Static and Dynamic Data Management in Networks

Friedhelm Meyer auf der Heide and Berthold Vöcking*

Department of Mathematics and Computer Science
and Heinz Nixdorf Institute, University of Paderborn
33095 Paderborn, Germany

Abstract. We survey strategies for distributing shared objects in large parallel and distributed systems. Examples of such objects are global variables in a parallel program, pages or cache lines in a virtual shared memory system, shared files in a distributed file system, and videos and pictures in a distributed multimedia server. We focus on strategies for distributing, accessing, and (consistently) updating such objects. The strategies are provably efficient with respect to various cost measures. We describe and analyse static, hashing based schemes that minimize the contention at the memory modules in worst case scenarios. Especially, the benefit of redundant placement schemes is discussed. We further take network congestion and bandwidth into account. Here we present schemes that are provably efficient w.r.t. information about access frequencies. Further, dynamic schemes are presented which have good competitive ratio, i.e., are efficient compared to an optimal dynamic distribution that is constructed using full knowledge of the dynamic access pattern.

1 Introduction

One of the basic problems in large parallel and distributed systems is the data management. In this paper, we consider the problem of distributing and accessing shared objects in such an environment. The objects are, e.g., global variables in a parallel program, pages or cache lines in a virtual shared memory system, shared files in a distributed file system, or media information (video, audio, text, graphics) on a media server.

Most work concerning with data management in parallel and distributed systems investigates either hashing or caching based strategies. Hashing distributes the shared objects uniformly at random among the memory modules, which yields an even distribution of the data and therfore achieves a good load balance. However, uniform hashing gives up any locality in the pattern of read and write accesses. Caching exploits locality by placing or moving copies of the objects at or close to the accessing processors. The basic idea is that this minimizes

* email: {fmadh,voecking}@uni-paderborn.de. Supported in part by DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen", by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), and by DFG Leibniz Grant Me872/6-1.

the distances and therefore decreases the total communication load. The main problem is that minimizing distances can produce bottlenecks in the system, e.g., if many objects are placed on a central processor in the network.

In this paper, we discuss hashing and caching based data management strategies for two different system scenarios. Of course, efficient strategies must depend on the characteristics of the system. These characteristics are determined by several parameters, including the processor and memory speeds, the network topology, and the bandwidths and latencies of the buses and links. Usually, the memory banks and the the routing network are the bottlenecks of the system, because increasing their performance is much more expensive than increasing processor speed. Our scenarios should reflect this phenomenon.

Scenario 1: Here we deal with data distribution in a system in which the memory modules are the bottleneck. This scenario is typical, e.g., for data servers in which archive data is stored on hard discs or in other relatively slow memory modules. These modules are usually connected via a network, which provides sufficiently large bandwidth, to several external communication units, e.g., ATM or SCI ports. For instance, the network is an indirect high-bandwidth butterfly or banyan network. Here caching is of interest only on the user or client side, because there is no notion of locality between the inputs and the outputs of these networks.

The main problem is to distribute the data among the modules in such a way that each module has to satisfy nearly the same amount of requests. Usually, the data objects are partioned into data packets of fixed size, e.g, 1 Kbyte. For simplicity, we assume that the time is divided into rounds of fixed length. Then the maximum number of requests that are directed to the same module in the same round is called the *contention*. Clearly, high contention means that the performance goes down. Thus, it is important to distribute the data packets such that the contention is as small as possible.

One of the most intuitive strategies to distribute the data packets among the modules is round robin, i.e., the packets are numbered in some order and the i-th packet is stored in module i modulo the number of modules. The advantage of this strategy is that it distributes the packets evenly among the modules. However, in some cases this yields high contention. A better strategy is hashing, i.e., a (pseudo-)random mapping of the packets to the memory modules. This also yields an even distribution among the modules. In particular, it can be shown that the expected contention is relatively small. Let n denote the number of modules, and m denote the number of requests in a round. Then the expected contention is $\Theta\left(\frac{m}{n} + \frac{\log n}{\log(1+\frac{n}{m}\cdot\log n)}\right)$, if all requests are directed to distinct objects. That means, if the slackness is sufficiently large, i.e, if $m \geq n \cdot \log n$, then we get optimal contention up to a constant factor. However, most data servers are required to give very fast responses, in particular, in realtime environments. In this case, slackness is not a good solution since it increases the response time.

