Hashing strategies for simulating shared memory on distributed memory machines

Friedhelm Meyer auf der Heide*

Heinz Nixdorf Institute and
Computer Science Department
University of Paderborn
4790 Paderborn, Germany

Abstract. We survey shared memory simulations on distributed memory machines (DMMs), that use universal hashing to distribute the shared memory cells over the memory modules of the DMM. We measure their quality in terms of delay, time-processor efficiency, memory contention (how many requests have to be satisfied by one memory module per simulated step) and simplicity. Further we take into consideration different access conflict rules to the modules of the DMM, in particular the c-Collision rule motivated by the idea of communicating between processors and modules using an optical crossbar.

It turns out that simulations with very small delay require more than one hash function. Further, simple simulations on DMMs with the c-Collision rule are only known if more than one hash function is allowed.

1 Introduction

Parallel machines that communicate via a shared memory, so called *parallel random access machines* (PRAMs) represent the most powerful parallel computation model considered in the theory of parallel computation. Further, it is relatively comfortable to program, because the programmer does not have to specify interprocessor communication, or to allocate storage in a distributed memory; rather she can even use common data structures, stored in the shared memory.

On the other hand, PRAMs are very unrealistic from the technological point of view; large machines with shared memory can only be built at the cost of very slow shared memory access. A more realistic model is the *distributed memory machine* (DMM), where the memory is partitioned in modules, one per processor. In this case a parallel memory access is restricted in so far that only one access to each module can be performed per parallel step. Thus *memory contention* occurs if a PRAM algorithm is run on a DMM; parallel accesses to cells stored in one module are sequentialized.

Therefore many authors have investigated methods to simulate PRAMs on DMMs. Often it is assumed that processors and modules are connected by a bounded degree

* Supported in part by DFG-Forschergruppe "Effiziente Nutzung massiv paralleler Systeme, Teilprojekt 4", and by the Esprit Basic Research Action Nr. 7141 (ALCOM II).

network, and packet routing is used to access the modules. (See e. g. ([11], [14], [14], [9], for a survey on packet routing see [15], [16]).

In this survey we focus on DMMs with a complete interconnection between processors and modules.

The most promising approaches are based on hashing: One or more hash functions, randomly drawn from a suitable universal class, are used to distribute the shared memory cells (we shall say "keys" for short) among the modules.

If one hash function h is used, the *delay* of the simulation, i. e. the time needed to simulate one PRAM step is governed by

- The *evaluation time* of h.
- The *memory contention*, i. e. the maximum number of memory accesses of a PRAM step that are mapped to the same module under h.
- The *quality of the access schedule.* If we want to benefit from the effect of parallel slackness, i. e. if we simulate a large PRAM on a smaller DMM, or if we have restricted access conflict resolution rules at the modules (e. g. as motivated by a realization of the communication among processors and modules via an "optimal crossbar", see below), we need a protocol that specifies how (e. g. when, by whom, in which order) the requests are sent to the modules. The time needed for the access is clearly bounded from below by the memory contention; the aim is to come close to this bound. Further, it is desirable that such schedules are *simple*, i. e. do not make complicated computations to decide which request to try to satisfy next, and do not distribute the requests among the processors before sending them to the modules.

In Chapter 5 we shall get to know several simulations that use two or three hash functions, i. e. that store each shared memory cell in two or three modules. It turns out that it is not necessary to access all copies of a requested cell in order to obtain a simulation. In this case, memory contention is constant with high probability, if an "ideal" access schedule can be found which specifies, for each keys to be accessed, which copies to access. Thus, in this case the *access schedule* is of particular importance.

The rest of the paper is organized as follows: In Chapter 2 we define the computational models and discuss several criteria for measuring the quality of simulations. In Chapter 3 we briefly sketch results on universal hashing.

In Chapter 4 we survey the simulations using one hash function, in Chapter 5 those using two or three hash functions.

In this paper we restrict ourselves to describing simulations, we do not give any hint towards the (in most case complicated) proof techniques used for proving the delay bounds.

2 Computation models and criteria for the quality of simulations

A parallel random access machine (PRAM) consists of processors $P_1, \ldots, P_m$ and a shared memory with cells $U = \{1, \ldots, p\}$, each capable of storing one integer. The processors work synchronously and have random access to the shared memory cells.

We distinguish PRAM models according to their capabilities of handling concurrent accesses to the same shared memory cell. We distinguish between the following rules:

- *exclusive read (ER)*: concurrent reading to the same shared memory cell forbidden
- *concurrent read (CR)*: concurrent reading allowed
- *exclusive write (EW)*: concurrent writing forbidden
- *concurrent write (CW)*: concurrent writing allowed.

