Shared Memory with Hidden Latency on a Family of Mesh-like Networks

Tim J. Harris

Doctor of Philosophy
University of Edinburgh

Abstract

In this thesis we consider the general problem of how to provide a shared memory model on a network of processors where memory is physically distributed among the processors. In particular, we consider the simulation of an EREW PRAM model on a family of mesh and ring like networks, and we are interested in latency hiding simulations. Our goal is first to provide a simulation which has delay proportional to the diameter of the network, and second to hide the simulation delay entirely though use of multithreading techniques.

We begin with a general introduction to the problem of PRAM simulation, and a brief survey of the state of the art in such simulations. We then highlight the importance of processor efficient simulations, where the latency of access to shared memory is hidden. We consider the use of multithreading for latency hiding in PRAM simulations in a general context, addressing the relationship between the number of threads run on each processor and diameter of the network. We provide evidence that in the general case bounded degree networks will not have enough bandwidth to support such processor efficient simulations, and we define a class of networks, known as *fat rings* and *fat meshes*, which provide the necessary bandwidth. We then implement the ideas we have discussed by providing a processor efficient EREW PRAM simulation on a family of networks consisting of fat meshes of arbitrary dimension. The simulation focuses on memory management and routing techniques for the networks. We provide evidence that concurrent access models are inherently poorly suited for multithreaded architectures. Given these difficulties we go on to describe a satisfactory CRCW PRAM simulation for the fat mesh, which by necessity has delay which is greater then the diameter of the network. We then reinforce our theoretical conclusions with experimental results generated from a trace driven simulation of our architecture. We conclude with an assessment of some performance characteristics of fat mesh machines, as well as a review of the main points of the thesis.

Acknowledgments

My time in Edinburgh has been both challenging and entertaining, and for that I have many to thank. Lennart Johnsson provided me with alot of the inspiration to go on to graduate school, and David Wallace gave me the funding and exciting project that led me to Edinburgh. My advisor, Murray Cole, provided me with the ideal combination of both time to dream about changing the world and time to get things done. My being able to finish in a timely manner is primarily due to him. Nigel Topham reminded me of the excitement in actually building parallel machines, after inviting me to join one of the most interesting European projects I know of. I'm thankful to all those at the Edinburgh Parallel Computing Centre, for their direct involvement in my first two years of research, and for their subsequent friendship after I moved on. And I'm thankful to Alison Monteith, who tried to not be too embarrassed that her boyfriend was "just a student", and put up with my wandering travels with work.

Beyond that there are many others that made smaller yet still substantial contributions to my work, whether they are aware of it or not. Graham Jones, Todd Heywood, Mikee Norman, Greg Wilson, Jop Sibeyn, and Leslie Goldberg all played important roles in my development as a researcher. I'm thankful for the mentorship and funding provided by Fabrizio Luccio, Gianfranco Bilardi, Wolfgang Paul, David Skillicorn, and Franco Preparata as well as all of their bright graduate students. And finally, thanks to all my former Thinking Machines colleagues, especially Alan Edelman, Mike McKenna, Kapil Mathur, and Anne Trefethen, who helped give me the ambitious dreams for parallel computing that I'm still chasing today.

Declaration

I declare that the following thesis was composed by me, and that all work in it is my own unless otherwise attributed. Some of the following has appeared previously in [Harris 1994].

Table of Contents

List of Figures

Chapter 1

PRAM Simulation

Parallel computers have long been acknowledged to have tremendous potential to outperform their serial counterparts. However, two fundamental problems have always existed with parallel machines: they are difficult to program, and the performance of the average program is often far below what is expected. The first problem can be addressed in a small part by the use of a shared memory model for programming, where users need not consider the details of an underlying interconnection network, but only assume a high level abstraction of a parallel machine. A partial solution to the second problem may result from the development of an architecture with dependable performance characteristics.

In the following we suggest a way to provide a parallel machine with these two attributes; a powerful shared memory model and dependable performance. More specifically, we suggest a way to support an n processor Exclusive Read, Exclusive Write PRAM on a family of mesh-like networks with physically distributed memory, such that the entire latency of memory access is hidden. The effect of this latency hiding is that processors should have very little idle time; a program running on p processors should run $O(p)$ times faster than that same program running on one processor. Though other work has addressed such latency hiding PRAM simulations, the novelty of our approach is in the use of mesh-like interconnection networks which have been augmented to serve our purposes. This introductory

chapter provides a thorough definition of the general problem of PRAM simulation, and will familiarize the reader with some of the techniques which we will exploit in later stages of the thesis. Chapter 2 continues the introduction, but by focusing on the specific problem of efficient PRAM simulations. Much of this introduction has been previously published in [Harris 1994].

1.1 Thesis Contributions

We now describe briefly the focus of the thesis, and where specific contributions have been made above and beyond the previous results in this area. We leave until later any statement of specific theorems, and instead identify the general topics which are being addressed in a new or novel way. These topics are:

1. The relationship between network diameter and slackness in multithreaded simulations.

2. The bandwidth requirements of multithreaded simulations running on bounded degree networks.

3. The performance benefits of multithreading techniques on networks of high diameter.

4. The complexity of supporting concurrent access models in multithreaded simulations.

5. The role of PRAM simulations in providing predictable multiprocessor performance.

We now provide a description of the context of the thesis, i.e. a concise problem definition as well as an explanation of related existing results.

1.2 Problem Definition

When setting out to design a parallel algorithm one must ask two questions of the problem at hand:

(i) Can enough parallelism in the problem be identified to allow a good solution?

(ii) Can the processors share the data necessary in the problem fast enough (given their organization) to allow a good solution?

The first question is the most fundamental, in that problems with little inherent parallelism will never have good parallel solutions. Furthermore the answer to this question is largely independent of what parallel computational model one chooses to use. On the other hand, the answer to the second question depends intimately on the model used. Questions of synchronization are also implicit in this question of communications.

Models of parallel computation can roughly be broken into two groups [McColl 1992]. Special purpose models are those where the processors communicate through a completely specified network of links, and where attempts are made to exploit the locality of the processor organization as much as possible. These models require that the algorithm designer solve both problems 1 and 2 explicitly. The term special purpose refers to the fact that an algorithm designed for one such model will seldom be portable to other such models, i.e. its applicability is specialized. Examples of such models are hypercube, tree, and mesh models.

The other type of models, general purpose models, are those where powerful and general communications are assumed, typically in the form of a large synchronized shared memory accessible by all processors. Such assumptions allow researchers to focus on the fundamental characteristics of a parallel computation, and ignore the issues which arise through particular choices in architecture and

interconnection networks. More specifically, general purpose models allow one to consider problem 1 above, while not being distracted by the compounded difficulty of solving problem 2 simultaneously. In addition to its simplicity, the use of such an abstraction is further justified by the rate at which parallel architectures change in practise, which causes results regarding special purpose models to have limited relevance over time. The most common general purpose model is the Parallel Random Access Machine, or PRAM, as shown in figure 1–1.

An (n, m)-PRAM consists of n processors and m memory locations, where each processor is a random access machine. All processors share the memory, and hence communicate via that memory. During a given cycle each processor may read an element from the shared memory into its local memory, write an element from its local memory to the shared memory, or perform any RAM operation on the data which it already has in its local memory (e.g. addition, multiplication, or boolean operations). It is a synchronous model, in that no processor will proceed with instruction $i+1$ until all have finished instruction i. Within this synchronous restriction a PRAM may execute in SIMD mode or in MIMD mode, though the complexity of analyzing a MIMD algorithm means that in practice few MIMD PRAM algorithms have been designed. The original definition of the PRAM can be found in Fortune and Wyllie [Fortune and Wyllie 1978], though related early models are described in [Schwartz 1980, Goldschlager 1982].

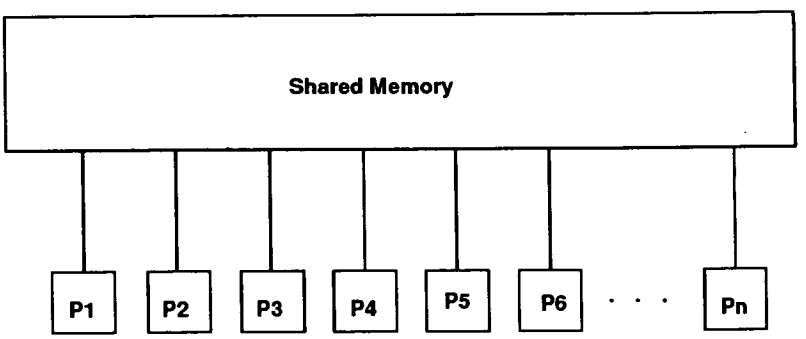


Figure 1–1: The PRAM model of computation

The above description still leaves some ambiguity regarding the behaviour of the PRAM. In particular, it is not specified whether various processors may access the same memory location on a given cycle or not. There is a family of PRAM models, each of which differs in its characteristics on this point. The members of this family are:

(i) The Exclusive Read, Exclusive Write (EREW) PRAM, where at most one processor may read or write to a particular memory location.

(ii) The Concurrent Read, Exclusive Write (CREW) PRAM, where multiple processors may read from a particular memory location, but at most one processor may write to a particular memory location.

(iii) The Concurrent Read, Concurrent Write (CRCW) PRAM, where multiple processors may read or write to any memory location.

ERCW PRAMs are not considered, as a machine with enough power to support concurrent writes should also be able to support concurrent reads.

We also need to specify a conflict resolution strategy for CRCW PRAMs, i.e. what is written when more than one processor writes to a particular memory location on a given cycle? These additional variants are classified as:

(i) The COMMON CRCW PRAM, where all values written concurrently must be identical. If the values written are not identical then an error is flagged and computation halts.

(ii) The ARBITRARY CRCW PRAM, where the processor that succeeds in its concurrent write is chosen arbitrarily from the writing processors.

(iii) The PRIORITY CRCW PRAM, where the processor that succeeds in its concurrent write is the processor with the highest priority, e.g., the smallest processor index.

(iv) The COMBINING CRCW PRAM, where the value written is a linear combination of all values which were concurrently written, e.g., a sum of the values. Values may be combined with any associative and commutative operation which is computable in constant time on a serial RAM.

The above are listed roughly in increasing order of power [Kucera 1982]. For a more thorough definition of the above see [Akl 1989b].

The simplicity and generality of the PRAM model has led to its wide acceptance as a research tool, and there are a large number of PRAM algorithms and results in the literature (see for example [Cook 1984, Gibbons 1988, Akl 1989b, Karp and Ramachandran 1990, McColl 1992]). However, there are still questions about the applicability of this work to realistic machines. The PRAM cannot be constructed with current technology beyond a few processors, and it appears unlikely that this will change in the future. In particular, a multi-ported memory which is shared by a large number of processors is infeasible. Instead, a realistic and scalable parallel computer typically consists of a set of processor/memory module pairs which are connected by a sparse network of links. Each memory module will be able to service one memory request per cycle and one message may each travel across one link per cycle. If more than one request arrives at a module in a cycle then they will be serviced sequentially. This architecture may be scaled to many thousands of processors, particularly if the interconnection network is of fixed degree, i.e. has a constant number of links leaving or entering each node.

Given the fact that the PRAM is not physically realizable, one may attempt to make use of the large body of PRAM results by modifying them, one by one, to apply to a particular parallel machine which is currently of interest. However, this promises to be an arduous task, and one which can be entirely subsumed within the task of developing a general simulation of a PRAM on more realistic parallel machines. The problem of simulating a PRAM therefore consists of designing algorithms that allow instructions of the PRAM to be executed on a feasible

parallel computer with minimum slowdown. A successful PRAM simulation will allow a large body of theoretical results to be of practical use. For a non-technical discussion of simulations and other PRAM issues see [Sanz 1988].

Definition 1 *A simulation of machine M_1 on machine M_2 is an algorithm that allows any instruction from M_1 to be executed on M_2.*

When we refer to the problem of PRAM simulation we mean the simulation of a CRCW PRAM on a realistic parallel machine, namely one with distributed memory and an interconnection network of fixed degree. This is the central problem we consider. However, for the purpose of this survey of current work we will break the problem into three disjoint phases, each of which is a simulation in itself. The reason for treating these problems separately is that they are all fundamental problems in theoretical computer science, and the solutions which are identified for these subproblems may be reused in other contexts. In some cases combining the solutions of the subproblems to solve the entire simulation problem results in a PRAM simulation that is as good as a direct solution of the problem can produce. In other cases, it is necessary to address the larger problem all at once in order to achieve good performance, rather than combining solutions to the smaller sub-problems. In either case, addressing the subproblems independently plays an important role in providing intuition about the utility of various techniques. Other papers have suggested such a separation, notably [Mehlhorn and Vishkin 1984].

The three subproblems are:

- The Concurrent Access Problem:
  Assume that on each cycle the processors of an (n,m)-PRAM may request concurrent access to any of the m memory locations, using one of the CRCW variants outlined above. The problem is to service these requests correctly on hardware that disallows concurrent access, namely an EREW PRAM.

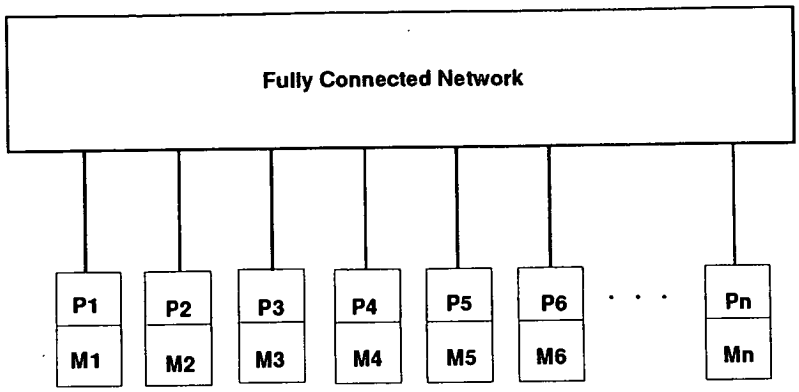


Figure 1–2: The Module Parallel Computer

This problem has a well known optimal solution which described later in this chapter.

- The Memory Management Problem:
  Consider an (n, m)-PRAM which is to be simulated on a machine with M memory modules, and assuming that $m \geq n^2$, so that each memory module will hold $m/M \geq n$ memory locations. Also assume that the processors are fully connected, so any processor can communicate with any other in constant time. If each processor issues a request to memory, then in the best case each request will go to a different memory module, and the set of requests may be serviced in $O(1)$ time. However, if an adversary chooses the requests such that all n are directed to the same memory module, then this step will require $\Omega(n)$ time. The problem of memory management is how to layout memory such that the amount of module contention is minimized given any set of n requests which are to be serviced. This problem is called the granularity problem in [Mehlhorn and Vishkin 1984].

- The Routing/Interconnection Problem:
  Assume that each of the n PRAM processors holds a request for a memory element which specifies the module and location desired. The routing/interconnection

problem is to specify a fixed degree (and hence sparse) interconnection network and a routing algorithm that will allow servicing of all of these requests with the minimum slowdown. It will be assumed in our specifications that the memory management scheme may have already manipulated the memory requests before the router takes control.

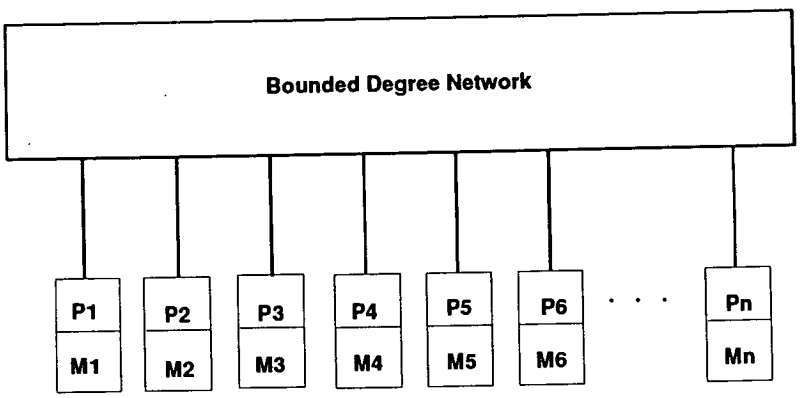


Figure 1–3: The Bounded Degree Network Model

The problems can be made disjoint by considering three independent simulation problems. To deal with concurrent access we need to simulate a CRCW PRAM on an EREW PRAM. To solve the memory management problem we consider simulation of an EREW PRAM on a fully connected parallel computer (called a Module Parallel Computer or MPC). The MPC consists of n RAM processors, each of which has an associated memory module, where a memory module is a collection of memory locations, each of which stores one data value (see figure 1–2). All requests that arrive at a memory module in a given cycle will be processed sequentially, thereby causing a slowdown, and each RAM in the MPC is connected via a communications link to all other processors. This type of interconnection network is infeasible to build, but allows one to address memory management issues without considering routing, since routing is trivial on a fully connected graph. The key is to specify the arrangement of PRAM memory locations among the modules of the MPC such that contention for memory modules is reduced.

The routing problem can be addressed by simulating an MPC on a bounded degree network, or BDN. A BDN is a similar set of n RAM/Memory module pairs, but they are connected to each other via a sparse interconnection network which has a fixed degree (i.e. a constant number of links) at each node, as shown in figure 1–3. The solution to the routing problem is given by a pair (G, R), where G is a graph denoting the interconnection of our n processors and R is a routing algorithm. The series of subproblems which compose the general problem of PRAM simulation may be seen in figure 1–4.

Figure 1–4: PRAM Simulation problem decomposition

The quality of a simulation is determined primarily by the *slowdown* of the simulation.

Definition 2 *If a program requires T steps on an n processor PRAM, and when the program is run on top of a PRAM simulation it executes in time $O(Tf(n))$, then the* slowdown *of that simulation is $O(f(n))$.*

In our formulation of the problem all three subproblems will have a slowdown and therefore may contribute to the slowdown of the overall problem of simulating a CRCW PRAM on a BDN. Various simulation techniques also require an increase in the amount of memory utilized, referred to as *memory-blowup*, and this will also be a factor in assessing the quality of a simulation.

Definition 3 *If a PRAM requires memory M to execute a program, and when the program is run on top of a PRAM simulation it requires memory $O(Mg(n))$, then the memory blowup of that simulation is $O(g(n))$.*

One additional metric of the quality of a simulation is the efficiency. A simulation's efficiency is the ratio of time-processor products for different levels of the simulation.

Definition 4 *If a program takes time T on an n processor PRAM, and a simulation executes in time T' on p processors, then the* efficiency, *E, of the simulation is:*

$$E = Tn/T'p$$

Simulations where E is a constant independent of n and p are often referred to as *constant time-processor product* or simply efficient simulations, and will be discussed in the next chapter. Except where we consider efficiency, we will assume that the simulating and simulated machines both have the same number of processors.

In the first part of this chapter we explain techniques for simulating any variant of the CRCW PRAM on an EREW PRAM. This will then allow us to focus on the two primary problems, firstly of simulating an EREW PRAM on an MPC, and subsequently of simulating an MPC on a BDN. We then summarize known deterministic solutions to these two problems, and consider the goal of reducing the amount of replicated memory necessary while reducing contention for memory modules. We will then outline the analogous known randomized solutions, namely uniform hashing and two-phase routing.

1.3 Concurrent Access

The ability to access a memory location concurrently is a powerful one, and can lead to algorithms that have significantly smaller time complexity than those designed for models that forbid concurrent access. For example, the multiplication of two $N \times N$ matrices on N^3 processors requires $\Omega(\log N)$ time on a EREW PRAM,

where COMBINING CRCW PRAMs may solve this problem in $O(1)$ time [Aggarwal et al. 1990]. Algorithms have been designed for a variety of conflict resolution strategies in concurrent access models, eg. [Shiloach and Vishkin 1981, Kucera 1982, Akl 1989a]. In order to isolate our simulation from this variety, we now show that a simple strategy can allow simulation of all CRCW PRAMs on an EREW model with optimal slowdown. This allows us to focus on more difficult simulation problems in later sections. Such CRCW simulations have also been used to imply the equivalence of all CRCW variants [Akl 1989a, Kucera 1982].

This simulation requires $O(n)$ extra memory, and is based on that found in [Karp and Ramachandran 1990], though simulations appearing in [Vishkin 1982] are similar. We assume at the beginning of a cycle that each processor holds a memory request of the form (j, i) where j is the address of the requested memory location $1 \leq j \leq m$ and $1 \leq i \leq n$ is the index of the requesting processor. The pairs (j, i) are then sorted, first on j and then on i. This sort will require $\Omega(\log n)$ time, and if it is running on an n processor EREW PRAM we may use Cole's Merge sort algorithm [Cole 1988]. The algorithm uses a binary tree, and pipelining among the levels of the tree to achieve such optimality.

After the sort the processors will eliminate duplicate requests by cooperating as if they were arranged in a binary tree. At every level of the tree the participating processors compare two sorted CRCW requests, and combine them if they are destined for the same address, thereby eliminating up to half of the existing concurrent requests. Requests are combined as per the appropriate conflict resolution strategy. After $O(\log n)$ such steps all concurrent requests have been eliminated, and we now have a set of EREW requests to be dealt with in the normal way. If the operation is a read, then a multi-broadcast will need to be executed after the location is fetched from memory, and this can also be done in $O(\log n)$ steps by having the EREW processors combine to build spanning trees [Akl 1989b]. We provide more detail to CRCW combining in later chapters, but for now we simply state the following result:

Theorem 1 *[Karp and Ramachandran 1990] Any variant of the CRCW PRAM may be simulated on a EREW PRAM with slowdown* $\Theta(\log n)$.

1.4 Deterministic Simulations

Deterministic PRAM simulations are in some sense more desirable than randomized simulations, as their behaviour is more consistent. However, the performance of deterministic algorithms is adversely effected by undesirable worst case behaviour. This is unlike the case of randomized algorithms, where we consider only cases that occur with high probability as relevant.

1.4.1 Memory Management

As mentioned above, the trivial solution to the simulation of an EREW PRAM on an MPC results in a worst case slowdown of $\Omega(n)$. This initially discouraged consideration of deterministic solutions. Mehlhorn and Vishkin first proposed the use of multiple copies of each memory location to solve the memory management problem [Mehlhorn and Vishkin 1984]. However, while their algorithm used copies to reduce the cost of a memory read, it did not improve the performance of memory writes beyond the trivial $O(n)$. The paper advocates that reads can be made to the copy which is easiest to access, and n read requests can be serviced in $O(cn^{1-1/c})$ time, where c is the number of copies of each memory location.

The next substantial improvement in deterministic solutions of this problem came from Upfal and Wigderson in [Upfal and Wigderson 1987]. They proposed that the copies could reduce the time necessary for a write, as well as a read, despite the added coherence problems introduced by multiple copies on write operations. This technique has since been used in most deterministic solutions. It is referred to as the majority method, and comes originally from the field of database theory

[Thomas 1979]. The general idea is that it is not necessary to update every copy of a particular memory location on a write, but only to update a majority of them if each location is augmented with a time stamp. In particular if there are $2c-1$ copies, then at least c must be updated and timestamped on each write. This guarantees that if each read accesses at least c copies also, then the intersection of the set of memory locations read and the set of those that are current is size one or greater. The reading processors will then check the time stamp to verify that they accept only the most recent value. During the simulation of a PRAM step the $2c-1$ copies of each variable all begin as *live*, but are then designated *dead* if c or more copies of the variable have been accessed while fulfilling the current memory request.

The scheme is made feasible through the following lemma:

Lemma 1 *[Upfal and Wigderson 1987] Given n sufficiently large and $b > 4$, there is a $c = O(\log m/\log b)$ such that there is a way to distribute the $2c-1$ copies of each variable among the processors and ensure that, for any set of $q \leq n/(2c-1)$ live variables, the live copies reside in at least $(2c-1)q/b$ processors.*

This lemma ensures that copies of a variable will be spread out among processors sufficiently to allow relatively quick access. We now give an informal explanation of the techniques for accessing memory within this scheme. The processors are arranged in $k = n/(2c-1)$ clusters, each with $2c-1$ processors (see figure 1–5). In order to fulfill the n memory requests for a given cycle, the memory management algorithm will proceed in two phases. The memory map is distributed in the machine, such that the i-th processor of each cluster will know the location of the i-th copy of each variable in memory. In the first phase each cluster will try to satisfy as many of the requests of its members as it can. In each step the clusters will choose one of their $2c-1$ memory requests to fulfill, and then every processor in the cluster will try to access a unique copy of that variable, i.e. the i-th copy will be accessed by P_i. Some of these access attempts will be successful,

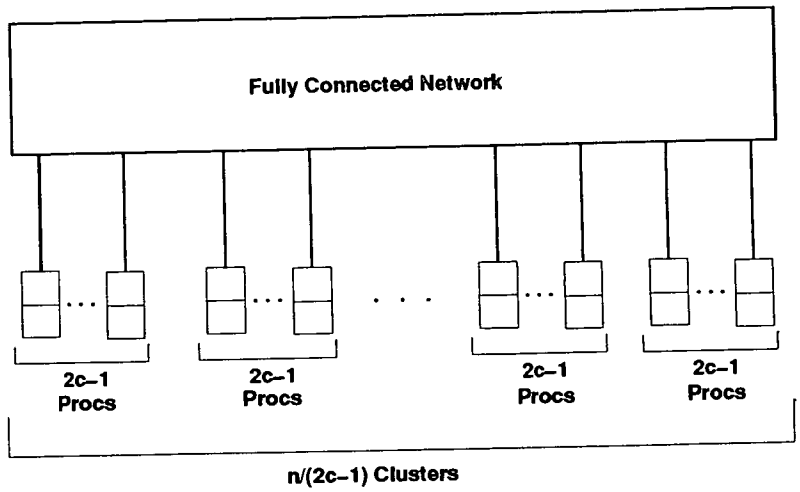


Figure 1–5: Majority Scheme for Deterministic Memory Management

but others will find contention at the memory module which holds that copy, and will therefore be aborted. The copies that have been accessed will be routed back to the leader processor for that memory request, which will count the accessed copies. If c or more copies have been returned to the leader, then the request has been fulfilled and the variable is dead. Otherwise, the variable is still alive and the request still pending. $2c-1$ such steps will be executed in the first phase, each one attempting to satisfy one of the requests of a processor in its cluster. It can be shown that after the entire phase 1 that at most $n/(2c-1)$ requests will remain unsatisfied. This upper bound is a result of the initial mapping described in the lemma [Upfal and Wigderson 1987].

