A Survey of PRAM Simulation Techniques

TIM J. HARRIS

Computer Science Department, University of Edinburgh, EH9 3JZ, Scotland

The Parallel Random Access Machine (PRAM) is an abstract model of parallel computation which allows researchers to focus on the essential characteristics of a parallel architecture and ignore other details. The PRAM has long been acknowledged to be a useful tool for the study of parallel computing, but unfortunately it is not physically implementable in hardware. In order to take advantage of the broad base of algorithms and results regarding this high-level abstraction one needs general methods for allowing the execution of PRAM algorithms on more realistic machines. In the following we survey these methods, which we refer to as PRAM simulation techniques. The general issues of memory management and routing are discussed, and both randomized and deterministic solutions are considered. We show that good theoretical solutions to many of the subproblems in PRAM simulation have been developed, though questions still exist as to their practical utility. This article should allow those performing research in this field to become well acquainted with the current state of the art, while allowing the novice to get an intuitive feeling for the fundamental questions being considered. The introduction should provide a concise tutorial for those unfamiliar with the problem of PRAM simulation.

Categories and Subject Descriptors: F.1.1 [**Theory of Computation**]: Models of Computation—*bounded-action devices*; *relations among models*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*interconnection architectures*; *parallel processors*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*routing and layout*; *sorting and searching*

General Terms: Algorithms

Additional Key Words and Phrases: Bounded-degree networks, models of parallel computation, Parallel complexity theory

INTRODUCTION

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

(1) Can enough parallelism in the problem be identified to allow an efficient solution?
(2) Can the processors share the data necessary in the problem fast enough (given their organization) to allow an efficient 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 work was partially performed while the author was resident at the Edinburgh Parallel Computing Centre. It was funded by the British Science and Engineering Council's Novel Architecture Computing Committee, contract no. B18534, as well as European Community ESPRIT project SHIPS, no. P6253.

CONTENTS

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 be broken roughly 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 practice, 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.

An (n, m)-PRAM consists of n processors and m memory locations, where each processor is a random-access machine (see Figure 1). All processors share the memory, and hence communicate via it. 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 [1978], though related early models are described in Schwartz [1980] and Goldschlager [1982].

The above description leaves some ambiguity still regarding the behavior 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:

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

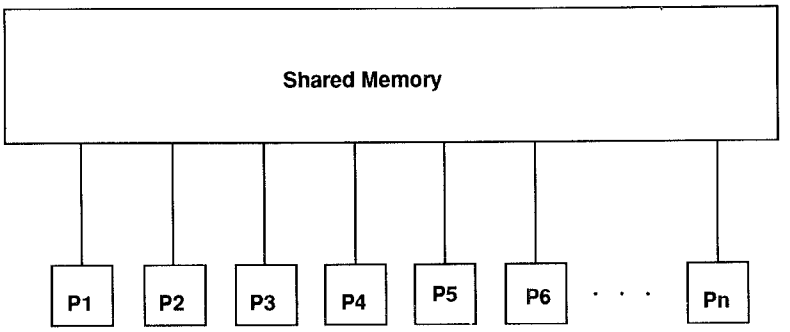

Figure 1. The PRAM model of computation.

may write to a particular memory location.

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

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

We need to specify also 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:

- 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.
- The ARBITRARY CRCW PRAM, where the processor that succeeds in its concurrent write is chosen arbitrarily from the writing processors.
- 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.
- 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], and 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 multiported memory which is shared by a large number of processors is infeasible. Instead, a realistic and scalable parallel computer consists typically 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 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 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 a 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 consists therefore 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 nontechnical 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, we have chosen to break the problem into three disjoint phases, each of which is a simulation in itself. The reason for solving 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 subproblems. In either case, addressing the subproblems independently plays an important role in providing an intuition as 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 location, 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. This problem has a well-known optimal solution which is described in later sections.

