This is Chapter 1 from the second edition of :

Networks, Routers and Transputers:

Function, Performance and applications

Edited by M.D. May, P.W. Thompson, and P.H. Welch

IOS Press
Van Diemenstraat 94
1013 CN Amsterdam
Netherlands

IOS Press, Inc.
P.O. Box 10558
Burke, VA 22009–0558
U.S.A.

IOS Press/Lavis Marketing
73 Lime Walk
Headington
Oxford OX3 7AD
England

Kaigai Publications, Ltd.
21 Kanda Tsukasa–Cho 2–Chome
Chiyoda–Ka
Tokyo 101
Japan

This chapter was written by M.D. May and P.W Thompson.

1 Transputers and Routers: Components for Concurrent Machines

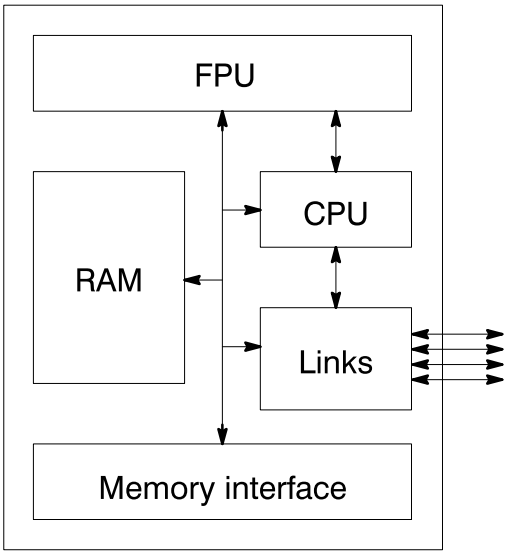
1.1 Introduction

This chapter describes an architecture for concurrent machines constructed from two types of component: 'transputers' and 'routers'. In subsequent chapters we consider the details of these two components, and show the architecture can be adapted to include other types of component.

A transputer is a complete microcomputer integrated in a single VLSI chip. Each transputer has a number of communication links, allowing transputers to be interconnected to form concurrent processing systems. The transputer instruction set contains instructions to send and receive messages through these links, minimizing delays in inter-transputer communication. Transputers can be directly connected to form specialised networks, or can be interconnected via routing chips. Routing chips are VLSI building blocks for interconnection networks: they can support system-wide message routing at high throughput and low delay.

1.2 Transputers

VLSI technology enables a complete computer to be constructed on a single silicon chip. The INMOS T800 transputer [1], integrates a central processor, a floating point unit, four kilobytes of static random access memory plus an interface for external memory, and a communications system onto a chip about 1 square centimetre in area.



T800 Transputer

As a microcomputer, the transputer is unusual in that it has the ability to communicate with other transputers via its communication links; this enables transputers to be connected together to construct multiprocessor systems to tackle specific problems. The transputer is also unusual in that it has the ability to execute many software processes, sharing its time between them automati-

cally, to create new processes rapidly, and to perform communication between processes within a transputer and between processes in different transputers. All of these capabilities are integrated into the hardware of the transputer, and are very efficient. This is discussed in more detail in chapter 2.

The use of transputers for parallel programming has been greatly simplified by the development of the occam programming language [2]. The occam language allows an application to be expressed as a collection of concurrent processes which communicate via channels. Each channel is a point-to-point connection between two processes; one process always inputs from the channel and the other always outputs to it. Communication is synchronised; the first process ready to communicate waits until the second is also ready, then the data is copied from the outputting processes to the inputting process and both processes continue.

Each transputer has a process scheduler which allows it to share its time between a number of processes. Communication between processes on the same transputer is performed using the local memory; communication between processes on different transputers is performed using a link between the two transputers. Consequently, a program can be executed either by a single transputer or by a collection of transputers connected in a network. Three different ways of using transputers to execute the component processes of a typical program are shown below.

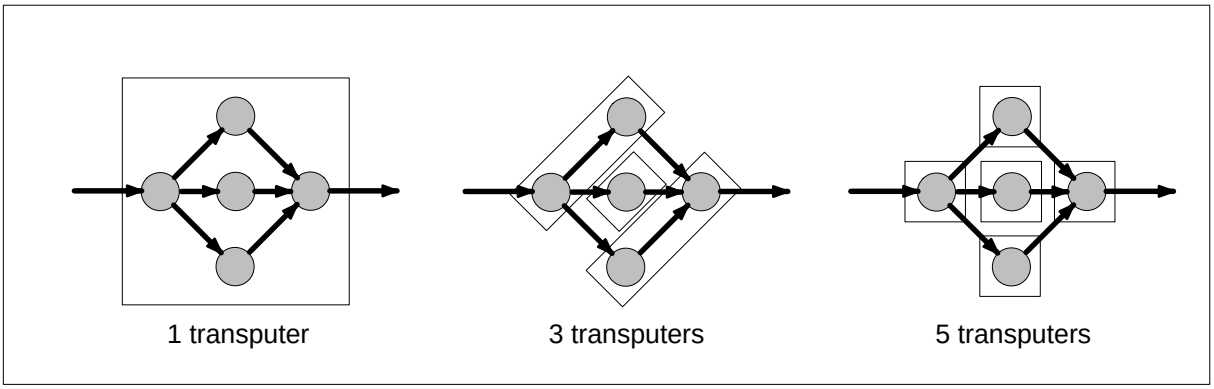


Figure 1.1 Allocations of processes to processors

Figure 1.1 shows the same collection of processes executed on three different specialised networks. In the first network, which is a single transputer, each communication channel connecting two processes is implemented using the local memory of the transputer. In the other examples some or all of the channels are implemented by physical links between different transputers.

Transputers have also been used to construct a number of general purpose computers, which all consist of an array of transputers connected together in a network. In some machines the network can be configured by software, for example by connecting the links via a programmable crossbar switch. Many applications have been successfully ported to these machines and have demonstrated efficient parallel processing.

One of the problems with existing general purpose transputer machines is the need to carefully match algorithms to the interconnection networks of specific machines, which results in a lack of software portability. It has become clear that a standard architecture is needed for these general purpose message-passing machines. An attractive candidate is a collection of transputers connected by a high throughput, low delay communication network supporting communication channels between processes anywhere in the network.

