

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

Christof Krick* Friedhelm Meyer auf der Heide*

Harald Rucke* Berthold Vocking† Matthias Westermann‡

Abstract

This paper deals with data management for parallel and distributed systems. We present the *DIVA (Distributed Variables) library* that provides direct access to shared data objects from each node in a network. The current implementations are based on mesh-connected massively parallel computers. Our algorithms dynamically create and discard copies of the data objects in order to reduce the communication overhead. We use a non-standard approach based on a randomized but locality preserving embedding of “access trees” into the network.

The access tree strategy was previously analyzed only in a theoretical model. A competitive analysis proved that the strategy minimizes the network congestion up to small factors. In this paper, the access tree strategy is

*Heinz Nixdorf Institute and Department of Mathematics and Computer Science, Paderborn University, Germany. {krueke, fmadh, harry}@uni-paderborn.de. Partially supported by the DFG-Sonderforschungsbereich 376 and the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

†Max-Planck-Institut fur Informatik, Saarbrucken, Germany. voecking@mpi-sb.mpg.de. Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). This research was conducted while he was staying at the Heinz Nixdorf Institute, with support provided by the DFG-Sonderforschungsbereich 376.

‡International Computer Science Institute, Berkeley, CA, USA. marsu@icsi.berkeley.edu. Supported by a fellowship within the ICSI-Postdoc-Programme of the German Academic Exchange Service (DAAD). This research was conducted while he was staying at the Heinz Nixdorf Institute, with support provided by the DFG-Sonderforschungsbereich 376 and the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

evaluated experimentally. We test several variations of this strategy on three different applications of parallel computing, namely matrix multiplication, bitonic sorting, and Barnes-Hut N -body simulation.

We compare the access tree strategy with a standard caching strategy using a fixed home for each data object. Our experiments show that the access tree strategy outperforms the fixed home strategy clearly as it avoids network congestion. Furthermore, we do comparisons with hand-optimized message passing strategies. In fact, we observe that the access tree strategy comes reasonably close to the performance of the hand-optimized message passing strategies while the fixed home strategy performs poorly.

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. A data management service allows to access shared data objects from the individual nodes in the network. Dynamic data management strategies create copies of the data objects at runtime. These copies are distributed among several memory modules so that requests can be served more efficiently. In this paper, we compare different concepts for distributing and managing the copies. Our focus lies on an experimental evaluation of the so-called “access tree strategy” which we have introduced and theoretically analyzed in [27].

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 as improving communication bandwidth and latency is typically more expensive or more difficult than increasing processor speed. Whereas several methods are known for hiding latency, e.g., pipelined routing (see, e.g., [15, 17]), redundant computation (see, e.g., [3, 4, 22, 28, 29, 33]) or slackness (see, e.g., [41]), the only way to bypass the bandwidth bottleneck is to reduce the communication overhead by exploiting locality. Clearly, simply reducing the *total communication load*, i.e., the sum, taken over all links, of the data transmitted by the link (possibly weighted with the inverse of the bandwidth of the link), can result in bottlenecks. In addition to reducing the communication load, the load has to be distributed evenly among all network resources. In other words, one needs to minimize *congestion*, i.e., the maximum, taken over all links, of the amount of data transmitted by the link (possibly weighted with the inverse of the bandwidth of the link).

The access tree strategy exploits locality in order to prevent links from becoming a communication bottleneck. It explicitly aims to minimize the congestion. Several theoretical studies for store-and-forward routing [24, 30, 34, 36] and worm-hole routing [15, 17, 36] show that reducing the congestion is most important in order to get a good network throughput. The theoretical analysis of the access tree strategy in [27] considers data management in a competitive model. In this model, it is shown that the access tree strategy is able to anticipate the locality included in an application. In particular, it is proven that the access tree strategy minimizes the congestion. 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 a d -dimensional mesh with n nodes. Further theoretical results, including competitive ratios for other networks, non-uniform cost measures, and limited memory resources, are given in [27, 31, 32, 43, 45].