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

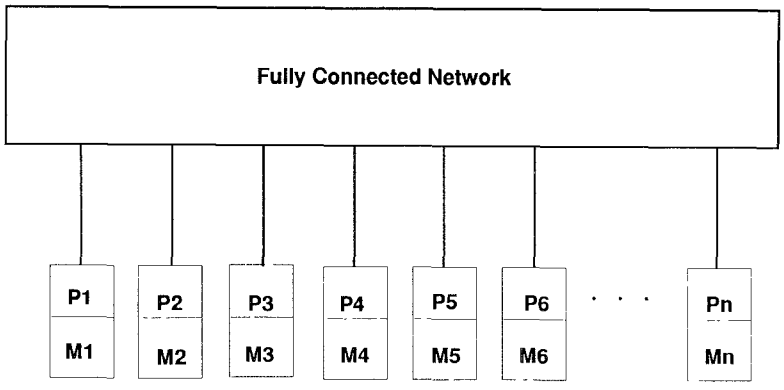

Figure 2. The module parallel computer.

management scheme may have manipulated the memory requests already before the router takes control.

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 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 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 4.

The quality of a simulation is determined primarily by the *slowdown* of the simulation, i.e., if a PRAM program requires T steps and if it can be executed on a MPC with the same number of processors in $O(Tf(n))$ steps, then the slowdown is $f(n)$. 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 require also an increase in the amount of memory utilized, referred to as *memory-blowup*, and this will be a factor also in assessing the quality of a simulation. If a PRAM requires memory M to execute a program, and an MPC requires $O(Mg(n))$ memory to execute the same program, then the memory blowup is $g(n)$.

One additional metric of the quality of a simulation is the efficiency. A simulation's efficiency is the ratio of

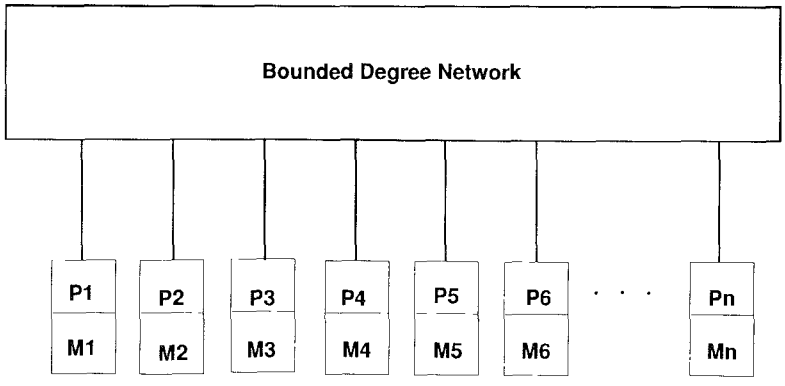

Figure 3. The bounded-degree network model.

Figure 4. PRAM simulation problem decomposition.

time-processor products for different levels of the simulation, i.e., if a PRAM program takes time T on n processors, and a simulation requires time T' on n' processors, then the efficiency of the simulation is $E = Tn/T'n'$. Simulations where E is a constant independent of n and n' are often referred to as *optimally efficient* or *constant time-processor product* simulations, and will be discussed later. Except where we consider efficiency, we will assume that the simulating and simulated machines both have the same number of processors.

In Section 1 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, of simulating an EREW PRAM on an MPC, and subsequently of simulating an MPC on a BDN. Section 2 will describe deterministic solutions to these two problems and will consider the goal of reducing the amount of replicated memory necessary while reducing contention for memory modules. Section 3 will discuss the analogous randomized solutions, namely, uniform hashing and two-phase routing. Section 4 will consider the special case of optimally efficient simulations. These simulations are randomized simulations, but are described in a separate section since the goals of the simulation are maximum efficiency, rather than minimum slowdown as in Sections 2 and 3. In Section 5 we summarize the results of the survey and highlight open issues. The primary goal is to provide a concise summary of the diverse arguments which address this problem, meanwhile developing an intuition about the techniques which may be difficult to establish from the highly technical nature of the original references.