1.3 Routers

There are many parallel algorithms in which the number of communication channels between processes on different transputers is much greater than the number of physical links available to

connect the transputers. In some of these algorithms, a process executed on one transputer must communicate with processes on a large number of other transputers. These requirements for system-wide communication between processes can be met by:

- new transputers including hardware to multiplex many 'virtual links' along a single physical link (see chapter 2)
- new VLSI message-routing chips (routers) which can be used to construct efficient communication networks

This new communications architecture allows communication channels to be established between any two processes, regardless of where they are physically located in the system. This simplifies programming because processes can be allocated to transputers to optimize performance after the program has been written. For general purpose message-passing computers, a further benefit is that processes can be allocated to transputers by a compiler, which effectively removes configuration details from the program, thereby enhancing portability.

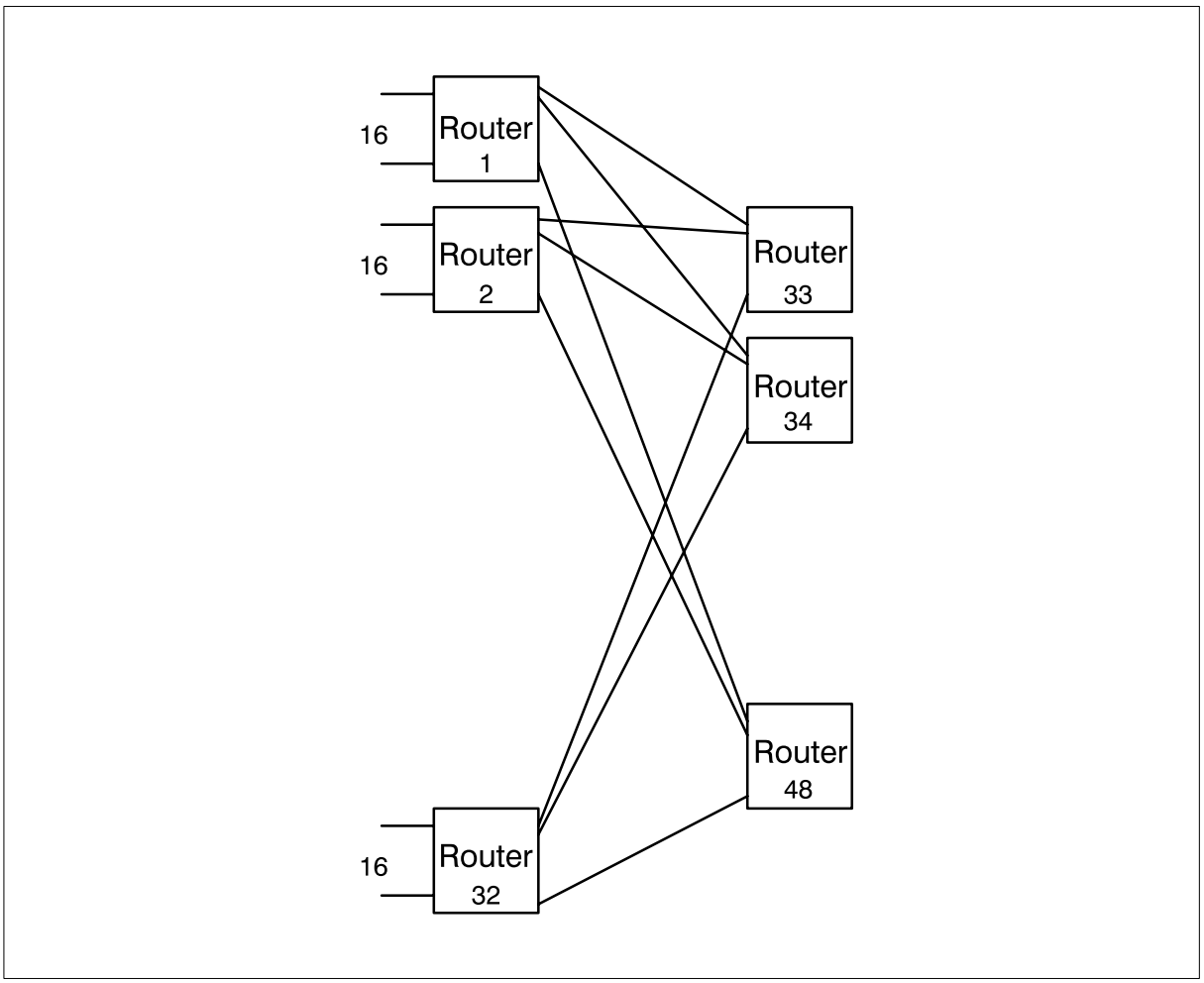


Figure 1.2 Network constructed from routers

The use of two separate chips, one to perform computing (the transputer) and one to perform communication (the router) has several practical advantages:

- Transputers can be directly connected without routers in systems which do not require message routing, so avoiding the silicon cost and routing delays.
- It allows routers to have many links (e.g.32) which in turn allows large networks to be constructed from a small number of routers, minimizing the delay through the network. For example, 48 such routers can connect 512 terminals with only 3 routing delays, as in figure 1.2.

- It avoids the need for messages to flow through the transputer, reducing the total throughput of the chip interface. This reduces the pin count, power consumption and package costs of the transputer.
- It supports scalable architectures in which communication throughput must be balanced with processing throughput. In such architectures, it is known that overall communication capacity must grow faster than the total number of processors - a larger machine must have proportionately more routers.

Since the new architecture allows all the virtual links of a transputer to pass through a single physical link, system-wide communication can be provided by connecting each transputer to a routing network via a single link. The provision of several links on transputers allows each transputer to be connected to several different networks. Examples of the use of this technique are:

- The use of two (or more) identical networks in parallel to increase throughput and fault–tolerance [7]
- The use of a main network and an (independent) monitoring and debugging network
- The use of a main network and an independent network for input and output (or for access to discs)

Another technique for increasing the communications throughput is to construct the network using two (or more) links in parallel for each connection. An example of a 2-dimensional network of this kind is shown in figure 1.4.

In some cases, it is convenient to construct a network from routers and attach transputers to its terminal links. An example is the multi-stage network shown in figure 1.2. An alternative is to construct a network such as a hypercube or an array from a number of nodes, each node consisting of one or more transputers and a router as shown in figure 1.4.

