Writing numerical applications for the T9000 *

Rod Cook Gabriel Howard Tim Oliver
Bruce Stephens
Centre for Mathematical Software Research,
University of Liverpool,
Liverpool, L69 3BX

Abstract

In T8 systems, the physical configuration of the network is of great importance; applications are written to run on rectangular grids of transputers, or rings, or trees. Much effort for some current applications is spent on writing routines to perform communications between non-adjacent transputers. The authors are investigating numerical algorithms for a machine where point-to-point communications via channels is provided in hardware, such as INMOS's T9000 Transputer and C104 switch chip. Such a system will allow applications which use processors connected in any *virtual* topology, and will allow such applications to run regardless of the *physical* configuration, although performance on different configurations will in general vary.

1 Introduction

Networks of T9000 transputers are formed by connecting together the transputers with the C104 switch-chips; it is really the configuration of C104 that defines the network topology. Such networks provide automatic routing of messages in such a way that although each T9000 has only four links, a large number of channels may use these links and may connect through the switch-chips to any other T9000.

With practically arbitrary numbers of channels even between different, non-adjacent transputers, it becomes easier to write applications in terms of topologies which fit the problem, rather than in terms of topologies which are physically easy to realise. It is possible to write Occam code without worrying about the constraint of four links per transputer, and without worrying about whether the processes can be configured so that communication is between adjacent transputers. Code is inherently less dependent on specific details of the problem and hardware, and so is potentially much more reusable. Similar functionality can be achieved using various software tools, for example CStools, Tiny, Express, etc. However, using virtual channels is more natural in Occam than using named processes as required by the other systems.

Another consequence of virtual channels is the ability to use several such virtual topologies in a single algorithm, for example a rectangular grid for the bulk of a computation together with an n-ary tree for summation. In particular, it is straightforward to implement an application that requires the use of two, or more, virtual topologies at the same time. This is

*This work was supported by the Commission of the European Community under the PUMA P2701 ESPRIT 2 project

```
[no.of.slaves]CHAN OF message to.slave :
PLACED PAR
  PROCESSOR 0 TB
      master ( fs, ts, to.slave )
  PLACED PAR i = 0 FOR no.of.slaves
    PROCESSOR (i+1) TB
      slave ( to.slave[i] )
```

Figure 1: Configuration File for Data Distribution

useful for hiding message latency when two parts of an application, which both require communication with other processes, can be run in parallel: The `T9000` style of coding allows this to be expressed in a natural way.

In order to execute the code on a particular `T9000/C104` system there does, of course, need to be a physical configuration level. There are two configuration levels, one specifies the number of `T9000`s and `C104`s and their physical connections and is not the user's responsibility. The other, written by the user, specifies the number of processes and assigns the appropriate code and channels to them. This configuration level is written in Occam and requires no knowledge of the physical configuration. Thus, matching a program to a physical topology becomes a *performance* problem only—programs can run on any network, but they will run at different speeds on different networks.

These points are illustrated using a virtual tree topology for a dot product, a multidimensional grid topology for a sparse matrix multiply and using these two routines to build an iterative solver for an elliptic partial differential equation. Occam code fragments under the Virtual Channel Router (VCR) [2] are used to illustrate the style of programming and the ease with which applications that require complicated topologies can be written.

2 Example Algorithms

In this section we give code fragments for distributing a vector to a set of processes, a distributed dot product and matrix vector multiply, each of which require a different process topology. We then show how simple it is to combine these building blocks together to form a complete application.

2.1 Distribution of Data

To distribute a vector from a master process to a set of processes we can use a virtual channel from the master to each slave. The master splits up the vector into appropriate slices and sends them down a virtual channel to each slave. These channels are defined and passed to the master and slaves processes at the configuration level, see figure 1. Note that it is not necessary to specify the underlying processor topology or how the processes are to be assigned to the processors.