1. CONCURRENT ACCESS

The ability to access a memory location concurrently is a powerful one, and it 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, e.g., Shiloach and Vishkin [1981], Kucera [1982], and Akl [1989a]. In order to isolate our simulation from this variety, we show that a simple strategy can allow simulation of all CRCW PRAMs on an EREW model with optimal slowdown. Such arguments have been used before 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. At the beginning of a cycle, we assume 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 process the requests as if they were arranged in a binary tree. First, all even-numbered processors P_i will compare requests i and $i + 1$. If the destination addresses are the same, then the processor will use the appropriate conflict resolution strategy to eliminate one of the requests. This elimination may correspond to performing an arithmetic operation in the case of a COMBINING strategy, or simply a validation in the case of the COMMON strategy. Next, each fourth processor P_i compares requests i and $i + 2$ and resolves conflicts between concurrent writes, and so on, until after $O(\log n)$ steps any concurrent accesses have been eliminated. At this point the memory requests may take place as normal on the EREW model. If the operation is a read, then a multibroadcast 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]. A related simulation can be seen in Vishkin [1984]. Thus we have:

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

2. DETERMINISTIC SIMULATIONS

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

2.1 Memory Management

As mentioned above, the trivial solution to the simulation of an EREW PRAM on an MPC results in a worst-case time of $\Omega(n)$. Initially, this discouraged consideration of deterministic solutions. First, Mehlhorn and Vishkin [1984] proposed the use of multiple copies of each memory location to solve the memory management problem. 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 $O(n)$. Their work 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 [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 timestamp. 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. Then, the reading processors will check the timestamp 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 ensured 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 5). In order to fulfil 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 ith process of each cluster will know the location of the ith 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 fulfil, and then every processor in the cluster will try to access a unique copy of that variable, i.e., the ith copy will be accessed by P_i. Some of these access attempts will be successful, 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 will be fulfilled then 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 timestamps.

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

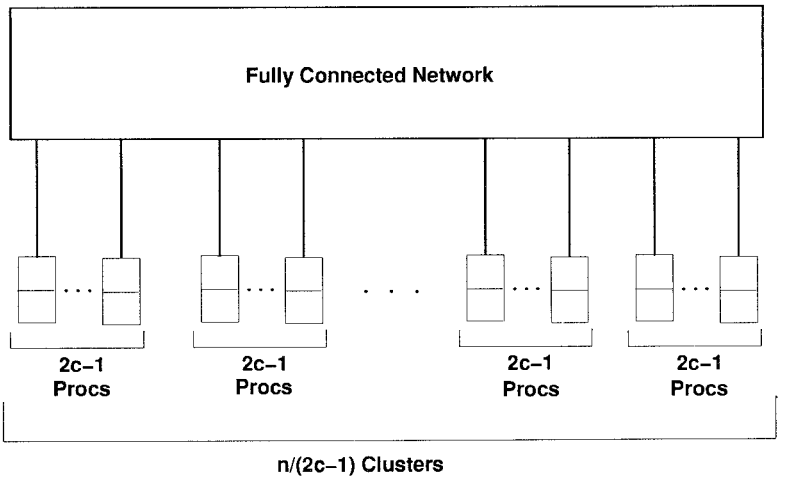

Figure 5. Majority scheme for deterministic-memory management.

PRAM on an 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 timestamps. The amount of memory used by the timestamps 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 et al. 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 described above are nonconstructive, 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 are questionable.

Herley and Bilardi [1988] 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 [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.

2.2 Routing and Interconnection

Naive deterministic solutions to the problem of simulating an MPC on a BDN will typically have good average-case behavior, but poor worst-case behavior. 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 [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 et al. [1990] into the following:

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

A permutation occurs when each of n processors holds a request for a distinct memory location. This worst-case behavior 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 the simulation of 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 mod-

ule 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-Oh" notation that render them unrealistic to build.

One of the most practical sorting networks 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. [1983], and further improved by 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 [1989], 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 support routing also in the optimal $O(\log n)$ time. 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 been shown also within the context of nonblocking 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. Such expander arguments also apply to the nonconstructive proofs of memory management schemes. For more background on expanders see Alon [1986] and Paterson [1987].

2.3 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. First, 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 in Section 1, 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 that 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 have 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 simply combined the solutions of subproblems we have 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

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

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.

Earlier, we described how Herley and Bilardi [1988] (Theorem 2) 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 an 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 BDN 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 $O(\log^2 n/\log\log n)$.*

3. 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.

3.1 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 a 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 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, first we 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 $1 \le a \le m$ will be stored in physical address $h(a)$, where $1 \le h(a) \le n$. This function $h(a)$ will generate the physical memory module that holds loca-

tion a, as well as the address within that module. 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 insuring 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 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 2. 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 $\text{Prob}_{h \in H}[h(x_1) = y \wedge h(x_2) = y] = 1/|B|^2$.

