Data Management in Networks: Experimental Evaluation of a Provably Good Strategy

Christof Krick* Friedhelm Meyer auf der Heide* Harald Räcke* Berthold Vöcking†
Matthias Westermann*

Abstract

This paper deals with data management for parallel and distributed systems in which the computing nodes are connected by a relatively sparse network. We present the *DIVA (Distributed Variables) library* that provides fully transparent access to global variables, i.e., shared data objects, from the individual nodes in the network. The current implementations are based on mesh-connected massively parallel computers. The data management strategies implemented in the library use a non-standard approach based on a randomized but locality preserving embedding of "access trees" into the physical network. The access tree strategy was previously analyzed only in a theoretical model using competitive analysis, where it was shown that the strategy produces minimal network congestion up to small factors.

In this paper, the access tree strategy will be evaluated experimentally. We test several variations of this strategy on three different applications of parallel computing, which are matrix multiplication, bitonic sorting, and Barnes-Hut N-body simulation. We compare the congestion and the execution time of the access tree strategy and their variations with a standard caching strategy that uses a fixed home for each data object. Additionally, we do comparisons with hand-optimized message passing strategies producing minimal communication overhead. At first, we will see that the execution time of the applications heavily depends on the congestion produced by the different data management strategies. At second, we will see that the access tree strategy clearly outperforms the fixed home strategy and comes reasonably close to the performance of the hand-optimized message passing strategies. In particular, the larger the network is the more superior the access tree strategy is against the fixed home strategy.

*Department of Mathematics and Computer Science and Heinz Nixdorf Institute, University of Paderborn, Germany. Email: {krueke, fmadh, harry, marsu}@uni-paderborn.de. Supported in part by DFG-Sonderforschungsbereich 376 and EU ESPRIT Long Term Research Project 20244 (ALCOM-IT).

†International Computer Science Institute, Berkeley, CA 94704-1198. Email: voecking@icsi.berkeley.edu. Supported by a DAAD postdoc fellowship. This research was conducted while he was staying at the Heinz Nixdorf Institute, with support provided by DFG-Sonderforschungsbereich 376.

1 Introduction

Large parallel and distributed systems such as massively parallel processor systems (MPPs) and networks of workstations (NOWs) consist of a set of nodes each having its own local memory module. These nodes are usually connected by a relatively sparse network such that communication is often the system's major bottleneck. A dynamic data management service allows to access shared data objects from the individual nodes in the network. In this paper, we investigate different strategies for distributed dynamic data management. Our focus lies on an experimental and comparative evaluation of the "access tree strategy" that we have introduced and theoretically analyzed in [21].

The performance of MPPs and NOWs depends on a number of parameters, including processor speed, memory capacity, network topology, bandwidths, and latencies. Usually, the buses or links are the bottleneck in these systems, because improving communication bandwidth and latency is often more expensive or more difficult than increasing processor speed and memory capacity. But whereas several standard methods are known for hiding latency, e.g., pipelined routing (see, e.g., [13, 15]), redundant computation (see, e.g., [3, 4, **?**, **?**, **?**, **?**]) or slackness (see, e.g., [25]) the only way to bypass the bandwidth bottleneck is to reduce the communication overhead by exploiting locality.

The theoretical analysis of the access tree strategy in [21] considers data management in a competitive model. It is shown that the access tree strategy is able to anticipate the locality included in an application such that the congestion is nearly minimized. The *congestion* is defined to be the maximum amount of data that is transmitted by the same link during the execution of an application (possibly weighted with the inverse of the bandwidth of that link). Thus, the access tree strategy prevents that some of the links become communication bottlenecks. Several classes of networks are considered. For example, it is shown that the access tree strategy achieves minimum congestion up to a factor of $O(d \cdot \log n)$ for every application running on d-dimensional meshes with n nodes. Furthermore, Internet-like clustered networks are investigated.

We have implemented variants of the access tree strategy for meshes in the *DIVA (Distributed Variables) library* that provides fully transparent access to global variables, i.e., shared data objects. In this paper, we introduce the DIVA library and present experimental results. We test several variations of the access tree strategy on three different applications of parallel computing, which are matrix multiplication, bitonic sorting, and Barnes-Hut N-body simulation. We compare the congestion and the execution time of the access tree strategy and their variations with a standard caching strategy that uses a fixed home for each data object and, additionally, with hand-optimized message passing strategies producing minimal communication overhead.

Related practical and experimental work. Several projects deal with the implementation of distributed shared memory in

hardware. Generally, there exists the CC-NUMA (cache-coherent non-uniform-memory-access) concept (see, e.g., [1, **?**, 22]) and the COMA (cache-only memory architecture) concept (see, e.g., [10, 18]).

In a CC-NUMA, each node contains one or more processors with private caches and a memory module that is part of the global shared memory. The address of a shared memory block specifies the memory module and the location inside the memory module. For an application to deliver high performance it is important that a very large fraction of its cache misses is satisfied by the local memory module. In general, however, it is often hard for the operating system or compiler to place the data in the memory module of the node that is most likely to access the data.

In a COMA, the data is moved to the memory module of the node that is currently accessing the data. Thus, each memory module acts as a huge cache where each block has a tag with the address and the state. The hardware supports migration and replication of the data across the different memory modules as dictated by the application access patterns. But, the special hardware needed therefore is very expensive. Detailed experimental evaluations to compare both concepts can be found, e.g., in [24, 27, 28].

Most implementations of distributed shared memory in software are designed for relatively small systems including between 4 and 64 processors [2, 8, 11, 12, 17, 20]. In contrast, the DIVA library which uses a COMA-like concept (without using any kind of special hardware) is designed for systems of arbitrary size reaching from 4 up to several hundreds or thousands of processors. Its implementation is based on the access tree strategy. We compare this strategy with a "fixed home strategy" that realizes a CC-NUMA-like concept on the investigated machine.

Related theoretical work. Our implementations are based on the theoretical work of Maggs et al. [21]. New static and dynamic data management strategies for tree-connected networks, for meshes of arbitrary dimension and side length, and for Internet-like clustered networks are presented. All of these strategies aim to minimize the congestion. The dynamic strategies, on which our work is based, are investigated in a competitive model. For example, it is shown that the competitive ratio is 3 for trees and $O(d \cdot \log n)$ for d-dimensional meshes with n nodes. Further, an $\Omega(\log n/d)$ lower bound is presented on the competitive ratio for on-line routing in meshes. This implies that the achieved upper bound on the competitive ratio is optimal for meshes of constant dimension.

