are about 10 times faster than on the T8 at up to 25MFlops compared with 2.5MFlops for the T8. Also the communication rate on the 4 links is increased by a factor of 5 to 10MBytes/sec from 2MBytes/sec. Hence, a program developed for an array of T8s can be ported, unchanged, to an array of T9s which have their links directly connected in the same configuration as the T8 array. This program will benefit from the greater performance of the T9 chip and may run up to 10 times quicker than the T8 version. However, because the computation rate/communication rate ratio is different for the T9, programs tuned for maximum performance on a T8 array are unlikely to be as well balanced on a T9 array; the grain size of T9 programs should be greater than that of T8 programs since this ratio has increased.

To gain the greatest benefits from the T9, programs should be written to take advantage of its architectural features which are described in [7]. The performance of the floating point unit is greatly improved with instructions such as multiply or add taking only 2 cycles to execute. One drawback of this is that housekeeping functions like array index calculations and control flow operations now have a much more significant cost. Similarly the cost of a context switch is now more significant.

Perhaps the most important feature of the T9 architecture is the *virtual link* capability. Although both the T8 and T9 have 4 hardware links, the T9 has the ability to multiplex many virtual links down each hardware link. This is achieved by packetizing each channel communication and outputting individual packets on the hardware links and then waiting for an acknowledge packet before outputting the next packet on the channel. This allows packets on multiple channels to be interleaved on each hardware link. When virtual links are combined with a network of C104 switch chips, the programmer views the network as a set of processors, each of which may be directly connected to any other processor by a channel on a virtual link, and there is (almost) no limit on the total number of channels that each processsor may use to

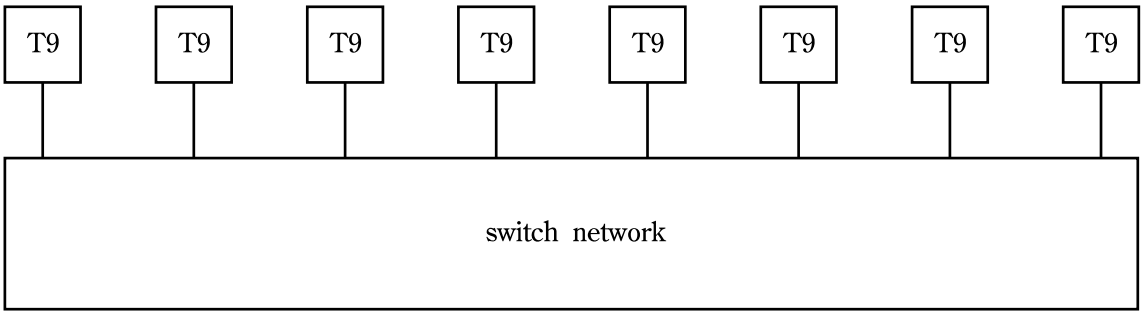


Figure 1: Hardware configuration of switch network and processors

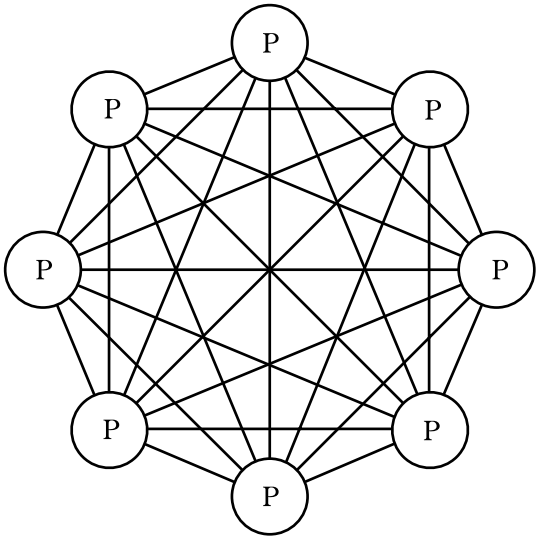


Figure 2: Programmers view of processor configuration

connect to other processors.

