Contention and Achieved Performance in Multicomputer Wormhole Routing Networks

Stephen C. Tweedie

Doctor of Philosophy
University of Edinburgh
1998

Abstract

In modern wormhole-routed multicomputer interconnection networks, contention plays an increasingly significant role in limiting performance at high loads, especially if there is poor communication locality in the workload or if the communication load is non-uniform.

However, the relationship between the level of contention in the communication network and performance degradation in a running workload is complex. At high loads, the communication network may be affecting the rate of injection of new packets into the network just as much as the workload's packet injection rate is affecting performance in the network.

In this thesis we will aim to provide a way of untangling this relationship. We will present a methodology based on discrete event simulation which will allow us to separately identify the cost of contention to a running program, and the amount of contention actually occurring.

We describe a dedicated discrete event simulator used to host performance evaluations of a set of workloads on a 2-D mesh of wormhole routing elements based on the T9000 transputer and its associated C104 routing device. Our simulator is capable of selectively running without contention effects, allowing us to observe not only the amount of contention taking place in the network but also the performance degradation it is causing relative to an ideal, contention-free environment.

We describe a set of metrics which can be used to measure these contention effects. We make a strong distinction between contention internal to the communication network and contention taking place at or before the injection buffers into the network: these are shown to have very different implications for performance.

We also describe a method of classifying synchronisation properties of workloads for which packet injections are not necessarily independent. If there is a feedback loop between network performance and workload performance, then we need to understand if and how the workload may react to changes in the network's performance before we can predict the impact of contention.

Finally we show that the workload classifications and contention metrics we have identified do allow us to distinguish between different levels of workload sensitivity to contention in our networks.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

We live in a time which is seeing an explosive yet sustained growth in the power of computer hardware. Computing power unheard of a few years ago is now common-place in homes, and yet the users of these systems are continually pressing for even more power. In many fields of scientific and numerical computation, the demands of users and of applications have continually exceeded the ability of traditional CPUs to deliver on their own. The parallel multicomputer has emerged as a way of providing unrivaled performance by combining many fast CPUs together into a single unit.

As CPU speeds have increased, it has been necessary to develop better interconnection networks for these multicomputers in order to maintain a balance between computation and communication performance. As the power of multicomputers has grown, more and more CPUs have been combined together. As a result, there is a demand not only for faster interconnection networks, but for more scalable networks too.

Wormhole routing [Dal87, DS87] has emerged as the favoured way to achieve speed and scalability. By allowing packet routers to begin routing packets as soon as enough of the packet's header has arrived to determine its destination, the per-hop cost of forwarding data is reduced to a constant factor, independent of the message's length. Packets can be incrementally routed through a network, being in various stages of transmission by subsequent routers at the same time.

The bogeyman of wormhole-routed network performance is contention [Lee85, KP86, LK90]. If a packet is to be forwarded on a link which is already busy, then that packet must wait for the link to become free. This enormously increases the forwarding latency of that hop, and as path lengths through the network grow, so does the opportunity for contention: contention can take place at each and every hop through the routing network. The onset of contention is hard to model exactly: not only do the characteristics of the workload incur contention, but the

response of the communication network will also have its own effect on the rate at which the workload can inject packets.

In this thesis we will aim to provide a way of untangling this relationship. We will present a methodology based on discrete event simulation which will allow us to separately identify the cost of contention to a running program, and the amount of contention actually occurring.

Our main contributions will be:

- The development of a simulator tool which is instrumented to observe contention, and which can run with contention effects selectively disabled to measure the speedup possible with contention removed.
- The identification of synchronisation patterns within the workload which affect the workload's sensitivity to contention.
- The observation of two very different contention effects: flow-control effects which merely throttle packet injection by the workload, and true internal contention within the communications network. We identify ways of measuring the presence of internal contention reliably in the presence of other throttling contention.
- We observe separately the onset of contention in the communication network, and the onset of contention-induced performance degradation in the workloads, and describe how this relationship depends on the synchronisation characteristics present in the workload.

Chapter 2 will describe the background of wormhole routing networks and of parallel performance evaluation techniques. In chapter 3, we will present a description of the models of communication and computation we shall be using in our investigation, and will present in more detail the simulation tool and the methodology used to carry out that investigation. The simulator tool will be validated in chapter 4.

Chapter 5 will look at the problems of measuring contention, and its impact on workload performance, in the presence of other conflicting factors affecting performance. This chapter will identify some of the ways in which increasing contention affects performance. It will also introduce a classification of workload synchronisation characteristics which have a significant effect on the workload's sensitivity to contention.

In chapter 6, we will look more closely at the relationship between communication performance—the time it takes for the communications network to transport

a packet from source to destination router—and workload performance, the actual amount of work achieved by the workload. We will see how different workloads may be sensitive to contention in different ways.

Chapter 7 will summarise our work and our conclusions, and will present some thoughts for possible future work on this subject.

Chapter 2

Background

In this chapter, we wish to lay out the background for our investigation of contention in multicomputer routing networks. We will look at the various forms of interconnect router and wormhole routers available, and will describe known performance models for these routers and for contention in them. We shall also describe various techniques used to evaluate the performance of parallel systems.

2.1 Multicomputer Interconnection Routers

If scalability is not an objective in designing a multicomputer router, then single-stage routers such as a bus, star or crossbar scheme may be used. However, any truly scalable router architecture will require the ability for multiple routers to be connected together to form arbitrarily large routing networks. We refer to these multiple-node networks as multistage interconnection networks.

Whenever we have got multiple routers passing packets between them, we need to be aware of the possibility of deadlocking. There are many ways of dealing with this problem. One technique is to incorporate enough buffering in the network routers to guarantee that we cannot ever block, but the required buffering in this scenario is dependent on the application workload, which is not typically under the network designer's control. Another method is not to avoid deadlock, but to detect it once it has happened and to break it somehow.

The preferred mechanisms for deadlock avoidance in contemporary multicomputer routers are to either use a routing algorithm which can be proven never to deadlock, or to use a randomised algorithm in which the system is only probabilistically deadlock-free (the probability that a packet is delivered in time t converges to one as $t \rightarrow \infty$).

There are other classifications of routers which can be made:

- A router is *minimal* if it always routes packets along a minimally-lengthed

path.

- A router in which the path is uniquely determined by the start and destination of the packet is said to be *oblivious*;
- Conversely, a non-oblivious router which can choose between multiple routes may be free to choose the least loaded route: such a router is *adaptive.*
- A *deterministic* router always behaves the same way given a certain current state and a certain packet to route, whereas
- A router need not always route a packet on the same channel given a certain state, or always route the same packets in the same order. Routers which uses random decisions as part of its routing algorithm are termed *randomised.*

Other terms are used to describe the routing algorithm (the process of deciding when and where to forward a received packet). In traversing any multidimensional network of routing nodes, a packet may have to travel in more than one dimension to arrive at its destination. We say that a packet *needs* dimension i if its current coordinate in that dimension is different from the destination's. On a given router, a certain output channel is *profitable* for a packet if forwarding the packet on that channel will take it closer to its destination. A packet is *derouted* if it is forwarded on a non-profitable channel.

2.1.1 Deterministically deadlock-free routing

The archetypal deadlock free router for grid and hypercube architectures is the greedy dimension-order router [Gun81]. This has been extended to arbitrary k-ary n-cubes and toruses by Dally and Seitz [DS87] in the extended e-cube routing algorithm, which breaks links into separately-flow-controlled *virtual channels.* This router has the property that there is only one possible path for any message delivery through the network, but that the set of paths for all possible messages is such that cycles can never form and so deadlock cannot occur. This router has numerous advantages: it is simple to implement, its paths are always minimal and its behaviour is well understood.

The main disadvantage of the dimension-order router is that it is necessarily oblivious—packets always have exactly one possible next hop as they traverse the network, and there is no scope for choosing the least heavily loaded route amongst alternative paths. A number of ways of overcoming this have been proposed, such as the Planar-Adaptive Router [CK92], which use minimal adaptive

routing (which is not normally deadlock free in its simplest formulations). The solutions usually revolve around the use of virtual channels to remove the deadlock potential inherent in the adaptive communication patterns, either by restricting the adaptive flow of packets or by imposing extra packet priorities to avoid deadlock.

2.1.2 Probabilistically deadlock-free routing

Probabilistic deadlock freedom is another way of avoiding deadlock. Probabilistically deadlock-free systems are often non-minimal, in that they incorporate the ability to randomly deroute packets and hence do not always result in minimal path lengths for packet delivery. However, this independence of any given path through the network means that these systems cannot get caught in a cycle of path dependencies, so they are trivially deadlock free. A harder property to prove is livelock freedom, and various techniques are used to ensure this, including forms of timestamping which ensure that the oldest packet in the system is always guaranteed to be delivered.

One of the more interesting ways of achieving livelock freedom is simply to incorporate an element of pure randomness into the routing algorithm. By selecting packets to deroute at random, rather than (say) oldest first, we break the regular repeating loops which are a feature of livelock occurrence. In such systems we can prove that the probability that a packet has still not been delivered after time t decreases exponentially in t, and this is usually considered a satisfactory bound on performance.

2.1.3 Wormhole Routing

Traditionally, most multicomputer routers have used store-and-forward routing: a packet had to be completely received by a router before the router could begin to forward it out on another channel. Dally and Seitz [DS86] introduced wormhole routing to improve performance. Wormhole routers allow a packet to be forwarded once only its header has fully arrived, reducing the per-hop packet latency enormously and making larger diameter communication networks efficient. A Survey of wormhole routers can be found in [NM91]. [MJTR94] surveys collective communication techniques for wormhole-routed networks.

2.2 Existing Routing Algorithms

2.2.1 Greedy Routing

Perhaps the most common and certainly the simplest routing algorithm for many cases is the dimension-order greedy router. This router is the standard basic router for k-ary n-cube networks (regular networks of dimension n and with k nodes per edge). It works by imposing an arbitrary order on the dimensions of the network, and always routing packets along the most significant needed dimension. The imposition of dimension order on the network guarantees that the network is deterministically deadlock-free in all cases.

Greedy routing can be adapted for use on torus networks. The basic greedy router suffers from deadlock conditions when used naïvely on a torus, but in [DS87], Dally and Seitz propose a way of avoiding this problem. By splitting communication edges into pairs of independently buffered, *virtual* channels, they can impose a strict ordering on packets which guarantees deadlock-freedom. However, due to the way in these virtual channels are assigned to communication edges, the resulting torus is not completely uniform.

An important property of the greedy router is that, being both oblivious and fully deterministic, any packet's entire route is determined purely by its source and destination address.

2.2.2 Two-phased Random Routing

In [Val82], Valiant introduces a randomised, oblivious routing strategy called Two-phase Random Routing. The problem Valiant identified was that many common parallel algorithms require communication patterns which, under the greedy router, result in highly non-uniform utilisation of the underlying communication network. Such a pattern is usually bad for performance, since the most heavily loaded edges act as bottlenecks, limiting the utilisation possible on the rest of the network.

Valiant observed that while many common permutation communication patterns resulted in this non-uniform load, a random permutation would be expected to result in a much more even load. He proposed that instead of routing each packet deterministically to its destination, they should first be routed to an intermediate node, chosen uniformly at random, and then forwarded to the ultimate destination. (Note that to avoid deadlock, it is important that these two phases of the communication not interfere with each other; in practice this is most easily achieved by reserving separate physical communication channels for the first

and second phases.) The result is that the original permutation workload has been broken up into two random permutation exchanges, and Valiant was able to prove that this random routing had optimal average-case behaviour for performing permutations in $O(\log(N))$ time on sparsely-connected networks of N nodes.

The two-phase random router is particularly important in our transputer-oriented study due to the fact that the C104 transputer routing device is capable of performing random routing in hardware. The C104 allows any input channel to be configured such that a packet entering that channel gets prefixed with a new, randomly chosen destination address; or that a packet is stripped of its initial header and routed according to the remaining header. These two mechanisms can be used to route newly injected packets into the first-phase network and to forward them into the second-phase network once they have arrived at their intermediate random destination.

2.2.3 The Chaos Router

The chaos router [Bol93, BFS94] is an example of an adaptive, randomised, probabilistically livelock-free router. The chaos router adds to the normal input and output buffers a *multiqueue*, a buffer large enough to hold d packets where d is the degree of the router node (including injection and delivery channels). All buffers are virtual cut-through.

The routing algorithm for a dimension i proceeds as follows:

> A dimension i is *interesting* if output frame i is empty and a packet in either the multiqueue or one of the input frames needs dimension i.
>
> If the current output frame is interesting, the router performs, in the following order:
>
> - Select the oldest packet in the multiqueue that needs the current channel. Move it into the output queue and, if the current input frame is non-empty, move that packet into the multiqueue.
> - Randomly select an input dimension containing a packet that needs the current output channel. If the selected input dimension is the same as the current output dimension, move the packet into the output frame. Otherwise, if the current input input channel is not empty, do the following. If the multiqueue is full,

deroute a random packet in the multiqueue by moving it to the current output frame. Read the current dimension packet into the multiqueue. If the selected input dimension is not the current dimension, and the current dimension is empty, move the selected input packet to the current dimension output frame.

By choosing packets to deroute at random, probabilistic livelock-freedom is achieved although the possible time to deliver any packet is strictly speaking unbounded. The fact that derouting can happen makes this a non-minimal router.

2.2.4 The Forced Router

In [GBES93], Germain et al. present the *forced* routing strategy, an algorithm designed for efficient implementation on silicon. Like the Chaos router, Forced routing uses non-deterministic derouting of packets to try to avoid blocking packets for too long.

The Forced router makes use of the observation that if our communications hardware uses symmetric bidirectional channels, then the total input bandwidth of any router is precisely equal to its output bandwidth. This parity means that if we never buffer any incoming data, it is always possible to route any incoming packet immediately (although not necessarily on a profitable channel, of course). The forced router is a simple routing algorithm which randomly routes packets along any available profitable edge, and if none are available, along a randomly chosen non-profitable edge. The router is trivially deadlock free for any network topology, but Germain et al. also prove that for the case of the 2D grid, forced routing is also livelock free.

2.2.5 Other routing strategies

There are many other variants of routing strategy. In particular:

Displacement router. Similar to the forced router, the displacement router optimises the number of profitable forwardings at each time step. However, it is a synchronous router in which all packets are similarly sized and arrive in lock step, and in which the routing decision is performed simultaneously on all buffered packets once per timestep. The transputer communication architecture is fully asynchronous and is not suitable to this form of routing.

Hot-potato routing (and the related Hot-spot avoiding router). These were both implemented on T800 based systems and so are oriented towards

software-based packet forwarding systems. They go to great lengths to balance not only the communication load over each edge, but also the routing load at each CPU. Being software based, these routers can cheaply piggy-back local load information on outgoing packets and extract such information about their neighbours from incoming packets.

Chinn [Chi95] describes a minimal adaptive router for nxn grids, using constant sized queues and basing routing decisions on the distance a packet is from its destination; it routes permutations in $O(n)$ time.

Other objectives for wormhole routing networks include fault-tolerance. Linder and Harden [LH91] introduce an adaptive, fault-tolerant router using virtual channels on k-ary n-cubes,

Work has also been done to produce performance models of wormhole routers based on physical characteristics. Chien [Chi93] models performance on just two key characteristics: router delay and flow control delay, and [KC94] looks at cost models including software costs, again identifying flow control as a major cost.

A number of other routers have been proposed; see [NM91] for a description of some of the various other routers known.

2.3 Topologies

The 2-D mesh and hypercube, both examples of a k-ary n-cube, are amongst the most familiar network topologies available to the the multicomputer designer, but are by no means the only ones. Dally [Dal91] introduces the "express cube", an augmentation of the k-ary n-cube which adds extra paths which skip a number of intermediate nodes in a single dimension, reducing the total diameter of the network in each dimension.

Leiserson [Lei85] describes the "fat tree" topology, a scalable variant on the n-ary tree. The Connection Machine CM-5 [LAD+92] uses a 4-ary fat tree. Youssef and Narahari [YN90] describe a hybrid Banyan-Hypercube network with better diameter and average distance than a similar hypercube. Others include the "De Bruijn network" [SP89] and "star graph" [AHK87].

2.4 Contention and Hot Spots

Lee [Lee85] studied simple hot-spot patterns on multi-stage store-and-forward communication networks, and observed that in some cases extra buffering in the routers could alleviate some of the performance degradation. Lang and

Kurisaki [LK90] also look at this problem, and claim that in cases of congestion, derouting of packets—forwarding a blocked packet to a non-optimal next hop—can improve the performance around the hot spot substantially.

Kumar and Pfister [KP86] also study the onset of hot-spot contention, looking at the time it takes for a hot-spot to disrupt traffic patterns. They conclude that a hot-spot can seriously degrade network performance in a very short period of time (10 to 50 instruction execution times in their simulations).

Enbody two workload measures and a model to predict contention based on those measures: [KE95] introduces the metrics "max load traffic" (the ratio of maximum to average process graph traffic rates), and "path contention level", the average number of virtual circuits whose paths share any given physical link on the network. These two metrics are shown to give good ability to predict the performance of uniform and non-uniform loads.

2.5 Performance Evaluation Tools

It is often infeasible to obtain performance data about parallel systems by directly benchmarking a running system. The system to be evaluated may not exist in physical form, benchmarking may be too intrusive, or the system may simply not support the required instrumentation. Hence, we often want to build a model of some parallel system and to evaluate its performance algorithmically.

There are two main classes of performance evaluation tool which we can use: analytical tools and simulations. Simulations generally allow more detail to be included about the running system, but they are usually much slower to generate their results than analytical tools. Analytical methods, on the other hand, make simplifications about the system which allow them to solve its performance using numerical methods.

2.5.1 Queuing Networks

Queuing networks [Kle76, All90] are the most commonly used tools for analytical performance evaluation. Queuing networks model a system as a set of interconnected queues. As resources or jobs enter these queues, they are consumed at a rate controlled by random variables. This is the essence of queuing theory: it does not model the precise movement of resources from one queue to the next: queue states are represented not as definite occupancies but by statistical probability distributions. The solution of the queuing network involves solving the equilibrium state of all the queues.

A summary of queuing theory applications to computer science performance problems appears in Lavenberg [Lav89].

2.5.2 Petri Nets

The Petri net [Pet81, Rei85] is another way of describing a parallel system, in terms of resources ("tokens") passing between states ("places") according to predefined possible transitions. An attraction of Petri nets is that concrete statements may be made about their state spaces; tools such as GreatSPN [CFRR95] or others [KL92, Lin92, GKZH95] can automate the state-space reachability analysis of a Petri net, for example. A second advantage is that although the performance of a Petri net can be solved by constructing an equivalent queuing network to the Petri net, it is also possible to solve Petri nets directly by simulation thus encapsulating more of the fine detail of the system's behaviour.

2.5.3 Simulation

The use of simulation as a performance modelling tool has many advantages. A simulation model does not need to suffer from the statistical approximations which affect models such as queuing networks; complex timing effects can be mimicked precisely on a simulation. There are a number of ways in which a simulation may be driven. It may be performed automatically from a descriptive input such as a Petri net, or the simulation instructions may be written in a dedicated language such as SIMULA [Poo87]. (Of course, any general-purpose programming may also be used to construct a dedicated simulator.)

Chapter 3

Models and Tools

In this chapter, we will put in place the background behind the experimental results in subsequent chapters. We will describe the model of computation and communication used and the tools used to perform the study.

The first major section of the chapter will deal with our models. We identify the model of communication used in our experiments—the low level wormhole routing model we assume and the topology of our multicomputer interconnection networks. We also describe the computation model within which we will define our workloads. Much of this explanation will refer to discussion about what we are trying to achieve in our experimental work, identifying parts of the model whose performance we are interested in studying and parts which are of only incidental value to the work.

Our second major section concerns the methodology and tools we will be using to investigate performance in multicomputers. We will consider why discrete event simulation was chosen as the method for obtaining our performance data, and will describe the design and implementation of the software tools used to perform the simulations.

In this section we will also discuss aspects of the experimental methodology we used, and the software tools implemented to support that methodology. We will describe the front end used to run multiple simulations over the course of a single experiment, and the back end data extraction tools used to parse simulation log files.

3.1 Selecting a Multicomputer Architecture

Our first task will be to identify a multicomputer hardware architecture to use as the base for our communications models. We are interested only in wormhole routed architectures, but within that boundary we also want a model which is

realistic. For results obtained from experiments on a constructed communications model to be useful, the model must be reasonably close to at least some real implementations of multicomputer routing networks.

We have chosen to base our communication model on an existing wormhole routing architecture: the T9000 transputer plus its associated wormhole routing device, the C104. There are a number of attractive characteristics of this architecture:

- The T9000 and C104 are genuinely realistic: they have been commercially available and real, production multicomputers have been built from these components.
- The T9 transputer communications have been well documented [MTW93]. It is possible to build a model of the T9000/C104 which models their communications performance quite closely. These two points together are the most compelling reasons for having chosen the transputer architecture for the basis of our work: few production-quality multicomputer interconnection components have as much good quality documentation on their behaviour available.
- The C104 is a routing component with large in- and out-degree. Each C104 has 32 full duplex communication links, and this allows great flexibility when connecting routers together to form a multicomputer network. Many useful topologies may be realised using these devices.
- The T9000 is also well suited for execution of parallel programs, providing efficient and standard hardware support for multiple threads of execution and for inter-process communication over virtual links.
- Communication in the C104 network possesses several attractive properties: it is
  - deterministically deadlock-free (typically using greedy dimension-order routing)
  - lossless
  - self regulated—no software flow-control is necessary.

Note that when we look at our simulation environment later on in this chapter, we will identify a number of simplifications which can be made in the T9000 communication model for the purpose of increasing simulator efficiency. We will

describe these simplifications in more detail when we come to look at the simulator implementation in section 3.5.4 and will evaluate them in chapter 4, but for now it is sufficient to note that we are using the transputer model as a base for our own implementation, but are not necessarily bound to implement the transputer hardware precisely in every detail.

3.1.1 Features of the T9000/C104 Communications Model

The validation of the realism of our communication models will necessarily depend on the details of the real target architecture: in chapter 4, we will be using the documented behaviour of the T9000 family as an important tool in evaluating our simplified communication models. Furthermore, we will see in chapter 5 that there are a number of characteristics of the transputer's communication architecture which can substantially affect the interconnection network's behaviour under heavy load, so we now need to describe the important details of the T9000 and C104's communication mechanics.

3.1.1.1 T9000 Inter-Process Communication

Communication between two T9000 processes on different processors occurs over objects known as *virtual links*, which are supported directly by the T9000 hardware. (Note that these are very different from the virtual links described in chapter 2 as a mechanism for breaking deadlock in multicomputer packet routers.) Instructions for sending and receiving data over virtual links are present in the basic T9000 instruction set. A descriptor in memory for each virtual link determines the destination for data messages sent over that link. Communication between two local processes is implemented by a simple memory copy operation, but communication between processes on distinct processors invokes the T9000's internal communication hardware.

An important consequence of the use of virtual links is that communication can never exhaust available buffer space. We can only ever send to an existing virtual link, and there is always space reserved for one incoming packet for each virtual link.

Whenever a data message must be sent over an external link, the message is broken up into packets of not more than 32 bytes each plus a small routing header (defined in the virtual link control block). This distinction between *packets* and *messages* is important; the packet is the smallest unit of data transferred by the communication network, but the message is the smallest unit of data sent or received by a process. Every packet is part of a message, and every message

delivered to a non-local process is encapsulated for routing in one or more packets.

Each packet is acknowledged by its receiver upon receipt of the packet's header, but the acknowledgement will be deferred until buffer space becomes available if the receiving T9000 has not allocated space into which the incoming message may be copied. A sender will not start to transmit another packet on any virtual link until the previous packet's acknowledgement has been received—in short, the architecture naturally enforces flow control on each virtual link.

The use of separate, independently-flow-controlled virtual links for each virtual inter-process communication channel allows the physical communication links to fairly service all of their virtual links. No virtual link can monopolise a physical link by sending large messages; the component data packets will be queued on a first-in, first-out (FIFO) queue local to each physical link. Data and acknowledgement packets are queued separately at each link, and acknowledgements take priority over data packets.

3.1.1.2 Inter-Processor Communication: the DS-Link

The input and output links of both T9000 and C104 devices are completely compatible with each other. They conform to a standard referred to by Inmos as the *DS-Link*.

A DS-Link is a bi-directional, full-duplex serial communication link with integral flow control and dedicated input and output buffering on each link. Current DS-Link implementations operate at a transmission speed of 100 Mbits/s. Each T9000 and C104 can communicate simultaneously over its DS-Links; there are four such links on a T9000 and 32 on a C104.

Physically, communication takes place as the transmission of *tokens*, which can take one of five forms:

Null token : merely a filler token for an idle link.

End-of-packet : marks the end of a packet.

End-of-message : marks then end of the final packet of a (possibly multi-packet) message.

Flow-control : grants the receiver permission to send a further eight tokens.

Data : transfers a single byte of data belonging to a packet (either a header byte or the packet data contents).

Flow control is managed by the exchange of flow control tokens. Receipt of such a token on a DS-Link confers permission to send another 8 data tokens on

that link. Thus, a data stream in one direction over a DS-Link generates a small amount of traffic in the opposite direction. This means that the bandwidth of a DS-Link is different for uni- and bi-directional data loads.

3.1.1.3 The C104 Wormhole Routing Chip

The C104 routing chip consists of 32 buffered, bi-directional communication links, with routing logic duplicated on each link, plus a central crossbar switch. Incoming packets can be routed as soon as their routing headers have arrived. If the packet's destination output link is currently in use by another input link, then the packet will be queued for delivery once that output link becomes free.

The C104 maintains a small amount of buffering on each input and output link. The buffers are large enough to hold two normal data packet, assuming that the data packets are of the size generated by T9000 processors (the router will happily route much larger packets if they are injected into the network, however).

3.2 Multicomputer Model

We have now defined the individual components used in our model multicomputer architecture, but we have yet to assemble them into a complete multicomputer. There are two levels at which we need to define the multicomputer layout:

- the format of each node in the multicomputer, where the T9000 processing element meets the C104 routing network; and
- the overall, high-level topology of the routing network.

Format of the Multicomputer Nodes We start by looking at the individual nodes. Throughout this study, we will be using as a basic building component a single unit formed of one T9000 CPU connected directly to its own router, as in figure 3.1.

In every case, the T9000 is acting as a source and sink of data, but is not participating in the routing of messages (except for the case of communication between two local processes).

One of the important features of this model is that we permit a multiple-link ("wide") connection between the CPU and the router. Given that each C104 node in our network is eventually going to be connected to more than one neighbour at once, it will be possible for any one router to be transmitting multiple packets into the network simultaneously. If we restrict the CPU's direct connection to

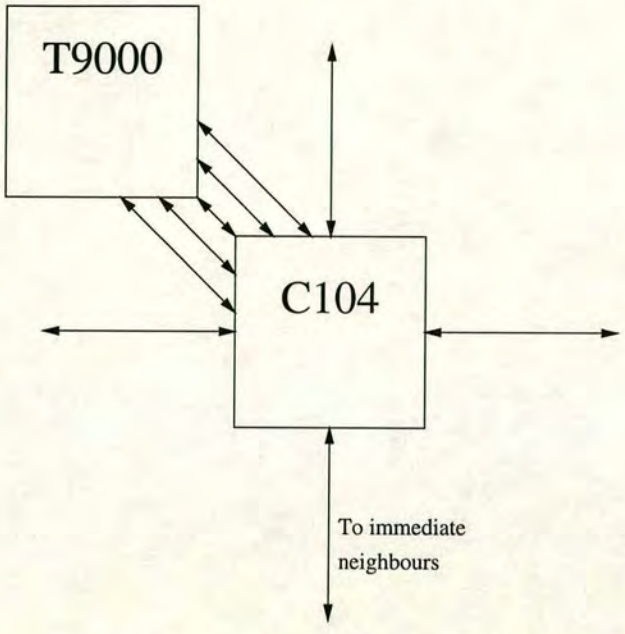


Figure 3.1: Basic multicomputer building block

one packet at a time then the bottleneck in the system may in fact occur at the introduction of the stimulus workload into the network, and not inside the network itself. Widening the CPU link enables us to reduce or avoid this effect and to concentrate on observing the behaviour of the network itself. This effect will be looked at more closely later on (see section 4.2.2).

The C104 allows these wide links to be configured very simply. A "fat link" is simply a set of DS-Links grouped together for routing purposes. The grouped set of DS-Links is referred to as a "hunt group". Any packet queued for output on one of the DS-Links in the hunt group may be forwarded as soon as any one of the DS-Links in that group becomes free.

We will use a consistent labelling of DS-Links to distinguish between links between the T9000 and C104, and links between adjacent C104s in the routing network:

Definition 3.1

An "injection" link is defined as an output link from a T9000, injecting packets into the communication network;

A "delivery" link is defined as an output link from a C104 which is connected to a T9000 and which delivers packets to that processor. Both delivery and injection links are known as "external" links; and

An "internal" link is a link between adjacent C104s which forms part of the routing network itself and is not connected to any processing elements.

The Multicomputer Topology We also need to define how these individual nodes will be assembled to form a complete multicomputer for our experiments. Throughout our study, we will be looking exclusively at 2-dimensional square grid configurations . This was not a completely unrealistic choice at the time when we began our work, but it certainly does not represent the state of the art in multicomputer design. Most current multicomputers use a 3-dimensional torus as the minimum interconnection network, with hypercube and fat-tree configurations also popular.

However, the 2-dimensional grid, although it suffers from relatively low interconnection degree and (hence) relatively high network diameter, is in fact a good choice to start with when studying contention, precisely because of the large network diameters. The increased opportunity for packets to meet and interact within the network may make it much easier to identify contention effects.

Our simulation tool is not restricted to 2-dimensional grids, and it is possible to run experiments on higher-dimensional grids or hypercubes in its current form, but all of the experiments described in this study will be using the basic 2-dimensional grid format.

3.3 A Model of Parallel Computation

Our main area of interest in this study is the performance under contention of multicomputer wormhole routing networks. However, to exhibit different behaviours in the network we of course need to start off by creating a workload to run on that network. In the final analysis, this workload is only required as a source of message injections and receives to exercise the communication network, so our model of computation need not be complex—all we want is to be able to produce various patterns of communication load.

We have already mentioned that our workload is only interesting in that it serves to inject a certain pattern of messages into the communication network: we have no interest in modeling the contents of the messages being passed between processes or the results of any computations. However, we do place some restrictions on the computation model. In particular, although the workloads we will be running in later experiments shall be synthetic and not necessarily characteristic of any particular parallel algorithm, we do require that the computation model, as the communication model, be realistic.

We first of all start by identifying exactly what we are modelling. Figure 3.2 illustrates the general process by which a parallel program is first expressed,

then compiled and executed. We can split the entire run-time system into two parts. The first part is the one which is responsible for coordinating the many individual processes which make up the parallel program. This layer may decide how to partition the workload into individual processes, and how to map those processes to the available processors.

The layer in which we are particularly interested for the purpose of this study is a layer below this. In other words, we want to consider the workload only as some set of communicating processes, ignoring the details of how those processes were requested by the application programmer. We are looking at the workload from below. At this level, we model the application as quite discrete processes, and any communication between these processes (either requested explicitly in the programmer's source code, or undertaken on the programmer's behalf by the run-time libraries or skeletons) forms the communication workload that this layer sees.

So, we define the workload not in terms of any high-level application or computation, but simply as a fixed set of distinct processes which consume CPU time and communication bandwidth in a fixed pattern. We explicitly ignore the possibility of an application which adapts dynamically to the architecture's performance under load. Considering such applications would create a feedback loop—the application behaviour depending on the network performance depending on the workload behaviour—which would distort the behaviour we want to measure. Studying these adaptive applications is a legitimate exercise, but not the one we are performing here.

The fact that we are adopting a computation model which requires the set of communication processes to be fixed does not necessarily constrain our ability to model algorithms in which processes are created or die, but it does make simulating such algorithms more difficult. Such processes would have to be modelled as a set of long-lived processes which become active or inactive at various points in their lifetime.

In figure 3.2, we describe the interface between the workload and the routing hardware as being governed in the general case by an intermediate Operating System. In the case of the transputer architecture, process communication actually takes place without any O/S intervention, using dedicated machine language instructions for communications. Operating System operation is therefore not represented at all in our computation model.

