This is Chapter 8 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 C. Barnaby, M.D. May and D.A. Nicole.

8 General Purpose Parallel Computers

8.1 Introduction

Over the last decade, many different parallel computers have been developed, which have been used in a wide range of applications. Increasing levels of component integration, coupled with difficulties in further increasing clock speed of sequential machines, make parallel processing technically attractive. By the late 1990s, chips with 10^8 transistors will be in use, but design and production will continue to be most effective when applied to volume manufacture. A ‘‘universal” parallel architecture would allow cheap, standard multiprocessors to become pervasive, in much the same way that the von Neumann architecture has allowed standard uniprocessors to take over from specialised electronics in many application areas.

Scalable performance

One of the major challenges for universal parallel architecture is to allow performance to scale with the number of processors. There are obvious limits to scalability:

- For a given problem size, there will be a limit to the number of processors which can be used efficiently. However, we would expect it to be easy to increase the problem size to exploit more processors.
- There will in practice be technological limits to the number of processors used. These will include physical size, power consumption, thermal density and reliability. However, as we expect performance/chip to achieve 100-1000 Mflops during the 1990s, the most significant markets will be served by machines with up to 100 processors.

Software portability

Another major challenge for a universal parallel architecture is to eliminate the need to design algorithms to match the details of specific machines. Algorithms must be based on features common to a large number of machines, and which can be expected to remain common to many machines as technology evolves. Both programmer and computer designer have much to gain from identifying the essential features of a universal parallel architecture:

- the programmer because his programs will work on a variety of machines - and will continue to work on future machines.
- the computer designer because he will be able to introduce new designs which make best use of technology to increase performance of the software already in use.

8.2 Universal message passing machines

A universal message passing machine consists of:

- p processing nodes with concurrent processing and communication (and preferably process scheduling).
- interconnection networks with scalable throughput (linear in p) and bounded delay (scaling on average as $log(p)$).

Programs for message passing machines normally consist of a collection of concurrent processes which compute values and periodically communicate with each other. These programs must take into account the relationship between the communication throughput and the computation throughput of the message passing machine. We will call this ratio the *grain* (*g*) of the architecture, and measure it as operations/operand. For simplicity, we will assume that a processor performs an operation in one clock *tick*, so that we can measure the grain in ticks/operand.

The importance of achieving a good balance between computation and communication can be understood by considering a simple example. Suppose that a two dimensional image is to be processed by an array of transputers. Each transputer stores and processes a portion of the image. Each step of the computation involves updating every element of the image in parallel. Assume that at every step of the computation, every element of the array `a[i,j]` is to be updated to:

```
        f(a[i,j], a[i-1,j], a[i+1,j], a[i,j-1], a[i,j+1])
```

and that function `f` involves 4 operations. The following table shows the operations performed for each item communicated for four possible mappings.

elements per transputer	**operations per communication**
1	1
4	2
16	4
256	16

If we chose a mapping which allocates one element to each transputer, we would need each transputer to perform one operation in the same time that it can communicate one data item. This is often referred to as *fine grain* processing. If, on the other hand, we allocate a large number of elements to each transputer, the communications requirements are small. This is often referred to as *coarse grain* processing. It can be seen from the example that as the grain is decreased, the communications capability becomes the limiting factor. At this point, it is impossible to use more transputers to increase performance, but easy to use more transputers to process a larger image.

Specialised transputer configurations can often be used to provide fine grain processing. In the above example a two-dimensional array of transputers could be used, as communication is required only between adjacent transputers in the array. However, a general purpose machine should be able to provide fine grain processing for a wide variety of algorithms, and for software portability it should allow automatic allocation of processes to transputers. To do this it must support a high rate of *non-local* communication, which can be achieved with a suitable network of routers.

Another important factor affecting the performance of parallel computers is the *latency (l)* in communication. A transputer may idle awaiting data from another transputer even though the communication rate between the transputers is adequate. This is normally overcome by using extra parallelism in the algorithms to *hide* communication delays. Instead of executing one process on each transputer, we use the transputer process scheduler to execute several processes on each transputer.

Whenever a process is delayed as a result of a communication, it is descheduled and the transputer activates another process. This in turn will eventually become descheduled as a result of a communication. Execution proceeds in this way through several processes. Whenever a communication completes, the corresponding process is rescheduled ready for subsequent execution. Provided that there are sufficient processes, the transputer will never idle as a result of communication delays.

To understand the use of excess parallelism, consider the following simple *worker* process suitable for use in a processing farm. In a typical farm a controller process would hand out packets of work to many such worker processes.

```
local data, result
loop
{       input ? data
        result := compute (data)
        output ! result
}
```

This process performs input (**?**) to a local variable, computation and output (**!**) from a local variable sequentially. Any delay in performing communication will be directly reflected in the time taken for each iteration of the loop.

