APPLICATIONS OF LARGE TRANSPUTER ARRAYS

K.C. Bowler

Physics Department, University of Edinburgh,
Edinburgh, Scotland, UK

ABSTRACT

In these lectures we discuss some of the practical issues involved in using large arrays of transputers for problems in science and engineering. The discussion centres on experience with the Edinburgh Concurrent Supercomputer (ECS), a large Meiko Computing Surface, comprising at present some 400 Inmos T800 Transputers. We first describe the architecture of the ECS and the user interface. We then focus on the various types of parallel program, illustrating the key ideas with examples taken from the applications and demonstrations which have been successfully mounted on the ECS. Finally we address some of the wider issues involved in exploiting distributed memory, multi-processor machines.

1 THE ECS PROJECT

The Edinburgh Concurrent Supercomputer (ECS) is a transputer-based supercomputer, established in collaboration with the Bristol-based company, Meiko, to provide a facility for front-line research applications in computational science.

1.1 Background

We recall briefly the key features of the transputer and of the occam language, describe general features of the Meiko Computing Surface and give a brief history of the project.

1.1.1 Transputers and occam

Other contributors have discussed in some detail the transputer and its native programming language, occam. Here we merely remind you that the transputer implements the process model of concurrency, expressed through occam, which is based upon the idea of communicating sequential processes [1]. Communication between processes is effected by uni-directional channels, which may connect processes on the same transputer or on different transputers. Each transputer link implements two such channels, one in each direction. Channel communication between two processes running on the same transputer is effected by writing to and reading from a memory location. The basic processes, from which all others are constructed, are assignment of a value to a variable, output of a value down a channel, and input of a value from a channel.

1.1.2 Meiko Computing Surface

Physically, a Computing Surface is contained in one or more cabinets or modules, each housing a mixture of boards, chosen from a hardware 'library', to meet particular

requirements. For example, the M40 module has 40 slots, at least one of which is occupied by a local host board carrying a transputer, several Mbytes of DRAM and various interfaces for connection to external devices. The local host performs maintenance tasks such as monitoring for hardware errors, control of the electronic routing network, hardware reset and may also run the interactive program development environment. Other slots may be occupied by compute boards, which carry 4 T800 transputers, each with from 1 to 8 Mbytes of memory, by display boards, with a single transputer plus dual-ported video memory, or by mass store boards which also feature a single transputer but with up to 32 Mbytes of memory plus a SCSI interface.

The Computing Surface uses custom VLSI switching chips which permit the user to select, in software, the interconnection topology appropriate for a particular application, subject of course to the constraint that each transputer can be directly connected to at most four others. General information about the transputer, the occam language and the Computing Surface may be found in [2].

1.1.3 Project origins

The ECS Project was originally conceived in the Physics Department of the University of Edinburgh during 1986 by members of the Theory & Computation group. Six years experience in parallel computing, exploiting initially the ICL Distributed Array Processor (DAP) at Queen Mary College in London, and from 1982, two dedicated DAPs at Edinburgh resulted in over 180 publications [3] across a range of fields, extending significantly beyond those areas of physics for which support had initially been given by the UK Science & Engineering Research Council. Anticipating the decommissioning of the DAPs in 1987 we were convinced that the only way we would have access to the required power with the budgets we might expect was through exploiting novel architecture parallel machines. We were fortunate to obtain one of the earliest Meiko Computing Surfaces in April 1986, with the support of the UK Department of Trade & Industry and the Computer Board. This demonstrator system consisted of 40 T414 transputers each with $\frac{1}{4}$ Mbyte, along with a display system, and was file-served and networked through a MicroVAX host. The reliability of this system, the imminent loss of the DAPs and a survey of existing parallel machines formed the the cornerstone of the proposal for the Edinburgh Concurrent Supercomputer.

The proposal, in collaboration with Meiko, sought some £3.4M from the SERC, DTI and Computer Board to fund a machine built around 1024 T800 transputers, each with 1 Mbyte of memory, to provide an electronically reconfigurable multi-user resource. Phase 1 funding for the machine infrastructure and compute resource of 200 T800s, each with 4 Mbytes, was secured during 1987, multi-user service for code development was established in September 1987, and the first T800 compute resource installed at the end of that year.

1.2 Present status

We now describe the current configuration of the machine, the user interface and give an indication of the kind of performance that is attainable in applications.