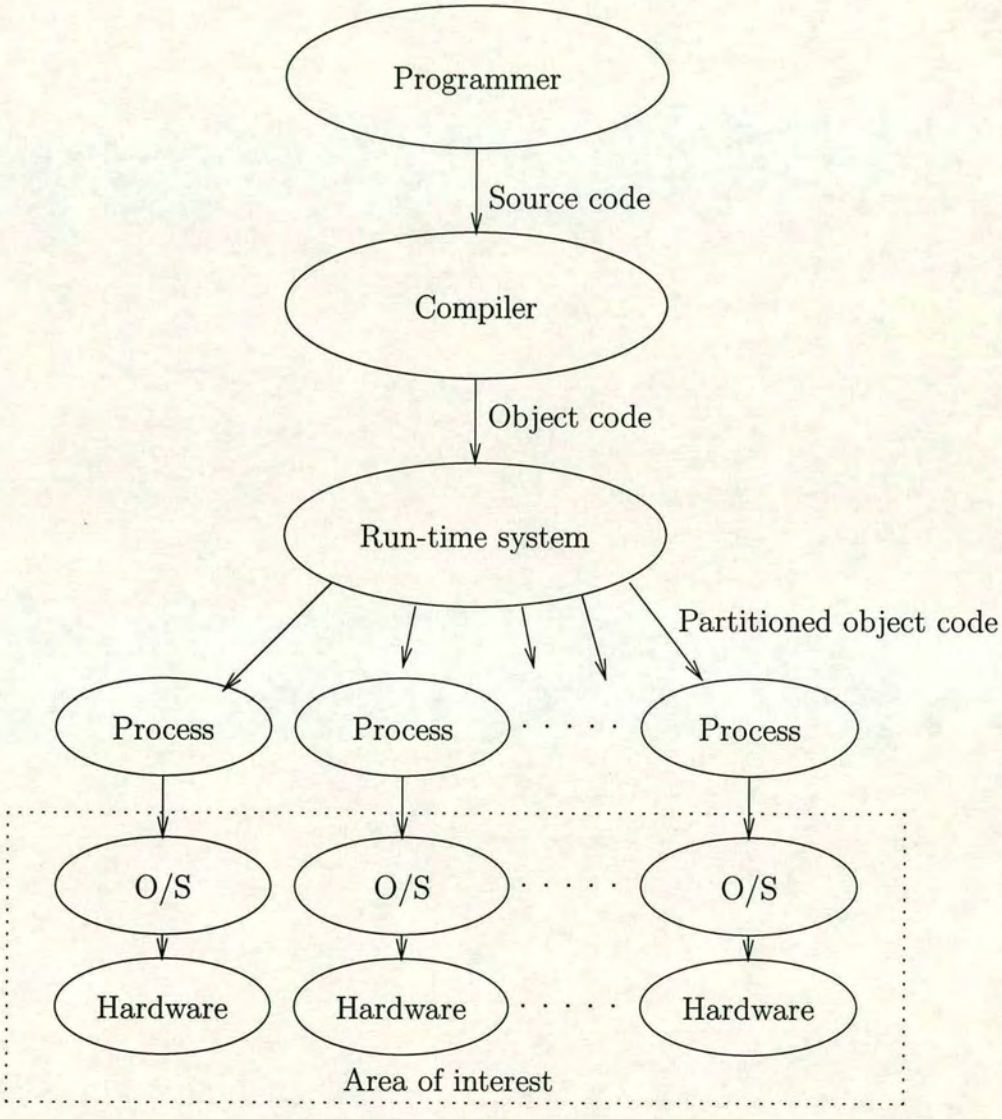


Figure 3.2: The parallel program's path to execution

3.4 Simulation Methodology

We now come to look at the experimental approach we will take to study the performance of wormhole-routed interconnection networks. We have a number of points to discuss:

- The selection of discrete event simulation as our performance modelling tool;
- Description and properties of the simulator code developed for this study; and
- Our experimental methodology: the tools and techniques used to run entire experiments composed of multiple simulation runs, and to extract meaningful data from those experiments.

3.4.1 Selecting a Performance Modelling Technique

The first question is, why was discrete event simulation selected as the tool for performance modelling? We described in chapter 2 a number of approaches to performance evaluation, including benchmarking, simulation, queuing networks and petri nets. In our case, benchmarking was eliminated as an option straight away, partly due to the unavailability of appropriate hardware, but more importantly because much of our work will involve making observations on systems which would not easily be made using software instrumentation on a live system, and also because using a real multicomputer as our platform would prevent us from changing some of the properties of the communications network in ways we would like to do.

Both queuing networks and petri nets are quite popular as tools for performance modelling. However, our investigation here is concerned with communication contention, and contention is a property which arises from the coincidence of specific events—coincident injection of packets or arrival of packets for forwarding between a specific pair of nodes. Moreover, we will be trying later on to investigate the effects of synchronisation properties within the workload on contention patterns. To perform this work, we will need an evaluation platform which will allow us to reproduce these synchronisation effects.

Unfortunately, while queuing networks are useful for evaluating throughput effects, they are not good at encapsulating synchronisation effects. Queuing networks are solved for the average occupancy and average throughput of queues,

but they take a statistical approach to encoding queue traffic and are not as well suited for investigating effects concerning the precise timings of certain events.

Petri nets are in theory capable of modelling these timing effects in detail. Performance evaluation of a Petri net can be done using either simulation of the Petri net itself or by decomposing the Petri net into an underlying queuing network model, but packages which implement the simulation method [GKZH95, CFRR95] are currently too slow for use on accurate models of large multicomputer interconnection networks, and our goal is to be able to evaluate networks of hundreds or indeed thousands of nodes. Using an analytic solution via queuing networks would involve the same sacrifice of timing precision as if we were to use queuing networks directly as our model. We will see in later chapters that certain details of timing synchronisation between processes or of packet ordering properties in the communications network can produce large effects on overall performance, and we want to be able to capture and study such effects.

Hence, discrete event simulation is our chosen performance evaluation technique. This is a technique which has been much used in the past for studying wormhole routing networks ([Chi95, KE95]...) and contention ([Lee85, KP86, LK90]...). The MIMD [Gui88, GCS89] modelling system, built from Simula [Poo87, Bir86] was already in use at Edinburgh when this project started, but suffered from performance problems: in previous work using MIMD grid-topology simulations, 4 by 4 was a common size for the processor grid and the upper limit was about 8 by 8 processors. To achieve our required network scales of hundreds of nodes, we chose to implement our own dedicated simulator for wormhole routed networks.

3.5 A Discrete Event Simulator for Wormhole Routed Interconnect Networks

In this section we will outline the design of the simulator we constructed for performance evaluation of wormhole-routed multicomputer networks. We will describe the design of the communications model as well as the overall way in which the software was built.

First of all, we decided to construct the simulator in C++. This gave us the performance advantages of dedicated compiled code, without sacrificing the object-oriented software approach which Simula proved so appropriate for simulation design. Portability was another main design goal which suggested the use of C++: much of the development work (and indeed the experimental work) was

performed on PCs running the Linux operating system, and the use of the *g++* C++ compiler has allowed our code to run without modifications on Sparc workstations running either Solaris or SunOS and on Alpha and PC systems running Linux.

The simulator core is purely event-driven, without any of the threading features of Simula. Generic C++ objects are used to represent most of the interesting "things" in a multicomputer, including processors, processes, packets and routers. Programming the simulator is done at compile-time, not at run-time—again, this was a deliberate decision aimed at improving performance.

Programming is generally performed by creating a specialised class derived from some core base class in the simulator, and defining functions within that class to modify the behaviour of the object. For example, to program a process with a given behaviour, one would create a class derived from the parent "Process" class and define various member functions to specify that process's behaviour on events such as "completed the last request to compute" or "request to receive a packet has completed".

The simulator has a highly modular design which makes much use of the C++ language's features for object inheritance. The design makes it simple for specific components of the simulator to be removed and replaced with alternative versions. For example, this feature is used to allow the simulator to support many different implementations of the transputer communications architecture: a compiled simulation can be built against any one of these different implementations simply linking the existing object files against a different communication library. In the rest of this chapter we will see several places where the modular design allows the user to pick and choose between different underlying implementations when building a simulation experiment.

3.5.1 Components of the Simulator

In the sections which follow, we will describe the major components of the simulator. These broadly fall into three categories:

Core system At the heart of the simulator sits a central event loop which handles all timing and event dispatching for all components of the system. The simulator core also includes various general-purpose components designed for use by other parts of the system, including flexible random number generation and results logging facilities.

Computation support There is a fairly simple implementation of the T9000

processor and the user process, sufficient to allow construction of simulation workloads fitting the models above.

Communication support This is by far the largest and most complex part of the simulator. The communications support includes a number of high-level C++ objects which encapsulate key concepts such as messages and packets, routers and communication links. Separate to this, we provide a number of distinct low-level objects which implement the communications architecture in different ways.

3.5.2 The Simulator Core

The Primary Event Loop The simulator core provides a single place where all of the setup, running and termination of a simulation run takes place. The core simulator event routines are not simply functions provided for the user to call from the customised program code when an experiment is run: rather, the simulator provides a complete program skeleton which makes calls out to setup routines which the user can customise to create the multicomputer objects and initialise the workload objects when a simulation begins.

These core functions track the progress of simulated time in the multicomputer. They provide a mechanism by which simulated events may be queued for dispatch at some future simulation time. This simulated clock runs at nanosecond resolution, but the core event loop does not have to run for every clock tick: we simply dispatch the event with the soonest expiration timestamp at each iteration of the event loop.

Random Number Generation Random number generation is an important function within the simulator. We provide functions by which random numbers of various distributions may be obtained. Most of these functions use the underlying random number distributions provided by the *libg++* class library supplied with the GNU C++ compiler.

One feature of the random number support is particularly worth mentioning. We support the use of several distinct random number streams, and these separate streams may be initialised individually with distinct starting seeds. This allows different components of the simulator to use different but repeatable random number streams.

For example, the setup routines supplied by the user to initialise the simulated workload may use a different random number stream from the routines which control the communications layers, so that we can repeat an experiment with

a different communication implementation (which may use its random stream in a different order) without changing the workload when we run the second experiment.

Simulator Instrumentation We have provided our simulator with a logging engine to automatically log summaries of any simulation measurements recorded by any other components of the simulator. All of these results are written in a standard, machine-readable format to a log file after each simulation run. Just as importantly, we have integrated logging of all the input parameters to any simulation run into the same log, so that for every simulation run we record precisely the conditions set up for that run.

For each variable x observed in the simulation, we record the values N (number of samples), $\sum x$, $\sum x^2$ and $\sum x^3$, as well as the minimum and maximum sample. These recordings are enough to calculate the mean, variance, standard error and skew for the distribution of each variable. By storing the raw data as $\sum x \ldots \sum x^3$ rather than preprocessing them into their statistical summaries, we are able to combine variables together at a later date and still accurately determine the distribution of the new variable. For example, we record separately the delivery times of data and acknowledgement packets, but we can still combine these later to list the distribution of delivery times over all packets if we choose.

Builtin log variable types include a basic integer log, logging samples of integer variables; time-average logs, which record the average value over time of any variable whose value changes at distinct points in time (for example, the occupancy of a buffer or the utilisation of a resource); and the time-average-set, which records the distribution of time-average values for a set of resources. For example, each CPU in a simulation will have its own utilisation log, but the final log recorded will be the distribution of the utilisations over all processors in the network.

3.5.3 The Process and Processor models

As we do not need to model computation in a great deal of detail for the purposes of our study, both our process and processor models are simple. The computation side of the processor has very little functionality: all it can do is maintain a run-queue of runnable processes, which it schedules on a first-come, first-served basis. Time-sharing is supported by allowing the processor to reschedule a process to the tail of the run-queue after it has consumed its scheduling time quantum.

The process model is not much more complex. Processes are connected to-

gether by "sockets" (analogous to UNIX sockets). A socket is merely a two-ended pipe which represents a potential virtual link within the multicomputer network. Every socket is associated with a sending process, a receiving process or both; if one endpoint is not connected then any process may act as that endpoint, allowing a process to listen from messages from any other process or to send messages to any process. (This is an extension to the transputer communications model which allows us to code complex communication interactions more easily without having to create virtual links for all possible sender/receiver pairs in advance.)

As described above, the user programs the simulator by creating a derived process class which defines the process's response to certain events, so our model of the process is necessarily event-oriented. A process can respond to the following events:

Initialisation

Send completed Passes in the sending socket

Receive completed Passes in the receiving socket and number of bytes received

Compute or sleep completed

and must take one of the following actions following the event:

Sleep(c) c is the time to sleep for (in clock cycles)

Compute(c) c is the time to compute for (in clock cycles)

Send(s, n) s is the socket to send on; send n bytes

Receive() Wait for a packet to arrive on any socket

In addition, send() actions may be either blocking or non-blocking. The only difference between the Sleep() and Compute() actions are that Compute()s grant exclusive use of the CPU if there are multiple processes running on the same processor.

3.5.4 The Communication Subsystem

We now describe the communication components of our simulator. As we mentioned before, modularity is an importing feature of their implementation, so we will describe here both the overall design of the communication subsystem and the various implementations available to model packet transmission.

The Simulator Communications Design The base C++ classes which make up the simulator's communications architecture are designed to allow the user to manipulate a multicomputer network topology without knowing any details of the implementation of the communications between the nodes in that network. There are two sets of top-level abstract objects which implement this: first of all, a set of communications objects which tie communicating processes together, and secondly a set of multicomputer objects which bind processors and routers together into a network with a given topology.

Inter-process communication is implemented by a set of classes including:

Socket A Socket is the object which processes use to communicate over. All transmit and receive traffic is passed through a Socket object. The Socket may implement transmit or receive buffering on behalf of the process. The Socket is an abstract class which does not actually implement any sending or receiving methods itself.

SendSocket A SendSocket is a Socket which implements transmission routines. There are two main implementations of SendSocket provided: one which sends all transmitted data directly to the recipient Socket (allowing a parallel workload to be run without modeling any underlying communications hardware), and a second which injects all non-local transmissions into the local T9000's data queues.

ReceiveSocket A number of receiving socket classes are also implemented. The SimpleSocket implements reception of packets, either from a local SendSocket (if the communication is within a single processor) or from a router object (if the communication is non-local). This socket implementation does not buffer any data above the single packet of buffering required in the T9000 architecture: if there is not currently a process waiting to receive an incoming packet, then the transmission of an initial acknowledgement to that packet is deferred until some process becomes ready to accept the data.

There are several other receiving Socket classes derived from the SimpleSocket, and they implement various amounts of buffering to allow data to be received asynchronously even before any process is ready to accept it.

To select the appropriate buffering semantics for use in any given simulation run, C++ multiple inheritance is used: the user simply declares a class which is derived both from the required SendSocket and ReceiveSocket classes, and the

shared base Socket class will provide the top-level primitives required to access the resulting hybrid object.

We also have a number of objects which represent communication between processors, not between processes:

Router A Router object represents any device which can accept incoming packets and do something appropriate with them to ensure their delivery.

DSLink A DS-Link communication channel is managed by three types of object: a DSPacket, an extension which adds various performance monitoring features to the basic Packet class; the DSILink, which acts as a packet source to a Router object; and the DSOLink, which transmits packets out from a Router object. The DSILink and DSOLink types are both abstract base classes: they do not by themselves implement the mechanism by which a packet gets from one output link to the next input link. They do however form a framework by which a network topology can be constructed and within which a Router can perform its routing and buffering operations.

Node A Node object simply represents a computing element in the network. It contains both a Processor object for scheduling workload processes and a Router object for passing packets between the workload and the network. The T9000 class is a subtype of class Node, and T9000 objects are responsible for splitting long messages into multiple packets and for generating acknowledgement packets in response to received data.

Using these Router, Node and DSLink types, a multicomputer interconnection network can be laid out without any knowledge of the mechanisms being used to actually transfer data between the Routers. Two built-in layouts are supported by the simulator libraries: the 2-torus (of which the 2-grid is a special case) and the hypercube.

The layout libraries provide general purpose routing functions which describe, for any pair of Routers (current, destination), a list of all DSLinks from the current Router which are profitable for the given destination. The geometry part of the routing algorithm is thus separated out from the decision of which profitable channel to use for routing. Using this basic routing function, we can implement both greedy dimension-order routing and adaptive routers such as the chaos router.

The Simulator Communications Implementations The communications framework we have described allows us to implement multiple different commu-

nications infrastructures inside our simulator. The user can select which of these infrastructures to use simply by linking against the appropriate communications library. The C++ type system ensures that no recompilation is necessary: all of the different classes implemented by the different communications models are derived from the same set of base classes and thus look the same to the rest of the software.

Why do we have multiple implementations of the communications subsystem? There are two main reasons:

- For performance reasons, we provide simplified but more efficient communications implementations. In fact, the simulator contains three distinct implementations of the communication model. Each implementation is successively more simple (and therefore more efficient to execute) than the previous. In the absence of a real machine against which to compare it, the most costly but most accurate implementation will be a valuable tool by which to validate the simulator.

- We have the ability to selectively disable elements of the run-time cost of the system. Specifically, we can run our workloads in a contention-free environment in which point-to-point communication delays are modelled accurately, but network links are all assumed to be of infinite width; and we can re-run the workload with no communication costs at all, to get an indication of the true amount of parallelism inherent in the chosen workload mapping. This facility will be crucial to our quantification of contention costs in chapters 5 and 6.

The simulator fully implements T9000 message decomposition into multiple data and acknowledgement packets. The C104 architecture of input and output links connected via a central crossbar is also modelled accurately. However, detailed modelling of the low-level DS-Link communication involves transmitting large numbers of tokens for each packet sent over each hop in the multicomputer network. It has been our aim to simplify this level of DS-Link communication as much as possible without losing the realistic behaviour of the simulator as a whole: dispatching separate simulator events for every single token of a packet can be rather expensive.

To this end, we present three distinct models of DS-Link communication:

- **Token-passing:** we model the real-life DS-Link accurately by transferring individual data and control tokens over each link. This provides us with a

reference model against which to compare the other models, but is expensive to simulate: a separate simulation event must be dispatched for each token's arrival over the link.

- **Stream-passing:** the passage of a packet over a DS-Link is modelled as a continuous, undifferentiated data stream. A packet is simply a period of activity on a link: the length of the packet determines the necessary period of activity. Flow control is implemented on demand. Rather than requiring the receiver to explicitly grant permission to send data, we allow data to flow freely until explicitly blocked by out-of-band *block* and *unblock* signals on the link. In the absence of blocking, the simulator need generate no events other than the start and end of the packet at each hop.

  Under simulation, this model of communication does execute substantially faster than the token-passing model on lightly loaded networks. However, the requirement of out-of-band signalling to propagate flow control messages upstream enormously increases the cost of flow control transitions, and we found in practice that under moderate to high network load, this model executed no more quickly than the token-passing model. Since the only reason to simplify the model is to improve simulation performance, we decided not to pursue this model any further.

- **Packet-passing:** packets are still undifferentiated data streams, but now all flow control is performed at the packet level, rather at the data-stream level. Once a packet is sent, the sender will not send another packet until the receiver has sufficient buffering space left to receive it in its entirety.

  This has a major implication for flow control behaviour. In effect, we can only send a flow control token at the start of a packet, giving permission to send an entire packet at once. This may leave packets pending in an upstream buffer for longer than the realistic, token-passing model would. We investigate this effect in the following chapter.

In implementation, these three schemes are progressively more efficient to run, but represent progressively more simplified views of the complexities of flow control synchronisation occuring within the multicomputer network. In chapter 4, we will justify the use of the more efficient but more abstract models of T9000 communication.

Note that we completely ignore the effects of latency delays within DS-Links and over communication wires themselves. In practice, these delays may be negligible when compared to the routing latency delay incurred within a C104 routing

chip, so this is not important. (Accurate data on the precise number of clock cycles taken by the C104 to perform its routing function was not available when construction on the simulator began.) Furthermore, buffering within the routing network allows network throughput to remain largely unaffected by small changes in such low-level latency characteristics [BGT93, pp.91–94].

In our model, we also assume that an acknowledgement packet can be generated by a T9000 immediately upon receipt of an incoming packet header, ignoring any realistic processing delays. The simulator model is capable of modelling these delays, but an investigation of their effects is beyond this particular study. In practice, this effect only has an impact on bandwidth when the DS-Link is carrying a single virtual link's traffic, since the latency is hidden in the presence of multiple senders [BGT93, pp.94–95].

Also note that the two less-detailed DS-Link models perform flow control by control mechanisms within the simulator which do not correspond to real DS-Link behaviour. These two models do not try to model the realistic negotiation of flow control by low-level token passing, and so are not capable of exhibiting the real DS-Link's difference between uni- and bi-directional data throughput capacities.

3.5.5 Simulator Performance

Our objective here has been to design an efficient and flexible discrete event simulator. Have we succeeded? Before we leave our discussion of the simulator tool, we need to give some idea of how rapidly it can run relatively large experiments. We also need to look at how much memory it consumes, as that can easily become a limiting factor when running large simulations.

We can demonstrate performance by running a synthetic workload on a fairly modest machine by today's standards—a PC running Linux, with a 200MHz Pentium-MMX processor. We will define the workload more fully in chapter 5.1, but to summarise it, we have a grid of 32 by 32 T9000s and C104s, with one process on each processor sending messages randomly to other processors on the network. Each process waits 1 microsecond between sending each message. The performance of the simulator over a run of 20 simulated milliseconds for each communications model is show in table 3.1.

On this simulation run over 1024 processors, a total of 844,342 packets were sent through the network, over an average path of 16.6 hops, or a total of over 14,000,000 individual packet forwarding operations. The entire simulation completed in a little under 12 minutes for our simplified, efficient communications model, in contrast to nearly an hour for the laborious token-passing implementa-

Communications model	Events dispatched	Elapsed time (mins)	Memory consumed (kB)
Token-passing	394975769	54:59	19608
Packet-stream	75927848	15:40	24604
Packet-passing	40564755	11:54	17300

Table 3.1: Performance of Each Simulator Communication Model

tion which models each transmitted token individually. For the efficient model, the simulator was dispatching over 55,000 internal events every second.

Memory consumption was quite acceptable: a simulation of 1024 processors occupied only around 17 or 18MB of memory (rising to nearly 25MB for the most complex communications model). The resources consumed by our simplified communications model are quite modest enough to allow us to run experiments involving many tens of such simulation runs within a reasonable timescale.

3.6 Experimental Methodology

So far, we have described only our basic simulation tool. However, running a single simulation under a single set of conditions is not enough to provide any meaningful data: we really need to have a framework in which we can make multiple simulation runs and extract useful trends from their results.

In this next section we will describe some of the methods and tools which we use to create complete experiments out of the raw simulator tool. We will look both at the front end—the code and techniques we use to run and control the simulator over the course of a full experiment—and the back end—the tools used to extract results from the data accumulated over that experiment.

3.6.1 Controlling the Simulator

There are three issues we need to look at when moving from running single simulations to setting up and executing entire experiments:

- First of all, we need a mechanism by which we can automate the repeated running of the simulation core with all of the parameters set correctly for each run within the experiment;
- We have to be able to record the results of these experiments in a way which will allow us to extract useful data later; and

- We need some form of quality control, both to ensure that the simulation experiments are run for an appropriate length of time and to provide a numerical confidence interval in our results once the experiment completes.

3.6.1.1 The Experiment Support Tools

To address the requirement for automated running of multiple simulations, we implemented a front-end control tool, "`front`", for controlling entire experiments. This tool is a script-driven program which can run a pre-compiled simulator instance multiple times and assemble the results together into a single machine-readable log file for the experiment.

Option handling Central to the operation of the `front` program is the uniform way in which options are passed to the simulator program. One of the C++ classes provided by the simulator is a complete option-passing class. The simulator is a command line program, and as such it can accept a number of options on the command line when it is run, and the option-passing classes take care of this automatically: declaring a new variable of Option type transparently tells the simulator to look for the appropriate named option on the command line during startup and to use that to initialise the variable. The values of all such options are automatically recorded in the simulator log file at the end of the run, so we have a record of which parameters were set for each simulation run during the experiment.

The script file allows for parameters to be set to single values, ranges of values or enumerated sets of values. It also allows the value of one parameter to be computed based on the value of another parameter. The regular parsing of options by the simulator allows the front-end tool to pass these arbitrary parameters to each simulator run on the command line. All the user has to do is to record, in the script file, all the parameters which must be set for the experiment, and the ranges of values which they must take. The front tool will then run the simulator multiple times, once for every possible combination of parameter values from the script file.

"Backend" Mode The normal operation of the simulator is to output all of its results to the screen in human-readable format at the end of each run, and to append both the results and the values of all simulator options in machine-readable format to a named log file. However, the front-end tool uses a separate, back-end mode of the simulator, in which it produces no human-readable output

but simply prints out all of its results in the machine-readable format directly. `front` then appends these results to its own log file, also recording information which delimits the start and end of entire experiments, and time-stamping the experiment as a whole.

Parallel Execution One final feature of `front` is its ability to run multiple executions of the simulator at once. Remote execution is also supported, so that an experiment may run over multiple machines simultaneously. The number of processes we may launch on each machine, and the maximum load we allow each machine to reach, can be specified in the script file. This raises the possibility that different executions of the simulator will complete out-of-order, but `front` simply buffers the output of each simulation run until all previously launched simulations have completed and have been appended in correct order to the log file.

Extracting Experimental Results The second auxiliary tool we use to deal with entire experiments is the "`extract`" program for parsing the machine-readable log files created by `front`. `extract` reads the log file simulation by simulation, and outputs one line of ASCII, space-separated numbers for each set of data recorded. This output is suitable for post-processing by `awk` or `gnuplot` if further manipulation of the data or plotting of the results is required. The fields which are output can be selected by command-line options to `extract`. Fields may also be output with error-bars if desired.

Sometimes, however, we do not want to have one line of output for each separate simulation run. This is particularly true if we have rerun a given simulation multiple times, varying between each run only parameters which are of no interest to the experiment as a whole. For example, a simulation might be repeated with different initial values of the random number seeds, to increase the accuracy of the results obtained. `extract` automatically merges together simulation runs which were run with the same input parameters, and it can optionally also merge runs which differ only in such non-interesting parameters. This allows us to repeat experiments to improve our confidence intervals without having to perform any tedious reprocessing of the data already gathered.

3.6.2 Simulator Stability

One other issue we need to pay close attention to when building entire experiments from our simulator is the question of the reliability of our results. To obtain

sufficiently accurate results, we do need to make sure that the simulation is run for a sufficiently long time, but we do not want to have to run it for any longer than necessary.

A standard statistical method used to indicate the accuracy of a result is the "confidence interval" [FW87], which we can use to determine how much confidence we have that our measurements of some variable in the simulation have become sufficiently accurate. We can observe our ever-improving confidence intervals on some privileged metric at regular intervals to determine at what point our observations are sufficiently accurate.

There is a second, slightly less obvious issue which also arises. A common problem when running simulations of any system is that the behaviour of the system may change significantly over time. In this study, we are really interested in the equilibrium state of an application running on some multicomputer, since it is this equilibrium which determines our long-term performance. Even when it does reach an obvious equilibrium state, the behaviour during the startup transient period may be markedly different to the equilibrium behaviour.

If we want to make efficient use of simulation run time, it may well be important to identify the transient startup period and to discard observations taken in that period. By taking simple steps to eliminate this obviously inaccurate data, we can improve the accuracy with which we measure the equilibrium state, given a fixed run time; or conversely, we can reduce the run time required to achieve a predetermined accuracy in our equilibrium observations.

The way we address this is by adopting some (quite arbitrary) measure of performance, such as the number of computation steps completed per given simulated time interval, as an indicator of system activity. We observe this measure over time, sampling it at regular intervals, and once it stabilises we assert that the startup transient period has completed.

This has an effect on the way in which we gather observations about other variables in the network. At any point in time, we may decide that we know when the transient completed. However, we may conclude that the transient actually ended at some point in the deep past in simulated time. We also want to throw out any samples we made of logged simulator variables during the transient period, but we don't know in advance just how many such samples we will need to discard. To get around this, we checkpoint our samples of each recorded variable at regular intervals, and we keep all of these checkpoints stored separately. Once the length of the initial transient is known, we simply discard all observation checkpoints which overlap with that transient period.

3.6.2.1 Periodicity and the sampling rate

It is also possible that the system will behave in a cyclic manner if observed at sufficiently short intervals. If there is indeed a bottleneck, then events at that point may radiate brief storms of activity outwards; and the applications themselves may have a periodic nature.

All of these factors have implications when we consider how long to run our simulations for. However, this is dealt with in [Jai91, pp.423–433], with the method of "batch means". This method relies on on recombining existing, fine-grained observations to allow us to choose an appropriate sampling interval after the samples have been taken. The number of samples combined together in this way at once — the batch size — is chosen so as to minimise the time-dependency and periodicity in the observations.

Chapter 4

Validating the Simulator

In earlier chapters we have presented a model of a multicomputer network and have discussed a simulator tool for studying the behaviour of this model. We can now move on to examine the results obtained with this toolset.

In this chapter, we will not be interested in studying contention in any detail. What we need to do here is to establish some groundwork for our later experiments on contention performance.

It is possible to make many distinctions between different types of effect which occur in a complex multicomputer system. It will be useful to consider one such distinction here—the distinction between the local, *microscopic* properties of the system, and the global, or *macroscopic* properties. Note that this distinction is entirely arbitrary and is only being used to organise our results.

By "macroscopic", we refer to those properties which emerge only when we consider a number of routing components joined together to form a network, and we are interested in the interactions between different messages as they travel their separate paths through the network. At this level of detail, we are explicitly looking at overall network throughput, as measured (say) by average channel utilisation or by the distribution of end-to-end message throughput and latency. In this study, we look upon different routing strategies and their effects as macroscopic features of the system.

By "microscopic", we refer to the properties of the system visible at a smaller scale. We still consider bandwidth and latency measurements, but we look at the performance of individual components in the system. We do still consider interactions between colliding packets, but in this case we are interested in the local details of how we buffer and prioritise these packets, not of how we route them forward through the network as a whole. Of course, the microscopic properties will affect the overall performance of our networks, but the effects we class as microscopic are visible even when we consider the simple case of communication

between two adjacent routers or CPU elements.

This chapter will be concentrating purely on the microscopic properties of the simulator. We will be explicitly avoiding looking at overall contention performance. Rather, our aims will be to:

- Assert the correctness of the simulator output, including:
  - verification of the simulator components;
  - validation of the simplified communication model described in chapter 3, by comparison with the more accurate but more inefficient forms.
- Study the performance impact of certain specific microscopic properties, involving:
  - the exploration of the model's performance by studying the effects of changes in purely local properties of the communication system. For example, we can control the local queuing strategies employed at each node of the network, the buffer sizes, and the width of each interprocessor communication link.
  - enabling the creation of more complex experiments by establishing suitable values for these local properties to be fixed when we start to consider more global properties of the system in the next section.

  As a side effect of this work we will identify certain interesting performance relationships, but a more important goal of this section will be to find and eliminate parameters of the architecture which are uninteresting, and which do not substantially affect the performance trends that we observe under load. This information will be particularly useful as a preliminary to the set of more complex experiments presented in chapter 5.

4.1 Validation of the Transputer Implementations

In this section we will address the issue of testing the correctness of our simulator. We can divide this into two major tasks:

- *Verification* of the individual simulator components; and
- *Validation* of both the simulator as a whole, and of the simplifications made in our efficient communication model.

Our task is first of all to ensure that our simulated components — primarily the T9000 cpu and the C104 router — are correct, by measuring the performance between these components in simple scenarios where the expected performance (bandwidth and latency) can be precisely calculated.

We then move to examine more complex systems, combining larger numbers of components into examples of realistic multicomputer configurations. Our aim here will be to show that the simplifications we have made in our efficient communication implementation (the *packet-passing* model) do not substantially alter the properties of the system, and that we can use this model to investigate large multicomputers without losing accuracy. Of course, some differences between the *token-passing* and *packet-passing* models are expected, but we aim to understand those differences. In particular, we wish to show that the difference between the observed performance of the two systems can be explained in terms of the known differences between the implementations. Only with this reassurance can we be reasonably certain that in scaling our simplified implementation up to large scales we do not introduce unrealistic artificial effects into our results later on.

4.1.1 Low level verification of the individual components

As described in chapter 3, our communication model includes a network of interconnected wormhole routing devices with a single CPU device connected by a dedicated channel to each router. In the simulator, we model the router and CPU devices on the C104 and T9000 members of the Transputer family.

In this model, the T9000 devices always communicate with each other through two or more intermediate C104 routers. However, the implementation of these components in the simulator also allows us to create networks in which T9000s are connected together directly. We can exploit this in our verification of the simulator components to test the T9000 implementation both separately and in combination with the C104 implementation.

We also have the advantage of Inmos's own published performance models ([BGT93]) against which we can test our components.

4.1.2 Validating the T9000 implementation

In our model of the T9000, we are ignoring the only significant latency present in communication between two adjacent T9000s (the acknowledgement processing latency described in section 3.5.4 on page 38). The only significant sources of latency present in our model are

- Competition for CPU processing time between processes;
- Competition for insertion into the output channel of a T9000 or C104; and
- Forwarding delay as the C104 router cannot start to forward an incoming packet until its header has been received, and similarly the T9000 cannot generate an acknowledgement packet for a message until it receives the header of the final packet for that message.

We first of all consider the T9000 component, for now ignoring the C104 router. Without routing support, the T9000 cannot forward any incoming data, but it can still perform useful communication with neighbouring T9000s. Since no forwarding is possible, we can ignore the router forwarding latency entirely.

It is possible for us to consider the CPU and communication competition effects separately. Our process model allows us to simulate communication-only processes which impose no CPU load at all, so that we can concentrate on observing only the behaviour of the communication network. Likewise, we can generate compute-only processes which allow us to test the time-sharing properties of the simulated T9000.

4.1.3 The T9000 CPU

The computation model we employ is extremely basic. Processes may consume CPU time on demand, but we do not implement any hidden, implicit CPU load. A genuine T9000 will incur extra CPU latency on scheduling interrupt or on process wakeup due to an incoming message, but since the T9000 has a built-in scheduler and is designed for fast context-switching, these costs are small, and our simplified model of computation ignores them entirely.

Our CPU implementation is a straight-forward FIFO scheduler with the ability to perform time-sharing context switching. A simulator Process object may request periods of computation, and the Processor implementation notifies the Process when that period has expired. The Process may then either block (sleeping for a fixed real time period or waiting for IO), or continue to run by requesting a further computation period.