Provided that the result output at each iteration of the loop is not used (by the controller) to produce input for the next two iterations, this process could be replaced by the following version which allows input, computation and output to take place in parallel.

```
local data, result, nextdata, nextresult
loop
{       parallel
        {       input ? nextdata
                nextresult := compute (data)
                output ! result
        }
        data, result := nextdata, nextresult
}
```

Here delays in communication will affect the total time taken for the loop only if one of the communications takes longer than the computation. Even larger delays in communication can be tolerated by executing several such processes in each transputer, as in the following version. The *n* processes are all independent of each other, and each operates on its own local variables (data, nextdata, result).

```
parallel i = 1 to n
{       local data, result, nextdata, nextresult
        loop
        {       parallel
                {       input ? nextdata
                        nextresult := compute (data)
                        output ! result
                }
                data, result := nextdata, nextresult
        }
}
```

Here every communication can be delayed by up to *n* computation steps. An algorithm of this kind can be efficiently executed even in the presence of long communication delays.

8.2.1 Using Universal message passing machines

Designing a program for a particular grain (*g*) is a significant task, so we would like to keep g constant for all sizes of machine. In practice we would also like to keep *g* low, as this simplifies programming and allows fine–grained parallelism. A low and constant *g* allows programs to be written at a higher level using, for example,

- Array manipulation
- Big DO-PARs
- Explicit parallelism with lots of small processes

The programmer and compiler will take into account the grain *g*, and will construct a program as a collection of *v* virtual processors (processes) of grain >*g* and cycle-time *c* (ticks). We assume that the processes are cyclical, and in each cycle perform *c/g* communications and *c* operations. Notice that we want to keep the grain of the software as low as possible so as to exploit all possible parallelism for a given problem size, but the grain must be at least *g* to avoid processor idling.

The output of the compiler is a program suitable for use on all universal machines of grain *g*. We expect to keep the program in this form, and perhaps distribute it in this form. We note that *g* is fixed for a range of machines based on the same components, and further that there is likely to be little variation in *g* even for machines based on different components. This means that the compiled program is likely to be re-useable.

To load a compiled program for execution, we make use of a loader which takes as parameter the latency of communication: *l* (ticks). This will vary from machine to machine and will scale as *log*(*p*) for realizable networks. The loader will allocate at least *l/c* virtual processors to at most $(v \times c)/l$ processors. There would be no point in attempting to use more processors than this, as this would result in processors idling some of the time. It would be better to leave some processors available for some other purpose. Thus the program will run with optimal efficiency on a *p*–processor machine provided $(v \times c) > (l \times p)$.

Notice that our loader ensures that there will always be enough processes on each processor to ensure that (at least) one is executable; the others will be waiting for communication to complete. This means that we will need to use at least *log*(*p*) more processes than processors. Another way to think of this is that we could use a specialised machine exactly matched to the algorithm in which each processor executes only one process; this would offer *log*(*p*) more performance. Specialised parallel computers will still be needed for maximizing performance where the problem size is limited!

We note that our proposal for universal message passing is closely related to Valiant's proposal for Universal PRAMs [8] in which $l = log(p)$ and $c = 1$.

8.3 Networks for Universal message passing machines

Universal message-passing machines consist of a number of concurrent processors, connected by a communication network. A suitable network is a *Universal Communication Network*, where the throughput per terminal link remains constant with varying network size, and the delay per terminal link grows slowly with increasing network size. Such a machine is *universal*, as the algorithm running on the machine (made up from the processes on the processors) does not depend on the underlying structure of the machine. This structural independence means that the program structure will not need to be altered if the underlying machine is changed, for instance if it has more or less processors. The machine may be characterized by the parameters *g* and *l*.

Suppose that a process sends a message which will take time *l* to get to its destination. The communication delay may be hidden by the processor scheduling another parallel process (or other parallel processes) during the communication delay. Given the network delay, *l*, we can predict the number of processes which are required to hide the communication latency. It has been shown that several networks have constant throughput per terminal and latency growing with *log(p)*, where *p* is the number of processors. Among them is the n–cube.

8.3.1 A simulation of the n–cube

The 6-dimensional cube is examined. From the distribution of packet arrival times, the probability that a packet takes longer than a certain amount of time is derived. The probability, in turn, is used to predict the amount of parallel slack required. The results compare to the theory of Valiant [8], and follow similar arguments.

The probability (derived from simulation) that a packet delivery time is greater than time T is shown in figure 8.1.