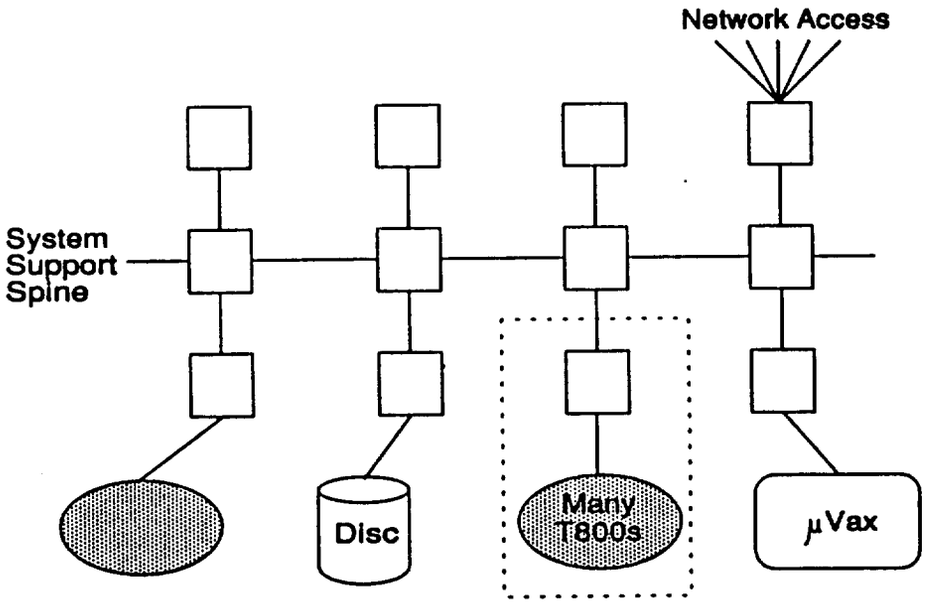

Figure 1: Schematic diagram of Computing Surface organisation

1.2.1 ECS configuration

Phase 2 DTI support has enabled us to acquire another 195 T800s and 900 Mbytes of memory, bringing the total compute resource up to 395 T800s. Meiko have also contributed very significantly in discount, maintenance and software; in addition they site two people at Edinburgh and have considerable in-house software activity to meet the Project requirements. At the time of writing (July 1989), there are a total of 14 personnel in the core of the Project (i.e. excluding specific application teams), including two on a part-time basis; the use of the income from an Industrial Affiliation scheme in supporting 7 of these people has, with the Meiko support, been a crucial factor in the successful launch of the Project.

The machine organisation is shown schematically in Fig. 1. The operating system, MMVCS, which stands for Meiko Multiple Virtual Computing Surfaces, supports a model in which the Computing Surface appears to be a network of workstations, known as *domains*, and *fileservers*. Each domain has a host processor, typically a T414, and a local network of zero or more T800s. The domains can be of arbitrary size and shape, and can incorporate special components such as graphics boards, mass store boards or special high-bandwidth i/o devices. They are connected by a transparent communication spine, the Computing Surface Network, which is also based on transputers and to which one or more fileservers are attached. The microVAX host of the original demonstrator system is now retained as one of the file servers, and to provide a VMS environment. The user can also file serve off a number of Hewlett-Packard discs running MEiKOS, Meiko's UNIX[1]-based operating system, back-up for which is provided by Exabyte cartridges. Terminal access is provided by direct lines into the spine and from JANET via a reversed PAD, soon to be replaced by ethernet. The size and number of domains is controlled by the system manager, and may be changed

[1]UNIX is a trade mark of AT&T Bell Laboratories

(exploiting the software reconfiguration) for example to match day and night time user needs. At present there are typically more than 20 domains on the service machine; a separate machine for system development can support a further 8 users.

1.2.2 User interface

At login, the user is presented with a menu of domains; s/he boots an available domain and connects to a file server and has then a personal reconfigurable parallel machine, a virtual Computing Surface. Further details may be found in an ECS Technical Note [4]. MMVCS provides the user with the occam programming system, OPS, and UNIX file-serving utilities. There are also C and Fortran compilers for single transputers and a range of utilities is available or under development at Meiko. At the time of writing, a diskless version of MEiKOS to run on the host transputer of each domain, is mounted and under test on the development machine. Among the work being performed on the development machine is a porting of the AT&T System V utilities, which is close to completion. Other standard packages which have been ported include GKS.