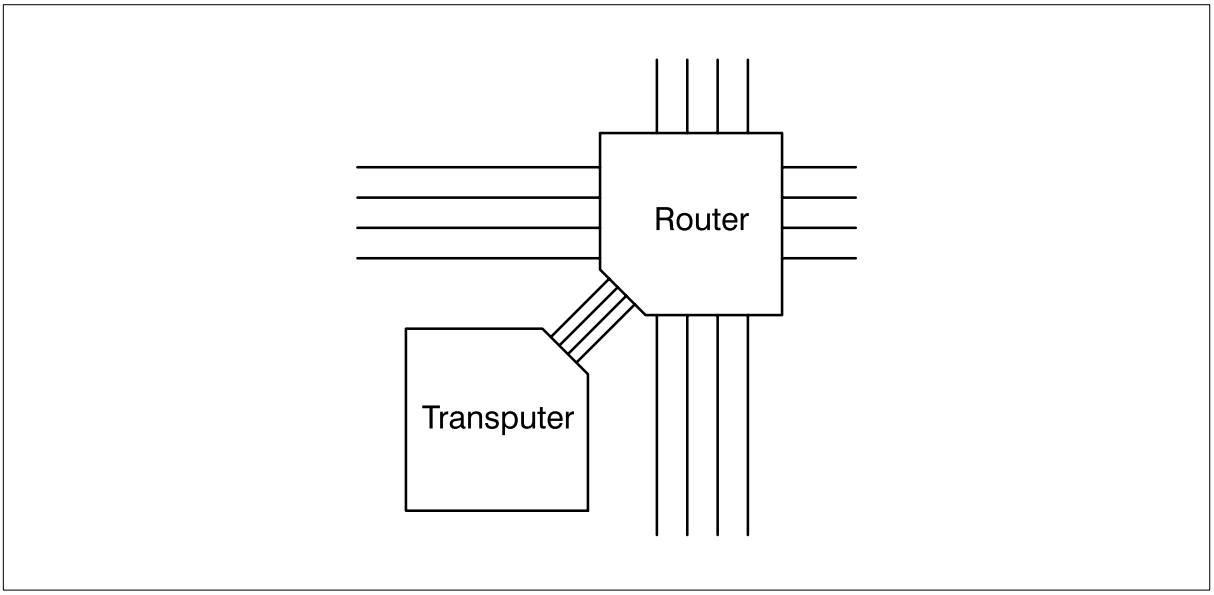


Figure 1.3 Node combining a transputer and a router

Operation of Routers

Each router has a number of communication links and operates as follows:

- It uses the header of each packet arriving on each link to determine the link to be used to output the packet;
- It arbitrates between two (or more) packets which must both be output through the same link, and causes them to be output one after another;

- It starts to output each packet as early as possible (immediately after the output link is determined, provided that the output link is not already in use for another packet).

The overall throughput of the router is determined by the number of links which can be operating concurrently. An important benefit of employing serial links for packet routing is that it is simple to implement a full crossbar switch in VLSI, even for a large number of links. Use of a full crossbar allows packets to be passing through all of the links at the same time.

The ability to start outputting a packet whilst it is still being input can significantly reduce delay, especially in networks which are lightly loaded. This technique is known as *wormhole routing*. In wormhole routing, the delay through the switch can be minimized by keeping headers short and by using fast, simple, hardware to determine the link to be used for output.

The use of simple routing hardware allows this capability to be provided for every link in the router. This avoids the need to share it between many links which would increase delay in the event of several packets arriving at once. Equally, it is desirable to avoid the need for the large number of packet buffers commonly provided in some packet routing systems (in which each packet is input to a buffer before output starts). The use of small buffers together with simple routing hardware allows a single VLSI chip to provide efficient routing between a large number of links.

The simple communications architecture allows a wide variety of implementations:

- CMOS VLSI can be used to construct routers with a large number of links;
- It is straightforward to combine transputers and small routers on a single chip;
- It is possible to construct routers in ECL or Gallium Arsenide technology to support extremely high speed implementations of the link.

For some purposes, it may be useful to combine a router together with each transputer in a single chip (or a single package). One example is the construction of a two dimensional array of simple transputers for image processing (for this application, no off-chip memory is needed, and most communication is local). The architecture of the routing system makes such a combination possible, as in figure 1.4.

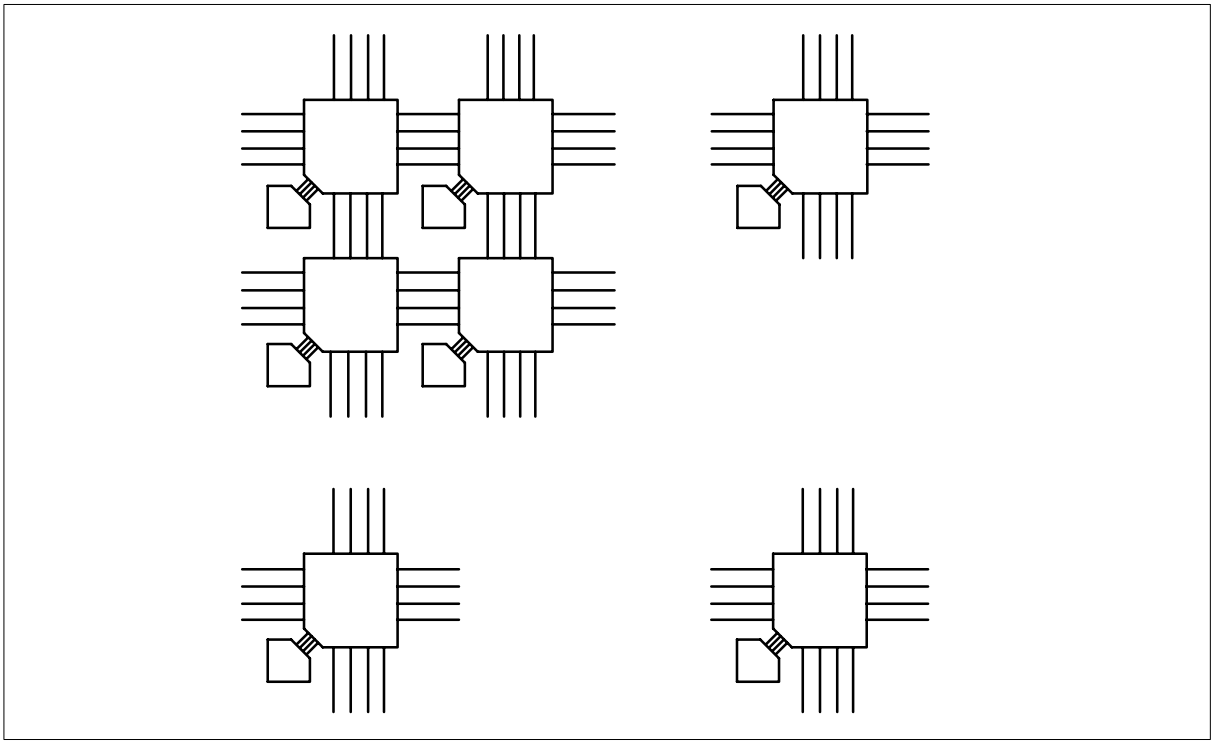


Figure 1.4 Two dimensional array of nodes

1.4 Message Routing

1.4.1 Avoiding Deadlock

The purpose of a communications network is to support efficient and reliable communication between processes. Consequently, an essential property of a communications network is that it should not deadlock, i.e. arrive in a state where further progress is impossible. However, deadlock can occur in most networks unless the routing algorithm is designed to prevent it. For example, consider the square of four nodes shown in figure 1.5. Suppose that every node attempts to send a packet to the opposite corner at the same time, and that the routing algorithm routes packets in a clockwise direction. Then each link will become 'busy' sending a packet to the adjacent corner and the network will deadlock.