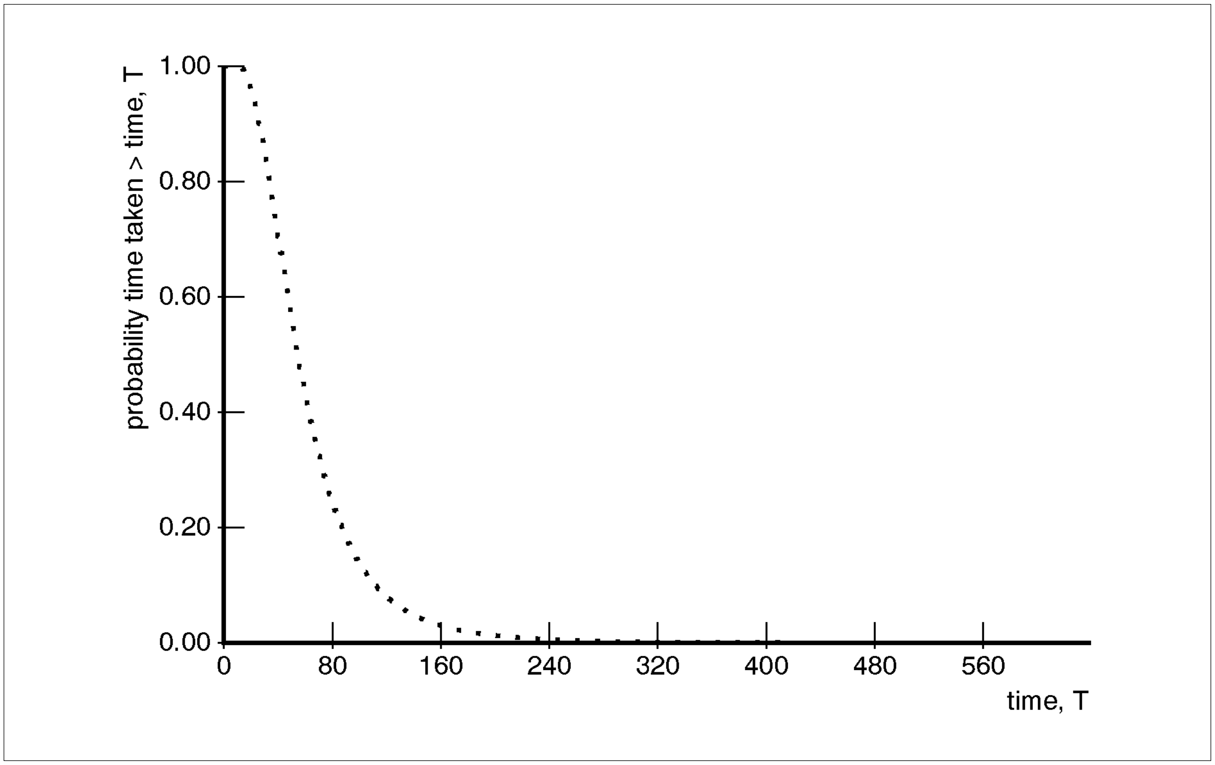

Figure 8.1 Probability that a packet takes longer than time *T* on a 6–cube

There are a number of processes on each processor, in this case 6, which operate one after another, for instance process 1, process 2, ..., process 6, then process 1 again. The time required between process 1 finishing and process 1 starting again is *l*, the latency of the communication. However, although process 1 may not have received its communication, and therefore not be ready to run again, process 2 may have received its communication, and be able to restart. This implies that the probability of no process being ready is actually the product of the probabilities that any one of the processes is not ready (assuming that these events are independent). Note that process 1 has time *l*, whereas process 2 has time 4*l*/5, process 3 has time 3*l*/5, and so on. This does not take account of the compound probabilities of many delays happening in a very short time. The probability of waiting, against the network latency *l*, is shown in figure 8.2.