In case of concurrent write we have to specify the semantics of a concurrent write access to a shared memory cell. There are many rules of resolving such write conflicts considered in literature.

Write conflict resolution rules: The result of the attempt of processors $P_{i_1}, \ldots, P_{i_s}, i_1 < \ldots < i_s, s > 1$, to write concurrently $x_1, \ldots, x_s$ to cell j is, for example, as follows.

Tolerant: cell j remains unchanged
Arbitrary: cell j contains any of $x_1, \ldots, x_s$, all of these choices have to lead to a correct result of the algorithm
Priority: cell j contains x_1
Minimum: cell j contains $\min\{x_1, \ldots, x_s\}$.

In [13], it is shown that all the above rules are almost identical, if concurrent read is allowed: an $n \log^*(n)$-processor CRCW-PRAM with the strongest rules, Minimum, can be simulated in a randomized fashion on a n-processor CRCW-PRAM with the weakest rule, Tolerant, such that the delay is $O(\log^*(n))$ with probability $1 - 2^{-n^\varepsilon}$, for some $\varepsilon > 0$.

In this paper we therefore only distinguish between exclusive-read exclusive-write PRAMs (EREW-PRAMs) and concurrent-read concurrent-write PRAMs (CRCW-PRAMs). It is convenient to assume the arbitrary rule in our considerations.

A *distributed memory machine* (DMM) consists of n processors $Q_1, \ldots, Q_n$ and n memory modules $M_1, \ldots, M_n$. Each module has a communication window where it can read from or write into. For the processors, these windows act like shared memory cells.

Again we distinguish DMM model with respect to how concurrent accesses at the communication windows are handled. It is easily checked that the result from [13] also implies that the computation powers of these CRCW-models are almost identical. In this paper we assume the arbitrary rule if we refer to CRCW-DMMs.

In case of DMMs we take into consideration a further rule for handling read/write collisions, which is motivated by the idea of using an optical crossbar to communicate between processors and (communication windows of) modules, compare [12], [17], [23]. Here a processor that wants to access module M directs a beam of light to it. If M only gets one message (i.e. only one beam is directed to its window), it acknowledges it, or, in case of read, sends back the requested data. If more than one processor sends a message to M, all of them get back a collision message.

We generalize this concept by assuming that a module can handle not only just one, but a constant number c of concurrent accesses. If at most c requests arrive, all of them are satisfied, otherwise, all issuing processors get a collision message. We refer to this model as a *c-Collision-DMM*.

In this paper we consider step by step simulations of PRAMs by DMMs. In all these simulations we use 1, 2, or 3 hash functions $h : U \rightarrow \{1, \ldots, n\}$ that specify in which module(s) each shared memory cell is maintained. We consider the following criteria for the quality of our simulations.

- **delay:** We want to simulate a PRAM fast on a DMM, i.e. we want that the delay, the (perhaps amortized) time needed to simulate one PRAM step, is small.
- **time-processor-efficiency:** If an m-processor PRAM is simulated on an n-processor DMM the smallest possible delay is $\frac{m}{n}$. We want to come close to this delay. If we achieve delay $O(\frac{m}{n})$ we talk about a *time-processor-optimal* simulation.
- **memory contention:** Assume that the PRAM processors access large blocks of data in a read or write request. In this case it is important to find access schedules that guarantee that each memory module only has to process few request, i.e. to keep memory contention as small as possible. This may even be of advantage if a (fast) precomputation for finding the schedule is necessary.
- **access conflict rules:** We aim to simulate a strong PRAM, i.e. a CRCW-PRAM on a weak DMM, i.e. a EREW-DMM or 1-Collision-DMM, or find simulations which come close to this ideal.
- **simplicity:** We want to have very simple simulations, in particular very simple access schedules. In particular, we prefer schedule in which processors that issue an access request, do not send it to another processor, but pass it to the module(s) itself.

3 Universal Hashing

Let $U = \{1, \ldots, p\}$ be the shared memory cells of the PRAM. In all simulations they are distributed among the modules of the DMM using one or more hash functions $h : \{1, \ldots, p\} \rightarrow \{1, \ldots n\}$, randomly drawn from a *universal class of hash functions.*

The analyses of the simulations require high performance universal classes, a randomly chosen function of which has properties very much like a random function. On the other hand they have to be generated fast using little space, and have to be evaluated in constant time, at least if time-processor optimality is desired. Suitable classes are introduced in [10] and in [3], [4]. In [19] a combination of the above classes is introduced. This is necessary for all simulations presented in this paper which use more then one hash function.