We have implemented several variants of the access tree strategy for meshes and other standard strategies in the DIVA (Distributed Variables) library in order to evaluate the access tree strategy also in practice. This library provides access to global variables, i.e., shared data objects, from every node in the network. The implemented data management strategies are fully transparent to the user, that is, the user does not have to worry about the distribution of copies, she just specifies read and write accesses from the processors to the data objects. In this paper, we present experimental results on the performance of different strategies implemented in the library. In particular, we test several variations of the access tree strategy on three different applications of parallel computing, namely matrix multiplication, bitonic sorting, and Barnes-Hut N -body simulation. We compare the congestion and the execution time of the access tree strategy with the respective values of a standard caching strategy that uses a fixed home for each data object. Additionally, we do comparisons with hand-optimized message passing strategies that produce minimal communication overhead.

1.1 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, 25, 35]) and the COMA (cache-only memory architecture) concept (see, e.g., [12, 20]).

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. The memory module in which a data object is stored is called the *home* of the object. Each node is assumed to have a cache in which it can hold copies of the data objects. Usually, the home keeps track of all copies and ensures that all copies are coherent. The performance of an application implemented on a CC-NUMA system mainly depends on the implemented caching strategy. Furthermore, the selection of the homes of the data objects is of some importance, too.

In a COMA, data objects do not have fixed homes. All memory modules act as a huge cache in which the data objects move dynamically. The performance mainly depends on the strategy that is used to create, migrate, delete, and locate the copies of the data objects. Usually, it is assumed that an efficient realization of the COMA concept requires special hardware to support migrations and replications of the data objects. Also the data tracking is usually realized by large hardware tables (directories) that specify the current locations of the copies. Detailed experimental evaluations comparing CC-NUMA and COMA can be found, e.g., in [40, 49, 50].

Most software implementations of distributed shared memory use a CC-NUMA-like concept and are designed for relatively small systems ranging from 4 to 64 processors [2, 10, 13, 14, 19, 26]. In contrast, the access tree strategy implemented in the DIVA library uses a COMA-like concept and is designed for systems of both small and large size reaching from 4 up to several hundreds or even thousands of processors. (Although the library implements a COMA-like concept, there is no need for any kind of special hardware.) In addition, we have implemented a CC-NUMA-like strategy for comparisons, which is called “fixed home strategy”.

1.2 Related theoretical work

Our implementations are based mainly on our theoretical work in [27]. In this work, we introduce new static and dynamic data management strategies for tree-connected networks, meshes of arbitrary dimension and side length, and Internet-like clustered networks. All of these strategies aim to minimize the congestion. The dynamic strategies create, migrate, and discard copies of the data objects at runtime. They 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. Furthermore, an $\Omega(\log n/d)$ lower bound is given for the competitive ratio for on-line routing in a d -dimensional mesh with n nodes. As dynamic data management includes on-line routing, this lower bound implies that the upper bound on the competitive ratio is optimal up to a factor of $\Theta(d^2)$. Similar results

for other networks, non-uniform cost measures, and limited memory resources are given in [27, 31, 32, 43, 45].

The most comparative work is done by Awerbuch et al. [5, 6] who consider data management on arbitrary networks. In [5] they present a centralized strategy that achieves optimum competitive ratio $O(\log n)$ and a distributed strategy that achieves competitive ratio $O((\log n)^4)$ on an arbitrary n -node network. In [6], the distributed strategy is adapted to networks with memory capacity constraints. All competitive ratios are with respect to the total communication load i.e., the sum, over all links, of the data transmitted by the link, instead of the congestion. We believe that the congestion measure is more appropriate for the design of strategies than the total communication load as it prevents links from becoming bottlenecks.