Earlier theoretical work on dynamic data management concentrates on minimizing the total communication load, i.e., the sum, over all links, of the data transmitted by the link rather than the maximum over the links. We believe that the congestion is more relevant for practice as minimizing the total communication load can lead to some very congested links. However, the theoretical results obtained for the total communication load are more general than the one for the congestion. For example, Awerbuch et al. investigate data management in arbitrary networks with non-uniform costs for accessing and migrating a data object. In [5] they present a centralized algorithm that achieves optimum competitive ratio $O(\log n)$ and a distributed algorithm that achieves competitive ratio $O\left((\log n)^4\right)$ on an arbitrary network of size n. Both algorithms are deterministic. Furthermore, in [6] the distributed algorithm is adapted also to systems with limited memory capacities.

2 The strategies implemented in the DIVA library

The DIVA library provides fully transparent, distributed data management for large parallel and distributed systems in which the processors and the memory modules are connected by a relatively sparse network. The current implementations focus on mesh-connected massively parallel processors (MPPs) (for details, see [19]). Our implementations are based on the *access tree strategy* introduced in [21] that aims to minimize the congestion. The access tree strategy describes how copies of the data objects are distributed in the network. In particular, it answers the following questions:

- How many copies of a data object should be created?
- On which memory modules should these copies be placed?
- How should consistency among the copies be maintained?

The access tree strategy is based on a hierarchical decomposition of the network. This decomposition allows to exploit topological locality in an efficient way. We describe recursively the hierarchical decomposition of 2-dimensional meshes. Let m_1 and m_2 denote the side lengths of the mesh. We assume $m_1 \geq m_2$. If $m_1 = 1$ then we have reached the end of the recursion. Otherwise, we partition M into two non-overlapping submeshes of size $\lceil m_1/2 \rceil \times m_2$ and $\lfloor m_1/2 \rfloor \times m_2$. These submeshes are then decomposed recursively according to the same rules. Figure 1 gives an example for this decomposition. The hierarchical decomposition has associated with it a decomposition tree, in which each node corresponds to one of the submeshes, i.e., the root of the tree corresponds to the mesh itself, and the children of a node in the tree correspond to the two submeshes into which the submesh corresponding to that node is divided.

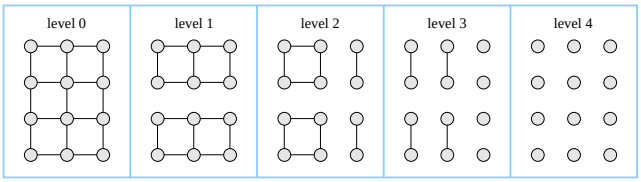

Figure 1: The partitions of $M(4,3)$.

For each global variable, define an *access tree* to be a copy of the decomposition tree. We embed the access trees randomly into the mesh, i.e., for each variable, each node of the access tree is mapped at random to one of the processors in the corresponding submesh. On each of the access trees, we simulate a simple caching strategy. All messages that should be sent between neighboring nodes in the access trees are sent along the *dimension-by-dimension order paths* between the associated nodes in the mesh, i.e., the unique shortest path between the two nodes using first edges of dimension 1, then edges of dimension 2, and so on. The strategy running on the access tree works as follows. For each object x, the nodes that hold a copy of x always build a connected component in the access tree. Read and write accesses from a node v to an object x are handled in the following way.

- v wants to read x: v sends a request message to the nearest node u in the access tree holding a copy of x. u sends the requested value of x to v. A copy of x is created on each access tree node on the path from u to v.

- v wants to write x: v sends a message including the new value, i.e., the value that should be written, to the nearest node u in the access tree holding a copy of x. u starts an invalidation multicast to all other nodes holding a copy of x, then modifies its own copy, and sends the modified copy to v. This copy is stored on each access tree node on the path from u to v.

Note that all nodes in the above description refer to access tree rather than mesh nodes. Further, all the communication for reading and writing including the invalidation multicast follows the branches of the access tree.

The above tree strategy achieves optimal competitive ratio 3. By the randomized but locality preserving embedding of access trees into the mesh this strategy achieves competitive ratio $O(d \cdot \log n)$ on d-dimensional meshes with n nodes. This is also optimal for constant d since any on-line routing algorithm on meshes has competitive ratio $\Omega(\log n/d)$. The proofs of these results can be found in [21].

Implemented variations of the access tree strategy. The DIVA library includes several variants of the access tree strategy. These variations use trees of different heights and degrees. The idea behind varying the degree of the access trees is to reduce the overhead due to additional startups: Any intermediate stop on a processor simulating an internal node of the access tree requires that this processor receives, inspects, and sends out a message. The sending of a message by a processor is called a *startup*. The overhead induced by the startup procedure (inclusive the overhead of the receiving processor) is called *startup cost*. Obviously, a more flat access tree reduces the number of intermediate stops and, therefore, the overall startup costs.

The original access tree strategy uses a 2-ary hierarchical mesh decomposition. Alternatively, we use 4-ary and 16-ary decompositions leading to 4-ary and 16-ary access trees, respectively. The 4-ary decomposition just skips the odd decomposition levels of the 2-ary decomposition, and the 16-ary decomposition then skips the odd decomposition levels of the 4-ary one. Another way to get flatter access trees is to terminate the hierarchical mesh decomposition with submeshes of size k. An access tree node that represents a submesh of size $k' \leq k$ gets k' children, one for each of the nodes in the submesh. We define the *ℓ-k-ary access tree strategy*, for $\ell = \{2,4\}$ and $k \geq \ell$, to use an ℓ-ary hierarchical mesh decomposition that terminates at submeshes of size k.

Practical improvements to the access tree strategy. In order to shorten the routing paths we use a more regular embedding of the access trees than described in the theoretical analysis. We assume that the processors are numbered from 0 to $P-1$ in row major order. The root of an access tree is mapped randomly into the mesh. The embedding of all access tree nodes below the root depends on the embedding of their respective parent node: Consider an access tree node v with parent node v'. Let M denote the submesh represented by v and M' the submesh represented by v', which includes M. Suppose v' is mapped to the node in the ith row and jth column of M'. Then v is mapped to the node in row i mod m_1 and column j mod m_2 of M, where m_1 and m_2 denote the number of rows and columns of M, respectively.

The modified embedding adds dependencies between the mappings of different nodes included in the same access tree such that the theoretical analysis does not hold anymore. However, we have not recognized any bad effects due to these dependencies in our experiments. The major advantage of the modified embedding is that it decreases the expected distances between the processors simulating neighbored access tree nodes.

Besides, we note that the original description of the access tree strategy intends that the embedding of an access tree node is changed when too many accesses are directed to the same node. In theory, this remapping improves the granularity in the random experiments. We omit this remapping as we believe that the constant overhead induced by this procedure will not be retained in practice.