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

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

This is a generalization of Definition 2, and similarly implies that the chances that s memory locations from the logical space will all be mapped onto the same memory module is μn^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.

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

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 in 1986 to allow simulation of a PRAM on an 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 an 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 $h(x) = (a_1 x + b) \bmod p$ can give

reasonable performance [Ranade et al. 1988]. Further hashing results in the context of optimally efficient simulations will be discussed in Section 4. Also see Mansour et al. [1990], Luccio et al. [1991], and Matias and Vishkin [1991] for recent work on the subject.

3.2 Routing and Interconnection

We have seen in Section 2 that worst-case behavior for deterministic routing of permutations may require $\Omega(n^{1/2})$ time. However, we know also 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 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, since it appears to double the distance any packet needs to travel. However, Valiant [1983] has further shown that packets must travel at least twice the diameter in such oblivious routing algorithms, and hence this scheme is optimal.

Valiant originally described these techniques in terms of interconnection networks with logarithmic degree, such as the hypercube, making them indirectly 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 [1984b] 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. Such randomized routing techniques have since been used routinely [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 [1991] gave a straightforward algorithm with similar characteristics, $O(\log n)$ time and $O(1)$ length buffers, which had no such deadlock problems. Ranade's paper has been particularly 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].

3.3 Composition of Subproblems

In order to simulate a CRCW PRAM on a BDN we may use the same deterministic preprocessing suggested previously to eliminate concurrent access, again assuming our BDN is connected so 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.*

4. OPTIMALLY EFFICIENT SIMULATIONS

Until this point we have determined the quality of a simulation by the slowdown it incurred. Now, we 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 n' processors such that:

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

Such simulations are typically referred to as either constant time-processor product or optimally efficient. One trivial optimally 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. Optimally efficient simulations are those that require no more steps than does this trivial solution, despite the fact that memory access in parallel solutions requires $\Omega(\log n)$ 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 try to build an intuition about such simulations, and then describe specific results.

One may refer to the number n as the number of processes or threads in the PRAM, while n' is the number of processors used in the machine upon which the simulation is taking place. The ratio n/n' is called the 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 $n/n' > 1$, 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 6). If each processor uses a simple round-robin scheduling strategy and if our simulation has slowdown L, then using L threads on each processor will allow an optimally efficient simulation. Such techniques are common now in practical research of parallel computing where they are referred to as *Multithreading* [Weber and Gupta 1989].

As the slowdown caused by the latency of the simulating network increases, then a larger degree of parallel slackness may allow an optimally 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 optimally efficient simulations cannot take place on fixed-degree networks, since these networks do not have the bandwidth necessary to insure that no contention will take place [Kruskal et al. 1990]. Therefore it is only

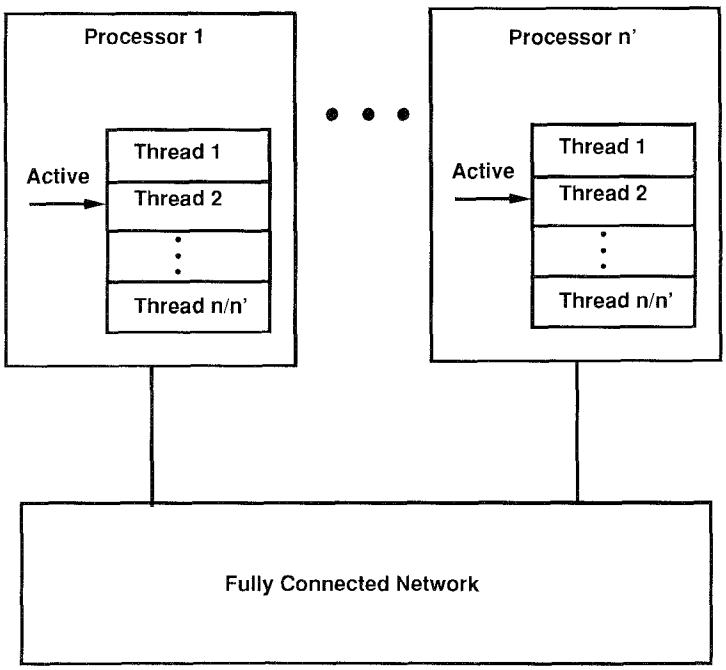

Figure 6. A multithreaded architecture.

the memory management issue which comes into such simulations, and these simulations will be based on fully connected networks. As we have seen above, 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 Section 3, 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 Section 3, will not be useful for our purposes here, because 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 optimally efficient simulations, we need different classes of hash functions, where the evaluation time is $O(1)$. Early work in the pipelining of PRAM simulations can be seen in Vishkin [1984] and Mehlhorn and Vishkin [1984].