In phase 2 the outstanding requests will be remapped so that each cluster has at most one to satisfy. The requests then will be fulfilled in a similar manner to phase 1, but if there is contention for a memory module then the request will queue there and be processed serially. This process will continue until the leaders for these requests declares that at least c copies have been returned to it, and the up-to-date copy can be determined by use of the time stamps.

Another contribution of the Upfal and Wigderson paper is a lower bound on

the slowdown in terms of the redundancy (i.e. number of copies) necessary in the scheme. They showed that the slowdown will be $\Omega((m/n)^{1/2r})$ for a scheme requiring r copies. Therefore to get a slowdown of $O(\log n)$ one will need at least $\Omega(\log(m/n)/\log\log n)$ copies. The Upfal and Wigderson scheme uses $\Theta(\log m)$ copies, and allows simulation of an EREW PRAM on a MPC with slowdown of $O(\log n(\log\log n)^2)$.

One penalty of this scheme is the additional memory that the use of copies will require. Another memory cost is the time stamps. The amount of memory used by the time stamps may be reduced slightly if time is counted modulo m. This is possible if after every m steps each memory location is cleaned, i.e. time is set back to 1 for valid copies, and 0 for invalid copies [Alt et al. 1987]. Cleaning of one location may be done in $O(\log n)$ time, and so may be done with only a constant slowdown during a typical read or write cycle.

Alt, Hagerup, Mehlhorn and Preparata showed how the time of simulating an EREW cycle on a MPC can be reduced to $O(\log m)$ while using a similar degree of redundancy, or $O(\log n)$ if we assume m is polynomial in n [Alt et al. 1987]. However, both this simulation and the Upfal and Wigderson simulation described above are non-constructive, i.e. it is proved that memory organization schemes supporting such simulations exist, but it is not shown how to construct one. Building such a scheme would, in fact, be more difficult than constructing a general expander graph, which is itself a well known open problem. Therefore the practical merits of these memory management techniques is questionable.

Herley and Bilardi achieved slightly better results, summarized in the following theorem, which applies if m is polynomial in n.

Theorem 2 *[Herley and Bilardi 1988] An EREW PRAM may be simulated on a Module Parallel Computer with redundancy* $O(\log m/\log\log m)$ *and slowdown* $O(\log n/\log\log n)$.

They also provide further discussion of the use of expander graphs in deterministic simulations.

In a recent paper an attempt is made to reduce the amount of memory blowup to a constant [Aumann and Schuster 1991], though the slowdown of the simulation is as much as $O(\log n(\log\log n)^2)$. A reduction in blowup is achieved through use of a information dispersal and recovery technique suggested by Rabin [Rabin 1989]. In the scheme a memory of size m is divided up into b chunks of size $b = m/d$, and with any d of the pieces the entire memory can be reconstructed. Therefore the memory blowup of the scheme is b/d, but both b and d may be chosen as $\Theta(\log n)$, and hence $b/d \approx 1$. A variable is stored in a block, and to access it one needs to access $(d + b)/2$ locations in its block. The scheme also allows the elimination of time stamps.

Constant memory blowup was also achieved in the paper [Hornick and Preparata 1991] through different techniques, though time stamps were still required in that simulation. The simulation was based on a different model from the MPC, one where there are more memory modules than processors, and hence where each module has fewer locations. This model is called the Distributed Memory Module Parallel Computer, or DMMPC, and the lower bounds mentioned above do not apply to it. The paper showed that with this model one can simulate an arbitrary step of a PRAM in $O(log^2 n/\log\log n)$ time with effectively constant redundancy, if the number of memory modules, $M = n^{1+\epsilon}$, and $\epsilon > 0$. Such an assumption will clearly reduce the contention to well below what one would expect in a normal MPC. Hornick and Preparata used the mesh of trees network to solve the routing and interconnection problem, as originally proposed by Luccio et al. [Luccio et al. 1990]. Such a network provides a physically realistic implementation of a bounded degree network, but requires an additional $O(n^2)$ simple switches for routing. A paper from Herley provides an effective solution to the deterministic memory management problem for the special case where $m = n$ [Herley 1989], though this case will rarely be seen in practice.

1.4.2 Routing and Interconnection

Naive deterministic solutions to the problem of simulating an MPC on a BDN will typically have good average case behaviour, but poor worst case behaviour. This is due to the problem of "hot spots", where particular memory access patterns may lead to many packets needing to traverse the same link. A more fundamental result, in the form of a lower bound for worst case routing time with an oblivious routing algorithm, was initially discovered by Borodin and Hopcroft [Borodin and Hopcroft 1982]. A routing strategy is oblivious if routing decisions for a packet are based solely on the source and destination of the packet, i.e. there is no information available about the global state of the machine. This is a realistic assumption in the most general case, though we will see later that sorting networks do not strictly conform to this oblivious restriction. Greedy methods are typical examples of such oblivious routing algorithms.

Borodin and Hopcroft's result was tightened slightly by Kaklamanis into the following:

Theorem 3 *[Kaklamanis et al. 1990] Any oblivious deterministic routing method on a degree d graph with n processors will do no better in the worst case than* $\Omega(n^{1/2}/d)$ *time for routing a permutation.*

A permutation occurs when each of n processors holds a request for a distinct memory location. This worst case behaviour will be seen in practice in various applications which depend on the execution of permutations that cause particularly bad hot spots. Typical examples are the bit-reversal phase of an FFT, or matrix transpose, which is a common subroutine in numerical applications. A good explanation of this problem of "Hot Spots" from the practical perspective may be found in the results of the RP3 project [Pfister and Norton 1985].

The problem we are concerned with is simulating a fully connected graph (an MPC) on a more realistic bounded degree graph, and in this section we would like

to find a deterministic solution to this problem which matches the lower bound of $\Omega(\log n)$ time. This lower bound is a simple consequence of the fact that a bounded degree network will have diameter of $\Omega(\log n)$, and hence even in the best case, with none of the contention problems discussed above, we will need $O(\log n)$ time to move from one end of the network to the other. We will assume that memory management has already been performed as described in the previous section, and therefore that any module will receive at most $O(\log n)$ requests.

A common alternative to the oblivious deterministic techniques already discussed is to use sorting networks. Routing on a sorting network is not oblivious, since it will depend on comparisons between packets in the system and hence upon some limited degree of global information. However, routing on a bounded degree sorting network is considered a relatively practical technique, though some sorting networks have constants hidden by the "big-O" notation that render them unrealistic to build.

One of the most practical sorting network for this application is the Batcher network [Batcher 1968]. It requires $O(\log^2 n)$ time, which is not optimal, but it has been shown that the circuit has quite small constants. The first $O(\log n)$ depth and hence optimal circuit for this problem was derived by Ajtai et al., and further improved by Leighton [Ajtai et al. 1983, Leighton 1985]. However, this circuit is specified in terms of expander graphs and the only explicit algorithms for the construction of such graphs results in graphs of very high degree and with large constants. Other related results are those of Upfal, who proposes that a butterfly graph with some degree of randomness in its wiring will result in an expander graph with high probability, and hence can be shown to also support routing in the optimal $O(\log n)$ time [Upfal 1989]. It is difficult to verify the expander characteristics of such a randomly wired butterfly, and the constants in such a network may be quite large [Leighton 1989]. This technique is entirely deterministic once the multibutterfly has been constructed. Optimal time routing

has also been shown within the context of non-blocking networks on the related "multi-Benes" network of [Arora et al. 1990].

The $O(\log n)$ circuits described here are assumed by most to provide an optimal solution to the deterministic problem of routing and interconnection, but until simple and verifiably good constructions of such circuits are found, an open problem still exists here; whether or not optimal time routing can be done with a non-expander based graph. Questions as to the practicality of expanders also apply to the many of the randomized memory management schemes considered in later sections. For more background on expanders see [Alon 1986, Paterson 1987].

1.4.3 Composition of Subproblems

After considering the three subproblems of a deterministic PRAM simulation we now consider the larger question of how to simulate a CRCW PRAM on a Bounded Degree Network. Firstly, we point out that a set of n CRCW memory requests may be converted to corresponding EREW requests deterministically in optimal $O(\log n)$ time as described earlier, as long as our BDN is suitably powerful to allow sorting in $O(\log n)$ time. This is the case if our BDN has the expander characteristics we described in the routing and interconnection discussion above; those originally proposed in [Ajtai et al. 1983]. Again, a preprocessing phase with sorting is used to eliminate concurrent requests, while a postprocessing phase is used to ensure concurrent reads are multicast back to the original requesting processors.

To solve the remaining problem of simulating an EREW PRAM on a BDN it is necessary to combine the techniques we've discussed earlier as solution of the subproblems. In randomized simulations, as we will see, the larger simulation problem of EREW PRAM on a BDN can be solved considerably faster than if we were to simply combine the solutions of subproblems we've described above. However, in the deterministic case this is not so. This is due to the high demands

made on the bounded degree network through the use of multiple copies, and requires a balance between having either fast reads or fast writes. If many copies of each variable are maintained, then writes will be slow, but if few copies are kept then reads may be slow. Consider the writing of a variable with redundancy r. The network will need $\Omega(r \log n)$ cycles to service n such writes if on average each variable is $O(\log n)$ distance away in the network from the requesting processor. It can also be shown [Alt et al. 1987] that reading will require $\Omega((m/4n^2)^{1/4r})$ cycles, and hence that the best one can do in such a simulation is

$$\Omega(min((\frac{m}{4n^2})^{1/4r}, r \log n)) = \Omega(\frac{\log^2 n}{\log \log n})$$

This important lower bound was established independently in both [Karlin and Upfal 1986] and [Alt et al. 1987]. The lower bound makes the assumption that all communications are *Point to Point*, i.e. that a separate message must be sent to update each variable, despite the numerous copies in the network during deterministic simulations. More efficient techniques, such as embedding spanning trees in the network and copying messages as they proceed down the tree, are used in various simulations [Alt et al. 1987]. No one has yet determined a more general lower bound, or established a better upper bound while using communications which are not point to point.

We earlier described how the paper [Herley and Bilardi 1988] provides a deterministic simulation of an EREW PRAM on an MPC with $O(\log n / \log \log n)$ slowdown. This result has been extended to provide a solution to the larger problem of a CRCW PRAM on a BDN and thereby achieve the above lower bound of $O(\log^2 n / \log \log n)$ for m polynomial in n. Their scheme assumes expander graphs in both the memory map used and in the BND interconnection network, and hence the main problem is again the difficulty in constructing these graphs.

Theorem 4 *[Herley and Bilardi 1988] A CRCW PRAM may be simulated on a Bounded Degree Network with redundancy* $O(\log m / \log \log m)$ *and slowdown* $O(\log^2 n / \log \log n)$.

1.5 Randomized Simulations

As mentioned previously, many of the problems involved in PRAM simulation have straightforward deterministic solutions that seem to have good average case performance, but have poor worst case performance. The role of randomization is to reduce the probability of this worst case taking place.

1.5.1 Memory Management

The worst case scenario in memory management is where each processor will request access to the same memory module on the same cycle. To make this case unlikely one may hash memory locations, i.e. map their locations from a logical space of consecutive addresses to a physical space where memory locations are randomly distributed over the n memory modules. After the memory locations have been initially hashed each processor is provided with appropriate hash functions such that it may quickly perform an address translation between the logical and physical spaces. When assessing the quality of a hashing scheme we will be considering the expected queue length, i.e. the largest number of memory requests that will need access to one module in a given cycle, as well as the time needed to evaluate the hash function and the amount of space required to store and compute the hash function. The slowdown from using randomized memory management techniques is the sum of the time to evaluate the hash functions and the memory contention time, i.e. the time to serve the expected queue length of serialized requests.

To hash memory we first select a hash function h at random from a class of such functions H. Ideally, elements of this class will be small and easy to derive, allowing the memory and the computing requirements of this initialization step to be small. Then, this hash function will be stored in each processor of

the MPC. During the simulation of the PRAM by the MPC we require that any PRAM memory location (or "key") with logical address a where $1 \leq a \leq m$ will be stored in physical address $h(a)$, where $1 \leq h(a) \leq m$. In this thesis we are concerned with the degree of memory module contention which results from a chosen hash function. As this module contention is independent of the distribution of addresses within a module, we will focus only on a hash function's ability to distribute addresses across modules, and neglect the question of how to distribute addresses within a module, as it is unimportant in terms of the time bounds of a simulation.

The goal of hashing is to reduce the set of hash functions H to one that is effective, i.e. one where with high probability the expected queue size will be small, meanwhile ensuring that all $h \in H$ may be computed quickly and require few random bits to construct. If we are unlucky and choose an h which is poorly suited for our memory access pattern, then once we determine this (e.g. by noting the particularly poor performance of an application) we may choose another such h and rehash memory. This is a potentially expensive process, but will occur rarely [Valiant 1990b].

Most hashing results have been based on the notion of "universal" hash functions, which were introduced in [Carter and Wegman 1979].

Definition 5 *Let A and B be two sets of memory addresses and H be a family of functions that map A onto B. H is a universal family of hash functions if for every $x_1 \neq x_2 \in A$ and $y \in B$ we have that $Prob_{h \in H}[h(x_1) = y \wedge h(x_2) = y] = 1/|B|^2$*

Intuitively a universal hash function is one where the chances of mapping two addresses of A into the same location in B is inversely proportional to the square of the size of B. Several constructions of such hash functions exist and they have been used widely. In the case of PRAM simulation we are concerned primarily with those hash functions that will result in expected queue lengths of $O(\log n)$,

such that the associated slowdown may be subsumed in the $\Omega(\log n)$ time which routing on a BDN requires. The notion of such a hash function has been formalized by [Mehlhorn and Vishkin 1984] in the following definition:

Definition 6 *A family of hash functions H which maps A onto B is $s_\mu - wise$ independent if $\forall y_1, ..., y_s \in B$, and $x_1, ..., x_s \in A$ with $x_i \neq x_j$ for $1 \leq i < j \leq s$:*

$$|\{h \in H : h(x_i) = y_i, i = 1, ..., s\}| \leq \mu \frac{|H|}{|B|^s}$$

This is a generalization of the definition 2 above, and similarly implies that the chances that s memory locations from the logical space will all be mapped onto the same memory module is μ/n^s. The hash functions most of interest in PRAM simulation are then $\log n$-wise independent, where μ may be any constant. A well known class of such functions are those consisting of polynomials of degree $O(\log m)$, which we refer to as H_1.

$$H_1 = \{h | h(x) = ((\sum_{i=1}^{k \log m} a_i x^i) mod\ p) mod\ n\}$$

for a prime number $p > m$, randomly selected values of $a_i < p$, and some constant $k > 1$. The use of H_1 was shown to allow simulation of a PRAM on a MPC in $O(\log n)$ time [Karlin and Upfal 1986]. H_1 requires $O(log^2 m)$ random bits to compute, in contrast to the $O(m \log n)$ bits that the construction of a entirely random hash table would require.

In [Mehlhorn and Vishkin 1984] tradeoffs between the complexity of the hash function and the expected maximum queue length were derived, and the following result was proven:

Theorem 5 *[Mehlhorn and Vishkin 1984] An EREW PRAM may be simulated on a Module Parallel Computer with slowdown $O(\log n / \log \log n)$ with high probability.*

In summary, randomized memory management in the form of hashing has proven an efficient and simple technique for simulating an EREW PRAM on an MPC. Additionally, the time to evaluate hash functions and to resolve the module contention which still exists can be subsumed in the time for routing in simulations which are mapped to BDNs, as we see in the next section. Practical work has suggested that in the average case even simple linear hash functions of the form $h(x) = (a_1 x + b) mod\ p$ can give reasonable performance [Ranade et al. 1988]. Further hashing results in the context of optimally efficient simulations will be discussed later. Also see [Mansour et al. 1990, Luccio et al. 1991, Matias and Vishkin 1991] for recent work on the subject.

1.5.2 Routing and Interconnection

We have seen that worst case behaviour for deterministic oblivious routing of permutations may require $\Omega(n^{1/2})$ time. However, we also know that random permutations may be routed with a simple greedy algorithm in $\Theta(\log n)$ time on interconnection networks such as the hypercube and butterfly [Valiant 1983]. Therefore finding an efficient random solution to the routing and interconnection problem is akin to making all permutations behave like random permutations.

The common way of doing this while making no assumptions about the memory mapping is called *two-phase random routing.* In the two phase approach a permutation is realized by first sending each packet to a random destination, and then sending them from the random destination to the final destinations specified by the original packet. This technique was originally suggested by Valiant, and was shown to perform as if each phase was totally random, independent of the permutation specified by the user [Valiant 1982, Valiant and Brebner 1981], and therefore achieves the $\Theta(\log n)$ bound desired. Initially this technique may seem counterintuitive, as it appears to double the distance any packet needs to travel. However, Valiant has further shown that packets must travel at least twice the

diameter in such oblivious routing algorithms, and hence this scheme is optimal [Valiant 1983].

Valiant originally described these techniques in terms of interconnection networks with logarithmic degree, such as the hypercube, making them not directly applicable for a network with fixed degree. Furthermore, he assumed that the nodes of the hypercube could send data out on each link at each cycle. Upfal then adapted the techniques to the more standard model of a BDN, where only a constant number of messages can leave or enter a node in a given cycle [Upfal 1984b]. Such randomized routing techniques have since been routinely used [Aleliunas 1982, Pippenger 1984, Karlin and Upfal 1986].

Once such techniques were established to allow $\Theta(\log n)$ expected routing times on bounded degree networks, researchers attempted to reduce the queue size, i.e., the number of memory locations at each processor required to hold messages which are in transit. In [Pippenger 1984] a randomized strategy which requires only constant length queues was established, though the scheme allowed a small probability of deadlock occurring. More recently, Ranade gave a straightforward algorithm with similar characteristics; $O(\log n)$ time and $O(1)$ length buffers, which had no such deadlock problems [Ranade 1991]. This paper has been particular influential due to the simplicity of its approach, and the practical use of combining to support CRCW operations. Ranade's routing scheme, strictly speaking, is deterministic, but depends on randomized memory mappings to achieve its time bounds, and so is included here. Lower bounds related to queue size may also be seen in [Krizanc 1991].

1.5.3 Composition of Subproblems

In order to simulate a CRCW PRAM on a BDN we may first use the same deterministic preprocessing suggested previously to eliminate concurrent access, again assuming our BDN is connected as to allow sorting in $O(\log n)$ time. Since this

optimal deterministic solution exists there is clearly no need for a randomized solution.

In the case of randomized simulation the direct simulation of an EREW PRAM on a BDN has a substantially smaller slowdown than would be seen by simply combining the solutions of the subproblems described above. With the reduced bandwidth requirements of a randomized simulation, a direct simulation of an EREW PRAM on a BDN can execute with only optimal $\Theta(\log n)$ slowdown. This direct simulation allows the $O(\log n)$ network routing time to be followed by the $O(\log n)$ module contention delay, such that the costs of the two phases are added together rather than multiplied together. Such an optimal simulation was first shown by [Karlin and Upfal 1986].

Theorem 6 *[Karlin and Upfal 1986] A CRCW PRAM may be simulated on a Bounded Degree Network with slowdown* $\Theta(\log n)$ *with high probability.*

1.6 Summary of Existing Results

Simulation	Slowdown
CRCW $\rightarrow$ EREW	$\Theta(\log n)$
EREW $\rightarrow$ MPC	$O(\log n / \log\log n)$
CRCW $\rightarrow$ MPC	$\Theta(\log n)$
MPC $\rightarrow$ BDN	$\Theta(\log n)$
EREW $\rightarrow$ BDN	$\Theta(\log^2 n / \log\log n)$
CRCW $\rightarrow$ BDN	$\Theta(\log^2 n / \log\log n)$

Figure 1–6: Upper bounds of Deterministic Simulations

To summarize we now provide tables containing the upper bounds for known solutions to PRAM simulation problems. Some of the solutions will be incor-

porated into our own simulations, which are developed in later chapters. Table 1–6 gives deterministic results, while table 1–7 summarizes randomized simulation results.

Simulation	Slowdown
CRCW → EREW	—
EREW → MPC	$O(\log \log n \log^* n)$
CRCW → MPC	$\Theta(\log n)$
MPC → BDN	$\Theta(\log n)$
EREW → BDN	$\Theta(\log n)$
CRCW → BDN	$\Theta(\log n)$

Figure 1–7: Upper bounds of Randomized Simulations

Chapter 2

Efficient Simulations

In the previous chapter we determined the quality of a simulation by the slowdown it incurred. We now consider the efficiency of a simulation. In particular, if an n processor PRAM requires time T to execute a program, then we are interested in simulations where the program may be simulated in time T' on p processors such that:

$$E = \frac{Tn}{T'p} = O(1)$$

Such simulations are typically referred to as either constant time-processor product or simply as efficient simulations. One trivial efficient technique is to simulate a PRAM on one serial processor by simply executing the n PRAM instructions of each cycle in round-robin fashion. Efficient simulations are those that require no more steps than does this trivial solution, despite the fact that memory access in parallel solutions generally require $\Omega(\log n)$ time. To produce such simulations we need to mitigate the effect which the slowdown of our simulation has on the utilization of our processors. In this section we first try to build an intuition about such simulations, and then describe existing results.

We will refer to the number n as the number of processes or threads in the PRAM, while p is the number of processors used in the machine upon which the simulation is taking place. The ratio $s = n/p$ is called the degree of parallel

slackness of the simulation, and as it increases beyond one we may begin to pipeline memory accesses and thereby attempt to hide the slowdown of memory accesses. More specifically if one thread of the PRAM requests a memory access, and $n/p > 1$, then instead of the simulating processor remaining idle from the time the request is initiated until it is fulfilled, it may context switch to another thread of the PRAM in order to maintain high utilization (see figure 2–1). If each processor uses a simple round-robin scheduling strategy and if our simulation has slowdown L, then using L threads on each processor may allow an efficient simulation. However, pipelinable solutions to routing and memory management issues still need to be solved to allow efficiency. Such techniques are now also common in practical research of parallel computing, and are frequently referred to as multithreading techniques [Weber and Gupta 1989, Boothe and Ranade 1992].

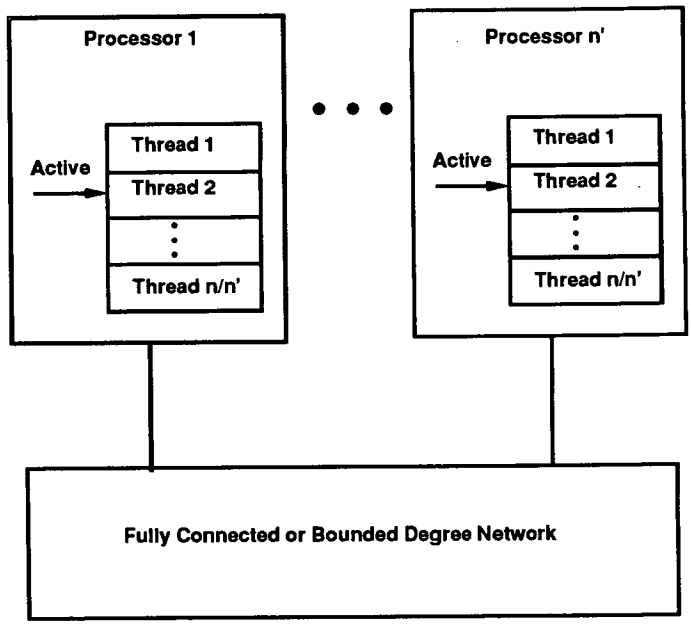


Figure 2–1: A Multithreaded Architecture.

2.1 MPC Based Simulations

As the slowdown caused by the latency of the simulating network increases, then a larger degree of parallel slackness may allow an efficient simulation. However, if the bandwidth of a network is too small, then such techniques are insufficient, and any simulation will be necessarily inefficient. There are two types of delay which may be incurred in a network. Communications delay which is attributable to the sheer distance a message needs to travel may be amortized through pipelining. However, delay which is caused by contention inside an overloaded network may not be hidden in this way. More specifically, if a network has no contention and has latency L, and memory requests are initiated at times 0 and 1, then the requests will be fulfilled at times L and $L+1$ respectively. If, however, a network has a delay L which is solely attributable to contention for resources, and memory requests are initiated at times 0 and 1, then they will be fulfilled at times L and $2L$ respectively. For these reasons efficient simulations can not take place on traditional fixed degree networks, as these networks do not have the bandwidth necessary to ensure that no contention will take place [Kruskal et al. 1990]. Therefore efficient simulation are generally targeted at fully connected Module Parallel Computers, which implies that only the memory management problem is relevant (clearly routing on a fully connected graph is trivial).

Probabilistic solutions to the memory management problem (in the form of hashing) are often faster than deterministic solutions, and therefore are commonly used for efficient simulations. In chapter 1, finding randomized solutions to the memory management problem which had slowdowns of less than $O(\log n)$ was not a priority because they were typically used in conjunction with routing techniques which had slowdowns of $\Omega(\log n)$. However, since in efficient simulations we will be working only with MPCs, we will want the smallest possible slowdown, and we may be willing to pay a higher price in terms of memory blowup or amount of

random bits required. The class of hash functions, H_1, as introduced in the last chapter, will not be useful for our purposes here, as polynomials of degree $\log n$ will require $\Omega(\log n)$ time for evaluation, and this delay is unpipelinable (i.e. $\log n$ such evaluations will require $\Omega(log^2 n)$ time). For efficient simulations, we need different classes of hash functions, where the evaluation time is $O(1)$.