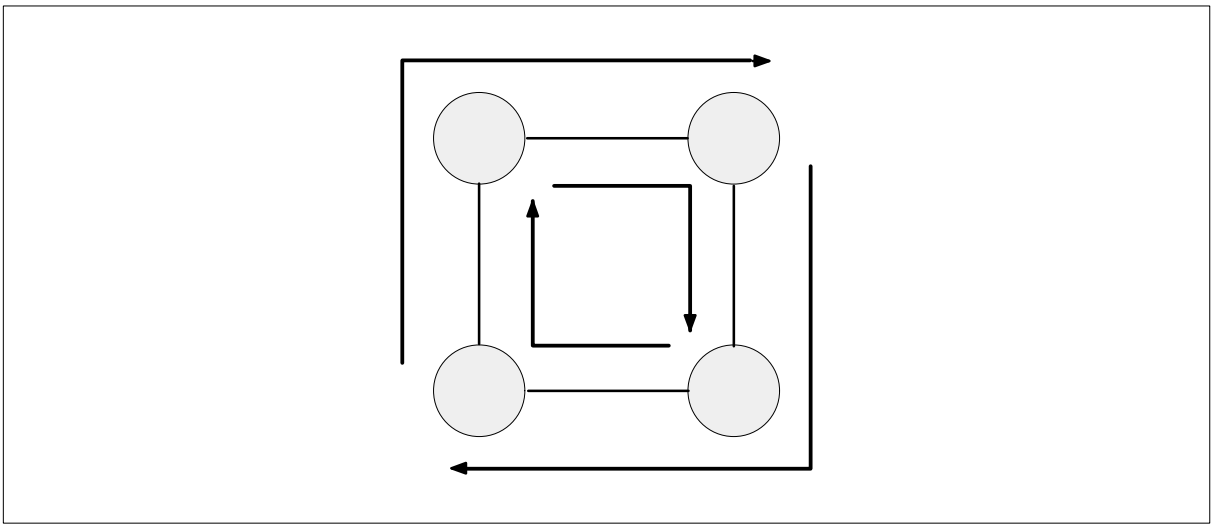


Figure 1.5 Deadlock in a simple network

It is important to understand that deadlock is a property of the network topology and the routing algorithm used; it can also arise with buffered packet routing. In the above example, a single packet buffer at each node is sufficient to remove the deadlock but, in general, the number of packet buffers needed to eliminate deadlock depends on the network topology, the routing algorithm and the applications program. This is clearly not a satisfactory basis for a general purpose routing system.

All of the above problems can be avoided by choosing networks for which deadlock-free wormhole routing algorithms exist. In such networks, buffers are employed only to smooth the flow of data through the network and to reduce congestion; often a buffer of size much less than the length of a packet is sufficient for this purpose. Most important of all, the buffering needed is not dependent on the network size or the applications program. It is possible to construct a single universal router which can be used for networks of arbitrary size and for programs of arbitrary complexity. An essential property of such a router is that, like a transputer, it can communicate on all of its links concurrently.

It turns out that many regular networks constructed from such routers have deadlock free routing algorithms. Important examples are trees, hypercubes and grids.

A deadlock free routing algorithm for Trees

A tree consists of a collection of nodes with a single external link from the root. Assume that

two trees[1] T_1 with root link r_1 and T_2 with root link r_2 are both deadlock free; they will always perform internal communication without deadlock, and will accept and transmit packets along the root link without deadlock.

A new tree is formed by connecting the root links r_1 and r_2 to a new root node R; a further link r on this node is the root link of the newly constructed tree T.

Any packet arriving at R along r_1 is routed either to r_2 or to r. If it is routed to r_2, it will be consumed by T_2, because T_2 is deadlock free. If it is routed to r, it will eventually be consumed by the environment. By symmetry, packets arriving along r_1 will also be consumed. A packet arriving along r will be routed to either T_1 or T_2; in either case it will be consumed because both T_1 and T_2 are deadlock free.

It remains to show that a tree with only one node is deadlock free; this is true because the node can send and receive packets concurrently along its single (root) link.