2.2 Dot Product

The dot product is a common component of many numerical algorithms, its calculation on a distributed memory multi-processor is best performed on a tree of processors. Once the data

```
[(tree.span+1)*(no.of.slaves+1)] CHAN OF REAL tree:
PLACED PAR
  PROCESSOR 0 TB
      master ( fs, ts, [tree FROM 1 FOR tree.span] )
  PLACED PAR i = 0 FOR no.of.slaves
    PROCESSOR (i+1) TB
      slave ( tree[i+1] , [tree FROM (tree.span*(i+1))+1 FOR tree.span] )
```

Figure 2: Configuration File for Dot Product

```
PROC slave( CHAN OF REAL to.parent , [] CHAN OF REAL from.child )
  ... declarations
  ... get data (i.e. x and y) from master
  SEQ
    sum := zero
    PAR
      SEQ i=0 FOR r
        sum := sum + (x[i]*y[i])
      PAR j=0 FOR tree.span
        IF
          ((tree.span*me)+(j+1)) <= no.of.slaves
            from.child[j] ? tmp[j]
          TRUE
            tmp[j] := zero
    SEQ j=0 FOR tree.span
      sum := sum + tmp[j]
    to.parent ! sum
```

Figure 3: Slave for Dot Product

has been distributed, each slave forms its own sub-dot product in parallel with receiving the results from its children, these are then summed and sent to the slave's parent. The code on the master is similar but it only receives the results from its children and forms the final result. Outline codes are given for the configuration level and slave in figures 2 and 3 respectively. At the configuration level the tree is obtained by declaring a sufficiently long array of channels and passing appropriate slices to the master and slaves. Note that the slaves at the leaves of the tree have dangling channels and the user must ensure that they are not used. It is possible to write a configuration file which does not leave any dangling channels; it is, however, more complicated than the example given here. Each slave checks whether or not it is a leaf of the tree and if not gets the results from its children.

On existing Transputer machines a tree of processors with span three is achievable in hardware. However, difficulties arise if the application requires another topology for a different part of the calculation. An application illustrating this point is given in section 3.

2.3 Matrix Vector Multiply

A matrix vector product is another common component of applications. In this section we consider a particular form of matrix arising from a finite difference approximation to an

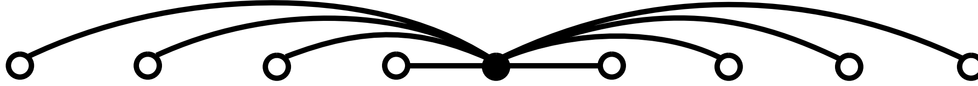


Figure 4: Topology for efficient matrix-multiplication

```
[no.of.slaves+1][dim.beta] CHAN OF real.vector.channel forwards,backwards :
PLACED PAR
  PROCESSOR 0 TB
      master ( fs, ts )
  PLACED PAR i = 0 FOR no.of.slaves
    PROCESSOR i+1 TB
      to.right   IS forwards[i+1] :
      from.right IS backwards[i+1] :
      to.left    IS [ARRAY k = 0 FOR i OF backwards[i- k ][ k ]] :
      from.left  IS [ARRAY k = 0 FOR i OF forwards[i- k ][ k ] ]:
      slave ( from.left , to.left , from.right , to.right )
```

Figure 5: Configuration File for Matrix Vector Product

elliptic partial differential equation. These matrices are symmetric positive definite with a number of diagonals offset from the main diagonal. Suppose that the matrix and vector are distributed by rows, then each slave needs to get components of the vector from other slaves to complete its calculation. For the structure of the matrix given here this can be achieved by the topology given in figure 4.

3 An Application

The Preconditioned Conjugate Gradient Method is an iterative method for solving large sparse systems of linear equations. In particular, there are two dot products with the consequent broadcast of the results; this requires all the processors to synchronise while the resulting communication cannot be overlapped with any computation.

In [1] PCGM was reformulated to increase the amount of algorithmic parallelism in the method and a task graph is given in figure 6.