In this paper, we describe a strategy which uses redundant placement. That means our strategy places the data packets in more than one module. For in-

stance, 2 copies are placed in randomly selected modules. We show that this redundant placement yields nearly optimal contention without requiring any slackness, e.g., redundancy 2 yields contention 2 for read-only servers.

Scenario 2: Here we assume that the routing mechanism is the system's bottleneck, i.e., we focus on data management in parallel processor systems in which the processors are connected by a relatively sparse network. Each processor is assumed to have its own local memory module such that shared objects have to be distributed among these modules. This scenario is typical for most of today's parallel computers, including the Parsytec GCel and GCpp, Intel Paragon, Fujitsu AP1000, and Cray T3D and T3E. The processors in all these systems are connected by mesh- or torus-networks. Clearly, the larger the number of processor in these systems, the more the communication bandwidth becomes the bottleneck, because the bisection width of these networks increases less quickly than the number of processors.

For this scenario, hashing yields an even distribution of the data among the processors and also an even distribution of the communication or routing load among the links in the network. Several hashing based strategies have been analyzed in the context of PRAM simulation. For instance, Ranade [17] describes a hashing based PRAM emulation for the direct butterfly network. He shows that an N processor PRAM can be emulated by an N processor butterfly network with slowdown $\Theta(\log N)$. This scheme can also be adapted to other networks, which, e.g., yields an N processor PRAM simulations for the $\sqrt{N} \times \sqrt{N}$ mesh with slowdown $\Theta(\sqrt{N})$. Note that slowdown $S \cdot \sqrt{N}$ is optimal for any PRAM simulation with slackness S on the two-dimensional mesh because of the small bisection width. That means that the above result cannot be improved for general PRAM simulations. Nevertheless, it is completely unsatisfying for applications including locality. This shows that the main drawback of uniform hashing is that it gives up any locality in the pattern of read and write accesses.

In order to get rid of the slowdown caused by the limited bandwidth, we have to minimize the communication load by exploiting locality. This can be done by minimizing the distances from the accessing processors to the accessed objects. This problem is widely studied in the context of file allocation and distributed paging, see, e.g., [1,12,2]. A survey on these topics is given in [3]. Clearly, minimizing the distances minimizes the total communication load. Unfortunately, it can also increase the congestion, i.e., the maximum number of data packets which have to cross the same link. The congestion describes the worst bottleneck of the system. Therefore, it gives a lower bound on the execution time of a given application. Moreover, several results on store-and-forward- and wormhole-routing (see e.g. [11,15,4,16]) indicate that this value is also a good approximation for an upper bound on the execution time of coarse grained applications with high communication load and low synchronization requirements. This shows the importance of considering the congestion rather than the total communication load.

For this scenario, we describe static and dynamic data management strategies for meshes. These strategies aim to minimize the congestion. The static strategy

maps the objects statically to the modules according to some knowledge of the access pattern of a given application. The dynamic strategy makes all placement decisions on-line, i.e., it has no knowledge about the access patterns beforehand. This strategy combines hashing and caching techniques in order to exploit locality without producing bottlenecks. We compare the congestion of our strategies with the congestion of an optimal strategy and show that it is close to optimal.

2 Scenario 1: Systems with Slow Memory Modules

In this section, we focus on data management for data servers consisting of relatively slow memory modules that are connected by a high-bandwidth network to ports, communication units or processors. Since the memory modules are assumed to be the slowest building blocks in the server, it is useful that their number is larger than the number of processors. Therefore, we assume that we have n processors and $a \cdot n$ memory modules, with $a \geq 2$ being a suitable integer. The $a \cdot n$ memory modules are denoted M_j^i for $1 \leq i \leq a$ and $1 \leq j \leq n$. This means there are a classes of memory modules each consisting of n modules. In a round, each processor wants to access one object. The set of shared data objects is denoted by X. These objects can be read and updated by the processors, and a read should always return the value of the last update. (Of course, it is also possible that the server has to handle only read requests.) In this paper, we do not care about the contention induced by accesses to the same object. In particular, we assume that either all accesses are directed to distinct objects, or that there is some mechanism that is able to combine and split accesses directed to the same object.

We give a data management strategy that aims to minimize the contention without using slackness. The strategy consists of a placement strategy and an access scheme. The placement strategy maps the objects to the modules. It uses redundancy, each object is replicated a times. The access scheme determines which request is satisfied by which module. We will show that our strategy achieves small contention, i.e., each module has to read or update at most a small, constant number of objects in each round. Further, we will show that the access scheme can be computed very efficiently. In particular, it can be computed on-line by the modules and the processors of the data server.