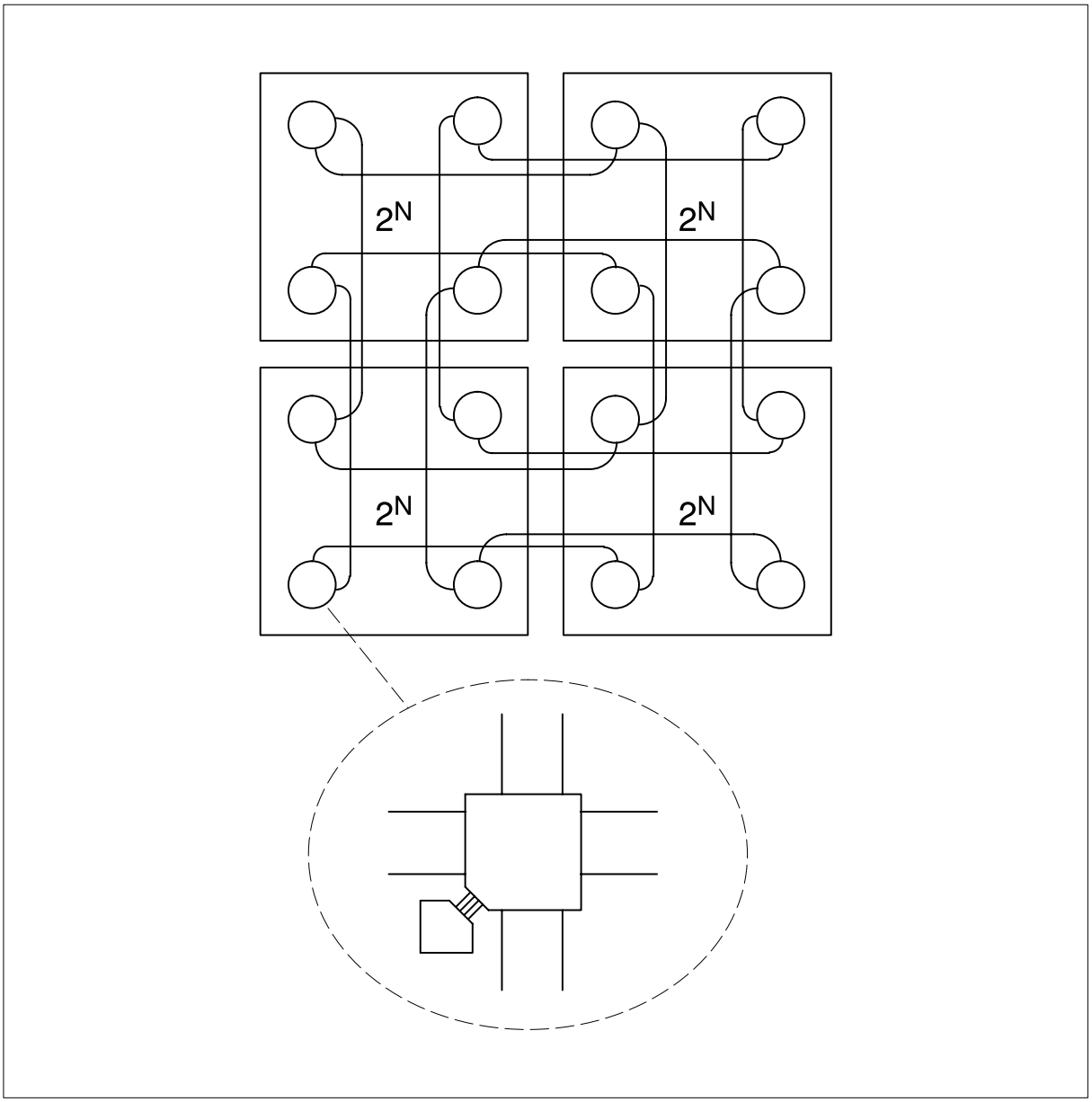


Figure 1.6 Hypercube constructed from 2^{N+2} Nodes

1. Note that this construction can easily be generalized from binary to n-ary trees.

A deadlock free routing algorithm for Hypercubes

To avoid deadlock in a hypercube, each packet is successively routed through the dimensions, starting from the highest.

A simple inductive argument can be used to show that this routing algorithm is free of deadlocks. Suppose that the order-N hypercube is deadlock free. Combine two such order-N hypercubes H_1 and H_2 to form an order-(N+1) hypercube by linking corresponding nodes of H_1 and H_2. Any packet originating at a node n in H_1 and destined for a node in H_2 will first travel along the link joining n to the corresponding node in H_2; from this node it will be delivered by routing within H_2 and this is deadlock free by assumption. Similarly, any packet originating at a node n in H_2 and destined for a node in H_1 will first travel along the link joining n to the corresponding node in H_1; from this node it will be delivered by routing within H_1 and this is deadlock free by assumption. An important property of the node is that it is able to send and receive along a link at the same time; this is needed to ensure that a packet can flow from node h_1 in H_1 to the corresponding node h_2 in H_2 at the same time as a packet flows into h_1 from h_2.

It remains to show that the order-0 hypercube is deadlock free (which it is, being just a single node)!

The effect of the routing algorithm can easily be understood in terms of the example shown in figure 1.5 above, which shows a 2–cube. Instead of routing all packets in a clockwise direction, the deadlock-free algorithm routes two of the packets anti-clockwise. Since the links are bi–directional this allows all of the packets to be routed without deadlock, as illustrated in figure 1.7.

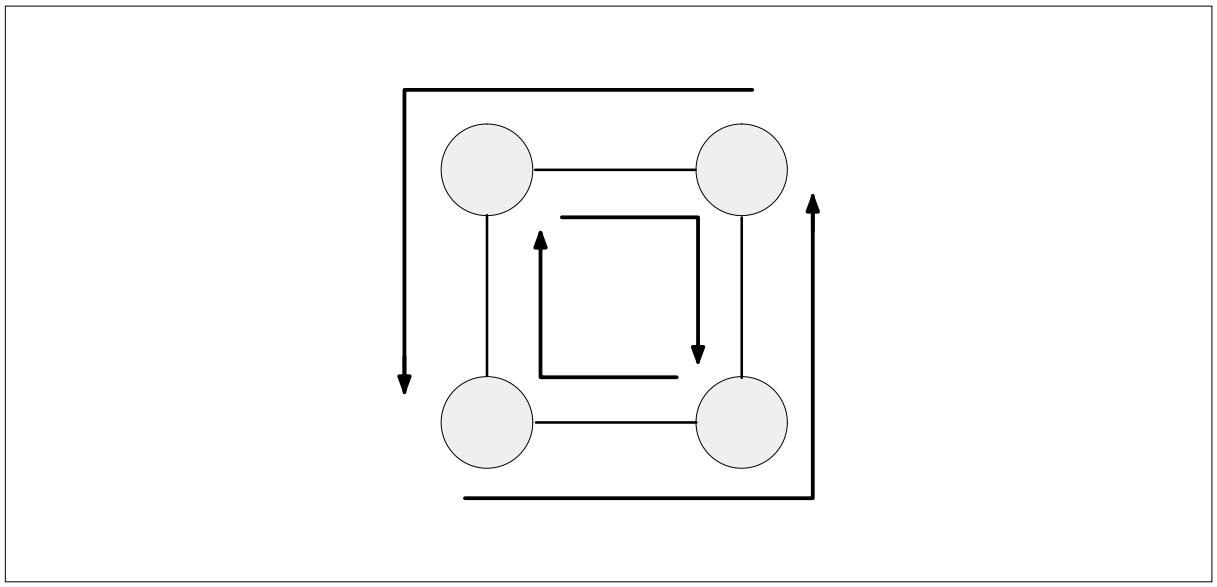


Figure 1.7 Avoiding deadlock in a simple network

The fact that the hypercube is symmetrical means that the order of sequencing through the dimensions does not matter; it is important only that every packet is sequenced in the same order.

A deadlock free routing algorithm for Arrays

The technique of routing a packet by systematically sequencing through the dimensions can be applied to any processor array. In fact, any rectangular processor array - whatever its size and dimension - is deadlock free! To prove this it is first necessary to establish that a line of processing nodes (a one-dimensional array) is deadlock free; this is guaranteed if a packet generated at a node takes the shortest path to its destination node.

A simple inductive argument similar to that used for the hypercube can now be used to establish that this routing algorithm is deadlock free.