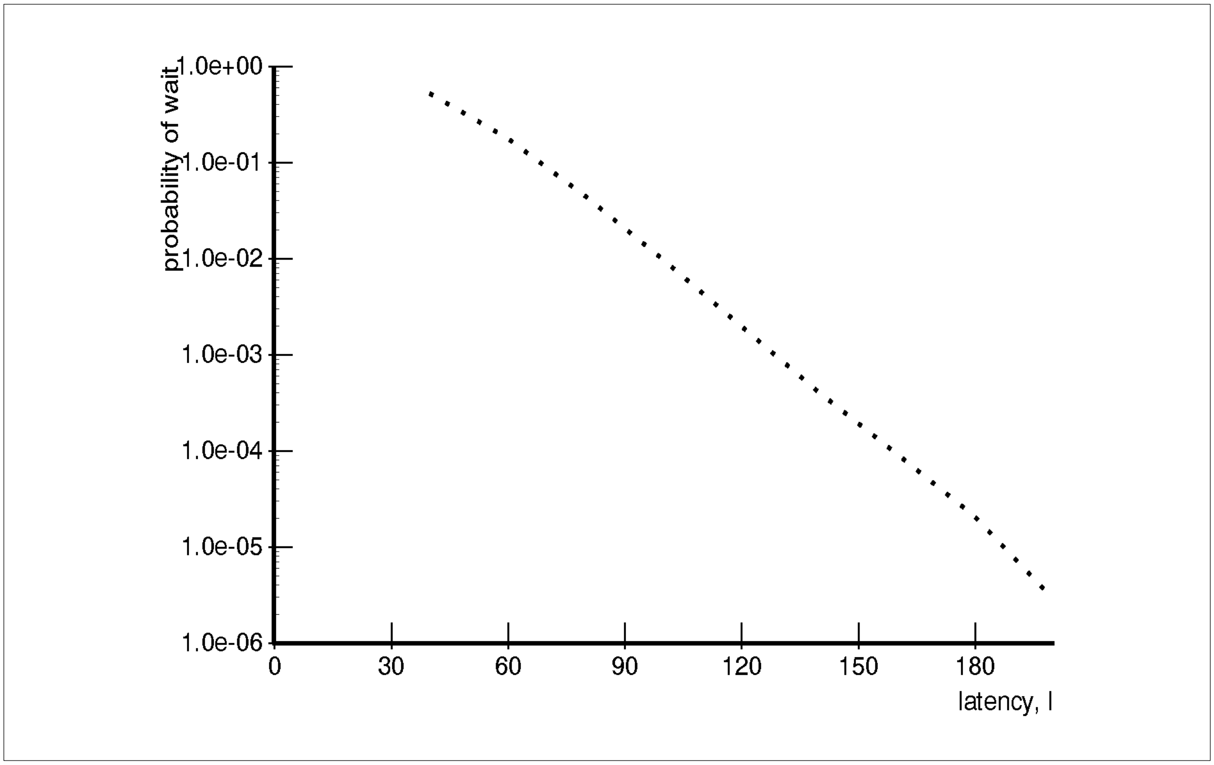

Figure 8.2 Probability that a processor will have to wait

For 6 processes, the graph shows that for a probability of waiting of 10^{-2}, we need about 100 cycles between successive executions of a process. For a probability of 10^{-3}, we need about 130 cycles. These suggest that each process needs to run for about 20 or 26 cycles respectively. This is the cycle size c, defined earlier. In the next section, the effect of the probability of waiting is shown.

The effect on program runtime

In our model, each processor has 6 processes. Each of the 64 processors run their 6 processes repeatedly. Suppose that a delay to any of the processors means that all the processors have to wait for the one which is delayed. Then there are 384 (=6 $\times$ 64) processes each of which require their packet to be delivered within time l in order to avoid a delay to the system. If the system is delayed, it waits for a further l units of time before it continues.

Because we require that all 384 packets are delivered, if the probability of any one packet being delayed is 10^{-2}, nearly all of the cycles will take time $2l$ rather than l. The time required to run the program consequently doubles.

If the probability of a delay is 10^{-3}, about one third of cycles will be delayed. If the probability is 10^{-4}, then about one in 26 cycles will be delayed. These probabilities correspond to particular values of c. The factor of increase in runtime over the case where there are no communication delays, is shown in figure 8.3.

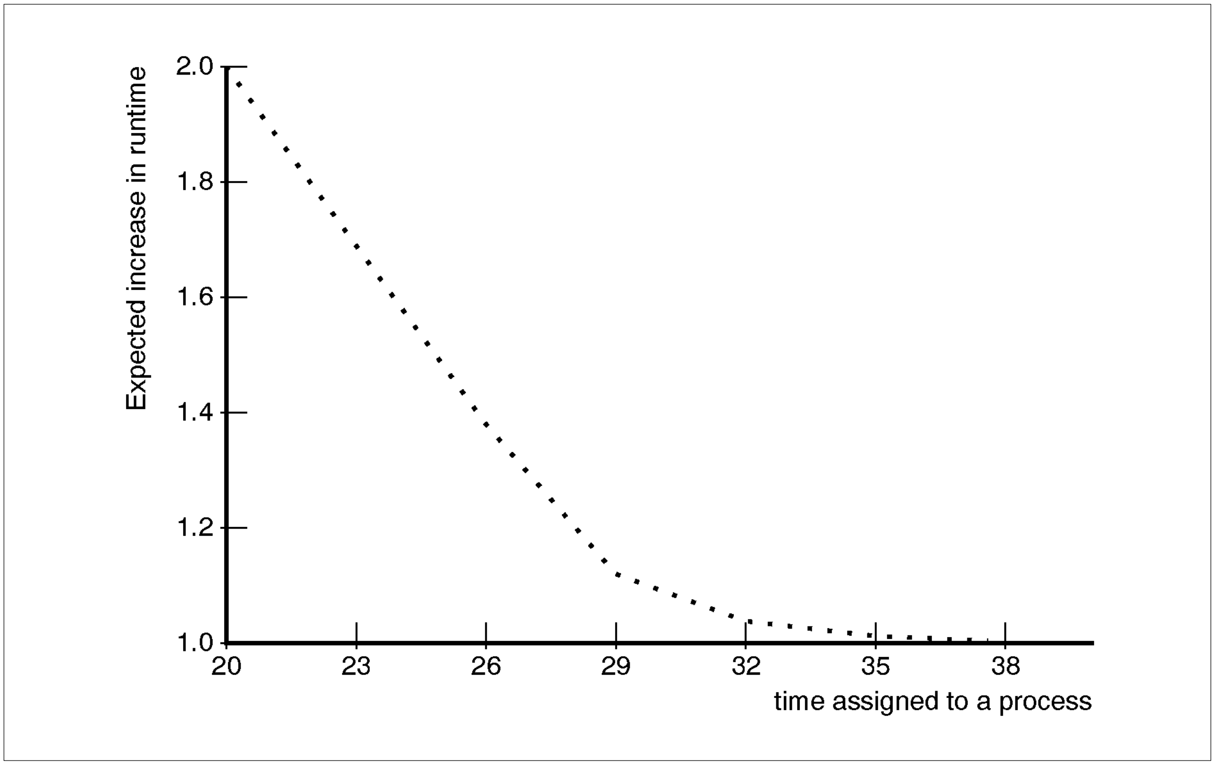

Figure 8.3 Increase in runtime due to latency as a function of cycle time *c*

8.3.2 An example

Suppose we want to run an image smoothing algorithm on a parallel machine. Then to operate where the runtime will be minimally affected, we want to hide a latency of 160 cycles. For 6 processes on each processor, this gives a cycle size, c of (160/5)= 32. A network throughput of about 80% (as simulated for the n–cube) means that about there will be about 25 units of output per 32 units of time.