There has been considerable effort at Edinburgh to develop utilities which provide greater flexibility and ease of porting of codes to the Computing Surface. The cornerstone of this effort is the development of fast, topology-independent message-passing systems [7,8] of which we shall have more to say later.

1.2.3 Node Performance

Reasonable performance on a single node is obviously an important prerequisite for supercomputer performance across an array. We summarise here experience gained in a range of applications.

For well-structured Fortran or C code which is floating-point intensive, benchmarks for single precision give up to 0.6 or 0.7 Mflops per node. A number of applications written in occam are running at in excess of 1 Mflops per node.

To achieve maximum performance with minimum effort, BLAS1 routines have been written in assembler for a single T800 [D. Roweth and L.J. Clarke, unpublished]. The table below illustrates the performance obtained in these routines.

Table 1
Performance figures for BLAS1 routines

routine	Mflops	routine	Mflops
saxpy	1.17	daxpy	0.72
sdot	1.17	ddot	0.78
snorm	1.58	dnorm	1.05
sscal	0.78	dscal	0.49
ssum	1.35	dsum	1.03

2 PARALLEL PROGRAMS

Although occam provides the means for expressing process level parallelism within a single transputer and processor level parallelism within networks of transputers, and

the syntax for each is very similar, it should be clear by now that there are major differences between the implementations of each. For example

- the connectivity of a processor is currently restricted to eight channels arranged as four bi-directional pairs, whereas a process may have arbitrary connectivity
- the channel communication between processors is performed orders of magnitude more slowly than communication between processes on the same transputer
- processor level parallelism affords significant speed-up over serial processing whereas process level parallelism tends to introduce unnecessary process-switching overheads.

Perhaps the easiest way that we might envisage using a multi-processor, distributed memory machine is to run multiple copies of the same, or perhaps different, programs, one per processor, as independent, non-interacting tasks. Each processor is thus used as a batch queue server. At the other extreme, all the processors may co-operate in a single program and require complex inter-processor communications.

2.1 Paradigms

At present there is no completely general framework for discussing parallel programming. However, there have been useful attempts to identify strategies for exploiting multi-processor machines and here we focus on two such.

2.1.1 The Southampton Model

The Southampton group [9] has devised a model which identifies three types of parallelism: *event, geometric* and *algorithmic.*

Event parallelism also known as *task parallelism*, describes the situation in which a problem can be divided up into independent tasks, which may be allocated to processors, either statically or dynamically, and results collected up in some manner. The parallel batch server belongs in this category. The only inter-processor communication required is usually just the passing on of tasks and results.

The canonical example of event parallelism is *ray tracing*, a rendering technique for generating images on a graphics screen. The basic idea is to reproduce what happens in a pinhole camera. The image on the screen is built up from rays of light emanating from objects in a 3-dimensional 'world'. These rays may be identified by starting at a pixel on the screen and tracing back through a hypothetical pinhole, on to a surface in the 'world'. Each ray is then reflected backwards to determine whether it comes from another surface or from a light source. This backward tracing continues until the ray ends at a light source or passes out of the 'world'. Once the source of a ray has been determined, the path of the ray is retraced from the source to determine the colour and brightness of the corresponding pixel on the screen. The complete picture is built up by tracing one ray for each pixel. The paths of the reflected rays and the levels of illumination depend on the nature of the light sources and on the type of reflecting surface; refraction can also be incorporated.

Figure 2: A ray-traced image produced on the ECS

The ray tracing algorithm involves a great deal of computation, but the key point is that the rays can be traced *independently* and so the method lends itself to the simplest type of parallel computation. An example of a ray-traced image is shown in Fig. 2

Geometric parallelism perhaps better described as *data parallelism*, caters for the case where a program operates on some large data space in such a way that essentially the same operation is performed in every region of the space. The data can then be distributed over a number of processors, with each processor operating on that portion of the data held in its own memory. Unlike the previous category, interprocessor communications will usually be necessary during the computation to access data held on other processors.