Finally, in the theoretical analysis we have assumed that the memory resources are unbounded. In the experiments we will not investigate the effects of bounded memory capacities either. The strategies implemented in the DIVA library, however, are able to deal with this problem. If the local memory module is full then data objects will be replaced in least recently used fashion. However, in all our experiments there will be a sufficient amount of memory so that no data objects have to be replaced (unless otherwise stated).

The fixed home strategy. The variants of the access tree strategy will be compared to a data management strategy following a standard approach in which each global variable is assigned a processor keeping track of the variable's copies. This processor is called the *home* of the variable. The home of each variable is chosen uniformly at random from the set of processors. In order to manage the copies of the data objects, we use the well known *ownership scheme* described, e.g., in [**?**]. Originally, the ownership scheme was developed for shared memory systems in which many processors with local caches are connected to a centralized main memory module by a bus. The scheme works as follows.

At any time either one of the processors or the main memory module can be viewed as the *owner* of a data object. Initially, the main memory module is the owner of the object. A write access issued by a processor that is not the owner of the data object assigns the ownership to this processor. A read access issued by another processor moves the ownership back to the main memory. Write accesses of the owner can be served locally, whereas all other write accesses have to invalidate all existing copies and create a new copy at the writing processor. Read accesses by processors that do not hold a copy of the requested data object move a copy from the owner to the main memory module (if the main memory module is not the current owner itself) and a copy to the reading processor. In this way, subsequent read accesses of that processor can be served locally.

In a network with distributed memory modules, the home processor plays the role of the main memory module. It keeps track of all actual copies and is responsible for their invalidation in case of a write access. In the original scheme, invalidation is done by a *snoopy cache controller*, that is, each processor monitors all of the data transfers on the bus. Of course, this mechanism does not work in a network. Here the home processor has to send an invalidation message to each of the nodes holding a copy.

If each write access of a processor to a data object is preceded by a read access of this processor to the same object then the fixed home strategy using the ownership scheme corresponds to a P-ary access tree strategy with P denoting the number of processors. Interestingly, this condition is met for every write access in the three applications that we will investigate. Therefore,we think, the fixed home strategy is very well suited for comparisons with the access tree strategy.

Synchronization mechanisms. In addition to the pure data management routines, the DIVA library provides routines for barrier synchronization and for the locking of global variables. These routines are implementations of elegant algorithms that use access trees, too.

More details. More details about the implemented strategies, e.g., about the data tracking (i.e., how the nodes locate the copies), the modified embedding of the access tree into the mesh, and the synchronization mechanisms can be found in [**?**].

3 Experimental evaluation of the access tree strategy

In this section, we will compare the congestion and the execution time of the variations of the access tree strategy to the fixed home strategy and to hand-optimized message passing solutions. We will consider three applications of parallel computing, which are matrix multiplication, bitonic sorting, and a Barnes-Hut N-body simulation adapted from the SPLASH II benchmark [23, **?**].

The implemented algorithms for matrix multiplication and sorting are *oblivious*, i.e., their access or communication patterns do not depend on the input. The reason we have decided to include these algorithms in our small benchmark suite is that they allow us to compare the dynamic data management strategies with hand-optimized message passing strategies.

The third application, the Barnes-Hut N-body simulation, is non-oblivious. We believe that a communication mechanism that uses shared data objects is the best solution for this application (in contrast to the other two applications). However, we cannot construct a hand-optimized message passing strategy achieving minimal congestion for this application. Therefore, we concentrate on the comparison of different dynamic data management strategies.

The hardware platform. Our experiments were done on the Parsytec GCel, which nodes are connected by a 32×32 mesh network. The GCel has the following major characteristics. The used routing mechanism is a wormhole router that transmits the messages along dimension-by-dimension order paths as assumed in the theoretical analysis. We have measured a maximum link bandwidth of about 1 Mbyte/sec.. This bandwidth can be achieved in both directions of a link almost independently of the data transfer in the other direction. However, fairly large messages of about 1 Kbyte have to be transmitted to achieve this high bandwidth. The processor speed is about 0.29 integer additions a micro sec., which we have measured by adding a constant term to all entries in a vector of 1000 integers ($\widehat{=}$ 4 bytes). According to these values, the ratio between the link and the processor speed is about 0.86.

The reason why we have chosen the GCel as our experimental platform is that it allows to scale up to 1024 processors. We believe that the results regarding the efficiency of different data management strategies hold in similar way also for other mesh-connected machines having comparable ratios between link and processor speed (e.g., Intel ASCII red, Fujitsu AP 1000). Of course, the results on the congestion of different data management strategies given in the following are independent from hardware characteristics like link bandwidth and processor speed.

3.1 Matrix multiplication

The first application is an algorithm for multiplying matrices. We have decided to implement the matrix square $A := A \cdot A$ rather than general matrix multiplication $C := A \cdot B$ because the matrix square requires the data management strategy to create and invalidate copies of the matrix entries whereas the general matrix multiplication does not require the invalidation of copies. Note that the invalidation makes the problem more difficult only for the dynamic data management strategies but not for a hand-optimized message passing strategy.

For simplicity, we assume that the size of the mesh is $\sqrt{P} \times \sqrt{P}$ and that the size of the matrix is $n \times n$, where n is a multiple of $\sqrt{P}$. Let $p_{i,j}$ denote the processor in row i and column j, for $0 \leq i, j < \sqrt{P}$. The matrix is partitioned into P equally sized blocks $A_{i,j}$ such that block $A_{i,j}$ includes all entries in row i' and column j' with $i \cdot n/\sqrt{P} \leq i' < (i+1) \cdot n/\sqrt{P}$ and $j \cdot n/\sqrt{P} \leq j' < (j+1) \cdot n/\sqrt{P}$, for $0 \leq i, j < \sqrt{P}$. The block size is $m = n^2/P$. Processor $P_{i,j}$ has to compute the value of "its" block $A_{i,j}$, i.e., $A_{i,j} := \sum_{k=0}^{\sqrt{P}-1} A_{i,k} \cdot A_{k,j}$.

Each block $A_{i,j}$ is represented by a global variable $A[i,j]$. We assume that $A[i,j]$ has been initialized by processor $p_{i,j}$ such that the only copy of this variable is already stored in the cache of this processor. At the end of the execution, the copies are left in the same configuration. Hence, we measure the matrix algorithm as if it is applied repeatedly in order to compute a higher power of a matrix. The matrix multiplication algorithm works as follows.

The processors execute the following program in parallel. At the beginning, each processor $p_{i,j}$ initializes a local data block H to 0. Then the processors execute a "read phase" and a "write phase" that are separated by a barrier synchronization. The *read phase* consists of $\sqrt{P}$ steps: In step $0 \leq k' < \sqrt{P}$, processor $p_{i,j}$ reads $A[i,k]$ and $A[k,j]$, where $k = (k' + i + j) \bmod \sqrt{P}$, then computes $A_{i,k} \cdot A_{k,j}$, and adds this product to H. Note that the definition of k yields a *staggered* execution, i.e., at most two processors read the same block in the same time step. In the subsequent *write phase*, each processor $p_{i,j}$ writes its local data block H into the global variable $A[i,j]$.