Let the unit of time be 0.5 microseconds. This is the about time required to transmit a floating-point number using DS-Links. Floating-point values will be bundled into 4 packets, one for each of the $\pm x, \pm y$ directions. The header overhead is very small, so the 25 units of time corresponds to transmitting 4 groups of 6 floating-point numbers.

The image smoothing operation consists of 5 operations per pixel (4 additions and one division). This suggests that splitting a picture into 6 by 6 pixel squares will give four communications (each of six floating point numbers), and 36 calculations per process. Therefore within the 16 microseconds, a total of 36 $\times$ 5 = 180 floating point calculations need to be performed. The corresponding calculation rate is about 11.25 MFlops per processor. Each processor runs six such processes, giving a total of 64 $\times$ 6 = 384 processes in the network. This suggests that an array of such processors will process an image of 386 $\times$ 36 = 13896 pixels without loss of efficiency.

A corresponding calculation for a network throughput rate of 60% suggests that 5 Mflop processors could process a 6144 pixel image without loss of efficiency. As expected, a smaller problem requires a higer ration of communication to computation.

In this example we have taken an algorithm which could be executed on a dedicated two–dimensional grid and re–written it so that it can execute efficiently on universal message–passing machines of varying sizes.

8.4 Building Universal Parallel Computers from T9000s and C104s

Throughout the remainder of this chapter we assume that the basic architecture of the general–purpose parallel computer consists of T9000 processing nodes connected via C104 switches, and examine a number of practical issues in the construction of such machines.

8.4.1 Physical organization

A T9000 runs somewhat hotter than first–generation transputers; a typical T9000 processing module, with dynamic memory and drivers, might be expected to dissipate around ten watts. This power budget can, if necessary, accommodate an error correcting memory subsystem. A small mothercard, with ten T9000s and some C104 switches, might therefore dissipate about 150 watts in an area of about one tenth of a square metre. Such a board would require a cooling air flow of around twenty cubic metres per hour. This is not a huge requirement by the standards of high–performance computer design; a conventional backplane/crate implementation using forced air cooling with a 30mm card pitch is quite reasonable. Fan noise may, however, be considerable and a substantial volume will be occupied by air ducting and fans.

Higher component densities may easily be achieved using contact and/or fluid cooling. The published design for the Parsytec GC supercomputer [1] implements a sixty–four node subsystem in a total volume of about 500 by 300 by 200 mm. This *GigaCube* uses large aluminium contact plates and heatpipes to transport heat away from the active components. Two alternative cooling systems can be provided for the ‘‘cold” end of the heatpipes: a fan and fin module for forced air cooling, or a water cooling block accepting an external water supply. Either module may be accommodated within the GigaCube volume, as is a secondary power supply converting a 42V 40kHz AC power feed down to the 5V required by the modules.

We may contrast these densities with the degree of compactness required to minimize signal propagation delays. Assume that only T9000 data links travel between cards in the computer. Low level flow control on such a link network is maintained on groups of eight tokens (see Chapter 3); such a group takes about 800nS to transmit at the 100 Mbits/s rate. An end–to–end delay of half this figure corresponds to a separation of sixty metres in free space; thus, even allowing for velocity factors, we are able to build very big machines.

Overall, it can be seen that the choice of component density is not constrained by the T9000/C104 architecture; the relatively low power requirements and long permissible cable runs allow the designer full flexibility in mechanical design.

8.4.2 Network Performance Issues

A primary part of the design of a T9000 and C104 system is the design of the data link routing network. Raw throughput and latency are two important issues that must be considered.

Early work by Dally [3] on routing networks suggested that two dimensional grids formed good routing networks for supercomputers. These results were, however, based on parameters which do not apply to C104 networks. In particular, it is desirable to use the very high valency of the C104 to real effect; connecting many links in parallel to form a low valency network wastes much of the routing capability.

There are several possible measures of network performance. One, popular with computer manufacturers, is the peak point–to–point bandwidth between a pair of processors in an otherwise unloaded network. This measure gives some information about the behavior of the processor to network interface, but it conveys almost nothing about the performance of the network itself. Realistic measures must quantify the behavior of the network under reasonable load conditions, taking into account contention between messages within the network. Important effects can arise as a network is loaded:

- Even if the network saturates uniformly as regards throughput, individual message latencies may become very high as the network approaches saturation; serious unfairness may also arise between different processors. Randomization, as offered by the C104, can be shown [8] to make highly delayed messages improbable.

- Certain particular patterns of communication [3] can cause a dramatic build–up of message traffic at particular intermediate nodes in the network. This is a universal property [4] of deterministic sparse routing networks. It is unfortunate that many popular networks (grid, n–cube, Clos...) show this bad behavior on traffic patterns that would be expected to arise in typical computations. Randomization can again be shown to render these systematic collisions improbable.

The use of randomization in a C104 network can be seen to offer important simplifications in the network's behavior. It can completely decouple the network topology from the algorithmic message pattern. One can then essentially characterise a network by its throughput and average latency for randomly distributed traffic under high load. In practice, the adaptive routing offered by the C104 normally provides all the benefits of randomization, along with a useful increase in average bandwidth as will be shown below.