There are a number of levels at which to view the configuration of a T9/C104 array. At the system level the array is composed of a network of C104 switch chips, interconnected by the 32 serial links on each chip. The T9 processor links are connected to links in this switch network. Hence, two T9 processors are not generally directly connected, but rather are connected via the switch network. The number and connectivity of switch chips determines the performance of the switch network [3]. System software must program the switch chips so that messages on a virtual channel from one processor are routed to the correct destination processor. Ignoring the detail of the switch network, the array can be viewed as a set of T9 processors connected by their 4 links to a switch network (see Figure 1). Finally, at the programmer's level the switch network may be ignored completely and the array viewed as a set of processors connected by any number of virtual links, each link connecting any pair of processors (see Figure 2).

In comparison with T8 arrays, the T9/C104 allows direct channel communication between any pair of processors and a large number of channels per processor. However, the switch network introduces a significant latency into channel communication between processors. This means that the communication rate achieved on a channel is much lower than the 10MBytes/sec throughput of the hardware links. Hence, to gain good communications performance it is even more important to consider executing multiple parallel processes on each T9 processor than was the case for the T8. These parallel processes will generally improve performance by overlapping communication with calculation.

3 Communication

The development of efficient communication routines for different local memory architectures is an important but time-consuming task. It is therefore desirable to have access to a library of communication routines that provide the functionality generally required by programmers. If programs are developed using these library communication routines then porting the program from one architecture to another (T8 to T9, even) becomes much easier provided the library is available on both architectures. Hence it is recommended that all programs perform communications through a general communications library, even if this library has to be written by the programmer. The library would include such routines as broadcast, multicast, distribute, collect, and global operations (such as inner products), as well as point to point communications.

As well as aiding portability between different architectures, the use of a communications library also hides the underlying implementation of efficient communication from the programmer. Detailed implementation of a routine might vary greatly between architectures, even between T8 and T9 machines. For example, communication on T8 arrays is closely tied to the configuration of the processors, which is usually as a grid or chain. On T9 arrays on the other hand, a communication routine can be written independently of the underlying hardware configuration. Instead, the routine will be written using the programmer's view of the array as a set of processors connected by any number of virtual links. Using a communication library decreases the program development time. Initial versions of a program can use simple but functional communication routines whilst the program is developed. In later stages efficient communication routines can be written to exploit the underlying hardware and provide improved performance. A communications library may also be reused in every program avoiding the need to re-develop communication routines.

In developing efficient communication routines for the T9 several factors must be considered. We will assume that all 4 hardware links on each T9 are connected to the switch network. The most important consideration is that the programmer may use as many virtual links connecting to as many processors as desired. This frees the programmer from the details of the hardware configuration and allows him to code communication operations in a more natural manner. In addition, a general principle that gives improved performance is to use multiple channels (virtual links) simultaneously on each hardware link. This makes better use of the link bandwidth than a single channel because of the high latency on channel communication due to the delays introduced by the switch network. For example, a distribute operation, which distributes segments of a vector from the source processor to the array of p processors, might be coded as a star network of p channels connecting destination processors to the source. The source processor would output a segment on each of the p channels in parallel. Packets from each channel communication are interleaved on the hardware links and the best link throughput achieved. Similarly, a broadcast routine may make use of some form of tree configuration of channels. It should be noted that an individual T9 program is likely to use many different communications topologies such as trees, stars, and chains dynamically, using the most suitable topology of channels for the intended purpose. This is in contrast with a T8 program which usually has to map all communications routines onto the single hardware topology.

The programmer must have some method for defining and managing all these channel structures when programming in `occam` under the current version of the VCR. In this environment, the language is still essentially static and configuration is achieved by a configuration file in which the top level processes and the connecting channels are declared. Hence the communication channels must be passed down through the procedure levels to the actual communications routines. The most suitable form for declaration of channel structures is as vectors of channels; passing large numbers of single channels to processes is messy. Use has shown that the most flexible way of making multiple channel topologies available to the processes is by passing 2 vectors of length p to each process; one vector has an output channel at index i to process i, whilst the other vector has an input channel. This provides an all-to-all channel configuration. Both of these vectors are passed to communications routines which select individual channels for use depending on the communication structure required for the communication routine and the position of the process within that structure.