1.5 Addressing

Every packet must carry with it the address of its destination; this might be the address of a transputer, or the address of one of a number of virtual channels forming input channels to a transputer. As a packet arrives at a router, the destination address must be inspected before the outgoing link can be determined; the delay through the router is therefore proportional to the address length. Further, the address must itself be transmitted through the network and therefore consumes network bandwidth.

It is therefore important that this address be as short as possible, both to optimize network latency and network bandwidth. However, it is also important that the destination link can be derived from the address quickly and with minimal hardware. An addressing system which meets both of these requirements is *interval labelling*.

1.5.1 Interval Labelling

An *interval labelling scheme* [6] assigns a distinct label to each transputer in a network. For simplicity, the labels for an N transputer network can be numbers in the range [0,1, . . . ,N–1]. At each router in the network, each output link has one or more associated *intervals*, where an interval is a set of consecutive labels. The intervals associated with the links on a router are non-overlapping and every label will occur in exactly one interval.

As a packet arrives at a router, the address is examined to determine which interval contains a matching label; the packet is then forwarded along the associated output link.

The interval labelling scheme requires minimal hardware; at most a pair of comparators for each of the outgoing links. It is also very fast, since the output link can be determined, once the address has been input, after only a single comparison delay provided all the comparisons are done concurrently.

There remains the question of how to assign labels to an arbitrary network. The following examples give labelings for networks constructed from nodes as shown in figure 1.3. Intervals are represented with the notation [a,b), which means the set of labels greater than or equal to a and less than b; note however that the comparisons are performed modulo the total number of labels, and intervals are permitted to ‘wrap around’ through zero.

Trees can be labelled

The transputers in a binary tree[2] with N nodes are labelled as follows. Suppose there are L nodes to the left of the root node. Then the transputers to the left of the root are numbered 0, . . . , L–1; the transputer of the root node is labelled L; the transputers to the right are labelled L+1,. . . ,$N-1$,

Any node n in the tree is itself the root node of a subtree S with nodes s_l, . . . , s_h. The interval associated with the left link of n is [s_l, . . . , n); that associated with the right link is [n+1, . . .,s_h+1); that associated with the root link is [s_h+1, . . . ,s_l). The interval [s_h+1, . . . ,s_l) consists of all of the labels in the tree apart from those in S; numerically it consists of the two intervals [s_h+1, . . . ,N+1) and [0, . . . ,s_l). An example is shown in figure 1.8. This shows the labels assigned to each node, and the intervals assigned to the links of two of the nodes.

2. This construction can easily be generalized from binary to general trees, as illustrated in figure 1.8.

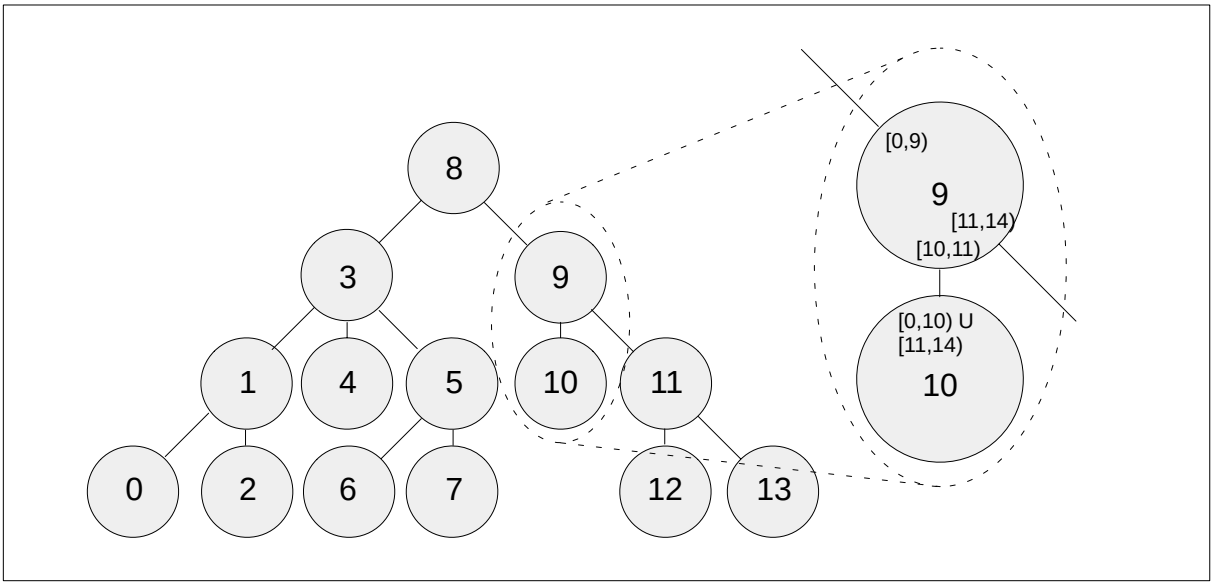


Figure 1.8 A Tree with Interval Labelling

Hypercubes can be labelled

The labelling of the hypercube follows the construction given for the deadlock free routing algorithm. In combining the two order-n hypercubes H_1 and H_2, the transputers in H_1 are labelled $0, \ldots, 2^n - 1$ and those in H_2 are labelled $2^n, \ldots, 2^{n+1} - 1$. The link from each node h_1 in H_1 to the corresponding node h_2 in H_2 is labelled with the interval $[2^n, \ldots, 2^{n+1})$ at h_1, and with $[0, \ldots, 2^n)$ at h_2. This inductively constructs a hypercube together with the deadlock-free routing algorithm described above.

Arrays can be labelled

The labelling for an array follows the construction of the deadlock free routing algorithm. An n-dimensional array is composed of m arrays of dimension $n-1$, with m corresponding nodes (one from each $n-1$ dimensional array) joined to form a line. If each of the $n-1$ dimensional arrays has p nodes, the nodes in the $n-1$ dimensional arrays are numbered $0, \ldots, p-1; p, \ldots, 2p-1; \ldots; (m-1)p, \ldots, mp-1$. On every line the link joining the i^{th} node to the $(i+1)^{th}$ node is labelled $[ip, \ldots, mp)$ and the link to the $(i-1)^{th}$ node is labelled $[0, \ldots, (i-1)p)$. This inductively labels an array to route packets according to the deadlock free algorithm described above. An example is shown in figure 1.9. This shows the labels assigned to each node, and the intervals assigned to the links of one of the nodes.