An optional time-sharing feature in our Processor implementation allows the Processor to move a running Process to the back of the run-queue after a certain fixed interval (measured from the time this process reached the head of the run-queue). This feature will be rarely used, however, since in typical multicomputer environments, if we have processes running with long communication-free periods

(relative to the scheduling interval), then we tend to have only one process running per processor anyway.

As a brief check that our CPU model is fair, we run a number of extremely simple processes in parallel on a single processor. Each process simply loops infinitely around the operation

```
compute(period)
```

We observe both the total work done per ms, and the number of cycles completed by each separate process per ms, in figure 4.1 (a *period* of 1000ns is used in this figure). As we expect, the total work completed is constant as we change the number of processes, and represents a perfect 100% cpu load. Fairness is reflected in the fact that for any given run, all processes complete precisely the same amount of work per ms. If we repeat the test with a larger or smaller compute period (testing values substantially smaller or greater than, or close to, the scheduler's constant timeslice interval), the results show similarly consistent behaviour.

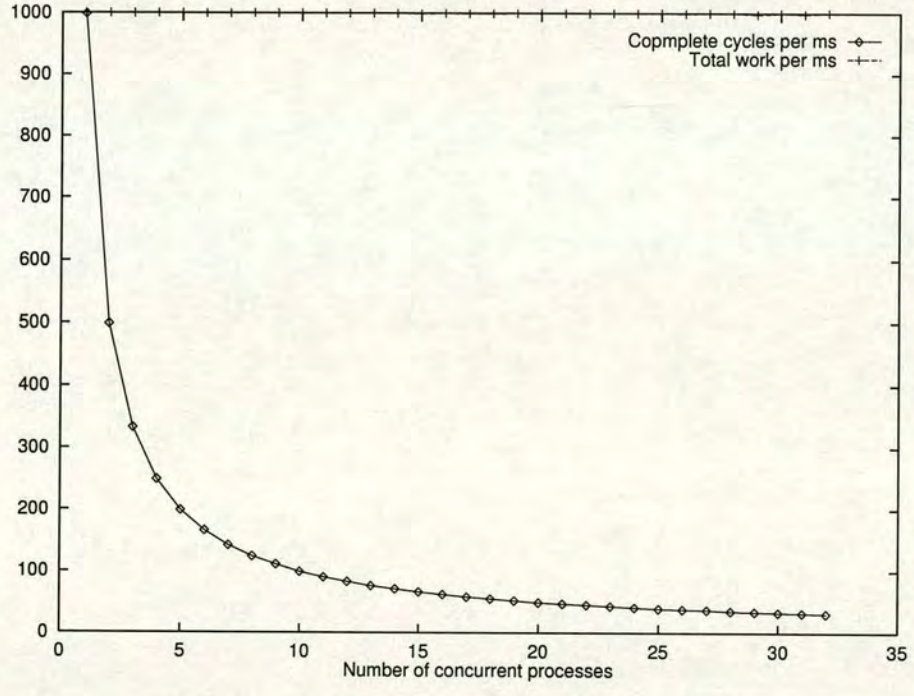


Figure 4.1: Compute-bound throughput on a single T9000

4.1.4 T9000 Communication Properties

We next look at our implementation of the T9000's communication facilities. There are essentially three separate properties we have to consider: throughput

and bandwidth, properties common to all communicating systems, and the queuing of packets when more than one message is being sent down a single physical DS-Link.

4.1.4.1 T9000 Bandwidth

Several factors reduce the bandwidth available to messages crossing a raw DS-Link, quite apart from any delays in queueing incurred while waiting to enter the DS-Link. The best-case scenario for raw bandwidth is of unobstructed one-way communication between processes in adjacent processors. Even in this case, the full theoretical bandwidth of the DS-Link bit-stream will not be realised. The primary extra cost is the packetisation overhead due to extra packet termination and header bytes in the data stream, but flow control tokens and acknowledgement packets will also drop the bandwidth available for data bytes.

In a DS-Link carrying data in only one direction, these effects will be small, but present. The outgoing and incoming data streams are almost independent, so that the incoming acknowledgement packets arrive independently of the outgoing data packets. There is a small hidden cost to the outgoing data stream, however — flow control packets on behalf of the incoming data stream must be interleaved with the outgoing data.

Bi-directional data is significantly more expensive to carry. Each direction must carry not only data, but also acknowledgement packets; furthermore, it must carry flow control tokens not only for returning acknowledgements, but also for the opposing data stream.

May, Thompson and Welch [BGT93] present models of this ideal uni-directional and bi-directional bandwidth, parameterised by message size m and header size s. In this chapter we ignore the variable header size, and assume that $s = 1$ in all cases, although in real life larger headers will be used to address random-routed destinations and/or machines with more than 256 processors.

Given a message of length m bytes, and given that the T9000 breaks up messages into packets of at most 32 bytes, we require n_p packets to carry each message, where

$$n_p = \left\lceil \frac{m}{32} \right\rceil$$

Since there is a per-packet overhead of 10 bits per header byte, plus a 4-bit end-of-packet token, or 14 bits, the number of bits required to represent the entire message is

$$b_d = 10m + 14n_p$$

There must also be an incoming acknowledgement packet for each data packet transmitted. The acknowledgement is represented by an empty message, so for our implementation, 14 bits comprising two tokens are required for an ack, the number of acknowledgement bits required for the message is

$$n_{dt} = 2n_p$$

Since one flow control token is sent for every eight received tokens, the outgoing link must carry

$$\begin{aligned} n_{ft} &= \frac{n_{dt}}{8} \\ &= \frac{n_p}{4} \end{aligned}$$

flow control tokens per message. Each token being 4 bits long, so the total number of outbound bits required per message is

$$\begin{aligned} B &= b_d + 4n_{ft} \\ &= b_d + n_p \end{aligned}$$

The number of actual data bits transferred in a m byte message is $8m$, so the total uni-directional bandwidth available to the CPU over a DS-link is

$$D = \frac{8m}{B} \times 100 \text{ Mbits/sec} \tag{4.1}$$

We can apply similar calculations to the bi-directional case, where there is a single sending process and a single receiving process at each end of the DS-link. As before,

$$n_p = \left\lceil \frac{m}{32} \right\rceil$$

but our data rate is different because outgoing data packets must be interleaved with acknowledgement packets for the incoming data stream. Since the link is symmetrical, we consider just the outgoing data stream. For each outgoing message, we must transmit:

- n_p data packets, including $10s + 4$ bits of header and trailing information;
- One acknowledgement packet of $10s + 4$ bits;
- Flow control tokens for the incoming data and acknowledgement packets.

The number of non-flow-control bits sent is

$$\begin{aligned} b_d &= (10m + (10s + 4)n_p) + (10s + 4)n_p \\ &= 10m + 28n_p \end{aligned}$$

and the number of tokens required to encode this is

$$\begin{aligned} n_{dt} &= (m + (s+1)n_p) + (s+1)n_p \\ &= m + 4n_p \end{aligned}$$

The incoming data link must also carry n_{dt} data and acknowledgement tokens, so that we must add the necessary flow control tokens to the outgoing data stream:

$$n_{ft} = \frac{n_{dt}}{8}$$

The total number of outgoing data bits per message is then

$$B = b_d + 4n_{ft}$$

and the bandwidth is

$$D = \frac{8m}{B} \times 100 \text{ Mbits/sec} \tag{4.2}$$

We can validate the simulator's T9000 component by comparing against these models. We connected two T9000s together via a single DS-link, and repeatedly sent fixed-size messages from a process on one to a process on the other. For the bi-directional case, we simply created a sender and a receiver on each processor.

In figure 4.2, we see both the bandwidths we observed, and the theoretical bandwidths according to equations 4.1 and 4.2. Link bandwidth (measured in data bytes transmitted per second) is plotted against message size.

Since messages are divided up into 32-byte packets, the total packet overhead per message increases once for every 32 bytes in the message. The corresponding drops in bandwidth are clearly visible. Also obvious is the drop in bandwidth in each direction which occurs when data is being sent in two directions simultaneously.

More significantly, the observed and predicted throughputs are in extremely close agreement. The small discrepancy in the observed bi-directional throughput can be easily explained. The T9000 can queue an acknowledgement packet as soon as only the header of a data packet has been received, and we would expect this to allow a sender to transmit data packets back-to-back.

However, if we start up the sending processes on each processor simultaneously, the outgoing and incoming packets will be transmitted in lock step. This means that the acknowledgement for an incoming data packet cannot be sent until the incoming packet has been completely accepted (because the outgoing link will be in step with the incoming link, and hence will be sending a corresponding data packet itself). Outgoing data therefore stalls while the incoming acknowledgement is received.

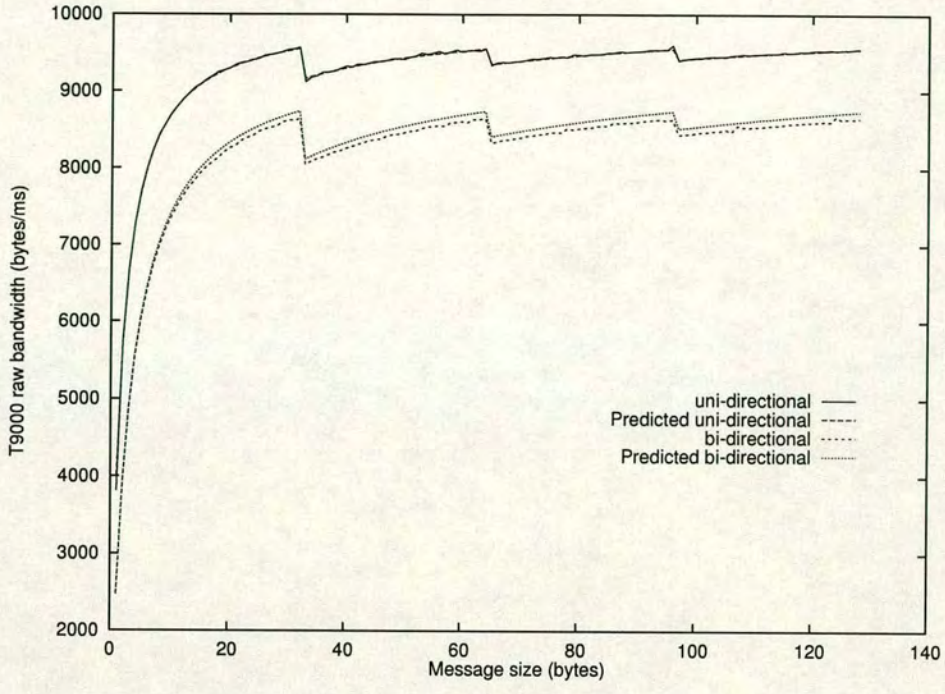


Figure 4.2: Raw T9000 output bandwidth against message length

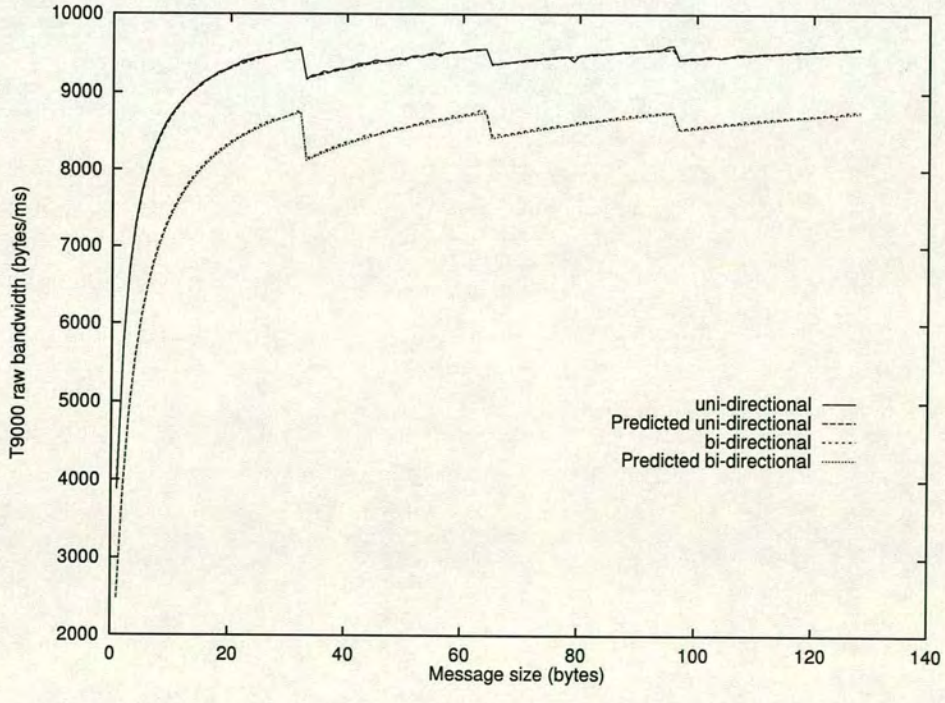


Figure 4.3: T9000 output bandwidth against message length, Staggered process startup

If we stagger the processes so that we start up the sending process on one processor a short interval after we start the other, we observe the behaviour in figure 4.3, and the discrepancy between observed and predicted throughput is entirely eliminated. The residual differences between predicted and observed bandwidths (excluding packets of length less than 4 bytes) can be seen in table 4.1.

		Maximum deviation	Average deviation	Experimental standard error
Lock-step	Uni-directional	0.50%	0.09%	0.14%
Lock-step	bi-directional	1.38%	0.94%	0.14%
Staggered	uni-directional	0.64%	0.14%	0.16%
Staggered	bi-directional	0.79%	0.25%	0.12%

Table 4.1: Deviation between expected and observed T9000 bandwidth

So, accounting for latency problems when the two units work in lock step, we find no difference between observed and expected performance above 0.8%, and a typical difference of only a quarter of one percent for the bi-directional case (and less for the uni-directional case).

4.1.4.2 The Simplified DS-Link Implementation

We recall that our simplified implementation of the T9000 model ignores per-byte flow control, synchronising at the start of each packet only. The actual cost of transmitting flow control data is ignored. We can adjust the predicted throughput of the DS-link to compensate for this by eliminating the flow control tokens from the calculated value of b_d, the bits transmitted per message. We obtain the following models for theoretical uni- and bi-directional throughput.

For uni-directional data transmission:

$$\begin{aligned} B &= b_d \\ &= 10m + 14n_p \end{aligned}$$

and for bi-directional data,

$$\begin{aligned} B &= b_d \\ &= 10m + 28n_p \end{aligned}$$

As before, the total data bandwidth

$$D = \frac{8m}{B} \times 100 \text{ Mbits/sec} \tag{4.3}$$

Plotting the predicted and observed throughputs in figure 4.4, we again see that our implementation's performance is correct. Table 4.2 shows as before the deviation between predicted and observed performance, and the difference is well within experimental error, being less than 0.1% on average.

	Maximum deviation	Average deviation	Experimental standard error
Uni-directional	0.81%	0.09%	0.13%
Bi-directional	0.40%	0.06%	0.13%

Table 4.2: Deviation between expected and observed T9000 bandwidth for the simplified (packet-passing) implementation

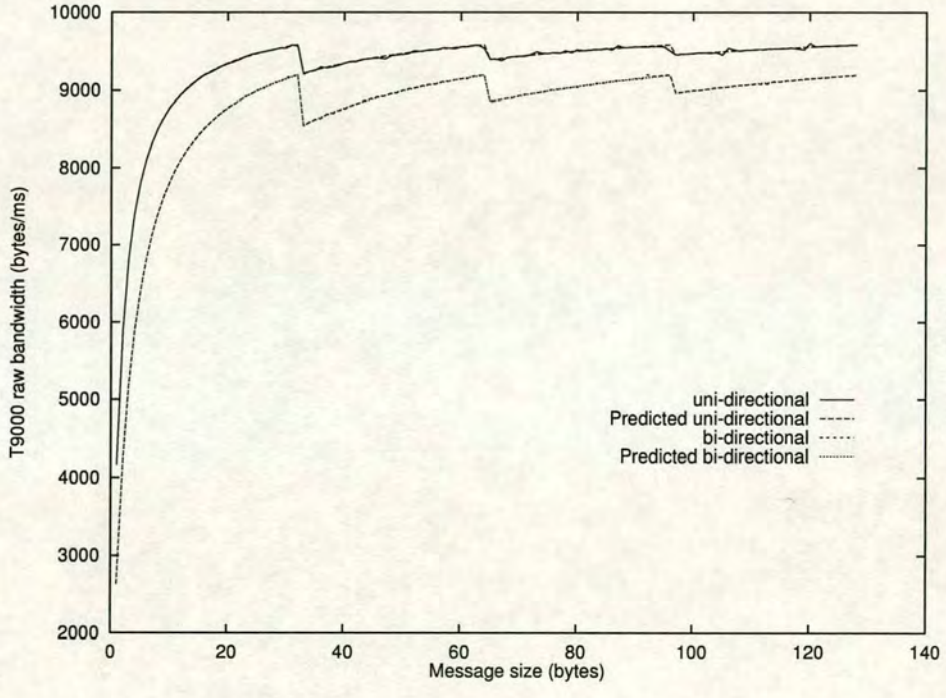


Figure 4.4: Packet-passing T9000 output bandwidth against message length

4.1.5 Validation of C104 Routing Network Models

The introduction of C104 routers into a T9000 network introduces two new timing effects which do not affect communication between adjacent T9000s. First of all there is the routing latency experienced by packets as they pass through a C104. Secondly, there is the contention for output links within the crossbar of a C104.

We considered again the timing experiments from section 4.1.4.1, but this time the two T9000s were not communicating directly but were connected indirectly

through a single C104 routing switch. We can compare both the throughput (number of messages transmitted in a given time) and transit time (time elapsed between initial transmission of a message and final receipt by the destination) between two T9000s with and without an intervening C104, for various message lengths.

Message Length	**Throughput**		**Transit time(ns)**	
	Direct	C104	Direct	C104
0	2667	1482	212	362
1	2623	1468	312	462
2	2424	1454	412	574
3	1951	1454	512	674
4	1632	1428	612	775
5	1403	1403	712	863
6	1230	1230	812	962

We can see that although the time taken for a message to pass from source to destination was consistently and uniformly longer for the C104-connected transputers than for the directly-connected ones, the total throughput was identical as long as the packets were sufficiently long. This pattern is repeated for larger message lengths when the last packet of a message is small enough.

The differences in message transit times are a direct result of the extra routing latency incurred through the C104, and results indicate that this latency is precisely that expected. The throughput of the system is largely unaffected by this latency; only for small packets is the handshaking between the two T9000's delayed sufficiently by the C104 to cause extra interruptions in the flow of data between the processors. If two processes attempt to send data simultaneously over two virtual links on the same DS-Link in this environment, the latency due to the C104 is completely hidden by the extra traffic and the presence or absence of the C104 no longer has any effect on the bandwidth.

4.2 Considering Connectivity

We will now progress to study the behaviour of somewhat more complex systems. We have up to now looked only at two or three components connected together in a simple configuration whose expected performance can be easily calculated. We now turn to look at larger systems, where we will have a larger number of independent communicating units in the network.

Before we start, we must carefully define the workload we will be using in the rest of this chapter. We will then look in more detail at some of the design

decisions we have to take when setting up our large-scale routing experiments. In this examination we will be paying particular attention to two things. First, we want to tune some of the parameters of our simulator to avoid introducing artificial bottlenecks into the system. Secondly, we will identify the specific differences between our efficient and our accurate DS-linkcommunication implementations, and show that even for these more complex simulations, we can fully explain the differences between their behaviours in terms of the known differences in their design, giving us confidence that the simplifications have not introduced side effects which would invalidate their use on large-scale simulations.

4.2.1 A Synthetic Workload

In the following sections we will be much more interested in looking at the raw throughput of our simulated multicomputer than at any contention effects. This will be an initial test-drive of the simulator, and we will not yet be trying to draw any conclusions about contention performance. As a result, all that we require of a workload at this stage is the ability to inject a fairly high rate of messages into the network (especially when we start to look at network bottlenecks below).

As the multicomputer architecture for these experiments, we will use a regular grid-connected combination of the basic building blocks (figure 3.1). The synthetic application will also be organised as a regular grid of communicating processes, but torus- rather than grid-connected. This creates a very artificial workload but for now we are still validating the simulator, not drawing performance conclusions about multicomputers in general, so realism is not as important a factor as it will be in subsequent chapters. This topology will serve our purposes well for the current chapter. It can exert a high load on the network (due to the large numbers of packets being exchanged with neighbouring processors) while still generating enough long-distance traffic between the edges of the grid to allow us to probe timing and ordering effects of packet routing.

The workload shall be a simple boundary exchange. Each process shall be supplied with 4 virtual communication channels $s_0 \ldots s_3$, connecting it with each neighbour, so that the channels for a process $p_{(i,j)}$ on an $N \times N$ grid, are

$$
\begin{aligned}
s_0 &= \{p_{(i,j)}, p_{((i+1) \bmod N, j)}\} \\
s_1 &= \{p_{(i,j)}, p_{(i,(j-1) \bmod N)}\} \\
s_2 &= \{p_{(i,j)}, p_{((i-1) \bmod N, j)}\} \\
s_3 &= \{p_{(i,j)}, p_{(i,(j+1) \bmod N)}\}
\end{aligned}
$$

and the process loop is

```
compute(period)
send(s0, m)
send(s1, m)
send(s2, m)
send(s3, m)
recv(s0)
recv(s1)
recv(s2)
recv(s3)
```

For deadlock avoidance, non-blocking sends are used here.

For the purpose of measuring performance, we count one complete iteration of this loop by one process as a single application exchange. To realistically compare the results obtained from different sizes of workload we would hence have to normalise the total application exchange rate by dividing by the number of processes, N^2.

The workload is completely parameterised by $(N, m, period)$. Unless otherwise stated, we will be using 32-byte messages (the maximum size for a single packet) in all the experiments in this chapter.

The other characteristic of the workload we define is the mapping from process to processor. We will employ a processor grid of dimension $N/2$ for our experiments, and will use a mapping function

$$f(p_{(i,j)}) = P_{(i/2,j/2)}$$

so that four adjacent processes are mapped onto each processor.

There is one other factor which we must determine to define our experiments: the network routing strategy. Unless indicated otherwise, we will be using a simple, greedy routing strategy, which routes deterministically, first by column and then by row.

4.2.2 Providing fat links into the network

In section 3.2, we described the basic components which we use to construct our simulated multicomputers: the networks we construct consist of interconnected C104 routers with a single T9000 connected at each node. We also observed that if we are limited to passing a single packet into the network at once, then the bottleneck in our system may be the introduction of packets into the multicomputer network. Since it is our aim to study the response of that network itself, we don't want it to stall simply because there is not enough bandwidth between the running workload and the network.

It is easy to demonstrate that this danger is very real. Figure 4.5 shows the performance (in application exchanges per millisecond) of the multicomputer as we increase the number of links working in parallel between the CPU and the network. We see no performance improvement in increasing the parallelism of the CPU link beyond 4. This is expected, since it is not possible under any conditions for the C104 router to send more than 4 packets out at once (one in each direction) from the CPU, at least in this configuration (and the incoming bandwidth is similarly limited). In the study below (§4.2.4) of the effects of local queuing policy on performance, we will again see no case where increasing the CPU link parallelism beyond 4 offers any performance improvement.

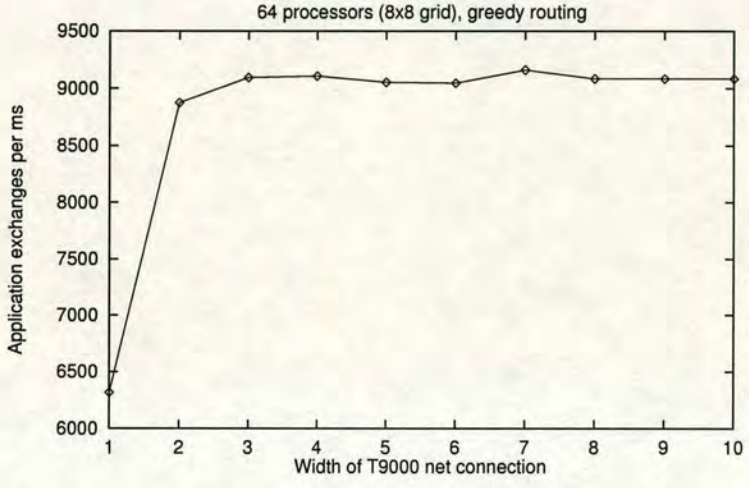


Figure 4.5: Performance dependency on CPU–router link width

It is reasonable to assume that a good policy would be to make the width of the link between a CPU and its router equal to the degree of the network at the point where the router is attached. Our experiments here confirm that this policy removes the risk of this link becoming an obstructive bottleneck in the network. Basak and Panda [BP96] also identify the width of the external link as an important factor limiting performance, and they conclude that a link width of 2–4 is typically effective.

4.2.3 The size and placing of routing buffers

We recall from chapter 3 that the C104 router is organised as sets of paired input and output links connected by a central crossbar switch, and that there is a small amount of buffering available at both the input and the output channels.

It is a property of the T9000 family's communication design that no data sink (the T9000's input link) will ever block, and that no router input will ever block

unless either its connected output has blocked, or there is contention within the crossbar between two input links attempting to send a packet to the same output link. The former case is simply a matter of an existing blockage propagating backwards through the network; it is only crossbar contention which can introduce new blocking into the network.

The presence of buffering is designed to help to alleviate this problem. If the crossbar can offload a packet into a local output channel, then the crossbar is free to continue accepting new packets even if that output channel is itself blocked. Buffering does not eliminate crossbar contention, but it does reduce the effect of that contention propagating backwards through the network and reducing performance upstream too.

We can see a similarity between this effect and the effect of the flow control strategy we have adopted for our efficient, packet-passing communication implementation. The genuine T9000, and our accurate DS-linkimplementation, perform flow control with 8-byte granularity; every flow control token received grants permission to send another eight tokens of data. The effect of our simplification is to perform only packet-level flow control, so we don't start to accept any packet on any input link unless we know we have enough buffering available to completely accept that packet. The primary consequence we expect to observe is that if our buffers fill, the simplified implementation will take longer to start accepting new data than the accurate implementation.

A second property of our simplified implementation is that it has no output buffering. If we look at figure 4.6, we can consider the difference between having a large buffer on our output buffers, at **A**, or at the input buffer at **B**. If the buffers are at **A** then the switch's crossbar can drain data rapidly into that buffer, as this involves only local on-chip communication. If the buffering is at **B**, in the input buffer, then draining data through the crossbar is limited by the speed of the physical DS-Link's serial data line.

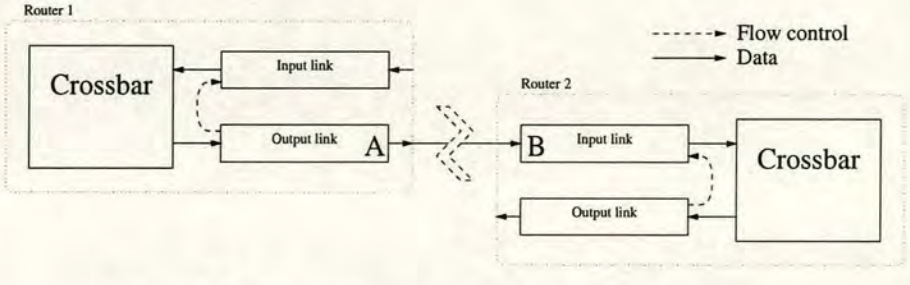


Figure 4.6: Connecting buffers in adjacent routers

Whereas the real T9000 implements a small amount of buffering on both

the output and the input links, our simplified implementation has only an input buffer. The impact of this will be that the crossbar will be occupied for longer in sending data out over the DS-link, perhaps delaying any subsequent packet buffered in the input link feeding into that crossbar.

To study these buffering effects, we set up a simple boundary-exchange experiment. We used a simple 8x8 processor grid carrying an optimally mapped 16x16 logical grid of processes. We now look at how the performance of this system varied as we changed the amount of buffering available for two experiments: one run on the accurate, token-passing communication implementation (with the buffering shared equally between the input and output links), and the other on the more efficient packet-passing implementation.

In figure 4.7, we show the results when greedy routing is used. The first observation is that, as seen in §4.2.2, performance is markedly degraded when we only have a single link between the CPU and the router. This does not come as too much of a surprise, and we have already decided to use a link width equal to the degree of the routing network (four, in this case) for our experiments.

The second obvious point is that performance increases in steps on the packet-passing implementation. Since this model performs flow control on packet boundaries, not byte boundaries, we cannot transmit a packet until there is a complete packet's worth of free space in the destination input buffer. Adding a few extra bytes of buffering has no impact; only when we add enough space for an extra complete packet in the input buffer does performance improve.

We observe that the packet-passing implementation performs noticeably worse than the token-passing implementation. However, we have already predicted that the packet-passing throughput is going to suffer somewhat from the lack of any output buffers on the router, and in figure 4.8 we try to mimic this effect in the token-passing implementation. This graph shows how performance varies as we move buffering space from the output buffer to the input buffer, keeping the combined input plus output buffering space constant at 200 bytes throughout.

This does show that there is a drop-off in performance at each extreme, and the best performance occurs when we have at least one packet's worth of buffering at both the input and the output link. The most important observation however is that when we reduce the amount of input buffering on the token-passing implementation towards zero, the performance approaches precisely that observed with the packet-passing implementation.

Table 4.3 lists the observed results numerically. We record here the performance of the packet-passing implementation with 200 bytes of buffer space, and

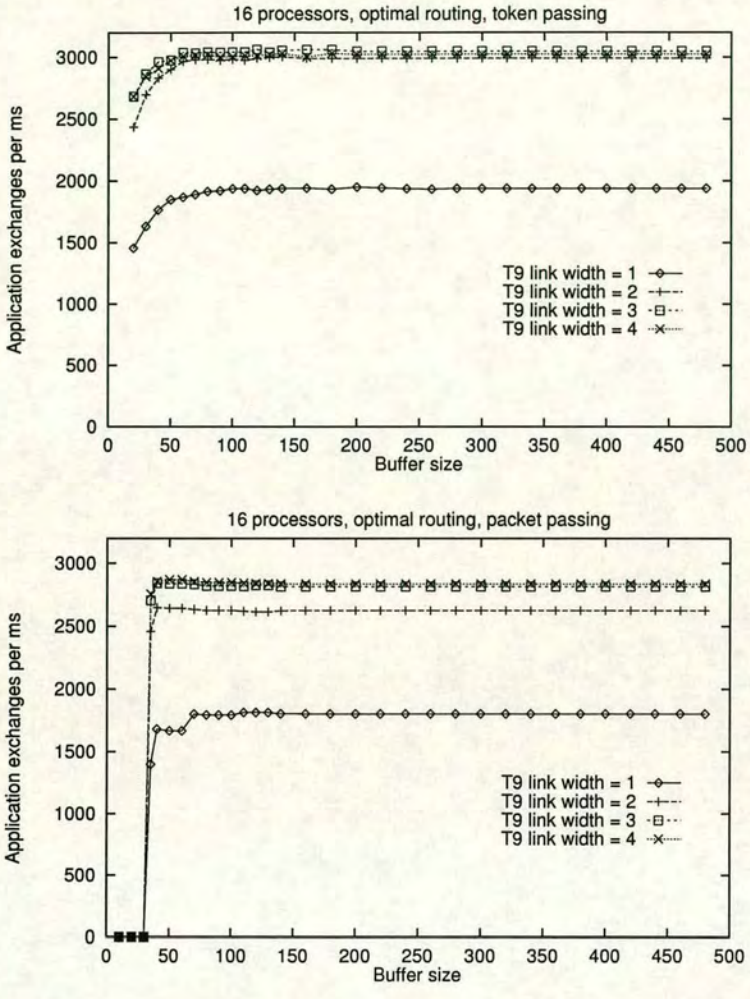


Figure 4.7: Performance dependency on buffer size: greedy routing

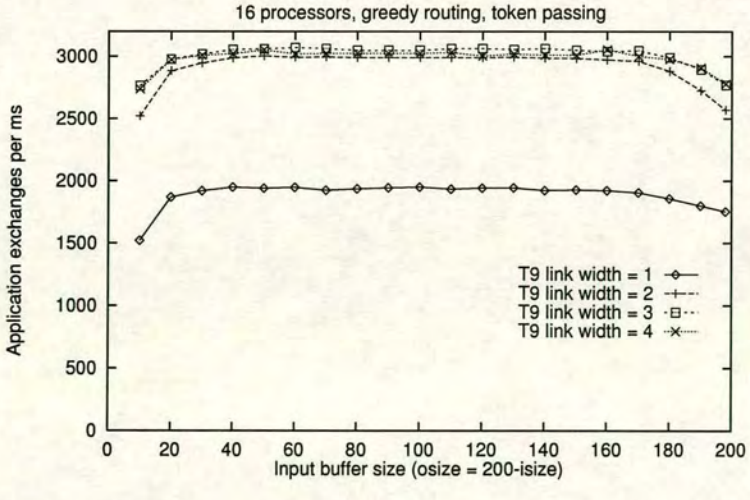


Figure 4.8: Performance dependency on buffer placement: greedy routing

CPU link width	Token-passing model	Packet-passing model	Adjusted token-passing model	Residual difference
1	1753	1804	1840	2.00%
2	2569	2625	2697	2.74%
3	2770	2814	2908	3.34%
4	2780	2835	2919	2.96%

Table 4.3: Performance of the token- and packet-passing implementations

of the token-passing implementation with 198 bytes of input buffer and only two bytes of output buffer. The token-passing implementation has got to transmit flow-control tokens interleaved with data. These tokens account for 4 bits for every 8 data tokens (of 10 bits each) sent, or 5% of the system throughput. In column 4 we account for this by normalising the token-passing performance by a factor of 1.05, and we now observe a numerical correspondence between the two implementations of within about 3%.