2.1 Description of the Strategy

The proposed strategy uses several techniques introduced in [9,10,14] in the context of PRAM simulations on so called c-collision DMMs. The placement strategy uses a many hash functions $h_1, \ldots, h_a$ randomly, independently drawn from some high performance universal class of hash functions $H \subseteq \{h \mid h : X \to \{1, \ldots, n\}\}$. We do not go into details about hash functions and refer the reader to, e.g., [20,18,8,10] for information about suitable classes of hash functions. To simplify understanding one should assume that these functions are

truly random functions. The i-th copy of an object $x \in X$ will be stored in $M^i_{h_i(x)}$, for $1 \leq i \leq a$.

We first explain, why even using just two copies implies the existence of access patterns with constant contention, if only one of the two copies of each desired object has to be accessed. Consider the bipartite *access graph* H connecting each of the n requested objects to the two modules holding its copies. As the hash functions are (pseudo-)random, H is a kind of random graph. It is easy to verify that, with high probability, H has a subgraph that contains one edge incident to each requested object, and has constant degree. This subgraph describes an access scheme with constant contention. A similar result is true for larger a and for the case that not just one but some number $b < a$ of copies of each requested object has to be accessed. Note that $a = 1$ cannot yield constant contention, but contention $\Theta(\log n / \log \log n)$.

The technique to maintain consistency, i.e., to ensure that a read returns always the value of the last update, is borrowed from [19]: the majority technique. This technique ensures that we are consistent even if we do not always update all copies. It is sufficient to consider always a majority. Consider an object x having a different copies in the memory modules. Let $b > a/2$. Then a processor that wants to read or update x always accesses b out of the a possible copies. If the processor wants to update x, it updates all b copies and adds a time-stamp indicating the number of the round. If it wants to read x, it reads one from the b copies with the latest time-stamp. It is easy to see that this technique ensures consistency. (Our strategy becomes a little bit easier if our data server has to handle only read accesses, then it is sufficient to access just one of the a copies.)

Finally, we have to describe how to compute a schedule that guarantees that each memory module has to access at most a constant number of copies in each round. This schedule can be computed efficiently by the processors and the modules. The computation is done in phases. Let c be a suitable constant. Consider a processor P that want to access object x. That means P wants to access b out of a copies of x. P sends out requests for all the a copies of x to the modules storing them. Each module that receives at most c requests answers with "yes" to each of these request, and each module that receives more than c requests does nothing. (This is the c-collision rule). There are two rules for skipping requests in a phase: Processor P skips a request to a copy of x if an earlier request of P to this copy has already been answered with "yes", and Processor P skips a request to a copy of x if it has already received b many "yes"-answers for other copies. If each processor got b "yes"-answers for its object, the computation stops. The desired schedule consists of every object accessing the b copies answered with "yes". This schedule has contention c.

The following theorem illustrates the efficiency of this simple strategy. It shows that the computation of the schedule requires only a very small number of phases, and in particular, it illustrates the dependencies between the redundancy a, the number of accessed copies b, and the contention c.

Theorem 1 ([14]). *Let* $2 \leq a \leq \sqrt{\log n}$, $b < a$ *and* $2 \leq c = O((\sqrt{\log n}/(a-b))^{1/3})$ *be chosen such that*

$$\binom{a-1}{a-b}\left(\frac{1}{c!}\right)^{a-b} \leq \frac{1}{2} \ .$$

Then, with high probability, the above process takes only $\log\log n/\log(c(a-b))+3$ *phases, and it produces a schedule with contention* c.

The theorem shows that the above scheme yields very small contention already with small redundancy. For instance, with redundancy $a = 3$ and $b = 2$ the scheme yields contention $c \leq 3$, and if the server has only to handle read accesses, i.e., b=1, then already redundancy $a = 2$ yields contention $c \leq 2$, with high probability. Moreover, simulations show that the computation of the schedule takes usually only 3 phases, for $a = 3$, $b = 2$, and $n = 1,000,000$.

In order to get an idea of why a small number of phases suffice, let us have a closer look to the access graph H. (We again assume $a = 2$ and $b = 1$. Further, suppose that only a constant fraction of the processors issues a request.) It is shown in [10] that H consists of connected components of only logarithmic size, each being almost a tree. Thus, the above computation of the schedule in fact consists of independent computations on the small trees, which can be shown to only need $O(\log(\text{maximum size of the tree})) = O(\log\log n)$ phases.

