Reconfigurable Transputer Networks: Practical Concurrent Computation [and Discussion]

A. J. G. Hey and D. May

Phil. Trans. R. Soc. Lond. A 1988 **326**, 395-410
doi: 10.1098/rsta.1988.0095

Phil. Trans. R. Soc. Lond. A **326**, 395–410 (1988) [395]
Printed in Great Britain

Reconfigurable transputer networks: practical concurrent computation

By A. J. G. Hey
Department of Electronics and Computer Science, The University, Southampton, SO9 5NH, U.K.

The architecture of a reconfigurable multitransputer machine capable of a gigaflop (10^9 floating point operations per second) performance will be described in some detail. Problems in efficiently coding such a distributed memory MIMD (multiple instruction multiple data) machine are also discussed. These problems will be illustrated with reference to some practical strategies for exploiting different types of parallelism present in many scientific and engineering problems. Some examples will be discussed in detail.

1. Introduction

Programming a scientific or an engineering problem on a conventional 'Von Neumann' computer with only one CPU (central processing unit) requires the problem to be forced into a sequential mould. The sequential nature of the solution is also reflected in the choice of programming language, usually one of the older imperative languages such as FORTRAN, C or PASCAL. None the less, it is evident that many, if not most, real-life engineering problems possess a natural parallelism which would allow many subtasks to be performed concurrently. Solving such problems with parallel multiprocessor computers will free us from the sequential straight-jacket and is clearly possible in principle. It does, however, require us to 'unlearn' our sequential programming strategies and use new (or extended) programming languages. Why should we bother?

The benefits of a parallel approach to problems are not only the dramatic reduction of hardware costs per megaflop (10^6 floating point operations per second) but also the potential for constructing massively parallel machines with a computing power far in excess of that achievable by conventional vector supercomputer technology. For example, one of the well-known supercomputer problems is computational fluid dynamics. In aircraft design, computer programs are required to calculate the airflow round the aircraft for a wide range of parameters. Rather than attempt a full solution of the Navier–Stokes equations, simplified flow codes incorporating approximations to these equations are used, which themselves require many hours of supercomputer time for each parameter setting. A full solution to the Navier–Stokes equations over a wide range of Reynolds numbers, and for changing aircraft configurations, is out of reach of present (and foreseeable) vector supercomputers. Performance in the teraflop (10^{12} floating point operations per second) range can only be achieved by using massively parallel machines.

Besides such examples of massive parallelism, the next few years will also see parallel hardware available on a more everyday basis. Powerful engineering workstations capable of present-day 'supercomputer' performance will soon come on the market and these will incorporate parallel hardware allowing real-time 3D graphics displays. The ability to visualize solutions to complex systems of equations and see how the form of the solutions vary as parameters or conditions are changed will soon become an indispensable tool of all scientists and designers.

2. Parallel programming

Programming multiprocessor machines presents new problems to the programmer. To highlight some of these problems and illustrate several useful paradigms for parallel processing we shall begin with a very simple example: 'Fox's Wall' (Fox & Messina 1987). Consider the problem of building the wall shown in figure 1. If one bricklayer takes T hours to complete the task, how long will it take four bricklayers? As is obvious to everybody, the answer will not be $\frac{1}{4}T$! With four people working on one wall there is an organizational problem to ensure that they work together constructively and do not get in each other's way. Let us consider several different ways in which we might organize this task; each method will have a direct analogy with real parallel programming problems.

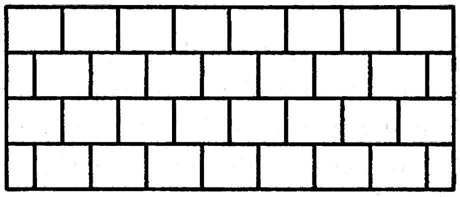
Figure 1. Fox's wall.

(a) *Pipeline solution*

The problem 'domain' is divided horizontally and each bricklayer assigned one row of bricks. They can now all work on the wall, but the efficiency will be less than 100% because the higher rows cannot be started until after the lower rows (figure 2). This solution illustrates the overheads incurred in filling and emptying a vector pipeline on vector machines: only if the wall is long enough, so that all of the bricklayers can be working most of the time, will this be an efficient method of parallelizing the problem.

Figure 2. Pipeline solution to Fox's wall.

(b) *Geometric solution*

Divide the wall up into vertical sections and assign each bricklayer one section. All workers can start at the same time but now they have the problem of synchronizing their work at the joins of the sections (figure 3). Not until the neighbours have communicated the fact that the joining bricks on the row below have been laid, can the next layer be laid. In addition to this

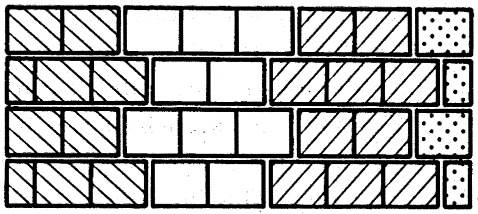
Figure 3. Geometric solution to Fox's wall.

communication and synchronization overhead, there is also a requirement for a good load balance to obtain a high efficiency. In the example shown in figure 3, there is clearly a significant inefficiency due to an uneven load distribution.

(c) *Farm solution*

In the 'farm' solution, bricks and cement are not assigned to each bricklayer independently but kept in a central resource. Each bricklayer then picks up a brick and cement, takes them to the wall and lays them in the next available position (figure 4). Once again there will be inefficiencies due to startup and finishing, but such a method is clearly capable of achieving high efficiency on an appropriate problem.