We do not go into details about hashing in this survey, and refer the reader to the above papers and to [2], [8] and [18] for information about polynomials as hash functions. To simplify understanding of the simulations below one should assume that the hash functions used are random functions.

4 Simulations using one hash function

Let $U = \{1, \ldots, p\}$ be the set of registers (or cells) of the shared memory of a PRAM, and let $M_1, \ldots, M_n$ be the memory modules of a DMM. Let $H_{n,p} \subseteq \{h : U \rightarrow \{1, \ldots, n\}\}$ be a high performance universal class of hash functions. In this chapter we assume that, for a randomly chosen $h \in H_{n,p}$, cell x is stored in $M_{h(x)}$.

4.1 Simulations of n-processor PRAM on n-processor DMMs using one hash function

Consider a PRAM with processors $P_1, \ldots, P_n$, to be simulated by a DMM with processors $Q_1, \ldots, Q_n$. Let $X = \{x_1, \ldots, x_n\} \subseteq U$. In a given PRAM step, P_i wants to access cell x_i. Q_i simulates P_i, i.e. has to simulate P_i's access to cell x_i, for $i = 1, \ldots, n$. It can be shown that the maximum bucket size of X under a randomly chosen $h \in H_{n,p}$ is $\Theta(\frac{\log(n)}{\log\log(n)})$ with high probability, i.e. that $\text{Prob}(\max_{1 \le i \le n}\{|h^{-1}(i) \cap X|\} = \Theta(\frac{\log(n)}{\log\log(n)}))$ is very large. In other words: with high probability each module is only accessed by at most $D\frac{\log(n)}{\log\log(n)}$ *different* requests. As $\frac{\log(n)}{\log\log(n)}$ is also a lower bound, $\Theta(\log(n)/\log\log(n))$ delay and memory contention is the best we can hope for.

n-processor CRCW-PRAM $\rightarrow$ n-processor CRCW-DMM. In this case the access schedule is very simple. (Recall that we assume an Arbitrary-rule at the modules.) In each round, each Q_i that was not yet successful tries to access $M_{h(x_i)}$. Each M_j answers one request. Q_i is successful if it gets an answer from $M_{h(x_i)}$, or if a Q_j with $x_i = x_j$ gets the answer.

Thus delay and contention equal the maximum bucket size $O(\frac{\log(n)}{\log\log(n)})$, and the schedule is very simple.

n-processor CRCW-PRAM $\rightarrow$ n-processor EREW-DMM. In this case we use the $O(\log(n))$-time sorting algorithm from [20] for sorting n numbers with an n-processor EREW-PRAM. As this algorithm uses space $O(n)$, it can be implemented on a n-processor EREW-DMM with constant delay.

The *access schedule* first sorts $(h(x_1), x_1), \ldots, (h(x_n), x_n)$ according to the lexicographic order. Now it is obvious how to schedule the requests such that no collisions happen.

This needs time $O(\log(n))$ with high probability, i.e. the delay is by a factor $O(\log\log(n))$ away from the maximum bucket size. The contention is still $O(\frac{\log(n)}{\log\log(n)})$, with high probability. The schedule is very complicated, requests are not passed to the modules by the issuing processors because of the sorting; answers have to be redistributed.

4.2 Optimal simulations allowing parallel slackness, using one hash function

Consider a PRAM with $m = n \cdot t$ processors $P_{i,j}, i = 1, \ldots, n, j = 1, \ldots t$, and a DMM with n processors $Q_1, \ldots, Q_n$. Let $X = \{x_{i,j}, i = 1, \ldots, n, j = 1, \ldots, t\}$. $P_{i,j}$ wants to access cell $x_{i,j}$, $P_{i,1}, \ldots, P_{i,t}$ are simulated in Q_i. It can be shown that the maximum bucket size $\max_{1 \le i \le n}\{|h^{-1}(i) \cap X|\}$ is best possible, i.e. $\Theta(t)$, with high probability, only for $t = \Omega(\log(n))$. Thus, with one hash function, time-processor optimal simulations must have delay $\Omega(\log(n))$.

$n^{1+\epsilon}$-processor CRCW-PRAM → n-processor EREW-DMM. The following algorithm is presented in [6]. It assumes that p is polynomial in n. In this case it is possible to base the simulation on fast integer sorting, using a similar idea as in Section 4.1.2.