Many physics simulation problems are of this kind. The simplest example is probably the two-dimensional, nearest-neighbour Ising model which is widely used in condensed matter physics and statistical mechanics. It models the behaviour of ferromagnets, of binary alloys and of certain fluids, for instance. It consists of spins, s_i, which live on the sites of a square lattice and are binary variables taking the values ± 1 only. A particular arrangement of spins is known as a *configuration.* The energy of a configuration of spins, $\{s\}$, is defined by the Hamiltonian, or energy function, H:

$$H(\{s\}) = -J \sum_{<ij>} s_i s_j$$

In thermal equilibrium at a temperature, T, the configurations are distributed according to the Boltzmann probability distribution:

$$p(\{s\}) = \frac{1}{Z} \exp(-H(\{s\})/kT)$$

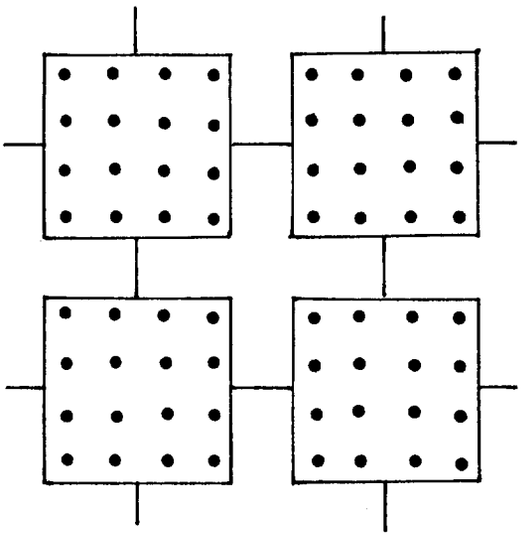

Figure 3: Geometric decomposition of the Ising model

where the normalisation factor, Z, is known as the *partition function* and is given by

$$Z = \sum_{\{s\}} \exp(-H(\{s\})/kT).$$

Macroscopic properties of interest, such as the average energy of the lattice, are computable by averaging over the ensemble of configurations:

$$< H >= \sum_{\{s\}} H(\{s\})p(\{s\}) = \frac{1}{Z} \sum_{\{s\}} H(\{s\}) \exp(-H(\{s\})/kT)$$

Direct evaluation of averages by explicit summation over all configurations, although possible in principle, is ruled out on practical grounds for any but the tiniest of lattices - the number of configurations grows exponentially with system size. Monte Carlo methods are therefore used to generate a sequence of configurations distributed according to the Boltzmann law by allowing each spin to evolve according to a set of rules based upon the status of neighbouring spins, for example, the so-called Metropolis algorithm [10]. Averages may then be estimated by taking a simple arithmetic mean, over the ensemble of configurations so generated, of the desired observable.

Fig. 3 illustrates the geometrical decomposition of the Ising model lattice over an array of transputers. Spins which lie in the interior of the sub-lattice stored on each transputer may be updated trivially but those spins which lie on the boundary of a sub-lattice need information about the state of spins which lie on the boundary of nearest-neighbour sub-lattices, stored on adjacent transputers, and hence require inter-processor communication.

The importance of this example cannot be overstated. Almost identical methods are used to simulate quantum field theories, such as QCD, numerically. The spin degrees of freedom of the Ising model are replaced by quark and gluon fields, and the action, S,

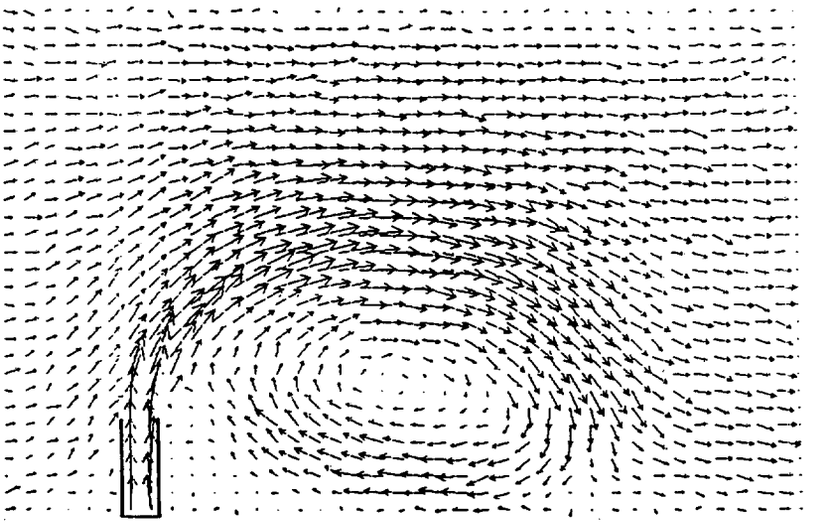

Figure 4: A cellular automaton fluid flow simulation

of the field theory plays the role of the Hamiltonian. The thermal fluctuations of the spin system become the quantum fluctuations of the field theory.