More dedicated analyses of the access graph can be found in [6,5,7]. Using very complicated algorithmic tricks, strategies are developed that need approximately $\log\log\log n$ phases to compute a schedule with constant contention.

3 Scenario 2: Low-Bandwidth Systems

In this section, we focus on data management in parallel processor systems in which the processors are connected by a relatively sparse network. In particular, we consider the $n \times n$ mesh $M = (V, E)$. A complete description of this material and extensions can be found in [13]. Each processor is assumed to have its own local memory module. The set of objects is denoted by X. We describe a data management strategy that exploits locality in order to minimize the congestion.

This data management strategy consists of a placement strategy and a routing strategy. The placement strategy specifies the distribution of the objects among the modules. In particular, it has to answer the following questions.

- How many copies of an object should be made?
- In which memory modules should these copies be placed?
- Which access is directed to which copy?

We distinguish between strategies that use static and dynamic placement.

- Static Placement: Here we assume that the data management strategy has some knowledge of the access frequencies of a given application, i.e., the

strategy are given two functions $h_r : V \times X \longrightarrow \mathbb{R}$ and $h_w : V \times X \longrightarrow \mathbb{R}$ that describe the rates of read and write accesses, respectively, from the nodes in V to the objects in X. Each object can be placed statically in one or more memory modules. Then a read access to an object can be satisfied by one of its copies, but a write access has to update all copies.

- Dynamic Placement: Here all placement decisions have to be made on-line, i.e., there is no knowledge about the access patterns of the application beforehand. We assume that an oblivious adversary initiates the read and write accesses arbitrarily at execution time. ("Oblivious adversary" means that the adversary is not allowed to react on the decisions of the data management strategy.) Copies can be migrated, created, and deleted (i.e., invalidated) during the execution of the application. A read access can be satisfied by any copy and a write access has to update or invalidate all copies. But at least one copy must be updated and preserved.

We aim to minimize the congestion. Of course, this value depends also on the routing. Therefore, we also have to give a routing strategy that minimizes the congestion. In particular, we have to describe the routing paths from the accessing processors to the processors that hold the respective objects.

We measure the quality of a data management strategy by comparing the congestion achieved by this strategy with the congestion achieved by an optimal strategy, i.e., a strategy that achieves minimal congestion. For a given application $\mathcal{A}$, the congestion of an optimal strategy is denoted by $C_{\text{opt}}^{\text{stat}}(\mathcal{A})$ and $C_{\text{opt}}^{\text{dyn}}(\mathcal{A})$ in the static and dynamic models, respectively. A data management strategy is said to be k-competitive in the static or dynamic model if it achieves congestion at most $k \cdot C_{\text{opt}}^{\text{stat}}(\mathcal{A})$ or $k \cdot C_{\text{opt}}^{\text{dyn}}(\mathcal{A})$, respectively, for any application $\mathcal{A}$.

Finding an optimal static placement is NP-hard, which can shown by a straightforward reduction to an NP-hard routing problem. Therefore, it is interesting to find an approximately optimal solution that can be computed efficiently.

In the dynamic model, an optimal strategy has an advantage over an on-line algorithm, because it has full knowledge of the dynamic access pattern. For the competitive analysis, we restrict the optimal strategy slightly: we assume that it answers the requests in the same order as the analyzed on-line algorithm does. Note that without this restriction, the optimal strategy could defer write accesses to later time steps in order to save repeated read accesses from the same processor to the same object.

3.1 Description of the Strategy

In this section, we present a master algorithm for static and dynamic data management for the $n \times n$ mesh $M = (V, E)$. Generalizations of this algorithm to meshes of arbitrary dimension can be found in [13].

We have to describe the distribution of the objects and their copies among the processors in M and the routing paths between the accessing processors and

the respective copies. The main problem is that it is difficult to calculate lower and upper bounds for the load and congestion induced by a specific placement of the copies because there are several possible routing paths between every pair of nodes.

Our basic tool is a randomized but locality preserving embedding of *access trees* into the mesh and a simulation of the relatively simple data management strategies for trees. What makes things much easier in trees is that we do not have to specify the routing paths in these networks because there is only one simple path between any pair of nodes. In the following, we describe the embedding of the access trees and then we describe how to apply the optimal static or close-to-optimal dynamic tree strategies from [13] to these trees.