We give a brief description of the very intriguing distributed strategy of Awerbuch et al. because this will illustrate the influence of the cost measure on the design of caching strategies. Their strategy uses a hierarchical network decomposition, introduced in [7], that decomposes the network in a hierarchy of clusters with geometrically decreasing diameter. If a node accesses a file x that has no copy in a cluster of some hierarchy level, then the cluster leader is informed and increases a counter for the file. When the counter reaches D , the requesting node gets a copy of x , where D denotes the ratio between the cost for migrating and the cost for accessing a file. This strategy ensures that the total communication load is minimized up to some logarithmic factors in the size of the network. The cluster leaders, however, can become bottlenecks. In particular, the edges incident to the leader of the top level clusters may become very congested. As a consequence, the competitive ratio in the congestion based cost model becomes bad.

Wolfson et al. introduce an alternative model for dynamic data management [46]. They assume that the access patterns do not vary too much with time, that is, “the pattern of a processor in a time period (of fixed length) is repeated for several time periods”. An adaptive data replication (ADR) algorithm is called *convergent-optimal*, if starting with an arbitrary allocation of copies, the algorithm converges to an optimal allocation within some time periods, under the assumption that the access frequencies of the nodes do not vary within these periods. Here an *optimal allocation* refers to an allocation that yields the minimum total communication load for the respective access frequencies.

Wolfson et al. present a convergent-optimal ADR algorithm for trees, which is a variant of the static placement strategy of the algorithm in [47]. The algorithm converges within a number of periods that is bounded by the diameter of the network. It is easy to check that the allocation obtained by the ADR algorithm when it converges is equivalent to the allocation of our static nibble strategy

presented in [27]. Hence, our nibble strategy for trees and also the static access tree strategies for meshes and clustered networks can be adapted to work in a dynamic context, too, in which access patterns change slowly.

2 The strategies implemented in the DIVA library

The DIVA library provides 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 [23]). Our implementations are based on the *access tree strategy* introduced in [27] 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 shared data object should be created?
- On which memory modules should these copies be placed?
- Which requests should be served by which copies?
- How should the copies of a shared data object be located?

The access tree strategy is based on a hierarchical decomposition of the network. This decomposition allows us to exploit topological locality in an efficient way. We describe recursively the hierarchical decomposition of a 2-dimensional mesh M . 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* $T(M)$, in which each node corresponds to one of the submeshes, i.e., the root of $T(M)$ corresponds to M itself, and the children of a node v in the tree correspond to the two submeshes into which the submesh corresponding to v is divided. In this way, the leaf nodes of $T(M)$ correspond to submeshes of size one and, thus each leaf node represents a node of M .

We interpret $T(M)$ as a virtual network that we want to simulate on M . In order to compare the congestion in both networks we define bandwidths for the edges in $T(M)$, i.e., for a tree edge $e = (u, v)$, with u denoting the parent node, define the

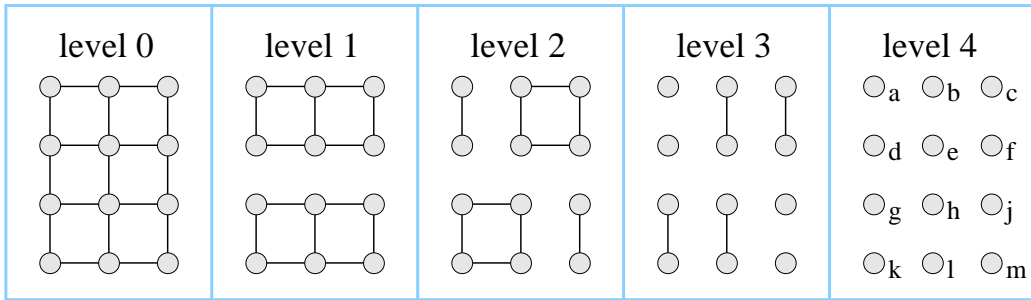


Figure 1: The partitions of $M(4,3)$.

bandwidth of e to be the number of edges leaving the submesh corresponding to v . Note that this bandwidth definition is only required for the description of the theoretical results and does not affect the access tree strategy in any way. Figure 2 gives an example of a decomposition tree with bandwidths.