Based on the randomized EREW-PRAM algorithm for integer sorting from [21], in [22] a randomized algorithm is presented for sorting $n^{1+\epsilon}$ keys from $[1, \ldots, n^k]$ on a n-processor EREW-DMM in time $O(n^\epsilon k) = O(n^\epsilon)$ if k is constant. Using this algorithm a similar schedule as in 2.1.2 leads to an access schedule, which implies a randomized simulation of a $n^{1+\epsilon}$-processor-CRCW-PRAM on an n-processor EREW-DMM.

The simulation is time-processor optimal, but the delay is very high. Further the access schedule is very complicated, again requests are not passed to the modules by the issuing processors, a redistribution of answers becomes necessary.

$n \log n$-processor CRCW-PRAM → n-processor CRCW-DMM. The following algorithm is presented in [3]. We assume that we (virtually) have n further processors $W_1, \ldots, W_n$, the working processors, available in the DMM. The W_i do the following in each round: they randomly choose a Q_i to ask it for a new request x. Each Q_i delivers a constant number of keys to asking working processors. A working processor tries to deliver its key x to $M_{h(x)}$. Each $M_{h(x)}$ returns a constant number of answers, which are transmitted from the working processor to the Q_i that issued the request.

This schedule can be shown to run in expected time $O(\log(n))$, i.e. achieves optimal expected delay, if an EREW-PRAM is simulated.

The same optimal expected delay can also be achieved for simulations of CRCW-PRAMs on CRCW-DMMs using a complicated algorithm that searches for concurrent accesses during the competition of duplicates of a key x to access $M_{h(x)}$.

This simulation reaches expected optimal delay $O(\log(n))$, i.e. is best possible in this respect, if only one hash function is used. On the other hand the schedule is still complicated; even if EREW-PRAMs are simulated, keys have to be distributed among the processors.

$n \log n$-processor EREW-PRAM → n-processor CRCW-DMM. The analysis of the following simple simulation is shown in [25].

The access schedule is very simple: The processors satisfy their requests in the given order. In each round, each processor tries to pass its currently processed request to the module. Each module answers one (arbitrary) of the incoming requests per round.

This schedule needs optimal delay $O(\log(n))$ with high probability, if an $n \log n$-processor EREW-PRAM is simulated on a CRCW-DMM. The importance of this schedule lies in its simplicity.

$n \log n$-processor EREW-PRAM → n-processor 1-Collision-DMM. This simulation is shown in [17] based on an access schedule from [23]. This schedule works not only if the destinations of keys $x, M_{h(x)}$, are random, but already if no module gets more than $c \log(n)$ messages, for arbitrary constant $c > 0$.

In a first phase, in each round, each Q_i randomly chooses one of its not yet satisfied requests. With a certain probability, it passes it to the corresponding module. The attempt is successful if the module gets no other request at the same time.

A schedule based on the above idea is designed in [23]. The expected number of not yet satisfied request after $O(\log(n))$ rounds is shown to be $O(n)$.

It is easy to finish up the remaining accesses in time $O(\log(n))$ using a parallel prefix algorithm to distribute the remaining keys evenly among the processors and to define the schedule.

This simulation is optimal, has asymtotically best possible expected delay $O(\log(n))$, and works already on 1-optimal DMMs. The access schedule is still very complicated, in particular, in the second phase accesses are not processed by the issuing processors, and a redistribution of answers to read requests become necessary.

5 Simulations using two or three hash functions

In this section we consider simulations using two or three hash functions h_1, h_2 or h_0, h_1, h_2, randomly, independently drawn from some high performance universal class $H_{n,p} \subseteq \{h : h : \{1,\ldots,p\} \to \{1,\ldots,n\}\}$. Each key $x \in U$ will be stored in $M_{h_1(x)}, M_{h_2(x)}$, and, in case of three hash functions, in $M_{h_0(x)}$. We refer to the representants of x in the $M_{h_i(x)}$'s as its copies.

In some simulations we further assume that few, i.e. $O(n)$ keys may be intermediately stored at further positions. This will be done by using a perfect hash table of size $O(n)$. In [1] and [7] it is shown how to implement such a table on a CRCW-PRAM using space $O(n)$ in time not exceeding $O(\log^*(n))$ with high probability. Because of the space bound it also can be implemented on a CRCW-DMM within the same time bound, as long as only $O(n)$ keys have to be stored in it.