The VOC provides a new declaration statement, `ARRAY`, that facilitates the declaration of channel arrays at the configuration level of an `occam` program. This statement permits new arrays of channels to be constructed from arbirary slices of other channel arrays. Without this statement each configuration level channel would have to be declared individually and passed to a process as a separate parameter. This is very untidy and for all-to-all configurations means that the configuration code is dependent on the total number

of processes, since a channel connected to every other process must be passed into each process. With this construct the configuration is elegant and reasonably intuitive; the following code fragment shows an all-to-all configuration for p processors:

```
[p][p]CHAN OF ANY all.to.all:
PLACED PAR i = 0 FOR p
  in  IS all.to.all[i]:
  out IS [ARRAY j FROM 0 FOR p OF all.to.all[j][i]]:
  process(in, out)
```

Within each process these channel arrays may be passed to communications routines to perform operations like broadcast, distribute and global inner products. The process code might look something like this:

```
PROC process ([]CHAN OF ANY in, out)
  SEQ
    ...  work
    inner.product (in, out, num.procs, id, root, veca, vecb, size, res)
    ...  more work
:
```

This method hides as much as possible of the underlying communication from the programmer. However, the programmer must still be aware of how the routines work when he wishes to execute multiple communications routines in parallel. In this situation the programmer must check whether the two routines make conflicting use of a single all-to-all structure of channels. If this is so, or if the programmer is unsure of the situation, then the programmer must provide another complete all-to-all channel structure for use by one of the parallel routines. Code might then be as follows:

```
PROC process ([]CHAN OF ANY in1, out1, in2, out2)
  SEQ
    ...  work
    PAR
      inner.product (in1, out1, num.procs, id, root, veca1, vecb2, size1, res1)
      inner.product (in2, out2, num.procs, id, root, veca2, vecb2, size2, res2)
    ...  more work
:
```

A direct communication between two processes on a channel can be coded as:

```
-- process <source>
out[dest] ! data

-- process <dest>
in[source] ? data
```

However, to improve portability to other languages or local memory MIMD architectures which are not channel-based it may be better to code these communications as subroutine calls even though in `occam` this is not as elegant as the method shown above:

```
-- process <source>
send (in, out, dest, data)

-- process <dest>
receive (in, out, source, data)
```

We conclude this section by looking briefly at one particular communications routine, a global inner product. This routine forms the inner product, $\sum_{i=0}^{n-1} veca[i] * vecb[i]$, for a pair of distributed vectors, $veca$ and $vecb$ and broadcasts this result. The vectors are assumed to be distributed in an identical manner. An outline `occam` procedure listing is given below:

```
PROC inner.product([]CHAN OF ANY in, out, VAL INT root, num.procs, id,
      VAL []REAL32 vec1, vec2, VAL INT dim, REAL32 res)
  -- in, out provide all-to-all channels, not just tree channels
  SEQ
    PAR
      ...  calculate my partial sum
      PAR i = 0 FOR b
        ...  get partial sums from children
    ...  add partial sums from children and self together
    ...  output resulting partial sum to parent
    ...  broadcast final inner product value <res>
:
```

The routine is based on a tree configuration of processes with a branching factor (number of children) of b. The value of b is not limited to 3 as was the case for T8 algorithms. In fact, as explained earlier in this section, more than 1 channel can probably be mapped onto each physical link before the link bandwidth is saturated. For example, if $b = 8$, i.e., 8 channels are used, 2 on each link, the number of steps in the algorithm is reduced, compared with using only 4 channels, without increasing the cost of communication within each step. The best branching factor will vary from one T9 machine to another depending on the number and configuration of switch chips. The algorithm starts with each process calculating its partial inner product. These partial sums are accumulated up the tree to the root process, `root`, which then broadcasts the final inner product value, `res`, to all the processes. The broadcast is achieved by calling a standard broadcast routine. This broadcast routine may use the same tree structure of channels as the rest of the inner product routine, or it may use a different structure. For example, a broadcast routine could be implemented as a wavefront algorithm where at each step in the algorithm each process holding the data sends the data to m new processes. Again, the best value for m may vary from machine to machine, but is probably the same value as b. This method of broadcasting is more efficient than using the simple tree structure since fewer steps are required.

4 Parallel programming techniques for T9 systems

The previous two sections outlined the principles to be considered when designing a T9 program. In this section we give code fragments as examples of the use of these principles.