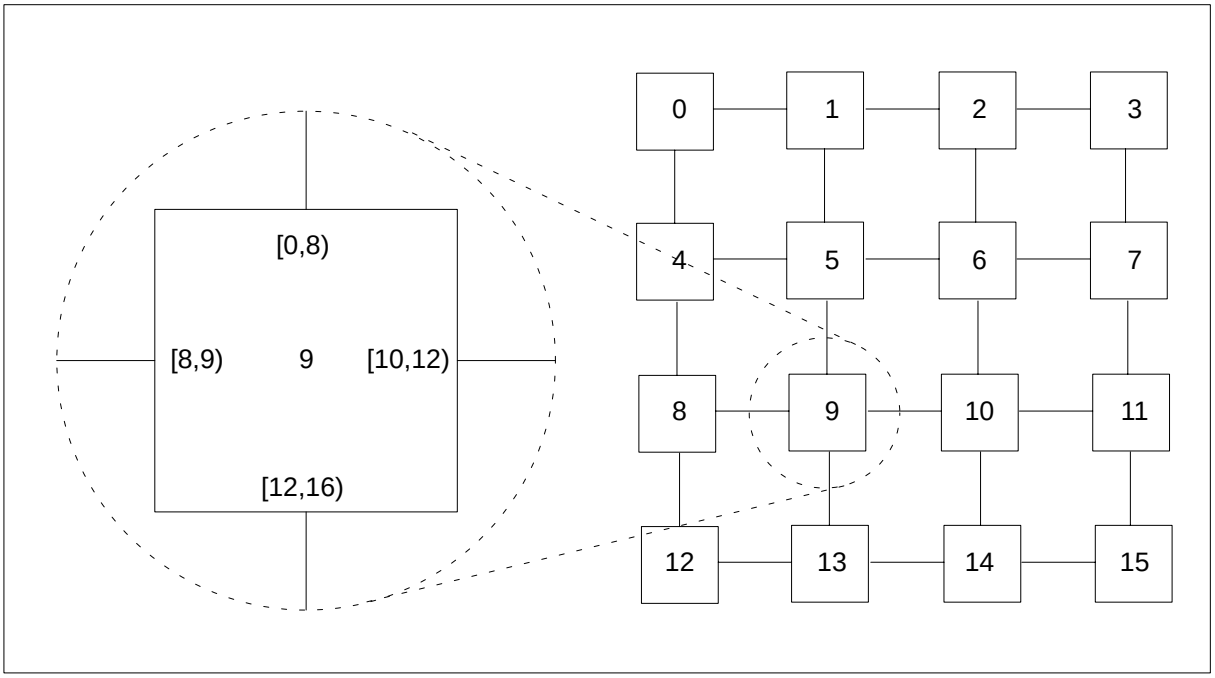


Figure 1.9 An Array with Interval Labelling

Labelling arbitrary networks

The above labelings provide optimal routing, so that each packet takes one of the shortest paths to its destination. It can easily be shown [6] that any network can be labelled so as to provide deadlock free routing; it is only necessary to construct a spanning tree and label it as described above. This may produce a non-optimal routing which cannot exploit all of the links present in the network as a whole. Optimal labelings are known for all of the networks shown below:

trees
hypercubes
arrays
multi-stage networks
butterfly networks
rings[3]

In high performance embedded applications (or in reconfigurable computers) specialised networks are often used to minimize interconnect costs or to avoid the need for message routing. In these systems, a non-optimal labelling can be used to provide low-speed system-wide communications such as would be needed for system configuration and monitoring.

1.5.2 Header Deletion

The main disadvantages of the interval labelling system are that it does not permit arbitrary routes through a network, and it does not allow a message to be routed through a series of networks. These problems can be overcome by a simple extension: *header deletion*. Any link of a router can be set to delete the header of every packet which passes out through it; the result is that the data immediately following becomes the new header as the packet enters the next node.

Header deletion can be used to minimize delays in the routing network. To do this, an initial header is used to route the packet to a destination transputer; this header is deleted as it leaves the final router and enters the transputer. A second header is then used to identify the virtual link within

3. Note that the optimal labelling of a ring requires that one of the connections be duplicated in order to avoid deadlock.

the destination transputer. As the number of transputers is normally much less than the number of virtual links, the initial header can be short, minimizing the delay through each router.

Another important use of header deletion is in the construction of hierarchical networks. In the 2-dimensional array of figure 1.4, each transputer could be replaced with a local network of transputers as shown in figure 1.10. Headers are deleted as packets leave or enter a local network. A single header can be used to route a packet within a local network, whilst three headers are needed to route a packet via the 2-dimensional array.

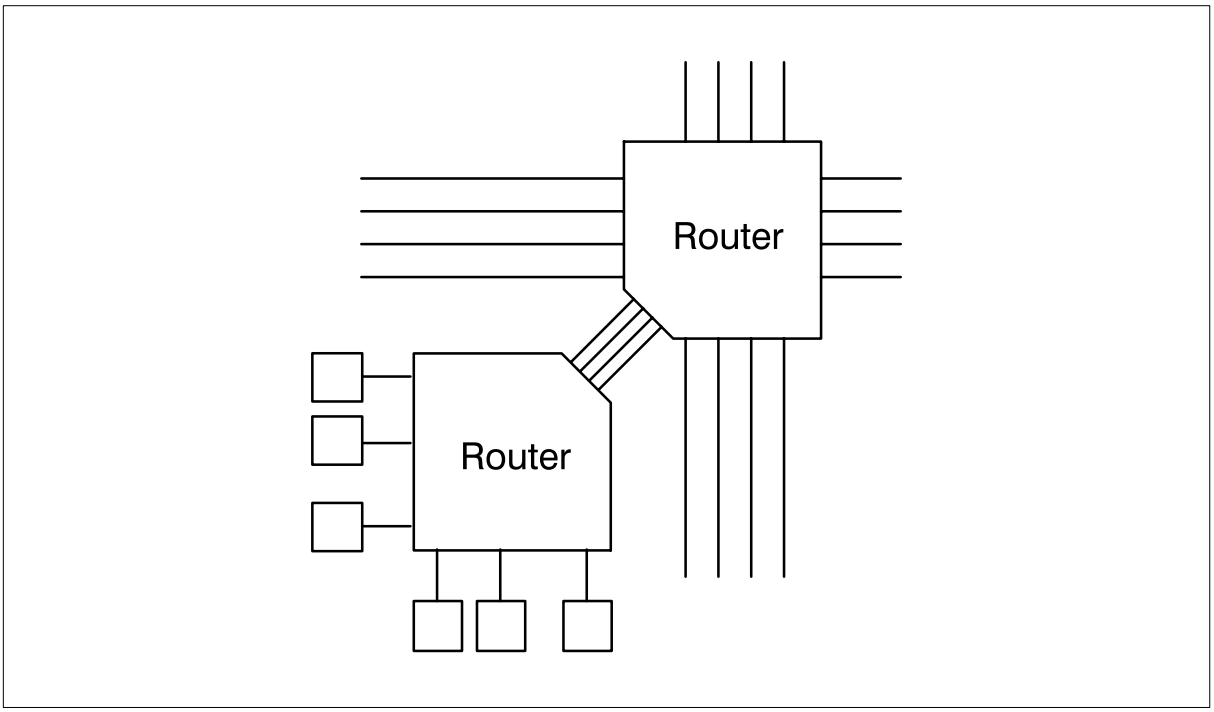


Figure 1.10 Local network of transputers and a router

1.6 Universal Routing

The routing algorithms described so far provide efficient deadlock free communications and allow a wide range of networks to be constructed from a standard router. Packets are delivered at high speed and with low latency provided that there are no collisions between packets travelling through the same link.