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.

$$H_2 = \Big\{ h | h(x) = \left(\left(\sum_{i=1}^{d} a_i x^i \right) \bmod p \right) \bmod n \Big\}$$

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

Hashing functions of the class H_2 were used in the optimally efficient simulation of Kruskal et al. [1990]. They show that a PRAM with $O(n^{1+\epsilon})$ threads, where $\epsilon > 0$, may be simulated with optimal effi-

ciency on an MPC with n processors and $O(n^{\epsilon})$ parallel slackness.

In Siegel [1989], tradeoffs were established between the time necessary to compute a hash function and the number of random bits needed for the computation. 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. Siegel also showed straightforward techniques for extending the $O(\log n)$ slowdown simulation of Ranade [1991] to an optimally efficient simulation through the addition of parallel slackness.

The hash functions of Siegel [1989] were then used by Valiant [1990b] to derive an optimally efficient simulation with expected delay $O(\log n)$ time. 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 it does not strictly qualify as either a fully connected graph or a fixed-degree network. Valiant [1990a] also extended the simulation to other networks with less bandwidth by restricting the PRAM model simulated to one where memory accesses are by necessity followed by a number of local operations, and hence where demands on the network are reduced.

Similarly, an efficient simulation was shown 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 then the other determines the offset of the key within the computed bucket. We call this new class H_3.

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

where $f \in H_2^r$ and $g_1, \ldots, g_r \in H_2^n$, 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 optimally 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 with optimal efficiency and slowdown $O(\log \log n \log^* n)$ with high probability.*

The authors showed also 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.

5. CONCLUSIONS

We have addressed the problem of PRAM simulation as a series of disjoint subproblems with both deterministic and, in some cases, randomized solutions. The best known lower bounds for all phases of PRAM simulation are shown in Figures 7 and 8 and correspond to the results outlined in an earlier section of this survey. The randomized simulation of CRCW PRAMs on bounded-degree networks may be considered to have an optimal solution, but many open problems

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

Figure 7. Upper bounds of deterministic simulations.

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 8. Upper bounds of randomized simulations.

still exist in other areas, some of which we mention now.

PRAM simulations tend to depend heavily on expander graphs for optimal results. Examples include deterministic $O(\log n)$ routing networks, deterministic memory management schemes, and the *weak-concentrator* hashing results outlined above. The well-known difficulties in constructing such graphs are unfortunate, and progress on such constructions would have important implications for the practicality of many PRAM simulation techniques. Some interesting work in this area can be seen in Leighton [1989].

Perhaps the most active area currently is in optimally efficient simulations such as those described in Section 4. No deterministic such simulations are yet known. The use of more than one hash function, as in Karp et al. [1992], also shows promise.

ACKNOWLEDGMENT

The author is indebted to Murray Cole and Todd Heywood for comments on earlier drafts of this paper.

REFERENCES

AGGARWAL, A., CHANDRA, A., AND SNIR, M. 1990. Communication complexity of PRAMs *Theor. Comput. Sci. 71*, 1, 3–28

AJTAI, M., KOMLOS, J., AND SZEMEREDI, E. 1983 Sorting in C log n parallel steps. *Combinatorica 6*, 2, 83–96.

AKL, S. G. 1989a. On the power of concurrent memory access. In *Computing and Information*, R. Janickyi and W. W. Koczkodaj, Eds. Elsevier Science Publishers, New York, 49–55.

AKL, S. G. 1989b *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, N.J.

ALELIUNAS, R. 1982 Randomized parallel communication. In the *1st Annual Symposium on Principles of Distributed Computing.* ACM, New York, 60–72

ALT, H., HAGERUP, T., MEHLHORN, K., AND PREPARATA, F. P. 1987. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM J. Comput. 16*, 5, 808–835.