An important principle is the exploitation of parallelism within code for a single processor. This use of parallel processes on one processor enables the program to make the best use of the processor's resources; specifically, it allows a processor to continue doing useful work whilst a thread of the program is blocked on communication. We divide this use of parallelism into three main areas:

1. simple overlapping of individual communication and computation blocks,
2. placing multiple identical copies of the process code on each processor, and,
3. performing several threads of the algorithm in parallel.

The following three subsections explain each type of parallelism and give a specific example of its use. A fourth subsection describes the technique of "spawning" processes onto other processors which programs can use to perform simple dynamic load-balancing. A particular program can combine any and all of these techniques to achieve the best performance.

4.1 Overlapping communications and computation

One way to continue performing useful work (computation) on a processor when a communication between processors is required by the algorithm is to explicitly code two parallel threads. One of these threads performs the necessary communication whilst the other thread does some computation. Care must be

exercised since the two threads operate in parallel, and therefore neither thread must alter the value of variables or vector elements used by the other thread.

Many numerical algorithms have fragments of code which communicate vectors between processors and perform some computation on the elements of these vectors. In order to execute the computation and communication as parallel threads the communication thread may read a vector into one buffer whilst the computation thread is manipulating a vector in another buffer. This operation is very much like a pipeline operation; at the start a vector must be read into the first buffer, then a sequence of overlapped communication and computation steps can proceed, with only computation performed in the last step.

```
WHILE loop
  PAR
    ...  perform computation
    ...  and communication
```

Hence, better performance is attained if the initial communication is short, e.g., a complete vector operation is divided into many smaller vector operations. Examples of the use of this technique can be found in [10, 9, 4].

4.2 Multiple identical processes on each processor

One way in which communications and calculation can be overlapped is to divide the problem into smaller segments and have multiples processes, say two, on each processor. One process can calculate while the other identical process is communicating. The code for the individual processes is straightforward to write and does not require any internal parallel structures. The drawback is that data needed by all processes has to be sent twice to each processor. This could be handled by having a separate data distribution buffer process on each processor but then we have lost the simplicity of replication. Consider a matrix vector multiplication where we wish to calculate

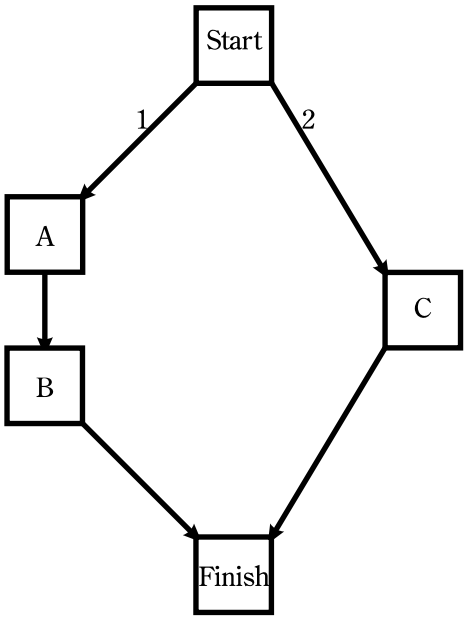
$$y_k = \sum_{l=1}^{n} A_{k,l} x_l \qquad k = 1..n$$

The $n*n$ matrix A is distributed by rows across the p processors with $m = n/p$ rows on each processor and the vector x is stored on some processor. We could broadcast the vector first and then do the calculation but it is better to broadcast the data in chunks so that computation can start as soon as possible as explained in section 4.1 This code can be written either using explicit buffering or with multiple processes per processor. In both cases the communications and calculation are overlapped but duplicate processes per processor will produce a greater overall volume of communication, n data items sent $2p$ times. In the above example the data traffic can be reduced by dividing the matrix into blocks rather than full rows. However the example shows that when the technique of multiple processes per processor is used to hide communications it is important to check how much extra message traffic would be generated.