We can repeat the experiment using two-phase random routing throughout, and figure 4.9 shows the observed performance. Again we see the same steps in the packet-passing model's performance curve, and again the performance with packet passing is asymptotically nearly 10% worse than the token-passing performance.

This time the performance of the token-passing implementation with 198 bytes of input buffer and 2 bytes of output buffer in figure 4.10 is equal to the performance of the packet-passing implementation with a 200 byte buffer in figure 4.9, to within under 2%.

4.2.4 Local Queuing Policy

The final experiments we will look at in this section involve the effects of the local queuing policy implemented by the routing hardware. There are two places where packets get prioritised:

- The T9000 maintains separate internal queues for data and acknowledgement packets, and always transmits any pending acknowledgements first; and
- When the crossbar switch of the C104 router relinquishes a connection between an input link and an output link, it must arbitrate between any other input links which are presenting a packet destined for that output link.

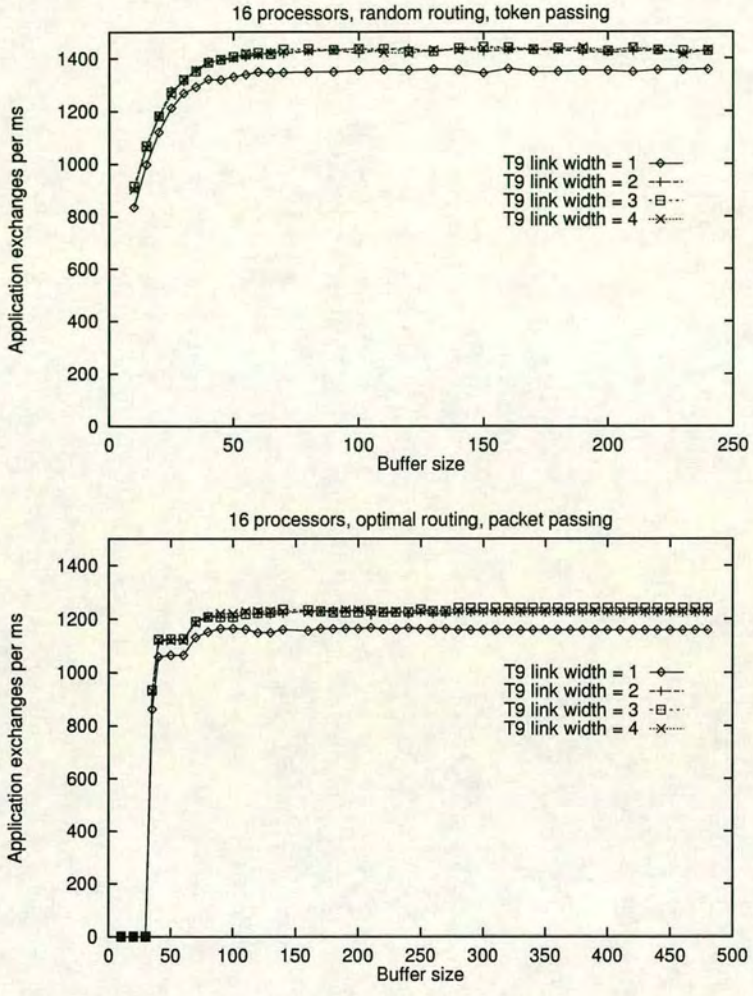


Figure 4.9: Performance dependency on buffer size: random routing

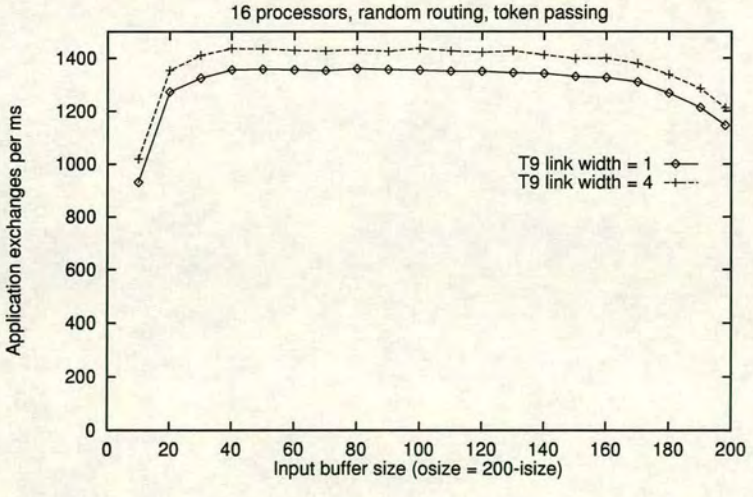


Figure 4.10: Performance dependency on buffer placement: random routing

We will refer to these as "ack prioritising" and "crossbar queue ordering" respectively.

The real transputer devices implement ack prioritising in the T9000, but do not do so in the C104. The only ordering implemented between contending input links at the C104 is simple FIFO queuing. However, a simple synthetic workload can identify possible problems with this regime. Recall that we are mapping a torus-connected boundary-exchange problem onto a grid-connected routing network. This will obviously exhibit edge effects, since the logically adjacent processes mapped onto opposite edges of the processor grid will have to transmit their packets over the entire diameter of the network, while other processes are communicating with nearest neighbours.

This is a pathological example which is worth looking at, since this type of bottleneck, where just a few logical message paths are unusually long, is not uncommon in real parallel systems. The trouble is that if we do not prioritise the packets which are travelling the longest distance, we may end up delaying these packets at each and every hop of their path. The cumulative effect is that the packets with the longest paths — the bottleneck packets, that is — are also the packets which suffer most from queuing delays in the network.

We implemented a number of alternative packet queuing policies to determine whether we can improve things at all. The alternatives we chose are:

- FIFO (the default behaviour as implemented on the C104)
- Random: select an input link at random from the possible alternatives
- Net-first: Prioritise packets already in the routing network over packets arriving from a CPU, to try to expedite multi-hop packets
- Oldest-first: chose the oldest available packet to send
- Hops-first: chose the packet with the highest hop count so far to send

In addition, we implemented optional prioritisation of acknowledgements over data packets in the C104 router.

The results show a quite surprising difference in performance between these different mechanisms. Figure 4.11 shows how the different combinations of queuing policy with or without ack prioritisation behaved on networks of 64, 256 and 576 processors. For efficiency, we used the packet-passing transputer implementation. The first observation is that in every case, implementing ack prioritisation in the router improved performance over the equivalent queuing policy without such prioritisation.

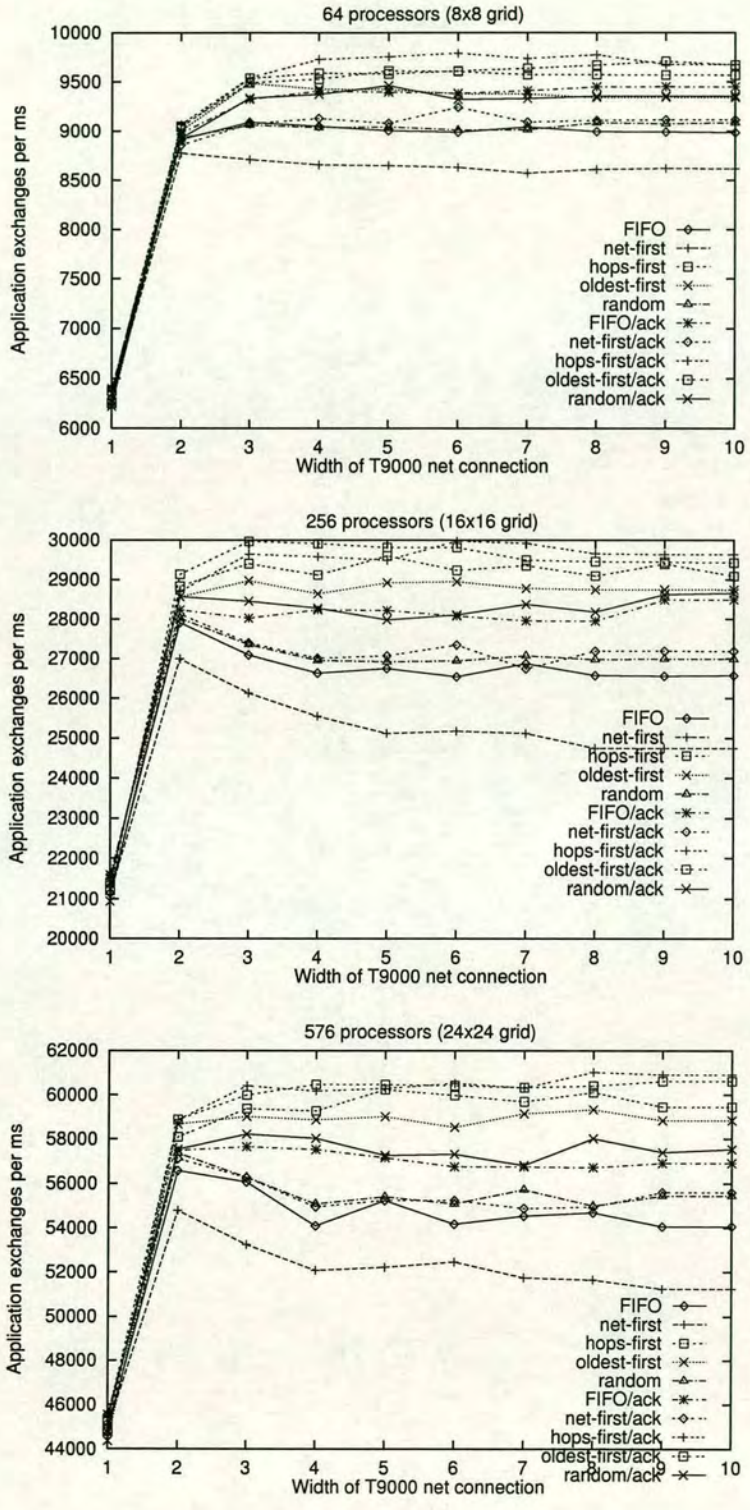


Figure 4.11: Queuing policy effects for optimally-placed boundary exchange

Secondly, we see that the best queuing mechanisms are consistently the age sensitive hops-first and oldest-first policies. There is little to choose between these two in the experiments here, which is reassuring since in a real implementation of packet queuing, it will be much easier to attach a simple hop count to packets than to add packet timestamping throughout the communication system. Even without ack prioritisation, these two policies outperformed every other.

With the exception of net-first queuing, every ordering policy outperformed the default C104 FIFO policy.

Before we leave this experiment, we will take a last chance to validate our simplified packet-passing communication simulation against the accurate token-passing model for this more realistic experiment. In figure 4.12, we show the results of repeating the 16-processor experiment from this section with a number of different implementations of the communication system. We used the packet-passing and token-passing (both with balanced buffers and with all buffer space allocated to the input buffer) implementations, and we ran both with the default 70 bytes of total buffering and with double-sized buffers.

Although there is a clear difference between the responses of the token-passing and packet-passing implementations, this difference vanishes when we remove the output buffer from the token-passing experiment. This behaviour closely follows the pattern we saw in section 4.2.3.

4.3 Conclusions

We have developed a simulator which closely follows the behaviour of the T9000 processor and C104 wormhole router. We believe that this system allows us to accurately investigate the timings and synchronisation characteristics of large T9-based multicomputer networks, but the implementation simulates each individual token passed on each communication link individually, and so suffers from poor performance.

We have presented an alternative communication model which promises the possibility of much more efficient simulation without losing the ability to express the complex behaviours of large communication networks. This packet-passing implementation generally runs complex simulations about an order of magnitude faster than the reference model.

We have validated the simplifications made in this implementation against the reference model, and have found that the simplified system behaves just as we would expect from a real transputer modified to have all of its buffer space

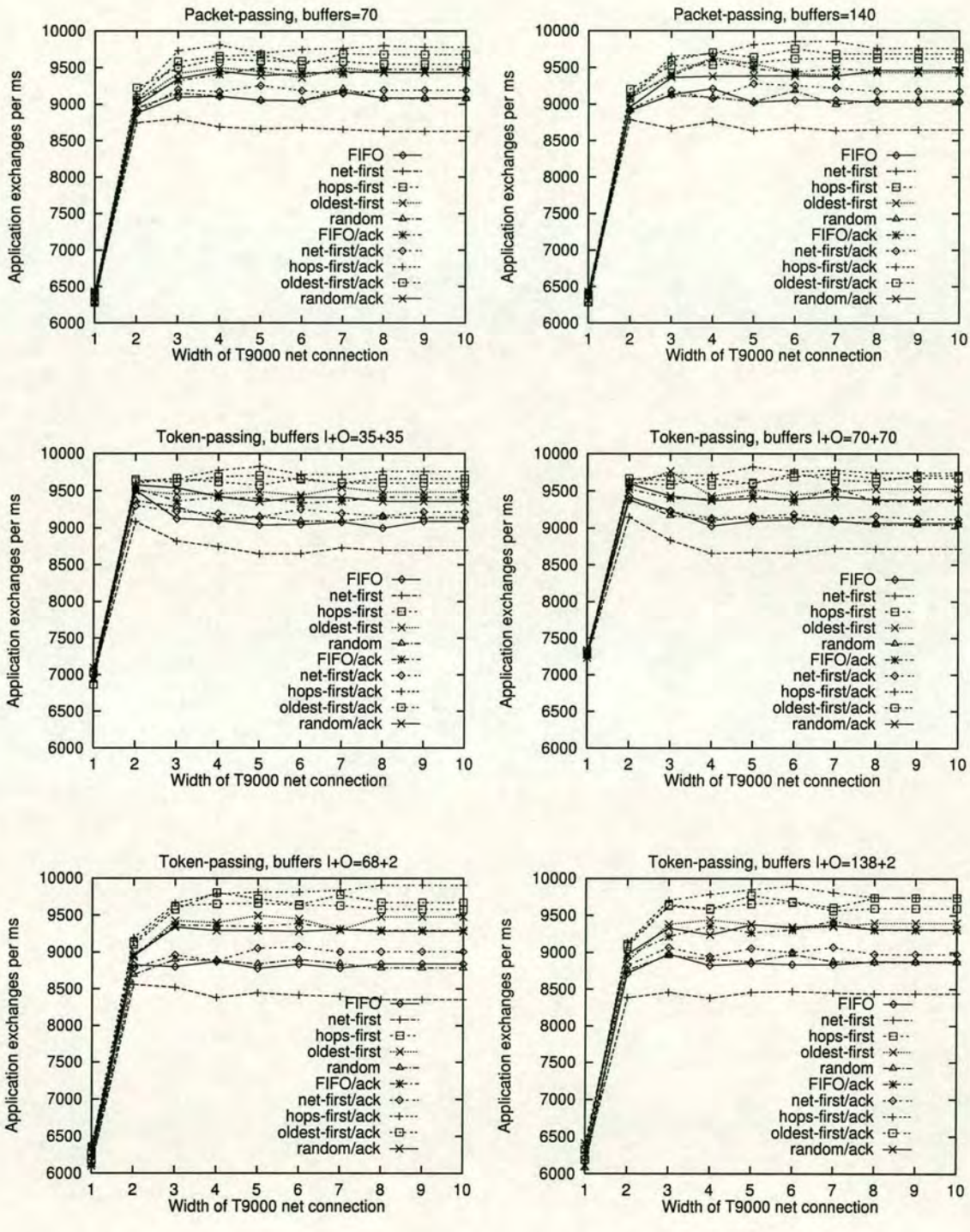


Figure 4.12: Comparing packet- and token-passing results

placed on its input links rather than shared between the input and output links. Although the faster implementation does not correspond precisely with the real transputer architecture, it does correspond to a possible and realistic variation of that architecture, and thus it satisfies the requirements of the simulator to be used to investigate complex routing phenomena in large multicomputer networks.

Existing empirical studies of simulated multicomputer performance have rarely considered networks of more than 128, or occasionally 256, processors. We conclude that a carefully designed simulator running on today's sequential compute servers should easily be able to handle networks over an order of magnitude larger than this, so that it becomes possible to investigate of the performance of the much larger communication networks made possible by modern wormhole-routing interconnection networks.

Chapter 5

Measuring Contention Effects

5.1 Introduction

How do we measure the performance of a multicomputer interconnection network? When we measure its "performance", what exactly are we measuring? In this chapter we will try to give some answers to these two questions, paying particular attention to the cost of communication contention within the network.

Up until now, we have been setting the scene. We have established that we can simulate, with accuracy, the transmission of packets through a transputer interconnection network. We can now start to at some of the effects which can be probed with this simulator. As we do so, we retain one primary goal: to be able to separately quantify the cost of contention against the network, and against the workload running on that network.

To this end, we will be asking three questions in this chapter:

How do we measure performance? We will introduce two quite distinct performance metrics to measure first of all the performance of the communication network, in terms of the communication time required for individual packets; and secondly the performance of the workload running on that network, in terms of the aggregate amount of useful work being achieved.

What performance are we measuring? We will also identify a number of different effects which affect "performance". In particular, we will need to clearly distinguish between flow-control effects in which packets are queuing up in a strict queue for injection into the network, and true contention in which packets already accepted into the network are interfering with each others' routing.

How do we quantify the cost of contention? An important objective of our measurement of performance is that we would like to be able independently

to measure both the presence of contention in the network, and the actual cost of that contention to the performance of the workload. In order to measure that cost, we will introduce three separate communication models which will allow us to run our experiments with contention effects selectively disabled: this will allow us to tell exactly how much faster the workload might have progressed if it were not for contention.

5.1.1 Packet Interactions

The complexity we will be trying to deal with throughout this chapter is the presence of many different types of interaction between separate communication packets in our network. Every interaction between any two packets during a simulation experiment creates some form of synchronisation between those two packets. Contention is the obvious form of interaction between packets in a communication network. At a higher level, applications also impose ordering constraints between packets.

As a result of contention, any given packet journey through an interconnection network may take a variable amount of time, and as a result of interactions in the workload, that variability may change the activity of the application. How do we untie this knot? Ultimately, we can make as many measurements as we want about contention in our network, but we still need to be able to relate that information to predictions about workload performance.

In this chapter we concentrate on trying to unravel this chain of cause and effect. To properly understand the effects of contention on performance, we need to be able to quantify two separate things:

- The *amount* of contention taking place in the network, and
- the *overall effect* this contention has on the performance not only of the communications network, but of the workload as a whole.

This chapter will lay the groundwork required to measure these effects. We will define a synthetic workload which allows us to simulate different degrees of interaction within the workload. Using this synthetic workload to drive our simulator, we then identify different interactions which can occur between packets, and we show that we can measure both the presence of contention, as measured by an increase in packet delivery times, and the effect of that contention as measured by a decrease in workload performance. Critical to the measurement of contention costs will be the ability to selectively disable internal network contention in the simulator.

One very specific effect we will have to identify in this chapter is flow control. During this chapter we will have to distinguish between true contention—congestion within the network—and flow control effects where we are merely queuing packets up for injection into the network.

5.2 Network and Workload Synchronisation Effects

In the rest of this chapter we will be looking both at packet interactions which arise from properties of an interconnection network and those which result from properties of the workload. Before we start to define a workload to express these properties, it will be useful to summarise the types of interactions we expect to see. In particular, in both the workload and in the communications network, there is a clear distinction between two quite different forms of interactions between packets:

Predetermined interactions in which the interaction is entirely deterministic and predictable. The packets are strictly ordered: one packet cannot be generated by a particular processing node until another packet has been received at that node

Incidental interactions in which the interaction results from a coincidental meeting between two packets.

There are many examples of both types of interactions. Predetermined interactions include

- Application ordering constraints: the flow graph for a workload may dictate that a process receives certain data, performs some computation and then passes on a result to neighbouring processes. In this case, there is an obvious predetermined ordering between the packets received by a process and those sent by it.

- Message processing: a T9000 communicates by sending data packets of up to 32 bytes and receiving acknowledgement packets in return. Acknowledgement packets cannot be sent from a node until the arrival of a data packet at that node; subsequent data packets cannot be sent until the acknowledgement for the prior packet has been received. Flow control can be considered to be a form of this type of interaction if the workload uses blocking sends.

In addition, both the workload and the communications network may result in extra, incidental interactions between packets:

- Contention between packets results when one packet is awaiting output on a given communications channel but another packet is already occupying that channel;
- The T9000 maintains separate output queues for acknowledgement packets over data packets and transmits acknowledgements in preference to data, imposing an extra ordering constraint between the two packet types;
- If the workload is not entirely deterministic (its flow graph is not fixed in advance), then the behaviour of any process—and hence the injection of new packets—is to some extent a response to the delivery of other packets to that process, creating an implicit interaction between incoming and outgoing messages.

5.3 Selecting an Appropriate Workload

We can now start to define a workload which will be used to look at these synchronisation effects. In this chapter, we are primarily concerned with being able to observe and to measure synchronisation properties inside the communication network, and their effect on overall workload performance. As a result, we are not necessarily bound to use a workload modelled on any one specific real-world application. It is much more important for now that our workload is capable of being tuned to display different synchronisation properties.

Broadly speaking, the workload stimulus used in existing work on performance analysis of parallel systems (including experimental, analytical and simulation studies) can be divided into two classes: workloads generated purely at random according to some parameters, and workloads which follow the pattern of some definite algorithm, either generated from a trace of a running program or produced on-the-fly. In this chapter we will be using a specific class of the former variety of random synthetic workloads.

One of the most important features of a synthetic workload is one which is frequently overlooked, especially in analytical models of multicomputer performance. The synchronisation characteristics of a parallel workload—the way in which activity of any one process in the workload depends on the order and timing of the arrival of incoming messages—can have a profound impact on the performance of that workload, and on the response of its performance to changes in the

underlying communication network.

To put the synchronisation issue into context, consider first the situation when intra-application synchronisation is ignored. A very common simplifying assumption made when dealing with random synthetic workloads is that the parallel workload consists of multiple independent processes, where each process is generating messages to other points in the network in a manner oblivious to the activity on those other nodes.

There are number of reasons why this assumption is commonplace. First of all, there is a substantial body of literature dealing with the properties of large, telecoms networks which are similar in many ways to multicomputer interconnection networks. A telecoms provider can, to a large degree, safely assume that the sources of data coming into the network are independent of each other, although they may still be correlated through dependence on common variables such as the time of day.

Secondly, there exist powerful analytical tools for performance analysis, such as queueing theory, which rely on the assumption that the workload is generated by independent processes with exponentially distributed generation of new work.

However, there is a large class of applications which behave substantially differently to this. Many parallel applications are decomposed into a set of communicating processes, where each process performs a repeated cycle of exchanging information with some of its neighbours and then performing some local computation with the information obtained for this timestep. The Bulk Synchronous Processes paradigm employs workloads of this form, and many common numerical codes such as the physical domain decomposition found in fluid analysis or the solving of systems of linear differential equations also reduce to processes which communicate with a fixed set of near neighbours at each time step.

The important property of these applications is that between computations, any one process in the workload communicates synchronously with a subset of the other processes in the system. The workload is not fully synchronous — the processes never synchronise themselves to any global clock. Rather, it is *loosely synchronous*.

In such a system, if one process stops, its neighbours will stop as soon as they try to exchange information with it; those stalled neighbours will in turn stall more distant processes at the next timestep; and in time the entire application will come to a complete stop. It is a fundamental property of loosely synchronous systems that although two logically distant processes can operate independently in the short term, in the long term all processes in the application must progress

at the same rate. They are all bound by the speed of the slowest process.

Throughout this chapter, unless specifically stated otherwise, the phrase "synchronous workload" will be used strictly to refer to a "loosely-synchronous workload". Globally clocked, tightly-synchronous workloads will not be discussed further here.

Note that we are explicitly excluding from our discussion here workloads which are adaptive and non-deterministic. Some classes of parallel workloads have the ability to adapt to the running speed of each process, giving more work to processes which complete quickly or mapping more processes to underutilised processors and thus avoiding the bottleneck which a slow process (or communication link) incurs in a loosely synchronous workload. The task farm is an obvious, extreme example of an adaptive workload. Studying such systems encounters the problem that the workload can no longer be considered to be constant: it is now a function (at least in part) of the communication network response, and so determining the response of that network to the workload becomes much harder. By explicitly considering only non-adaptive workloads with predefined process dependency graphs, we avoid this feedback loop.

5.3.1 Properties Required of the Synthetic Workload

What properties do we require of the synthetic workload? Obviously, our aims in this chapter require that we have control over the synchronisation properties of the workload. If we are to observe how the network responds under different load conditions, we also need to be able to control the amount of work generated by the workload.

To achieve these aims, we want to have control over:

Synchronisation diameter The workload must be able to display synchronisation over different ranges, from direct communication with nearest neighbours up to communication over the entire diameter of the processor network.

Communication/CPU ratio The amount of communication occurring relative to CPU time consumed, including both the frequency and size of the communications.

Local synchronisation Amongst the most important factors we want to be able to control are the synchronisation characteristics of the workload. We can identify both local and global synchronisation characteristics. We define local synchronisation to be concerned only with the existence of events

where a process synchronises itself with purely local events, where the event is local in either time or space.

For example, if a process uses blocking message transmits (ie. it waits in any way for an acknowledgement from the receiving CPU), then it is necessarily subject to local synchronisation, in that it is subject to the packet load experienced over some path through the network at some point in time. However, it is not necessarily subject to global synchronisation—just because it has to wait for a message to get through the communication network does not mean it has to wait for any specific process to reach a certain point in its lifetime.

Global synchronisation Global synchronisation we can define as the property characteristic of loosely-synchronous workloads. It is marked by repeated points in the lifetime of a process when that process must synchronise itself with a specific point in the lifetime of neighbouring processes, with the transitive closure of the adjacency function fully connecting the entire process graph.

There is a profound difference between local and global synchronisation. The workload-wide loose coupling of processes which characterises global synchronisation requires that processes synchronise at *pre-defined* intervals in their lifespans, and no matter how far ahead in its operations one process has progressed in advance of another, on a synchronisation event the first processes must simply wait until the second process has reached its own synchronisation point. As long as the dependency graph between processes in the workload is fully connected, it is simply impossible for any process to get arbitrarily far ahead of any other process in the system as execution proceeds.

Note that the existence of synchronisation points between two processes does not by itself constitute global synchronisation. Take the client-server model of computation as an example; when a client submits a request to a server, the ensuing dialogue may require both processes to wait for messages from the other. However, the server does not have any pre-determined list of requests which have to be processed in order: it simply deals with requests on demand. There is no synchronisation between fixed points in the life of the server and any points in the life of the clients. The ordering of synchronisation events in the client is reactive to the incoming workload, not predestined. Such client-server communication would only count as local synchronisation.

A workload need not necessarily display even that much synchronisation. A

process could be created which would perform computation for some time and then fire off some outgoing communication message asynchronously. If that message is to be carried reliably to its destination, then we obviously cannot afford to forget the message: the message must either be retained locally in the sending process for retransmission on failure or in the communication subsystem for guaranteed delivery. We *must* assume that the buffering space for these messages is finite, and so if the workload generates messages faster than they can be transmitted to their destination, then either messages must be dropped or the sending process must eventually stop creating new messages.

If the process performs reliable communication under these conditions, then it is still locally synchronised to the communication medium—there is a flow control mechanism to stall the process when communications saturate, which counts as an implicit synchronisation between the process and the current state of the communication network. However, if packets are simply dropped silently under these loads, then the workload is truly unsynchronised.

It is possible to criticise these unsynchronised workloads as somewhat unrepresentative of real parallel programs. However, they are widely used as synthetic communication load generators in many studies of multicomputer routing networks, where the simulated load consists of the random injection of independent messages; in such cases, communication saturation does result in packet loss, the saturation level being used to characterise the network's performance under stress.

We may construct a workload that displays none of these properties, purely local synchronisation, or both local and global synchronisation, but we need to do so carefully. If we are to be able to accurately observe the effects of global synchronisation on overall performance, we need to be careful when selecting our workloads so that we can change the synchronisation characteristics while not preserving as many as possible of the other properties of the workload.

Given the constraints above, we can now define the workload which we shall use throughout this chapter.

5.3.2 Definition of our Synthetic Workload

Our experiments in this chapter shall all take place on the same simple 2-dimensional grid of transputers as used in chapter 4. The network is parameterised by its diameter D, resulting in a network of $N = D * D$ nodes each composed of a single T9000 processor connected by a fat link of 4 DS-Links to a single C104 router (following the result from chapter 4 that the width of this link be at least

as wide as the degree of the routing network).

Given the 2D grid as our standard multicomputer architecture, we restrict ourselves for now to similarly organised workloads. We will make a number of extensions to the very simple process grid network with nearest neighbour communications as used in chapter 4. However, we will retain the uniformity of that workload, with precisely the same workload running on each processor in the network.

Our workload for this chapter uses a Process very similar to the simple Process used in chapter 4, but with a different pattern of communication. We place a constant number n of identical processes on each CPU in the system. At each communication point in the process's lifetime, it sends a constant number of messages, m, each of length M (where M may be constant or may be a random variable). The amount of time each process spends computing between communication activity is defined by another random variable c. Finally, each message's destination is chosen uniformly at random from amongst all of the processes in the $d * d$ square grid centered on the sending process: d is the *communication diameter* for the workload. The set of possible destinations is simply truncated at the edges of the processor grid: near the edges, processes select uniformly between all those processes whose x and y coordinates are both no more than $d/2$ removed from the sending process.

The synchronisation characteristics of the workload can also be selected from amongst three alternatives:

Loosely-synchronous In this mode, processes are loosely bound in lock-step. Processes need not synchronise with their all of their neighbours on every iteration, but only when they happen to select a neighbour as destination for a message. When a message is transmitted, it is both transmitted synchronously by sender and received synchronously by the destination process—once a process has sent all of its messages for the current workload iteration, it blocks until all of those messages have been acknowledged by the receiving process and all incoming messages for the current iteration have been received.

The result of this exchange is that although any two processes may become out of step with each other as they proceed, they are forced back into step as soon as either tries to communicate with the other.

Asynchronous with blocking sends The workload may also operate without any loose synchronisation between processes, by eliminating the synchro-

nisation of processes with incoming messages. In this asynchronous mode, there is no global synchronisation, but messages are still always sent synchronously, using blocking sends. The difference here is that incoming messages for each process are always acknowledged immediately they arrive (although the messages are ignored by the receiving process: to do otherwise would reintroduce global synchronisation back into the system).

The reason for using blocking sends is simply to maintain the same local synchronisation characteristics as in the first, loosely-synchronous mode. Upon sending a message, a process must wait for it to reach its destination and for the acknowledgement to return before proceeding. This naturally limits the number of messages which can be outstanding at any time, so that the workload can necessarily consume only a finite amount of buffer space at any time. The blocking send provides a natural flow control mechanism which prevents the network from ever saturating.

Asynchronous with non-blocking sends In this final mode, we not only discard global synchronisation but local synchronisation too. The process activity is the same as for the previous mode, except that non-blocking message sends are used. Now, each process simply repeats a loop in which it computes for some interval and then queues its outgoing messages for the current interval for asynchronous transmission.

In this mode, the workload processes have no synchronisation with either outgoing or incoming messages. There is no flow control at all between the workload and the communication network, so it becomes possible to saturate the network if sufficiently many messages are generated. The transputer architecture has no mechanism for dealing with saturation: the architecture defines that there must always be at least a buffer for the first packet of any message on a virtual channel, so there is always enough buffering available for as many outstanding messages as may be present at any time.

In our simulations, we deal with effective saturation by keeping a count for each process of the number of messages queued for transmission but not yet acknowledged as received. If this reaches a certain arbitrary threshold, we deem the network to be saturated and any further transmit requests are dropped and accounted as a saturation failure rather than being queued. This is equivalent to giving each process a finite number of virtual channels over which to send outgoing messages; once the quota is exceeded, the process simply cannot generate new messages until a transmission completes

on one of the existing channels.

To summarise:

Definition 5.1 *We define our synthetic workload by the variables*

$$(D, d, n, c, m, M)$$

and one of the modes loosely-synchronous, blocking asynchronous or non-blocking asynchronous, where

$\boldsymbol{D}$ *is the diameter of the CPU network, a constant;*

$\boldsymbol{d}$ *is the communication diameter, a constant;*

$\boldsymbol{n}$ *is the number of processes per processor, a constant;*

$\boldsymbol{c}$ *is the per-iteration computation period, a random variable, in μs;*

$\boldsymbol{m}$ *is the number of messages generated per iteration of each process, a random variable; and*

$\boldsymbol{M}$ *is the size of each message, a random variable, in bytes.*

5.4 Three Workloads: Initial Observations

In the remainder of this chapter, we will be looking at the observed performance of the three variants of workload we have described—asynchronous, locally synchronous and globally synchronous. In this first section, we will not be concentrating specifically on contention. Rather, we will be getting a feel for how the workloads behave under load.

We will be looking briefly at some of the various alternative performance metrics available to us, to get an idea of the sort of load being placed on the network by the three workload classes. We will look in quite some detail at the interactions between application and network when we approach saturation, and will see that some initially counter-intuitive results can arise when we ignore the precise packet ordering behaviour of the network and the precise flow control mechanism used by the workload.