Similar ideas underpin an implementation of *lattice gas hydrodynamics* on the ECS. Two-dimensional fluid flows are represented by cellular automata consisting of particles moving on a triangular lattice. The system evolves in discrete time-steps in which first each particle moves from one lattice site to the next in the direction of its velocity vector, then it collides with any other particles which arrive at the same site, conserving particle number and momentum. Hydrodynamic variables are computed by averaging over the particles in subregions of, say, 20×20 sites. This coarse-graining procedure can be shown to model the Navier-Stokes equations of fluid flow. Fig. 4 shows the result of simulating flow along a channel by this method on the ECS [11]. The channel is decomposed into strips transverse to the flow and each strip is processed by a different transputer, the processors being connected in a chain.

A slightly less obvious application is the problem of N planets with fixed masses and given initial positions and velocities, moving under Newtonian gravity, a *long-range interaction* where every particle interacts with every other. A strictly geometric decomposition, in which the particles in different sub-regions of space are associated with different processors, is no use. However, the *data* space consists of the positions and velocities of the planets and may be divided amongst the available processors, so that each is given the job of following the time evolution of a subset of the particles, which may be at widely scattered locations. This is illustrated in Fig. 5. The algorithm may be mapped on to a closed ring of transputers. Each processor picks one of its subset of particles and sends its mass and coordinates to the next processor in the ring. Each processor then computes the force on each of its particles due to the incoming 'travelling' particle, and then passes on the information about the travelling particle. This procedure is repeated until every particle has 'visited' every processor. This will

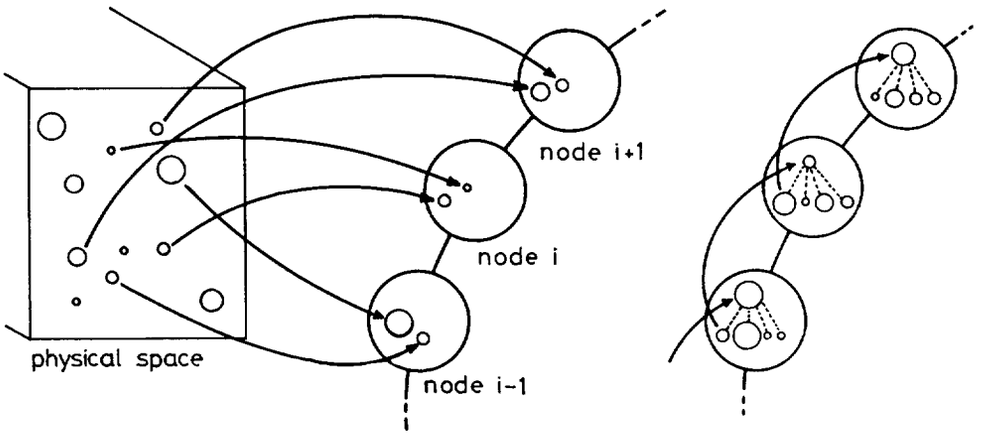

Figure 5: Algorithm for system with long-range forces

be efficient provided that the time taken to communicate a planet's data between processors is small compared to the time taken to do the force calculations. This becomes easier as the number of planets per processor increases, since the force calculation grows as N^2, whereas the communication time grows like N. Indeed, for a transputer system it should be possible to overlap communication by computation completely.

Algorithmic parallelism is analogous to the production line in a factory. The operations performed by a program, rather than the data, are divided between the processors. Each processor will receive a copy of part or all of the data space, perform its operations on that data and pass on the resulting modified data to another processor or processors. Typically such programs require a more complex control structure, and are consequently more difficult to write and debug. Load balancing is also a problem, since the performance is limited by the speed of the slowest element.

An important example of algorithmic parallelism is the graphics pipeline. The data space consists of a large number of polygons, representing some object to be displayed. The polygons will need to be scaled, rotated and translated to transform them from world coordinates to viewing coordinates before rendering to produce pixel rasters. Clipping and hidden surface removal are then applied, and the image displayed. Each of these operations may be allocated to a different processor, with each processor taking its input from the previous stage, applying its operation and passing the results on to the next stage.

Combinations It is clear that we can combine these three types of parallelism in a great variety of ways. For example, the ray-tracing application combines event and algorithmic parallelism in an obvious way. More generally, the *master-slave* model in which the bulk of the computation is performed by a pool of slaves, with a master which handles global decisions and i/o, is a very common method employed in physics

simulations based upon geometric decomposition, such as the Ising model simulation described earlier.