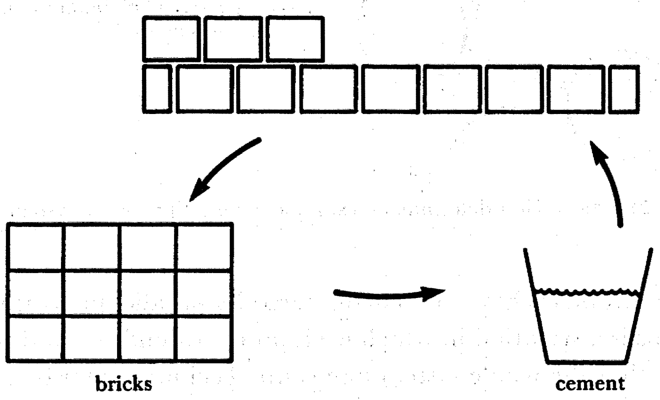

Figure 4. Farm solution to Fox's wall.

Although we have illustrated these three paradigms in a very simple context, the same techniques are found to be commonly applicable to the problem of parallelizing scientific problems for multiprocessor machines. There is, however, one other type of parallelism that will be relevant to our discussion of transputer-based machines. This I will illustrate with a 'Feynman story' (Feynman 1985). During the Manhattan project to develop the atomic bomb, Feynman was given direction of the 'IBM' team who were required to perform very complex calculations on the effect of cavity shape on the amount of energy released. This was before the days of electronic computers and the name IBM referred to their adding machines, multipliers, card punches and so on. Before Feynman took over management of the team, three problems had taken nine months to complete. Feynman raised this rate to nine problems in three months by using 'pipelining': it was possible for several different jobs to be processed at the same time. This success, however, only led to further problems. The management now assumed that each job took $\frac{3}{8}$ month and demanded the results of a vital calculation within one month! Feynman then had to solve the much more difficult problem of parallelizing a *single* problem to meet the deadline. This type of parallelism we shall refer to as 'algorithmic' parallelism. The name is meant to imply that the algorithm, not the data, is broken down into parts that can be performed concurrently. This introductory section concludes with a brief mention of two new problems that the would-be parallel programmer will encounter: non-determinacy and deadlock. Consider the simple three-processor system indicated in figure 5. Processor 3 is programmed to receive inputs from processors 1 and 2 and assign the first input to a variable called a and the second to variable b. Processor 1 sends the value 100, say, and processor 2 the value -1. Because each processor is computing independently on different data, it may not be possible to determine in advance which result will arrive first and thus which of the two possibilities ($a = 100$, $b = -1$) and ($a = -1$, $b = 100$) is selected. Such non-determinacy is an inherent property of multiprocessor machines of this type. One needs to be aware of such potential problems and, if necessary, program around them.

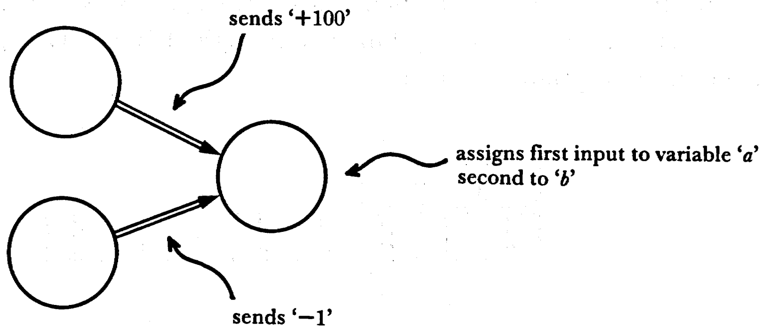

FIGURE 5. Non-determinacy example for multiprocessor system.

The most common problem, however, encountered by parallel programmers is undoubtedly that of deadlock. This is a situation in which each processor ends up waiting for an input from another processor so that the whole system hangs up. Techniques for deadlock avoidance and correct termination of parallel programs soon become a standard part of the parallel programmer's armoury. Nevertheless, mere deadlock avoidance does not guarantee an efficient implementation of a problem. Consider the following variant (Pritchard, personal communication) of Dijkstra's Dining Philosopher Problem (Hoare 1985). At the famous college, the five philosophers are all sitting round their circular table having a party. They decide to play pass-the-parcel, but each has a parcel so if each continues holding a parcel, no one is free to take a parcel from his neighbour nor can he pass his on: the classic deadlocked situation. Introduction of a butler between two of the philosophers breaks the deadlock: one parcel is passed to the butler and this effect ripples round the table. It is immediately clear, however, that a more efficient implementation can be achieved by introducing five valets. Each can now receive a parcel, and five parcels can circulate at the same time! Some strategies for deadlock avoidance are contained in Dathi (1986) and Welch (1987).

3. Transputers and occam

We shall only be concerned with a summary of the essential elements of the transputer: a more detailed discussion can be found elsewhere (Homewood *et al.* 1987). On a single VLSI (very-large-scale integration) chip the Inmos 'T800' transputer (figure 6) provides processing power, memory and communication hardware. The T800 has two processors, one a 32 bit, 10 Mips (million instructions per second) CPU and the other a floating-point coprocessor capable of 1.5 megaflops performance. The on-chip memory consists of 4 Kbytes of fast, 50 ns static RAM (random access memory), and the communication hardware comprises four fast 20 Mbit per second serial links. Both processors and all four links (each in two directions) can operate concurrently. The transputer hardware makes it easy to construct large and powerful MIMD (multiple instruction multiple data) arrays of transputers; just two wires per link are needed to provide bidirectional, point-to-point communication between transputers and no additional buffering is required.

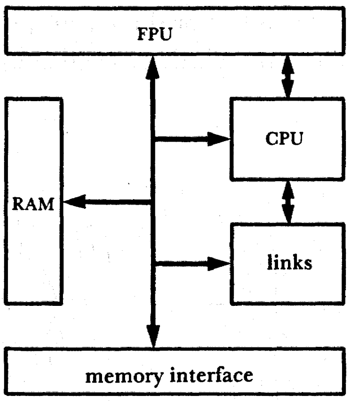

Figure 6. T800 Transputer: layout of main features.