For each object $x \in X$, define the access tree $T(x)$ to be a complete 4-ary tree of height $\log n$. The embedding of $T(x)$ in the mesh M is based on a hierarchical decomposition of M. This decomposition is defined recursively. If $n = 1$ then we have reached the end of recursion. Otherwise, we partition M into 4 non-overlapping $\frac{n}{2} \times \frac{n}{2}$ submeshes. Figure 1 shows an example of this decomposition. We associate a submesh $M(v)$ with each node v of $T(x)$: the mesh M itself is

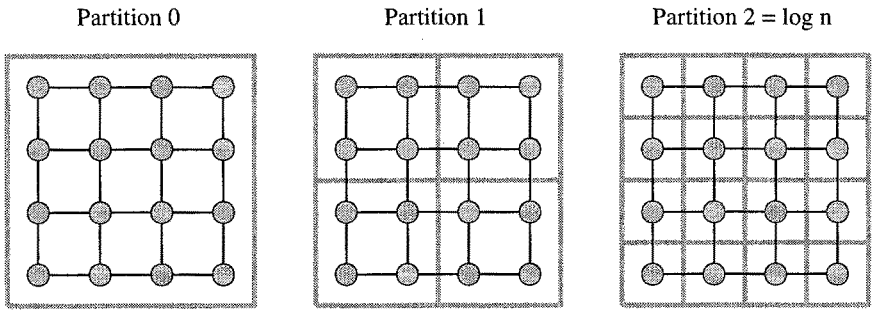

Fig. 1. The decomposition of a 4×4 mesh.

associated with the root of the tree, and for each node v of $T(x)$ that is not a leaf, each of the 4 children of v is associated with one of the 4 submeshes of $M(v)$. We embed the access trees randomly into M, i.e., each interior node v of $T(x)$ is mapped by a random hash function $h(x, v)$ to one of the processors in $M(v)$, and each leaf v of $T(x)$ is mapped onto the single processor in $M(v)$. (For simplicity we assume that these hash functions map in a truly random fashion, i.e., uniformly and independently.)

The remaining description of our data management strategy is very simple: For object $x \in X$, we simulate the static or dynamic tree strategy described in [13] on access tree $T(x)$. The static tree strategy achieves minimal load on each tree edge, i.e., it describes an optimal mapping of the objects to the tree nodes with respect to the read and write frequencies. Moreover, this mapping can be calculated efficiently and distributed by the nodes of the tree and therefore also by the processors of the mesh. Also the dynamic tree strategy achieves

optimal load for the data transfer, but it requires some amount of distributed bookkeeping, which increases the load on the edges. However, it is shown that the load due to bookkeeping messages is relatively small compared to the load of the data transfer.

For simulating the access trees we have to route messages sent between neighboring access tree nodes along some paths in the mesh. For this, we choose the paths that are provided by almost every standard router for the mesh, i.e., the messages are sent along the dimension-by-dimension order path in the mesh. The following theorem shows that these strategies achieve close-to-optimal congestion.

Theorem 2 ([13]). *The static and the dynamic access tree strategies are $O(\log n)$-competitive, with high probability w.r.t. the random choices of the hash functions.*

Note that this result cannot be improved for the dynamic model, because it is shown in [13] that for any dynamic strategy there is an application for which the expected congestion is approximately $\log n$ times the optimal congestion.

We now sketch the proof of the above theorem. In order to prove the competitiveness, one has to prove a lower bound on the congestion of the optimal strategy and an upper bound on the congestion of the access tree strategy.

Consider a complete 4-ary tree T of height $\log n$. This tree is isomorph to the access trees, and as in the case of the access trees, we associate a submesh $M(v)$ of M with each node v of T. In particular, each leaf of T is associated with a submesh of size one, which means that each leaf corresponds to a processor in M. The bandwidth of an edge of height i in this tree is defined to be $2 \cdot 2^i$, where the edges incident to the leaves are assumed to have height 1.

For any application and any data management strategy $\mathcal{S}$ on the mesh M, the tree T can simulate the behavior of M according to strategy $\mathcal{S}$. In the simulation, any message that is sent between two nodes in M is routed along the unique shortest path between the respective nodes in T. Define the relative load of an edge e in T to be the number of packets sent along e during the simulation divided by the bandwidth of e, and define the congestion in T to be the maximum relative load over all edges in T.