2.1.2 The Caltech model

An alternative classification of parallelism has been developed by the Caltech group [12], based upon the *granularity, connectivity* and *synchrony* of the application.

Granularity refers to the *available parallelism.* For example, in ray tracing, each ray may be regarded as a grain. Similarly, in Newtonian dynamics, each planet may be thought of as a grain. Applications generally display grains at several different levels. The abundance of grains often far exceeds the number of available processors and it is important to consider how grains can be combined to form larger grains, so that the granularity of the application matches that of the machine, thus minimising the overheads associated with process level parallelism on individual processors; optimal performance usually dictates that a single compute process should run at a time on a given processor.

Connectivity is the number of interactions of a grain. Given the grain size, this connectivity can be measured. For example, in the simple Ising model, if the grain is taken to be an individual spin, the connectivity is simply the *co-ordination number*, the number of nearest-neighbour spins, which is 4 in two dimensions. As a general rule it is useful if the connectivity of the grain matches the connectivity of the processors. Matters are also simplified if the connectivity is regular so that the application can be mapped on to a regular array of processors. Referring again to the Ising example, it is easy to see that if the spins are grouped into grains consisting of regular, connected regions, the connectivity remains 4. The planetary example on the other hand has global connectivity because each planet needs to know the position of every other planet, and this remains true if the planets are clumped into larger grains.

Synchrony may be thought of as representing connectivity in execution time. Fox talks of the Ising model as a synchronous *application* because each spin may be updated simultaneously on each timestep (actually, only half of the spins - all even sites or all odd sites - may be updated simultaneously in order to satisfy a fundamental requirement of statistical mechanics known as detailed balance). There is thus microscopic synchronisation between grains. However, in the geometric decomposition described earlier, the *implementation* is only loosely synchronous, in the sense that it is only the exchange of boundary spin values between neighbouring transputers which imposes any restrictions on the time sequence of the processing. On the other hand, when there are load imbalances associated with the grains, the *application* itself is loosely synchronous. All implementations of synchronous or loosely synchronous applications show loose synchrony.

There is a class of applications where, although loose synchrony is present, a global synchronisation step is required. The conjugate gradient algorithm for matrix inversion is an example, containing a matrix-vector multiply which may be distributed over many processors, but also involving the calculation of a global scalar product combining results from each processor. Such applications do not fall neatly into the Caltech

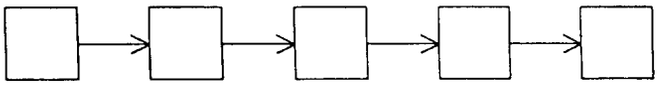

Figure 6: Loading data on to a chain of transputers

classification and have been dubbed *synchronised* rather than synchronous by Clarke and Norman [6].

3 GENERAL CONSIDERATIONS

3.1 Communications Harnesses

3.1.1 Why use one?

Occam encourages us to do all the communications in a multi-transputer program for ourselves. To take a trivial example, we might wish to load data on to a chain of transputers as in Fig. 6, which is easily done with the following piece of occam:

```
SEQ
  SEQ i=0 FOR num.transputers.further.on
    SEQ
      in  ? data  -- receive data and
      out ! data  -- pass it on
  in ? data       -- receive own data
```

In more complex programs there may be no predefined pattern of communication; one processor may suddenly wish to talk to another processor to which it is not directly connected. It then makes sense for the programmer to use some standard routing software, known as a *communications harness*. Even if the final program doesn't need general communications, it is a good discipline to build a prototype using a harness, thus providing a clean separation between communications and computation.

3.1.2 Background

The harnesses which have been developed in Edinburgh had their origins in a program written by Mike Norman [5] to analyse and display volume data produced by an NMR body scanner. The naïve approach to this problem would be to adopt a geometric decomposition of the data space, dividing it into equal, block-shaped chunks and giving one to each transputer. Unfortunately, part of the problem was to apply image enhancement and surface-tracking algorithms to the data, requiring processing concentrated in just one part of the volume and thus resulting in one or two transputers doing all the work - a classic case of bad load balancing. The solution adopted was to employ *scattered domain decomposition*, where each transputer is assigned many small cubes

scattered round the data set, instead of one large block. This increases the chance of each transputer having useful work to do, but means that for each cube that a transputer controls, communication with *all* the transputers controlling neighbouring cubes will be required. This led Mike to develop a general-purpose packet-switching network as part of the body-scan program. The communications element was abstracted from this work to become the Topology Independent Transputer Communications Harness, TITCH [7].