5.1 Fast simulations of n-processor PRAMs on n-processor DMMs

Consider a PRAM, DMM and a set X as described in the beginning of Section 4.1. Already if two hash functions are used, there is an access schedule that needs constant time, if only one arbitrary copy of each $x \in X$ has to be accessed. To see this consider the bipartite graph with node set $X \cup \{1,\ldots,n\}$, where each $x \in X$ is connected to $h_1(x)$ and $h_2(x)$. If h_1, h_2 are random and independent, then this graph is random. It is well known that the nodes in X can be covered by a constant number of matchings, with high probability. Thus a schedule that processes one matching after the other needs constant time, even on an EREW-DMM. In particular, the memory contention of such a schedule is constant. The problem is to find such a schedule efficiently.

n processor CRCW-PRAM $\to$ n-processor CRCW-DMM. This simulation is presented in [19]. It uses two hash functions and has delay $O(\log\log(n))$ with high probability. We present a variant which only has constant memory contention.

We distinguish between write- and read steps. For writing, we maintain two perfect hash tables SM_1, SM_2 of size $O(n)$, each. In order to update the copies of $x_1,\ldots,x_n$ w. r. t. $h_1, \{x_1,\ldots,x_n\}$ is added to SM_1 using the perfect hashing

strategy mentioned above. Then, for each key x in SM_1 an attempt is made to update its copy in $M_{h_1(x)}$. Only constantly many of the updates directed to M_j are performed, and the corresponding keys are removed from SM_1. The time for such a step is governed by $O(\log^*(n))$, the time to set up a perfect hash table, as long as this table has size at most $c \cdot n$ for a suitable constant $c > 0$. It can be shown that this size bound is satisfied with high probability. The same procedure is executed w. r. t. h_2, using the hash table SM_2.

This write algorithm has delay $O(\log^*(n))$ with high probability.

The read algorithm is very simple. Note that the up-to-date value of x is stored in SM_1 and SM_2, if x is in one of these hash tables. Otherwise the copies of x in $M_{h_1(x)}$ and $M_{h_2(x)}$ are up-to-date. Thus, in the read algorithm, each Q_i inspects SM_1 and SM_2 to find the contents of cell x_i. This takes constant time by definition of perfect hashing. Each unsuccessful processor tries to access alternately the two copies of its key, until it gets an answer (no matter from which copy, both are up-to-date.) In each round, each module answers all requests, as long it only gets some constant c many. If it gets more it answers none of them.

Each of the above rounds takes constant time. It can be shown that $O(\log\log(n))$ rounds suffice to answer all requests, with high probability.

Thus we have got a simulation with delay not exceeding $O(\log\log(n))$ with high probability. The memory contention is only constant, and reading is very simple. On the other hand, writing is complicated because it uses perfect hash tables; again processors that process a request, i.e. pass the corresponding key to a module, are not those that issue the request, and redistribution of answers becomes necessary.

n-processor EREW-PRAM $\rightarrow$ n-processor c-Collision-DMM. This simulation is presented in [25].

The basic idea is borrowed from the deterministic simulation from [24]. We use three hash functions. In the write algorithm, arbitrary two of the three copies of each cell x_i are updated.

Thus, both for reading and writing, we need a schedule that accesses two arbitrary of the three copies of each x_i. This schedule is very simple.

Each round consists of three phases 0, 1, 2. In phase j, each Q_i tries to access the j'th copy of x_i. It skips an access to copy j, if it was earlier successful in accessing this copy. Q_i quits as soon as it gets two answers. Each module answers all request it gets in one phase, if it gets at most c requests. Otherwise it answers no request (this is the c-Collision rule).

It can be shown that this schedule finishes within $O(\log\log(n))$ rounds with high probability. Further it guarantees constant memory contention, is very simple, and runs on the weak c-Collision-DMM. The time bound $O(\log\log(n))$ already holds with reasonable probability for $c = 2$.

5.2 Optimal simulations allowing parallel slackness, using three hash functions

We use the notations introduced in the beginning of Section 4.2. Whereas, if only one hash function is used, the contention can be reduced to $O(\frac{m}{n})$ only if $m =$

$\Omega(n \log(n))$, this reduction is possible for all $m \geq n$ in case of two or more hash functions, as described in the beginning of Section 4.1. Again the problem is to find such schedule.