ALON, N. 1986. Eigenvalues and expanders *Combinatorica 6*, 2, 83–96.

ARORA, S., LEIGHTON, T., AND MAGGS, B. 1990 On-line algorithms for path selection in a nonblocking network. In *Proceedings of the 22nd ACM Symposium on Theory of Computing.* ACM, New York, 149–158.

AUMANN, Y., AND SHUSTER, A. 1991. Improved memory utilization in deterministic PRAM simulation *J. Parall Distrib. Comput. 12*, 2, 146–151

BATCHER, K E 1968. Sorting networks and their applications. In *Proceeding of the AFIPS Spring Joint Computer Conference* AFIPS, Washington, D. C., 307–314.

BORODIN, A., AND HOPCROFT, J. E. 1982. Routing, merging and sorting on parallel models of computation, In *Proceedings of the 14th ACM Symposium on Theory of Computing.* ACM, New York.

CARTER, J. L., AND WEGMAN, M. N. 1979. Universal classes of hash functions. *J. Comput. Syst Sci 18*, 2, 143–154.

COOK, A 1984. A taxonomy of problems with fast parallel algorithms. In *Proceedings of the 1983 International FCT Conference.*

COLE, R. 1988. Parallel merge sort *SIAM J Comput. 17*, 4, 770–784.

DIETZFELBINGER, M. AND MEYER AUF DER HEIDE, F. 1990. How to distribute a dictionary in a complete network. In *Proceedings of the 22nd Symposium on Theory of Computing.* ACM, New York.

FORTUNE, S. AND WYLLIE, J. 1978. Parllelism in random access machines. In *Proceedings of the 10th Symposium on Theory of Computing.* ACM, New York, 114–118

GIBBONS, A. AND RYTTER, W. 1988. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, England.

GOLDSCHLAGER, L. M. 1982. A unified approach to models of synchronous parallel machines. *J. ACM 29*, 4, 1073–1086.

HERLEY, K. T. 1989. Efficient simulations of small shared memories on bounded degree networks. In *Proceedings of the 30th Symposium on Foundations of Computer Science*. 390–395.

HERLEY, K. T. AND BILARDI, G. 1988. Deterministic simulations of PRAMs on bounded degree networks In *Proceedings of the 26th Allerton Conference on Communication, Control and Computation*. IEEE, New York

HORNICK, S. W. AND PREPARATA, F. P. 1991. Deterministic P-RAM simulation with constant redundancy. *Inf. Comput. 92*, 1, 81–96.

KAKLAMANIS, C., KRIZANC, D., AND TSANTILAS, T. 1990. Tight bounds for oblivious routing in the hypercube. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York, 31–36.

KARLIN, A. R. AND UPFAL, E. 1986. Parallel hashing—An efficient implementation of shared memory. In *Proceedings of the 18th Symposium on Theory of Computing*. ACM, New York, 160–168.

KARP, R. M. AND RAMACHANDRAN, V. 1990. Parallel algorithms for shared memory machines. In *Handbook of Theoretical Computer Science*. Elsevier Science. New York.

KARP, R. M., LUBY, M., AND MEYER AUF DER HEIDE, F. 1992. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th Symposium on the Theory of Computing*. ACM, New York.

KRIZANC, D. 1991. Oblivious routing with limited buffer capacity. *J. Comput. Syst. Sci. 43*, 2, 317–327.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. 1990. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci. 71*, 1, 95–132.

KUCERA, L. 1982. Parallel computation and conflicts in memory access. *Inf. Process. Lett. 14*, 2, 93–96.

LEIGHTON, F. T. 1989. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proceedings of the 30th Symposium on Foundations of Computer Science*. 384–389.

LEIGHTON, F. T. 1985. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput. C-34*, 4, 344–353.

LUCCIO, F., PEITRACAPRINA, A., AND PUCCI, G. 1991. Analysis of parallel uniform hashing. *Inf. Process. Lett. 37*, 1, 67–69.

LUCCIO, F., PEITRACAPRINA, A., AND PUCCI, G. 1990. A new scheme for the deterministic simulation of PRAMs in VLSI. *Algorithmica 5*, 529–536.

LUCCIO, F., PEITRACAPRINA, A., AND PUCCI, G. 1988. A probabilistic simulation of PRAMs on a bounded degree network. *Inf. Process. Lett. 28*, 3, 141–147.