We will then look at the alternative performance metric we are using—packet delivery times. Finally we will look at the same three workloads under our alternative communication models, to see how we can determine the performance being lost due to contention.

Throughout most of this chapter we will be using a standard setup for our workload: we run a single synthetic workload as described in definition 5.1. In our first examples we use a uniform processor grid of 16 by 16 nodes with one process running per processor, with our workload set to send exactly one 32-byte message per timestep over a distance up to 50% of the processor diameter, and messages of length 32 bytes used throughout. These are quite natural values to use for now: the use of a grid of this size allows us to generate long enough communication path lengths to see interesting contention effects, and limiting the message size to 32 bytes for now avoids the splitting of messages into multiple-packet by the T9000 (which would introduce extra synchronisation effects into the communications load).

To exercise the system at various load levels, we vary one parameter which is under our complete control in the workload, the inter-communication process compute period. Recalling the basis of our system's timings in section 3.1.1, a complete point-to-point communication of a 32-byte message takes 3.4μs plus a per-hop latency of at least 100ns. We run our processes with compute periods of between 1μs and 200μs, corresponding to compute cycles from about a third of a best-case message latency up to around 60 messages'-worth.

In terms of our workload parameters from definition 5.1, we are using the parameters $(16, 8, 1, c, 1, 32)$, where $1 \leq c \leq 200$. We run the experiment with each of the three workload models: loosely synchronous, and asynchronous with and without blocking sends; and we run it on each of the three communication models defined below: full, throttled contention-free and fully contention-free. All simulations are run until the observation of the primary response variable (mean successful packet sends per unit time) is accurate to within a 95% confidence interval of $\pm 1\%$.

The communication diameter of this workload, 50% of network diameter, is quite high. Although not quite at the 100% level equivalent to a truly random choice of destination for each packet transmitted, it is certainly high enough to induce fairly long message paths through the network. This high average path length is deliberately selected for our first experiments as a tool for provoking contention.

5.4.1 Parameters and Responses: Measuring Workload Performance

Before we start to look at some of the performance characteristics of simple simulated synchronous and asynchronous workloads, we should look more closely at

some relationships between input parameters to the simulations, and response variables in the experimental results.

Specifically, we need to look at the expected differences in response variables when we change between the three classes of application workloads described above in section 5.3.2. This is particularly important to get straight because of some fundamental differences between the classes: a variable which is an input parameter for simulations of one class may become a response variable for another class. This has implications for the choice of which response variable to use to represent aggregate performance of the parallel workload.

Since our ultimate interest is in the performance of parallel applications, an obvious candidate for our primary performance response variable is simply the amount of work performed per unit time by the workload (or its inverse, the amount of time taken to perform a fixed amount of work).

Given our workload model of fixed processes each performing some period of computation followed by a communication exchange with some neighbours, we have several equivalent measures of how many computation cycles any given process has completed in a given time interval. In particular, since the number of messages generated by each process in each time interval is constant (or at least, in cases where the messages are more randomly generated, has a fixed mean and is therefore convergent over time), the number of communication messages generated by the workload is just as good an indicator of progress as is the amount of CPU time accumulated or the number of iterations of the workload completed.

However, on a fully asynchronous system, the number of packets generated by the workload is in fact not a response variable at all. We have already noted that our asynchronous workload (with non-blocking sends) is quite permitted to generate messages faster than the communication network can consume them, and that messages may be dropped in this situation. Another property of the fully asynchronous workload is that its message generation rate is fully defined by parameters under our control when we set up the experiment. As a result, this single variable—message generation rate—can realistically be used as a response variable when we are running locally or globally synchronous workloads, but the same variable is merely a trivial function of our input parameters when used on a fully-asynchronous workload.

Using the number of messages *successfully* transmitted as our primary performance metric does not suffer from this problem. In the following experiments, we will indeed be using the number of complete messages sent (per unit time) as our primary indicator of workload progress.

Another obvious choice of metric might be to use the percentage CPU utilisation, and indeed for the globally- and locally-synchronous workloads this works well. However, it is worth noting here that the CPU utilisation breaks down as a useful metric on the asynchronous, non-blocking workload: that workload, by definition, is guaranteed to consume 100% of available CPU time, ignoring any errors if the communication network is unable to keep up with the demands placed on it.

5.4.2 System utilisation for three workloads

As we begin our initial observations of the three workloads it may be useful to relate the applied load level parameter for these experiments, c in this case, to some traditional measurements of the amount of activity taking place in a multicomputer: processor and communication utilisation levels. In figure 5.1 we can see the load on the communications network expressed as an average percentage utilisation for all of the active internal network DS-links. We can see that we are covering a wide range of network load levels, starting from a utilisation of only a few percent and proceeding up to a maximum saturation of 50% of total network bandwidth. The accompanying figure 5.2 shows the corresponding percentage CPU utilisation at each load level.

As expected, the asynchronous, non-blocking workload has 100% CPU utilisation throughout, but there is a dramatic drop in CPU utilisation as we move into the region where we are dominated by communication costs. It is worth noting at this point that there is an obvious difference in the responses of the loosely and globally synchronous workloads. We now want to look more carefully at the differences between the workloads by using the metric identified above: the rate of injection of new messages into the network.

5.4.3 Message delivery rates for three workloads

As mentioned in section 5.4.1 above, the measurement of the rate of *successful* injection of new packets into the network, as opposed to the attempted injection rate, allows us to measure the actual amount of useful work achieved by the communications network under the three different types of workload.

In figure 5.3, we are observing this performance (specifically, the number of messages successfully transmitted per CPU per millisecond) for each of the three workloads using the full communications model. The first thing to observe is that although at low levels of load (corresponding to long inter-communication compute periods and hence a low communication-to-computation ratio), the num-

Figure 5.1: Network link utilisation

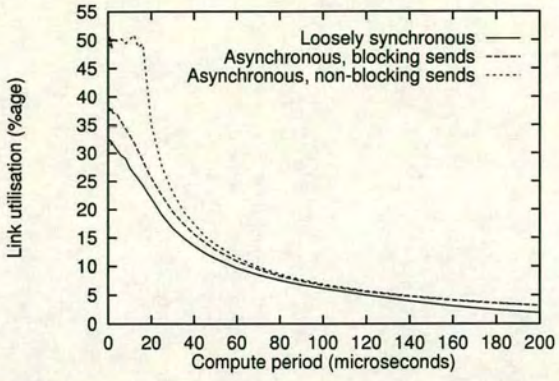


Figure 5.2: CPU utilisation

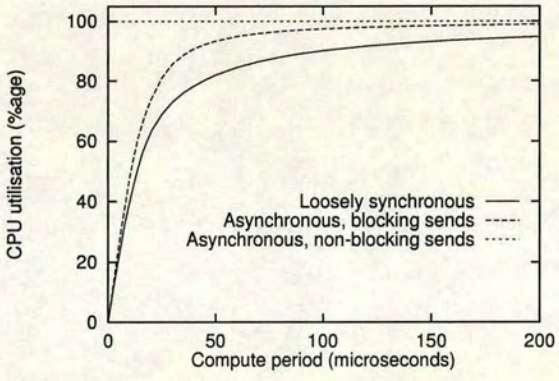
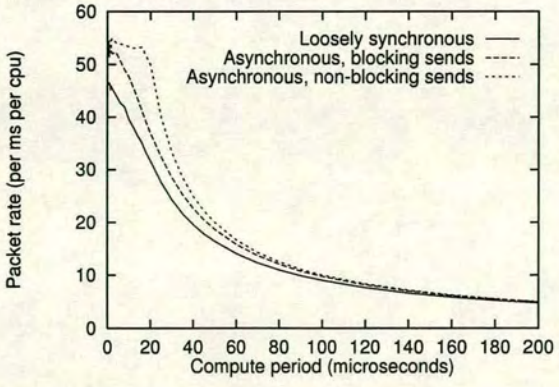


Figure 5.3: Message rates for the three workload classes



ber of messages we can generate is relatively insensitive to the synchronisation characteristics of the workload. In this domain, communication is simply not a dominant cost, and having to wait for communication to occur does not substantially degrade performance.

Compute period	Loosely sync.	Async, blocking	Async
1	46.4	52.6	53.4
2	45.6	52.1	57.1
4	44.1	52.2	57.4
8	42.0	49.0	52.4
12	38.2	45.7	58.1
16	35.2	41.3	56.5
30	24.3	28.5	33.3
60	14.1	15.8	16.7
120	7.6	8.2	8.3
200	4.7	4.9	5.0

Table 5.1: Table of message rates (per CPU per ms) for the three workloads

Compute period	Loosely sync.	Async, blocking
1	87%	98%
2	80%	91%
4	77%	91%
8	80%	93%
12	66%	77%
16	62%	73%
20	63%	74%
30	73%	85%
60	84%	96%
120	92%	98%
200	95%	99%

Table 5.2: Table of message rates as percentage of maximum rate

Table 5.1 summarises the same data in numeric form, and the accompanying table 5.2 shows the performance of our two internally synchronised workloads as a percentage of the work rate achieved by the fully asynchronous workload. The effect of synchronisation effects within the workload becomes very obvious in this format: we can see that at the point of greatest divergence in the workload's performance, our loosely synchronous workload is achieving only 62% of the data throughput of the fully asynchronous workload, and at that same level of applied load, the asynchronous workload with blocking sends itself achieves only 73% of the throughput of the highest level.

Our first conclusion here is that as the applied load on the network increases,

the synchronisation characteristics of that workload become more important in determining overall performance, and that the greater that level of synchronisation — the more that each process depends on other communications to arrive before it can proceed — the greater more its achieved performance will degrade relative to a fully asynchronous workload.

The second observation is less obvious. Although our three performance curves do diverge as CPU periods decrease and load increases, this effect seems to reverse as we near saturation of the network. In this particular experiment, the performance achieved by the fully asynchronous workload became saturated at load levels corresponding to CPU periods $c \leq 16$, and it is at this point that the locally and globally synchronous workloads begin to catch up again with the asynchronous workload. At the very highest levels of applied load, corresponding to essentially no computation between communications, the locally synchronous workload achieves fully 98% of the throughput of the asynchronous workload, and the loosely-synchronous workload achieves 87% itself.

Why should this be so? Note especially that in figure 5.1, we saw a divergence in the network utilisation between the fully asynchronous workload and the other, locally or globally synchronous workloads which persisted right up to the maximum load levels. Yet now we see that the workloads are actually achieving the same amount of work under these same conditions. In short, we are observing substantially different network loads under the asynchronous and locally-synchronous workloads with no difference in the overall achieved application performance. To understand why, we need to look much more closely at the behaviour of the network and of the workload at or near saturation.

5.4.4 Behaviour near saturation

We have just seen our first hint that something peculiar may be happening in our simulated system near saturation: two different workloads achieving the same aggregate amount of work are resulting in two very different amounts of network traffic. We will now look in some detail at the behaviour of our system under high load. Remember that in this chapter we are interested in identifying some of the different types of interaction between packets which are possible. We will find that this study of saturated networks illuminates some important and unexpected interaction effects.

As we push our networks to the limit, the precise behaviour which results will depend on where our system bottlenecks appear. We have already described some very obvious bottlenecks which we might expect: in the loosely-synchronous and

globally-synchronous workloads, the workload is defined in such a way that we simply cannot transmit packets faster than the network can return their replies.

The same is not true of the fully asynchronous workload, and we have already mentioned this: in section 5.3.2 we had to define a mechanism to prevent our fully asynchronous workload from creating more messages than could possibly be transmitted and hence growing the network buffers without limit on the host T9000. Our mechanism was to impose a limit on the number of unacknowledged messages that are allowed to be in transit from any one process at any one time. If that limit is reached, further attempts to send a message will fail until the final packet of an existing outstanding message has been acknowledged. We in fact limited the number per process to 16.

However, given the long path lengths being generated by this workload, it is quite possible for 16 messages per process to be in circulation at once inside the network—we are not necessarily guaranteeing that the T9000's output queues are full by imposing this limit. If we are to fully explore network saturation behaviour, we need to be able to push our workload model right up to the point where the output queues can be kept full and we have no artificial workload-imposed limits on the amount of traffic outstanding at once.

We can try instead to impose a limit which at first glance seems as if it ought to saturate the network, by placing an upper bound not on the outstanding message count but on the outgoing data packet queue size on the local T9000. For now, the actual value of the threshold does not matter much: we wish only to demonstrate some general properties of the system near saturation. We set the threshold to 16 messages in each case.

Figure 5.4 shows the performance achieved when we impose both of the two limits under otherwise identical load. At the same applied load level, $c \leq 16$, where we identified saturation in the network above, we find a sudden change in behaviour: at the very highest performance drops to a level around 20% lower when using the queue length limiting than when using outstanding message limiting.

What could be happening here? The first clue is to be found if we look at the statistical distribution of the number of packets successfully transmitted by each processor in the network. In figure 5.5 we are plotting the average number of messages transmitted by each processor (averaged over the whole network for each load level), with the error bars at each point indicating the standard deviation of the message rates for the different processors during each single simulation run. We can see that at very high load levels, the amount of work being achieved is far

Figure 5.4: Performance at and beyond saturation

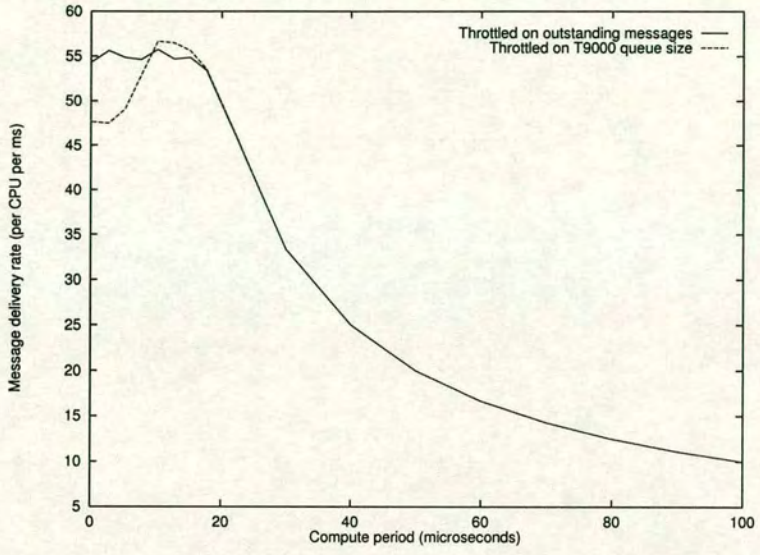
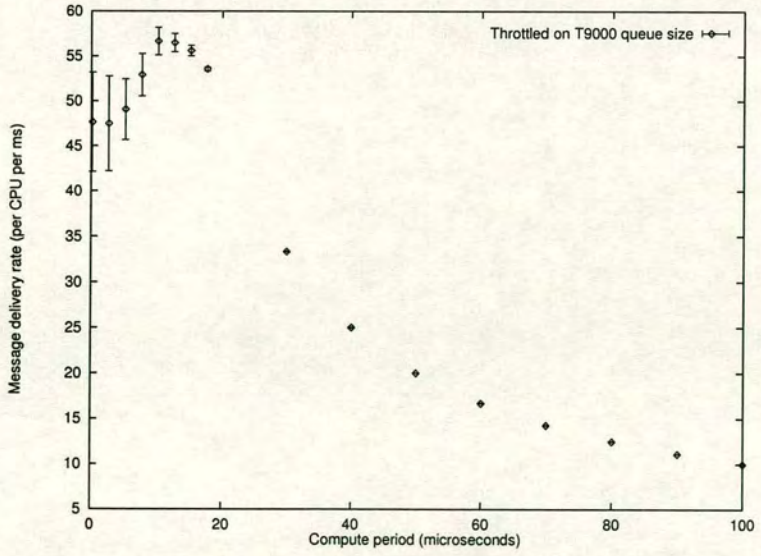


Figure 5.5: Distribution of performance near saturation



from constant over the network: the increasing error bars indicate an increasing disparity between the amount of work being achieved by the fastest and the slowest node in the network during each run, suggesting that we need to look at the exact distribution of load over the network to understand why our aggregate performance is falling.

5.4.4.1 Load Distribution in Saturated Networks

Figure 5.6 shows the observed distribution of load over our 16 by 16 processor network for loads corresponding to $c = 1000$, $c = 10000$ and $c = 100000$. The left-hand column displays the achieved workload: the number of packets successfully injected into the network by each processor per millisecond. The right-hand column shows the number of packets routed by each C104 per millisecond (hence the overall numbers in the right hand column are scaled by the average packet path length compared to the left hand column).

Two effects are dramatically illustrated here. First of all, we can see that under low load conditions, it is the application which is determining the amount of work achieved, as expected. At $c = 100000$, there is no significant variation of achieved performance from one CPU to the next.

Equally clear is that something unexpected is happening under conditions of high load. The first two sets of surfaces in figure 5.6 show a marked rotational asymmetry where none exists in the workload itself, and they show that processors along the lines $x = 0$ and $x = 15$ are achieving absolutely no work at all over the course of the simulation. Indeed, at the very highest load level this effect extends inwards toward the center of the network as far as $x \leq 4$ and $x \geq 11$.

There is only one place in our communications model where we have a clear asymmetry between the x and y axes, and that is in our use of greedy dimension-order routing: we always route packets in the x direction before routing them in the y direction. Could this be responsible for the observed asymmetry? It is very easy to reproduce the previous experiment using a modified router which routes along y first and then x, and figure 5.7 shows the resulting workload and router load for $c = 1000$. The pattern we see is identical to that which we saw in figure 5.6, only rotated though 90 degrees. The direction of our asymmetry is indeed dictated by the the dimension order in which we are routing our packets.

What is the origin of this behaviour? One last clue as to what is happening here was observed during the running of the experiments presented above: during these runs, the physical memory in use by the simulator was seen to increase steadily during the experiment, a behaviour not seen on any previous experiments.

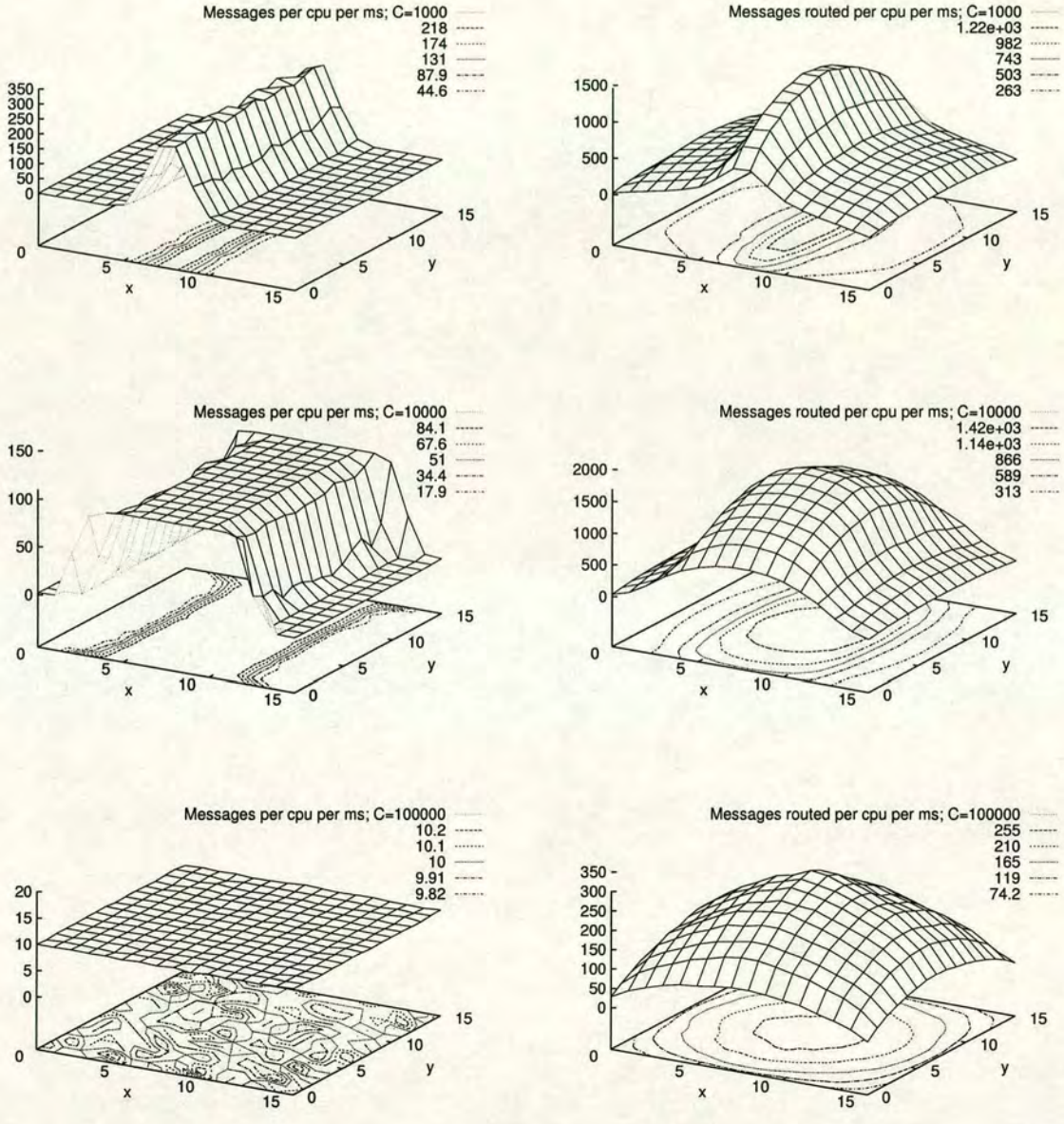


Figure 5.6: Workload performance and router traffic per node: throttled by T9000 output queue size

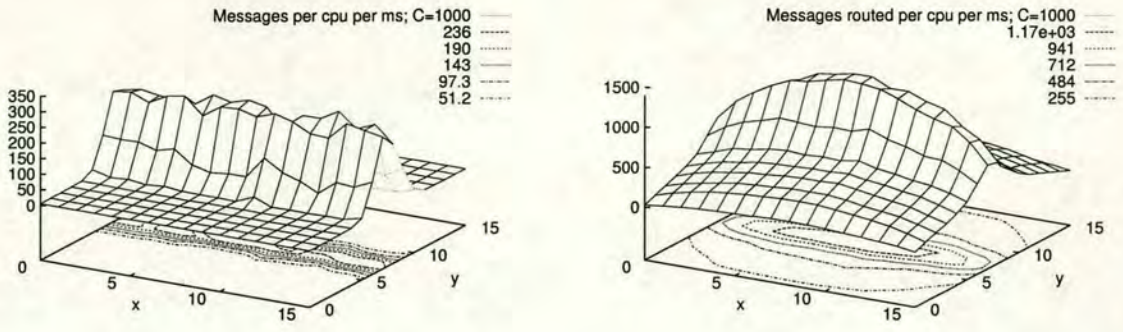


Figure 5.7: Workload performance and router traffic per node: routing along y then x

Although this was initially assumed to be a due to a coding error, it turned out that it was indeed a side effect of a genuine property of the system being simulated: the acknowledgement packet queues in certain T9000 processors were growing without bound. To be specific, this effect was seen on precisely those processors where the achieved workload remained at zero throughout the experiment. The lack of any workload progress is in fact quite expected if the acknowledgement queues are never fully emptied: it is a property of the T9000 that acknowledgement packets are always sent in preference to any data packets which may also be queued.

This observation, together with the fact that the direction of symmetry in the resulting load map depends on our routing dimension order, allows us to explain these results. There are two factors we need to consider:

- Because of our use of dimension order routing, the T9000 output queues at any point in time are much more likely to be full of packets waiting for transmission on the x axis than the y axis of the router network. If our workload is able to keep these output queues busy, then there is an unlimited queue of packets waiting at each C104 for routing in the x direction. On the other hand, most packets waiting to be routed in the y direction arrive at the C104 not from the connected T9000, but from neighbouring C104s. We can therefore expect these packets to be arriving at a rate already restricted by the performance of the network.

  The net result of this is that routing in the x direction is much more likely to be a bottleneck than routing in y in our network. Indeed, we see that achieved workload performance for any given node at high load is almost completely insensitive to the y coordinate of the node within our routing

grid.

- We know that even at low load levels, the center of our network is carrying more traffic than the edges simply due to grid edge effects: the network load surface for $c = 100000$ in figure 5.6 is evidence enough for this. For a uniform rate of message arrival at each router, we may expect that those messages travelling from the center of the network towards the edge will experience less contention than messages travelling in the opposite direction.

If we put these together, we might predict that traffic in the x direction will be more congested when routing towards the center of the network—towards the region $7 \leq x \leq 8$—than towards the edges at $x = 0$ and $x = 15$. If we consider messages being sent from nodes near the edge towards nodes in the center, then the effect of this will be that the data packets experience more congestion than the return acknowledgements. However, for data travelling in the opposite direction, towards the edges, it will be the acknowledgements which are more congested.

This can result in pathological behaviour. We have already had to introduce limits on the injection of messages into the network, either by throttling the number of outstanding packets per process or (less realistically) the size of the T9000 data output queues, in order to flow-control our workload. No such throttling exists on the generation of acknowledgement packets; we must generate such a packet for every single data packet received regardless of the local load conditions, even if conditions are such that data is arriving faster than we can send the responses.

5.4.4.2 Workload Throttling and Saturation Behaviour

We now ask: is this effect important? Is it relevant to real transputers, or is it merely an artifact of our synthetic workload?

Looking back to figure 5.4, we recall that the odd effect of aggregate performance dropping with increased load only affected the fully asynchronous workload when we implemented per-node throttling by limiting the size of the T9000 send buffer. When we only limited the total number of outstanding messages per process, performance did level off but did not actually drop with increased load.

However, the limiting of the T9000 buffer size is a rather artificial limit: in particular, removing the limit on the number of outstanding messages per node is not necessarily realistic. It is a reasonable thing to try, on the assumption that packet acknowledgements travel through the network faster than data packets (for in this case the number of outstanding packets in the network cannot grow

indefinitely). If that assumption breaks down, then certainly we expect to see unusual effects.

In real life, the T9000 architecture requires that buffer space for virtual channels be preallocated. There can only be a finite number of virtual channels on each T9000, and there can be at most one outstanding acknowledgement packet for each such channel at once. In this more realistic model, can we still reproduce the effect of nodes being unable to send any useful traffic?

It is important to recognise that there are two distinct issues here:

- the constant presence of packets on the T9000 acknowledgement packet queue, suppressing outgoing data packets; and
- the growth without limit of the acknowledgement queue.

There is a difference between these two. Unbounded growth of the acknowledgement queue implies that we will never be able to send any data packets, but the reverse implication does not hold: data starvation can occur whenever we have a constantly busy acknowledgement queue, even without unbounded growth of that queue. The unbounded acknowledgement queue growth may be impossible in a realistic workload model, but now that we have identified why that model results in starvation at some nodes, we may try to reproduce the effect of a constantly active acknowledgement queue in a realistic workload.

Recall that in figure 5.4, we were comparing workloads in which either the T9000 send buffer, or the total number of outstanding packets per process, was limited to 16. Let us now look at the latter, more realistic case, but with a larger limit on the outstanding packets. In figure 5.8 we reproduce the load maps for $c = 1000$ on this workload but with workload bounds of 16, 32 and 64 packets outstanding per process at once. We can see that the load surface is indeed sensitive to the number of outstanding packets we allow at once, and at the highest end, with up to 128 packets queued per process in the network at any time, we observe the same effects we were seeing on the unrealistic load before: translational symmetry along y, and starvation of processes along the edge of the grid at $x = 0$ and $x = 15$.

In other words, this effect is not only an artifact of an artificially high-load workload with no relevance to real transputer architectures. Using the normal transputer communications model, with a large but finite number of channels per processor, we may well see starvation of processes due to the priority of acknowledgement packets over data packets.

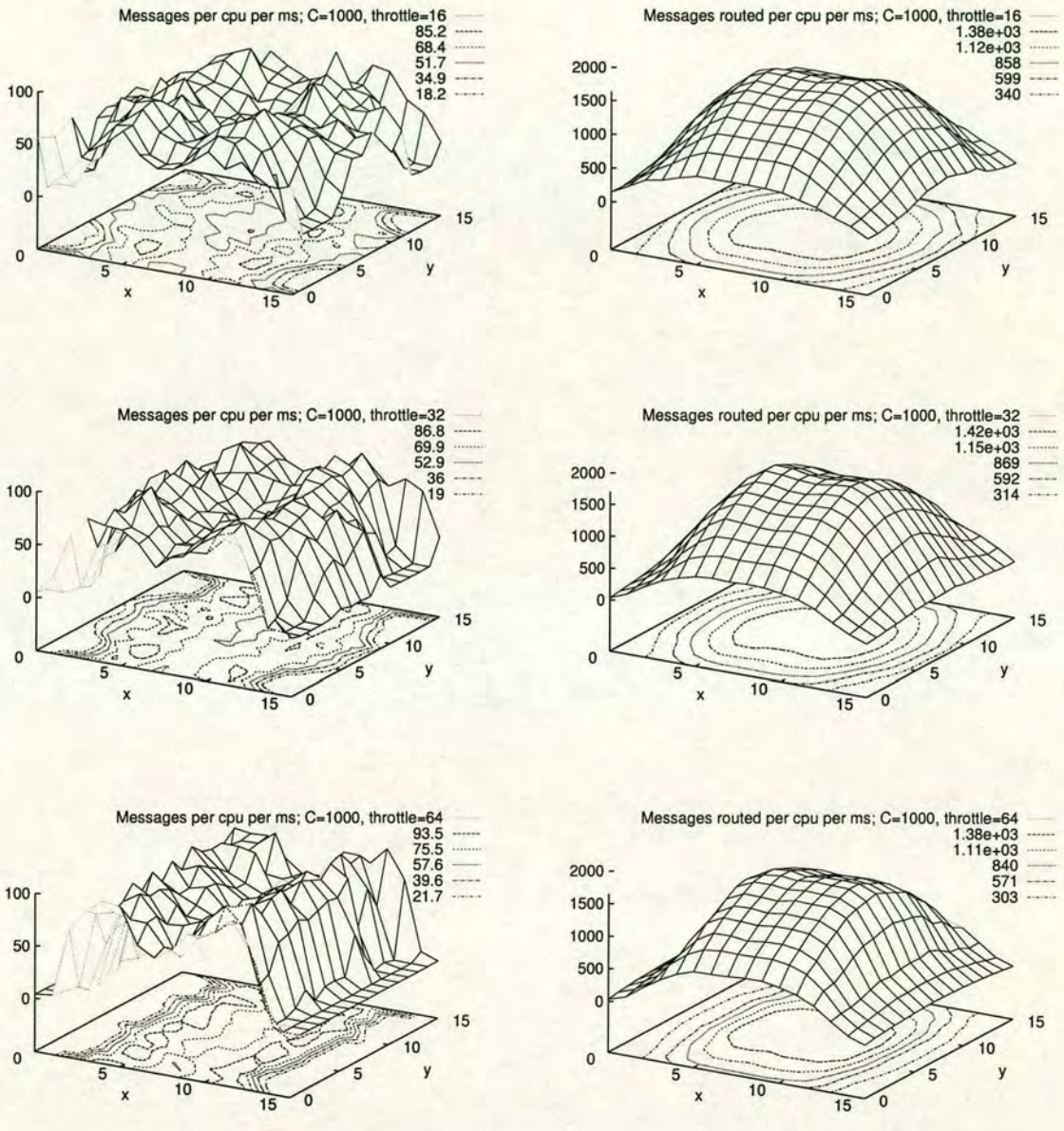


Figure 5.8: Workload performance and router traffic per node: throttled to 16, 32 and 64 outstanding packets per process

There is one last thing we can do to verify that this acknowledgement prioritising is in fact responsible for the effects we are seeing here, and that is to modify our communication model to eliminate that prioritising. Our simulator supports this functionality: the "`--noackpri`" switch commands the simulator to place both data and acknowledgement packets in the same T9000 output queue, so that the communications code does not distinguish between them and they are routed in true first-in, first-out order. Figure 5.9 shows the results when we do this: it gives the load maps of exactly the same experiments as we ran in Figure 5.8, only without acknowledgement packet prioritising. That single change to the simulation is enough to completely eliminate the bands of starved processes at $x = 0$ and $x = 15$, confirming our hypothesis that the buildup of acknowledgement packets is indeed responsible for starving processes of data sending bandwidth under high load conditions.

5.4.4.3 Saturation Performance: Summary

To conclude this section on performance in saturated- or nearly-saturated systems, we can summarise by stating that under very high load, there are a number of unanticipated effects which can arise from the specific way in which packets interact with each other.

First of all, we saw that the normally benign behaviour of prioritising acknowledgement packets over data packets can, under extreme load, result in starvation of processes unable to send data packets: the data's injection into the network can be stalled indefinitely by a constant stream of acknowledgement packets. This is most likely to be observed in a system under non-uniform load (such as a grid topology with its edge effects), where packets in one direction experience greater congestion than those in the opposite direction.

Conclusion 5.1 *The implicit ordering of one type of packet over another by a component of a multicomputer interconnect has the capacity to starve packets of another type. The T9000 architecture can exhibit this effect by stalling data packet delivery indefinitely under conditions of sufficiently high load.*

Secondly, we saw that the x/y asymmetry inherent in greedy dimension-order routing can cause more congestion when routing packets in one dimension rather than another, and that this asymmetry can be reflected in the map of work achieved by processes at different locations in the network. Indeed, this effect can be so great as to dominate the overall shape of the network's load surface.