The key question for a user, however, is whether or not such powerful distributed memory arrays of processors can be easily programmed. In fact, concurrently with the design of the transputer, Inmos also developed a programming language called OCCAM (Inmos 1988). This language embodies Hoare's communicating process model of concurrency (Hoare 1985) and incorporates communication primitives and concurrency *ab initio*. Moreover, the features present in the OCCAM language represent the result of an elegant engineering compromise between desirability of a given language construct and its ease of implementation in silicon. The transputer is therefore engineered not only to execute the OCCAM language primitives efficiently but also to support both simulated concurrency on a single processor as well as a truly distributed implementation on a network of transputers.

The OCCAM process model is shown in figure 7. The three sequential processes P1, P2 and P3 can all execute in parallel and communicate with each other via one-way communication 'channels'. Notice that this model of concurrency is very different from that embodied, for example, in Ada and in shared-memory multiprocessor machines. Here, there is no shared memory and variables can only be passed between processes via channels. This has the advantage of avoiding contention problems and provides a secure and side-effect-free multiprocessor 'system' language.

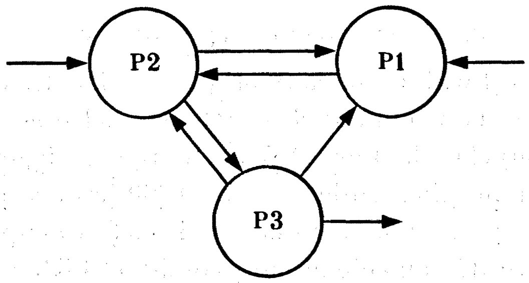

Figure 7. The occam process model: sequential processes P1, P2 and P3 and point to point communication channels.

In contrast to approaches to parallel programming in which all the parallelism is left implicit for the compiler to extract (if it can), occam requires the programmer to make the parallelism entirely explicit. Thus, for example, the three processes of figure 7 may be run on one transputer or divided between two or three transputers as indicated in figure 8. The choice between implementing the multiprocess code on two or three transputers may be dictated by issues such as load balancing, communication bandwidth or even simple economics!

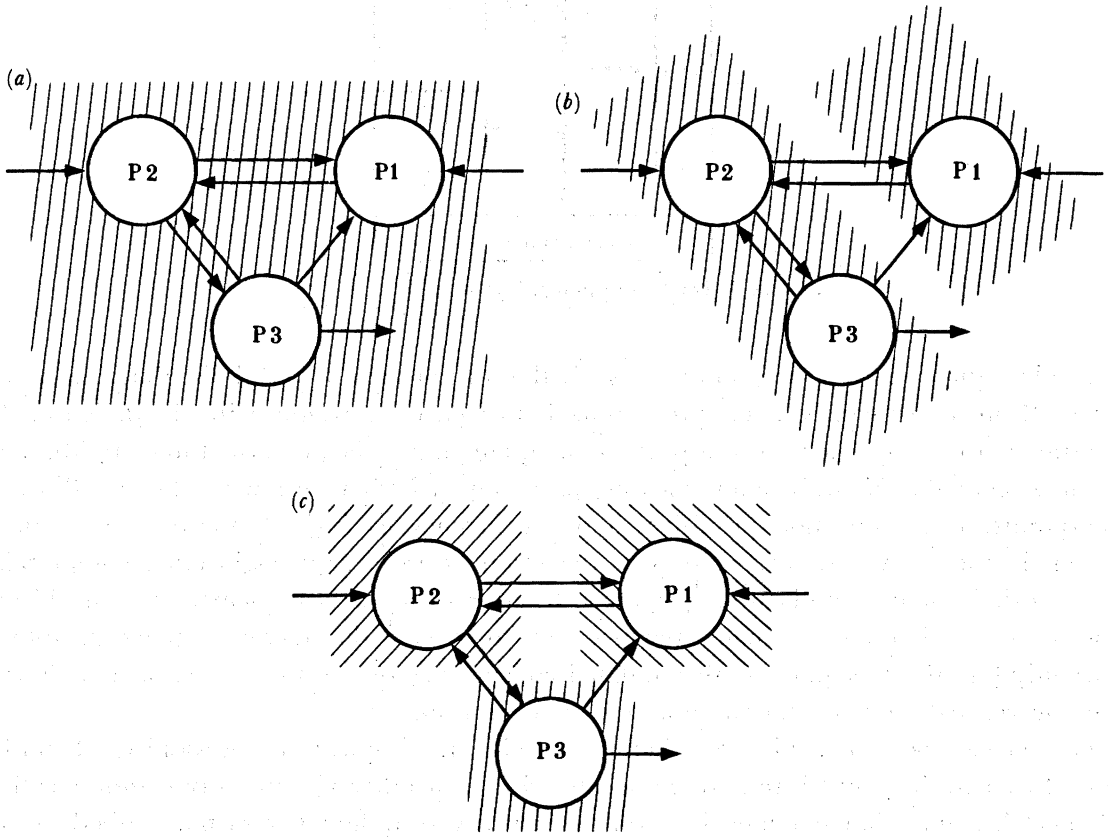

Figure 8. Simulated concurrency or true concurrency: possible distributions of processes P1, P2 and P3. (*a*) One transputer; (*b*) two transputers; (*c*) three transputers.

4. Practical methodologies

To program multiprocessor machines efficiently it is necessary to have some knowledge of both the underlying architecture and the basic hardware parameters. In this discussion we shall restrict ourselves to distributed-memory MIMD computers although some of the issues raised have more general validity. For such an N-processor machine it is convenient to define the speedup or efficiency as follows

$$E = \frac{(\text{time to compute problem on one node})}{N \times (\text{time to compute problem on } N \text{ nodes})}.$$

For conventional multiprocessor machines such as CalTech's 'Cosmic Cube' and the other hypercube machines, this efficiency formula can be rewritten in the form (Fox & Otto 1985)

$$E = T_{\text{calc}}/(T_{\text{calc}} + T_{\text{comm}}),$$

where $\quad T_{\text{calc}} = \text{total calculation time}$

and $\quad T_{\text{comm}} = \text{total control and communication time.}$

The inefficiency is thus seen to be introduced by the additional control and communication involved in distributing the problem over the N-processing nodes.