One of the most obvious hash functions with constant evaluation time are those that are similar to H_1, but have constant degree, referred to here as H_2.

$$H_2 = \{h | h(x) = ((\sum_{i=1}^{d} a_i x^i) mod\ p) mod\ n\}$$

where $p > m$ is a prime number and d is a constant.

Hashing functions of the class H_2 were used in the efficient simulation of [Kruskal et al. 1990]. In the paper they show that a PRAM with $O(n^{1+\epsilon})$ threads, where $\epsilon > 0$, may be simulated efficiently on an MPC with n processors and $O(n^{\epsilon})$ parallel slackness.

Tradeoffs were established between the time necessary to compute a hash function and the number of random bits needed for the computation in [Siegel 1989]. This resulted in classes of hash functions which are $\log n$-wise independent (and therefore better than H_2 above) but that can be computed in $O(1)$ time. The hash functions took the form of bipartite graphs mapping address space A to B, but these graphs were by necessity *weak concentrators*. Weak concentrators are in the family of expander graphs, and hence again limited by the lack of explicit constructions. The Siegel paper also showed straightforward techniques for extending the $O(\log n)$ slowdown simulation of [Ranade 1991] to an efficient simulation through the addition of parallel slackness.

The hash functions of [Siegel 1989] were then used by Valiant to derive an efficient simulation with expected delay $O(\log n)$ time [Valiant 1990b]. The simulation is based on a hypercube model, where data may be passed across each of the $\log n$ links of each node at every cycle, so does not strictly qualify as either

a fully connected graph or a fixed degree network, but still is more realistic than many of the networks assumed in efficient simulations.

Similarly, an efficient simulation was shown on a fully connected network in [Dietzfelbinger and Meyer auf der Heide 1990] with $O(\log n)$ delay. They used a new class of hash functions, which are composed of $r+1$ different polynomials from H_2, where $r > 1$ is a constant. One of the functions is used to split the set of keys into r buckets, and the other then determines the offset of the key within the computed bucket. We call this new class H_3.

$$H_3 = \{h(f, g_1, ..., g_r) | h(x) = g_{f(x)}(x)\}$$

where $f \in H_2^r$ and $g_1, ..., g_r \in H_2^n$, and we have designated the range of the polynomials by superscripts.

[Karp et al. 1992] developed hash functions that were $\log n$-wise independent using an approach similar to H_3, but where the constituent functions are made up of weak concentrators from [Siegel 1989], instead of the polynomials of constant degree as in H_2. The authors then used double hashing, where each memory location of the PRAM is hashed into two or more locations of the MPC using two or more unique functions from the class H_3. The resulting simulation has a slowdown of only $O(\log \log n \ log^* n)$, and therefore requires only modest parallel slackness to be efficient. The algorithm also benefits from the technique of delaying writes when memory contention prevents a write from being executed in a single cycle. The result is summarized below:

Theorem 7 *[Karp et al. 1992] An EREW PRAM may be simulated on an MPC efficiently with slowdown $O(\log \log n \ log^* n)$ with high probability.*

The authors also showed that this simulation may support CRCW operations if we target the simulation at a Distributed Memory Machine (DMM) instead

of the MPC. Each processor of the DMM has access to a *communications window*, which serves as a single cell of a CRCW ARBITRARY shared memory, and thereby provides CRCW operations. Other related DMM results appeared in [Dietzfelbinger and Meyer auf der Heide 1993].

2.2 Simulations for Generalized Networks

In this thesis we are primarily interested in efficient simulations which are targeted at networks which are as close to Bounded Degree Networks as possible. Here we describe other work that has previously considered the ramifications of attempting to use multithreading for efficiency on networks other than Module Parallel Computers.

By far the most influential such consideration is the Bulk Synchronous Parallel (BSP) Computer, as described in [Valiant 1990b]. Valiant had previously provided an efficient simulation targeted at a powerful variant of a hypercube, which was similar to the work of [Upfal 1984a].

BSP is an attempt to generalize such simulations to arbitrary networks. The approach is to firstly provide parameters which specify some of the salient features of a network regarding the bandwidth and routing capabilities for a network. BSP can be considered a framework for providing a simulation which is as efficient as possible given these network capabilities.

BSP simulations use multithreading techniques to hide latency when sufficient bandwidth exists to service the multiple memory requests being issued by each processor during simulation of a PRAM step. However, in the case of a lower bandwidth network, the BSP programming model will be altered such that non-local communications events are only able to be issued periodically. In [McColl 1992] this reduction in non-local communications is refereed to as *communications slackness*. Concisely stated, the BSP solution to providing efficiency on bounded

degree networks is to not support a full PRAM model, but instead to provide a more restrictive model which makes less demands on the bandwidth of the network. Work which quantifies the bandwidth demands of particular algorithms can be seen in [Gerbessiotis and Valiant 1992], and related BSP work also appears in [Bisseling and McColl 1994].

Another related model is the Logp model [Culler et al. 1993]. The Logp model extends the parameterized scheme of the BSP model to include a total of 4 network performance characteristics. The emphasis of the model is to provide an accurate correlation between the complexity of an algorithm designed for the model, and its subsequent performance for a given architecture, once the parameters of that architecture have been determined. Though the emphasis is not on multithreading and efficiency, Logp results are ideally general enough to apply to both networks that allow efficient support of the model and those that do not. With respect to multithreading, the authors also point out the significance in context switch overhead in practical usage of such techniques. Another attempt at a parameterized modeling of PRAM performance can be seen in [Harris and Cole 1993].

Two more recent works provide insight into the specific problem which we consider in this thesis, though through utilization of different techniques. A practical project in the commercial sector is attempting to build a Tera machine [Alverson et al. 1990], a 3-D mesh based multiprocessor system that uses multithreading to hide latency. A precursor of this machine which was designed by the same primary architect was the Denelcor Hep [Smith 1978], which also used multithreading, but not a mesh type interconnection network.

A recent PhD thesis also addresses the issue efficient PRAM simulation on a mesh of processors [Leppanen 1993]. Similarly to the Tera machine, this work exploits the fact that a mesh where some nodes are simple switches, rather than full processors, can provide the bandwidth and switching capacity necessary for full latency hiding. The work also focuses on use of combining queues to support CRCW simulations, similar to [Ranade 1991].

The results of the following chapters differs substantially from this previous work in the area. One may compare the work to that regarding the BSP model by pointing out the following: BSP chooses to deal with the high communications requirements of multithreaded simulations by weakening the target PRAM model, such that the user no longer has the power to request shared memory access sufficiently to cause network contention. Our technique may be seen as dealing with the same pressure of high communications requirements by augmenting our network with additional bandwidth, and therefore preserving the full PRAM model. This direction is suggested by the fact that traditional bounded degree networks, such as a mesh, will only support variants of the BSP model that are quite obtrusive; e.g. a model where access to shared memory may only occur every $O(n^{1/2})$ steps on a n processor mesh. Our choice of relatively simple topologies such as the mesh is reinforced by work presented in [Bilardi and Preparata 1992], where it is suggested that as clock speeds increase in the future, the long wires of networks such as hypercubes will render them inappropriate for multithreaded simulations.

The work of [Leppanen 1993] and [Alverson et al. 1990] are similar to this thesis in the sense that they strive to support PRAM style models, but they diverge primarily in their use of resources. As we will now show, the reduction of the number of processors in a high diameter network has substantial performance benefits for a network, and these benefits are unavailable if we use switching nodes to augment our network bandwidth. We contract the diameter and hence routing time of the network, as well as preserving resources by reducing the number of nodes in the graph. Additionally, the use of simple non-combining nodes allows us to establish a hierarchy of PRAM simulation complexity by providing lower bounds on the use of general sorting routines to support CRCW access. Some ideas related to our discussion of concurrent access are presented in [Kruskal et al. 1990].

Chapter 3

Optimal Efficiency and Bounded Degree Networks

Though the majority of processor efficient PRAM simulations have been targeted for Module Parallel Computers, we intend to focus on a more practical platform; that of variants of traditional Bounded Degree Networks. In this chapter we firstly consider general issues regarding the degree of parallel slackness necessary to hide the latency of a simulation on a bounded degree network of arbitrary diameter. Then we discuss the relationship between this slackness and the amount of bandwidth available in such networks.

3.1 Processor Counts and Slackness

Assume that we have a simulation of a chosen PRAM model running on an arbitrary (and ideally realistic) interconnection network, such that the delay of the simulation is $L(n)$ for an n processor machine with no multithreading. The question we address now is how to convert this into an optimally efficient simulation, running on a network with an equivalent topology. We will need memory management and routing techniques that are fully pipelinable, but for the moment we

assume these will be provided, and consider only the necessary processor counts. We expect the optimally efficient simulation to run on $p < n$ processors, and we define s as the degree of parallel slackness, i.e. the number of threads being run concurrently on each processor. We define an optimally efficient simulation as following:

Definition 7 *An* optimally efficient simulation *is a PRAM simulation that has the following three characteristics: Firstly, that the delay of the simulation is on the order of the diameter of the network, i.e.*

$$L(p) = O(d) \tag{3.1}$$

Secondly, that the time processor product is of the same order as that of the PRAM model being simulated, i.e.

$$E = \frac{Tn}{T'p} = O(1) \tag{3.2}$$

And lastly, we require that the simulation use only as much slackness as is required to hide the delay of the simulation, i.e.

$$s = L(p) \tag{3.3}$$

In our terminology we require all three attributes for a simulation to be considered optimally efficient. Previous authors have assumed that condition 3.2 is alone sufficient to warrant "optimal efficiency" [Valiant 1990b]. However, we consider simulations which hide delays which are larger than the diameter of the network to be inherently non-optimal, and this leeds naturally to the more demanding definition.

The efficient simulations surveyed in previous chapters generally assumed an underlying network with p processors which will have a diameter of $O(\log p)$. They typically assume slack $s = \log n$, and $p = n/\log n$. Therefore, they are neglecting the fact that the diameter of the network decreases as the number of processors is decreased to allow multithreading. Though the difference is negligible on such low

diameter networks, a truly optimal simulation as defined above would be described as having $p = n/\log p$ processors, and $s = \log p$ slack. Providing enough slack to hide the latency of a n rather than p processor simulation is hardly a substantial over-estimate for such $\log p$ diameter networks, but in the general case of higher diameter networks it is quite significant. Restating with more formality we note that previous efficient simulations have tried to maintain the following conditions:

$$n = ps \tag{3.4}$$

$$L(n) = s \tag{3.5}$$

Equation 3.4 provides that every thread of the original PRAM algorithm is in fact simulated, where 3.5 ensures that the entire delay is hidden. However, note that the slackness of the simulation is dependent upon the delay of the n processor non-multithreaded simulation, rather than on the delay of the simulation based on the reduced p processors of our multithreaded simulation. The correct equations which will use in the following are:

$$n = ps \tag{3.6}$$

$$L(p) = s \tag{3.7}$$

where the delay of the simulation in equation 3.7 is a function of the number of physical processors in our smaller multithreaded machine. This modification will be particularly meaningful given the high diameter of the ring, as well as that of networks we consider in following chapters.

As an example consider we have a simulation running on a ring. Clearly the diameter of a ring with p processors is $O(p)$. Substituting into equations 3.6 and 3.7 we get:

$$\begin{aligned} n &= pL(p) \\ &= p^2 \end{aligned}$$

therefore

$$p = n^{1/2}$$

and

$$s = n^{1/2}$$

So any ring based simulation that is potentially optimally efficient will use $p = n^{1/2}$ physical processors, and $s = n^{1/2}$ threads per processor.

We will also be considering meshes of arbitrary dimension. For the case of a two dimensional mesh an optimally efficient simulation will have $L(p) = \Omega(p^{1/2})$, and hence

$$p = n^{2/3}$$

which corresponds to a $n^{1/3} \times n^{1/3}$ mesh. If we consider the most general case of an r dimensional mesh, assuming $L(p) = \Omega(p^{1/r})$ we see:

$$p^{(1+r)/r} = n$$

or

$$p = n^{r/(1+r)}$$

So an r dimensional mesh capable of supporting an optimally efficient simulation will have dimensions $n^{1/(1+r)} \times n^{1/(1+r)} \ldots \times n^{1/(1+r)}$.

3.2 Bandwidth Requirements

We now provide some detail to our claim that a traditional bounded degree network will not provide enough bandwidth to allow an optimally efficient simulation. We then determine how much bandwidth is necessary in the worst case. We consider optimally efficient simulations so we may assume that the delay of the simulation is $O(diameter)$ of our network.

We consider an attribute of a simulation which we call the *contention factor*, which we define as the bandwidth required by the simulation in a given PRAM step, divided by the available bandwidth in the underlying network. I.e.

$$contention\text{-}factor = \frac{bandwidth\text{-}required}{bandwidth\text{-}available}$$

We consider bandwidth in units of link-cycles, defined as the amount of bandwidth used by one request when traversing a single cycle. Furthermore we assume that each link of a network is made up of multiple wires, where each wire has enough bandwidth to move exactly one request per cycle across the link. Most networks have, by default, one wire per link, though this is not the case for the networks considered in most of this thesis.

Optimally efficient simulations require that all requests are pipelined in the network such that each request arrives at its destination in $O(diameter)$ time, and that the processor is able to inject $O(diameter)$ requests into the network during that time. Clearly any contention for network resources during routing will disallow this form of pipelining, and hence only a simulation with $contention\text{-}factor = O(1)$ is potentially optimally efficient. Note that now we are considering only the capability of a network to support an efficient simulation, whereas in late chapters we will discuss specific algorithms that provide routing and memory management for an optimal simulation which may be run on networks with $contention\text{-}factor = 1$.

3.2.1 Ring Contention Factors

As a simple example we consider the contention factor for a typical ring. Given the processor number suggested above, an optimal ring based simulation will have $p = n^{1/2}$ processors, $s = n^{1/2}$ slackness per processor, and have delay of $O(p)$ time. During simulation of a PRAM step the p processors will each inject s requests into the network, each of which will travel up to p distance, and therefore occupy $\Omega(p)$

wires, each for one cycle. Therefore the bandwidth requirements for that step are:

$$p \cdot s \cdot p = p^3$$

wire-cycles. Available bandwidth for a simulation step of $O(p)$ time is p links for $\Omega(p)$ cycles each, or a total of:

$$p \cdot p = p^2$$

link-cycles. Therefore the contention factor for a ring is:

$$contention\text{-}factor_{ring} = \frac{p^3}{p^2} = p$$

Here we are interested in augmenting the bandwidth of traditional BDN type networks to allow them to support optimally efficient simulations. In particular, we are interested in maintaining the general topology of networks, but increasing the bandwidth per link to allow the communication of more than one request per cycle across them. The above result suggests that any such multithreaded ring network which has a contention factor of $O(1)$ will require $\Omega(p)$ wires per cycle. This corresponds to the *fat ring* network discussed in following chapters.

3.2.2 Mesh Contention Factors

Similarly to our ring discussion, and recalling the processor count considerations, we observe that the bandwidth requirements for an optimally efficient simulation on a r dimensional mesh, will be as follows. We will have $p = n^{r/r+1}$ processors, each of which will inject $p^{1/r}$ requests per PRAM step, and each request will travel $\Omega(p^{1/r})$ links before arriving at its destination. Available bandwidth will be the $\Omega(p)$ links of any bounded degree mesh, multiplied by the $\Omega(p^{1/r})$ time of the optimally efficient simulation. This results in:

$$contention\text{-}factor_{mesh} = \frac{p^{(r+2)/r}}{p^{(r+1)/r}} = p^{1/r}$$

This suggests that in order to augment a traditional mesh to allow it to support an optimally efficient simulation we will require that each link may service up to $p^{1/r}$ requests in a given cycle, and hence will have $\Omega(p^{1/r})$ wires.

Chapter 4

Fat Rings

In this chapter we describe an efficient simulation of an EREW PRAM on interconnection network which we refer to as a *Fat Ring*. As the name implies the network is very similar to a traditional ring, except for the fact that it has a capacity per link that is higher than one packet per cycle. We choose the term *fat* to imply increased bandwidth as in the *Fat Tree* of [Leiserson 1985]. We will show how the fat ring network may be combined with memory management, and routing techniques to achieve an optimally efficient EREW PRAM model. In a later chapter we show that through the use of simple preprocessing and postprocessing phases the model can also accommodate CRCW requests, though with less efficiency.

4.1 The Ring Model

Rings are one of the first interconnection networks considered for parallel computing, and have recently begun regaining some of their previous popularity. A ring consists of p nodes connected with a set of links, L, such that a link exists between any two nodes whose node IDs differ by one. More precisely, $L = \{(p_i, p_j) | p_i, p_j \in 1...p-1, j = i \pm 1 Mod p\}$.

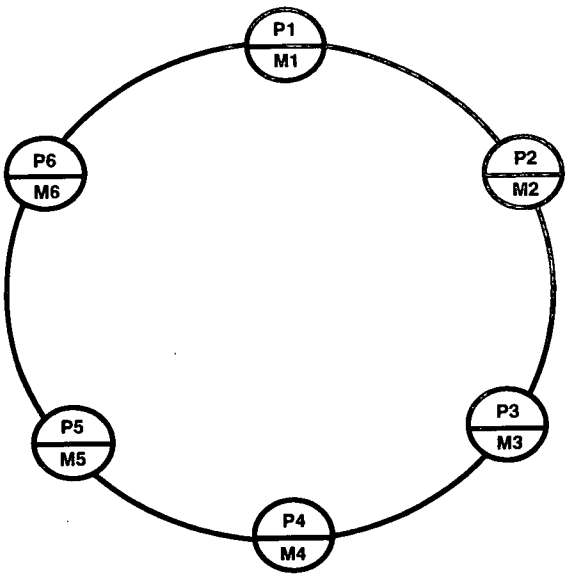


Figure 4–1: A six processor ring.

The obvious disadvantage of ring networks is their large diameter, which grows linearly with the number of processors, i.e. $d = O(p)$. However, the simplicity of both constructing rings and of performing basic operations such as routing and sorting on rings has contributed much to their attractiveness. The Kendal Square Research machine is one recent example of their practical use [Kendal Square Research 1991]. The networks structure serves to simplify the hardware cache coherency scheme of the KSR, as a message routed around the ring can be guaranteed to be seen by every node. Similarly, workstation clusters are commonly connected as rings.

Given its longevity and simplicity it seems appropriate to ask if the ring can be extended to serve as the interconnection network underlying an optimally efficient PRAM simulation. In the next section we define more clearly our target model of shared memory.

Here we are concerned primarily with EREW shared memory. Our general goal is to support shared memory such that we can make specific claims on the latency

of any memory access, and then try our best to hide these latencies as best as we can. To succeed in hiding the latency of memory accesses we must ensure that a processor is never idle for more that a constant number of cycles after issuing a memory request, despite the potentially high latency of such a request.

As outlined in the previous chapter, an optimally efficient simulation on a ring will have $p = n^{1/2}$ processors, and each processor will be running $s = p$ threads. We firstly address the issue of memory management for such a machine. Then we go on to suggest a way that the necessary bandwidth may be included in a ring. Finally we discuss fat ring routing, which then leads us to our EREW simulation on the fat ring.

4.2 Memory Management

An effective memory management scheme is a vital component of any simulation, but efficient simulations make even more difficult demands on such schemes. One attribute we have already described in Chapter 2 is the pipelinability of memory management; i.e. any phase of an algorithm can not require more than $O(1)$ time from a processor if that phase is to be repeated. Another attribute of efficient simulations which makes demands on the memory management scheme is that we are issuing more requests than there are processors, as we execute multiple threads per processor and each thread will potentially want to make a memory request on any given cycle.

The requirements of our memory management scheme for the fat ring are implicit from the above information regarding the number of physical processors to be used. We are trying to provide an $O(d) = O(n^{1/2}) = O(p)$ time simulation, and we will be issuing up to a total of p^2 memory requests to be serviced in any one PRAM step. As each processor can service at most one request per cycle, and a processor will need to service all its request within the delay of the simulation $O(p)$,

at most $O(p)$ out of the $O(p^2)$ requests can be directed towards any one module. It can be argued that this is a perfect mapping, in the sense that the requests need to be distributed evenly, with $O(p)$ requests arriving at each processor. Remember that we are concerned with EREW requests in this chapter, so each request will be destined for a unique address.

Unfortunately there is no obvious deterministic scheme that can provide such a good distribution. To use a majority scheme we would need a scheme which uses only a constant number of copies, as updating a non-constant number of copies would require more than constant time, and therefore violate the pipelinability premise of our simulation. Very few such schemes exist, with the most likely example being [Pietracaprina et al. 1994]. This work is particularly laudable in that it provides explicit constructions for the expander type graphs it exploits. However, the power of these graphs is less than that of graphs which do not have explicit constructions. Correspondingly, it appears unlikely that such explicit schemes can result in deterministic memory management that is powerful enough to ensure that no one node is overloaded enough to slow the simulation to below the performance required for optimal efficiency. However, further consideration of deterministic efficient simulations is an interesting topic for further work.

Alternatively, we now show that a quite simple randomized scheme can serve our needs. It is a class of hashing functions which we call H_*, closely related to one we defined defined earlier:

$$H_* = \{h | h(x) = ((\sum_{i=1}^{d} a_i x^i)) mod\ m\}$$

where m is a prime number and d is a constant. If m is not a prime, we may substitute in this equation the first prime m' which is larger than m. The fact that the degree of the polynomial, d, is a constant means that we may evaluate the function in constant time, thereby fulfilling our pipelining constraints. Some research has even suggested that such hash functions of low degree behave better than those of $O(\log n)$ degree [Engelman and Keller 1993].

We assume each virtual shared memory address is a single value such as x, such that $1 \leq x \leq m$. To access x the processor must first determine the physical address, represented by a tuple of the form (i, j), where $1 \leq i \leq p$ is the module ID and $1 \leq j \leq m/p$ is the memory element within that module. Both these values are incorporated in the result of the hashing equation, $h(x)$. The module ID can be determined simply as

$$i = \frac{h(x)}{(m/p)}$$

for an p processor network, whereas the address within the module, j, is computed as

$$j = h(x) \bmod (m/p)$$

We are concerned with the effect module contention has on the delay of our simulation, and hence we will now focus on the ability of our hash functions to effectively distribute requests among modules. Distribution of requests within the module has little effect on performance given our assumption of no memory hierarchy within a module.

We now provide details as to the suitability of this hash function H_*. But first we need the use of the following lemma from [Mehlhorn and Vishkin 1984]. In the following notation R is used as the maximum queue size, which for clarity we often refer to as the maximum number of requests arriving at any one module while simulating a PRAM step.

Lemma 2 *[Mehlhorn and Vishkin 1984] If m is prime, then the probability of having more than k requests arrive at any one module given a total of v requests being issued is:*

$$Pr\{R \geq k | h \in H_*\} \leq \binom{v}{k} p^{1-k} e^{kp/m}$$

Proof: This proof follows the lines of that presented in [Valiant 1990a]. Consider a polynomial of the form:

$$h(x) = ((\sum_{i=1}^{d} a_i x^i))mod\ m\}$$

and some set of k addresses $j_1, ..., j_k$ and destination memory locations $l_1, ..., l_k$. There will be at most one polynomial of this form such that all these k addresses are mapped to these memory locations, i.e. $h(j_r) = l_r$ for all $r = 1, ..., k$. If we now consider only the destination module instead of destination address, i.e. $l_1, ..., l_k \in < p >$, then there will be at most $(m/p + 1)^k$ such functions $h' \in H'$, as there are up to $(m/p + 1)$ memory locations in each module.

Now if we consider a fixed set of k out of the total of v accesses, the chances they are mapped under a randomly chosen h to the same module in $< p >$ is the number of such functions that map them to the same module, divided by the total number of hash functions:

$$Pr(R \geq k) = \frac{(m/p + 1)^k}{|H|}$$

If we assume $d = k$ for H above, then the total number of unique hash functions, $|H|$ is:

$$|H| = (m/p)^k p^k$$

Leading to:

$$\begin{aligned} Pr(R \geq k) &= \frac{(m/p + 1)^k}{(m/p)^k p^k} \\ &= (1 + p/m)^k p^{-k} \\ &< e^{kp/m} p^{-k} \end{aligned}$$

Hence the chance that some set of k of the v requests are all mapped under a chosen h to the same memory module in p is less than:

$$\binom{v}{k} p^{1-k} e^{kp/m}$$

□

If the size of our memory m is not prime, then we instead compute our hash functions using an appropriate m', such that m' is the smallest prime that is larger than m.

Before we consider the behaviour of H_*, we firstly prove the following useful lemma.

Lemma 3 *For any* $0 < y < x$,

$$\binom{x}{y} < \left(\frac{xe}{y}\right)^y$$

For a proof of this lemma we refer the reader to [Leighton 1992].

The following theorem shows that the class of hash functions H_* serves our purposes with high probability. The following is formulated in terms of the number of physical processors, which is of course $p = n^{1/2}$.

Theorem 8 *Consider any set of* $n = p^2$ *EREW memory requests which address any of* m *memory locations spread out in* p *modules, such that* $m \geq n^2$. *If the memory is hashed with a function* h *selected from the class* H_*, *then with high probability no more than* $O(p)$ *requests will be destined for any one memory module.*

Proof: We will be using lemma 2 by plugging in the relevant values from our application. But first we simplify the equation through use of lemma 3 and by bringing the p^{1-k} term inside the exponential, such that lemma 2 becomes:

$$Pr\{R \geq k | h \in H_*\} \leq \binom{v}{k} p^{1-k} e^{kp/m} = \left(\frac{ve}{kp}\right)^k p e^{kp/m}$$

In this particular case $v = p^2$ and $k = O(p)$. We will choose $k = 4p$ to simplify later calculations. Additionally, as $m \geq n^2$ and $n = p^{1/2}$, we know $m \geq p^4$.