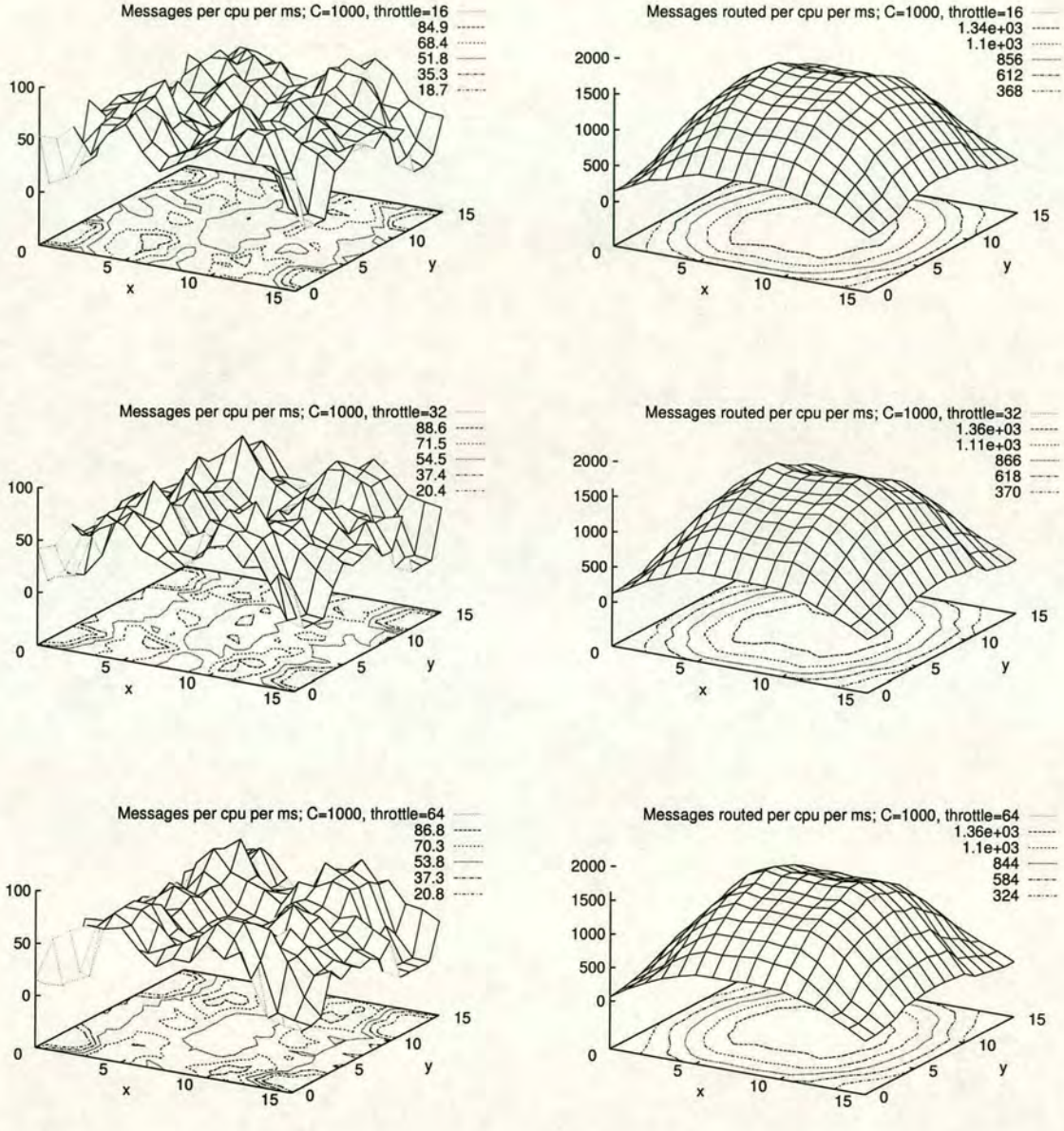


Figure 5.9: Workload performance and router traffic per node: throttled to 16, 32 and 64 outstanding packets per process, but with no acknowledgement prioritising

Conclusion 5.2 *The precise order in which individual packets traverse their routing links can give rise to gross overall structures in the multicomputer load surface. We cannot make the simplifying assumption that only the overall link load matters without losing the power to predict these effects.*

Finally, we have seen that the assumption that we can realistically probe network behaviour by simulating a workload in which messages are injected fully independently at each node has an unexpected complication if all those messages require acknowledgements to be sent in return: we cannot successfully throttle the network load simply by putting a bound on the number of packets buffered at each point of packet injection in the network. There is an interaction between those injected data packets and the network's internally-generated acknowledgements: by explicitly generating a packet at one point in the network we are also implicitly generating an acknowledgement elsewhere. Placing a limit on the size of our outgoing data queues at any point ignores the fact that we also need to limit the size of the acknowledgement queues at those nodes where the data is being received.

Conclusion 5.3 *If there are implicit dependencies between packets in a network, such as the transputer's association of acknowledgement packets with data packets, then we cannot necessarily assume that we can inject messages at each node independently. The implicit generation of new packets at other nodes in response to our injected packets creates a dependency between the loads at and between those two nodes which we may not be able safely to ignore.*

5.5 Measuring Contention

So far, we have been looking at defining our workload and have observed several effects which are related to interactions between packets but which do not really constitute contention. We have seen the impact of ignoring certain dependencies between packets, but we have not concentrated on the overall performance degradation due to contention within the network.

In the following sections, we will begin looking at the measurement of contention costs and of system performance when contention is present in the interconnection network. In particular, we will be looking very carefully at the definition of "performance" as applied to the routing network. It is common to view a routing network as a means of carrying multiple individual messages from point to point, and from this point of view the key characteristic of network performance may be the amount of data it can carry between two points, or the

latency between one point and another, performance being defined in terms of the delivery of individual messages.

However, these performance characteristics do not necessarily tell us how fast some specific application will run on the network. In the following experiments, we will be looking at ways of observing separately the network performance and the workload performance, and of the cost to each of contention.

Contention is, by definition, the blocking of a packet in a network buffer while waiting for an output link to become available to forward it to its next hop. This is a relatively straightforward factor to measure: in the first instance, a fairly simple metric such as the total time each packet spends in this blocked state may serve as an indicator of how much contention is taking place.

If our overall objective is to improve the delivery times of packets through the network, then the net effect of contention on individual delivery times may be a sufficient measurement not only of the presence of contention but also of its effect. However, in both the globally and locally synchronous workload models defined above, our workload's processes will block regularly waiting for a specific packet to arrive before proceeding. This extra synchronisation means that packet delivery times may not be a sufficient measurement of overall performance, and that communication performance degradation due to contention does not necessary equate directly with workload performance degradation.

In particular, the following effects may be present:

Latency hiding At moderate loads, and especially given more than one process per processor, the effects of contention may be sufficiently low that they have no effect at all on overall application performance. If a packet is delayed, there may be sufficient work left to do on the same processor that overall performance is unaffected.

Latency sensitivity At high loads, however—especially on loosely synchronous workloads—we may find that the delaying of the arrival of one message at a process may itself delay the sending of the next message from that process. Increases in overall latency may result in a knock-on delaying effect which the workload cannot hide.

One of the tools we have available in our simulation environment is the ability to selectively disable communication contention, routing packets through the network without interfering with other packets travelling the same paths at the same time. By repeating a simulation experiment with all contention effects disabled, we may hope to quantify the total net effect of contention on the performance

of the system as a whole. It is a more simple matter to observe the distribution of delivery times of individual messages. Putting both of these observations together under varying conditions of network load, we may start to investigate the differences between the effect of contention on individual messages, and its effect on the overall performance of the workload.

5.5.1 Contention and Throttling

Now that we are beginning to make quantitative measurements of contention, we need to make one very important distinction. The fact that any given network link can carry only a single packet, and that other packets may have to queue up behind it, has consequences for both the packet's sender and its receiver. If a single packet is injected into the network and its progress is blocked at multiple points along the way, then contention is at work and it is the receiver who has to wait. This is the effect normally referred to as contention, and it is this effect we have been thinking of as "contention" up until now.

However, there is another important effect of the serialisation of packets through network links, and that is flow control of the packet source. Consider as a simple example the case when two processors on an otherwise idle network are exchanging packets as rapidly as possible. In this case, the serialisation of packets entering the link has no effect on the overall network performance—there being no other traffic on the network, each packet is delivered in the minimum possible time after entering the network. However, contention still has a strong effect on performance: flow control effects working backwards from the first network link in the path connecting the two communicating processors limit the workload's achievable packet transmission rate.

We can expect to see this effect whenever a processor can rapidly generate multiple packets for injection into the network. The individual packets, once they enter the network, may experience no contention, and yet the contention between packets for transmission on the first network link on their path acts as a natural limiter on the transmission rate.

Indeed, when we were describing our initial experiments in section 5.4, one of the most important effects we had to consider was this flow control. When we were looking at performance near saturation, we described how we could "throttle" the workload by limiting in some way the number of outstanding messages it was allowed to generate. This mechanism for limiting outgoing data rate acts in such a way as to prevent messages from being injected into the network faster than the network can consume those messages, but that rate depends very much on

the amount of contention present. This is the very essence of flow control.

It makes sense, therefore, to differentiate between these two effects of contention:

- the limiting of the rate at which packets can enter the network, and
- the impact on packet transmission times due to other unrelated activity on the network.

In other words, we distinguish the effect that contention has on packet sources from the effect on packet destinations. We use the term *throttling* to refer explicitly to the first effect:

Definition 5.2 *Throttling is defined as contention between two packets competing for output on the same DS-link, where both are queued waiting for their first hop within the routing network (not counting the hop between the T9000 and the C104 router).*

There is an important reason for making this distinction. If we ignore throttling, then we effectively allow arbitrarily high message generation rates at each node. We want to be able to observe the effects of interference between packets travelling different paths through the network, but unless we distinguish this internal contention from workload throttling, we are in danger of measuring nothing more than the bottleneck of packet injection into the network.

One last point about contention is worth remembering explicitly here: there is potential for contention to occur every single time a packet passes through a router on its way through a multicomputer network. The potential for contention therefore increases as we increase the number of hops in a packet's path. In the experiments that follow, therefore, provoking contention is not only a matter of achieving high levels of network load as measured by, say, network utilisation: we also need to keep in mind that controlling path length will be an important factor in observing contention.

5.5.2 Contention-Free Communication Models

We have already defined the synthetic workload we are going to be running in this chapter. Let us now look at the communications model which will be used in the simulated interconnection network. Remember that we want to be able to observe the cost of contention in our network: we are proposing to achieve this by seeing the difference in performance when we disable contention entirely. In

this section, we describe in detail how we can modify our communications model to this end.

We define three different models for interaction between packets travelling through the network. In any given run of the simulator, exactly one of the following must be chosen:

Full contention model This is the basic, realistic model which observes contention fully, as discussed in chapter 4.

Contention-free model In the contention-free model, all contention between packets is ignored entirely. Note that the timings for any single packet passing through the communications network are unaffected: the network's bandwidth and point-to-point latency are still modeled correctly.

The difference in this model only occurs when two packets attempt to route along the same transputer DS-link at once. Whereas the full contention model forces packets to wait at the C104 crossbar switch until an appropriate output channel becomes free, the contention-free model allows packets to be transmitted along a DS-link even if it is already carrying another packet. The net effect is precisely as if each DS-link in the network were replaced by an infinitely wide fat link; individual packets cannot be carried any faster, but under conditions of contention, the overall network capacity is increased. Every packet achieves its best-case delivery time regardless of the existing load on the network.

Throttled contention-free model In the throttled contention-free communication model, contention is largely ignored: as in the fully contention-free model, most packets proceed without regard to any other packets using the same DS-link at the same time, eliminating the effects of contention on the overall workload performance. However, in the special case of the throttled model, we *do* observe throttling as defined above. Any packets routed out on any DS-link for their first hop through the network are placed into a separate per-DS-link queue, and these packets are serialised with respect to each other so that only one such packet can be in active transmission along the DS-link at once. (There is no interference between packets on this queue and any other packets passing through the network.)

If the fully contention-free model gives the effect of a network of infinitely wide fat links, the throttled model is equivalent to a similar network but where one single link out of each fat link is reserved especially for first-hop

packets, and all such first-hop packets must queue for transmission down that single DS-link.

Why do we choose precisely this definition of throttled contention-free communications? At first glance it may seem more natural to throttle packets entering the network by preserving the flow-control properties of the communication links between processing elements and routing elements in the network, ignoring contention altogether for communication between routing elements.

However, this model of communication suffers from hopelessly unrealistic throttling, due to the presence of parallel links between the T9000s and C104s in our model network. Consider a pair of processes communicating with each other from adjacent processors. If we eliminated all contention within the network from our model, then in this case the processes can send data to each other as fast as the T9000s can carry traffic down their fat links to the network. This completely fails to model the real flow control effect expected when we send data packets down a single path through the network. Instead of one packet being sent at a time down that path, we have as many packets at once as there are links between each T9000 and the network.

The problem is that by throttling data in the T9000, we are enforcing the flow control delay at the point at which packets enter the T9000-to-C104 link, not the point at which they enter the internal C104 network itself. By serialising packets as they wait for transmission on their first hop inside the C104 internal network, instead of in the T9000, we are applying flow control just as packets enter that network.

5.5.3 Performance with Contention Effects Eliminated

We have now fully defined both the workload and the communications models we will be using to probe contention effects, so we can begin to look at actual simulated performance of the workloads under these models. At first, all we want to do here is to demonstrate our ability to pick out the cost of contention on network and workload performance.

Figures 5.10–5.12 present the observed network performance, measured by the number of successful messages transmitted per CPU per millisecond, against c, the workload compute period. We plot each of the three workload types separately; for each workload, the results obtained for the workload when run under each of the three communications models in section 5.5.2 are overlayed. In these figures, the right hand graphs are merely reproductions of the left hand data with a restricted X axis.

Figure 5.10: The loosely synchronous workload

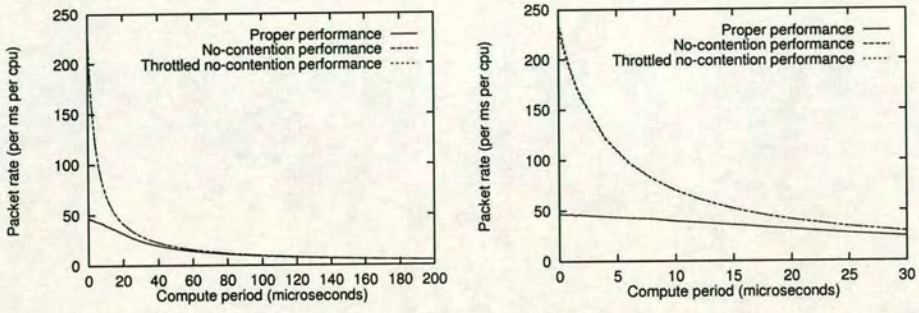


Figure 5.11: The asynchronous workload with blocking sends

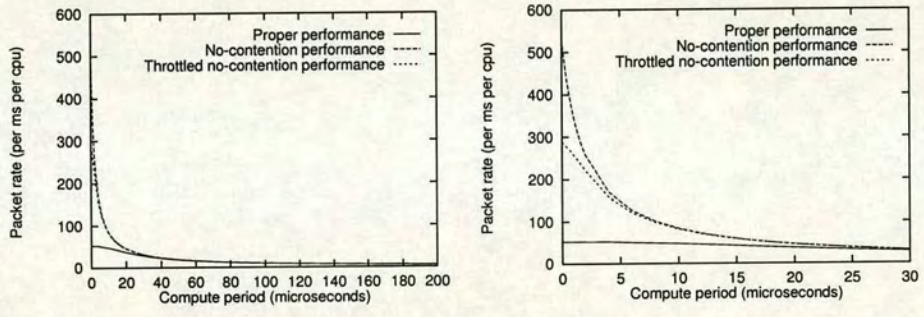
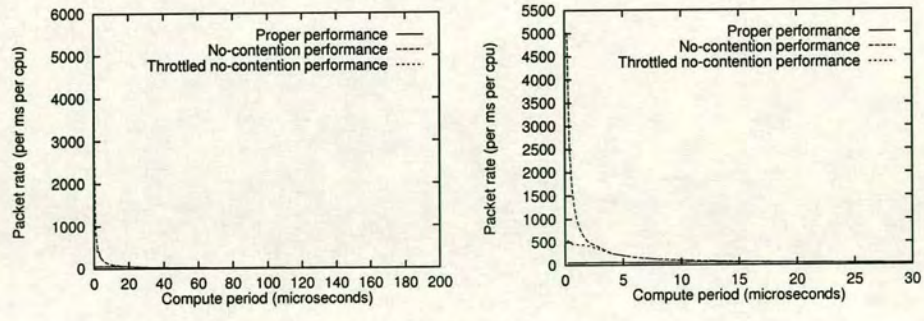


Figure 5.12: The fully asynchronous workload



5.5.3.1 Throttled and un-throttled performance

We start by looking at the distinction between throttling and contention. In the first case—the loosely-synchronous workload, in figure 5.10—perhaps the most obvious feature is that there is in fact no difference at all between the performance of the fully contention-free and the throttled contention-free models. This is the workload in which every process performs a cycle of both sending and receiving messages, and then performing a fixed period of computation. The rate at which the workload can progress is naturally limited by the rate at which messages can arrive at the process, and so it is not surprising to see that in this case imposing extra throttling within the network itself has no effect on workload throughput. The very nature of the workload practically guarantees that it cannot generate messages faster than the network can carry them.

This contrasts with the behaviour of the workloads which do not wait for their incoming messages on each iteration. In the asynchronous workload with blocking sends, the workload is free to continue once the acknowledgement from its last transmitted message has arrived, and in the T9000 architecture the recipient of a message is free to send that acknowledgement as soon as just the header of the last packet has arrived at its destination. There is nothing to stop that acknowledgement arriving before the previous message has been fully sent, and indeed this property is essential to maintain throughput when we are sending large sequences of packets to the same recipient. As long as an acknowledgement can arrive while the original packet is still being transmitted, we are open to the possibility of generating messages faster than they can be consumed by the network. Of course, the fully asynchronous workload does not have even this small level of inherent throttling, and there is no limit to the rate at which it could theoretically inject messages into the network.

This is why the distinction between throttling and contention is important: if we ignore throttling, then when we introduce our contention-free model, we are measuring not only the hypothetical performance improvement possible by eliminating contention *inside* the communications network; we are also measuring the improvement obtained by being allowed to inject arbitrarily high numbers of messages *into* the network in the first place.

The important point to emphasise here is that what we want to study is the performance of the communication network itself, under a given load. To observe the true cost of contention, we want to be able to disable contention inside the network, but to maintain the same level of packet load being placed on the network by the workload. However, at high loads and for our asynchronous

workloads, throttling becomes the main mechanism by which the applied load is limited. Throttling *only* affects the rate at which new packets can appear inside the communications network; it does not at all apply to packets which are already inside the network.

We can see this in figures 5.11 and 5.12, corresponding to the asynchronous workloads with blocking- and non-blocking sends. In these cases, we can see a sharp distinction at the highest levels of applied load between performance under the contention-free and the throttled models. This is particularly pronounced—and expected—under the fully asynchronous workload. In this case, the workload's ability to generate new messages is fully independent of the ability of the network to transport them. The difference between the throttled and non-throttled performance curves tells us just how much performance gain we would see if we disabled throttling, and the answer is that at the highest levels of workload activity, throttling is by far the largest effect limiting performance.

In the locally synchronous workload, we start to see this divergence at around $c \leq 5$, but by the region $c = 0$ we are achieving about 290 communications per CPU per millisecond in the throttled contention-free model against 530 for the non-throttled model, an increase of 85%. For the fully asynchronous workload, the corresponding figures are 544 against 5000, an increase of over 800% with throttling disabled.

Conclusion 5.4 *At high loads there is a major difference difference between the two effects of contention and throttling. We have demonstrated that this distinction can have a substantial impact on expected performance, and we conclude that we* must *make this distinction if we are to hope to understand the impact of performance on a real system under heavy load.*

In the light of this observation, we will concentrate exclusively on throttled performance from now on. This way, we maintain as much as possible of the pattern of messages being injected into the network when we move between the full and throttled, contention-free communications models: the difference between these two models is confined entirely to the behaviour of packets already inside the network.

Our second conclusion at this point is that although flow control may be an important factor in understanding performance, that flow control does not necessarily have to arise from throttling, that is from the behaviour at the sending node in the communications network. In the case of our loosely synchronous workload, we can observe no difference between its performance under the throttled and un-throttled contention-free models.

5.6 Workload Performance versus Communications Performance

So far we have been looking almost exclusively at workload performance: the aggregate amount of work being achieved by the all of the processes running on our system. We now consider the other main measure of system performance which interests us in this chapter: the performance not of the workload, but of the underlying network itself, as measured by the time taken for that network to carry individual packets between source and destination. We will be using precisely the same synthetic workloads for this as we used earlier to look at workload performance.

In these next experiments, measuring contention would appear to be straightforward. If we were running adaptive routing, a blocked packet might have the option of selecting a different output link to look for an alternative route to its destination, but under the greedy routing in use here, a packet which cannot be forwarded due to contention has no option but to wait, remaining queued until its desired output link becomes free. With greedy routing, then, the path taken by any given packet between source and destination is constant, and so therefore is the ideal, contention-free time taken to route that packet. If the communication routes are constant over our experiments, then any changes in the time spent by packets inside the network from one experiment to the next can only be a result of contention.

This allows us to try to observe the effect contention inside the network by looking at packet delivery times. Inherent in the definition of our synthetic workload is the fact that although the rate of new message generation is variable, the distribution of possible messages destinations from any given process is not. The average distance traveled per packet must be invariant under changes in workload type or throughput, except for one possible effect: in the non-loosely-synchronous workloads, it is conceivable that processes near the edge of the system might be affecting the average packet path length due to a combination of two effects:

- The grid of possible packet destinations is truncated at the grid's edges, so that processes near the edge may have different average neighbour distances to processes in the center of the grid; and
- without any global synchronisation between processes, it may be possible for these edge processes to run at different rates relative to the more central processes according to the applied workload.

It is worth checking therefore that we are in fact dealing with the same average path lengths in each experiment to make sure that this is not an issue. Taken over the entire set of experiments we are about to present, including each applied load level for each workload, we find a mean path length of 9.50 hops, with a standard deviation in the distribution of $\pm 0.29\%$ and a bounding range of $[-0.83\%, +0.39\%]$. Given that we have only run the experiments as a whole to an accuracy of $\pm 1\%$ at the 95% confidence interval, average path length does not appear to vary significantly over the experiment.

5.6.1 Measuring Packet Delivery Times

We can now look at the actual time taken to send data through the network at various load levels, for each of the three workloads. We are forced now to consider exactly what we want to measure here. There are a number of points at which we could begin and end our stopwatch to time the data's passage, including

- The initial generation of the message by a process;
- The queuing of a packet for transmission inside a T9000; or
- The initial injection of the packet into the communication network itself, marked either by the start of transmission on its first T9000 output link or its arrival at its first C104 input link.

as definitions of the staring point, and

- complete arrival of a packet at its destination T9000, or
- delivery of the message to its receiving process

as end points. Our easiest choice is to eliminate message (as opposed to packet) creation and arrival as useful metrics: the transmission of multi-packet messages, in particular, involves multiple packet transmissions from both ends of the virtual circuit, and may not tell us a great deal about the time it takes for a single packet to travel through the network. Message delivery times also depend on what is happening in the workload as much as the state of the network underneath, since in the transputer architecture, we do not deliver a message to a process, or even acknowledge the first packet of a multi-packet message, until the receiving process has started to read data from the virtual circuit. Our simulator does allow us to collect message delivery times, but we will ignore these for now.

So, we want to time the transmission of individual packets, not entire messages, through our communication network in order to get a good idea of how

much contention is occurring in that network. To that end, in figures 5.13 and 5.14, we are measuring, respectively, the total time taken for a data or acknowledgement packet newly generated by a T9000 to be completely delivered to its destination T9000. This time includes time spent in the source T9000 awaiting injection plus the full time spent within the network.

The first, most obvious property of these results is that our free-rein workload, the fully asynchronous workload, is generating very substantially longer transport times at high load levels than the other workloads. The rise starts at $c \leq 30$, and by the region $c \leq 15$ we are seeing over 700% longer transport times in the fully asynchronous workload than in the other workloads. In figure 5.3, we saw a difference of up to 40% in the throughput achieved by the three workloads at this level of applied load, but no sign of anything like the large degree of variation between the workloads that figures 5.13 and 5.14 display.

At this point we need to recall the distinction already made between throttling and contention. We have already observed that the fully asynchronous workload is able to saturate our communications network; beyond the saturation point, we can be creating new packets in the T9000 faster than they can be consumed by the network. Although we do use the queue limiting mechanism described in section 5.3.2 to limit the number of messages queued in any single T9000, we can still maintain quite a backlog of such messages.

Once we achieve this saturation, then increasing the rate of new packet injection can only increase the average backlog queue size of packets awaiting output from the T9000. There is now a bottleneck at the point where packets are being injected into the network: we can only offload packets from the T9000 at a certain maximum rate, and attempts to create packets faster than this are throttled by the flow control feedback from the C104 attached to the source T9000. Increasing the size of the backlog in the T9000 cannot increase the rate at which packets are injected into the network, but it *will* increase the time which these packets must spend queued inside the T9000 before they can start being transmitted into the C104 routing network.

By measuring packet transport time from the point of packet creation, we are including in this measure the time spent queued in T9000 output links. This is demonstrated graphically in figure 5.15, where we display the average time each packet spends queued in the T9000 before its first transmission. This time forms part of the throttling effect limiting packet injection rate, and has nothing to do with contention inside the C104 routing network itself.

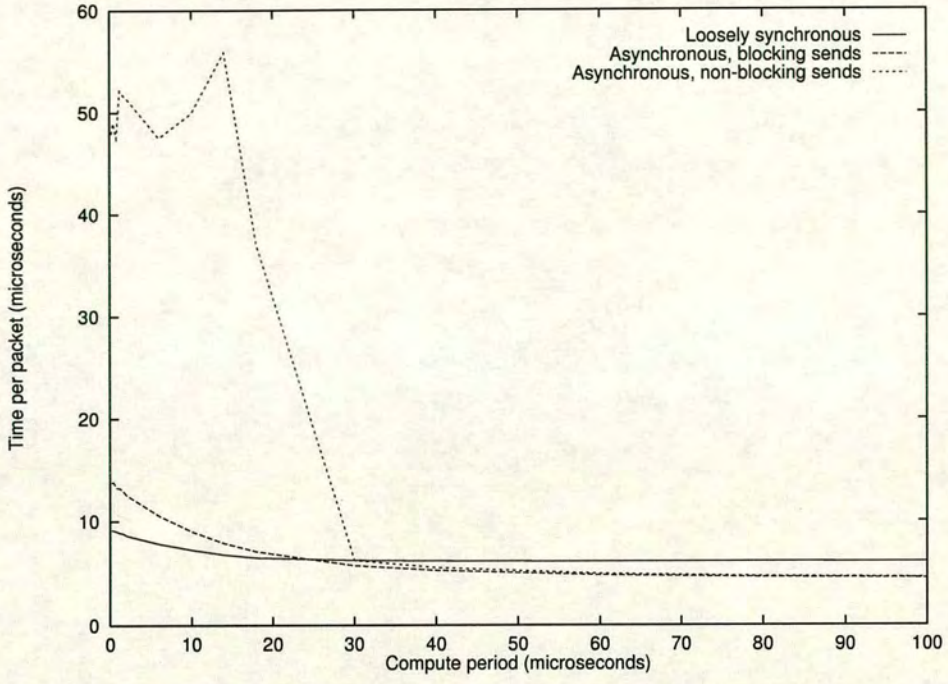


Figure 5.13: T9000 to T9000 data packet transport times:

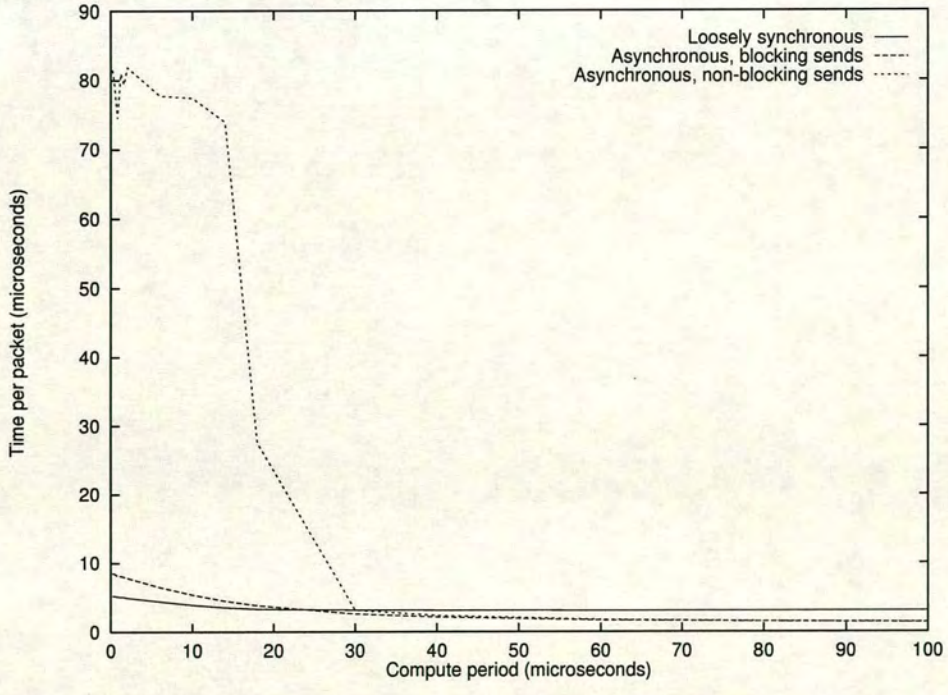


Figure 5.14: T9000 to T9000 acknowledgement packet transport times:

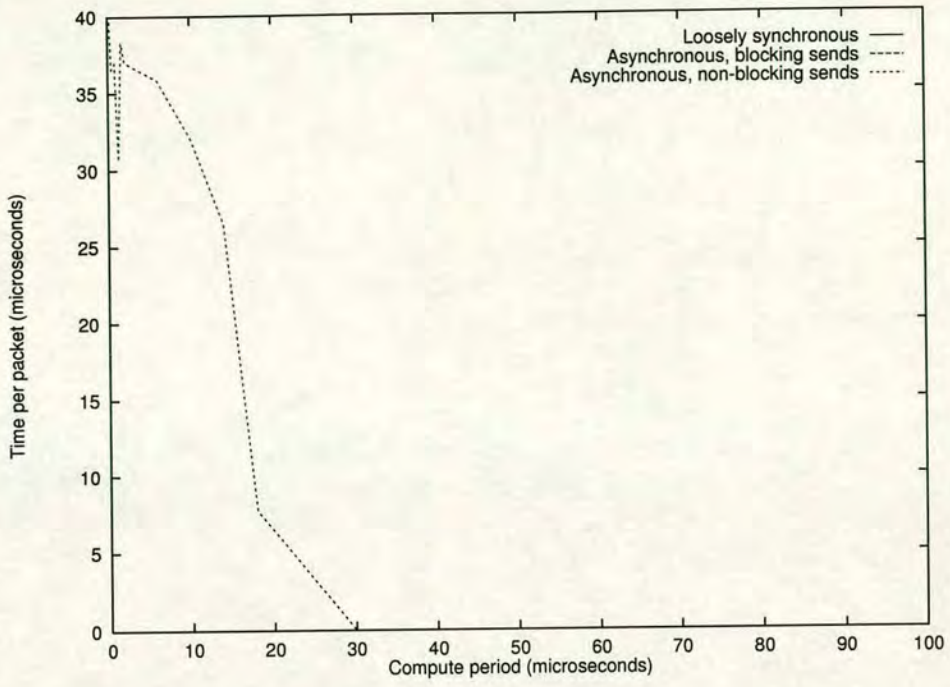


Figure 5.15: Mean time per packet spent in T9000 output queues

We can see in this figure that although this T9000 queueing time is simply absent in the self-throttled workloads (which can only ever permit one outstanding packet per process in the experimental setup under consideration here), for the fully asynchronous workload the situation is rather different. At precisely the load levels where we saw the peak in transport times in figures 5.13 and 5.14, figure 5.15 displays the onset of long queueing delays in the T9000. If we want to eliminate this effect, then we need to start timing packet transmissions from the point at which they enter the routing network—ie, the time of their initial transmission from T9000 to C104—rather than simply the time the packets were generated.

Unfortunately, throttling is still more complex than this. We have already described that our network architecture in this experiment consists of a grid of singly-connected C104 routers each connected to a T9000 via a fat link of width 4. At any point in time, up to four packets may be queued at a C104's injection links, but of those four packets it is likely (given a random destination for each packet) that they will be competing for fewer than the four output links from the C104. If the T9000 has used its wide link to inject multiple packets destined for a single router link in the network, then we are still seeing packets being delayed for reasons other than contention within the network — although these competing packets have all arrived at the first router along their desired path, none of them

have yet been transmitted along any link internal to the routing network itself.