This simulation scheme yields a lower bound on the congestion of the optimal mesh strategy. Consider a node v of the tree and its associated submesh $M(v)$. Each packet that crosses the tree edge e between v and its parent during the simulation has to enter or leave the submesh $M(v)$ of the mesh. As the number of edges leaving $M(v)$ is at most the bandwidth of e, the congestion in M is not smaller than the congestion in T. As a consequence, for any application, the congestion of the optimal tree strategy is at most the congestion of the optimal mesh strategy.

Similar, it can be shown that the congestion in M is not bigger than approximately $\log n$ times the congestion in T, with high probability. This gives an upper bound on the congestion of the access tree strategy. Note that the simulation of the access tree strategy on T (i.e., T simulates the behavior of M

according to the access tree strategy) yields the optimal strategy for T. Consequently, the congestion of the access tree strategy is at most approximately $\log n$ times the optimal congestion on T, and therefore, at most approximately $\log n$ times the optimal congestion on M. This gives the theorem.

4 Experimental Work

The algorithms described in this paper have been developed within the Sonderforschungsbereich "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen"[1] supported by the Deutsche Forschungsgesellschaft (DFG). In particular, we implement several variants of the presented static and dynamic data management algorithms for Scenario 2 in the DIVA library[2], in order to demonstrate that the introduced ideas also work well in practice. This library is portable to any parallel system supporting either Parix or MPI.

Similar strategies to the one introduced for Scenario 1 are implemented within the ACTS SICMA project[3]. The aim of this project is to design a scalable server for the delivery of images, data and continuous multimedia information and to demonstrate its efficiency by applying it to a relevant application, the "Virtual Museum".

References

1. B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proc. of the 25th ACM Symp. on Theory of Computing (STOC)*, pages 164–173, 1993.
2. B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. In *Proc. of the 7th ACM Symp. on Discrete Algorithms (SODA)*, pages 574–583, 1996.
3. Y. Bartal. Survey on distributed paging. In *Proc. of the Dagstul Workshop on On-line Algorithms*, 1996.
4. R. Cypher, F. Meyer auf der Heide, C. Scheideler, and B. Vöcking. Universal algorithms for store-and-forward and wormhole routing. In *Proc. of the 26th ACM Symp. on Theory of Computing (STOC)*, pages 356–365, 1996.
5. A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Improved optimal shared memory simulations, and the power of reconfiguration. In *Proc. of the 3rd Israel Symposium on Theory of Computing and Systems*, pages 11–19, 1995.
6. A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Shared memory simulations with triple-logarithmic delay. In *Proc. of the 3rd European Symposium on Algorithms (ESA)*, pages 46–59, 1995.
7. A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Contention resolution in hashing based shared memory simulations. Technical Report tr-rsfb-96-005, University of Paderborn, 1996.

[1] For more information see http://www.uni-paderborn.de/sfb376/
[2] For more information see http://www.uni-paderborn.de/sfb376/a2/diva.html
[3] For more information see http://www.uni-paderborn.de/cs/sicma/

8. M. Dietzfelbinger and F. Meyer auf der Heide. Dynamic hashing in real time. In *Proc. of the 17th Annual International Colloquium on Automata, Languages and Programming*, pages 6–19, 1990.
9. M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 110–119, 1993.
10. R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. *Algorithmica*, 16, 1996.
11. F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *jalgo*, 17:157—205, 1994.
12. C. Lund, N. Reingold, J. Westbrook, and D. Yan. On-line distributed data management. In *Proc. of the 2nd European Symposium on Algorithms (ESA)*, 1996.
13. B. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for networks of limited bandwidth. Technical Report tr-rsfb-97-042, University of Paderborn, 1997.
14. F. Meyer auf der Heide, C. Scheideler, and V. Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. *Theoretical Computer Science*, 162:245–281, 1996.
15. F. Meyer auf der Heide and B. Vöcking. A packet routing protocol for arbitrary networks. In *Proc. of the 12th Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 291–302, 1995.
16. R. Ostrovsky and Y. Rabani. Universal $O(\text{congestion} + \text{dilation} + \log^{1+\epsilon} n)$ local control packet switching algorithms. In *Proc. of the 29th ACM Symp. on Theory of Computing (STOC), to appear*, 1997.
17. A. G. Ranade. How to emulate shared memory. In *Proc. of the 28th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 185–194, 1987.
18. A. Siegel. On universal classes of fast high performance hash functions. In *Proc. of the 30th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 20–25, 1989.
19. E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34:116–127, 1987.
20. M. N. Wegman and J. L. Carter. New classes and applications of hash functions. In *Proc. of the 20th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 175–182, 1979.