Transputer arrays fall into this category of machine with the important proviso that the transputer hardware allows communication to take place concurrently with computation. Thus, with only a relatively small increase in code complexity, part of the 'wasted' communication time, in which a conventional processor would normally not be able to get on with useful computation, can be overlapped with useful computation. Thus, for transputer-based multiprocessor machines we may write

$$T_{\text{comm}} = T_{\text{setup}} + T_{\text{overlap}},$$

where T_{setup} comprises non-overlappable channel set-up and other overheads, and T_{overlap} consists of communication time that can be overlapped with calculation. Thus, for transputer arrays, we expect to be able to achieve higher efficiencies because now

$$E = T_{\text{calc}}/[T_{\text{setup}} + \max(T_{\text{calc}}, T_{\text{overlap}})].$$

To examine the validity of this (somewhat simplified) analysis let us look at an explicit example. We shall also use this and other examples to show the application of the three programming paradigms of geometric, algorithmic and processor farm parallelism.

We begin by considering a typical grid problem: Laplace's equation in two dimensions. Our analysis will focus on distribution techniques using a simple relaxation method rather than on a search for the most efficient parallel algorithm. We wish to solve Laplace's equation

$$\nabla^2\phi = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\phi = 0$$

in a square region with fixed non-zero boundary potentials. The laplacian may be approximated by finite differences on a uniform grid leading to the Gauss–Jacobi relaxation algorithm. In an obvious notation the updated field at the grid point (n, m) is given by

$$\phi(n, m) = \tfrac{1}{4}\{\phi(n+1, m) + \phi(n-1, m) + \phi(n, m+1) + \phi(n, m-1)\}$$

corresponding to the update 'stencil' shown in figure 9. We now consider both geometric and algorithmic implementations of this algorithm.

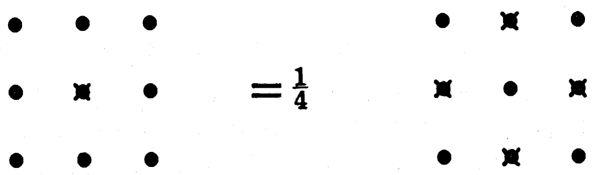
FIGURE 9. Update stencil for Laplace's equation.

(a) *Geometric parallelism*

Each of the N processors is assigned a subregion containing n gridpoints so that the whole domain consists of $N \times n$ points. As is evident from figure 10, the total calculation in each subregion is proportional to n, the area of the region, while the communication with neighbouring processors to obtain the necessary data to update the edge points is proportional to $\surd n$. Thus we expect

$$T_{\text{calc}} \sim n, \quad T_{\text{comm}} \sim \surd n$$

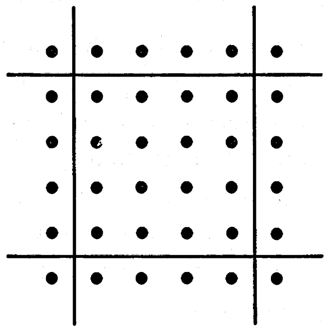
FIGURE 10. Geometric decomposition of Laplace problem.

and, for large enough n, high efficiency is assured because the n dependence has the form

$$E \sim 1 - A/\surd n,$$

where the coefficient A is specific to the particular multiprocessor hardware. As shown by Fox & Otto, these arguments generalize to higher dimensions and to a surprisingly wide range of problems (Fox & Otto 1985). We are concerned with a multitransputer implementation of this 'domain decomposition' technique.

Consider a 4×4 array of transputers connected as a regular two-dimensional grid. To map 130×130 grid with fixed boundary conditions on to this array, each processor is assigned a 32×32 subregion. To examine the effect of overlapping communication with calculation we mimic the effect of slowing down the communication speed by communicating the data M times. Writing

$$E = T_{\text{calc}}/D(M)$$

we have, for the non-overlapped case

$$D_1(M) = MT_{\text{comm}} + T_{\text{calc}},$$

and for the overlapped situation

$$D_2(M) = MT_{\text{setup}} + \max(MT_{\text{overlap}}, T_{\text{calc}}).$$

The results are shown in figures 11 and 12. For the non-overlapped case we see the expected linear dependence on M with slope T_{comm} and intercept T_{calc}. The actual efficiency (when $M = 1$) is 92%. With overlapped communications, we see that until M is around 120 the T_{overlap} term is entirely masked and the slope proportional only to T_{setup}. After the turnover region the slope reverts to T_{comm} as for the non-overlapped case. The efficiency ($M = 1$) has now increased to 99% as expected.

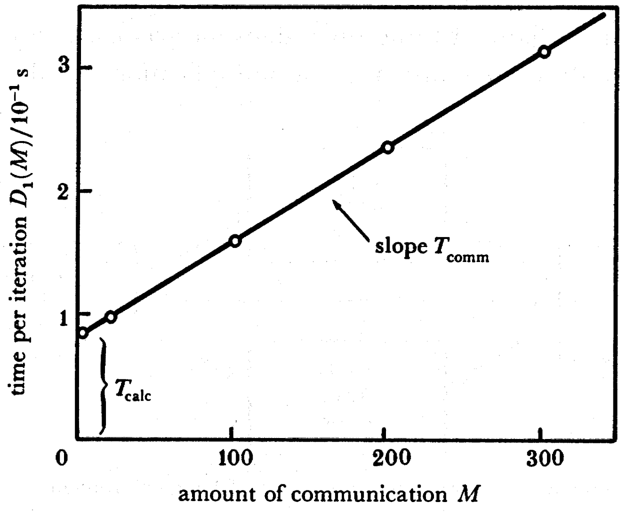

FIGURE 11. Laplace results: non-overlapped communications.