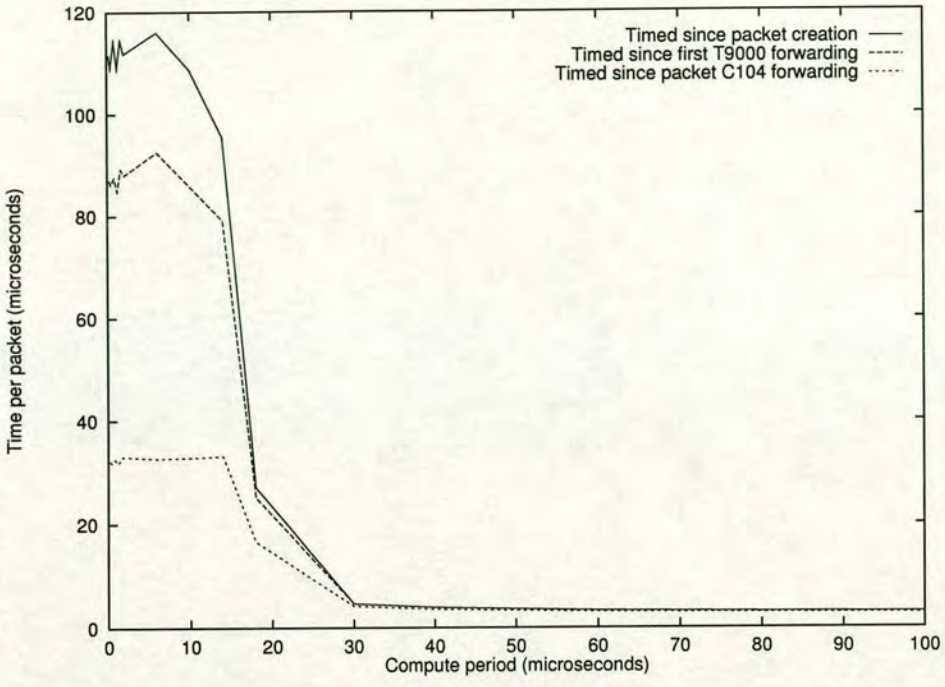


Figure 5.16: Mean transmission time per packet timed from various starting points

Figure 5.16 shows the packet delivery times for our fully asynchronous workload again, still timed up until the point at which the complete packet is delivered to its destination T9000 but this time beginning the timings from:

- The time of packet creation;
- The time at which the packet begins to be transmitted by its source T9000 towards the routing network; and
- The time at which the packet first starts to be transmitted along a DS-link internal to the routing network.

The results are quite striking. We can see that even at the highest levels of load on the network, the time spent crossing the network itself accounts for only a fraction of the lifetime of an average packet. By far the largest part of the packet's lifetime is spent blocked in one of the injection links in the first router it encounters, and this is a throttling effect, not true internal network contention.

Looking at the very highest load levels here, the packet lifetimes measured from these three starting points are listed in table 5.3. We can see that at this load level, on average only 32% of a packet's lifetime remains once it has begun to

be forwarded within the routing network, despite the fact that only one hop in the packet's history is behind it at that point out of an average distance of 9.5 hops traveled per packet. Of the throttling effect visible, only 13% of the packet's lifespan is accounted for in the T9000's output queues; fully 55% of its lifetime elapses between the packet's injection into the routing network commencing and its first forwarding within the network.

Measured from:	Lifetime (μs)	%age of total lifetime
Creation	111.6	100%
First T9 output	87.0	87%
First C104 output	32.2	32%

Table 5.3: Average packet lifetimes at saturation

From this we gain further support for our previous assertion that throttling and internal network contention are two very different effects, and that any investigation of contention levels within a network must address this distinction, especially at high workload packet generation rates, or risk the observations being overwhelmed by effects taking place at the boundary between the network and the processing elements, rather than fully within the network.

Definition 5.3 *The* routed lifetime *of a packet is defined as the lifetime of the packet from the point of its first transmission over a DS-link internal to the network (ie. not an injection link) to its complete delivery to the destination processing node.*

With this definition, we can also now say:

Conclusion 5.5 *The most appropriate metric by which to measure the amount of contention taking place within a network is the routed lifetime of a packet.*

5.7 Conclusions

In this chapter we have looked at the measurement of network performance, both in terms of achieved throughput and in terms of time taken to transmit a single message. Our conclusions are both positive and negative. We have identified a number of potential pitfalls which can disturb our attempts to observe and quantify contention and performance. On the other hand, we have seen that, given care in choosing our performance metrics, our simulator provides us with a tool which does enable contention and its effect on workload performance to be measured.

To begin with the problems encountered, we found that:

- Multicomputer network performance is not simple. We have identified at least two specific instances of effects where the precise details of the interactions between specific packets can affect the gross behaviour of the network: the prioritising of acknowledgement packets over data packets, and the order in which the greedy dimension-order router prioritises dimensions (conclusions 5.1 and 5.2).

- We cannot necessarily assume that packets are generated independently at each node when modelling the network load. We have seen that the presence of automatically-generated acknowledgement packets in the network can result in the implicit injection into our network of packets at a rate dictated by conditions elsewhere in the network, and that we ignore these packets at our peril (conclusion 5.3).

The main conclusion that we draw from these problems is that when modelling multicomputers under heavy load, a simulator is a very valuable tool to have. There are methodologies which allow us to deal with complex performance modelling scenarios by numerically attacking the time-averaged behaviour, but the fact that we need to model specific interactions between packets at such a level of detail, combined with the fact that we cannot reliably assume that packet generation processes are independent and exponential, means that we cannot safely make the simplifying assumptions needed to use these models. At the very least, we need a more accurate way of modelling packet interactions in order to validate any simplified models we may come up with.

But we have done more than to simply discover obstacles. We have laid the foundation for the measurement, by simulation, of the presence of contention in a multicomputer, and of its effect both on the performance of the network and of the workload running over that network. In particular:

- We have defined a synthetic workload which will allow us to explore different classes of workload synchronisation properties. We have proposed that ordering constraints in the workload may affect the workload's overall sensitivity to increased contention, and the workload defined in definition 5.1 will allow us control over those ordering constraints.

- We have defined two performance metrics—the rate of successful packet injection by the workload, and the packet routed lifetime (definition 5.3)—by which to measure the performance of our workload and of our network.

- We have identified throttling (definition 5.2) as a special case of contention which needs careful separate treatment.

- We have defined three separate communication models for the simulated interconnection network—the full, contention-free and throttled contention-free models (§5.5.2)—by which we can selectively disable contention and so measure just how much performance is being lost due to that contention.

Chapter 6

Communication and Workload Performance

6.1 Introduction

In chapter 5, we looked at the problems of measuring the behaviour of multicomputer routing networks at high loads, especially at or near saturation. We have shown that we can measure contention carefully, in that we can:

- distinguish between internal contention inside our communications network, and flow control at the point of message injection;
- measure separately the *presence* of contention, in terms of message delivery times ("communication performance"), and the *effect* of that contention on the running application ("workload performance"); and
- quantify the *cost* of contention relative to the equivalent workload without contention present.

However, in most of the previous chapter we looked at these measurements in isolation: we considered communications performance and workload performance separately. To justify making such a distinction between these two metrics, and (more importantly) to show how our contention-free communication models can allow us to measure contention costs precisely, we now need to compare these two measurements side by side for the same experiments. In this chapter we will be doing precisely that, looking at how different workloads respond differently to contention levels in the network.

Our ultimate aim here is twofold: to demonstrate

- that workloads with different internal synchronisation characteristics react differently to the onset of contention in the communications network; and

- that our methodology—the study of both communication and workload performance, and our use of cost measures which quantify performance loss due to contention—is useful in determining those workload reactions.

6.2 Basic Workload Synchronisation Effects

We will start by looking at the same workloads we have used before, but looking at them in a somewhat different light; the questions here is, just what extra insight can we achieve by using both of our cost metrics and by looking at the relative cost due to contention as well as the absolute performance response?

There are a number of effects that we will find. In particular, we will look at the impact of two properties of the workload:

- the *synchronisation properties* as defined in §5.3.2: we will be looking at all three synchronisation types—loosely-synchronous, and asynchronous with both blocking- and non-blocking sends; and

- *timing* effects, especially for the loosely-synchronous workload in which the overall workload tends to run in lock-step and activity in the network may exhibit bursts of high load rather than being uniform over time.

In both cases, we will see that there can be clear differences between the response of the network to increased load (in terms of longer message delivery times), and the impact of that changed network response on the workload.

To start with, we will use precisely the same workloads that we used in chapter 5. These workloads have been shown to be able to stimulate a very high load on the network (especially when running the fully asynchronous workload), and we have also seen in §5.4 that the three different workload types do show different behaviours under load when contention begins to be a problem.

To summarise, we are using our standard workload in its loosely synchronous, asynchronous with blocking sends, and fully asynchronous forms. According to the parameterisation in definition 5.1, the workload is specified by the parameters $(16, 8, 1, c, 1, 32)$, where c is the inter-communication compute period on each running process, this being the variable we use to control the level of communication load applied to the network. Our underlying platform as before is a 16 by 16 grid of C104s each connected horizontally and vertically by a single bi-directional link with a fat link of width 4 between each C104 and its corresponding T9000 processing element.

6.2.1 Synthetic Workload Performance: Communication and Workload Responses

We will first of all look at the picture which emerges when we study the performance of these three workloads under increasing load when we consider both the communications performance and the workload performance. For now we will look only at the results using the full communication model, rather than the contention-free models from §5.5.2—that is, we are looking at what can be seen when we measure only the overall performance responses, not the relative performance loss due to contention.

Figures 6.1 and 6.2 recall from the previous chapter the achieved rate of packet delivery, and the mean packet *routed lifetime* (definition 5.3), for our workloads at various levels of applied load. We have already looked at the differences between the three workload classes in some detail, so we will not dwell on that here. What we *are* interested in now is the difference in the two performance metrics presented, and more importantly in their differing responses to increased load.

The most obvious feature we see when comparing figures 6.1 and 6.2 is that the second figure, representing packet delivery times, shows an enormous plateau when running the fully asynchronous workload at high loads, and yet this feature is simply not present at all in the packet throughput response.

Recall that the metric in figure 6.2, packet routed lifetime, is defined as the lifetime of a packet from its first forwarding on an internal (non-injection) link in the routing network, until its final delivery. We described in chapter 5 how this metric is a product of only two factors: the path length taken by a packet (the mean of which is constant over this experiment), and the amount of contention experienced by the packet once it has entered the network. A change in the value of this metric is directly attributable to contention in the network.

In short, figure 6.2 is demonstrating a very sharp increase in the levels of contention in the communications network at high loads, but only for the fully asynchronous workload. The increase is not only sudden, it is also large: tables 6.1 and 6.2 enumerate the packet throughput and routed lifetime metrics at various load values. We can see that at the highest load level the average time taken to transmit a packet through the network for the fully asynchronous workload, even after accounting for all throttling effects, is at 32.9μs over four times greater than for the loosely synchronous workload and over three times greater than for the asynchronous workload with blocking sends.

Just as significant, however, is what is absent from this data. Two noticeable features of the packet delivery time data are absent from the throughput rate

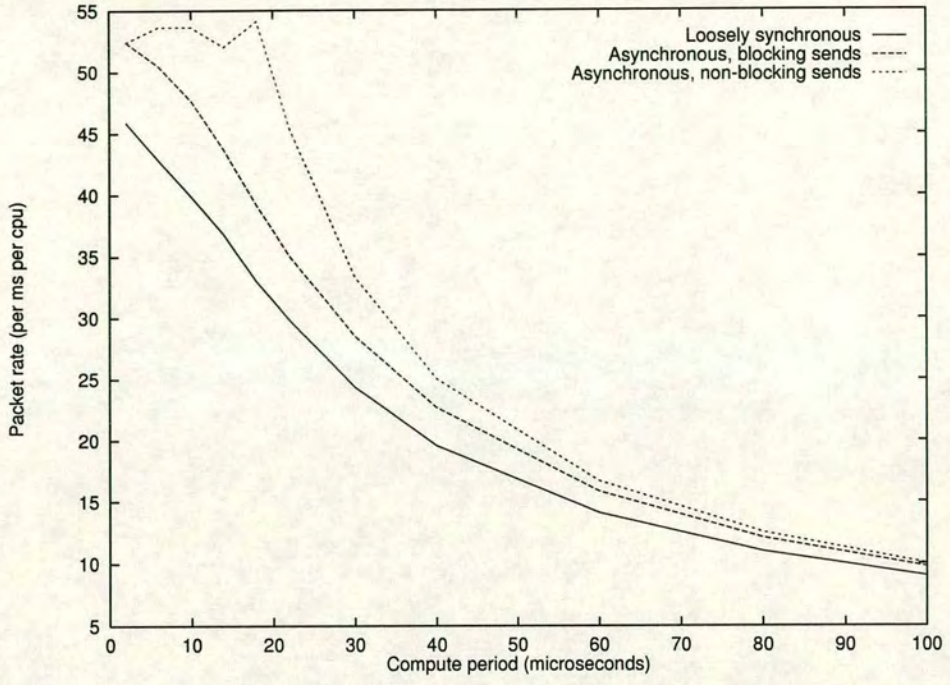


Figure 6.1: Workload throughput against load for each workload type

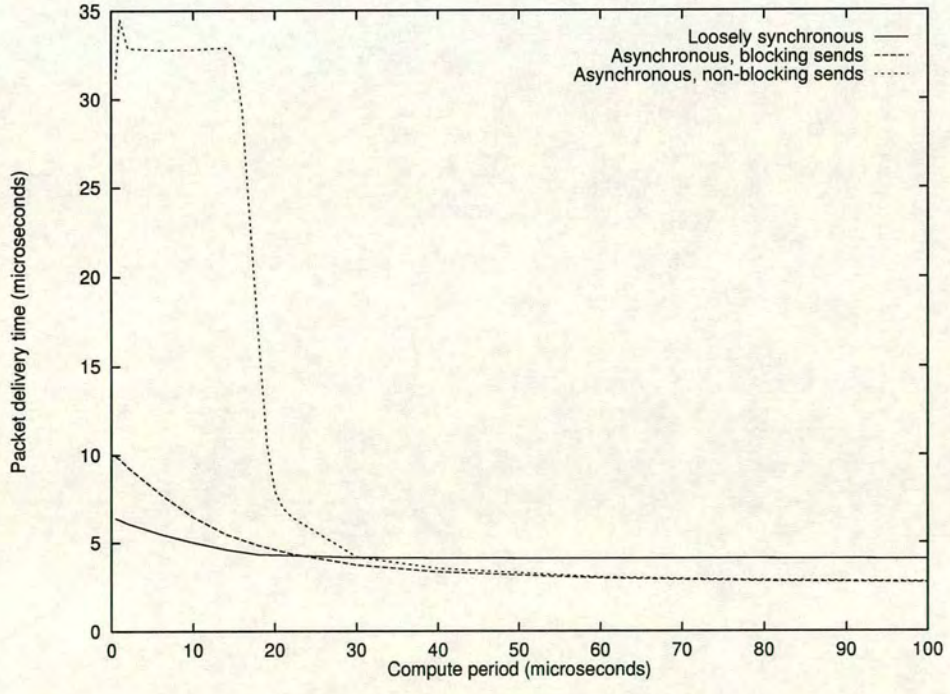


Figure 6.2: Packet delivery time against load for each workload type

Compute period (microseconds)	Workload:		
	Globally sync.	Locally sync.	Asynchronous
2	45.9	52.4	52.4
10	39.9	47.6	53.6
14	36.8	43.7	52.0
18	33.0	39.2	54.1
30	24.3	28.5	33.3
100	9.0	9.7	10.0

Table 6.1: Throughput for the three workloads (packets delivered per CPU per millisecond)

Compute period (microseconds)	Workload:		
	Globally sync.	Locally sync.	Asynchronous
2	6.1	9.3	32.9
10	5.0	6.5	32.8
14	4.6	5.5	32.9
18	4.3	4.9	16.7
30	4.2	3.8	4.2
100	4.1	2.7	2.8

Table 6.2: Packet routed lifetimes for the three workloads (microseconds)

data. First of all, at the point of sharpest increase in the amount of congestion, $14 \leq c \leq 18$, we see the individual packet delivery times approximately double, with almost no change at all in the total number of packets being delivered.

Secondly, there is a great deal of difference in the way that packet delivery times relate to overall throughput in the different workloads. We have seen that at the highest loads, packet delivery time in the fully asynchronous workload is much greater than for the other workloads. However, we do *not* see a commensurate difference in the total amount of traffic sent under the various workloads: the throughput figures for the fully asynchronous workload and the asynchronous workload with blocking sends are almost identical to each other at the highest load (at which point their communication times differ by a factor of over three), and the loosely synchronous workload is achieving only 12% less work than the other two.

The massive increase in packet delivery times for the fully asynchronous workload is certainly by far the largest effect here. To explain it, we need to refer back to figure 5.8, in which we looked at the observed workload performance separately at each CPU on our multicomputer grid, for the same asynchronous workload we are using in this section. When we last looked at this experiment we were more

interested in workload performance than in message transmission times, and we saw that at high loads, there was an uneven distribution of load over our CPU grid: some CPUs were achieving much more useful work than others. Even when the complete starvation of some CPUs' data delivery queues was addressed, there still remained significant non-uniformity in the amount of work being achieved by different processors on the same grid at high load.

The results we see now are entirely consistent with this observation that at high loads, we can experience increasing contention with no increase in the overall aggregate workload performance (recall that in §5.4.4, we even saw a *drop* in workload performance averaged over the network at sufficiently high loads). However, our interest right now is not in the actual dynamics of any one specific experiment, but in our ability to measure these effects. If we were to concentrate purely on either workload performance or the communication latency of our network as a measure of the system, we would have missed these effects entirely.

We have seen that by changing the synchronisation properties of our workload, we can reproduce very different levels of contention in our network without substantially altering the actual overall number of messages being carried by that network, and this leads us to draw two conclusions:

Conclusion 6.1 *We cannot properly understand the response of a multicomputer interconnect without some knowledge of the internal synchronisation properties of the workload running on it;*

and

Conclusion 6.2 *Observing the performance of a multicomputer network purely by looking at the delivery times of individual messages is not necessarily sufficient to describe the network's response: if we are interested in real workloads running on that network, then we need, in addition, some measure of workload achieved performance.*

6.2.2 Contention-Free Performance as a Baseline

We described in chapter 5 not only our ability to measure contention during a simulation run, but also our ability to determine that contention's effect on overall performance by observing performance gained when contention is eliminated. We presented alternative communication models in which we could selectively disable packet contention in our network to measure the relative performance gain. We identified a throttled contention-free model (§5.5.2) which removed contention interactions while still preserving flow control behaviour on the workload, and

proposed this as a way of measuring the relative performance loss in the workload due to communication contention.

In this section, we will utilise this ideal, contention-free performance to provide a baseline against which the actual achieved performance can be measured.

To be specific, we will be re-examining our two main performance metrics, communication performance and achieved performance as measured already in this chapter, but expressing both in terms of performance relative to the performance in the absence of contention. For workload performance, we will be recasting our primary performance metric, packets delivered per CPU per unit time, into a relative form: packets delivered as a percentage of the rate achieved under the ideal, throttled contention-free communication model. Similarly, we will be measuring communication times as the ratio of ideal to realised packet delivery times (still measured in both cases from the point of first entry into an internal link within the routing network).

We introduce two new variables, θ_t and θ_r to describe these ratios for packet transmission times and transmission rates respectively:

Definition 6.1

θ_t is defined as the ratio of the ideal packet transmission time to the achieved packet transmission time; and

θ_r is defined as the ratio of the achieved packet transmission rate to the ideal rate

where the ideal observation is the observation made using simulation under the full communications model, and the ideal observation is that made using the throttled contention-free model under identical input parameters.

Note that both of these variables are defined to be in the range $0 \leq \theta \leq 100\%$. In particular, since the ideal packet rate is expected to be greater than that achieved, and the ideal packet transmission time less than that achieved, we invert the ratio used for θ_t to be ideal:achieved instead of achieved:ideal.

The fact that the definitions of θ_t and θ_r are made in this manner allows these variables to be compared like-for-like. If we were to make a naïve assumption that improved communications time translates linearly into higher throughput, then by these definitions we ought to find $\theta_t = \theta_r$. Any differences between the two variables tell us that there is more going on, and that communication performance within the network, while it may be important, is not sufficient on its own to predict the overall performance of our workload.

A good way to look on these variables is that θ_r represents the *cost* to workload throughput of contention in the network, and θ_t represents the *amount* of

contention present. In fact, θ_t can be considered to be a direct measure of the amount of contention in the network: $\frac{1}{\theta_t}$ is precisely the factor by which packet routed lifetimes are being increased by contention effects.

6.2.2.1 Contention Costs for our Synthetic Workloads

Up to now in this chapter we have looked at the use of our two performance metrics—communication performance and achieved workload performance—both separately and together. Previously, we have also looked (in chapter 5) at the measurement of these two metrics not as absolute values, but as costs relative to an idealised, contention-free communications model. However, we have not yet considered communications and workload performance at the same time, for the same experiments, when studying these relative performance metrics. It is now, finally, time to think about the relative costs of contention to both communication latencies and to workload performance at once.

As before, we will be starting off with the three workloads described above. As before

Figures 6.3–6.5 chart the values of θ_t and θ_r for our three workload types at various levels of load. Presented this way, the data reveals a number of important characteristics of the relationship between workload throughput and packet delivery performance. One of the main points is that:

- at low load levels, significant levels of contention may have little impact on achieved workload performance, but
- under high load, the impact of contention on workload performance may catch up with and overtake its impact on the delivery of individual packets.

We can see this clearly in both the globally synchronous and the locally synchronous workloads (figures 6.3 and 6.4). First of all, when we look at low levels of network load, corresponding to high compute times (c), contention is still present in both workloads and is still hurting packet delivery times. For the globally synchronous workload, at $c = 100\mu s$ contention is still capping packet delivery speed to 58% of its contention-free ideal time, but the workload is still achieving 94% of ideal throughput. For the locally synchronous workload the values are 91% of delivery speed but 99% of workload throughput achieved.

Burst Patterns in Communication Load It is not hard to understand the high rates of contention cost on packet delivery performance here when one remembers that our workloads consist of processes performing a cycle of computa-

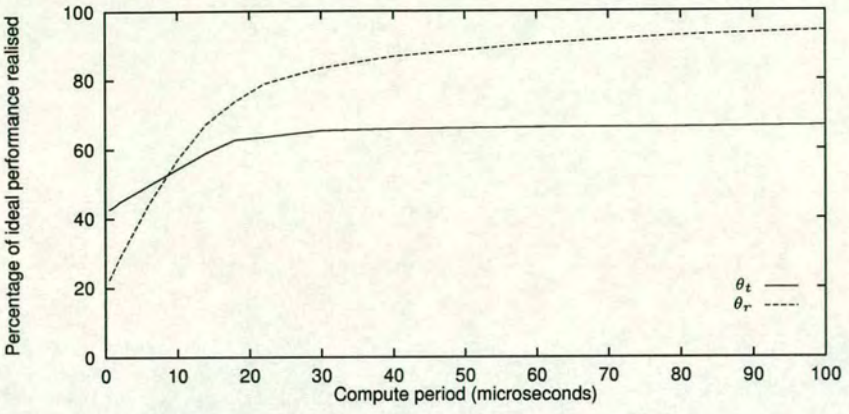


Figure 6.3: Percentage performance achieved: Loosely synchronous workload

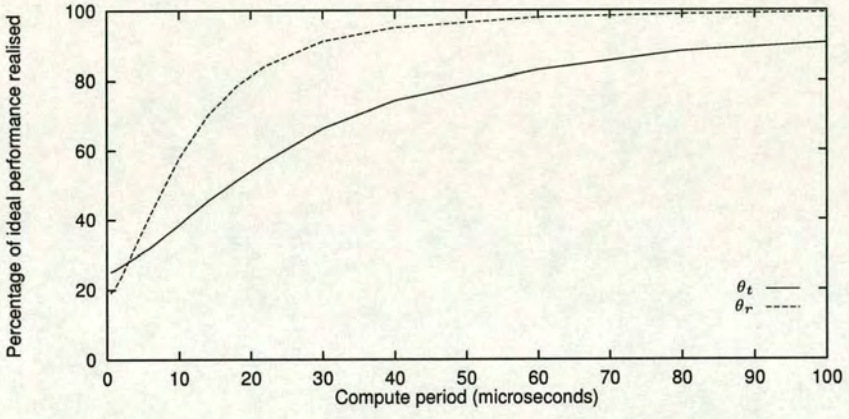


Figure 6.4: Percentage performance achieved: asynchronous workload (blocking sends)

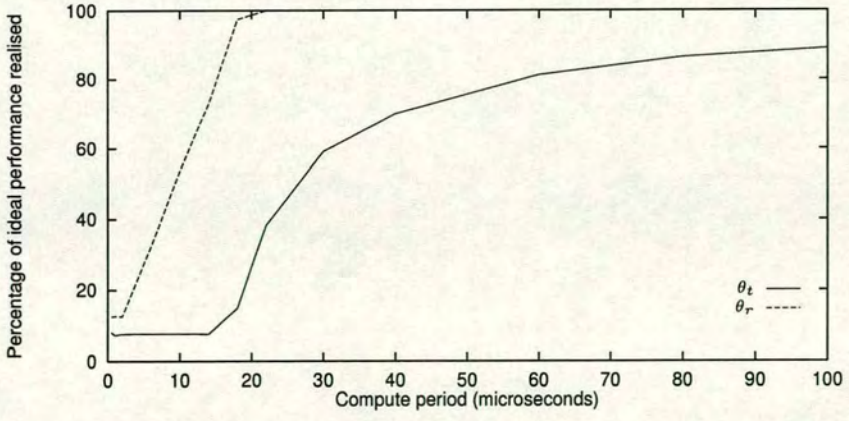


Figure 6.5: Percentage performance achieved: asynchronous workload (non-blocking sends)

tion then communication with (in this case) fixed computation periods between each communication. In the globally synchronous workload in particular, all of our processes tend to run in lock step as far as possible, so at high values of c, we end up with short bursts of communication activity throughout the network followed by relatively long periods of idleness. The only effect which can cause processes to relax this lock-step is variability in the time taken for the communication part of each process iteration to complete, but there is a limit to how far any two processes may drift apart: by their nature, all processes in a loosely synchronous workload *must* run at approximately the same speed over the long term, and the regular synchronisation of processes with their neighbours ensures that there is a limit to how far ahead any process can get of another.

In short, when there are synchronisation forces acting to keep processes in lock-step with each other, our communications network is may be subject to periodic bursts of load. (This need not necessarily apply to the same workload under heavy load: once the communication cost comes to dominate the performance, the repeating compute period will have much less effect on the overall pattern of message injection.) We will look more closely at this effect below.

Latency Sensitivity at High Load On the other hand, under heavy load, we see exactly the opposite effect. Under low load, a modest amount of communication contention has little effect on the workload performance, but we can see clearly in figures 6.3 and 6.4 that, for the loosely-synchronous and asynchronous workload with blocking sends, this is reversed when we increase the communication load.

When we last looked at these workloads and their performance under increasing load, the only effect that was immediately obvious was that as we increased the amount of traffic in the network by decreasing c, both packet transmission times and the achieved workload rate increased monotonically. We were getting uniformly "better" workload performance and worse transmission times as load increased. Nothing from those performance results told us just how much contention was costing us.

Looking at θ_t and θ_r, on the other hand, tells us much more about the actual cost of contention. We can see that in both of these workloads, contention in the network is having a measurable but small effect which starts off growing only slowly as we increase the load, but which by the time we are running under heavy load ($c \leq 20$), grows to have an enormous impact on performance.

This is expected, yes, but without being able to use the contention-free model

as a baseline, it is hard to precisely quantify this impact. With these results available, we can see quite clearly that for these workloads under light load, the impact of contention on individual packet delivery times is much more than its impact on the workload as a whole; but under heavy load, the overall workload performance becomes more sensitive to contention and a given drop in packet delivery times has a much greater impact on the workload. Most importantly, we can finally quantify *how much* more sensitive the workload is to contention under high load.

6.2.3 Distribution of Compute Periods

We mentioned above that in our loosely-synchronous workload, we saw here a new source of contention. We described how we can experience a lock-step between adjacent processes in the process graph which results in a high rate of collision between messages even although the overall network load is fairly low.

There is good reason to want our processes to proceed in lock-step, however, if the process graph is fixed in advance. If processes are proceeding at substantially different rates, then when a pair of adjacent processes synchronise with each other through a blocking communication, then the faster of the two processes much wait idly for the slower to complete its work, reducing the CPU utilisation achieved. This is not just a significant effect, it can indeed be one of the *most* significant factors affecting performance—Candlin and Phillips [CFPS92, CP93] use standard statistical modelling techniques to model the performance of parallel workloads and find that the standard deviation of process compute periods is in many cases the single most significant factor contributing to overall workload performance in their models.

However, we might think that changing the distribution of compute periods, although it will adversely affect workload performance, might be sufficient to eliminate the contention effects resulting from these simultaneous communication events. So far, we have defined the compute period c to be a constant value, but in our original formulation of the workload in definition 5.1, we permitted c to be a random variable too. In this section we look at the results of using a random variable for c: we let $\bar{c}$ be the mean process compute period and σ_c be the standard deviation in c. Do our performance metrics allow us to observe the change?

We will choose four different distributions for the random variable c:

- Constant, $\sigma_c = 0$;
- Normally distributed, $\sigma_c = \bar{c} * 10\%$;

- Normally distributed, $\sigma_c = \bar{c} * 50\%$; and
- Negative-exponentially distributed.

Figures 6.6 and 6.7 show the raw performance results: the mean packet routed lifetime and workload throughput for the loosely synchronous workload and asynchronous workload with blocking sends respectively, with each plot showing all four of the different distributions of c. We can see immediately that for the asynchronous workload, the exact distribution of the compute period is completely irrelevant: the processes are not synchronised with each other in the long term anyway, so as long as the mean load (and hence the level of network traffic) remains constant, the exact distribution of message injection intervals at each processor is seen to be completely unimportant.

We are much more interested in the loosely-synchronous workload, the workload in which we saw contention even at low load. Looking just at the figure 6.6 for now, the throughput metric shows exactly the expected trend: as σ_c increases, adjacent processes become more out-of-step and communication exchange events force the slower process to block for longer and longer, so overall performance is degraded.

However, a more important effect is demonstrated in figure 6.6: the substantial plateau in packet delivery times that we identified under the loosely synchronous workload is entirely eliminated if we allow adjacent processes to drift apart rather than maintaining lock-step over the system. As we allow σ_c to increase, packet delivery times at light load levels do start to converge to a lower value.

Of even more interesting to our themes in this chapter is the picture which emerges when we look not at the absolute performance results when varying σ_c, but at the relative performance compared to the contention free baseline models. In figure 6.8, we display these values, θ_t and θ_r for the loosely-synchronous workload. Does looking at the results this way shed any more light on the performance?

For a start, when we look at θ_t in figure 6.8, we can see quite clearly the contention effect which prompted us to look at variance in the workload period in the first place. For $\sigma_c = 0$, the continued presence of contention in the network is evident: θ_t approaches but does not exceed a value of about 63% as c increases, indicating that even at the lowest load levels, contention is degrading packet delivery times by over 50%.