Plugging in relevant parameters:

$$Pr\{R \geq k | h \in H_*\} \leq \left(\frac{p^2 e}{4p^2}\right)^{4p} p e^{4p^2/p^4}$$
$$= \left(\frac{e}{4}\right)^{4p} p e^{4/p^2}$$

We may then bring all terms inside the exponential.

$$Prob(R > 4p) < \left(\frac{e}{4} p^{1/4p} e^{1/4p^3}\right)^{4p}$$

Both the functions $p^{1/4p}$ and $e^{1/4p^3}$ are small, and may be bounded by a small constant. E.g. if we consider reasonably large values of p, then their product is very close to one. Therefore we may rewrite the entire expression as simply:

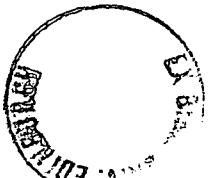
$$Prob(R > 4p) < \alpha^{4p}$$

for some value of $0 < \alpha < 1$. With our chosen value of $k = 4p$ we expect $\alpha \approx e/4$ for reasonable values of p. □

4.3 The Interconnection Network

Routing on a ring is a simple task; in our case we choose to route all messages in the same direction, e.g. clockwise. This simplifies routing decisions, and allows us to quantify the maximum degree of contention better than if messages moved in both directions, and the savings in time with bidirectional routing would be small.

Earlier we pointed out that a traditional ring does not provide sufficient bandwidth to support an optimally efficient simulation, and we suggested that each link should instead be made to allow p requests to travel across it in a given cycle. We suggested that each link of a fat ring must have at least p wires. In this case the entire network has a capacity of $O(p^2)$ requests per cycle, and results in a



contention factor of:

$$contention - factor_{fat-ring} = \frac{p^3}{p^3} = 1$$

which is a necessity to allow an optimally efficient simulation. Figure 4–2 is a schematic of such a network, shown with six multithreaded processors, six threads per processor and six wires per link. Clearly each processor will also have a memory module and routing hardware, which is not shown.

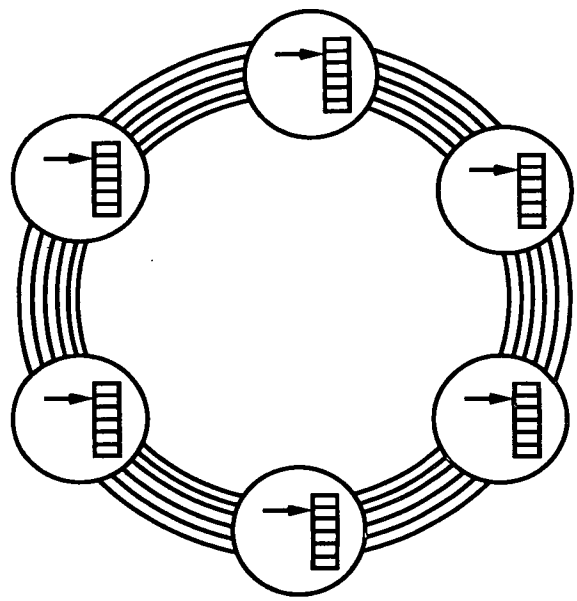


Figure 4–2: A six processor fat ring.

Despite the fact that our fat ring has increased bandwidth, the basic assumptions of BDN processors still largely apply to each processor. Namely, the memory module can service only one request per cycle, and the processor can only inject up to one request per cycle into the network. One additional job for a processor in a fat ring is to handle the arrival and forwarding of up to p requests in a given cycle.

4.4 Fat Ring Node Architecture

The primary goal of our efficient fat ring simulation is to eliminate virtually all processor idle time which is attributable to memory access latency. This will then result in a p processor machine which will run programs $O(p)$ times faster than the same program would run on an equivalent one processor machine. In the case of fat rings, an optimal simulation would have $p = n^{1/2}$. Though the basic work done by a fat mesh processor in a cycle is similar to that of a traditional BDN, we do need some additional functionality to be able to support the larger quantity of network traffic.

The basic attributes of any BDN node are that the processor may inject up to one memory request into the network each cycle, and the memory module may service up to one request per cycle. The fat ring links will have a capacity of $l = n^{1/2}$ requests per cycle, and we must not allow contention in the network to threaten the pipelined nature of the simulation if we are to hide memory latency.

Injecting memory requests into the network is relatively easy in our case where link width l is equal to the number of threads per processor s. The goal is to reduce the chance of contention, i.e. if a processor is injecting a request on wire $1 \leq i \leq p$, then it is important to ensure that no other request will be coming in on wire i. We do this by simple round robin scheduling. At the beginning of a PRAM step simulation, each processor injects its first request onto wire 1 of its outgoing link. At subsequent cycle i each processor injects its i-th request onto wire i, continuing until after p cycles it will have injected all its outstanding requests for that PRAM step. At any given cycle we can guarantee that the link about to be used is free, as it has never been used by any processor yet.

4.4.1 Selection and Forwarding

One important task for each fat ring processor is that of selection and forwarding. Selection is the job of choosing those requests which are arriving on the incoming link of the node and are destined for the local processor, and directing them to the memory module queue. Forwarding consists of determining which arriving requests are not destined for the local processor and directing them to the outgoing link of the node. Additionally, the processor may need to inject another request into the network at the same time. Most importantly, both these operations need to be pipelinable; any operation that will potentially take place during every cycle needs to be constant-time, and those that take place a constant number of times needs to be at most $O(p)$ time.

We depend on the fact that the memory management scheme described earlier is in use. This ensures that with high probability no more than $4p$ requests will be destined for any one node during the simulation of a PRAM step. However, we have proved nothing about the distribution of these arrivals; they may arrive one per cycle for $4p$ cycles, or all $4p$ may arrive in one cycle. It is clear that with the injection scheme above, after p cycles each processor will have injected up to p requests into the fat ring. Since each of these requests may travel up to p links before reaching its destination, there may be times at which virtually the entire link is active, i.e. up to p requests may arrive on the incoming link of any node in one cycle of the simulation. To maintain pipelinability it is clear that efficient selection will require a degree of on-chip parallelism. A block diagram of the relevant components of the node architecture are shown in figure 4–3.

The first task is to determine what incoming requests are destined for the local memory module. This will require a separate circuit for each incoming wire which is of constant depth, which we call a *matching* circuit. This circuit will compare the destination address of incoming requests, A, with the IDs of the local threads on that node, $\{p_i, ..., p_j\}$. We assume that the threads which are resident

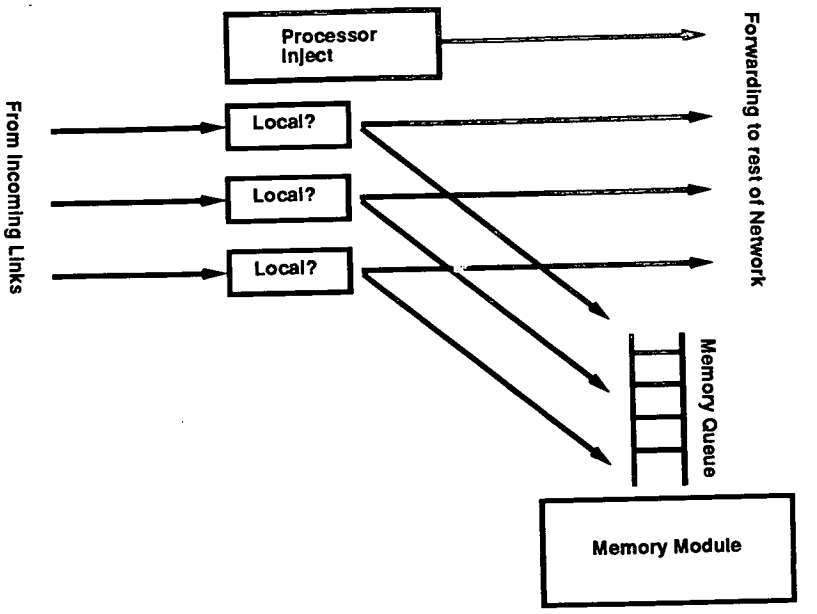


Figure 4–3: Node architecture for selection and forwarding.

on a processor are numbered contiguously. Hence the task is to determine if the address falls between two address limits which are determined based on the number of PRAM processors n we are simulating, as well as on the number of physical processors in our machine p. For each incoming request A on processor i we are trying to determine the boolean $Local_i$, (where $Local_i = (lower_i \leq A \leq upper_i)$, where $lower_i$ and $upper_i$ are determined simply:

$$lower_i = \lfloor p/n \rfloor i$$

$$upper_i = (\lfloor p/n \rfloor + 1)i$$

The matching circuit is composed of simple subtraction circuits as shown in figure 4–4, which are of constant depth. Each request will pass through two such circuits, at which time it will be determined if the request is destined locally or remotely. Such circuits are crucial in any model of a parallel node, though they are often not described. In particular, the hypercube model assumed by Valiant in the BSP work described earlier, will also need a constant time matching circuit

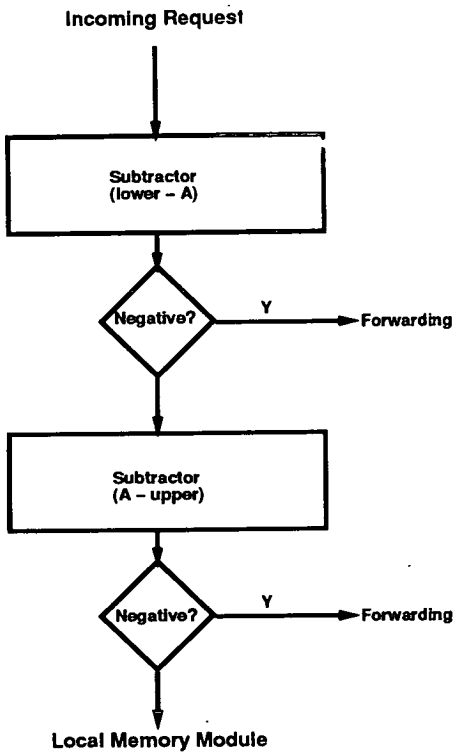


Figure 4–4: Fat Ring Node Matching Circuit.

on each of the $\log n$ incoming links [Valiant 1990a]. Additionally, if we were to compare the hardware of our p processor fat ring with that of a n processor normal ring, we would observe that both require n matching circuits of similar complexity.

After passing through the matching circuit all requests which have been determined to be destined for the local processor are placed in a memory module queue. We now show that, though up to p requests may arrive in any given cycle, the number of requests that will be destined for the local processor will be, with a high probability, small.

Theorem 9 *Given any set of p EREW requests which is arriving at a node in a given cycle, and with $m \geq n^2$. With high probability there will be no more than $O(\log p)$ destined for the local processor from this set.*

Proof: Again we use the lemma 2 in a similar way, but now we use the parameter $v = p$ and choosing $k = 4 \log p$. For simplicity we neglect the term $e^{kp/m}$, as it very close to one for any reasonable value of p, as we explained in the previous hashing proof.

Therefore we have:

$$\begin{aligned} Prob(R > 4 \log p) &< \left(\frac{pe}{4 \log p}\right)^{4 \log p} p^{1-4 \log p} \\ &= \left(\frac{pe}{4p \log p}\right)^{4 \log p} p \\ &= \left(\frac{e}{4 \log p}\right)^{4 \log p} p \end{aligned}$$

We then separate terms, and change the sign of the exponential of the e term:

$$\begin{aligned} Prob(R > 4 \log p) &< \left(\frac{e}{4}\right)^{4 \log p} \left(\frac{1}{\log p}\right)^{4 \log p} p \\ &= \left(\frac{4}{e}\right)^{-4 \log p} \left(\frac{1}{\log p}\right)^{4 \log p} p \end{aligned}$$

We know that the term $\left(\frac{1}{\log p}\right)^{4 \log p}$ will always be substantially less than 1, so we will now neglect this term. We use the relationship $a^{\log b} = b^{\log a}$ to obtain:

$$Prob(R > 4 \log p) < p^{-4 \log 4/e} p$$

and as $-4 \log \frac{4}{e} < -2$ we may write:

$$Prob(R > 4 \log p) < o(p^{-1})$$

□

We do not provide details as to how a non-constant number of requests may be entered into the memory module queue in constant time, but we point out that just such an assumption is also fundamental to the efficient hypercube simulations presented in both [Upfal 1984b] and [Valiant 1990b]. We will also provide experimental evidence in later chapters to support the claim that both the number

of arrivals during any given cycle and the maximum memory module queue size during simulation of a PRAM step are both small constants for a given machine size (see chapter 7).

After queuing the first requests, the memory module will then begin to dequeue and service one request per cycle from the queue. Any elements which are not destined for the local processor will be forwarded to the outgoing links of the node. At the same time, if all p outstanding requests from the local processor have not been injected into the network, then one more request will be placed on the outgoing link as previously described (request i placed on wire i). The simplicity of the ring allows us to prove routing results deterministically.

Theorem 10 *Any set of $n = p^2$ EREW memory requests which are initially distributed p per processor may be routed to their destinations on a p processor fat ring in $O(p)$ time.*

Proof: Each processor is the source of at most p requests, and can issue one per cycle, so will have injected all p requests into the network after p cycles. No link will have more than p requests injected into it, and each link has p wires, so there will be no contention in the network. The farthest any request will need to travel is to the counter-clockwise neighbour of its source, assuming messages travel only clockwise, and this is $p - 1$ nodes away. Upon arrival at a node, a request will simply pass through the matching circuit we have described, and then be placed on the outgoing link by the next cycle if it is still in need of forwarding. This will require $O(1)$ time, as described. Therefore, in the worst case, a request will require p cycles to get injected into the network, and then travel $p - 1$ nodes, each of which requires $O(1)$ time to traverse, and so the entire routing process will require $O(p)$ time. □

Given the above routing theorem we may now prove our final result for the fat ring.

Theorem 11 *With high probability an n processor EREW PRAM may be simulated on a $p = n^{1/2}$ processor fat ring with optimal efficiency and slowdown of $O(p)$.*

Proof: PRAM instructions will be either local operations, which by definition will execute in constant time, or communications events. Without loss of generality we assume all n PRAM instructions are reads, as they are the most time consuming. Theorem 10 has shown us that all n requests may be routed to their destinations in $O(p)$ time. Theorem 8 ensures that no more than p requests will arrive at any one node with high probability. However, we do not know if the p requests arriving at a node will arrive all in the first cycle, all in the pth cycle, or be evenly distributed.

Assume the worst case, that all p requests arriving at a node arrive on the last cycle of the routing time, $O(p)$ cycles after the beginning of routing. Now the memory module removing them from their queue and servicing them one per cycle will begin servicing them. After p time they will all be serviced. At this point we have the same routing problem in reverse, in order to get the requested data back to the reading processors. This route will also take $O(p)$ time. So the three stages of our simulation of read operations are: route, access memory, and route again. The times of the three phases in the worst case will be $O(p)$, p, and $O(p)$ respectively, or a total of $O(p)$ time. □

Chapter 5

Fat Meshes

We now generalize the results of the previous chapter to refer to a class of interconnection networks which we refer to as Fat Meshes. Again we are primarily interested in simulation of an EREW PRAM, though we consider concurrent access in a following chapter. Note that the fat ring of the previous chapter does not strictly fall within the class of fat meshes defined here, due to the fact that our fat meshes do not have toroidal connections. This reflects the fact that in practice rings appear to be a more common network than linear arrays, whereas toroidal meshes more difficult to build and hence less common than their non-toroidal counterparts.

5.1 The Mesh Model

We are interested in both the two dimensional mesh and in meshes of any arbitrary higher dimension, r. An r-dimensional mesh is a set of nodes, P, and links, L. A node ID for an r-dimensional mesh is an r-tuple, i.e. $p_i = w_1, w_2, ..., w_r$. Any two nodes share a link if their IDs are the same except for one element of the r-tuple and if that one element only differs by one. E.g. if the ID of node i is $p_i = w_1, w_2 ... w_r$, and of node j is $p_j = w'_1, w'_2 ... w'_r$, then $p_i, p_j \in L$ if

$w_1 ... w_{k-1}, w_{k+1} ... w_r = w'_1 ... w'_{k-1}, w'_{k+1} ... w'_r$ and $w_k = w'_k \pm 1$ for some $1 \leq k \leq r$. Nodes that share a link are also referred to as near neighbours. For simplicity we will assume that the extent of each dimension is the same for all the meshes we consider, i.e. our meshes are *square*, rather than rectangular. Therefore for an r-dimensional mesh the extent of all dimensions is $P^{1/r}$ and the diameter of the mesh is $d = r(P^{1/r} - 1)$, which for simplicity we approximate as $d \approx r \cdot P^{1/r}$. Hence the diameter can be quite small compared to the ring, but is still reasonably large compared to hypercubic networks for small values of r. Again, the meshes we consider do not have toroidal connections, but the basic nature of our results would be largely unchanged if they were included.

Recall from chapter 3 that for an optimally efficient simulation to be possible we need a multithreaded r dimensional mesh with $p = n^{r/r+1}$ processors, with each processor having $s = p^{1/r}$ threads. With a two dimensional mesh this results in a $n^{1/3} \times n^{1/3}$processor mesh[1], with $n^{1/3}$ threads per processor. Recall also that each link will need at least $\Omega(p^{1/r})$ wires to provide the necessary bandwidth, e.g. a two dimensional mesh will have link width $l = p^{1/2}$. We assume that the node architecture is equivalent to that of the fat ring described earlier, i.e. each wire of a link will have its own selection and forwarding mechanism, but the processor can only issue and service one request per cycle. A schematic of a 9 processor two dimensional fat mesh may be seen in figure 5–1, where each node is shown to have a memory module and set of threads from which instructions are fetched. We now describe the unspecified components of the simulation, memory management and routing.

---

[1]Throughout this work we will assume the total processor count in any mesh will be p, e.g. $\sqrt{p} \times \sqrt{p}$ processors in the case of a two-dimensional mesh. Some authors choose instead to describe a two-dimensional mesh as an $p \times p$ processor array, which makes comparison with other networks of the same number of processors more difficult.

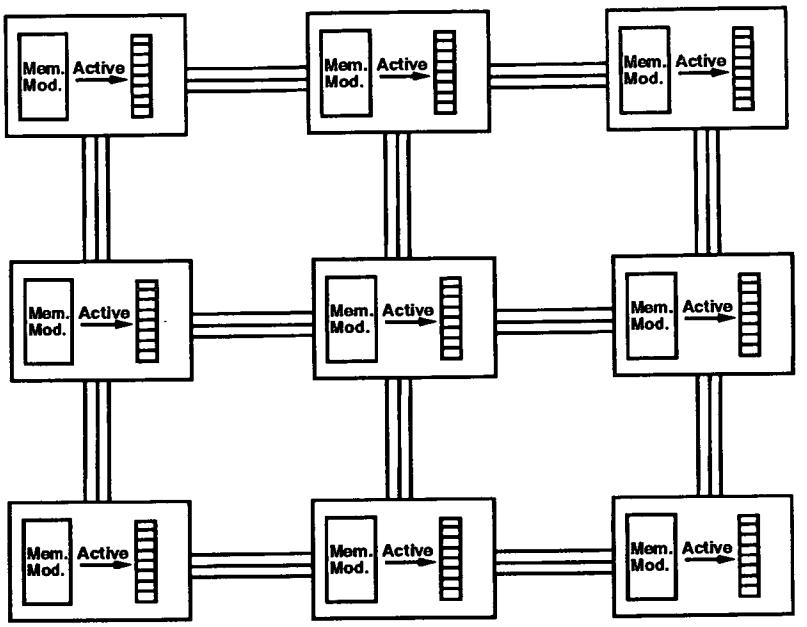


Figure 5–1: A nine processor fat mesh with multithreading nodes.

5.2 Memory Management

For meshes of all dimensions we will again use the hash function H_*, though the specific requirements for the scheme change as a function of the diameter of the network. E.g. for a two dimensional mesh, we will issue $n = p^{3/2}$ requests in order to simulate each PRAM step, and we need to ensure that no more than $p^{1/2}$ requests arrive at any one node during our simulation. In general we will have n requests for each PRAM step we are trying to simulate, and on a r-dimensional mesh of $p = n^{r/r+1}$ no more than $p^{1/r}$ requests may arrive at any one processor during a step.

Theorem 12 *Consider any set of n EREW memory requests which address any of the $m \geq n^2$ memory locations which are distributed amongst the memory modules of an r-dimensional mesh containing $n^{r/r+1}$ processors. If the memory is hashed*

with a function h selected from the class H_*, then no more than $O(n^{1/r})$ requests will be directed towards any one node with high probability.

Proof: The proof depends upon the lemma 2, and is a generalization of the proof for Theorem 8, and hence we use similar techniques here. In the general case of any r-dimensional mesh, the number of processors is p, the number of requests needing to be serviced for a given PRAM step is $n = p^{r+1/r}$, and the maximum number of requests that may arrive at any one node is $k = s = O(p^{1/r})$. We choose to use $k = 4p^{1/r}$ to simplify our calculations.

Substituting into the lemma as before we get:

$$Prob(R > k) \leq \left(\frac{ve}{kp}\right)^{k} pe^{kp/m}$$

$$= \left(\frac{p^{(r+1)/r}e}{4p^{(r+1)/r}}\right)^{4p^{1/r}} pe^{\left(\frac{4p^{(r+1)/r}}{p^{(\frac{r+1}{r})^2}}\right)}$$

$$= \left(\frac{e}{4}\right)^{4p^{1/r}} pe^{\left(\frac{4}{p^{\frac{r+1}{r}}}\right)}$$

Similarly to our earlier proof, we know that for interesting values of p the term $e^{\frac{4}{p^{\frac{r+1}{r}}}} \approx 1$, and hence we now neglect it. We also deal with the lone p outside the exponential as in previous proofs, i.e. if we bring it inside the exponential, then we obtain:

$$= \left(\frac{e}{4}p^{\frac{1}{(4p^{1/r})}}\right)^{4p^{1/r}}$$

where the term $p^{\frac{1}{4p^{1/r}}}$ may be bounded from above by a small constant. Therefore the entire expression reduces to the form:

$$Prob(R > 4p^{1/r}) \leq \alpha^{4p^{1/r}}$$

where $0 < \alpha < 1$ for reasonable values of p. Recall also that in practice r will be a small integer. Therefore the probability of a failure of H_* for a fat mesh is very small, and is similar to the case for the fat ring presented in Theorem 8. □

5.3 Routing

When routing on a fat mesh we maintain the invariant that a request injected into the network on step i will always stay on wire i. I.e. requests will not migrate between wires as they move through the network, but will stay on the wire upon which they were first injected. This allows us to treat the fat mesh as a set of s separate meshes, each of which is similar to a traditional mesh network. The main exception to this similarity is the requirement that only one request will be issued by any one processor in a given cycle, so the s requests issued by a processor on its s links will each be injected at separate times.

We now show that worst case bounds on routing in a fat mesh are quite bad for simple greedy routing schemes. However, if we consider routing in the context of the memory management scheme we have already suggested, then routing becomes quite simple. We initially focus on the problem of permutation routing, where each of the n PRAM threads will send and receive exactly one request. Equivalently, permutation routing in the context of a multithreaded fat mesh may be defined as s requests being sent and received by each of the p processors. We focus on permutation routing as an interesting special case; if we are unable to guarantee good performance for permutations then it is unlikely other routing patterns will perform well. Greedy routing of arbitrary patterns will clearly have daunting worst case bounds, particularly in the case that all requests are destined for one processor, which will require $\Omega(n)$ time.

5.3.1 Greedy Routing

Greedy routing is simply any scheme that routes requests along their shortest path through the network, without any particular behaviour to avoid contention. On a two-dimensional mesh, an algorithm that first routes all requests to the

correct column, and then routes them to the correct row within that column, is a greedy algorithm. More generally, in an r-dimensional mesh, requests will be routed along each of the r dimensions, one by one, until finally arriving at the destination. We assume that requests in the machine will only travel within one dimension at a time, i.e. we assume a degree of synchronization takes place between routing in some dimension $i \in 1...r$ and routing in dimension $i+1$, such that requests traveling in different dimensions will not interact. For simplicity we assume that requests are stored into memory after reaching their destination node within each of the r routing stages. In practice, performance benefits might be available by providing some fast buffering of messages instead, but our arguments below suggest that memory storage time will rarely be a bottleneck due to the even distributions of request arrivals. Since the farthest any requests can travel is the diameter of the network, greedy routing will always complete in $O(diameter)$ time if there is no contention. However, worst case contention can result in routing that is substantially slower than this optimal bound.

We now focus on the case of r dimensional greedy routing, where we first route all requests to the correct column[2] in dimension 1, then within that column to the correct column dimension 2, etc. Our primary concern is with quantifying node contention, i.e. how many requests may need to arrive or depart from a given fat mesh node during routing. However, we must first show that link contention is not a serious problem for a greedy routing scheme on a fat mesh.

Lemma 4 *Greedy routing for any set of $p^{1/r}$ requests within any column of $p^{1/r}$ processors which are aligned along one dimension of an r dimensional fat mesh, will not result in link contention.*

---

[2]Here, as elsewhere, we use the term column to signify the more general idea of a set of $p^{1/r}$ processors which are aligned in any one of the r dimensions of a r dimensional fat mesh.