Communication patterns of different data management strategies. The *hand-optimized* strategy works as follows. Each processor $p_{i,j}$ sends its block $A[i,j]$ along its row and its column, that is, the block is sent simultaneously along the four shortest paths from processor $p_{i,j}$ to the processors $p_{i,0}$, $p_{i,\sqrt{P}-1}$, $p_{0,j}$, and $p_{\sqrt{P}-1,j}$. Each processor which is passed by the block creates a local copy. The algorithm finishes when all copies of all blocks are distributed. Obviously, this strategy achieves minimal total communication load and minimal congestion. The congestion is $m \cdot \sqrt{P}$.

Next we consider the communication pattern of the 4-ary access tree and the fixed home strategy. In the read phase both strategies distribute copies of each block $A_{i,j}$ in row i and column j. In the write phase, both strategies send only small invalidation messages to the nodes that hold the copies that have been created in the read phase. Obviously, for large data blocks, the communication load produced in the read phase clearly dominates the communication load produced in the write phase.

Let us consider the communication pattern of the read phase in more detail. Figure 2 depicts how the copies are distributed among the processors in a row. (The distribution of copies in a column is similar.) The fixed home strategy sends a copy of variable $A[i,j]$ from node $p_{i,j}$ to the randomly selected home of the variable and then $2\sqrt{P}-2$ copies from the fixed home to each node that is in the same row or column as node $p_{i,j}$. The expected total communication load produced by the fixed home strategy is $\Theta(m \cdot P)$ for the read accesses directed to a single block. The 4-ary access tree strategy distributes block $A_{i,j}$ along a 2-ary subtree of the access tree to all nodes in row i and to all nodes in column j, respectively. This yields an expected total communication load of $\Theta(m \cdot \sqrt{P} \cdot \log P)$ for the read accesses directed to a single block.

Summing up over all blocks yields that the expected total communication load produced by the fixed home strategy is $\Theta(m \cdot P^2)$ whereas the access tree strategy only produces a load of $\Theta(m \cdot P^{3/2} \cdot \log P)$. Under the assumption that the load is distributed relatively evenly among all edges we get that the congestion of the fixed home

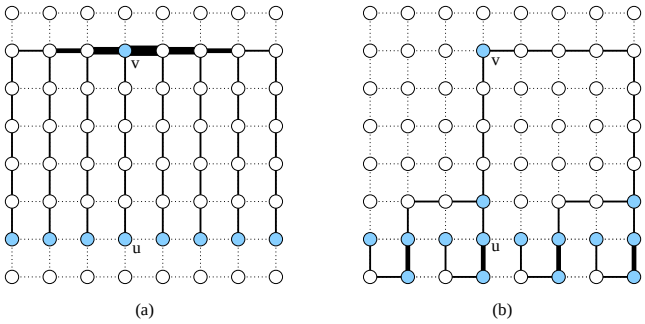

Figure 2: The pictures show the data flow induced by distributing the copies for a single data block in a row of the mesh during the read phase. Picture (a) shows the data flow for the fixed home strategy. Picture (b) shows the data flow for the access tree strategy. The width of a line represents the number of copies transmitted along the respective edge. Node *u* represents the node that is responsible for computing the new value of the data block. Node *v* denotes the randomly selected fixed home or the randomly selected root of the access tree, respectively. At the beginning of the read phase, the node *u* holds the only copy of the data block. At the end of the read phase, each of the colored nodes holds a copy of the data block.

strategy is $\Theta(m \cdot P)$ whereas the estimated congestion of the access tree strategy is $\Theta(m \cdot \sqrt{P} \cdot \log P)$. Thus, the congestion produced by the fixed home strategy deviates by a factor of $\Theta(\sqrt{P})$ from the optimal congestion whereas the congestion produced by the access tree strategy deviates only by a logarithmic factor from the optimum. A formal proof that the original access tree strategy fulfills the latter property is given in [21].

Experimental results. First, the different data management strategies are compared for a fixed numbers of processors. We execute the matrix multiplication for different block sizes ranging from 64 to 4096 integers on a 16×16 submesh of the GCel. Figure 3 depicts the results of a comparative study of the fixed home and the 4-ary access tree strategy, on the one hand, and the hand-optimized message passing strategy, on the other hand.

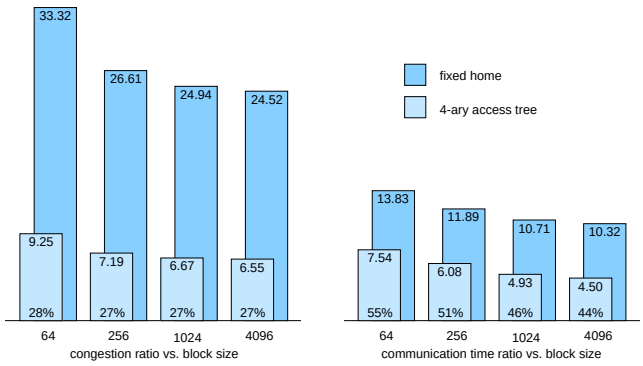

Figure 3: The graphics represent measured congestion and execution time for matrix multiplication on a 16×16 mesh, described as the ratio to the respective values of the hand-optimized strategy.

The *congestion ratio* of the fixed home and the access tree strategy are defined to be the congestion produced by the respective strategy divided by the congestion of the hand-optimized strategy. The hand-optimized strategy achieves minimal congestion growing linear in the block size. The congestion ratios of the dynamic data management strategies decrease slightly with the block size because read request and invalidation messages become less important when the block size is increased.

The *communication time* is defined to be the time needed for serving all read and write requests and executing the barrier synchronizations. The *communication time ratio* relates the communication time taken by the fixed home and the access tree strategy to the communication time taken by the hand-optimized strategy. In order to measure the communication time, we have simply removed the code for local computations from the parallel program. Hence, only the read, write, and synchronization calls remain. The reason for measuring the communication time rather than the execution time is that the time taken for the multiplication of large matrices is clearly dominated by the time needed for local computations, especially, for computing the products of matrix blocks. For example, using the hand-optimized strategy, the fraction of local computations for matrices with blocks of 4096 integers is about 95 %.