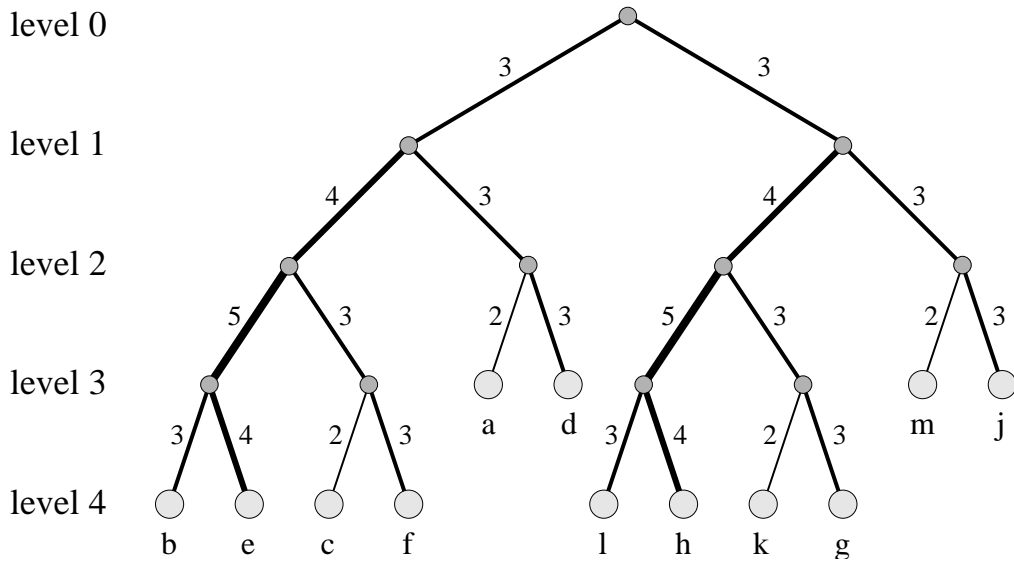


Figure 2: The decomposition tree $T(M(4,3))$. The node labels correspond to the labels in Figure 1. The edge labels indicate the bandwidths of the respective edges.

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 .

It remains to describe how the nodes locate the copies. As the nodes holding the copies of x always build a connected component, the data tracking problem can be solved very easily. A signpost pointing to the node that has issued the most recent write request is attached to each node. (Initially, this signpost points to the only copy of x .) Whenever x is updated the signposts are redirected to the node that has issued the corresponding write request. Note that this mechanism does not require extra communication, because only the signposts on nodes involved in the invalidation multicast have to be redirected. The number of signposts can be reduced by defining a root of the tree and omitting all those signposts that are directed to the root. Hence, we need signposts only to mark the trail from the root to the node that issued the least recent write request. Besides we attach markers to the copies that indicate the boundaries of the connected component built by the nodes that hold a copy. These markers are needed for the invalidation multicast. They are updated whenever the connected component changes. Also this mechanism does not require extra communication.

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

The following theoretical result shows that the access tree strategy achieves small congestion. Note that the bounds on the congestion hold with high probability

(w.h.p.), i.e., with probability $1 - n^{-\alpha}$, where n denotes the number of processors in the network, and α is an arbitrary constant.

Theorem 1 *For any application on the mesh M of dimension d with n nodes, the access tree strategy achieves congestion $O(C_{\text{opt}}(M) \cdot d \cdot \log n)$, w.h.p., where $C_{\text{opt}}(M)$ denotes the optimal congestion for the application.*

This result is optimal for constant d since any on-line routing algorithm on meshes has a competitive ratio $\Omega(\log n/d)$ [27]. **Theorem 1** is proven by a series of three lemmata, whose proofs can be found in [27]. The first lemma compares the mesh and the decomposition tree.

Lemma 2 *Any application that produces congestion C when it is executed on the mesh M can be executed on the decomposition tree T with congestion C , too.*

The lemma is shown using a straightforward simulation of the mesh M by the decomposition tree $T(M)$. Observe that we have defined the bandwidths of the edges in $T(M)$ in such a way that $T(M)$ covers the routing capabilities of M . The next lemma shows that the tree strategy used on the access trees achieves very small congestion.