3.1.3 How to use a communications harness

To use a topology-independent communications harness, we give each of our applications processes an ID, wire our transputers together in any way we like and put the following occam on each transputer:

```
#USE "some library name"
[some.number]CHAN OF ANY to.harness, from.harness:
... other declarations
PAR
  ... harness
  ... application process
  ... another application process
  ... yet another (as many as we like)
```

Inside one of the application processes, we might send, say, sixty 32-bit words of data from the array **message** to process 7 in the following way:

```
{{{ application process
[100]INT message:
SEQ
  ...
  pktWrite(to.harness[my.chan.id], 7, message, 60)
  ...
}}}
```

The data is picked up by the harness and delivered to process 7 (wherever that may be) without involving any of the other application processes. At the other end, the data would be delivered along with information saying where the message came from and how long it is:

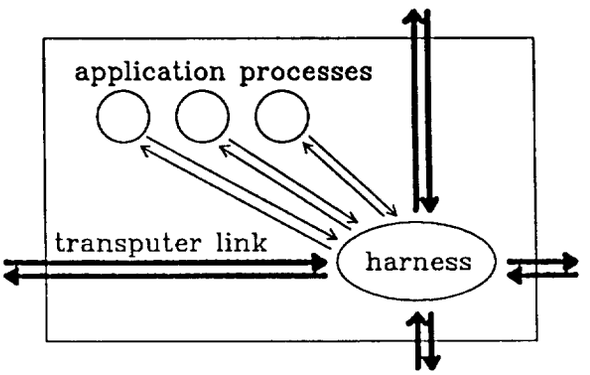

Figure 7: Program on a single transputer

```
{{{ process 7
[100]INT message:
INT message.length:
INT sender:
SEQ
  ...
  pktRead(from.harness[my.chan.id], sender, message, message.length)
  ...
}}}
```

Fig. 7 shows in outline what the program on each transputer looks like. Each application process is connected by a channel pair to the harness process. This assumes an in-tray/out-tray model, where messages are sent and received in queues. This is fine for applications which don't mind what order messages arrive in, but if the application process is waiting for a message from a particular source or of a particular type, it would need to store incoming messages in a buffer until it found the desired message. The latest harness, Tiny, thus allows any number of channel pairs to be connected to an application process.

There is an important sense in which TITCH departs from the occam model of communication; there will not necessarily be an instant when both the sender and receiver are participating in the data exchange. The ordinary packet send in TITCH is a *non-blocking* write; the sender proceeds with its program without waiting for the receiver to collect the message. This is a natural consequence of the buffering used in the packet-switched message routing. Generally, synchronised communications are not essential. Indeed, U.S. disjoint-memory machines, such as the Intel Hypercube, use a non-blocking write most of the time. When synchronisation is important, the application program can be made to include an acknowledge signal, but long-distance handshakes seriously impair performance and are not recommended for normal use.

3.1.4 Recent developments

There has been a continuing interest at Edinburgh in topologies and routing problems such as deadlock. The state of the art now is the utility called Tiny [8,13], which explores the transputer configuration at run-time and sets up point-to-point communications and broadcasts. Code does not have to be recompiled to run on different configurations. The harness also has fault tolerant capabilities; provided a booting path is available through each transputer, efficient routing between pairs will be set up even if some of the links are defective - they will simply not be utilised in setting up the routing tables. The utility can be called from Fortran or C as well as used in an occam program. Various flags permit the user to specify whether data must arrive in the same order as it was sent etc, and the size of the buffers can be varied to match the application requirements. The system has particularly good characteristics under heavy load; for example if messages arriving on link 0 cannot be forwarded on link 1 because the latter is blocked, other messages arriving on link 0 can be passed on by links 2 or 3 if these are available.

A recent major development has been the extension of this utility to provide deadlock-free communication. The utility finds the shortest distance solutions for regular graphs such as grids, and for irregular topologies the mean interprocessor communication distance is only modestly increased, for example by around 25% for a random transputer graph of 256 nodes. It should be said that in practice the original Tiny has only rarely been known to deadlock unless the user has written an incorrect program or insufficient buffers have been provided; the deadlock-free Tiny is important however both as a matter of principle and for safety-critical applications.