The communication times of all tested strategies grow almost linearly in the block size. Interestingly, the time ratios are smaller than the congestion ratios. The reason for this phenomenon is that a large portion of the execution time of the hand-optimized strategy can be ascribed to the startup cost because this strategy only sends messages between neighbored nodes. The number of startups of the hand-optimized strategy is about $2 \cdot \sqrt{P}$ per node, where P denotes the number of processors. For the other two strategies, let us only consider the startups of those messages including the program data because their startup cost are a lot larger than the startup cost for small control messages. The average number of these startups of the fixed home strategy is about $2 \cdot \sqrt{P}$ per node, which corresponds to the hand-optimized strategy. The average number of startups of the access tree strategy, however, is about $4 \cdot \sqrt{P}$ per node because it sends the messages along 2-ary multicast trees. Therefore, the time ratio of the fixed home strategy improves more on the congestion ratio than the one of the access tree strategy. Nevertheless, the access tree strategy is about a factor of 2 faster than the fixed home strategy.

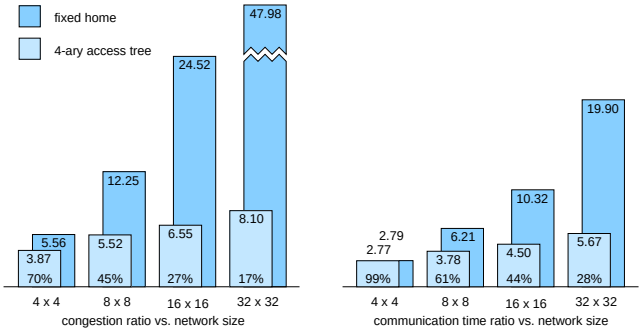

Figure 4: The graphics represent measured congestion and communication time for the matrix multiplication with a fixed block size of 4096, described as the ratio to the respective values of the hand-optimized strategy.

Next we scale over the network size. We run the matrix multiplication for a fixed block size on networks of size 4×4, 8×8, 16×16, and 32×32. Figure 4 illustrates the results. The congestion of the hand-optimized strategy grows linearly in the side length of the mesh. A theoretic analysis of the access pattern of the access tree and the fixed home strategy shows that the congestion

ratios of the two strategies are of order $\log P$ and $\sqrt{P}$, respectively. The increase in the measured congestion ratios corresponds to this assertion. The communication times behave in a similar fashion. As a result, the advantage of the access tree strategy over the fixed home strategy increases with the network size. In the case of 1024 processors, the access tree strategy is more than 3 times faster than the fixed home strategy.

We have also measured the congestion for the 2-ary, the 2-4-ary, the 4-16-ary and the 16-ary access tree strategies. In general, the smaller the degree of the access tree, the smaller the congestion. However, the 4-ary access tree strategy achieves the best communication and execution times because it chooses the best compromise between minimizing the congestion and minimizing the number of startups.

3.2 Bitonic sorting

We have implemented a variant of Batcher's bitonic sorting algorithm [**?**]. Our implementation is based on a sorting circuit. Figure 5 gives an example of the sorting circuit for $P = 8$ processors. A processor simulates a single wire in each step. Each step consists of a simultaneous execution of a specified set of comparators. A comparator $[a : b]$ connects two wires a and b each holding a key and performs a compare-exchange operation, i.e., the maximum is sent to b and the minimum to a. The number of parallel steps is the *depth* of the circuit.

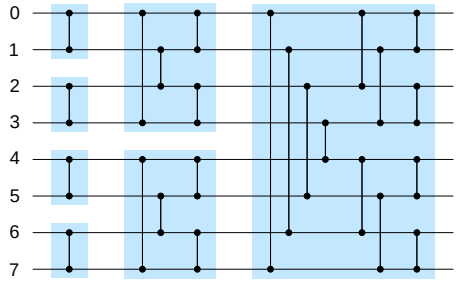

Figure 5: The bitonic sorting circuit for $P = 8$.

In our implementations, each processor holds a set of m keys in a global variable rather than only a single key, and we replace the original compare-exchange operation by a *merge&split operation*, i.e., the processor that has to receive the minimum gets the lower m keys, the other one gets the upper m keys. Several previous experimental studies have shown that bitonic sort is a well suited algorithm for a relatively small number of keys per processor [9, 14, 16, 26].

The algorithm consists of $\log P$ phases such that phase i, for $1 \leq i \leq \log P$ consists of i parallel merge&split steps. The merge&split steps of phase i implement $2^{\log P - i}$ parallel and independent *merging circuits* each of which has depth i and covers 2^i neighbored wires. The merging circuits are marked in grey in Figure 5. The arrangement of the mergers includes locality. Further the merging circuits include locality themselves. Both kinds of locality are exploited by the access tree strategy.

Experimental results. We compare the access tree and the fixed home strategy to a hand-optimized message passing strategy. The hand-optimized strategy simply exchanges two messages between every pair of nodes jointly computing a merge&split operation. Each of these messages is sent along the dimension-by-dimension order path connecting the respective nodes. This simple message passing strategy achieves optimal congestion for the used embedding of the bitonic sorting circuit into the mesh.

Analogously to the experimental studies of the matrix multiplication investigated in the previous section, we use ratios that relate the congestion produced and the time taken by the dynamic data management strategies to the respective values of the hand-optimized strategy. However, we consider the execution time rather than the communication time as the time spent in local computations during the execution of the sorting algorithm is very limited. (If the number of keys is sufficiently large in comparison to the number of processors then the time needed for the initial sorting of each processor's keys dominates the execution time. However, the parameter configurations we will investigate are far below this threshold.)

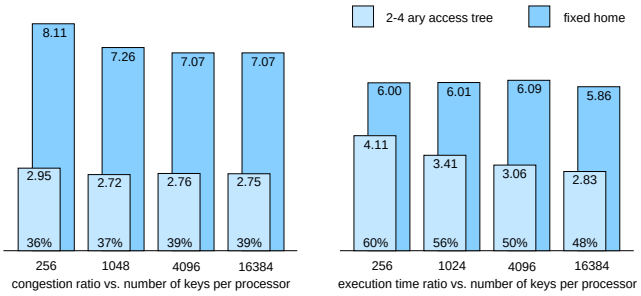

Figure 6: The graphics represent measured congestion and execution time for bitonic sorting on a 16×16 mesh, described as the ratio to the respective values of the hand-optimized strategy.

Figure 6 shows the congestion and the execution time ratios measured on a 16×16 mesh for different numbers of keys per processor. The congestion values of all strategies increase linearly in the number of keys. The congestion ratios of the fixed home and the access tree strategy are slightly decreasing because control messages, i.e., request, invalidation, and acknowledgment messages, become less important when the data messages become larger. The measured execution times of all strategies behave almost linearly, too. The measured ratios show that the execution time is closely related to the congestion. Especially, the execution time ratios for large numbers of keys are amazingly close to the congestion ratios.