Lemma 3 *The tree strategy minimizes the congestion on the decomposition tree up to a factor of 3.*

The third lemma shows that the simulation of the decomposition tree on the mesh via the randomized but locality preserving embedding of access trees into the mesh results in a very small congestion, too.

Lemma 4 *Suppose an application produces congestion C when it is executed on the decomposition tree $T(M)$. Then the simulation of the execution on $T(M)$ by M produces congestion $O(C \cdot d \cdot \log n)$, w.h.p.*

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 flatter 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 i th row and j th column of M' . Then v is mapped to the node in row $i \bmod m_1$ and column $j \bmod m_2$ 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 of the random experiments regarding the load on the edges. 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 [42]. 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.

The implemented barrier synchronizations allow to synchronize arbitrary *groups*, i.e., subsets of the processors. The groups can be generated dynamically at run time. With each group, we associate a randomly embedded access tree. Note that the dynamic generation of access trees at run time can be done without any additional communication cost.

A call for a barrier synchronization sends a message upwards in the access tree. Each node of the access tree waits until it receives a synchronization message from each of its children that represent a submesh including at least one processor of the group. After receiving all these messages the access tree node either sends a synchronization message to its parent node, or, if the node represents the smallest submesh that includes all members of the group, it initiates a multicast downwards along the branches of the access tree notifying all group members about the completion of the barrier.

The degree of the access trees for the synchronization messages correspond to the one that is used for the access trees of the global variables with one exception. For the barrier synchronizations of the fixed home strategy, we use 4-ary access trees since P -ary trees obviously perform very badly for large meshes.

The implemented locks are attached directly to global variables. A call for a lock deletes all of the variable's copies and creates a new exclusive copy on the processor that has requested the lock. All subsequent read, write, or lock requests of other processors are blocked as soon as they reach the node holding the exclusive copy. If the lock is released then the work of the blocked requests is continued as usual. Note that in case the degree of the access tree is relatively small, the processor holding the locked copy does not become a bottleneck even if all other

processors try to access the locked variable because the data management strategy running on each access tree guarantees that at most one request for a variable is blocked at each endpoint of a tree edge.

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 [39, 48].

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 why 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 by the Parsytec GCel, in which the 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 ($\hat{=}$ 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 3](#) 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, since each level in the 2-ary subtree causes an expected total communication load of $\Theta(m \cdot \sqrt{P})$ in this case.

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

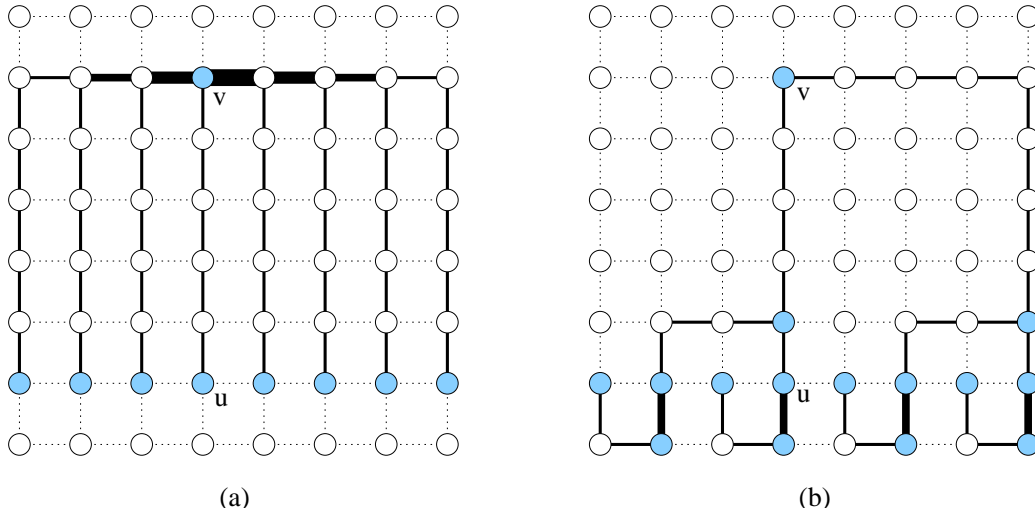


Figure 3: The pictures show the data flow induced by distributing the copies for a single data block in a row of the mesh during the read phase. Picture (a) shows the data flow for the fixed home strategy. Picture (b) shows the data flow for the access tree strategy. The width of a line represents the 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. Note that the embedding of an access tree is determined in our implementation as soon as the root is mapped into the mesh. 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.

fixed home 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 [27].