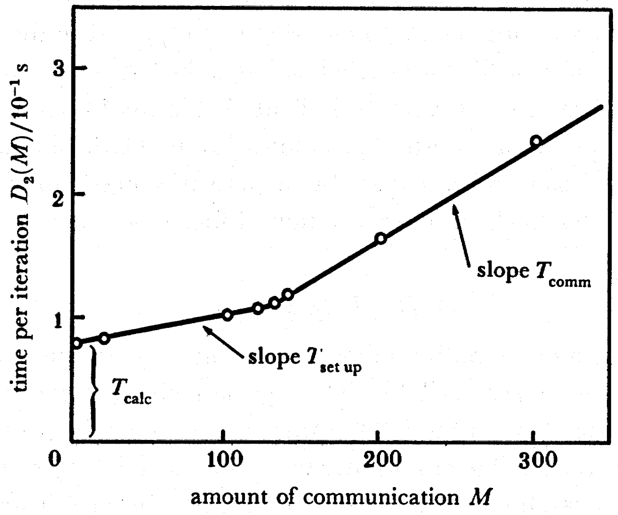

FIGURE 12. Laplace results: overlapped communications.

Although it is gratifying to be able to achieve such high efficiencies and validate the simple analysis outlined above, two words of caution are in order. Firstly, it may well be that for purely pragmatic reasons, such as simplicity of code and so on, it is better not to worry about extracting every last bit of speedup but rather code the problem more simply and throw more transputers at it! Secondly, if the program is written to communicate in all eight directions simultaneously and another process also needs to access external memory, then external memory bandwidth becomes a significant limiting factor.

(*b*) *Algorithmic parallelism*

In this approach to the problem, the program must be split up into roughly equal 'size' pieces. In this case there is a very simple basic algorithm which may be written as

$$\text{update} = \tfrac{1}{4}(\text{left}+\text{right}+\text{up}+\text{down}).$$

Dividing this up as shown in figure 13 into operations on vectors, two transputers perform additions on vectors and a third an addition and a multiplication. All these transputers need

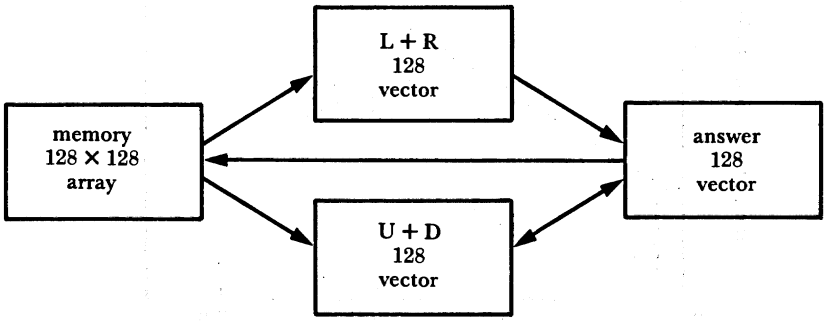

FIGURE 13. Algorithmic decomposition of Laplace problem.

little or no external memory: a fourth transputer with substantial external memory keeps the old and updated versions of the entire array. With four T414 transputers and a 40 × 40 array, an efficiency of about 50% was obtained (Pritchard *et al.* 1987). Given the relatively poor load balance between the transputers this lower efficiency is to be expected. Algorithmic networks for more complex algorithms can be very complicated. Figure 14 shows one such network constructed by Bryan Carpenter at Southampton for a Monte Carlo simulation of a statistical mechanical spin system (Askew *et al.* 1988). Such networks need not only care with load balancing but also with deadlock and termination. Efficiencies between 50 and 60% are typical.

(*c*) *Hybrid parallelism*

For certain applications a combination of geometric and algorithmic parallelism can make optimal use of the processing power available. This hybrid technique has been successfully used by Bryan Carpenter to code 1260 16-bit T212 transputers to solve the three-dimensional Ising ferromagnet (Carpenter 1987). To our knowledge, this is the largest MIMD machine ever programmed and one that can honestly be described as a 10 Gips (10^9 instructions per second) machine! The Ising model simulation can be programmed using several different algorithms so in table 1 we compare the performance of this 'B001260' machine against several other computers by using the same 'Metropolis algorithm'. As can be seen, this transputer array, assembled out of standard components over a few days, is faster than a special purpose machine built at Santa Barbara to solve just this one problem! Moreover, the B001260 achieves almost a third of the performance of a Cyber supercomputer for a small fraction of the cost. In fact, as the last column in table 1 shows, the world best performance for the Ising problem using this