Throughput in a C104 network is limited by *contention*, the simultaneous presence of two or more packets requesting the same output link from a C104. Under random traffic, this may be modelled very simply as discussed in chapter 6. The formulæ derived there may be simply modified to account for restrictions on the output links. It is then straightforward to calculate approximate throughputs for networks by cascading this calculation through the various layers of the network. There are two other direct results from this formula:

- A single 100 Mbit/s link between T9000s can deliver a unidirectional throughput of 8.9M bytes/s. A perfect network could route *permutation* traffic, in which it is guaranteed that no two processors are attempting to communicate to the same destination, at the same rate. With *random* traffic, even for a perfect network, contention at the destination T9000s reduces the maximum throughput per link to 5.6 Mbyte/s.

- Consider an *indirect* network: one in which there are layers of C104 switch that are not connected directly to T9000s but only to other C104s. Then the amount of traffic on these inner layers is reduced by contention in the outer switches adjacent to the T9000s. A balanced indirect network design will thus have a density of links that is highest near to the T9000s and is reduced between the inner switches.

8.4.3 A practical Routing Network

A simple and useful routing network is the folded Clos[23]. The network provides routing between the *mn* external ports on the left side of the network, where each of the *n* switches in the left–hand column provides *m* external ports. A 512–terminal version is illustrated in figure 8.4.

[23] The title is derived from an important early paper [5] on the design of telephone switching networks. The particular numbers of interconnections provided by Clos and the related Benes [6] networks are important for the establishment of telephone circuit connections without contention. These numbers have no special significance for packet routing networks such as those built using C104s.