The start-up latency for the utility is 17 microseconds on both the send and receive processors, and the through-routing CPU overhead time is 19 microseconds per node, with a realised bandwidth of around 1.4 Mbytes per second per link. These figures compare favourably with the iPSC2 for communication over up to three links, particularly when one bears in mind that we are comparing hardware and software through-routing capability, and that they refer to a lightly loaded network.

From the user point of view, utilities like Tiny are important because they assist flexible and portable code development. It is already the basis for a number of other topology independent utilities for example for task farming and 3-d graphics It has also been used to explore the properties of irregular graphs [14] which have many attractive features, including mean interprocessor distance and diameter which increase only logarithmically with the number of (fixed-valency) nodes, and are very close to the optimal bound. Such graphs provide a framework for shared memory emulation on distributed memory machines, following the work of Valiant [15].

Although the implementation does not utilise random graphs, it should also be mentioned here that Linda has also been implemented on the ECS [16]. The work to date is in the framework of Lisp, in fact Scheme, and extensions to C and Prolog are planned over summer 1989.

3.2 Concluding remarks

The examples which have been used to illustrate the ideas outlined in these lectures represent a small fraction of the applications which have been successfully mounted on

the ECS, and have been picked primarily for pedagogic reasons from the subset of programs which lend themselves rather easily to visualisation. For more details of the wide range of problems in science and engineering being addressed, we refer to the ECS Project Directory, published annually by the Project and available on request [17].

ACKNOWLEDGEMENTS

I wish to thank my colleagues in the Physics Department and in the Edinburgh University Computing Service for continuing support and collaboration. Special thanks must go to Lyndon Clarke, Mike Norman and Dominic Prior, on whose work much of these lectures is based.

* * *

REFERENCES

[1] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall (London 1985).

[2] K.C. Bowler, R.D. Kenway, G.S. Pawley and D. Roweth, *An Introduction to Occam 2 Programming*, Chartwell-Bratt (Bromley 1987) ISBN 0-86238-137-1.

[3] K.C. Bowler, A.D. Bruce, R.D. Kenway, G.S. Pawley, D.J. Wallace and A. McKendrick, *Scientific Computation on the Edinburgh DAPs*, University of Edinburgh Report, (December 1987).

[4] G.V. Wilson, *MMVCS: A User's Guide*, ECS Technical Note 1, Edinburgh University Computing Service (1988).

[5] M.G. Norman and R.B. Fisher, in Applications Using Occam, ed. J. Kerridge, IOS Press (Amsterdam 1988) 77-82.

[6] L.J. Clarke and M.G. Norman, *Transputer Supercomputing*, in preparation.

[7] M.G. Norman and S. Wilson, *The TITCH User Guide*, ECS Project, Edinburgh University Computing Service (1988).

[8] L.J. Clarke, *The* Tiny *User Guide*, ECS Project, Edinburgh University Computing Service (1989).

[9] D. Pritchard, C.R. Askew, D.B. Carpenter, I. Glendinning, A.J.G. Hey and D.A. Nicole , in *Parallel architectures and languages*, eds. R.J. Elliott and C.A.R. Hoare, Springer Lecture Notes in Computer Science, **258** (Berlin 1987) 278.

[10] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, J. Chem. Phys. **21** (1953) 1087. See also D.W. Heermann, *Computer Simulation Methods in Theoretical Physics*, Springer-Verlag (Berlin 1986).

[11] B.J.N. Wylie, *Focus: fluid flow simulation*, Edinburgh Concurrent Supercomputer Newsletter No 2 (August 1987).

[12] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon and D. Walker, *Solving Scientific Problems on Concurrent Processors*, Prentice-Hall (New Jersey 1988).

[13] L.J. Clarke, *Focus: communications in networks of parallel processors*, Edinburgh Concurrent Supercomputer Newsletter No. 7, (April 1989).

[14] D. Prior, N.J. Radcliffe, M.G. Norman and L.J. Clarke, *What price regularity?*, Concurrency: Practice and Experience, in press (1989).

[15] L.G. Valiant, in *Scientific Applications of Multi-processors*, eds. R.J. Elliott and C.A.R. Hoare, Prentice-Hall International Series in Computer Science (1989) 17.

[16] P. Dourish, *Implementing a parallel Lisp system on a transputer array*, ECS Project report, (1989).

[17] ECS Project Directory, 1988 & 1989, Edinburgh University Computing Service.