The simple code for the case where the vector x is distributed uses p parallel threads on each processor. Each thread read in the data from the appropriate processor and calculates a partial result to summed later. The code on a processor is of the form:

```
[p][m] REAL32 y.part:
PAR i = 0 FOR p
  [m]REAL32 tempx:  -- m = n/p
  SEQ
    ...  get x[i*m..(i*m + m-1)] from processor i and store in  tempx
    SEQ k = 0 FOR m
      ...  calculate partial sum y.part[i][k] involving x[i*m..(i+1)*m-1]
```

The result y is assembled from all the $y.part$ contributions. Overlap of calculation and communication occurs automatically but at the cost of huge amounts of temporary storage. In large sparse problems the vector x is too big to fit on one processor so the work has to be done in sequential stages. At stage j a




```
PAR
  – thread 1
  SEQ
      A
      B
  – thread 2
  SEQ
      C
```

Figure 3: Parallel algorithmic threads

processor i communicates with the processors $i \pm j \bmod n$. Again we can use either multiple identical processes or explicit buffering to overlap the communications and the calculation.

The method is attractive if the increased data traffic is not excessive because the individual process code can be written simply. However a program which does processor to processor broadcast is more efficient than one that does process to process broadcast. This implies buffering processes on each processor as well as the original duplicated communication and calculation processes. At this point the distinction between this approach and that of section 4.1 becomes blurred. The technique is discussed in a $\mathcal{PUMA}$ working paper, [6] on sparse matrix vector multiplication.

4.3 Multiple algorithmic threads

A second way to exploit parallelism within each processor is by introducing *algorithmic parallelism*. By this we mean that threads of the algorithm which are independent are coded to execute in parallel. Figure 3 gives a simple example of this; the two algorithmic threads 1 and 2 are independent of each other and so the operations A, B and C may be coded as shown. This technique has been used for a parallel preconditioned conjugate gradient algorithm [1].

4.4 Spawning processes

It is now possible (under the VCR) to spawn processes using remote procedure calls. This is effectively a PLACED PAR in the middle of the normal code and a `occam` compiler incorporating this facility is being developed by Syseca [11]. We expect the same functionality to be provided for T9/C104 systems. This simplifies the writing of divide and conquer and recursive algorithms but the hidden communication costs must be considered. Even if the code for a procedure is already in place on the target processor a remote procedure call requires at least three communications. The first initiates the call, the second transfers the parameter data and the third returns the result parameters. The time that the spawned process takes to compute must hide these overheads. Spawning is of particular interest in general codes for solving differential equations and similar cases where a function defining the user's problem is needed and supplied by the user. In the paper on adaptive quadrature [5] we analyse the overheads involved when 21 evaluations

each returning a single result depending on a single parameter value are done in parallel. This suggested that each function evaluation had to do 20 to 50 floating point operations in order to achieve any speed up at all. The better the single processor performance on the spawned code the more floating point operations are needed to overcome the overheads.

The technique can have advantages in parallelisation of existing serial code where the time consuming core of the computation is interleaved with problem set up code. There is a similarity between this approach and the use of libraries to handle the parallel sections.

In a short paper on the use of recursion David May [8] suggested code for evaluating the sines of all the elements in a large array. The code is neat but as it stands there is a the large amount of data shuffling through intermediate processors before it is processed. The code, rewritten using process spawning under the VCR is supplied as an example with the VCR system and is reproduced below. Although for clarity we have hidden the spawning detail in folds the code is as shown and an exact copy of the procedure `sin.vector` is spawned.

```
PROC sin.vector( []REAL32 sinv, VAL []REAL32 v )
  VAL s IS SIZE v :
  IF
    s > 1
      VAL bottomhalf IS s/2 :
      VAL tophalf    IS (s/2) + (s\2) :
      ...  setup for spawning the code under the VCR
      PAR       -- in full occam PLACED PAR
        ...  spawn bottom half using PLACED.RUN, the VCR equivalent of
        --  sin.vector([sinv FROM 0 FOR bottomhalf],
        --             [v    FROM 0 FOR bottomhalf] )
        ...  spawn top half using a PLACED.RUN
        -- sin.vector([sinv FROM bottomhalf FOR tophalf],
        --            [v    FROM bottomhalf FOR tophalf] )

    s = 1
      sinv[0] := sin(v[0])
:
```

The problem is that each of the n original data items are moved $log_2 n$ times and so are each of the n results. First we need to stop subdividing and start calculating as soon as each of the p processors available has been sent a set of data. Then each data item is sent $\log_2(n/p)$ times. But for efficiency we need to send the data partitions direct to the processor where they are to be used, in segments so as to overlap communications and calculation the full speed. However recursive algorithms can be used if the information sent via intermediate processors is a pointer to segments of data structures rather than the data structures themselves. As problems get larger the data structures will have be distributed and manipulated using pointers in some way.