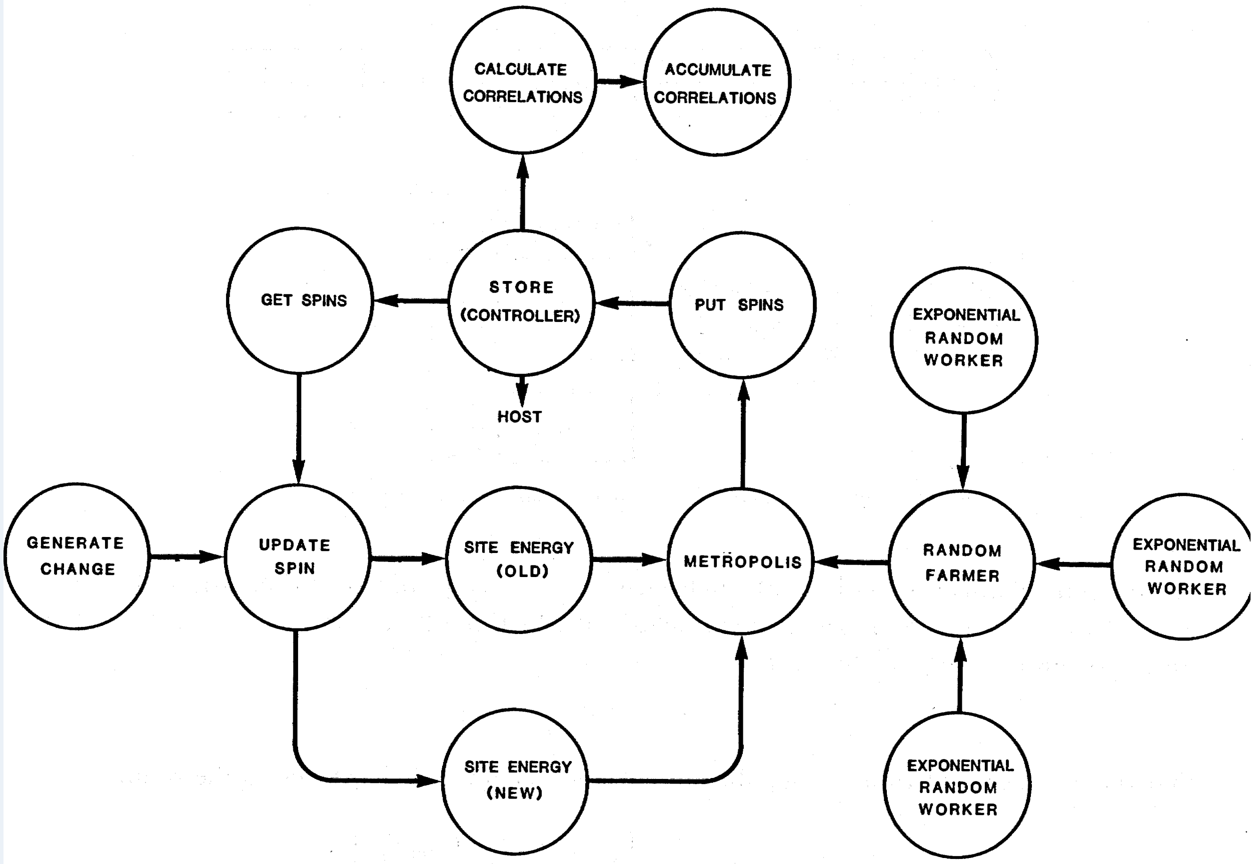

FIGURE 14. Algorithm network for Monte Carlo simulation.

TABLE 1. COMPARATIVE PERFORMANCES FOR THE METROPOLIS ALGORITHM FOR THREE-DIMENSIONAL ISING SYSTEM

	Santa Barbara	B001260	2-pipe CYBER	ICL DAP
update speed/(m s^{-1})	25	27	93	218

algorithm is probably held by the SIMD ICL DAP machine: the binary nature of the Ising model is particularly well suited to the single-bit processing elements of the DAP. For a more floating-point intensive problem, the DAP does not compare so well. Moreover, if the 1260 transputers had more than just the 2 Kbytes of on-chip memory and were more reconfigurable, it is probable that at least an order of magnitude improvement in performance could be achieved (Carpenter 1987).

(d) *Processor farm parallelism*

There are two basic types of processor farm: one in which the same entire code is held in each processor, which can then operate on entirely independent sets of data, and one in which independent pieces of 'work' are sent to each processor by a farm controller. They can use various topologies such as the linear chain or ternary tree shown in figure 15.

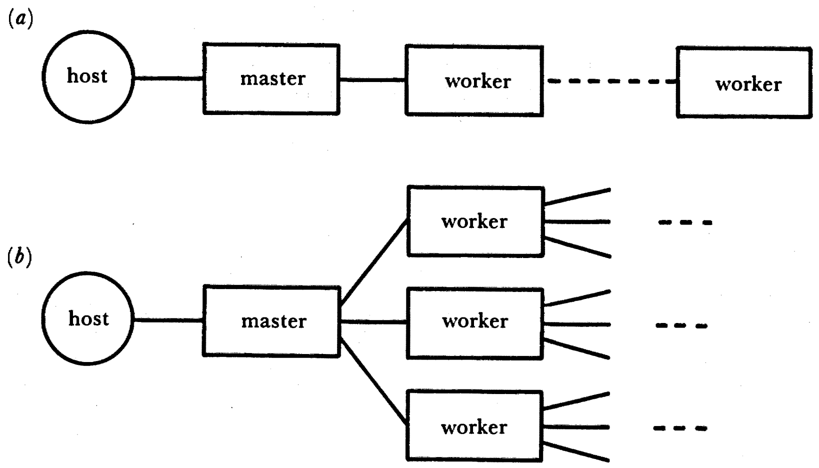

FIGURE 15. Farming networks: (*a*) linear chain; (*b*) ternary tree.

The mathematical analysis of all such farm types is very similar (Pritchard, 1987). For example, if one wants to maximize the rate, S_N, at which results are obtained from the end of an N processor chain, one finds, for

$$T_{\text{calc}} > T_{\text{setup}}$$

that there is a critical value N_c for the largest useful chain. Pritchard quotes the result

$$N < N_c : S_N = \frac{1}{2T_{\text{setup}}}\left[1-\left(\frac{(T_{\text{calc}}-T_{\text{setup}})}{(T_{\text{calc}}+T_{\text{setup}})}\right)^N\right];$$

$$N > N_c : S_N = \frac{1}{T_{\text{comm}}+T_{\text{setup}}}.$$

Processor farms also give the opportunity for porting certain types of existing FORTRAN, C or PASCAL programs with a minimum of effort. For example, we have ported a 3000-line FORTRAN 77 program for Monte Carlo simulation of events generated in electron–positron annihilations to run on a transputer farm. This program has been implemented on a Meiko system consisting of up to 30 transputers running in an OCCAM farming harness (Glendinning & Hey 1987). This application has very limited communication requirements and a linear speedup was observed. Comparative figures for this application on a VAX 750, T414 and T800 transputer are shown in table 2 (Cownie, personal communication).

TABLE 2. COMPARATIVE PERFORMANCES FOR FORTRAN MONTE CARLO PROGRAM

(N is the number of transputers in the farm: up to 30 in these experiments.)