In general, there might be a number of explanations for these unexpectedly long packet delivery times at low network loads. If we were simply looking at the complete message delivery times on their own, we would not know whether flow

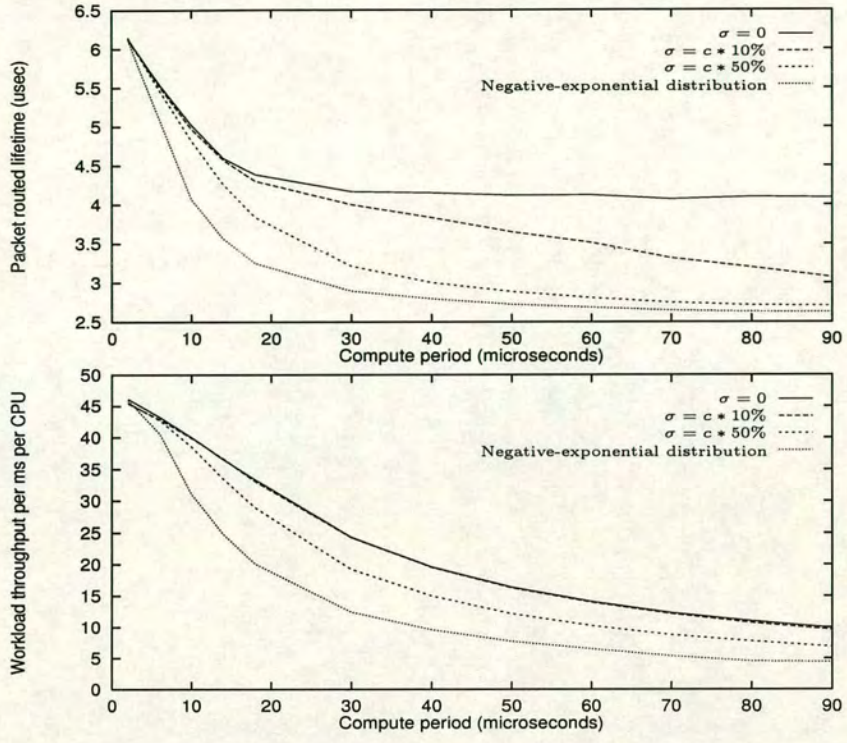


Figure 6.6: Packet routed lifetime and achieved workload throughput for the loosely synchronous workload

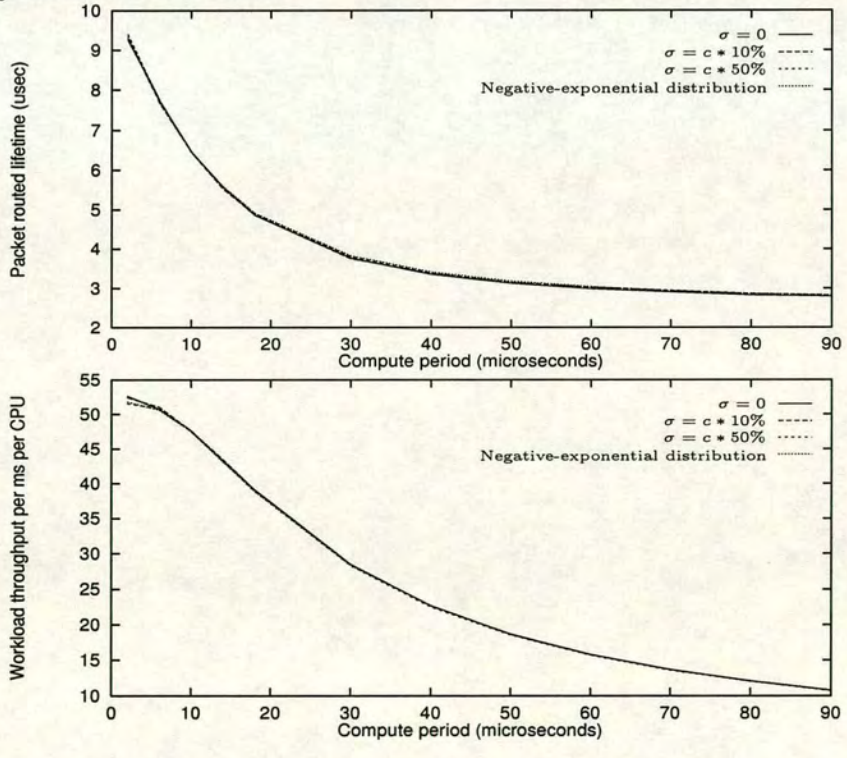


Figure 6.7: Packet routed lifetime and achieved workload throughput for the asynchronous workload

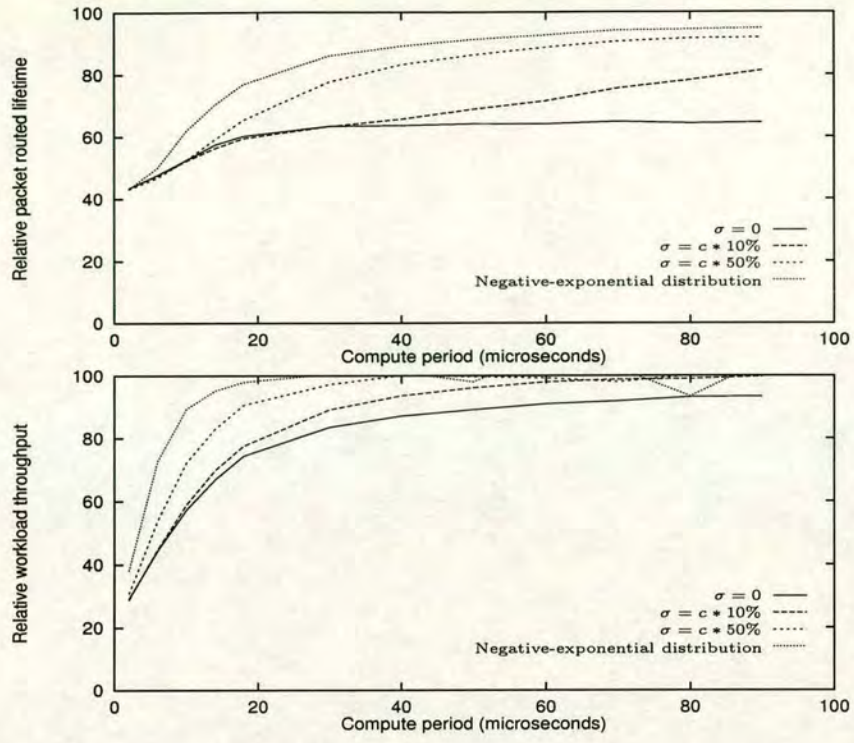


Figure 6.8: Relative packet routed lifetime (θ_t) and achieved workload throughput (θ_r) for the loosely synchronous workload

control was causing longer delivery times, or whether the packet path lengths were longer due to variability in the workload's communication patterns, or whether it was true contention internal to the communications network. The use of packet routed lifetime allows us to eliminate flow control effects as a possible cause, but the θ_t is an unambiguous indicator that contention is truly the cause of the communications slowdown.

But what of the actual achieved workload performance? We can see in figure 6.6 that although at low loads there is a great deal of difference in the packet delivery times between the workloads $\sigma_c = 0$ and $\sigma_c = \bar{c} * 10\%$, there is absolutely no difference at all in their achieved workload performance. It would be easy to leave it at that, and to conclude that the small 10% standard deviation in the compute periods is not interesting from the point of view of contention effects.

Figure 6.8 reveals that there is more to it than this. We can see here that if we remove contention effects, there *is* a difference between these two workloads' achieved throughput rates: in other words, it is only due to residual contention in the network that the $\sigma_c = 0$ workload proceeds no faster than the $\sigma_c = \bar{c} * 10\%$ workload. In the absence of this contention, the lower variance workload would indeed proceed more quickly, and the value of θ_r tells us precisely how much

faster.

6.2.3.1 Normalising Network Applied Load

There is only one more step we need to take before we can really assess the impact which these different levels of application period variance have on the network. One effect remains which we have not yet accounted for: when we change σ_c, we inevitably risk changing the amount of communication traffic we are injecting into the network. Since the loosely synchronous workload must synchronise each process to every other in the long term, adding extra, slower compute periods into the one process's cycle will necessarily cause longer stalls to any other processes which wait for a message from it.

So far we have been able to compare like for like: we have seen that there is not necessarily a requirement that increased contention in one workload's simulation run produces a commensurate decrease in that same workload's achieved performance. However, until we find a scale by which we can normalise our different workloads (bearing in mind that these workloads may impose very different stresses on the network for the same value of c), we cannot reliably compare different simulation runs at once.

One scale might be simply the total amount of network traffic per unit time. There are a number of good reasons why such a scale would have served as a very poor x axis for most of our previous experiments:

- The network traffic itself has been an important performance response in most of our previous experiments; and
- For the asynchronous workloads, there was no guarantee that the message injection rate was constant over the entire network (and in fact we saw good evidence that it would not be constant), so comparing experiments with equivalent average injection rates would give no guarantee that we were comparing equivalent network traffic.

However, we are now looking not at absolute performance but at relative performance degradation due to contention, and the workloads we are interested in here are all loosely synchronous (implying that the time-averaged network load at each node is constant over the network), so these objections no longer hold. If we use this as our x axis and look again at our contention costs θ_t and θ_r, we should be able to see the level and the impact of contention for each of the different workload period variances σ_c given equivalent overall network load. The *only*

difference left between the different σ_c plots here will be the timing characteristics of the message injections. We see the results in figure 6.9.

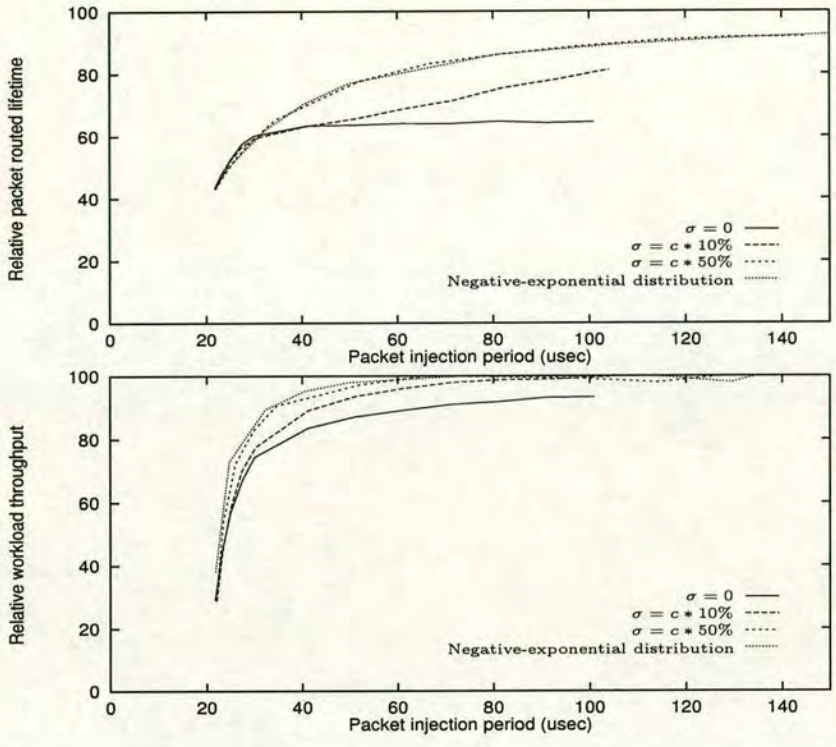


Figure 6.9: Relative packet routed lifetime (θ_t) and achieved workload throughput (θ_r) against injection rate for the loosely synchronous workload

Finally, we can really look the effect of changing the workload timing characteristics on the network. We can see precisely the amount of contention being generated, and the impact of that contention on workload performance, for several different workloads, for constant average network load.

Some of the features we observe are striking. The most important difference we see between these plots and the previous, non-normalised data in figure 6.8 is that there is in fact a lot less difference between the contention costs for each workload at high load levels than we saw previously. Having replotted the graph against real network throughput, we can see that a lot of the difference between workloads under heavy load is simply due to the different way in which the applied load parameter c affects overall load levels for different values of σ_c.

On the other hand, we can see that the choice of workload timing distribution has a significant impact on the amount of contention at low load, as we saw before. More importantly, the difference between the cost of that contention on achieved workload performance (θ_r) at light load levels has not been eliminated by the x-axis rescaling, unlike the situation we saw under heavy load. We can

confidently state that workload performance under light load is being hurt by approximately 10% due to the presence of contention for the $\sigma_c = 0$ workload, but that the other workloads would show absolutely no improvement at all at the same network load levels if we eliminate contention.

These properties remaining in figure 6.8 are true indicators of how much of the contention we see for a given load level depends on some workload characteristic (σ_c here) rather than on the average amount of traffic in our network, and how much better the workload at each load level could run in the absence of that contention. We find here that the amount of contention, and its cost to the workload, is *not* simply a function of the network load. The internal properties of the workload are involved in the equation too.

6.3 Impact of Per-Processor Message Synchronisation

We have now covered all of the tools and techniques we are presenting for the study of the cause-and-effect relationship between contention and the workload. This has been half of our goal throughout this thesis, and we have just now begun to round off the other half—to validate our assertion that this is an important relationship, and that we cannot really understand contention, or the true cost of contention, without understanding the properties of the workload running on our network.

In this final example section, we will demonstrate the application of our tools to a different workload property. Again, our aim is not so much to understand exactly what effect a certain workload change has on performance as to give an example of the types of effect which we need to be aware of when looking at contention, and how we can observe these effects with the tools now at our disposal.

In particular, we want to continue to look at how properties of the workload may materially affect the way contention affects system performance. So far, we have been using a synthetic workload with one process per CPU, transmitting and receiving an average of one message per compute cycle. What happens if we increase the number of messages being sent between each CPU during each compute cycle?

There are two very different ways in which we can send more messages: we can either simply run more processes on each CPU, or we can increase the number of messages being sent by each process. There is a great deal of difference between

these two methods.

On the one hand, if we increase the number of messages being sent by a given process on each cycle, we are increasing the number of other processes which are forced to synchronise with that process on each iteration. We are increasing the degree of the task graph, causing the loosely-synchronous workload to become more tightly synchronised; there is less opportunity for processes to drift apart over time.

On the other hand, increasing the number of processes on each CPU has exactly the opposite effect on the the task graph. The degree of the task graph remains constant, but the number of processes increases, so the number of iterations required for a stall in one process to affect all other processes is increased, not reduced. The entire workload becomes less tightly synchronised, and there is room for increased latency hiding in the application.

In this next experiment, we look at the difference between these two types of behaviour. There is good reason to think that the impact of contention will be depend on which path we choose: the more tightly synchronised our workload is, the more we might expect any delays in individual message deliveries to affect the overall performance.

To exhibit these two effects, we choose to increase the number of messages sent per cycle on each CPU from one to four by varying both n (the number of processes per processor in our workload model), and m (the number of messages sent per process per iteration) such that $n*m = 4$ (ie. $m = n/4$). We also need to modify c, the compute period per process, so that the overall compute cycle period per processor remains constant as we increase n: if our experiment specifies a per-processor compute period of c, then in any given simulation run each individual process needs to use a period of $c' = c/n$. According to our workload definition (definition 5.1), we are using a workload $(16, 8, n, c/n, 4/n, 32)$ for various levels of applied load c and for $n \in \{1, 2, 4\}$.

Figure 6.10 shows the absolute performance results we see when we run these modified workloads for $0 \leq c \leq 200$. We mentioned above that the workload becomes more tightly synchronised as we increase m, and we saw in earlier experiments (especially when comparing the asynchronous and loosely-synchronous workloads) that extra synchronisation within the workload leads to increased sensitivity to contention. We might therefore expect to see reduced performance and increased packet routed lifetimes for workloads with higher m.

In fact, figure 6.10 does show reduced performance under the heaviest loads for the $m = 4$ workload: there is up to a 30% deficit in performance for the same c

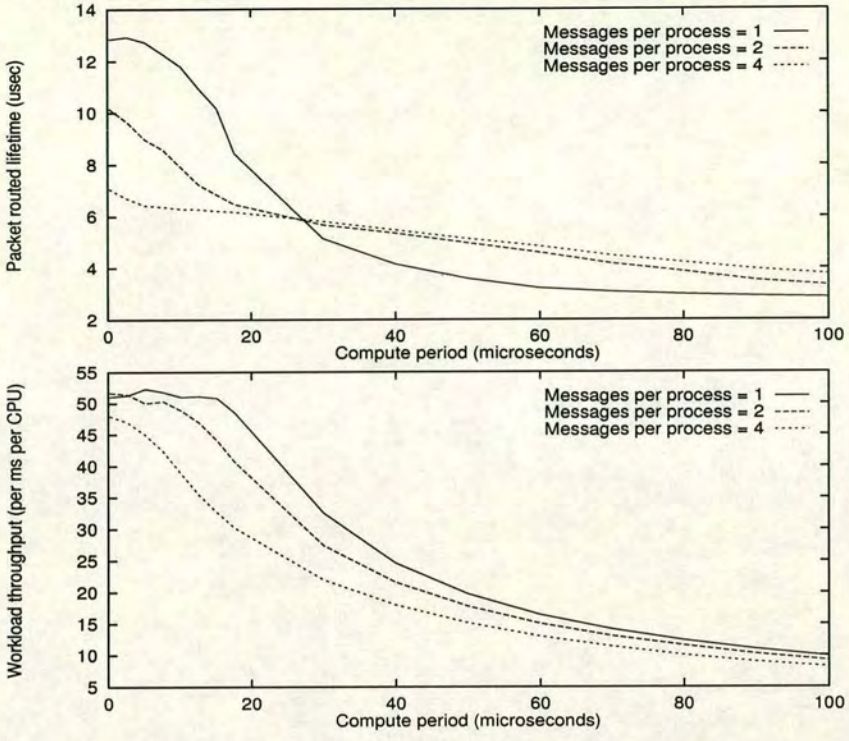


Figure 6.10: Absolute packet routed lifetime and achieved workload throughput for the loosely synchronisation workload

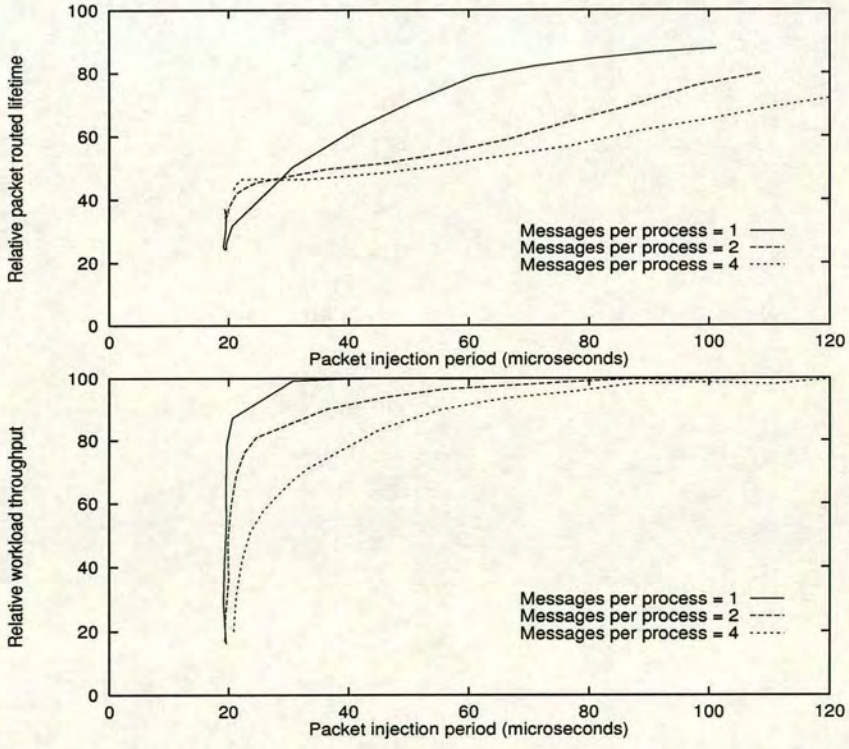


Figure 6.11: Relative packet routed lifetime (θ_t) and achieved workload throughput (θ_r) against injection rate for the loosely synchronous workload

compared to the workload $m = 1$. However, that is not reflected in a simple increase in contention for the more tightly synchronised workload. Under light loads, the more synchronised workloads show longer packet routed lifetimes as expected (presumably to the fact that messages are being injected in bursts), but at heavy loads, this effect does not persist: contention in fact increases sharply for the workload $m = 1$ as we increase the load, especially as we reduce c below about 30μs, but packet lifetimes for the other workloads increase much more slowly as load increases.

Does the methodology we described in section 6.2.3.1 help us to understand what is going on? In figure 6.11, we apply those techniques: we plot not absolute performance but performance relative to the ideal, contention-free model, and we plot those cost metrics against the actual rate of network packet injection, not against the applied load parameter c. We are now looking at the actual amount of contention, and the cost to the workload of that contention, against the average load on the network.

We can see that indeed, the different workloads do have substantially different responses to contention. Looking at θ_r in particular, we see that for any given level of overall load on the network, the more tightly synchronised workloads suffer much more from the impact of contention. Indeed, at a load corresponding to an average of 30μs between packet injections, we can see that the $m = 1$ workload is almost completely unaffected by the contention present in the network, but the $m = 4$ workload, while generating the same amount of network traffic, is achieving only about 70% of its potential performance.

The important point to realise here is that in real life many real applications may behave much more like the tightly synchronised workload than the less synchronised one. Applications such as VLSI simulation and finite element analysis all involve the decomposition of irregular data onto the multicomputer and the subsequent regular exchange of batches of messages between processes. However, if when modelling multicomputer performance we make an assumption that message injections are independent of each other, we lose this effect. We can see in figure 6.11 that if we ignore such workload interactions, we risk grossly underestimating the impact that network contention will have on our workload.

6.4 Summary

We have looked at communication performance and workload performance separately in previous chapters, but in this chapter we looked at the two together.

During this chapter we were interested in two main questions: how to properly measure contention effects, and to see if the nature of the workload was important when studying contention.

From the point of view of the workload's sensitivity to contention, we defined two variables, θ_t and θ_r to be the relative loss of packet transmission time and throughput performance due to contention (relative to performance in an idealised, contention-free environment). We saw that these variables allow us to separately quantify the amount of contention present in a network, and the impact that contention is having on aggregate workload performance.

We found that although the throughput of the fully asynchronous workload displays little or no sensitivity to contention up until the point at which the network saturates, both the locally and globally synchronous workloads display low sensitivity to contention only at low load levels (indeed, the phase-locked bursty nature of message injection in the loosely synchronous workload results in a high level of contention between packets even at low time-averaged load levels, with little performance degradation to the workload). However, as the load increases, so does the sensitivity to contention.

We saw that when we tried to look at the impact of modifying the characteristics of our workload, we risked changing the actual load on the network in unpredictable ways, making it hard to see what change in performance was due only to the reformalisation of the workload and what was actually caused by contention. We showed that we were able to overcome this by plotting θ_t and θ_r against the actual average rate of packet injection into the network.

Given these techniques, we were able to identify two specific characteristics of the workload—the variance in the process compute periods over time, and the rate of synchronisation between processes—which markedly changed the workload's response to contention. We showed that for a given load on the network, different workloads could suffer very differently from contention, some workloads being insensitive even to quite high levels of contention and other workloads suffering more from relatively low levels of contention in the network.

Chapter 7

Summary and Conclusions

In this thesis we have presented an approach to the investigation of contention effects on the performance of wormhole-routed multicomputer interconnection networks. Our main achievement has been that we were able to separately identify the presence of contention in our networks and the effect that contention was having on workload performance.

In chapter 3, we described the models and tools used for the investigation. We presented a model of computation which abstracted out the important characteristics of the workload—in our case, the pattern and ordering of communication requests made of the multicomputer network by the workload. We presented an efficient monolithic simulator developed specifically to investigate these workloads running on a simulated multicomputer network modelled on the T9000/C104 communication architecture. We described simplifications to the communication architecture which were made to improve the performance of the simulator.

In chapter 4, we performed our verification and validation of the simulator. We verified that the simulator produced correct results for a number of easily predictable workloads, and validated our simplified model of the T9000 communication mechanism. Finally, we layed out the groundwork for more complex experiments by assessing the impact of certain design decisions in the layout of our multicomputer networks. In particular, we identified the width of the connection between each T9000 processing element and its associated C104 router as a key variable; with a link width less than the degree of the communications network, we may end up bottlenecked waiting for injection or delivery of packets rather than fully exercising the interconnection network.

Chapter 5 introduced a class of workloads which display various synchronisation characteristics, from loosely-synchronous behaviour through to oblivious behaviour with no flow control at all. We used these workloads to stimulate levels of stress on our multicomputer network from very light loads up to saturation

loads.

We identified contention between newly injected packets for forwarding along their first internal DS-Link as a major factor influencing performance, and defined *throttling* as this natural flow control effect on packets entering the internal routing network. We looked at the time spent by packets in various components of the network, and found that under high load, well over 50% of a packet's lifetime could be accounted for in throttling, either inside the T9000 output queues pending a free injection link, or in the C104 input buffer of an injection link awaiting forwarding onto an internal routing link. We defined a packet's *routed lifetime* to be the time a packet spends in communication starting at the point at which it succeeds in entering its first internal routing link, to provide a metric of packet transmission times independent of throttling effects.

We also looked at the measurement of ideal throughput performance achieved by eliminating contention costs, and concluded that it was important here too to take into account throttling effects, and to preserve throttling contention in our otherwise contention-free simulations to avoid distorting the workload's packet injection behaviour.

Finally, we observed that there can still be contention in the absence of any serious time-averaged load on the network if the message injections are strongly bunched; and that at the very highest levels of load, it is possible to see static or even decreased overall network throughput by injecting more packets into the network. In particular, we saw that certain low-level details of the communications model, such as the prioritising of one packet type over another, could lead to dramatic unfairness in the network under very high load.

Chapter 6 used the techniques presented in chapter 5 to study the relationships between the performance of a workload, as measured by the amount of work achieved per unit time (or, equivalently, the number of successful communications per unit time) and the effect of contention on the performance of the communication network itself, as measured by the time taken to completely route a packet through the network (the packet routed lifetime). We introduced two new variables, θ_t and θ_r, to represent the relative loss in network routing time and workload throughput performance respectively due to contention.

We found that although the fully asynchronous workload remained largely oblivious to contention in the routing network up until saturation, the other workloads displayed a behaviour increasingly sensitive to contention as load increased. We also found that varying other properties of the loosely-synchronous workload could significantly affect its sensitivity to contention, and that different

workloads could suffer from substantially different levels of contention even at the same overall network load.

7.1 Conclusions

There are some very general conclusions we can draw about multicomputer network behaviour from this study.

Details matter. We have looked at a number of apparently minor ordering properties of the communications architecture, including the prioritising of packets at the C104 crossbar and acknowledgement prioritising in the T9000, and have seen that these details can have a profound impact on performance in some situations.

The workload matters. The precise details of our workload have also been shown to substantially affect the way contention affects overall performance. The level of synchronisation between processes, the use of blocking or non-blocking sends, the number of messages sent at once by a single process and the randomness in the timing of compute cycles—all of these have been shown to change the way in which the workload responds to contention. The presence of a certain amount of contention in the network does *not*, on its own, tell us how the performance of an application running on that network will suffer due to that contention.

As a result, we can conclude that in modelling the workload performance of systems experiencing contention, it may not be valid to assume that message arrivals are independent—the workload with independent message injections does not show the same behaviour under load as the workloads which do experience packet dependencies.

One of the reasons that it is attractive to assume independent message injection when modelling multicomputer performance is that this leads to tractable models. In this thesis, we have presented a simulator and methodology which is sufficiently efficient to be able to serve as a viable alternative to these models, but which retains the ability to take account of the fine details of packet interactions which we have found to be so important.

7.2 Future Work

So far, we have described a tool and a set of metrics for observing contention-related effects. We have shown these to be useful in understanding the behaviour

at high loads of a simple class of workloads. However, we have only begun to scratch the surface of effects which could be studied using these tools. There are a number of different ways in which this work could be extended.

In looking at our workload performance, we have concentrated on a single synthetic workload model. There are many other workloads worth studying; in particular, non-uniform traffic is often particularly prone to contention and would be well worth investigation. It would also be interesting to follow up this work by using workloads modelled after real-life algorithms or applications.

There are also a number of techniques available which aim to reduce contention in networks. The entire genre of adaptive routers exists precisely to achieve this, and the contention responses of networks based on such routers might be substantially different to the basic dimension-order greedy router used throughout this study.

Perhaps most importantly, this study could be used as the basis of a performance model of multicomputer networks which takes into account workload synchronisation patterns. Phillips *et al.* [CFPS92, CP93] use standard statistical modelling techniques to generate models for parallel workload performance, and these techniques could be used to form a performance model incorporating the features described here. Enbody [KE95, ENHS94] proposes a model of contention effects which assumes independent messages arrivals, and it would be useful to know how well that model could be used under workloads exhibiting synchronisation features.

Bibliography

[AHK87] Sheldon B. Akers, Dov Harel, and Balakrishnan Krishnamurthy. The star graph: An attractive alternative to the n-cube. In *Proc. Int'l Conf. Parallel Processing*, pages 393–400, 1987.

[All90] Arnold O. Allen. *Probability, Statistics and Queuing Theory with Computer Science Applications, 2nd ed.* Academic Press, Inc, 1990.

[BFS94] Kevin Bolding, Melanie L. Fulgham, and Lawrence Snyder. The case for chaotic adaptive routing. Technical Report CSE-94-02-04, Dept. of Computer Science and Engineering, University of Washington, 1994.

[BGT93] C. Barnaby, V.A. Griffiths, and P.W. Thompson. Models of ds-link performance. In M.D. May, P.W. Thompson, and P.H. Welch, editors, *Networks, Routers and Transputers*, chapter 6, pages 85–104. IOS Press, Amsterdam, 1993.

[Bir86] G. M. Birtwistle. *Discrete Event Modelling on Simula.* Macmillan, 1986.

[Bol93] Kevin Bolding. *Chaotic Routing—Design and Implementation of an Adaptive Multicomputer Network Router.* PhD thesis, University of Washington, 1993.

[BP96] Debashis Basak and Dhabaleswar K. Panda. Alleviating consumption channel bottleneck in wormhole-routed k-ary n-cubesystems. Technical Report OSU-CISRC-8/9-TR41, OSU, 1996.

[CFPS92] Rosemary Candlin, Peter Fisk, Joe Phillips, and Neil Skilling. Studying the performance properties of concurrent programs by simulation experiments on synthetic programs. In *Proceedings of the SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, pages 239–240, New York, NY, USA, June 1992. ACM Press.

[CFRR95] G. Chiola, G. Francheschinis, R.Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24:47–68, 1995.

[Chi93] Andrew A. Chien. A cost and speed model for k-ary n-cubewormhole routers. In *Proc. Hot Interconnects*, Palo Alto, California, August 1993.

[Chi95] Donald D. Chinn. *Packet Routing in Multiprocessor Networks.* PhD thesis, University of Washington, 1995.

[CK92] Andrew A. Chien and Jae H. Kim. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. *Journal of the ACM*, 41(1), 1992.

[CP93] Rosemary Candlin and Joe Phillips. Statistical modelling as a tool for studying the performance of parallel systems. In *Proc. 2nd Abstract Machines Workshop, Leeds*, 1993.

[Dal87] W. Dally. Wire-efficient VLSI multiprocessor communication networks. In *Proc. Stanford Conference on Advanced Research in VLSI*, pages 391–415. MIT Press, 1987.

[Dal91] William J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 40(9):1016–1023, sep 1991.

[DS86] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, 1986.

[DS87] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

[ENHS94] Richard J. Enbody, Peter Nesbeitt, Tim Harrison, and Rob Seward. Communication and contention on the Intel DELTA. Technical Report CPS-94-1, Dept. of Computer Science. Michigan State University, January 1994.

[FW87] John E. Freund and Ronald E. Walpole. *Mathematical Statistics.* Prentice-Hall, 4th edition edition, 1987.

[GBES93] Cécile Germain, Jean-Luc Béchennec, Daniel Etiemble, and Jean-Paul Sansonnet. A communication architecture for a massively parallel message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 19:338–348, 1993.

[GCS89] T. Guilfoy, R. Candlin, and N. Skilling. A modelling system for process-based programs. In *Proc. European Simulation Congress*, 1989.

[GKZH95] Reinhard German, Christian Kelling, Armin Zimmermann, and Günter Hommel. TimeNET: a toolkit for evaluating non-Markovian stochastic Petri nets. *Performance Evaluation*, 24:69–87, 1995.

[Gui88] T. Guilfoy. A modelling system for Posie. Master's thesis, Dept. Computer Science, Edinburgh University, 1988.

[Gun81] K. D. Gunther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communications*, COM-29:512–524, Apr 1981.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.

[KC94] Vijay Karamcheti and Andrew A. Chien. Do faster routers imply faster communication? In *Proc. Parallel Computer Routing and Communications Workshop*, Seattle, Washington, May 1994.

[KE95] Swamy S. Kocherlakota and Richard J. Enbody. Predicting the performance of a wormhole-routed multicomputer with non-uniform communication. Technical Report CPS-95-12, Dept. of Computer Science, Michigan State University, March 1995.

[KL92] G. Klas and R. Lepold. TOMSPIN, a tool for modeling with stochastic Petri nets. In *Proc. 6th Annual European Computer Conf.*, pages 618–623, Le Hague, Netherlands, 1992.

[Kle76] L. Kleinrock. *Queuing Systems.* Wiley, New York, 1976.

[KP86] Manoj Kumar and Gregory F. Pfister. The onset of hot spot contention. In *Proc. Int'l Conf. Parallel Processing*, pages 28–34. IEEE, 1986.

[LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the connection machine CM-5. In *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architecture*, 1992.

[Lav89] S. S. Lavenberg. A perspective on queuing models of computer performance. *Performance Evaluation*, 10(1), Oct 1989.

[Lee85] Rosanna Lee. On "hot spot" contention. *Computer Architecture News*, 13(5):15–20, dec 1985.

[Lei85] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[LH91] Danial H. Linder and Jim C. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1):2–12, January 1991.

[Lin92] C. Lindemann. DSPNexpress: a software package for the efficient solution of deterministic and stochastic Petri nets. In *Proc. 6th Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluations*, pages 15–29, Edinburgh, Great Britain, 1992.

[LK90] Tomas Lang and Lance Kurisaki. Nonuniform traffic spots (nuts) in multistage interconnection networks. *Journal of Parallel and Distributed Computing*, 10:55–67, 1990.

[MJTR94] Philip K. McKinley, Yih Jia Tsai, and David F. Robinson. A survey of collective communication in wormhole-routed massively parallel multicomputers. Technical Report MSU-CPS-94-35, MSU, June 1994.

[MTW93] M. D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers and Transputers.* IOS Press, Amsterdam, 1993.

[NM91] Lionel M. Ni and Philip K. McKinley. A survey of routing techniques in wormhole networks. Technical Report MSU-CPS-ACS-46, MSU, October 17 1991.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Englewood Cliffs, NJ., 1981.

[Poo87] R. Pooley. *An Introduction to Programming in SIMULA.* Blackwell Scientific Publications, 1987.

[Rei85] W. Reisig. *Petri Nets, An Introduction.* Springer-Verlag, 1985.

[SP89] Maheswara R. Samatham and Dhirak K. Pradhan. The de bruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI. *IEEE Transactions on Computers*, 33(4):567–581, April 1989.

[Val82] L. G. Valiant. Optimality of a two-phase strategy for routing in interconnection networks. 1982.

[YN90] Abdou S. Youssef and Bhagirath Narahari. The banyan-hypercube networks. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):160–169, April 1990.