Proof: To prove this we need to show that no one link will ever have more than $p^{1/r}$ requests passing through it in a given cycle of our r dimensional fat mesh simulation, since each link has no more than $O(p^{1/r})$ wires. Given that there are only $p^{1/r}$ processors in any fat mesh column, for there to be more than that number of requests arriving at a particular link, there will need to be more than one request from at least one of the processors arriving at that node. However, recall our simple greedy routing scheme, coupled with our multithreading scheme, disallows this. A processor can only inject one request per cycle into the network, and each request will travel exactly one link per cycle until it arrives at its destination within that column, at which point it will leave the network links and be written to memory before beginning the next routing stage. Therefore, the requests from a particular processor will never meet up again on any link once they leave their source. Therefore there will never be more than $p^{1/r}$ requests needing to cross any one link during a given cycle. □

Node contention, though, is not as easy to deal with. In a given column of a two-dimensional fat mesh there are $p^{1/2}$ processors, each of which will be the destination for $O(p^{1/2})$ requests. So a total of $O(p)$ requests in the machine will be destined for a given row. If in the worst case all these requests originated in a particular column (each column will also have p requests sent from it), then in the first step of the algorithm they will all be sent to the same processor. Therefore it will require p steps to receive all the requests, and p steps to send them out to their destinations within that row, as our fat mesh nodes can only write one request to memory per cycle, or inject one request per cycle into the network. So worst case routing within a two-dimensional fat mesh is a $\Omega(p)$ time operation, instead of the optimal $O(p^{1/2})$ we would like. We generalize this result to r-dimensions in the following theorem, where once again $p = n^{r/r+1}$:

Theorem 13 *The greedy routing of any set of n requests routed in a r-dimensional*

fat mesh such that any processor will be both the source and destination of $p^{1/r}$ *requests will require* $\Omega(p^{r+2/2r})$ *cycles in the worst case.*

Proof: Worst case node contention takes place at the stage of the algorithm when the largest number of requests needs to pass through a given node. We therefore consider the maximum number of source and destination nodes which may route through a given node during a particular stage. Clearly the number of requests which may be routed through a node in a given stage is no more than either the number of possible sources for requests from all previous stages which would potentially need to route through that node, or the number of all future destinations for the requests in subsequent routing stages. Recall too, that each node can be the source and destination for up to $p^{1/r}$ requests.

If we consider the first of the r routing stages, only one column of $p^{1/r}$ processors may be a source, and each source can send a maximum of $p^{1/r}$ requests, making a total of $p^{2/r}$ possible requests being sent. However, since these messages may be destined for any node in the machine, the number of eventually possible destinations is large, namely $p^{r-1/r}$ nodes, or a total of p requests received (as each node may receive up to $p^{1/r}$ requests. The maximum number of requests being routed through a node in stage 1 is the minimum of those two quantities, or $p^{2/r}$ requests. Conversely in the last (r-th) stage, anywhere up to $p^{r-1/r}$ processors may be valid sources for requests passing through a column, or a total of p requests. However, we know there are only up to $p^{2/p}$ possible destinations in a column on the last stage, so the most requests arriving at any one node in the last stage of routing is also $p^{2/r}$.

On the i-th routing stage, there are $p^{i+1/r}$ possible source requests, and $p^{r-i+1/r}$ possible request receipts. Given that the number of sources is an increasing function of i, and the number of destinations is a decreasing function of i, we know the maximum contention takes place when the number of sources and destinations is equal (an example of these functions for the case of $p = 64$ and $r = 2$ is shown in

figure 5–2). This happens in stage $p/2$ of the r stages. At this point there are a total of $p^{r+2/2r}$ possible sources and possible destinations for requests. Hence the worst case time for any one routing stage is $\Omega(p^{r+2/2r})$. Note that the increased bandwidth of our fat mesh links does not alleviate this problem, as the nodes still can only inject a maximum of one request per cycle into the network once it has received them. □

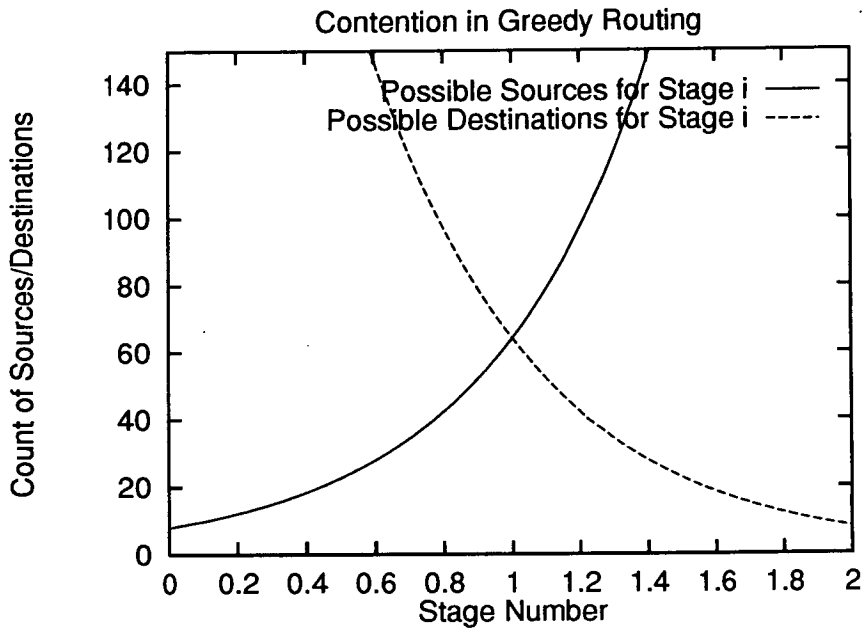


Figure 5–2: Example of source and destination count functions.

5.3.2 Routing and Memory Management

Given the worst case bounds above, a traditional option might be to introduce randomness to the routing algorithm, such as the two-phase random routing we've discussed earlier [Valiant and Brebner 1981]. However, an alternative approach is to exploit the pseudo-randomness we have already instilled in another phase of our simulation through the use of randomized memory management. We now consider the potential of greedy routing to provide optimal upper bounds once we

assume that all memory has previously been hashed. Though our earlier consideration focused on permutations within the multithreaded context, we now consider routing any set of n EREW requests. The addition of memory management into our routing strategy ensures that with a high probability requests will be reasonably well distributed, and hence we need not restrict our routing patterns to permutations.

As in other sections of this chapter, we will first consider the two-dimensional case, and then consider the more general r-dimensional case. Again, in the two-dimensional greedy algorithm, when first routing all requests to the correct column within their initial rows, there are up to p requests potentially converging on one node. However, if we remember that the destination addresses of the memory requests are all hashed in accordance to H_*, then we can assess the likelihood that these request are destined for the same column. More specifically, we need to ensure that, given any set of p requests which are located in a common row, the chances are small that more than $p^{1/2}$ of them are destined for the same column. Given this result, in addition to the bounds on the number of requests which will finally arrive at each node provided earlier, we can ensure that our routing will not be hampered by contention, and will therefore complete in $O(p^{1/2})$ time.

Rather than provide a proof for the special case of $r = 2$, we now show the corresponding generalized result for the r dimensional fat mesh. We are now considering any of the r stages of greedy routing, where each stage consists of moving all packets along the rth dimension until the destination address matches the location along that dimension. This corresponds to a group of $p^{1/r}$ processors, each with $p^{1/r}$ requests, routing amongst themselves. Again we assume synchronization in our greedy routing, such that routing for stage $i+1$ does not begin before stage i is completed. If no more than $O(p^{1/r})$ requests will be routed to any one node with high probability, then routing will be contention free and take $O(p^{1/r})$ time.

Lemma 5 *In greedy routing on a r-dimensional mesh, any of the r stages consist*

of routing $p^{2/r}$ requests to any one of $p^{1/r}$ nodes. For each stage, there will be no more than $O(p^{1/r})$ requests routed through any one node with high probability.

Proof:(Sketch) Again we use the lemma. However, in the lemma it is assume that there are p nodes, where in our case there are $p^{1/r}$, so we must scale our input parameters by this $p^{1/r}$ term. Now $v = p^2$ and $k = 4p$. The lemma then looks like:

$$\begin{aligned} Prob(R > k) &< \left(\frac{ve}{kp}\right)^k pe^{kp/m} \\ &= \left(\frac{p^2 e}{4p^2}\right)^{4p} p \\ &= \left(\frac{e}{4}\right)^{4p} p \end{aligned}$$

Which again results in a probability of:

$$Prob(R > 4p) < \alpha^{4p}$$

for some $0 < \alpha < 1$. For more details of a similar proof see Theorem 8. □

Given this result, we are able to prove the overall routing result we desire.

Theorem 14 *Any set of $n = p^{r+1/r}$ EREW memory requests which are initially distributed $p^{1/r}$ per processor and which are hashed with H_* may be routed by a r-dimensional p-processor fat mesh in $O(p^{1/r})$ time with high probability.*

Proof:(Sketch) Follows directly from the above lemma. □

Given this ability to route in diameter time for any degree fat-mesh, we may now easily prove the main result from this chapter; that an EREW PRAM may be simulated in a fat mesh efficiently and with $O(diameter)$ delay.

Theorem 15 *An n processor EREW PRAM may be simulated on a r-dimensional fat mesh (for any constant r) with $p = n^{r/r+1}$ processors with optimal efficiency and slowdown of $O(p^{1/r})$ with high probability.*

Proof: From Theorem 15 above we know that we may route any set of n EREW memory requests in $O(p^{1/r})$ time with high probability. Additionally, from Theorem 12 we know that with high probability there will be no more than $O(p^{1/r})$ requests destined for any one processor. Therefore this theorem follows easily. □

Chapter 6

Concurrent Access

The goal of this thesis so far has been to provide an efficient EREW PRAM simulation on fat meshes and rings. We now address the question of support for concurrent access on such machines. The results serve to underline the point that the CRCW model is not well suited to optimally efficient simulations.

As we have mentioned in introductory chapters, the standard technique for supporting CRCW simulations is to first sort all requests, and then begin eliminating all requests with common destinations. We therefore begin by considering sorting on fat meshes. We provide lower bounds to the effect that general sorting can not be done in diameter time, on this or any other multithreaded machine. We then provide a sort which achieves the lower bound, and provide more detail on how the sort is used to allow concurrent access.

6.1 Lower Bounds

Though sorting can take place in $O(diameter)$ time on any r-dimensional mesh without multithreading [Kunde 1987], the multithreaded fat mesh will not be able to achieve such bounds. We are concerned here with the general problem of

sorting, rather than any special case such as integer sorting or sorting when the range of the keys is known beforehand.

Theorem 16 *Sorting any set of n items on an r dimensional fat mesh with $p = n^{r/r+1}$ processors requires $\Omega(p^{1/r} \log p)$ time.*

Proof: The general sorting of any set of n items will require $\Omega(n \log n)$ comparisons. A fat mesh will have $p = n^{r/r+1}$ processors, so sorting will require

$$\Omega(\frac{n \log n}{n^{r/r+1}})$$

time. Recalling that $n = p^{r+1/r}$ we may simplify this as:

$$\begin{aligned} \frac{n \log n}{n^{r/r+1}} &= n^{1/r+1} \log n \\ &= \left(p^{r+1/r}\right)^{1/r+1} \log n \\ &= p^{1/r} \log n \end{aligned}$$

Given that $\log n = \frac{r+1}{r} \log p$ and r is a small constant, we may rewrite this as:

$$\Omega(p^{1/r} \log n) = \Omega(p^{1/r} \log p)$$

□

The basic premise of a multithreaded simulation targeted at a realistic network is to hide latency which is $O(diameter)$, by running $O(diameter)$ threads on each physical processor. Therefore we provide a more general result which applies to any Bounded Degree Network.

Theorem 17 *Sorting n items on any Bounded Degree Network with $p = O(n/diameter)$ processors will require $\Omega(\log^2 p)$ time.*

Proof: This follows from the fact that any p processor Bounded Degree Network has diameter $\Omega(\log p)$, and that sorting requires $\Omega(n \log n)$ comparisons. Therefore sorting will require

$$\Omega(\frac{n \log n}{n/logn}) = \Omega(\log^2 n)$$

time. Given that r is a constant, and $n = p^{r+1/r}$, we may rewrite this as:

$$\Omega(\log^2 p)$$

□

One approach to avoiding the problem of large sorting times is to use a special purpose network that does combining to eliminate concurrent requests as they are being routed. This type of architecture is considered in [Ranade 1991]. However, for CRCW simulations which run on networks without the additional hardware expenses of combining, such as a Bounded Degree Network, we provide the following additional lower bound.

Theorem 18 *Any CRCW Simulation based on a Bounded Degree Network which depends on general sorting to support concurrent access will not be optimally efficient.*

Proof: (Sketch) Recall that by definition any optimally efficient simulation will have delay on the order of the diameter of the network. Any such multithreaded machine can perform $O(n)$ operations in diameter time, on $p = \frac{n}{diameter}$ processors. Since general sorting requires $O(n \log n)$ comparisons, and this is greater then the $O(n)$ that can be done in diameter time, it will clearly require greater than diameter time to sort, and hence to support CRCW. Therefore no such CRCW simulation will be optimally efficient. □

The above results assume that a general sort will be used as part of the pre processing phase for concurrent access. However, in theory one may avoid these bounds by providing a faster sort which takes advantage of the fact that we know the range of requests will be between $0...m$, and that they are all integers. To allow an optimally efficient CRCW simulation on a fat mesh this special purpose sort runs in $O(n)$ time. We know of no such sorting algorithm which is practical, and we consider the likelyhood of a practical solution to the problem within the

$O(n)$ time bounds as low. This open problem is acknowledged in [Kruskal et al. 1990].

6.2 Cole's Merge Sort

Now that we have provided an EREW simulation on any r dimensional fat mesh we may run any EREW sorting algorithm on top of that simulation. In this way one sorting algorithm can be run on any of the fat mesh networks.

Sorting on parallel computers is one of the best studied problems in computer science. However, it wasn't until the early 1980's that a optimal depth sorting circuit was discovered, that of [Ajtai et al. 1983], which sorts n numbers in $O(\log n)$ time. However, as we have alluded to earlier, the solution is not particularly practical, primarily due to its dependence on expander graphs, which cause it to have very large constants despite its optimal asymptotic performance terms. Sorting techniques for some other computational models are discussed in [Harris 1992, Chin and McColl 1994].

Cole described the first known $O(\log n)$ time algorithm for sorting on a CREW and EREW PRAM [Cole 1988]. Cole's solution is not a sorting circuit, but is particularly appropriate for our purposes, where we would like a sorting algorithm which is portable to all our fat mesh variants. We hereby describe the general techniques used in Cole's algorithm. For a complete proof of the algorithm time complexity we refer the reader to [Cole 1988].

As in any merge sort, we begin by considering the PRAM processors as leaves of a $\log n$ depth binary tree, and with each processor initially holding one element of the array to be sorted. At each of the $\log n$ steps two sorted sublists are merged into a larger list, with each merger corresponding to a level of the binary tree. The mergers are done by constructing a selected sample of each list, and using comparisons among the samples to determine where to insert elements of the two

sublists into the resulting larger list such that the larger list is sorted. The merging procedure at each level of the tree takes $\log n$ time, and as there are $\log n$ levels, the trivial solution requires $O(\log^2 n)$ time. However, the primary observation that allows a $O(\log n)$ time solution is that the merges which take place at different levels of the tree may be pipelined. In particular, by beginning with the sample lists for level i, the sample list for level $i+1$ may be constructed in constant time.

Theorem 19 *[Cole 1988] A list of n items may be sorted on an EREW PRAM in $O(\log n)$ time.*

Combining this result with the EREW simulation we have earlier shown for the fat mesh and fat ring networks, we obtain the following corollary.

Corollary 1 *Sorting on a fat mesh may be done through use of Cole's merge sort in $\Theta(p^{1/r} \log p)$ time.*

Note that this achieves the sorting lower bound we provided earlier in this chapter.

6.3 Eliminating Concurrent Requests

In this section we describe how concurrent requests are eliminated given the ability to sort, and how this allows the support of CRCW access on fat meshes and the fat ring. As we discussed in introductory chapters of this thesis, it is sometimes the case that for reasons of efficiency one must solve the problem of concurrent access at the same time as addressing issues of routing and memory management. However, in this case we are able to provide support for concurrent access with optimal time complexity (but not with optimal efficiency) as an additional phase of processing on top of the EREW simulation with no degradation in performance.

All the following operations are assumed to take place on top of the memory management and routing schemes we discussed earlier. Therefore we are now describing a PRAM algorithm, rather than a fat mesh algorithm.

To process CRCW memory requests we simply add a pre and post processing phase onto the processing for each PRAM step. The role of preprocessing is to combine all duplicate (concurrent) requests for the current PRAM step, such that we may then process the existing EREW requests as normal. We do this by first sorting the set of n requests on their destination memory address by using the merge sort described above. After the sort we have $p^{1/r}$ sorted requests in each processor, such that any duplicate requests resulting from concurrent reads or writes will be consecutively located in the list. Additionally, the lists held by each processor will be globally ordered according to the processor IDs of the PRAM processors. Each processor then steps through the elements of the list it contains locally, eliminating duplicate requests which correspond to concurrent accesses. Concurrent writes are combined as prescribed by the conflict resolution rule of the CRCW model. Concurrent reads are combined by simply eliminating one while at the same time storing book-keeping information at each node as to which concurrent reads have been eliminated. This book-keeping information is then used to "uncombined" or duplicate the results of concurrent reads during the postprocessing phase.

Given that we have now eliminated all duplicate requests local to each processor, we must now combine globally across the set of EREW nodes in our simulation. We do this also by viewing the machine as a binary tree. Each processor will have at most two requests that may be further combined globally; those at the beginning and end of each sorted local list. Then we designate processors with odd IDs as senders and processors with even IDs as receivers. The sender with processor ID i sends its largest request to processor $i+1$, and sends its smallest to processor $i-1$. The receiver receives a large and a small request to be combined with the large and small requests of its local list they have identical destinations.

If they do not share the same destinations, then the new requests are placed at the end or beginning of the list to become the new largest or smallest requests, respectively. The sender from this step is then done with the preprocessing phase and holds only EREW requests. The receiver will continue to iterate in a similar way, but now communicating with a processor who's ID differs by 2, rather than 1. In the phase following that all senders communicate with processors who's IDs differ by 4, and then 8, etc. In total the preprocessing phase is a set of $\log n$ such steps, where on step j, processors $\{1 \cdot 2^j, 2 \cdot 2^j, 3 \cdot 2^j, ..., \frac{n}{2^j} \cdot 2^j\}$ are receivers, and $\{1 \cdot 2^j + 1, 2 \cdot 2^j + 1, 3 \cdot 2^j + 1, ..., \frac{n}{2^j} \cdot 2^j - 1\}$ are senders. The number of processors participating in each step is therefore reduced by a factor of two from the previous phase. Once all processors have completed this global operation all duplicate requests will have been eliminated and that any remaining requests will be EREW.

Theorem 20 *A set of n CRCW requests may be reduced to a set of $O(n)$ EREW requests in $O(\log n)$ steps on an EREW PRAM.*

Proof: Sorting requires $\Theta(\log n)$ EREW steps, and the procedure above will ensure that after $O(\log n)$ further steps any duplicate requests are combined, leaving at most n EREW requests. □

From this theorem we easily obtain the following corollary:

Corollary 2 *An n processor CRCW PRAM may be simulated on a r-dimensional fat mesh (for any constant r) with $p = n^{r/r+1}$ processors with a slowdown of $\Theta(p^{1/r} \log p)$.*

Proof: (Sketch) From Theorem 20 we know that reducing a set of n CRCW requests will require $\Omega(\log n)$ steps on an EREW PRAM. Since each EREW step on a r dimensional fat mesh requires $O(p^{1/r})$ time to simulate, a CRCW step will require $\Omega(p^{1/r} \log n)$ time. Given that r is a constant, and due to the sorting lower bounds of Theorem 16, we may write this as $\Theta(p^{1/r} \log p)$. □

Note that, somewhat disappointingly, this does not qualify as a processor efficient simulation, given the ratio of $\log p$ between the slowdown of the simulation and the number of processors used. In particular, we have violated equation 3.1 of our definition of optimal efficiency, as the delay of the simulation is greater than the diameter of the network.

6.4 Efficient Concurrent Access

Though optimal efficiency is impossible for a multithreaded machine, we can obtain a non-optimal processor efficient CRCW simulation if we increase the slackness of the simulation to cover the added overhead of sorting. From Corollary 2 we know that the latency of a CRCW access will be:

$$L(p) = \Omega(p^{1/r} \log p)$$

for any r dimensional fat mesh. Therefore to hide this latency we will need $s = p^{1/r} \log p$, and therefore

$$p = \frac{n^{r/r+1}}{\log p}$$

Naturally we will again need the bandwidth of each link to correspond to this increased slackness, i.e. $l = s = p^{1/r} \log p$. Now we once again consider memory management and routing for these new parameters. Due to its similarity to previous results we keep our explanation brief here.

The memory management scheme must provide the following guarantee; that any set of $n = p^{r+1/r} \log p$ requests will be distributed amongst the p processors such that no more than $O(p^{1/r} \log p)$ arrive at any one processor with high probability. For the lemma we use $v = p^{r+1/r} \log p$ and $k = 4p^{1/r} \log p$, giving:

$$Prob(R > k) < \left(\frac{ve}{kp}\right)^k p$$

$$= \left(\frac{p^{r+1/r} e \log p}{4p^{1/r} p \log p}\right)^{4p^{1/r}\log p} p$$
$$= \left(\frac{e}{4}\right)^{4p^{1/r}\log p} p$$
$$= \alpha^{(4p^{1/r}\log p)}$$

where $0 < \alpha < 1$.

Routing depends on the assumption that addresses have been previously hashed with H_*, as in earlier chapters. We note again that on any of the r stages in routing for a r dimensional fat mesh with our new parameters, with high probability we will not have contention in the network with simple greedy routing. In particular, we now show that among the $p^{2/r}\log^2 p$ requests whose origin is a given row or column of an r dimensional structure, there is a high probability that no more that $p^{1/r}\log p$ will arrive at any one node.

We use the hashing lemma precisely as we have in chapter 4, where we have scaled the variable p of the lemma to correspond to the number of destinations in the routing stage we are concerned with. As the number of sources is the square of the number of destinations, we use $v = p^2$ and $k = p$ for the lemma, and observe:

$$Prob(R > k) < \left(\frac{ve}{kp}\right)^k p$$
$$= \left(\frac{p^2 e}{4p^2}\right)^{4p} p$$
$$= \left(\frac{e}{4}\right)^{4p} p$$
$$= \alpha^{4p}$$

where $0 < \alpha < 1$.

These two smaller results lead to our CRCW simulation result:

Theorem 21 *An n processor CRCW PRAM may be simulated on a r-dimensional fat mesh (for any constant r) with $p = \frac{n^{r+1/r}}{\log n}$ and efficiently with slowdown of $O(p^{1/r}\log p)$ with high probability.*

Proof: (Sketch) This follows clearly from the routing and memory management results above, given that each fat mesh processor now has sufficient slackness to hide the delay of the CRCW simulation. □

Note, however, that to provide this non-optimally efficient simulation we have had to not only increase the number of threads per processor, but also correspondingly increase the number of wires per link.

Chapter 7

Experimental Results

An obvious problem with basing an architecture on theoretical results is that, while the performance bounds may be proven asymptotically, there may still be questions about the practical characteristics of the performance which have not been made clear through the theoretical analysis. In this chapter we will provide experimental results regarding the performance of various aspects of fat meshes and rings, in an attempt to reaffirm the theoretical decisions we made earlier. We have generated these experimental results through simulation of the hardware, in order to substitute measurements in units of machine cycles for the "big-O" notation used earlier. One would expect that complexity bounds on performance would be particularly relevant in machines with very large numbers of processors, i.e. $p > 10^{10}$, but with more realistic numbers of processors the non-asymptotic terms may become more important. Hence we focus our study on smaller machines with hundreds to thousands of processors. Additionally, we would expect such smaller machines to be used more frequently in practice.

The program we have constructed is designed to simulate salient features of our architecture, such as hashing, routing, and memory module servicing. It does this at a reasonably high level, neglecting chip technology and other low level details, but it attempts to implement the algorithms we have described as accurately as possible.

The primary input for the simulator is a trace of addresses which are generated during the execution of a program and then stored in a file. This program is serial, but the set of addresses generated for the trace file are clearly the same addresses that would be accessed by any parallel solution to the problem. The simulator is also a serial program, which simulates each processor in turn. When appropriate the simulator fetches addresses from the trace file, which become the memory request to be issued by the processor which is currently being simulated. Given the global shared memory of the PRAM model, it is unimportant what order the processors are simulated in, or what order the addresses are in, as long as we simulate only one n-thread PRAM step at a time. Similarly, we make no attempt to exploit locality through clever mapping of data to processors, but instead use the full generality of the shared memory model by allowing an arbitrary mapping.

However, because we are simulating an EREW PRAM, we do need to be careful to ensure that the serial programs used in generating the trace do so in an EREW manner, i.e., all the addresses accessed within one n thread PRAM step are unique. This is ensured through careful choice of problem size and looping constructs within the serial programs which generate trace files. The main goal of the simulator is to monitor the passage of time as processors of the system fetch addresses from the trace file, hash those addresses, and then inject those addresses into the network in the form of memory requests. We then monitor the progress of these requests through the network, as well as their service rate at the destination memory modules. One of the main goals of our experimental work is therefore to determine the time in machine cycles required for a given fat mesh or ring to execute one PRAM step. As we are focused on a synchronous model, we do not begin simulation of any PRAM step until the previous one has completed entirely.