	VAX 750	T414	T800
seconds per event	0.18	0.61/N	0.07/N

5. Reconfigurable transputer networks

Until recently our transputer systems at Southampton have had to be manually connected in the required topology. When presented with a complex wiring diagram instructing the user to 'connect processor 10 link 3 to processor 12 link 0' and so on, the benefits of a software-controlled switch seem obvious! For the more sceptical, the advantages of the use of a reconfigurable network of transputers over implementing applications on some fixed general-purpose network are threefold. Firstly, it allows us to simulate the performance of a wide range of hard-wired systems during design; secondly, direct link connections can be made to optimize the topology for a specific application; and thirdly, there is the possibility of dynamic reconfiguration for greater flexibility and dynamic load balancing. We have not yet investigated dynamic reconfiguration but we have seen definite performance gains with specifically configured networks rather than hard-wired networks for several specific applications. In what follows, reconfigurable transputer networks are therefore assumed to be desirable and the switch design analysis of Lloyd, Nicole and Ward (Lloyd *et al.* 1988) will be described. It may turn out that reconfigurability becomes less of an issue for networks with very large numbers of transputers (although our experience with the B001260 would seem to indicate otherwise) or for networks of future more powerful transputers with a routing engine dedicated to the control of the link traffic.

In designing a switch for transputer networks we wish not only to have a 'universal' static switch – universal in the sense that it will allow any valid transputer graph to be realized – but also to have simple and fast algorithms to translate the desired network topology into the appropriate switch setting. We begin with a few words of notation. Figure 16*a* shows a transputer with 4 links: each link consists of two channels, *a* to *b and b* to *a*. Figure 16*b* depicts a switch capable of connecting any pair of input links; figure 16*c* shows a switch that can connect any link on one side to any link on the other side of the switch.

Consider the example of transputers with only two links. Figure 17 shows a switching network that is *not* universal: only disconnected graphs with even numbers of transputers can be configured (figure 18). In contrast we see that figure 19 shows a universal switch, allowing any number of transputers in the disconnected subgraphs (figure 20).

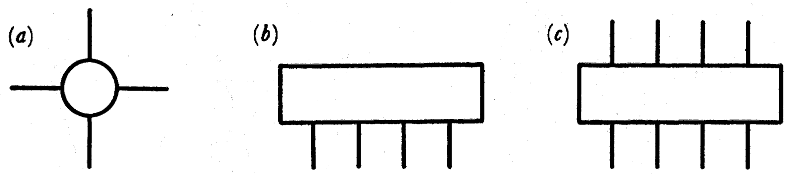

Figure 16. (*a*) 4-link transputer. (*b*) Switch for any pair of inputs. (*c*) Switch for any input to any output link.

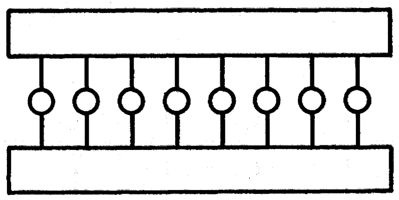

Figure 17. Non-universal switching network for two-link transputers.

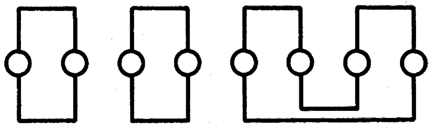
Figure 18. Disconnected graph allowed by switch of figure 17.

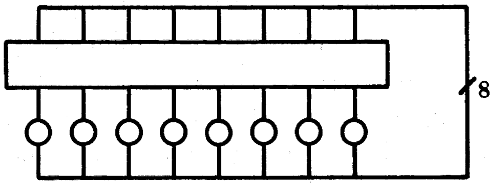

Figure 19. Universal switching network for two-link transputers.

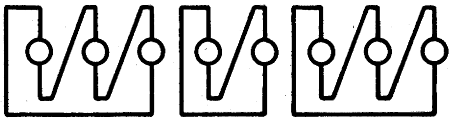
Figure 20. Disconnected graph allowed by switch of figure 19.

In analysing transputer networks there are two types of cycles that are of relevance: hamiltonian cycles and eulerian cycles. A hamiltonian cycle is a closed path that visits each vertex (transputer) exactly once. It is well known that there are many graphs that do not possess hamiltonian cycles. Some examples are shown in figure 21. Moreover, it is a computationally difficult problem to determine whether an arbitrary graph has one or more hamiltonian cycles. It is therefore inadvisable to base a reconfigurable transputer network on a switching network containing a fixed spine of links like that shown in figure 22. Such a switch cannot be universal.

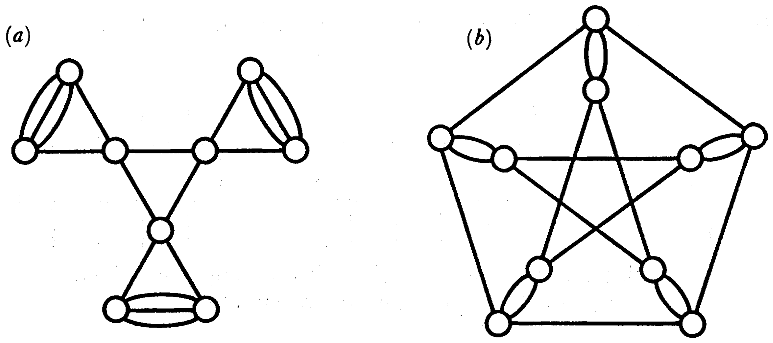

Figure 21. Simple networks with no hamiltonian cycles.

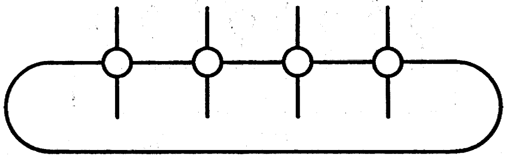
Figure 22. Non-universal switching network.

An eulerian cycle is a closed path that visits each edge (link) exactly once and, for transputers, visits each vertex exactly twice. Such cycles are possessed by all transputer networks. Furthermore, such cycles are easy to construct and can be used to decompose a given transputer graph into two subgraphs, each with the same number of vertices but with only two links per transputer (figure 23). Because we already know how to construct a universal two link switch, we can assign one of these for the E–W links and another for the N–S links to obtain a truly universal static four link switch. This analysis forms the basis for the switch design in the RTP supernode machine being built in ESPRIT project P1085 (Lloyd *et al.* 1988).