In contrast to the results for the matrix multiplication, the 2-ary and the 2-4-ary access tree strategy perform slightly better than the 4-ary strategy. The improvements upon the 4-ary strategy are about 5 % and 8 %, respectively. An explanation for this phenomenon is that the topology of 2-ary access tree matches best the locality in the bitonic sorting circuit. The 2-4-ary access tree strategy improves additionally because of the smaller number of startups in comparison to the 2-ary tree. A further difference to the results for the matrix multiplication is that the time ratio for the access tree strategy is larger than the congestion ratio. The reason for the larger time ratio is that the number of startups of the access tree strategy is much larger than the one of the hand-optimized strategy.

Figure 7 illustrates the behavior of the strategies when scaling over the number of processors. A theoretical analysis of the locality in the bitonic sorting algorithm predicted that the congestion ratio of the fixed home strategy is of order $\log^2 P$ whereas the congestion ratio of the access tree strategy is in $O(1)$ (for details, see [**?**]). The ratio of the fixed home strategy behaves as expected. On first view, the measured increase of the congestion ratio of the access tree strategy seems to be in contradiction to the results of the analysis. However, the measured increase just reflects the increase of a geometric sum. In fact, we can expect that the congestion ratio of

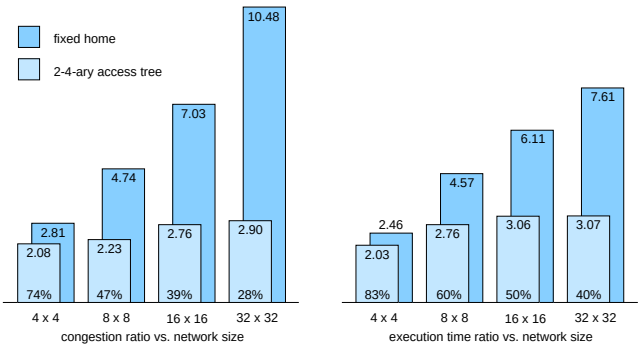

Figure 7: The graphics represent measured congestion and communication time for the bitonic sorting with 4096 keys per processor, described as the ratio to the respective values of the hand-optimized strategy.

the access tree strategy converges against a constant term close to 3.

The measured ratios on the execution time follow the course of the congestion ratios. Therefore, we can conclude that that the execution time of the access tree strategy deviates by a constant factor of about 3 from the hand-optimized strategy whereas the fixed home strategy loses more and more with an increasing number of processors.

3.3 Barnes-Hut *N*-body simulation

This application simulates the evolution of a system of bodies under the influence of gravitational forces. We have taken an implementation of the Barnes-Hut algorithm [7] from the SPLASH II benchmark [23, **?**], and adapted the program code to our DIVA library.

The program implements a classical gravitational N-body simulation, in which every body is modeled as a point of mass exerting forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the force on every body and thereby updating that body's position and other attributes. By far the greatest fraction of the sequential execution time is spent in the force computation phase. If all pairwise forces are computed directly, this has a time complexity of $O(N^2)$. Since an $O(N^2)$ complexity makes simulating large systems impractical, hierarchical tree-based methods have been developed that reduce the complexity to $O(N \log N)$.

The Barnes-Hut algorithm is based on a hierarchical octree representation of space in three dimensions. The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding particles into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single body. The result is a tree whose internal nodes are cells and whose leaves are individual bodies. Empty cells resulting from a cell subdivision are ignored. The tree (and the Barnes-Hut algorithm) is therefore adaptive in that it extends to more levels in regions that have high particle densities.

The tree is traversed once per body to compute the force acting on that body. The force-calculation algorithm for a body starts at the root of the tree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of mass of the cell, and the force exerted by this center of mass on the body is computed. If, however, the center of mass is not far enough away, the cell must be "opened" and each of its subcells visited. In this way, a body traverses those parts of the tree deeper down which represent space that is physically close to it, and groups distant bodies at a hierarchy of length scales.

The main data structure in the application is the Barnes-Hut tree. Since the tree changes in every time-step, we use pointers such that the tree can be reconstructed in every time-step. Each cell and each body is represented by a global variable.

Initially, each processor holds about an equal number of bodies, each of which having a fixed position and velocity. These values are updated over a fixed number of time steps representing a physical time period of length Δt. Each time step is computed within 6 phases:

1. load the bodies into the tree;
2. upward pass through the tree to find center-of-mass of cells;
3. partition the bodies among the processors;
4. compute the forces on all bodies;
5. advance the body positions and velocities;
6. compute the new size of space.

Each of the phases within a time-step is executed in parallel, and the phases are separated by barrier synchronizations. The tree building phase (Phase 1) and the center-of-mass computation phase (Phase 2) require further synchronization by locks since different processors might try to simultaneously modify the same part of the tree.

The parallelism in most of the phases is across the bodies, that is, each processor is assigned a subset of the bodies, and the processor is responsible for computing the new positions and velocities of these bodies. The force computation phase (Phase 4) needs almost all the sequential computation time (about 90 %, see [23]). Therefore, the load balancing is obtained by counting the work that has to be done for a body in the force computation phase within a time step, and using this work count as measure of the work associated to that body. Each processor is assigned about the same amount of work.

For the load balancing, we use a *costzones partitioning scheme*. This scheme yields a very good load balance. Ideally, the resulting partitions correspond to physical regions that are spatially contiguous and equally sized in all directions. Such partitions maintain the "physical locality" and, therefore, minimize interprocessor communication and maximize data re-use. This is of particular interest for the dominating force computation phase in which each processor has to open an expanded path from the root to each of its bodies.

Furthermore, the costzones partitioning scheme is not only able to exploit the spatial locality but it also translates physical locality into topological locality. Analogously to the sorting algorithm, we use processor ident-numbers that correspond to a numbering of the leaves of the mesh-decomposition tree from left to right. (Recall that the mesh-decomposition tree corresponds to all superimposed access trees.) Thus, the costzone partitioning yields a locality preserving mapping of the leaves of the Barnes-Hut tree to the leaves of the mesh-decomposition tree. As a consequence, bodies that are close together in the physical space have a tendency to be computed by the same processor or by two processor that are close together with respect to the distances defined by the mesh-decomposition tree.

Experimental results. First, we consider a fixed number of processors $P = 256$ and scale over the number of bodies N from 10,000 to 60,000. The Barnes-Hut algorithm is executed on a 16×16 submesh of the GCel. Physicists usually simulate hundreds

or thousand of time steps. Since the execution times for the individual time steps are already relatively stable after the simulation of the first two steps, we only simulate 7 time steps, from which only the last 5 are included in the measurement.