5 Converting T8 programs for the T9

As was mentioned in Section 2, T8 codes will run unchanged on a T9 machine, taking advantage of the increase in processing power of the newer chip. However, the direct communications ability of a C104 switch network is not exploited unless the original code is modified. For example, instead of sending a message directly between a pair of "non-neighbouring" T9 processors via the switch network, the unchanged program will buffer the message through all the intermediate processors as in the original T8 code, giving a far from optimal performance.

If the original program was coded in terms of high level communications routines, such as broadcast and distribute, then the conversion process consists of replacing the old T8 communications library with an optimised T9 version. Unfortunately, many existing `occam` programs have their communications coded directly in terms of individual channel communications. In these situations it would be nice to have an

automatic conversion utility which would translate T8 communications into efficient T9 communications. Unfortunately, again, such a utility would be difficult to develop.

Consider, for example, the following fragment of code which gives a chain of processors a processor number and a total count of the number of processors:

```
-- master
out.left ! 1; no.slaves

-- slaves
in.right ? iam; no.slaves
IF
  (iam < no.slaves)
    out.left ! (iam+1); no.slaves
  TRUE
    SKIP
```

A hand written version of this code fragment for a T9 might be something like:

```
-- master
PAR i = 1 FOR no.slaves
  out[i] ! i; no.slaves

-- slaves
in[0] ? iam; no.slaves
```

Let us assume that a conversion utility is available which performs static analysis of the program code, and that this is able to correctly pair up the two sections of code in the master and slave processes. The utility would certainly be able to find the channel configuration of the program from the configuration file and could therefore work out which processes are involved in this section of code. However, it would not be able to evaluate the conditional expression in the IF statement which may be variable at run-time. Hence, it cannot know which is the last process to input the message. But the variable `iam` can be evaluated statically since the master gives this an initial constant value of 1. This simplifies the expression to one in terms of the unknown `no.slaves`.

The utility must also decide whether the side effects, the variable assignments, produced by the communication are needed on each process that takes part. In this case the assignment of process number and the total number of processes to each process is essential. But, very similar code might be used to communicate data from one process to another distant process through intermediate processes which do not require the side effects. Further analysis of the codes might be able to determine which case applies for any given communication.

If the utility manages to correctly solve the preceding problems it could substitute a standard code form for the original code, inserting the unknown variable `no.slaves` into its correct place. If each object communicated is considered independently by the utility then for our example code the utility might correctly substitute a broadcast operation for the variable `no.slaves`, and individual independent communications to each slave process for `iam` (which may be performed in parallel):

```
-- master
PAR i = 1 FOR no.slaves
  out[i] ! i

-- broadcast routine (or simple code here)
PAR i = 1 FOR no.slaves
  out[i] ! no.slaves

-- slaves
in[0] ? iam
```

```
-- broadcast routine (or simple code here)
in[0] ? no.slaves
```

Considering all the objects as one composite object the utility might choose to substitute an individual communication to each process:

```
-- master
PAR i = 1 FOR no.slaves
  out[i] ! i; no.slaves

-- slaves
in[0] ? iam; no.slaves
```

The problems that face an automatic conversion program become even more difficult to solve when complex communications are used by the program which is to be converted.

The programmer can help a conversion utility by placing comments in his code which identify the type of communication operation that each section of code performs, for example marking broadcast, distribute and individual communications. This perhaps requires a similar amount of work to coding the program in terms of communication subroutines.

Without an automatic conversion utility the programmer must inspect the original program himself. Consider a simple example of data distribution. To distribute data efficiently over a linear chain on a T8 system requires code on the starting processor of the form

```
SEQ  i= 0 FOR no.workers
  send.values(out.right, adata[i])
```

and on the worker processors

```
SEQ
  in := 0
  out := 1
  get.values(in.left, xdata[in])
  SEQ  i= 0 FOR (no.workers -1)
    SEQ
      PAR
        get.values(in.left, xdata[in])
        send.values(out.right, xdata[out])
      out := in
      in := 1-in
```

This can usually be replaced by

```
PAR  i= 0 FOR no.workers
  send.values(to.worker[i], adata[i])
```

and

```
SEQ
  in := 0
  get.values(in.from.master, xdata[in])
```

However it may be that the two final sets of data `xdata[in]` and `xdata[out]` are required on the intermediate processors. It is uncertainties such as this which make it hard to convert T8 style code to full T9 style code automatically. Since all the data is coming from a single source there is a limit to how far the virtual channels are going to speed up the data distribution. The bottleneck is the speed at which the data can be despatched by the master processor.