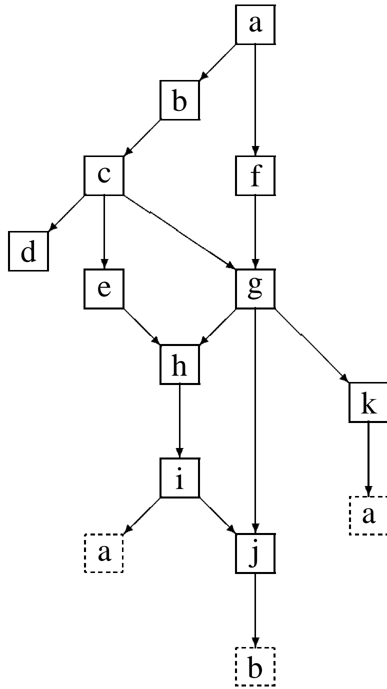
From the task graph we note that some of the building blocks in the method can be executed in parallel. In particular, we note that:

- The communication and computation of (b), (c), (d) and (e) can be performed in parallel with that of (f);
- The communication and computation of (h), (i) and (j) can be performed in parallel with that of (k).

This formulation now allows the overlap of communication and computation and hence increases the available parallelism. In particular, the communication associated with calculating the inner products and distributing the answer can be overlapped with computation elsewhere.

We give a configuration level code for the reformulated PCGM in figure 7. To highlight the required functionality many of the computational details are omitted.

There are three different virtual connectivity topologies in this implementation.



Task	Operation
a	vector update
b	dot product
c	scalar operation
d	vector update
e	vector update
f	matrix vector product
g	vector update
h	dot product
i	scalar operation
j	vector update
k	matrix vector product

Figure 6: Task Graph for Reformulated PCGM

```
[no.of.slaves]CHAN OF message to.slave,from.slave :
[(tree.span+1)*(no.of.slaves+1)] CHAN OF REAL tree:
[no.of.slaves+1][dim.beta] CHAN OF real.vector.channel forwards,backwards :
PLACED PAR
  PROCESSOR 0 TB
      master ( fs, ts, to.slave,  from.slave ,
               [down.tree FROM 1 FOR tree.span] ,
               [up.tree FROM 1 FOR tree.span] )
  PLACED PAR i = 0 FOR no.of.slaves
    PROCESSOR i+1 TB
      to.right   IS forwards[i+1] :
      from.right IS backwards[i+1] :
      to.left    IS [ARRAY k = 0 FOR i OF backwards[i- k ][ k ]] :
      from.left  IS [ARRAY k = 0 FOR i OF forwards[i- k ][ k ] ]:
      slave ( from.slave[i] , to.slave[i],
              up.tree[i+1] ,
              [up.tree FROM (tree.span*(i+1))+1 FOR tree.span] ,
              down.tree[i+1] ,
              [down.tree FROM (tree.span*(i+1))+1 FOR tree.span] ,
              from.left , to.left , from.right , to.right )
```

Figure 7: Configuration File for PCGM

- The communication of the initial data to the slaves requires a virtual channel between the master and every slave.

- The calculation and subsequent broadcast of a dot product requires the processes to be connected in a tree of virtual channels. The best span for this tree is an open question, the answer depends on many considerations, such as, the underlying hardware topology, the ratio of communication to computation, etc.

- The virtual topology required for calculating the matrix vector product depends upon the structure of the matrix and distribution amongst the processes of the matrix. See [3] for a discussion of matrix vector product under different assumptions about the structure and distribution of the matrix.

4 Conclusions

On the `T9000` and `C104` hardware each portion of the code can be written assuming multiple virtual topologies. The hardware through routing facilities of the `T9000` and `C104` give an efficient realisation of the required virtual topologies. This both increases efficiency and reduces code development costs.

References

[1] R. Cook. A reformulation of the preconditioned conjugate gradients sutiable for parallel compuation. In *IMACS Symposium on Iterative Methods for Linear Systems*, 1991.

[2] Mark Debbage, Mark Hill, and Denis Nicole. Virtual channel router. Puma Deliverable 3.1.1, University of Southampton, October 1990.

[3] Gabriel Howard. Sparse matrix vector products on a puma machine. PUMA working paper, Centre for Mathematical Software Research, University of Liverpool, June 1991.