Unfortunately, for general purpose concurrent computers, this may not be enough. In any sparse communication network, some communication patterns cannot be realized without collisions. Such collisions within the network can reduce system performance drastically. For example, some parallel algorithms require that all messages from one phase of a computation are delivered before the next phase starts; the late arrival of a single message delays *all* of the processors. In the absence of any bound on message latency it is difficult - and in many cases impossible - to design efficient concurrent programs. The problem of constructing general purpose concurrent computers therefore depends on the answer to the following question:

Is it possible to design a *universal* routing system: a realizable network and a routing algorithm which can implement all communication patterns with bounded message latency?

In fact, a universal routing system allowing the construction of scalable general purpose parallel computers was discovered by Valiant in 1980 [3]. This meets two important requirements:

- The throughput of the network increases proportionately with the number of nodes.

- The delay through the network increases only slowly with the number of nodes (proportional to $log(p)$ for p nodes).

Notice that the aim is to maximize capacity and minimize delay under heavy load conditions - a *parallel* communications network is a vital component of a parallel computer. This is not the same as, for example, minimizing delay through an otherwise empty network.

A p-node hypercube has a delay of proportional to $log(p)$ (written $O(log(p))$) if there are no collisions between packets. This is an unreasonable assumption, however, as all of the transputers will be communicating via the network simultaneously. An important case of communication is that of performing a *permutation* in which every transputer simultaneously transmits a message and no two messages head for the same destination. Valiant's proof [4] demonstrates constructively that permutation routing is possible in a time proportional to $log(p)$ on a sparse p-node network even at high communication load.

To eliminate the network hot-spots which commonly arise when packets from many different sources collide at a link in a sparse network, two phase routing is employed. Every packet is first dispatched to a randomly chosen intermediate destination; from the intermediate destination it continues to its final destination. This is a distributed algorithm - it does not require any central co-ordination - so it is straightforward to implement and scales easily. Randomization does not, in fact, strictly guarantee a delivery time which is $O(log(p))$ - but it gives it a sufficiently high probability to achieve the universality result. The processors will occasionally be held up for a late message, but not often enough to noticeably affect performance. Simulated results of universal routing are presented in chapter 7.

1.6.1 Randomizing Headers

How is the two-phase algorithm implemented? As a packet enters a randomizing network, it must be supplied with a new, random, header; this header will be used to route the packet to a router which will serve as the intermediate destination. Any input link of a router can be set to *randomize* packets as they arrive. Whenever a packet starts to arrive along such a link, the link first generates a random number and behaves as if this number were the packet header. The remainder of the packet follows the newly supplied random header through the network until the header reaches the intermediate (random) destination.

At this point, the first (randomizing) phase of the routing is complete and the random header is removed to allow the header to progress to its final destination in the second (destination) phase. The removal of the random header is performed by a *portal* in each router which recognizes the random header associated with the router. The portal deletes the random header with the result that the original header is at the front of the packet, as it was when the packet first entered the network. This header is now used to route the packet to its final destination.

Unfortunately, performing routing in two phases in the same network makes the paths of the packets more complicated. The result is that deadlock can now occur.

1.6.2 Avoiding Deadlock

A simple way to avoid deadlock is to ensure that the two phases of the packet transmission use completely separate links. The node numbers are partitioned into two halves: one half contains the numbers used for the randomizing phase. The numbers in the other half are used for the destination phase. Similarly the links are partitioned into two sets: one set is used in the randomizing phase and the other set in the destination phase.

Effectively this scheme provides two separate networks, one for the randomizing phase, and one for the destination phase, with only one set of routers. The combination of the two networks will

be deadlock free if both of the networks are deadlock free. The simplest arrangement is to make the randomizing network have the same structure as the destination network - and to make both employ one of the known deadlock free routing algorithms.

Universal routing can be applied to a wide variety of networks including hypercubes and arrays [5].

1.7 Conclusions

Concurrent machines can be constructed from two components: transputers and routers. Transputers can be connected via their links to form dedicated processing systems in which communication takes place only between directly connected transputers. They can also be connected via routers allowing system-wide communication.

The provision of system-wide inter-process communication simplifies the design and programming of concurrent machines. It allows processes to be allocated to transputers after a program is written in order to optimize performance or minimize cost. It ensures that programs will be portable between different machines, although their performance will vary depending on the capabilities of the specific communications network used.

The communications architecture allows a wide variety of implementations. VLSI routers can provide routing between a large number of links, minimizing network delays. Very fast routers with fewer links can be constructed using high-speed technology. Transputers and routers can be combined on VLSI chips to provide network nodes.

Transputers and routers can be used to build machines in which a balance is maintained between communication throughput and processing throughput. Universal routing can be used to achieve bounded communication delay, and fast process scheduling within the transputers allows this communication delay to be hidden by a small amount of excess parallelism. An immediate possibility is the development of a standard architecture for *scalable general purpose* concurrent computers, as discussed in chapter 8.

References

[1] M. Homewood, D. May, D. Shepherd, *The IMS T800 Transputer*
IEEE Micro **7** no. 5, October 1987

[2] INMOS Limited, *occam2 reference manual*, Prentice Hall 1988

[3] L.G. Valiant, *A scheme for fast parallel communication*
SIAM J. on Computing **11** (1982) pp. 350–361

[4] L.G. Valiant, *General Purpose Parallel Architectures*,
TR–07–89, Aiken Computation Laboratory, Harvard University

[5] L.G. Valiant, G.J. Brebner, *Universal Schemes for Parallel Communication*
ACM STOC (1981) pp. 263–277

[6] J. van Leeuwen, R.B. Tan *Interval Routing*
The Computer Journal **30** no. 4 pp. 298–307 1987

[7] P. Thompson, *Globally Connected Fault–Tolerant Systems*
in Transputer and occam Research: New Directions, J. Kerridge (Ed)
IOS Press 1993