On a square grid it is quite difficult to plot paths for data transfer of a message from i, j to k, l. First a path has to chosen and then the data has to be routed along this path with the cooperation of all intermediate

processors. The potential for deadlock is considerable. Programmers usually take a simple approach and send everything to the start of a row, up the first column to the kth row and back to the lth column transputer. It is not easy to tell from the code whether values are retained in intermediate processors. This may happen always, sometimes or never depending on the algorithm. Broadcast is also a problem in conversion; how do we decide that the functionality of a broadcast is needed and where it originates.

No doubt a number of sophisticated communications routines have been developed to handle data movement in specific programs. If these routines have been written in a modular fashion with the top level code reflecting function, such as collect and redistribute an updated vector, then it is straight forward to convert to T9 by substituting an appropriate function call.

5.1 A Farming example

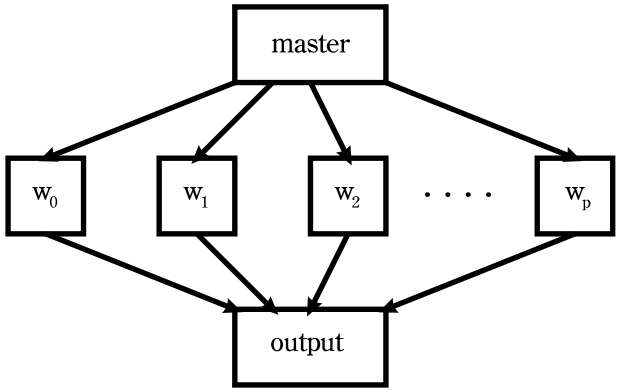


Figure 4: The T9 farming processor network

Programs which use the T9 efficiently may often be considerably more complicated than the original T8 code. Consider a simple farming code for processing blocks of data sent by a master processor to any available worker processor and then send to some other process for output. The worker code is complex and has to handle forwarding the data but the code on the farmer processor is a simple sequential loop. The master processor on a T8 in such a farming application sends data down the link to the first worker and it is passed along by the workers until a free slot is found. In outline the master code is

```
SEQ
  ...  set up problem
  --{{{  send the data blocks
  SEQ block = 0 FOR number
    from.master ! data.set; data[block]
  --}}}
  ...  code to shut down processing
```

Naive farming code on a T9, using the conceptual network shown in Figure 4 waits for a signal from a slave that it was ready to receive more data and then the next block of data is sent to that processor for processing. So the only change needed to the master code is an array of channels and an ALT construct to ensure that the data is sent to the next available worker.

```
SEQ
  ...  set up problem
  --{{{  send the data blocks
  SEQ block = 0 FOR number
    ALT i = 0 FOR p
      to.master[i] ? ready
```

```
        from.master[i] ! data.set; data[block]
  --}}}
  ...  code to shut down processing
```

There should be buffering on the worker processes in the T8 code so that they can continue processing while waiting for the next data set to arrive. The master code for sending the data blocks can be improved by initializing the workers in parallel.

```
--{{{  send the data blocks
SEQ
  PAR block = 0 FOR p
    from.master ! data[block]
  SEQ block = p FOR number - p
    ALT i = 0 FOR p
      to.master[p] ? ready
        from.master[p] ! data.set; data[block]
--}}}
```