Experimental results

First, the different data management strategies are compared for a fixed number 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 4 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.

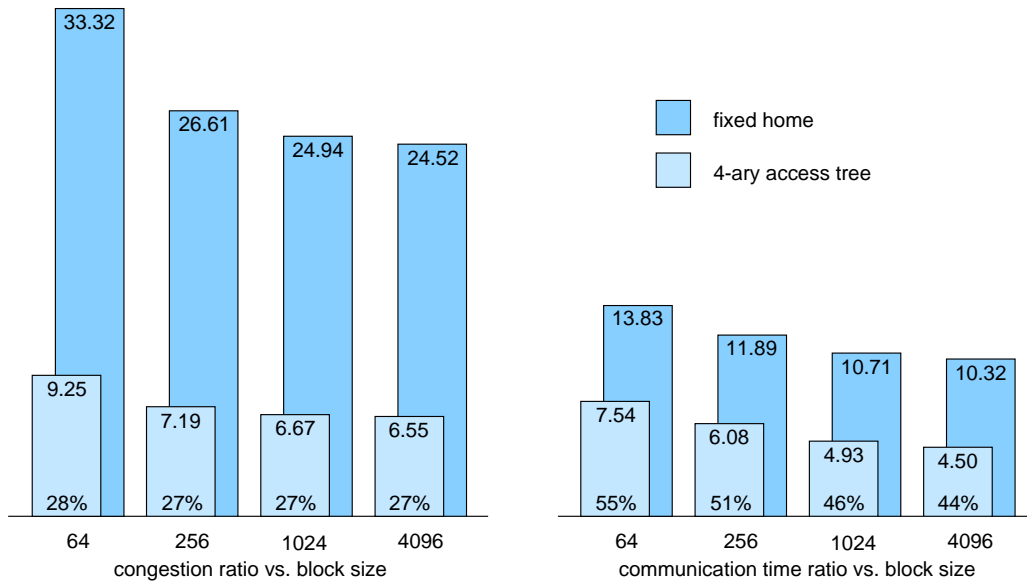


Figure 4: The graphics represent measured congestion and execution time for matrix multiplication on a 16×16 mesh, described as the ratio to the respective values of the hand-optimized strategy.

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 time spent in 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 neighboring 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.

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 5 illustrates the results. The congestion of the hand-optimized strategy grows linearly in the side length of the mesh. The 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.