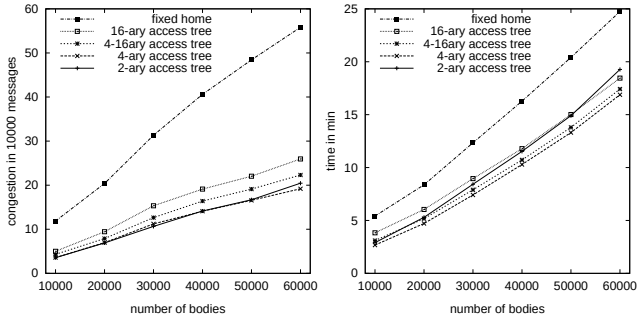

Figure 8: Congestion and execution time for the Barnes-Hut N-body simulation on a 16×16 mesh.

Figure 8 describes the congestion and the execution time of the measured rounds. The measured congestion corresponds to the maximum number of messages that have traversed the same edge during the execution of the 5 rounds, and the execution time corresponds to the total time needed for the 5 rounds.

A comparison between the access tree strategies and the fixed home strategy clearly shows that the access tree strategies are able to exploit the topological locality in the Barnes-Hut algorithm. In general, the higher the access tree is, the smaller is the congestion. (The increase of the congestion for the 2-ary access tree from 50,000 to 60,000 bodies is due to copy replacement, which starts at 60,000 bodies for the 2-ary access tree strategy. All other strategies do not have to replace copies as the storage capacities are sufficient for these strategies.) However, because of the large overhead due to additional startups, the 2-ary strategy is clearly slower than the 4-ary although it has a very small congestion. As a result, the 4-ary strategy performs best among all strategies.

Most of the execution time of the N-body simulation is spent in the force computation phase (Phase 4), e.g., for 60,000 bodies the 4-ary access tree strategy on the 16×16 mesh spends about 78 % of the execution time in this phase. Another phase which is of special interest for a small number of bodies is the tree building phase (Phase 1). For 10,000 bodies, the fixed home strategy spends about 44 % of its time in this phase. We investigate the tree building and the force computation phases in further detail.

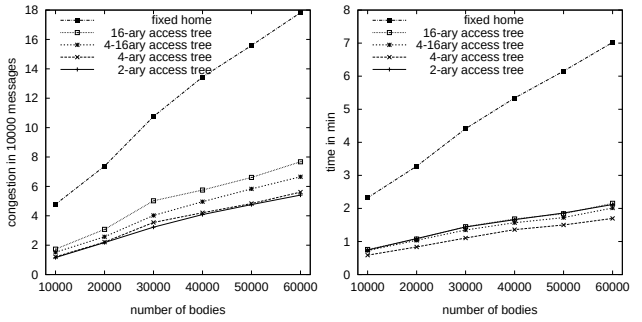

Figure 9: Congestion and execution time of the tree building phase on a 16×16 mesh.

Figure 9 shows the congestion and the execution time of the tree building phase. The Barnes-Hut tree is rebuilt in each simulated time step. For each of its bodies, a processor follows a path in the tree leading downwards from the root to a node which does not have a child that represents the region of space to which the body belongs. If this node is a cell then the processor can add its body to this cell as a child. If this node is another body then this body is replaced by a cell and added to this cell as a child. Afterwards, the process of loading the processor's body into the tree is continued. Locks are used in order to avoid different processors simultaneously changing the data of the same body, i.e., we attach a lock to each body.

Obviously, the root cell is the bottleneck of the algorithm for loading all the bodies into the Barnes-Hut tree. This cell has to be read once for every body, which, however, is only expensive in the beginning of the phase when several processors contend for locking the root as they want to add a body as a child of the empty root. Later on each processor holds a copy of the root such that reading the root is very cheap. The access tree strategies profit much from their capability to distribute the copy of the root cell very efficiently via a multicast tree, whereas, using the fixed home strategy, one processor, the home of the root, has to deliver a copy of the root to each processor one by one. Similar bottlenecks also occur for other nodes on the top level of the Barnes-Hut tree, resulting in a much higher offset for the congestion of the fixed home strategy. Obviously, this offset increases with the number of processors.

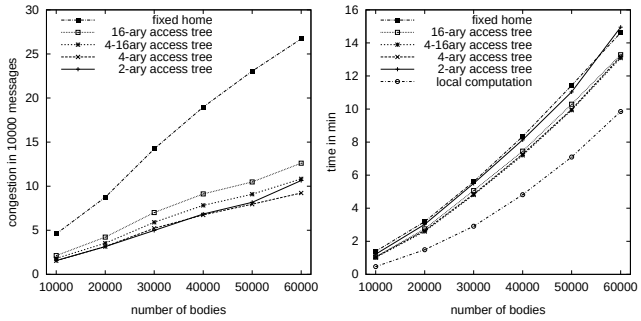

Figure 10: Congestion and execution time of the force computation phase on a 16×16 mesh.

Figure 10 depicts the congestion and the execution time of the force computation phase added over 5 time steps. Besides, the picture shows the time spent in local computations in this phase. For 60,000 bodies, the 4-ary access tree strategy only uses 25 % of the time for communication whereas the fixed-home strategy requires about 33 % for communication. As the force computation phase does not include write accesses to global variables but many read accesses, many copies are created during this phase.

In the force computation phase, each processor traverses the Barnes-Hut tree once per body to compute the forces acting on the body. The calculation follows the path from the root to the body and opens several cells and bodies in the neighborhood of this path. Due to the locality of the algorithm, the neighborhoods of the paths for a set of bodies that are computed by the same processor overlap to a great amount. This leads to cache hit ratios of about 99 % even for a relatively small number of bodies.

In the force computation phase, the variations of the access tree strategy win against the fixed home strategy because of their capability to efficiently distribute copies of a global variable to those submeshes where they are needed. Also in this phase, the 4-ary access tree strategy outperforms all other strategies.

Finally, let us investigate what happens if the number of proces-

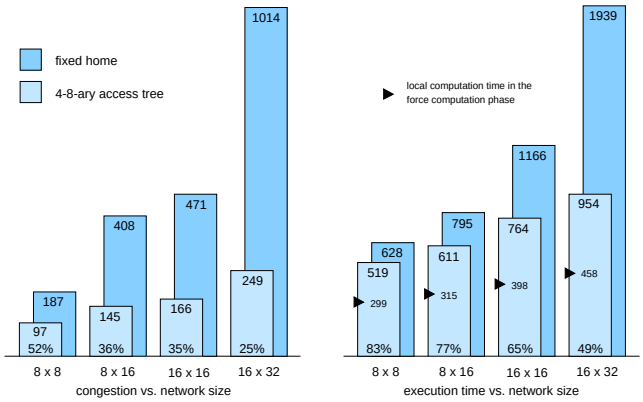

Figure 11: The graphic represents measured values for the Barnes-Hut N-body simulation for different numbers of processors P. The number of bodies is increased with the number of processors, that is, $N = 200 \cdot P$. The congestion is measured in units of 1000 messages. The time is measured in seconds.

sors is scaled from 64 to 512. We change the number of processors P and the number of bodies N simultaneously in such a way that $N = 200 \cdot P$. Figure 11 shows how the congestion and the execution time behave if the number of processors is scaled. The congestion produced by the access tree and the fixed home strategy is mainly determined by the largest side length of the mesh rather than by the number of processors. Obviously, the congestion produced by the access tree strategy grows less than the congestion of the fixed home strategy. The superiority of the access tree strategy against the fixed home strategy increases with the number of processors. For the largest number of processors that we have tested, the ratio of the execution times taken by the access tree and the fixed home strategy is about 49 %. The results on the communication time, i.e., the execution time minus the time taken for local computations in the force computation phase, are even more impressive. The best ratio of the communication time taken by the access tree and the fixed home strategy is about 33 % for 512 processors. Hence, the access tree strategy improves by a factor of 3 on a standard caching strategy using a randomly selected fixed home for each variable.