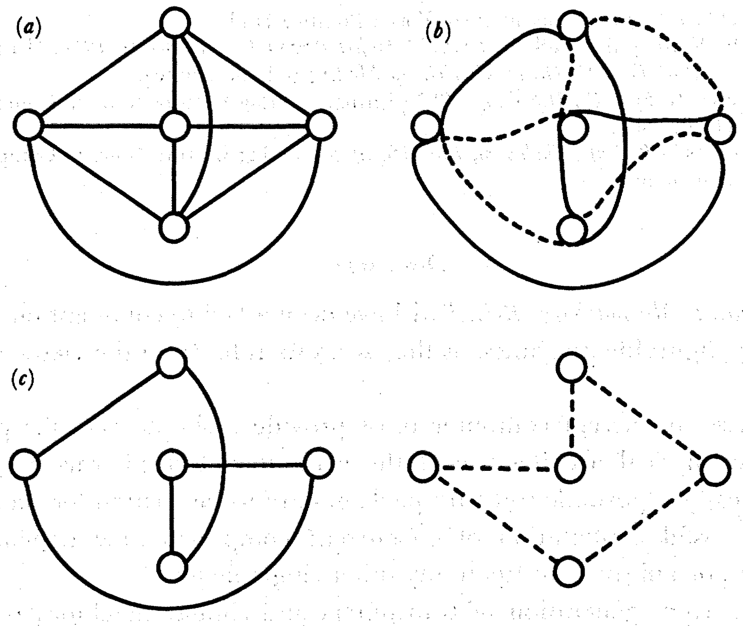

FIGURE 23. Eulerian cycle analysis of a transputer network. (*a*) The network; (*b*) eulerian cycle; (*c*) two-link subgraph decomposition.

6. CONCLUSIONS

It is clear that transputer networks with OCCAM as a system programming language can constitute the basis of very versatile and powerful MIMD parallel computers. However, it is also equally clear that most applications programmers desire better software tools and the provision of high-level languages and 'smart' compilers. These, I believe, will come although it is obviously going to be some years before parallel hardware can truly claim to be able to provide general-purpose, high-performance computing accessible to all. Nevertheless, even with the present level of software development, I hope that I have demonstrated that reconfigurable transputer networks can already provide powerful and cost-effective machines for concurrent computation that can be programmed safely and efficiently.

I am grateful for the help and advice of the following members of the Concurrent Computation Group at Southampton: James Allwright, Charles Askew, Bryan Carpenter, Paul Coddington, Ian Glendinning, Andrew Hislop, Andrew Jackson, John Merlin, Denis Nicole and David Pritchard. I am also grateful to Inmos and ESPRIT for providing us with the wherewithal to perform this research.

References

Askew, C. *et al.* 1988 Monte Carlo simulation on transputer arrays. Parallel Computing. (In the press.)
Carpenter, D. B. 1987 Southampton concurrent computation group Int. Rep.
Dathi, N. 1986 Technical Monograph, Programming Research Group, Oxford.
Feynman, R. P. 1985 *Surely you're joking, Mr Feynman!* New York: W. W. Norton and Company.
Fox, G. C. & Messina, P. C. 1987 Advanced computer architectures. *Scient. Am.* **257**, 45.
Fox, G. C. & Otto, S. W. 1985 In *Proc. Knoxville Hypercube Conference, Aug. 1985* (ed. M. Heath). Siam.
Glendinning, I. & Hey, A. J. G. 1987 *Computer Phys. Commun.* **45**, 367.
Hoare, C. A. R. 1985 Communicating sequential processes. New York: Prentice-Hall.
Homewood, M., May, D., Shepherd, D. & Shepherd, R. 1987 IMS T800 Transputer. *IEEE Micro*; October 1987, pp. 10–26.
Inmos Ltd 1988 *The Occam 2 reference manual.* New York: Prentice Hall.
Lloyd, K., Nicole, D. & Ward, J. S. 1988 In *ESPRIT project P1085 Internal Report, 1985.* (In preparation.)
Pritchard, D. J. 1987 In *Proc. Grenoble Occam User Group Meeting* (ed. T. Muntean).
Pritchard, D. J. *et al.* 1987 In *Proc. PARLE Conf. 1987* (Springer Verlag Lecture Notes in Computer Science **258**) (ed. G. Goos & J. Hartmanis).
Welch, P. 1987 In *Proc. PARLE Conf., Eindhoven, 1987* (Springer Verlag Lecture Notes in Computer Science **258**) (ed. G. Goos & J. Hartmanis).

Discussion

D. May (*Inmos Limited, Almondsbury, Bristol*). I have been asked to comment on Professor Hey's remarks about reconfigurable machines, as they seem to differ from the views expressed in my own paper.

A general-purpose concurrent computer must provide a simple way for programs to be mapped on to the physical architecture of the computer. Indeed, this mapping must be achieved automatically if portable software packages are to be written for such machines. In my view, the successful exploitation of concurrent computers now depends more upon achieving software portability than upon any other single factor.

Obviously, the current generation of transputers and similar machines requires that, in many cases, the algorithm is designed to suit a particular configuration. This is satisfactory for embedded applications, where the configuration can be determined by the application. It is obviously unsatisfactory for a general-purpose computer. Consequently, some concurrent computers include switches allowing the configuration to be programmed to suit a particular application.

Reconfigurable machines vary in the configurations they can implement. Indeed, some reconfigurable machines do not permit all possible configurations to be achieved. This is not particularly surprising, as for machines with many processors the interconnect needed would be very large. Consequently, these machines do not overcome the problem of software portability.

In the very near future, it will be possible to provide hardware support for message routing in each of the processing nodes. This will remove the need for reconfiguration in all but very-high-performance applications. It will also allow the construction of very large machines without excessive amounts of interconnection. Finally, and most importantly, it will allow *any* software configuration to be implemented directly on *any* machine with adequate hardware resources.