However we are still only listening and sending out the bulk of the data on one channel at a time. If we try to send in parallel we have to ensure that no data block is omitted or sent more than once. We could arbitrarily assign the blocks to workers in advance, for example sending every ith block to worker $i \bmod p$. But this negates automatic load balancing which is the big advantage gained by using the farming technique. In order to deal with this we need to have a control process on the master processor. This control process is a sequential loop but it can service requests quickly because it only communicates tags and a single data item with parallel processes which are on the same processor. There is one parallel process listening to each worker which waits for the ready signal from the worker and then obtains the block number of the next data set from the control process. That data set is then sent to the requesting processor. In this way data transfer can take place at full speed on all links out of the master transputer. The worker transputers may or may not be able to keep up but at least the master will be sending as fast as possible. The code for the fold, `send the data blocks`, becomes:

```
PAR
  --{{{  control process
  SEQ
    SEQ block = 0 FOR number
      ALT j = 0 FOR p
        to.control[j] ? ready
          from.control[j] ! reply ; block
    PAR j = 0 FOR p
      SEQ  -- close down messages
        to.control[j] ? ready
        from.control[j] ! reply ; -1
  --}}}
  --{{{ communication with workers
  PAR  i = 0 FOR p
    INT block.no:
    SEQ
      block.no := 0
      WHILE block.no > 0
        SEQ
          to.master[i] ? ready
          to.control[i] ! ready
          from.control[i] ? reply; block.no
          IF
            block.no => 0
```

```
            from.master[i] ! data.set; data[block.no]
          block.no < 0
            from.master[i] ! finished
--}}}
```

This is considerably more complicated than the 2 line sequential loop it replaces.

6 Conclusions

Most of the programming techniques discussed in this paper are equally applicable on T8 and T9 machines. One of the reasons why these techniques have not been widely used on current generation T8 machines is that developing even simple parallel programs on these machines can be difficult, so the niceties of improved efficiency are frequently not considered. The T9/C104 frees the programmer the main constraint of the T8 architecture—that of only having 4 channels to near-neighbours—and so permits easier development of parallel programs. This in turn allows the programmer to pay more attention to the efficiency of the program. Even so, to make the most of the capabilities of the T9 still requires complex programming, as is shown in Sections 4 and 5.1.

The complexity of efficient communications on both T8 and T9 systems, and the need to port programs from one architecture to the other stresses the need for the encapsulation of these operations in communications subroutine libraries. Similarly, arithmetic libraries such as BLAS libraries are essential to get good computational efficiency from a processor and still allow easy porting to other architectures.

When such standard libraries are widely available, the development of parallel programs will be greatly simplified. Resulting programs will be both reasonably efficient and more readily portable than most existing programs.

References

[1] Rod Cook. Timing models for preconditioned conjugate directions. $\mathcal{PUMA}$ Working paper 13, Centre for Mathematical Software Research, University of Liverpool, August 1990.

[2] Mark Debbage, Mark Hill, and Denis Nicole. Virtual channel router version 2.0 user guide. $\mathcal{PUMA}$ Working paper 25, University of Southampton, June 1991.

[3] Holm Hofestädt, Axel Klein, and Erwin Reyzl. Investigation of dynamically switched, scalable network structures. $\mathcal{PUMA}$ Deliverable 2.1.1, Siemens AG, October 1990.

[4] Gabriel Howard. Timing models for gauss elimination on the H1 transputer. $\mathcal{PUMA}$ Working paper 11, Centre for Mathematical Software Research, University of Liverpool, May 1990.

[5] Gabriel Howard. Adaptive quadrature on $\mathcal{PUMA}$ architectures. $\mathcal{PUMA}$ working paper, Centre for Mathematical Software Research, University of Liverpool, October 1991.

[6] Gabriel Howard. Sparse matrix vector multiplication on the H1. $\mathcal{PUMA}$ Working paper 28, Centre for Mathematical Software Research, University of Liverpool, July 1991.

[7] INMOS. H1 transputer: Advance information. Technical report, June 1990.

[8] David May. Implementing full occam. $\mathcal{PUMA}$ Working paper 2, INMOS, July 1989.

[9] Tim Oliver. The bitonic sort on transputer architectures. $\mathcal{PUMA}$ working paper, Centre for Mathematical Software Research, University of Liverpool, October 1991.

[10] Tim Oliver. Parallel algorithms for the BFGS update on a $\mathcal{PUMA}$ machine. $\mathcal{PUMA}$ Working paper 32, Centre for Mathematical Software Research, University of Liverpool, September 1991.

[11] Rémy Roire. Full `occam` implementation report. $\mathcal{PUMA}$ Deliverable 4.1.1, SYSECA, October 1990.