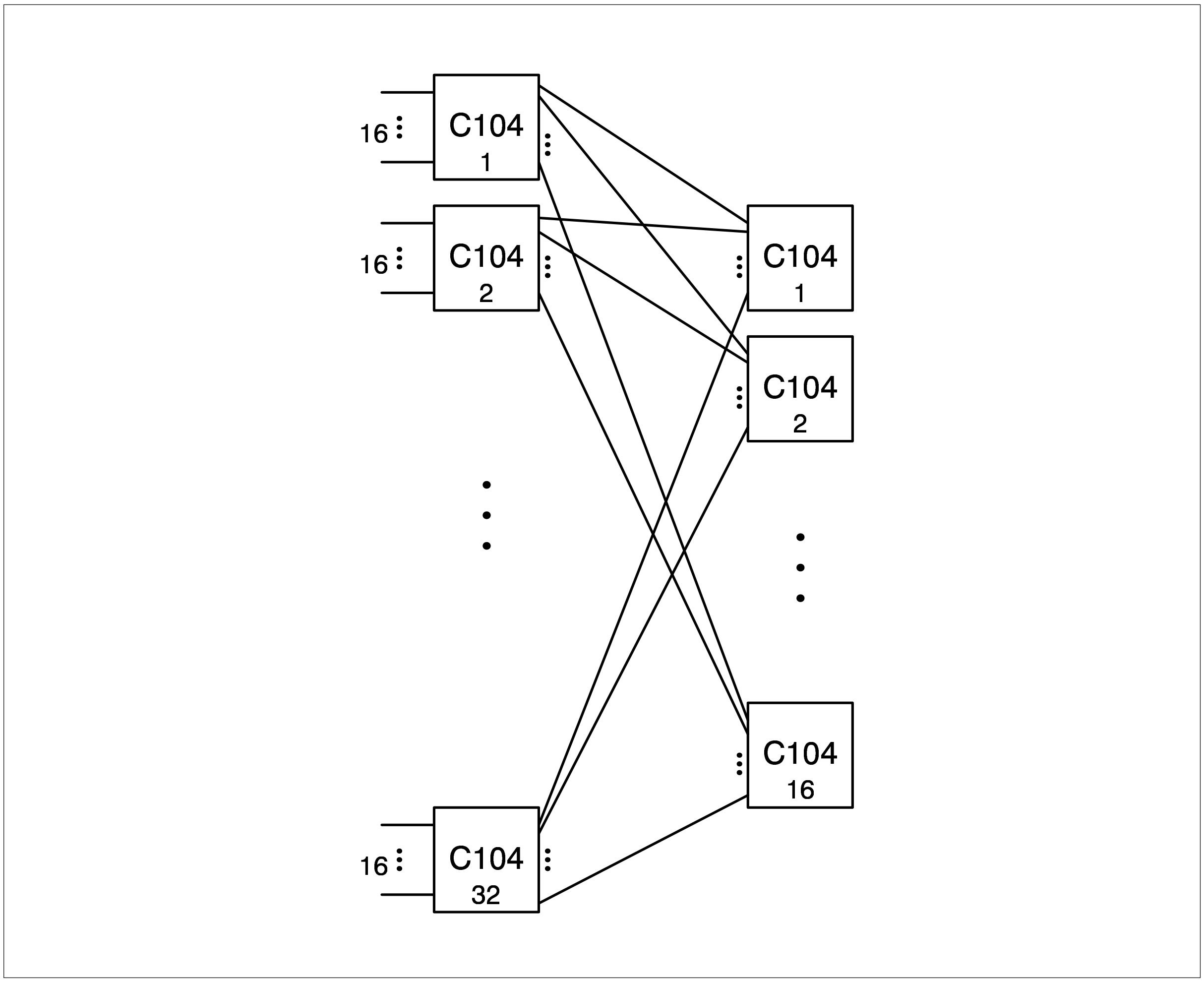

Figure 8.4 Folded Clos network

The simple model of chapter 6 can be used to evaluate the performance of the folded Clos network. If the network is programmed for random routing, then a random one of the *p* right hand switches is selected for each message. The probability of an output being active at the first stage is thus $P_1 = 1–e^{–m/p}$. The probability of an input being active at the second stage is also P_1 and the probability of an output being active, for random traffic, is $P_2 = 1–e^{–P_1}$. Finally, the probability of one of the external output ports being active is $P_3 = 1–e^{–pP_2/m}$, giving an average throughput per link of 8.9 × P_3 Mbyte/s. If grouped adaptive routing is used at the first stage, then the contention there is eliminated as long as $p >= m$. Thus, the previous formula is modified by replacing P_1 with *min(1,m/p)*. Table 8.1 below shows some calculated random traffic throughputs for typical Clos type networks.

Table 8.1 Sustained high–load throughputs for Clos–type networks

m	*p*	random throughput Mbytes/s	adaptive throughput Mbytes/s	random routed efficiency	adaptive routed efficiency
16	16	3.3	4.2	59%	74%
8	16	4.3	4.8	76%	86%

Note that the two networks differ only in that half the external ports are left unconnected for the *m* = 8 network. The randomization applied to folded Clos networks was effectively free; no additional links were traversed by randomized packets. Nevertheless, adaptive routing can be seen to be more efficient. Grid and n–cube networks impose more severe penalties. Random routing must be applied to all but one dimension of the grid or n–cube, almost doubling the traffic density.

Simple adaptive routing also achieves little in eliminating systematic contention from these networks.

Similar methods may be used to analyze a wide variety of networks. Homogeneous networks such as the Folded Clos and n–cube are straightforward. Inhomogeneous networks, such as two or three dimensional grids, have an unbalanced traffic pattern which peaks (linearly) in the center of the grid. Calculation of the contended throughput in the centre of the grid gives a good estimate of the overall throughput of the network.

8.4.4 Routing Network Simulations

Some detailed simulations [2] of C104 networks have been performed by Siemens as part of the Esprit PUMA project. This work was instrumental in the inclusion of grouped adaptive routing in the C104. These studies cover Clos, grid and n–cube networks. Concentrating first on the m=16, p=16 network examined above, the Siemens results find sustained throughputs on random traffic of 2.9 M byte/s for random routing and 3.1 M byte/s for adaptive routing which compare well with the crude calculations of the previous section. Interestingly, for a known bad message pattern, deterministic routing offers a throughput of only 0.3 M byte/s, random routing (of course) the same 2.9 and adaptive routing 8.8 M byte/s. It turns out that the bad pattern for deterministic routing, a block permutation, is a very good pattern for adaptive routing.

The Siemens simulations also give insight into the average packet delay in the network; for the m=16, p=16 system delivering 1M byte/s/link throughput we see an average delay of 6μs. Grids and cubes again perform worse than Clos networks in this parameter.

Overall, the Siemens results show comparable performance for n–cube and Clos networks of comparable cost, with a small advantage for the Clos designs. Two and three dimensional grids performed very badly.

There is a received wisdom that the two–dimensional nature of silicon die and PCBs leads naturally to a two–dimensional network structure. There is little justification for this notion; a real system of modules in boards in crates in cabinets is more naturally tree structured. It is, however, true that the realization of good global messaging networks requires many links bisecting the system. Parsytec [1] have demonstrated a construction technique appropriate for three dimensional grids. The folded Clos network also lends itself to a natural physical implementation, with the processors and outer switched arranged on vertical boards and the inner switches on horizontals as shown in figure 8.5. Such an arrangement will require careful selection of connectors and support boards, but can easily realize a 256 processor system in a single compact rack.