$n \log\log(n) \log^*(n)$-processor EREW-PRAM → n-processor CRCW-DMM. This simulation is shown in [19]. We do not go into details here. The simulation is based on the idea shown in Section 5.1.1. A simpler variant, also shown in [19] has delay not exceeding $O(\log\log(n) \log^*(n))$ with high probability, simulates even CRCW-PRAMs, but is by a factor $\log^*(n)$ away from time-processor optimality.

These simulations are very complicated, e. g. they still use perfect hashing.

$n \log(n)$-processor EREW-PRAM → n-processor c-Collision-DMM. This simulation is presented in [25].

It represents the simplest time-processor optimal simulation on a c-Collision-DMM.

The schedule is as in Section 5.1.2 with the extension that each processor starts processing its next key in the situation where it quits in Section 5.1.2. It now quits when all its accesses are satisfied.

This schedule has optimal delay $O(\log(n))$, with high probability. Further, it is very simple, and runs on the weak c-Collision-DMM. $c = 2$ is sufficient to make the delay bound reasonably reliable.

6 Conclusion

Simulations of PRAMs on DMMs that use more than one hash function have the disadvantage that they vaste storage (a factor 2 or 3) within the modules. (It is not required that the number of modules is enlarged!).

Nevertheless the simulations described in this paper show that the use of two or three hash functions has advantages that make them worth being considered more carefully, both theoretically and experimentally.

- The memory contention can be made constant.
- Simulations on c-Collision-DMMs (motivated by a communication technology using optical crossbars) become very simple, i. e. feasible for efficient implementation.
- Simulation with very small delay can be designed.

References

1. H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *Proc. of the 3rd Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 50–61, 1991.
2. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18:143–154, 1979.
3. M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 117–127, 1990.

4. M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In M. S. Paterson, editor, *Proceedings of 17th ICALP*, pages 6–19. Springer, 1990. Lecture Notes in Computer Science 443.
5. A. Karlin and E. Upfal. Parallel hashing — an efficient implementation of shared memory. In *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, pages 160–168, 1986.
6. C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoret. Comput. Sci.*, 71:95–132, 1990.
7. Y. Matias and U. Vishkin. Converting high probability into nearly-constant time - with applications to parallel hashing. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, pages 307–316, 1991.
8. K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.
9. A. G. Ranade. How to emulate shared memory. In *Proc. of the 28th IEEE Ann. Symp. on Foundations of Computer Science*, pages 185–194, 1987.
10. A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proc. of the 30th IEEE Ann. Symp. on Foundations of Computer Science*, pages 20–25, 1989. *Revised Version.*
11. E. Upfal. Efficient schemes for parallel communication. *J. Assoc. Comput. Mach.*, 31(3):507–517, 1984.
12. L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, chapter 18, pages 943–971. Elsevier, Amsterdam, 1990.
13. T. Hagerup. The log-star revolution, proceedings. *STACS 92, LNCS 577*, pages 259–280, 1992.
14. A. Karlin, E. Upfal. Parallel Hashing, an efficient implementation of shared memory. *Proc. 18th ACM STOC*, pages 160–168, 1986.
15. F. T. Leighton. Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. *Morgan Kaufmann Publishers*, San Mateo, 1992.
16. F. T. Leighton. Methods for packet routing in parallel machines. *Proc. 24th ACM STOC*, pages 77–96, 1992.
17. L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), pages 103–111, 1990.
18. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R. E. Tarjan. Dynamic perfect hashing, upper and lower bounds. *Proc. 29th IEEE FOCS*, pages 524–531, 1988, extended version appears in SIAM J. Comp.
19. R. Karp, M. Luby, F. Meyer auf der Heide. Efficient PRAM simulation on distributed memory machine. *Proc. 24th ACM STOC*, pages 318–326, 1992.
20. R. Cole. Parallel merge sort. *SIAM J. Comp.* 17(4), pages 770–785, 1988.
21. J. Reif. An optimal parallel algorithm for integer sorting. *Proc. 26th IEEE-FOCS*, pages 496–504, 1985.
22. R. A. Wagner, Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. *Proc. Int. Conf. on Parallel Processing*, Illinois, pages 924–930, 1986.
23. R. J. Anderson, G. L. Miller. Optical communication for pointer based algorithms. *Tech. Rep.* CRI 88–14, Comp. Sci. Dpt, Univ. of Southern California, Los Angeles, 1988.
24. E. Upfal, A. Wigderson. How to share memory in a distributed system. *J. ACM 34*, pages 116–127, 1987.
25. M. Dietzfelbinger, F. Meyer auf der Heide. Simple, efficient shared memory simulations, preprint 1992.