4 Conclusions

In order to illustrate the practical usability of the access tree strategy, we have implemented some variations of the strategy on a massively parallel mesh-connected computer system, and tested it for three standard applications of parallel computing. The experimental results show that the execution time and the congestion are closely related. Clearly, the access tree strategy looses against the hand-optimized message passing strategies, if we consider oblivious applications like matrix multiplication or bitonic sorting, where the full knowledge of the access pattern is used to exploit the locality inherent in the algorithm. Astonishingly, the on-line access tree strategy achieves execution times that are reasonably close to the times of the hand-optimized message passing strategies. This shows that our method to "automatically adopt to the locality of the application" is very powerful. However, the more important applications are highly non-oblivious like the Barnes-Hut N-body simulation. Besides, we compared the access tree strategy with a standard caching scheme in which a fixed home processor is assigned to each data object that keeps track of the object's copies. The access tree strategy clearly outperforms the fixed home strategy in all experiments. Moreover, the larger the network is the more superior the access tree strategy is against the fixed home strategy. One reason for the efficiency of the access tree strategy is that it is simple and produces only very small overhead for bookkeeping. The most important reason, however, seems to be the small congestion produced by the access tree strategy.

Our experimental results show that, apart from the congestion, the startup cost are a further important cost factor. This kind of communication overhead is not part of the theoretical model in which the access tree strategy has been devised. In our practical studies we have reduced the number of startups by increasing the degree of the access trees such that they become flatter. This decreases the number of startups required to serve the requests but possibly increases the length of some of the routing paths. On the investigated architecture, 4-ary access trees seem to be most useful. Only the sorting application performs better when using a 2-ary access tree because the pattern of locality in the bitonic sorting circuit corresponds to the 2-ary mesh decomposition. It is a challenging task to incorporate startup cost in an adequate way in a theoretical model and to develop efficient dynamic data management strategies in that model.

We believe that the access tree strategy can also be efficiently adapted to other networks than meshes and clustered networks. All that is needed is a hierarchical network decomposition that possesses similar properties to the mesh decomposition. The current implementations of data management strategies in the DIVA library are based on mesh-connected MPPs. Other networks, e.g., Linux workstation clusters based on SCI communication technology, will be addressed in future work.

References

[1] A. Agarwal, D. Chaiken, D. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of the 1st Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, pages 18–28, Feb. 1996.

[3] M. Andrews, F. T. Leighton, P. T. Metaxas, and L. Zhang. Automatic methods for hiding latency in high bandwidth networks. In *Proceedings of the 28th ACM Symposium on Theory of Computing (STOC)*, pages 257–265, 1996.

[4] M. Andrews, F. T. Leighton, P. T. Metaxas, and L. Zhang. Improved methods for hiding latency in high bandwidth networks. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 52–61, 1996.

[5] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proceedings of the 25th ACM Symposium on Theory of Computing (STOC)*, pages 164–173, 1993.

[6] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. *Journal of Algorithms*, 28:67–104, 1998.

[7] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[8] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the IEEE CompCon Conference*, 1993.

[9] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM–2. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 3–16, 1991.

[10] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the ksr1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, 1992.

[11] R. Cabrera-Dantart, I. Demeure, P. Meunier, and V. Bantro. Phosphorus: a tool for shared memory management in a distributed environment. Technical Report 95D003, ENST Paris, Comp. Science Dept., 1995.

[12] J. B. Carter, J. K. Bennett, and W. Zwaenepol. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, 1991.

[13] R. J. Cole, B. M. Maggs, and R. K. Sitaraman. On the benefit of supporting virtual channels in wormhole routers. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 131–141, 1996.

[14] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast parallel sorting under LogP. In *Portability and Performance for Parallel Processing*, pages 71–98, 1994.

[15] R. Cypher, F. Meyer auf der Heide, C. Scheideler, and B. Vöcking. Universal algorithms for store-and-forward and wormhole routing. In *Proceedings of the 28th ACM Symposium on Theory of Computing (STOC)*, pages 356–365, 1996.

[16] R. Diekmann, J. Gehring, R. Lüling, B. Monien, M. Nübel, and R. Wanka. Sorting large data sets on a massively parallel system. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 2–9, 1994.

[17] B. D. Fleisch, L. Hyde, and N. C. Juul. Mirage+: A kernel implementation of distributed shared memory for a network of personal computers. *Software-Practice and Experience*, 23(10), 1994.

[18] E. Hagersten, S. Haridi, and D. Warren. The cache-coherence protocol of the data diffusion machine. In M. Dubois and S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*. Kluwer Academic Publishers, 1990.

[19] C. Krick, H. Räcke, B. Vöcking, and M. Westermann. The DIVA (distributed variables) library. www.uni-paderborn.de/sfb376/a2/diva.html, Paderborn University, 1998.

[20] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computers*, 7(4):321–359, 1989.

[21] B. M. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for networks of limited bandwidth. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 284–293, 1997.

[22] A. Saulsbury and A. Nowatzyk. Simple coma on S3.MP. In *Proceedings of the 1995 International Symposium on Computer Architecture Shared Memory Workshop*, 1995.

[23] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1992.

[24] G. D. Stamoulis and J. N. Tsitsiklis. The efficiency of greedy routing in hypercubes and butterflies. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 241–259, 1991.

[25] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33, 1990.

[26] A. Wachsmann and R. Wanka. Sorting on a massively parallel system using a library of basic primitives: Modeling and experimental results. In *Proceedings of the 3rd European Conference in Parallel Processing (Euro-Par)*, pages 399–408, 1997.

[27] L. Yang and J. Torrellas. Speeding up the memory hierarchy in flat COMA multiprocessors. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA-3)*, 1997.

[28] Z. Zhang and J. Torrellas. Reducing remote conflict misses: NUMA with remote cache versus COMA. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA-3)*, 1997.