We attempt to provide results which are comparable to our previous complexity results, and hence we make powerful assumptions about what takes place in a given cycle of our machine. We acknowledge that a more detailed simulator, e.g. one which provided times for operations down to the gate level and provided

results in units of micro or nano seconds, would also be interesting. However, such results would be dependent on many specific technological assumptions, and therefore would have very limited applicability. Instead we attempt to maintain the generality of our original theoretical results. Examples of more detailed simulations for a different approach to latency hiding in shared memory architectures can be seen in [Harris and Topham 1994a, Harris and Topham 1994b, Harris and Topham 1994c].

7.1 Simulator Architecture

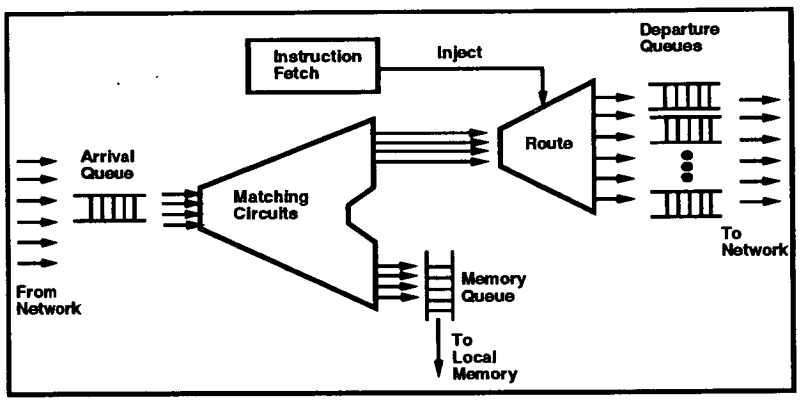


Figure 7–1: Node architecture being simulated.

A schematic of the node architecture being simulated can be seen in figure 7.1. This architecture corresponds to an abstraction of that detailed in the thesis; it is an architecture that is easy to simulate and will have the same performance characteristics as the theoretical architectures we have considered. For example, fat meshes and the fat ring will likely have a small buffer for each wire of a link, and will not need the potentially large queues suggested shown in figure 7.1. However, these queues allow easy monitoring of memory request routing through the network, particularly since our simulator is serial, and will need to step through all outstanding requests one by one. Conversely, the fact that true fat mesh

nodes assume a degree of on-chip parallelism makes a more distributed buffering of requests more logical and efficient in that model. This queuing mechanism also makes it easy to consider performance as a function of the number of wires of each link of a network, which is one of our focuses of the simulation.

The first phase of processing is referred to as "on node routing" , and begins after the set of requests which has arrived from the network is entered into the arrival queue. The requests in this queue are then evaluated to determine if they are local, in which case they are forwarded to the local memory queue. If not, then it is determined upon which outgoing link they will need to be routed, and they are entered into the corresponding departure queue. There will be one departure queue for each link of a fat mesh node. Additionally, if there are still outstanding requests to be injected for that PRAM step, then one is read from the trace file, hashed, and placed in the departure queue which corresponds to the appropriate outgoing link. Again, we are interested in simulation results for an EREW PRAM, so we use trace files which consist solely of EREW requests.

"Off node routing" consists of the communications events which takes place between neighbouring processors. Each processor will move a number of outgoing requests from its departure queues to the arrival queue of its neighbours. The number of requests moved out of each departure queue is equal to the number of wires in each link, l. Any requests which are not moved during off node routing will remain in their departure queues. All queues are FIFO ordered. However, as our routing results made no assumptions about queueing disciplines, we expect this to make little difference in practice.

In earlier chapters we have shown how both on and off node routing requires only a constant number of cycles. Therefore, to facilitate comparison of our simulator results we charge 1 cycle for each such two phase operation. The larger goal of experimental analysis is then to determine how many such cycles are required to implement a single PRAM step on a fat mesh.

7.2 Memory Management

Given a set of memory addresses which correspond to one PRAM step, it is the role of the memory management scheme to ensure that with high probability they will be well distributed amongst the memory modules. Our experimental results emphasize that the use of multithreading has an additional benefit over latency hiding; that hashing functions generally behave better when more than one request is destined for each processor.

Recall the class of hash functions we use, H_*, defined as:

$$H_* = \{h | h(x) = ((\sum_{i=1}^{d} a_i x^i)) mod\ m\}$$

where m is a prime number, d is a constant, and where the a_is are randomly chosen parameters.

We primarily consider two trace files in this chapter. One is from a simple matrix multiply routine, which has extensive spatial locality and strides regularly through memory, and the other is a synthetic address trace of randomly distributed addresses. Matrix multiply provides a trace representative of the class of matrix operations, where successive memory references are often ordered $s * i + o$ for all $0..i..n$, where s is the stride determined by a dimension of the matrix, and o is an initial offset. We will be focusing on using the set of requests from the matrix multiply trace, as it represents an access pattern with inherent locality, and hence a potentially difficult case for a memory management scheme. We also use a trace where each address is random, though generally only to verify that our scheme has done no harm in the sense of regrouping requests which were initially well distributed.

7.2.1 Hashing with Multithreading

Our initial goal is to provide an intuition as to the use of hashing in multithreaded architectures in an effort to ground the theoretical results presented earlier. As stated, we expect very good performance from our hash functions, in the sense that we expect our requests to be very evenly distributed amongst our processors. In particular, we were able to prove that for a set of n requests distributed among $p = n^{r/r+1}$ processors for an r dimensional fat mesh, we can expect with high probability to have $O(p^{1/r})$ arrive at each node. It is significant that our use of multithreading simplifies considerably achieving these demanding goals.

Consider the popular analogy for hashing of memory requests; the throwing of balls into randomly determined buckets. If we have the same number of balls and buckets, then it will be difficult to get one ball to arrive in each bucket by randomly throwing them. However, if we have many more balls than buckets, then it will be relatively easily to throw the balls randomly such that each bucket gets full to roughly the same level. Multithreading, where we have many requests destined for each processor, is clearly analogous to this latter situation. We now analyze experimentally the maximum number of requests destined for a particular node given an arbitrary set of addresses to be hashed. This number is often referred to as the maximum queue size for the processor's memory module.

We begin through use of a particularly simple linear hash function, i.e. we choose an $h(x) \in H_*$ such that $d = 1$ and $h(x) = (a_1 * x + a_0) mod\ m$, where values for a_0 and a_1 are random integers. Firstly we consider the case of no multithreading. We consider 16,384 (or 16K) requests to be distributed amongst 16K processors, and assess the resulting maximum queue size. The results are shown in figure 7–2, where we show the number of requests destined for each processor from a set of requests from the matrix multiply trace. For simplicity we show only the requests destined for the first 100 processors. In this case the optimal is clearly to have one request destined for each processor. We see that for

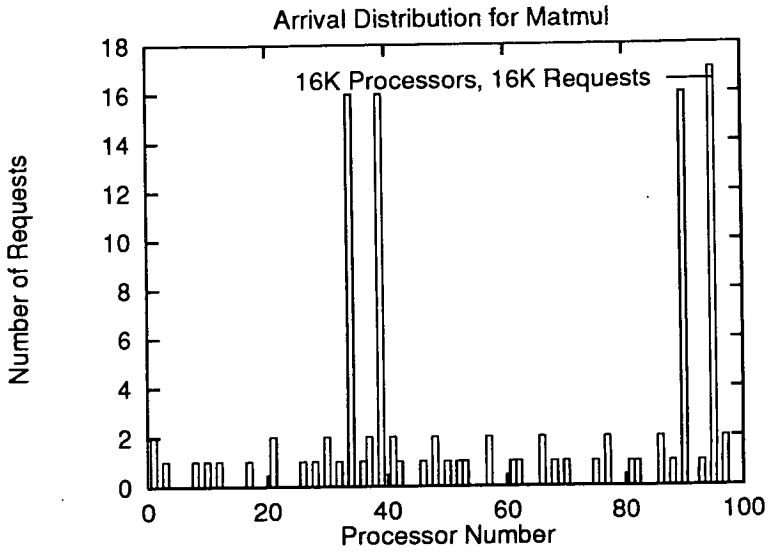


Figure 7–2: Matmul trace with no multithreading.

many processors are only the destinination for one request. In other cases there are zero, and in a few cases, there are substantially more than one, roughly 16 to 18 requests, destined for a particular processor. This is the type of distribution one expects from hashing without multithreading. A well known folk theorem of hashing states that with p requests distributed amongst p processors, the most heavily loaded nodes will receive at most $O(\log n)$requests, with high probability[1]. This corresponds well to our observed results, as $\log 16k = 14$, which is close to the maximum queue sizes shown.

As we introduce multithreading one might fear that we would compound our difficulty, i.e. with s threads per processor, we may have a maximum queue size of $O(s \log n)$. However, as evidence from our earlier theorems suggest, this is not the case. In fact, as we increase s relative to n, we see the heavily loaded threads

[1]We do not offer an explicit proof of this theorem, but the proof is very similar to that provided for theorem 9 in chapter 4. A complete version of the proof can be found in [Mehlhorn and Vishkin 1984].

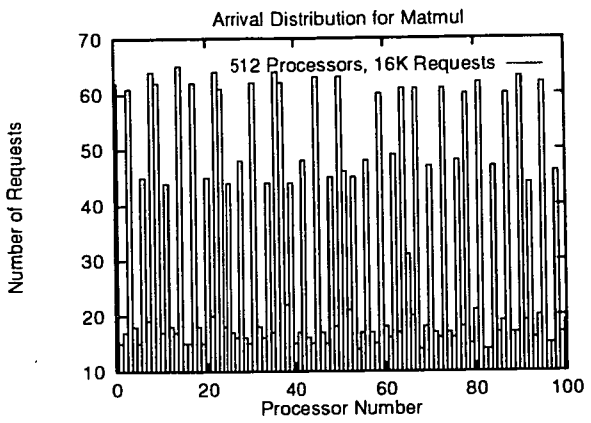


Figure 7–3: Matmul trace with 32 threads per processor.

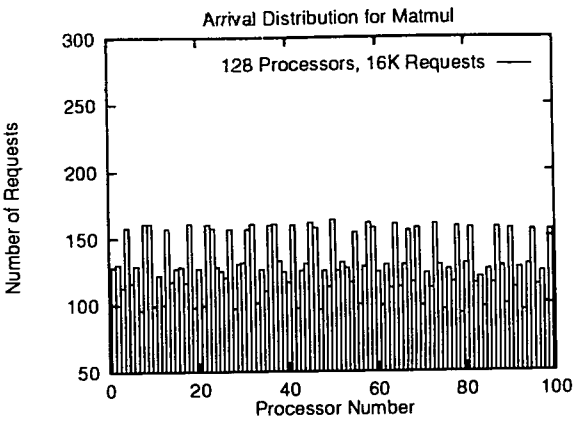


Figure 7–4: Matmul trace with 128 threads per processor.

being well distributed among the p processors, and the resulting distribution being nearer optimal than for the $s = 1$ case that we have just described. This is shown in figures 7–3 and 7–4. In figure 7–3 we again have 16k requests, but now $s = 32$ and $p = 512$. Therefore we would expect that 32 request arrivals per processor would be the average. We note that the more heavily loaded nodes, though there are more of them than in figure 7–2, are only overloaded by about a factor of two over the optimal of 32 requests per processor. In figure 7–4 we have 128 processors with 16k requests and 128 threads per processor, and we see that the maximally loaded nodes have only about 25 percent more requests than the optimal of 128 requests per processor. Hence the hash function behaviour gets better as we increase multithreading. The Y-axis in figure 7–4 has been fixed to allow easy comparison with the figures of the next section.

7.2.2 Hash Function Degree

One concession which is necessary to be able to use hashing in the context of an efficient simulation is to use a hash function of constant degree, rather than the logarithmic degree hash functions that might be used in inefficient simulations. In practice it is best if the constant degree is in fact 1, as it reduces the amount

of computations needed to hash or unhash any address. We now provide experimental evidence suggesting that a degree 1 hash function is a reasonable choice from the perspective of performance.

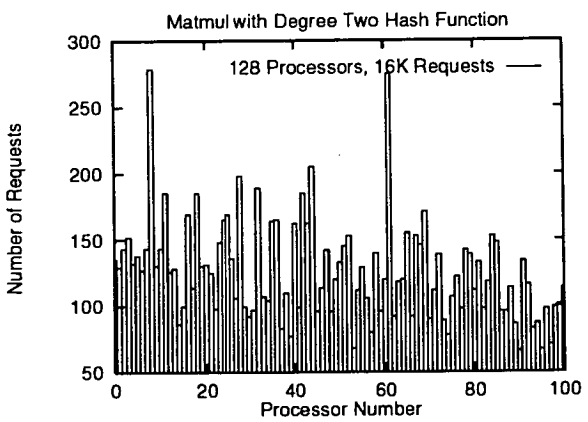


Figure 7–5: Matmul trace with degree two hash function.

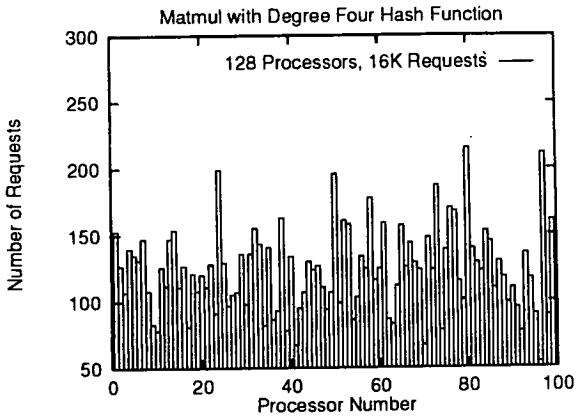


Figure 7–6: Matmul trace with degree four hash function.

In figures 7–5 and 7–6 we see the distribution of requests from the same matrix multiply trace, but using an $h(x)$ with with degree two and four respectively. Comparing to 7–4 we see that the worst case loading on nodes is worse in both the second degree and forth degree hash functions. This is consistent with other experimental results considering hash functions in context of non-multithreaded architectures. In particularly, [Engelman and Keller 1993] found that linear hash functions consistently outperformed their higher degree counterparts. In [Ranade 1991] linear hash functions also prove to be adequate in practical work regarding PRAM simulations.

7.2.3 Random Traces

The usual role played by a hash function is to randomize a set of requests which may initially have a degree of locality. If we consider a trace which consists of randomly generated traces, then we can assume that there is already a good distribution of memory module references. However, hashing random addresses is

useful in its ability to verify that a well distributed trace does not become poorly distributed after hashing.

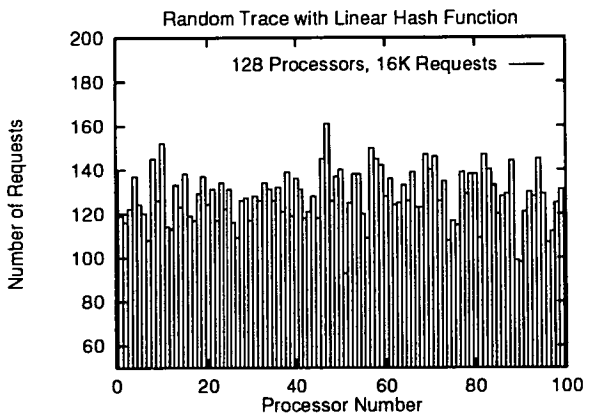


Figure 7–7: Random Trace with Linear Hashing.

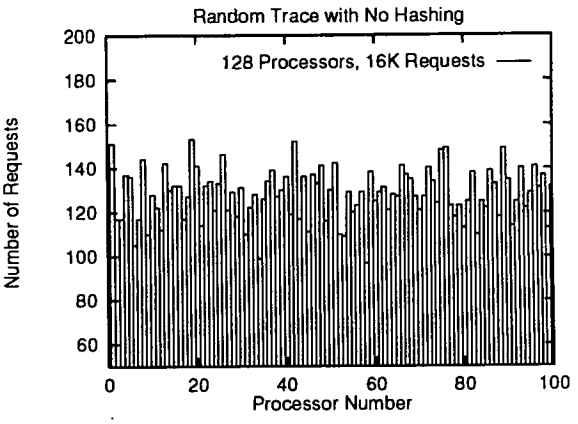


Figure 7–8: Random Trace with No Hashing.

We therefore compare the following two graphs, one where no hashing has been done, and one where the linear hash function described above is used. In the case of no hashing we use $h(x) = x \pmod{p}$ to ensure that the addresses are not out of range of our p processor machine. From observation of figures 7–7 and 7–8 it appears that there is little change in the distribution after the random trace is hashed. This suggests that little randomness is being removed by hashing.

7.3 Routing

Earlier we proved that simple greedy routing in conjunction our randomized memory management scheme allowed us to route within a fat mesh in $O(p^{1/r})$ time. However, these results are also dependent on the bandwidth provided in our fat meshes, namely the fact that the width of each link, l, is equal to the degree of parallel slackness on each node, s. In this section we consider these two results in greater detail, and from a more practical perspective.

We are particularly interested in the behaviour of the queues in each node; the rate at which they empty and fill relative to the width of the links in our fat mesh or fat ring. An example of this behaviour is shown in figures 7–9 and 7–10. It shows the maximum number of requests in the arrival queue and departure queues of our simulated two dimensional fat mesh during the routing process for one PRAM step. Recall that in our simulations we have only one arrival queue, while we have a separate departure queue for each possible near neighbour destination. Therefore the maximum number of requests in the arrival queue at any cycle will be $4s$ on a two dimensional mesh. Departure queues, on the other hand, will begin to accumulate requests if there is contention in the network resulting from the width of the links being too small. Hence their size is quite large in the case where $l < s$.

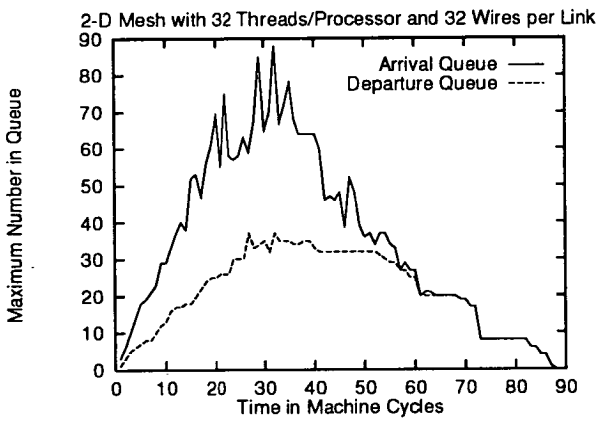


Figure 7–9: Two Dimensional mesh with links of width 32.

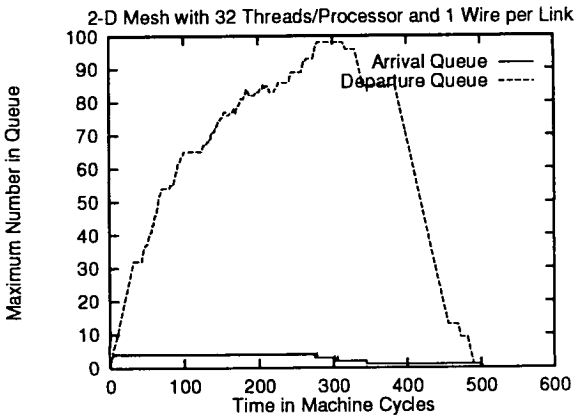


Figure 7–10: Two Dimensional mesh with links of width 1.

Figure 7–9 shows the routing process on a fat mesh with $l = s$. We see that both the arrival queues and departure queues begin filling up in a near linear way initially, as requests are injected into the network, one per cycle, by each processor. In particular, we would expect the maximum departure queue size to rise with a slope close to one, until requests begin to reach their destinations, in which case both slopes will begin to drop off. After s cycles all new requests will have been injected into the network (as we inject one per cycle, and have a total of n). Hence the peak of both curves will come near this point. However, keep in mind that we

are looking at the maximum size of each queue throughout the network, and small degrees of contention can take place at any node, causing the jaggedness which is especially visible in the line corresponding to the arrivals queue. When both the arrival and departure queues have emptied for each processor, then we have completed routing for that set of n requests. Therefore the time to finish routing the set shown in figure 7–9 is about 87 cycles.

Figure 7–10, on the other hand, shows the routing behaviour where the number of links is much less than the number of threads in each node, as we have $s = 32$ and $l = 1$. Therefore the arrival queue never has more than 4 requests (as there are only four incoming links, each just one wire wide). As the links are not wide enough to empty the departure queues, they will naturally fill up. Significantly, not only do they fill up for the first s cycles while new requests are injected into the network, but the maximum queue size also increases significantly beyond that point, as hot spots develop in the network. In this example we see that the maximum departure size is almost 100 requests, over three times the number each processor injects, and this hot spot takes place about 300 cycles after routing has begun. Eventually requests begin to reach their destinations in rapid succession after 400 cycles, and routing completes at about 500 cycles.

Our goal is to use similar graphical techniques for a variety of fat mesh networks to determine two important attributes: the minimum average routing time, and the minimum link width which allows routing to take place in this minimum time.

7.3.1 Fat Rings

In the figure 7–11 below, we display the maximum departure queue size of a fat ring as a function of the link width for a 128 processor machine. Recall that a fat ring uses multithreading such that $s = p$. One feature of the fat ring can be seen initially, that the simple routing scheme eliminates contention and related hot spots, unlike the results we presented above. This is evident from the fact

that after the first 128 cycles during which new requests are being injected all departure queues start to empty, even in the case of rings with very small links, e.g. $l = 1$. If hot spots were arising we would expect at least some of these lines to continue rising after the first 128 cycles. However, there is a substantial difference in the time to complete routing the n requests based on the link width.

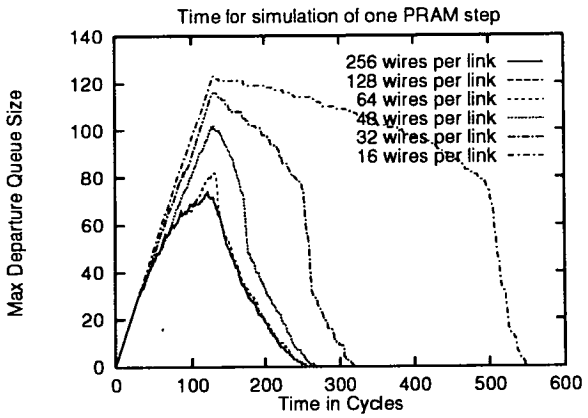


Figure 7–11: 128 Processor Fat Ring Routing Time.

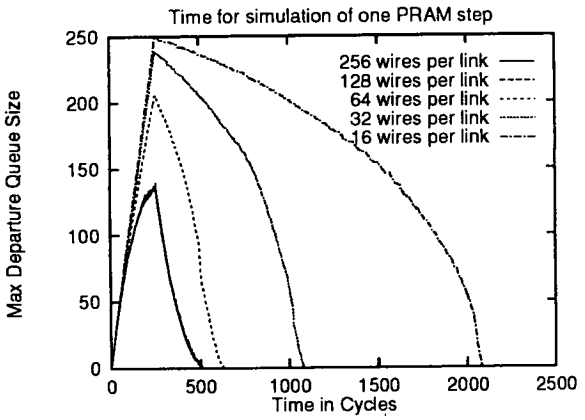


Figure 7–12: 256 Processor Fat Ring Routing Time

The minimum time is 255 cycles, or just under $2s$ (though in the case of the ring $s = p$). Recall that the last request will enter the network at time p, and the farthest any requests will need to travel is across p nodes. More surprisingly, we observe that this best case performance takes place with both $l = 128$ and $l = 64$, with the $l = 48$ case lagging behind by just a few cycles. Our theoretical estimate of the required bandwidth being $l = s$ does not take into account significant constants, as it appears from the figure that $l = s/2$ is sufficient to achieve optimal routing time. Note we have also included a curve corresponding to $l = 256$ or $l = 2s$, to emphasize that routing time will never improve further than the value achieved with $l = s$.

Similar results are shown in figure 7–12, but with a larger machine of 256 processors. Once again, the routing time is approximately $2p$ cycles, and architectures with link width $l \geq s/2$ all achieve this minimum routing time. If the link width is

p=128	p=256
255.4 cycles	511.4 cycles

Figure 7–13: Average Routing Times for Fat Ring

smaller we see substantially lower performance, such as the case for $l = 16$ where the maximum departure queue size is almost 250 requests and routing does not complete until after more than 2000 cycles.

The results shown in the graphs are the routing behaviour of one set of n EREW requests, corresponding to one PRAM step. However, other sets of requests gave very similar results, as one would expect given the randomizing nature of hashing. We give average routing times for these two sizes of ring in the below table 7–13.

7.3.2 Fat Meshes

Similar results for two and three dimensional fat meshes are shown in figures 7–14 and 7–15. Average values for fat mesh routing are shown in 7–16. From our theoretical arguments we expect routing to complete in $O(p^{1/r})$, but these graphs provide an indication of how the constants involved in routing time depend on the degree of the fat mesh. Recall that for a two dimensional mesh of 4K processors $p = 64$, while in the three dimensional mesh with the same number of processors $p^{1/r} = 16$. We see that as the dimension of the mesh increases, the constant c in the equation $routing - time = c \cdot p^{1/r}$ also increases. For the one dimensional fat ring we saw $c \approx 2$, and now we observe that the 2-D fat mesh has $c \approx 3$, whereas the 3-D fat mesh has $c \approx 4$. The worst case routing time, e.g. when the source and destination are diametrically opposite on the mesh, is $r \cdot p^{1/r} = diameter$. Furthermore, due to multithreading delays, the last request is injected into the network $s = p^{1/r}$ cycles after we begin processing, which results in:

$$routing - time = (r + 1) \cdot p^{1/r}$$

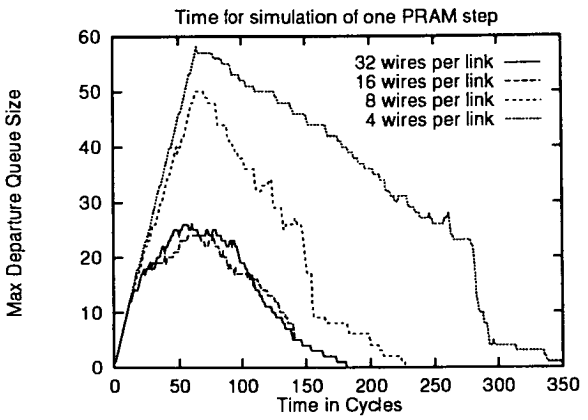


Figure 7–14: Routing Time for 64×64 Fat Mesh.