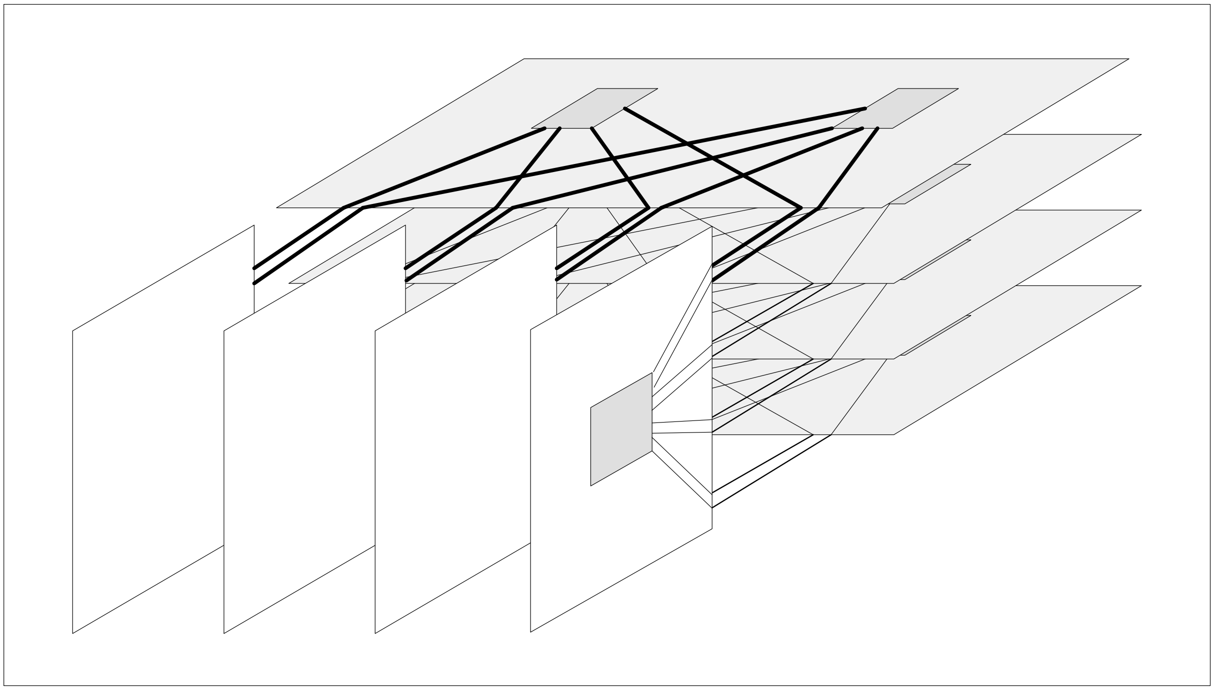

Figure 8.5 Horizontal boards containing the center stages of a Clos network

8.4.5 Security Implications of Network Topology

In some applications, secure multi–user T9000 parallel computers are required. This might be to provide conventional inter–user security in a general–purpose machine. It might also be to improve system ruggedness in the presence of some poor quality software modules. For instance, in a database system, one might hope that a client instance would be able to fail without bringing down the main database.

A simple solution to this problem would be to run all the untrusted processes in protected mode, with all communication and memory management controlled by trusted servers. Unfortunately, for a variety of reasons this is not always possible:

- Users might be using programming environments that insist on raw access to the processors, and do not support protected mode.

- The increased communication overheads of protected mode may not be acceptable.

It is possible to use a C104 routing network in order to provide some security against rogue processing nodes. The concern is that a rogue node might transmit a packet with headers that it is not authorized to use, causing corruption of a virtual channel which it should not use. One trick is to operate the C104s *without* header deletion at the boundary of the networ, so that the virtual channel number as seen by the receiving T9000 is actually used to route the packets. Careful design of the C104 network, and programming of its intervals, can ensue that individual processors can only access restricted ranges of virtual channels on the other processors. This scheme is at first sight attractive, but suffers from severe limitations:

- These schemes tend to require large numbers of C104s and an otherwise undesirable network topology.

- Large gaps are created in the range of virtual channels usable at each processor.

- Most standard programming environments [7] assume the use of header deletion at network boundaries.

- This technique offers error detection, but not error recovery. It is difficult to trace the author of bad packets, and almost impossible to protect against network flooding.

Overall, it seems wisest to accept that C104 networks are not intended to enforce protection, and to use gateway processors between trusted and untrusted subnetworks.

Most of the popular networks lend themselves naturally to rigid partitioning, but usually only in restricted ways. For example, n–cubes can realize sets of smaller n–cubes, grids can be dissected and Clos networks partitioned linearly. It is much harder to assemble closed subnetworks from arbitrary, non–adjacent sets of processors.

8.5 Summary

We have used the following result from contemporary computer science:

- the ability of certain networks together with randomized or adaptive routing to support scalable throughput and low delay (even when routing among the $p \times log(p)$ virtual processors distributed among p processors)

together with the existence of existence of message-passing hardware:

- processors with efficient process scheduling, in which processing throughput and communication throughput are balanced, and
- high–valency routers allowing the construction of compact communication networks with scalable throughput and low delay,

and have shown that we can already construct scalable universal message passing machines. For these machines, we can write scalable, portable software exploiting message passing. Such machines can easily be constructed from available commodity components.

References

[1] *Technical Summary parsytec GC, version 1.0*, Parsytec Computer GmbH, Aachen, Germany, 1991.

[2] A Klein, *Interconnection Networks for Universal Message–Passing System*, Esprit ’91 Conference Proceedings pp 336–351, Commission of the European Communities, 1991, ISBN 92–826–2905–8.

[3] W J Dally, *Performance Analysis of k–ary n–cube Interconnection Networks,* IEEE Trans Comput **6** pp 775–785, 1990.

[4] L Valiant, in *Handbook of Theoretical Computer Science.*

[5] C. Clos, *A Study of Non–blocking Switching Networks*, Bell Systems Technical Journal **32**, 1953

[6] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic* Academic Press 1965

[7] *Network Description Language User Manual*, Inmos Ltd, 1992.

[8] L. G. Valiant, *A Bridging Model for Parallel Computation*, Communications of the ACM, August 1990, pp 103–111