MANSOUR, Y., NISAN, N., AND TIWARI, P. 1990. The computational complexity of universal hashing. In *Proceedings of the 22nd Symposium on Theory of Computing*. ACM, New York, 160–168.

MATIAS, Y. AND VISHKIN, U. 1991. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proceedings of the 23rd Symposium on Theory of Computing*. ACM, New York, 307–316.

MCCOLL, W. F. 1992. General purpose parallel computing. In *Proceedings of the 1991 ALCOM Spring School on Parallel Computation*, Gibbons and Spirakis, Eds. Cambridge University Press, Cambridge, England.

MEHLHORN, K. AND VISHKIN, U. 1984. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica 21*, 4, 339–374.

PATERSON, M. S. 1987. Improved sorting networks with $O(\log N)$ depth. Tech. Rep. RR 89, Univ. of Warwick, Warwick, England.

PFISTER, G. F. AND NORTON, V. A. 1985. "Hot Spot" contention and combining in multistage interconnection networks. *IEEE Trans. Comput. C-34*, 10.

PIPPENGER, N. 1984. Parallel communication with limited buffers. In the *25th Symposium on Foundations of Computer Science*. 127–136.

RABIN, M. O. 1989. Efficient dispersal of information for security, load balancing and fault tolerance. *J. ACM 36*, 2, 335–348.

RANADE, A. C. 1991. How to emulate shared memory. *J. Comput. Syst. Sci. 42*, 3, 307–326.

RANADE, A. C., BHATT, S. N., AND JOHNSSON, S. L. 1988. The fluent abstract machine. In the *5th MIT Conference on Advanced Research in VLSI*. MIT, Cambridge, Mass., 71–94.

SANZ, J. L. C., ED. 1988. Opportunities and constraints of parallel computing. IBM Workshop, Almaden Research Center, San Jose, Calif.

SCHWARTZ, J. T. 1980. Ultracomputers. *ACM Trans. Program. Lang. Syst. 2, 4*, 484–521

SHILOACH, Y. AND VISHKIN, U. 1981. Finding the maximum, merging, and sorting in a parallel computation model. *J. ACM 2*, 1, 88–102.

SIEGEL, A. 1989. On universal classes of fast high performance hash functions, their time-space tradeoff, and their application. In *Proceedings of the 30th Symposium on Foundations of Computer Science*. 20–24.

THOMAS, R. H. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 1, 180–209.

UPFAL, E. 1989. An $O(\log n)$ deterministic packet routing scheme. In *Proceedings of the 21st Symposium on Theory of Computing*. ACM, New York, 241–250.

UPFAL, E. 1984a. A probabilistic relation between desirable and feasible models of parallel computation. In *Proceedings of the 16th Symposium on the Theory of Computing*. ACM, New York, 258–265.

UPFAL, E. 1984b. Efficient schemes for parallel communication. *J. ACM 31*, 3, 507–517.

UPFAL, E. AND WIGDERSON, A. 1987. How to share memory in a distributed system *J. ACM 34*, 1, 116–127.

VALIANT, L. G 1990a. General purpose parallel architectures. In *Handbook of Theoretical Computer Science*. Elsevier Science, New York.

VALIANT, L. G. 1990b A bridging model for parallel computation *Commun ACM 33*, 2, 103–111.

VALIANT, L. G. 1983. Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Trans Comput. C-32*, 8, 861–863,

VALIANT, L. G. 1982. A scheme for fast parallel communication. *SIAM J. Comput. 11*, 2, 350–361

VALIANT, L. G. AND BREBNER, G. J 1981. Universal schemes for parallel communication. In the *Symposium on Theory of Computing* ACM, New York, 263–277

VISHKIN, U. 1984. A parallel-design distributed-implementation (PDDI) general-purpose computer *Theor. Comput. Sci. 32*, 2, 157–172.

VISHKIN, U. 1982 Implementation of simultaneous memory address access in models that forbid it *J. Alg. 4*, 1, 45–50.

WEBER, W AND GUPTA, A. 1989. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture· Preliminary results In *Proceedings of the 16th International Symposium on Computer Architecture* ACM, New York.

Received October 1992, final revision accepted December 1993