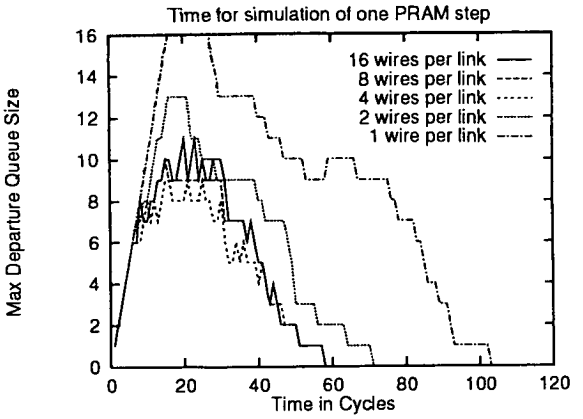


Figure 7–15: Routing Time for $16 \times 16 \times 16$ Fat Mesh.

where r is the degree or our mesh. This corresponds well to our practical observations.

This is the worst case time for a given request. We've used similar worst case arguments in determining our bandwidth bounds for each link of $l = s$. However, in practice we can expect to do fine in most cases with less bandwidth. In figure 7–15 we observed near optimal routing times with as little as 4 wires per link, despite somewhat higher maximum departure queue sizes for such low bandwidth. This highlights the disparity between our earlier worst case estimates and the average cases we typically see in practice. Note that in routing times we are concerned with worst case behaviour, as it only requires one request to take $O(diameter)$ time for our entire routing stage to be delayed. However, in the case of bandwidth considerations when we are considering the aggregate link bandwidth of the entire machine, and average case behaviour is more indicative of such bandwidth demands.

To obtain better average case requirements for bandwidth we take into account that messages will typically travel only $\frac{p^{1/r}}{2}$ links before reaching their destination. Therefore the correct amount of wires per link becomes also becomes $\frac{p^{1/r}}{2}$. It is conceivable that a routing pattern may require the worst case $p^{1/r}$ links per

64×64	$16 \times 16 \times 16$
183.2 cycles	56.2 cycles

Figure 7–16: Average Routing Times for Fat Meshes

cycle, but this appears unlikely to occur in practice given our randomized memory management scheme.

7.3.3 Memory Module Service Rates

One of the basic assumptions of our multithreaded nodes is that their basic power is unchanged over the RAM, i.e. each can only inject up to one request per cycle into the network, and each memory module can only service one request per cycle. Therefore the time required to simulate a PRAM instruction is not only the time required to route the requests to their destinations, but also to have those requests serviced (and possible returned to their source in the case of reads).

Our simulation also monitors closely the behaviour of the memory module. We have provided theoretical evidence that, despite the fact that a reasonably large number of requests will pass through a given node in any one cycle, the number of requests that will be destined for the local module rather than be forwarded on is relatively small. In Theorem 9 we show this number is likely to be less than $O(\log p)$ for the case of a fat ring, which is likely to be the worst case given it has the highest number of requests arriving in any one cycle.

We now augment this theoretical result by considering the experimental distribution of arrivals for a given processor as a function of time during our trace-driven simulations. Again we focus on what is likely the worst case, the fat ring. We show these results in figures 7–17 and 7–18. Each represent the number of requests destined for the local memory module of an arbitrary processor (processor number 1 in this case) during the simulation of a PRAM step. We see that the

distribution is quite even. As processing begins, few requests are destined locally. Then during the middle section of the PRAM step simulation we consistently see a small constant number of arrivals, rarely more than 3. And finally, towards the end of the processing of that PRAM step locally destined request arrivals become rare again, and virtually never do we see more than one arrival in a cycle.

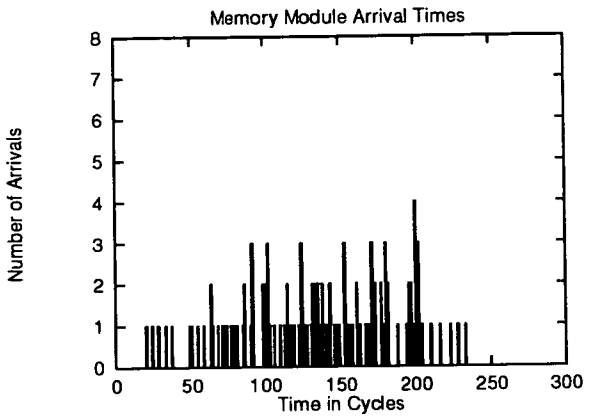


Figure 7–17: Memory Arrivals for p=128 Fat Ring.

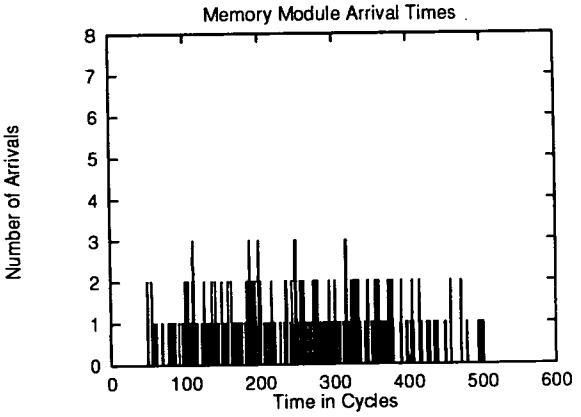


Figure 7–18: Memory Arrivals for p=256 Fat Ring.

The corresponding rates of growth for the memory module queue for this processor during execution of this same PRAM step are shown in figures 7–19 and 7–20. These curves correspond roughly to the integral of the arrival curves above, but with the constant service rate of the memory module subtracted. We see that during the early stages of the step the queue size is either one or zero, as few requests have arrived at the memory module. During the middle section of the step requests begin to arrive more frequently, but again, with no more than a small constant number arriving in any given cycle. The queue therefore does begin to slowly fill, but does not hold more than a small number of requests during any one cycle. And the final less busy stage of routing for this step allows the memory module to service any backlog which has developed during the middle stages, as each cycle without a new arrival will allow the module to consume one outstanding request from the queue. In practice, we rarely see more than one request queued at the end of routing for any PRAM step. The size of the queues observed for the

fat meshes of arbitrary dimension are yet smaller than those for the fat ring, as should be expected due to the smaller number of requests arriving in any given cycle.

Given this experimental evidence we believe that small memory module queues should be sufficient for all our architectures, though this size is likely to be a slowly growing function of the number of processors, as suggested by the theory. More importantly, we rarely expect memory module processing to be a performance bottleneck, despite our strict assumption that no more than one request will be serviced in any given cycle.

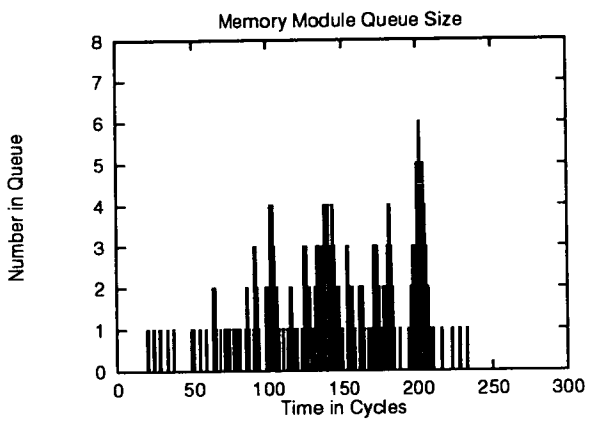


Figure 7–19: Queue Sizes for p=128 Fat Ring.

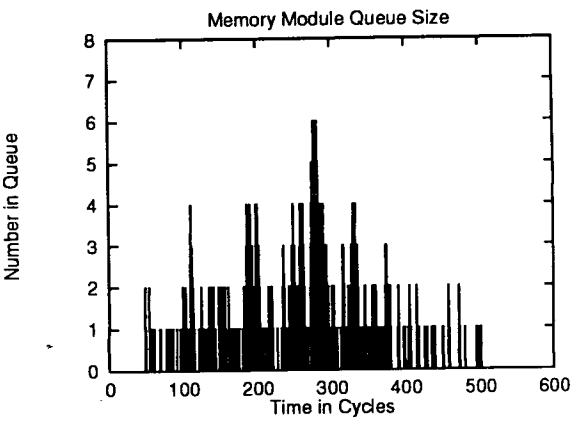


Figure 7–20: Queue Sizes for p=256 Fat Ring.

7.4 Processor Count

One other fundamental attribute of our proposed architecture that remains to be considered from the practical perspective is the relationship between the number of physical processors and threads within each processor. We earlier provided theoretical arguments for an r dimensional mesh having $p = n^{r/r+1}$ processors, and each processor running $s = p^{1/r}$ threads. Now we try to reinforce these arguments by providing routing times for networks for a range of processor/thread ratios.

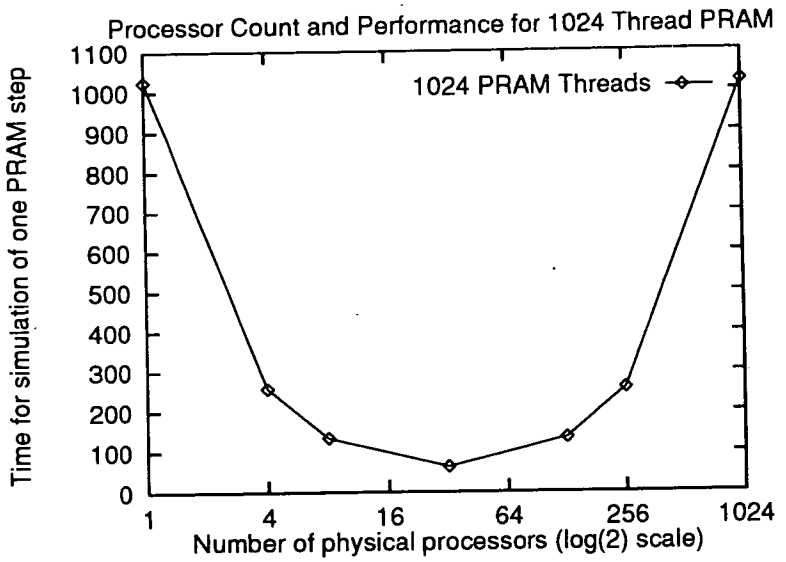


Figure 7–21: Processor Count with 1K Threads on Ring.

In figure 7–21 we see the results for the ring, with 1K PRAM threads, displayed on a semi-$\log_2$ scale. Given the high diameter of a ring, we expect the speedup from multithreading to be highest on such a network. We see that simulation of one PRAM step takes over 1000 cycles with either 1 or 1024 processors, and only about 64 with the optimal 32 processors. We observe a speedup of roughly 500 when comparing the multithreaded performance versus that of the non-multithreaded machine. This corresponds well to the $O(\sqrt{n})$ speedup we would expect, as we describe in the next chapter. Most importantly, the prescribed processor number appears to lead to a clear minima in routing times.

Results for the two dimensional and three dimensional fat meshes are presented in figures 7–22 and 7–23 respectively, now on a $\log_{10} \times \log_2$ scale. Both show clearly that the processor number determined theoretically as optimal does, in fact, result in the best performance. However, also as predicted, we see the benefits from use of multithreading being reduced as we consider higher dimensional meshes which naturally have lower diameters. The number of data points is reduced in figure 7–23 due to the practical difficulties in finding processor numbers with both cube

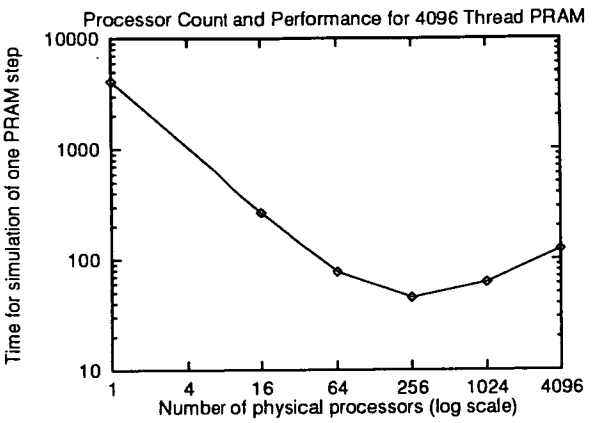


Figure 7–22: Processor Count on Two Dimensional Fat Mesh.

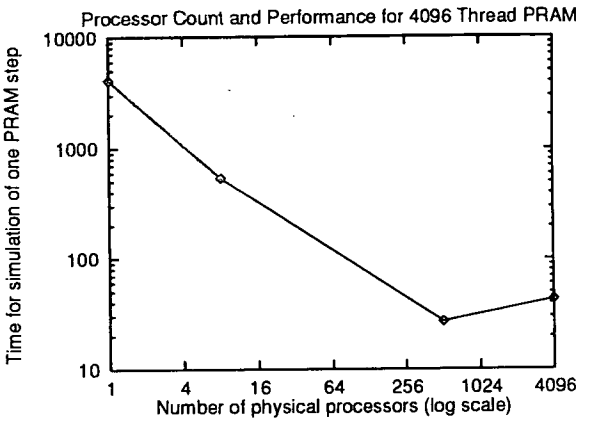


Figure 7–23: Processor Count on Three Dimensional Fat Mesh.

and forth roots which are integers. Both these figures assume a total of 4K PRAM threads.

Chapter 8

Conclusions

We hope that one of the quantitative suggestions resulting from this thesis to practical users of multiprocessors is to move away from the use of peak performance figures for characterizing an architecture, and instead focus on the sustainable performance of the architecture. The performance of multithreaded machines such as fat rings and fat meshes will provide consistent performance which will be lower than the peak performance of a similar architecture, but which will be consistently sustainable. We now provide some details regarding this point, as well as other main points of the thesis.

8.1 Multithreaded Performance

The initial idea behind using multithreading to hide latency is to increase the utilization of processors, thereby reducing the number of processors necessary to achieve roughly the same performance as a non-multithreaded machine supporting the same number of threads. Reconsider one of the main points from chapter 3, regarding the relationship between the delay of a simulation and the number of physical processors needed to make such a simulation efficient. Earlier efficient simulations maintained the following invariant:

$$n = ps$$

$$L(n) = s$$

Therefore the number of processors of a traditionally multithreaded machine would be $p = \frac{n}{L(n)}$, and ideally each of these p processors would be working with little or no idle time, thereby achieving nearly its peak performance. Therefore:

$$performance_{mt} = \frac{n}{L(n)} \cdot G$$

if we define G as the peak performance of one node. The performance of the equivalent non-multithreaded PRAM simulation would be n processors, each working at $\frac{1}{L(n)}$ times its peak performance. Hence the overall performance of that machine would be the same:

$$performance_{non-mt} = \frac{n}{L(n)} \cdot G$$

So the traditional role of multithreading is to achieve the same performance for a reduced number of processors, not to increase the performance.

One of the unique contributions of this thesis is to consider the relationship between multithreaded and non-multithreaded performance for high diameter networks. In this case, we achieve a reduction in the diameter of the network as we reduce the number of physical processors in the simulation. The multithreaded performance now becomes:

$$performance_{mt} = \frac{n}{diameter} \cdot G$$

which is clear from the fact that $p = \frac{n}{diameter}$ and all p processors will have idle time hidden such that they each run at near G performance. If we can achieve an optimally efficient simulation, as we defined earlier and showed exists for the fat ring and fat meshes, then $L(n) > L(p)$, and hence $performance_{mt} > performance_{non-mt}$.

Consider optimally efficient simulations on the class of fat meshes of degree r. For non-multithreaded simulations on meshes of degree r $L(n) = n^{1/r}$, whereas the equivalent n thread fat mesh efficient simulation will have $L(p) = n^{1/r+1}$. The performance of the inefficient simulation will be:

$$performance_{non-mt} = \frac{n}{n^{1/r}} \cdot G$$

The performance of the optimally efficient simulation will instead be:

$$performance_{mt} = \frac{n}{n^{1/r+1}} \cdot G$$

If we define the speedup from use of multithreading as the ratio of performance of the n processor non-multithreaded mesh and the p processor multithreaded fat mesh, where as usual $p = n^{1/r+1}$:

$$speedup = \frac{performance_{mt}}{performance_{non-mt}}$$

The speedup achieved with optimally efficient fat mesh simulations of degree r is:

$$\begin{aligned} speedup_{fat-mesh} &= \frac{n^{1/r}}{n^{1/r+1}} \\ &= n^{\frac{1}{r(r+1)}} \end{aligned}$$

Given in terms of p this becomes:

$$\begin{aligned} speedup_{fat-mesh} &= \left(p^{r+1/r}\right)^{\frac{1}{r(r+1)}} \\ &= p^{1/r^2} \end{aligned}$$

These calculations also apply to the special case of the fat ring, where $r = 1$, resulting in:

$$speedup_{fat-ring} = p = n^{1/2}$$

Clearly the magnitude of the reduction we would hope to achieve through optimally efficient simulations decreases as the dimension of the network increases, as shown in table 8–1. However, we see that asymptotically all fat mesh optimally efficient simulations are faster than their non-multithreaded counterparts.

dimension	*speedup*
1	$n^{1/2}$
2	$n^{1/6}$
3	$n^{1/12}$
4	$n^{1/24}$

Figure 8–1: Multithreading Speedup as function of mesh dimension.

8.2 Sustainable Performance

It is hoped that these performance gains will be considered within a wider context than solely that of PRAM simulations. One of the most significant assumptions in the consideration of PRAM simulations is that each PRAM processor is assumed to communicate on each PRAM step. Therefore the consideration of peak performance is avoided, as we are by definition only concerned with worst case performance.

In the practice of parallel computing, machine vendors routinely quote peak performance figures for their architecture. These typically relate to performance likely to be seen only on "Embarrassingly Parallel" applications; those that do little or no communications between computations. Users in practice often use applications which follow closer to the worst-case PRAM simulations assumptions; that each processor will frequently communicate remotely. Hence a large discrepancy often exists between what the purchaser of a multiprocessor expects in terms of performance, and what is regularly achieved on typical applications. We now consider how efficient fat mesh support for shared memory might help to reduce this discrepancy.

Figures 8–2 and 8–3 represent hypothetical performance curves for multiprocessors in various situations. The upper line in figure 8–2 shows peak performance

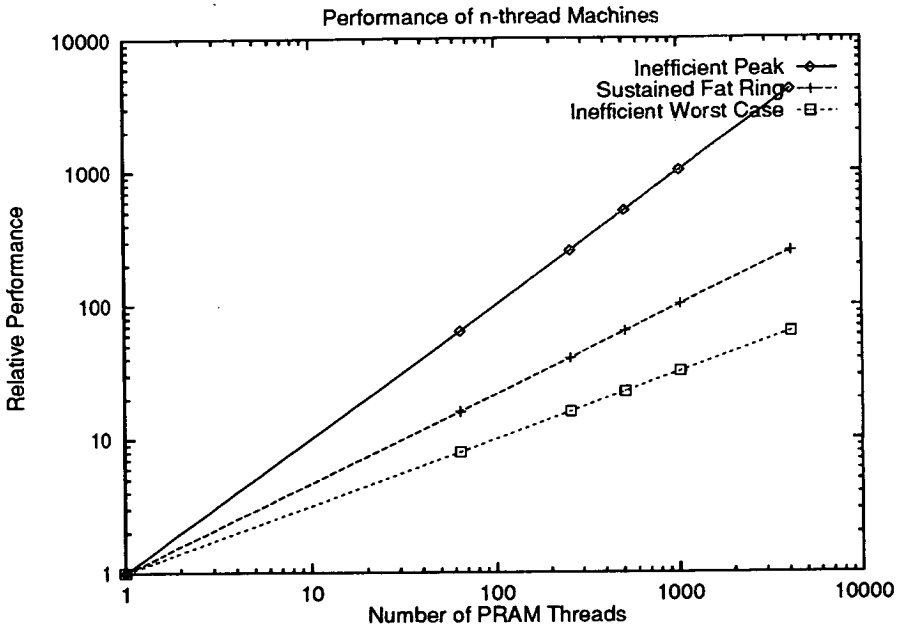


Figure 8–2: Two Dimensional Mesh Performance as Function of n.

for an n processor two-dimensional mesh. Again, this corresponds to fully local computations, and performance is designated in units of multiples of uniprocessor performance. Naturally this curve has a slope of one, as each processor is defined as running at near peak performance. If, instead, we consider sustainable performance of an inefficient multiprocessor by assuming that each processor will be routing messages a distance of $O(diameter)$ away on each step, then this corresponds to the lower *inefficient worst case* curve. In practice, the performance of virtually any application will fall between or on these two lines, depending on the type of communications patterns used. It is expected that performance will typically fall well below the peak performance curve. However, arguably more detrimental to the practical use of parallel computers, is the fact that it is rarely known where between these two curves the performance of any one application will lie, until that application has been actually run on the machine. This clearly hinders any *a priori* analysis as to the suitability of parallel computing for a given

set of applications, and is an impediment to the uptake of parallel computing on the whole.

The middle line corresponds to the sustained performance expected from a p processor fat mesh supporting the same number of threads of execution. The fact that this line is above the line for worst case performance of the inefficient machine is good news. However, even more important is the fact that virtually all applications will see performance which lands on the curve shown, and this is true whether they are embarrassingly parallel applications or make frequent accesses to shared-memory. Users of an optimally efficient PRAM simulation, such as those provided on the fat meshes and fat ring, can trivially estimate the practical performance of their applications by simply counting the number of PRAM steps to be simulated. Our focus on the PRAM model has benefitted us by its worst case communications assumptions, and use of multithreading has allowed us to improve our performance somewhat beyond the worst case, while still maintaining an emphasis on sustainable performance.

Note that we are now comparing architectures in terms of the number of execution threads they support. If we instead compare them in terms of performance as a function of processor number, then naturally the efficient simulations will show the same characteristics as the peak performance curves; they will have a slope of one, as the processors have little or no idle time. This situation is shown in figure 8–3.

8.3 Concurrent Access

Most PRAM simulations that support EREW access are also extended to support CRCW access. In fact, it has become common to extend simulations to allow concurrent access by simply pointing out that, in a $O(\log n)$ time inefficient simulation, one simple needs an additional $O(\log n)$ cycle pre and post processing

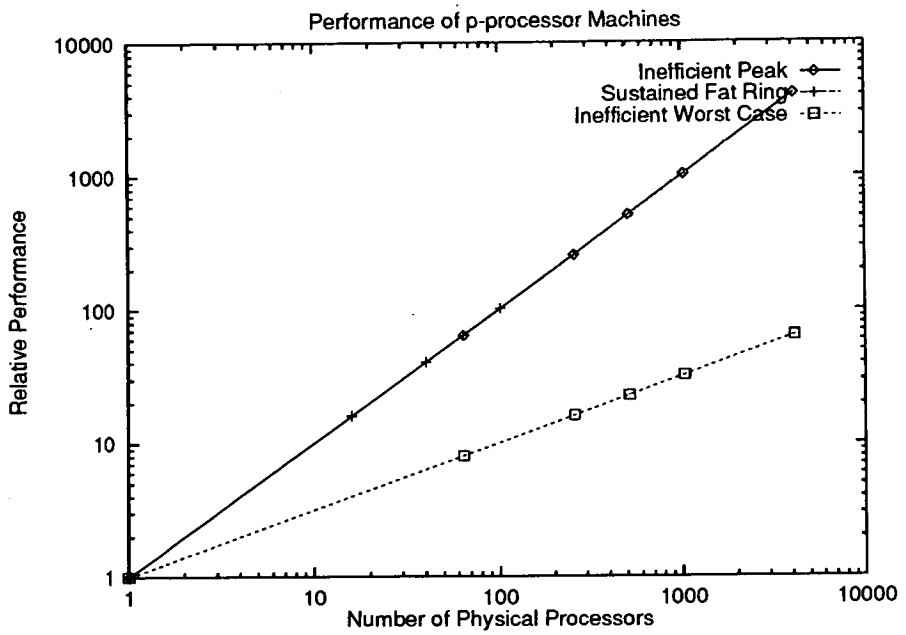


Figure 8–3: Two Dimensional Mesh Performance as Function of p.

phase, and in this way all concurrent access may be reduced to EREW [Upfal and Wigderson 1987, Karlin and Upfal 1986]. Additionally, many authors have argued for some form of equivalence between the various conflict resolution strategies for CRCW models, as all can be supported in $O(\log n)$ time on an MPC. On the whole these results have served to down play the difference between concurrent and exclusive access models, and to suggest that any hardware supporting one will be able to support the other.

The results of this thesis contradict this suggestion. We have described the difficulties in providing an optimally efficient CRCW simulation on a bounded degree network without combining, despite the fact that such an EREW simulation is relatively easy. In this sense the task of providing optimally efficient simulations provides a form of separation of the two PRAM models. In [Kruskal et al. 1990] a hierarchy of complexity classes are suggested for use in parallel computing. In particular, problems which may be solved efficiently, i.e. $E = O(1)$, are considered as a different class from those that can only be solved with $E < 1$. While effi-

cient EREW simulations fall clearly into the class of efficiently solvable problems, solving the problem of CRCW simulation efficiently is a difficult task, particularly without the use of a combining networks or special purpose sorting algorithms.

This situation is displayed in figure 8–4. Any problem which is in the class of efficient parallel solutions will exist in the innermost class, but also will be contained within the class of inefficient solutions, as an inefficient solution to any such problem may be trivially found. The figure shows the problem of simulating concurrent access through use of a general sorting routine as falling only into the inefficient class of problems. We consider it an open problem whether or not there exists a special purpose $O(n)$ steps sorting algorithm which would allow CRCW simulations to be provided with optimal efficiency on bounded degree networks without combining.

We hope that these ideas will serve to isolate the requirements for a CRCW simulation, and similarly provide insight into the costs and benefits of the two models. An interesting subject for further investigation would be an attempt to define in a general sense just what can be expected from a simulation that depends on hardware CRCW combining, though it is beyond the scope of this thesis.

8.4 Theory versus Practice in Architecture

It is hoped that the theoretical results of this thesis will have tangible implications for the practical construction and use of parallel computers. However, it is an open question just how faithful an implementation of theoretical ideas needs to be in order to be useful. Though we desire to only make theoretical assumptions that may be instituted in a practical sense, this desire is counterbalanced by the need to provide a simple enough theoretical structure to allow progress to be made.

An example of this tradeoff can be seen in the area of synchronization. The PRAM model is consistently assumed to be a synchronous model, whether it

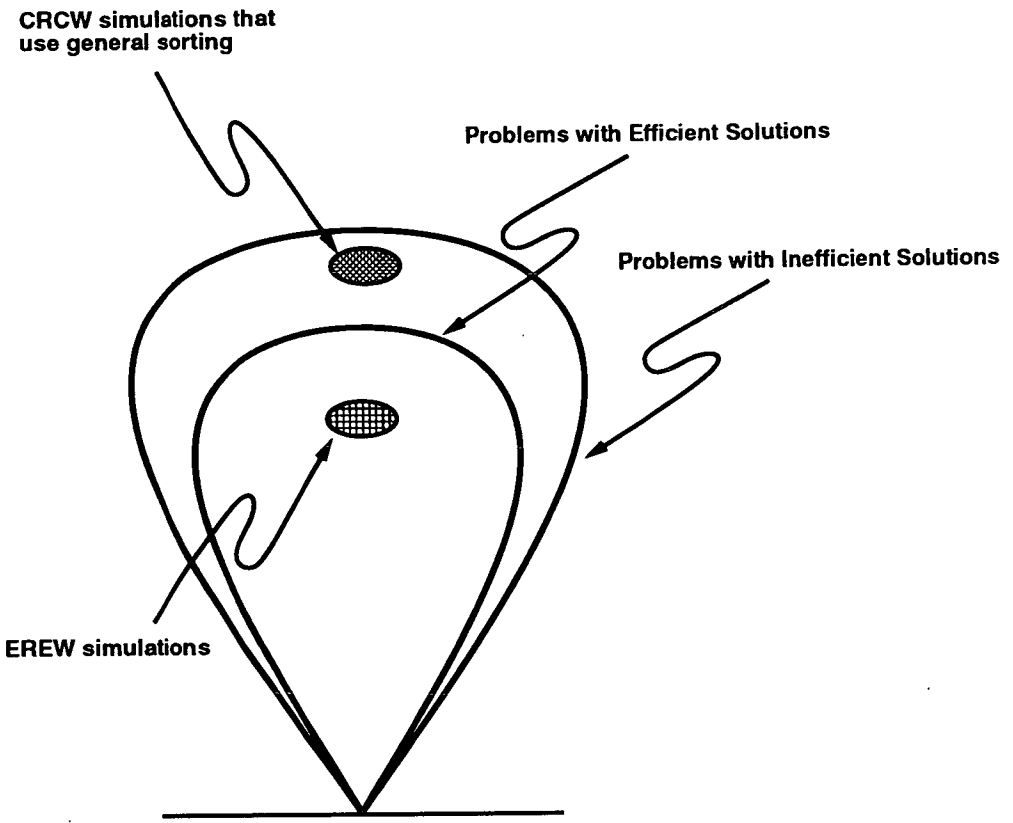


Figure 8–4: Simulation Complexity Classes.

is assumed to be a MIMD model, or strictly SIMD. Without this assumption, much that is easy on the model becomes difficult, and progress slows considerably. E.g., simply defining the meaning of one machine cycle in the context of a fully asynchronous MIMD multiprocessor requires substantial work [Cole and Zajicek 1990]. On the other hand, it is also clear that providing hardware support to ensure that a MIMD multiprocessor executes its instruction in synchronous lock-step with all other processors is a significant hardware overhead, which in practice would likely have substantial performance penalties. Much practical work in shared memory support now addresses the problem of just how much synchronization needs to take place, and how to reduce that level as much as possible [Harris and Topham 1994a, Gharachorloo et al. 1992]. Therefore we acknowledge that synchronization is important for the utility of the PRAM model, but perhaps

implementations are better off using restricted synchronization, as suggested by the practical literature.

Another questionable assumption is that of context switch time for such a multithreaded simulation. It is perhaps overly optimistic to assume that the flow of control can move from one thread of execution to another in one, or even a constant number, of cycles. Given the number of context switches we are assuming, even a small degree of overhead may have substantial performance penalties. This fact was also noted in [Bilardi and Preparata 1992]. Such penalties are considered from a practical perspective in [Boothe and Ranade 1992], where it is suggested that giving a thread control as long as possible, rather than switching each cycle, will provide improved performance. However, again we note that from a theoretical perspective, such program dependent scheduling techniques make generalizations difficult to apply. We also note that various practical research efforts have carried on with the fine grain scheduling approach we have considered, including that discussed in [Alverson et al. 1990].

In addition to performance issues, there are various other issues that are neglected in our high level consideration of multithreading. In practice, each thread has a substantial amount of state, which must be stored in the form of registers and pointers, each of which takes up VLSI area, and hence adds cost. Additionally, the added concurrency implicit in the use of multithreading depends on the lack of data dependencies between threads which are running in parallel, particularly if any asynchrony is present in the system. In such a working system the burden of proof as to such a lack of dependencies would fall upon the compiler, and hence present an added software cost to the designers and builders of such a compiler.

However, given these practical problems we consider theoretical tools worthwhile and productive, particularly given the lack of fundamentally sound alternatives. Therefore we advocate a perspective based on compromise; that theoretical assumptions should be made simple enough to allow swift progress within that framework, but such assumptions should only be implemented in practice if they

are not exorbitantly expensive. This, of course, allows for the possibility that theoretical performance will be substantially higher than practical performance, based on some hardware conflicts which appear once strict assumptions have been relaxed. However, it appears this middle ground is a necessary evil when mixing theory with practice.

8.5 Future Work

Many possible extensions to this work suggest themselves. The first of which is addressing the issue of how easy it is in VLSI to layout a fat ring or fat mesh. In the thesis we have focused on addressing the question of how a mesh and ring can support an efficient simulation, rather than asking the question of whether the VLSI area required for such an augmented network would be a justified expense. Despite this oversight, we hope that one could show that the overall area would be wisely spent on a fat mesh or fat ring. Intuitively we expect such networks to be easily amenable to the two or three dimensional requirements of standard VLSI models, however, to prove this for certain requires more work. Ideally if one were to pursue this topic it would result in a comparison of the VLSI area required for a p processor fat mesh with that required for a n processor non-multithreaded mesh. We would like to show that the area requirements are similar, or perhaps that, given the fact that $p < n$ that such a fat mesh actually requires less silicon area than the n processor mesh. It would likely be relatively easy to show that the layout of a fat mesh is simpler than a hypercube, which is inherently difficult to layout in two or three dimensions. Though we would not expect a fat mesh to constitute a universal network, we would expect similar analysis as appeared in [Leiserson 1985] for universal networks to bear interesting fruit.

More and more of the focus of the high performance computing industry is on reducing the price of large scale multiprocessors. Ideally a consideration of VLSI

layout issues would eventually lend itself to cost analysis, similar to that presented in [Ranade et al. 1988]. We consider our belief that multithreaded machines can provide more performance per dollar than their non-multithreaded counterparts as an important assumption that requires further consideration.

Along the lines of the context switching discussion above, we would also be interested in providing a programming model such that users with a clear knowledge of how often a context switch is necessary to hide latency, could receive benefits in the form of improved performance or lower bandwidth requirements. This would be moving towards the model of [Valiant 1990b], but would also provide an interesting theoretical analogy to the problems of programming real multiprocessors with weakly consistent memory models [Gharachorloo et al. 1992].

Bibliography

[Aggarwal et al. 1990] A. AGGARWAL, A. CHANDRA AND M. SNIR, 1990. Communication Complexity of PRAMs. *Theoretical Computer Science*, No. 71, pp. 3-28.

[Ajtai et al. 1983] M. AJTAI, J. KOMLOS, AND E. SZEMEREDI. Sorting in C $\log n$ parallel steps. *Combinatorica*, Vol. 6, No. 2, pp. 83-96.

[Akl 1989a] S.G. AKL, 1989. On The Power of Concurrent Memory Access. *Computing and Information*, pp. 49-55. Elsevier Science Publishers, R. Janickyi and W. W. Koczkodaj editors.

[Akl 1989b] S.G.AKL, 1989. The Design and Analysis of Parallel Algorithms. Prentice-Hall Publishers.

[Aleliunas 1982] R. ALELIUNAS, 1982. Randomized parallel communication. *1st Annual Symp. on Principles of Distributed Computing*, pp. 60-72.

[Alt et al. 1987] H. ALT, T. HAGERUP, K. MEHLHORN, AND F. P. PREPARATA, 1987. Deterministic Simulation of Idealized Parallel Computers on More Realistic Ones. *SIAM Journal of Computing*, Vol. 16, No. 5, pp. 808-835.

[Alon 1986] N. ALON, 1986. Eigenvalues and Expanders. *Combinatorica*, Vol. 6, No. 2, pp. 83-96.

[Alverson et al. 1990] R. ALVERSON, D. CALLAHAN, D. CUMMINGS, B. KOBLENZ, A. PORTERFIELD, AND B. SMITH, 1990. The Tera Computer System. *Proc. International Conf. on Supercomputing*, IEEE Press, pp. 1-6.

[Arora et al. 1990] S. ARORA, T. LEIGHTON, AND B. MAGGS, 1990 On-line Algorithms for Path Selection in a Nonblocking Network. *Proc. 22th ACM Symp. on Theory of Computing.*, pp. 149-158.

[Aumann and Schuster 1991] Y. AUMANN AND A. SHUSTER, 1991. Improved Memory Utilization in Deterministic PRAM Simulation. *Journal of Parallel and Distributed Computing*, Vol. 12, pp. 146-151.

[Batcher 1968] K. E. BATCHER, 1968. Sorting Networks and their Applications. *Proc. AFIPS Spring Joint Computer Conference*, pp. 307-314.

[Bilardi and Preparata 1992] G. BILARDI AND F. PREPARATA, 1992. Horizons of Parallel Computations. *Proc. of International Conference for 25th Anniversary of INRIA*, Bensoussan, Verjus, Eds., Paris, France.

[Bisseling and McColl 1994] R. BISSELING AND W. MCCOLL, 1994. Scientific Computing on Bulk Synchronous Parallel Architectures. *Proc. of 13th IFIP World Computer Congress*, Elsevier Publishing.

[Boothe and Ranade 1992] B. BOOTHE AND A. RANADE, 1992. Improved Multithreading Techniques for Hiding Communications Latency in Multiprocessors. *Proc. of International Symp. on Computer Architecture.*

[Borodin and Hopcroft 1982] A.BORODIN AND J.E.HOPCROFT, 1982. Routing, Merging and Sorting on Parallel Models of Computation. *Proc. 14th ACM Symp. on Theory of Computing.*

[Carter and Wegman 1979] J. L. CARTER AND M. N. WEGMAN, 1979. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, Vol. 18, pp. 143-154.

[Chin and McColl 1994] A. CHIN AND W. F. MCCOLL, 1994 Virtual Shared Memory: Algorithms and Complexity. *Information and Computation*, V. 113, pp. 199-219.

[Cook 1984] STEPHEN A. COOK, 1984. A Taxonomy of Problems with Fast Parallel Algorithms. *Proc. 1983 International FCT Conference*, 1983.

[Cole 1988] R. COLE, 1988 Parallel Merge Sort. *SIAM Journal of Computing*, Vol. 17, No. 4, pp. 770-784.

[Cole and Zajicek 1990] R. COLE AND O. ZAJICEK, 1990. The APRAM: Incorporating Asynchrony into the PRAM Model. *Proc. of 1st Symp. on Parallel Algorithms and Architectures.*

[Culler et al. 1993] D. CULLER, R. KARP, D. PATTERSON, A. SAHAY, K. SCHAUSER, E. SANTOS, R. SUBRAMONIAN AND T. VON EICKEN, 1993 LogP: Towards a Realistic Model of Parallel Computation. *Proc. 4th Symp. on Principles and Practice of Parallel Programming*, ACM Press.

[Dietzfelbinger and Meyer auf der Heide 1990] M.DIETZFELBINGER AND F. MEYER AUF DER HEIDE, 1990. How to Distribute a Dictionary in a Complete Network. *Proc. 22nd Symp. on Theory of Computing.*

[Dietzfelbinger and Meyer auf der Heide 1993] M.DIETZFELBINGER AND F. MEYER AUF DER HEIDE, 1993. Simple, Efficient Shared Memory Simulations. *Proc. 5th Symp. on Parallel Algorithms and Architectures.*

[Engelman and Keller 1993] C. ENGELMAN AND J. KELLER, 1993. Simulation-based Comparison of Hash Functions for Emulated Shared Memory. *Proc. of Symp. on Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science number 694.

[Fortune and Wyllie 1978] S. FORTUNE AND J. WYLLIE, 1978. Parallelism in Random Access Machines. *Proc. of 10th Symp. on Theory of Computing*, pp. 114-118.

[Gerbessiotis and Valiant 1992] A. GERBESSIOTIS AND L. VALIANT, 1992. Direct Bulk-Synchronous Parallel Algorithms. PROC. OF SCANDINAVIAN WORKSHOP ON ALGORITHM THEORY, LNCS Vol. 621, Springer-Verlag Press.

[Gharachorloo et al. 1992] K. GHARACHORLOO, S. ADVE, A. GUPTA, J. HENNESSY, AND M. HILL. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, Vol. 15, 1992.

[Gibbons 1988] A. GIBBONS AND W. RYTTER, 1988. Efficient Parallel Algorithms. Cambridge University Press, Cambridge, 1988.

[Goldschlager 1982] L. M. GOLDSCHLAGER, 1982 A Unified Approach to Models of Synchronous Parallel Machines. *Journal of the ACM*, Vol. 29, pp. 1073-1086.

[Harris 1992] T.J. HARRIS. Sorting on Novel Interconnection Networks. *Proceedings of Parallel Numerical Analysis Workshop '92*, Edinburgh, June 1992.

[Harris 1994] T. J. HARRIS, 1994. A Survey of PRAM Simulation Techniques. *ACM Computing Surveys*, June 1994, Vol. 36, No. 2, pp. 187-206.

[Harris and Cole 1993] T.J. HARRIS AND M. I. COLE, 1993. The Parameterized PRAM. *Proceedings of the International Workshop on Parallel and Distributed Processing '93*, Sofia, Bulgaria, May 1993, Elsevier publishing.

[Harris and Topham 1994a] T. J. HARRIS AND N.P.TOPHAM Performance of Weak Consistency Schemes on the DEC Alpha. Advances in Parallel Computing 9: Trends and Applications, G.R. Joubert et al. Eds, North-Holland publishers, 1994.

[Harris and Topham 1994b] T. J. HARRIS AND N.P.TOPHAM The Use of Caching in Decoupled Multiprocessors with Shared Memory. *Proceedings of International Workshop on Large Scale Shared Memory Systems*, Cancun, Mexico, April 1994, IEEE Press.

[Harris and Topham 1994c] T. J. HARRIS AND N.P.TOPHAM The Scalability of Decoupled Multiprocessors. *Proceedings of Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, IEEE Press.

[Herley 1989] K. T. HERLEY, 1989. Efficient Simulations of Small Shared Memories on Bounded Degree Networks. *Proc. 30th Symp. on Foundations of Computer Science*, pp. 390-395.

[Herley and Bilardi 1988] K.T. HERLEY AND G. BILARDI, 1988. Determinstic Simulations of PRAMs on Bounded Degree Networks. *Proc. of 26th Allerton Conference on Commication, Control and Computation*, Monticello, IL.

[Hornick and Preparata 1991] S. W. HORNICK AND F. P. PREPARATA, 1991 Deterministic P-RAM Simulation with Constant Redundancy. *Information and Computation*, No. 92, pp. 81-96.

[Kaklamanis et al. 1990] C. KAKLAMANIS, D. KRIZANC AND T. TSANTILAS, 1990. Tight Bounds for Oblivious Routing in the Hypercube. *Proc. of 2nd ACM Symp. on Parallel Algorithms and Architectures*, pp. 31-36.

[Karlin and Upfal 1986] A. R. KARLIN AND E. UPFAL. Parallel Hashing - An Efficient Implementation of Shared Memory. *Proc. of the 18th Symp. on Theory of Computing*, pp. 160-168.

[Karp and Ramachandran 1990] RICHARD M. KARP AND VIJAYA RAACHANDRAN, 1990. Parallel Algorithms for Shared Memory Machines. Handbook of Theoretical Computer Science. Elsevier Science Publishers.

[Karp et al. 1992] R. M. KARP, M. LUBY, AND F. MEYER AUF DER HEIDE. Efficient PRAM Simulation on a Distributed Memory Machine. *Proc. of the 24th Symp. on the Theory of Computing.*

[Kendal Square Research 1991] KENDAL SQUARE RESEARCH. Architecture of the KSR-1. *KSR Tech. Report*, 170 Tracer Lane, Waltham, MA.

[Krizanc 1991] D. KRIZANC, 1991. Oblivious Routing with Limited Buffer Capacity. *Journal of Computer and System Sciences*, Vol. 43, pp. 317-327.

[Kruskal et al. 1990] C.P. KRUSKAL, L. RUDOLPH, AND M. SNIR, 1990. A Complexity Theory of Efficient Parallel Algorithms. *Theoretical Computer Science*, Vol. 71, pp. 95-132.

[Kucera 1982] L. KUCERA. Parallel Computation and Conflicts in Memory Access. *Information Processing Letters*, Vol. 14, No. 2, pp. 93-96.

[Kunde 1987] M. KUNDE, 1987. Optimal Sorting on Multi-Dimensionally Mesh-Connected Computers. *Proc. 4th Symp. on Theoretical Aspects of Computer Sciences*, pp. 408-4-19, Springer-Verlag Press.

[Leighton 1985] F.T.LEIGHTON, 1985. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Trans. on Computers*, Vol. C-34, No. 4, pp. 344-353.

[Leighton 1989] F.T.LEIGHTON, 1989. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. *Proc. 30th Symp. on Foundations of Computer Science*, pp. 384-389.

[Leighton 1992] F.T.LEIGHTON, 1992 Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann Publishers, San Mateo, California.

[Leiserson 1985] C.E. LEISERSON, 1985. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Trans. on Computers*, Vol. C-34, No. 10, pp. 892-901.

[Leppanen 1993] V. LEPPANEN, 1993. PRAM Computation on Mesh Structures. *Research Report R-93-9*, Computer Science Department, University of Turku, Finland.

[Luccio et al. 1988] F. LUCCIO, A. PEITRACAPRINA, AND G. PUCCI, 1988. A Probabilistic simulation of PRAMs on a bounded Degree Network. *Information Processing Letters*, Vol. 28, pp. 141-147.

[Luccio et al. 1990] F. LUCCIO, A. PEITRACAPRINA, AND G. PUCCI, 1990. A New Scheme for the Deterministic Simulation of PRAMs in VLSI. *Algorithmica*, Vol. 5, pp. 529-536.

[Luccio et al. 1991] F. LUCCIO, A. PEITRACAPRINA, AND G. PUCCI, 1991. Analysis of Parallel Uniform Hashing. *Information Processing Letters*, Vol. 37, pp. 67-69.

[Mano 1979] M MANO, 1979. Digital Logic and Computer Design. Prentice-Hall publishing, Englewood Cliffs, New Jersey.

[Mansour et al. 1990] Y. MANSOUR, N. NISAN AND P. TIWARI, 1990. The Computational Complexity of Universal Hashing. *Proc. of 22nd Symp. on Theory of Computing*, pp. 160-168.

[Matias and Vishkin 1991] Y. MATIAS AND U. VISHKIN, 1991. Converting High Probability into Nearly-constant Time - with applications to Parallel Hashing. *Proc. of 23rd Symp. on Theory of Computing*, pp. 307-316.

[McColl 1992] W. F. MCCOLL, 1992. General Purpose Parallel Computing. *Proc. 1991 ALCOM Spring School on Parallel Computation*, Gibbons and Spirakis editors, Cambridge University Press.

[Mehlhorn and Vishkin 1984] K. MEHLHORN AND U. VISHKIN, 1984. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984.

[Meyer auf der Heide and Wigderson] F. MEYER AUF DER HEIDE AND A. WIGDERSON, 1987. The Complexity of Parallel Sorting. *SIAM Journal of Computing*, Vol. 16, No. 1, pp. 100-107.

[Natvig 1990] L. NATVIG, 1990 Investigating the Practical Valoue of Cole's $O(\log n)$ time CREW PRAM Merge Sort Algorithm. *Proc. of 5th International Symp. on Computer and Information Sciences.*

[Paterson 1987] M.S.PATERSON, 1987. Improved Sorting Networks with $O(\log N)$ Depth. *Tech. Report RR 89, University of Warwick.*

[Pfister and Norton 1985] G. F. PFISTER AND V. A. NORTON, 1985. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Trans. on Computers.* Vol. C-34, No. 10.

[Pietracaprina et al. 1994] A. PIETRACAPRINA, G. PUCCI, AND J. SYBEYN, 1994. Constructive Deterministic PRAM Simulation on a Mesh-Connected Computer. *Proc. of 6th Symp. on Parallel Algorithms and Architectures.*

[Pippenger 1984] N. PIPPENGER, 1984 Parallel Communication with Limited Buffers. *25nd Symp. on Foundations of Computer Science*, pp. 127-136.

[Rabin 1989] M. O. RABIN, 1989 Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance. *Journal of the ACM*, Vol. 36, No. 2, pp. 335-348.

[Ranade et al. 1988] A.C.RANADE, S. N. BHATT, AND S. L. JOHNSSON, 1988. The Fluent Abstract Machine. *Fifth MIT Conference on Advanced Research in VLSI*, pp. 71-94.

[Ranade 1991] A.C.RANADE, 1991. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, Vol. 42, pp. 307-326.

[Sanz 1988] J. L. C. SANZ ED., 1988. Opportunities and Constraints of Parallel Computing, IBM workshop, Almaden Research Center, San Jose, California.

[Schwartz 1980] J. T. SCHWARTZ, 1980. Ultracomputers. *ACM Trans. on Programming Languages and Systems*, Vol. 2, pp. 484-521.

[Shiloach and Vishkin 1981] Y. SHILOACH AND U. VISHKIN, 1981. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of the ACM*, Vol. 2, pp. 88-102.

[Sibeyn and Harris 1994] J. SIBEYN AND T. J. HARRIS, 1994. Exploiting Locality in LT-RAM Computations. *Proceedings of Forth Scandinavian Workshop on Algorithm Theory*, Aarhus, Denmark, July 1994, Springer-Verlag LNCS Series.

[Siegel 1989] A. SIEGEL, 1989. On universal classes of fast high performance hash functions, their time-space tradeoff, and their application. *Proc. 30th Symp. on Foundations of Computer Science*, pp. 20-24.

[Smith 1978] B.J. SMITH, 1978. A Pipelined, Shared Resource, MIMD Computer. *Proc. of International Conference on Parallel Processing.*

[Thomas 1979] R. H. THOMAS, 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, page 180.

[Upfal 1984a] E. UPFAL, 1984A. A Probabilistic Relation Between Desirable and Feasible Models of Parallel Computation. *Proc. of 16th Symp. on the Theory of Computing*, pp. 258-265.

[Upfal and Wigderson 1987] E. UPFAL AND A. WIGDERSON, 1987. How to Share Memory in a Distributed System. *Journal of the ACM*, pp. 116-127.

[Upfal 1984b] E. UPFAL, 1984B. Efficient Schemes for Parallel Communication. *Journal of the ACM*, Vol. 31, No. 3, pp. 507-517.

[Upfal 1989] E. UPFAL. An $O(\log n)$ deterministic packet routing scheme. *Proc. 21st Symp. on Theory of Computing,* pp. 241-250.

[Vishkin 1982] U. VISHKIN, 1982. Implementation of Simultaneous Memory Address Access in Models That Forbid It. *Journal of Algorithms*, Vol. 4, pp. 45-50.

[Vishkin 1984] U. VISHKIN, 1984. A Parallel-Design Distributed-Implementation (PDDI) General-Purpose Computer. *Theoretical Computer Science*, Vol. 32, pp. 157-172.

[Valiant 1982] L.G. VALIANT, 1982. A scheme for fast parallel communication. *SIAM Journal on Computing*, Vol. 11, No. 2, pp. 350-361.

[Valiant 1983] L.G. VALIANT, 1983. Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Trans. on Computers*, Vol. C-32, No. 9, pp. 861-863.

[Valiant 1990a] L.G. VALIANT, 1990A. General Purpose Parallel Architectures. Handbook of Theoretical Computer Science. Elsevier Science Publishers.

[Valiant 1990b] L.G. VALIANT, 1990B. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33:103–111.

[Valiant and Brebner 1981] L.G. VALIANT AND G. J. BREBNER, 1981. Universal Schemes for Parallel Communication. *Symp. on Theory of Computing*, pp. 263-277.

[Weber and Gupta 1989] W. WEBER AND A. GUPTA, 1989. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. *Proc. 16th International Symp. on Computer Architecture.*