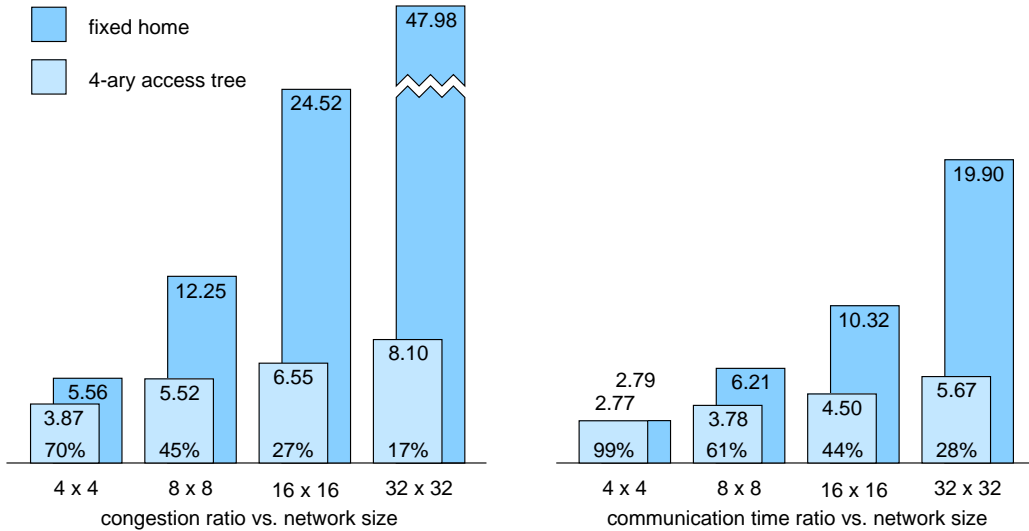


Figure 5: The graphics represent measured congestion and communication time for the matrix multiplication with a fixed block size of 4096, described as the ratio to the respective values of the hand-optimized strategy.

3.2 Bitonic sorting

We have implemented a variant of Batcher’s bitonic sorting algorithm [9]. Our implementation is based on a sorting circuit. Figure 6 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.

The bitonic sorting circuit described in Figure 6 is a slight variant of a circuit introduced by Knuth [21], which, in contrast to Knuth’s proposal, only uses *standard comparators*, i.e., $a \leq b$ for each comparator $[a : b]$. 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 [11, 16, 18, 44].

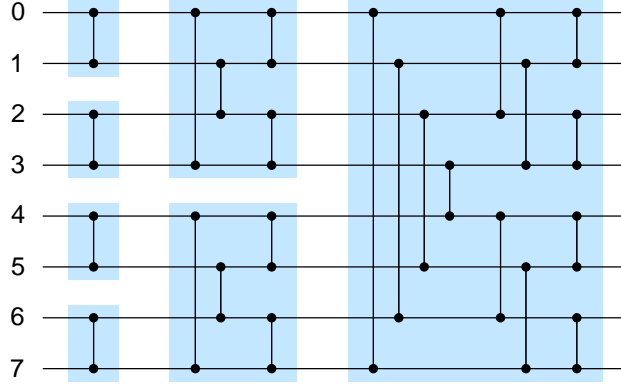


Figure 6: The bitonic sorting circuit for $P = 8$.

For simplicity, we assume that the size of the mesh is $\sqrt{P} \times \sqrt{P}$ and that $\sqrt{P} = 2^d$ with d denoting an arbitrary integer. The processor in row $x = x_{d-1} \cdots x_0$ and column $y = y_{d-1} \cdots y_0$ is assigned the unique ident-number $\text{pid} = x_{d-1}y_{d-1} \cdots x_2y_2x_1y_1x_0y_0$. Figure 7 gives an example of the ident-numbers for $P = 16$ processors. This assignment numbers the leaves of each access tree from left to right.

Initially, each processor is assigned a global variable $M[\text{pid}]$, which includes “its” m keys. We assume that variable $M[\text{pid}]$ has been initialized by processor pid such that the only copy of this variable is in the cache of this processor. Note that the sorting algorithm leaves the copies in the same configuration, too. Hence, we consider a situation in which the sorting algorithm is used repeatedly as a subroutine in another parallel program.

At the beginning of the algorithm each processor pid reads “its” keys from the global variable $M[\text{pid}]$ and sorts them locally with a fast algorithm, i.e., the quicksort routine from [37]. Then the sorted list of keys is written back to the variable $M[\text{pid}]$. Afterwards, the program in Figure 8 is executed by all processors in parallel.

It remains to describe the merge&split routine. Suppose that processor pid calls in step j of phase i $\text{merge\&split}(\text{pid}_1, \text{pid}_2, \text{pos})$. Then processor pid executes the following operations. At first, the processor issues a barrier synchronization call. Then processor pid reads the keys from $M[\text{pid}_1]$ and $M[\text{pid}_2]$ and issues another barrier synchronization call. Afterwards, the processor computes its new keys, that is, if $\text{pos} = \text{low}$ then it computes the m lower keys among the keys from $M[\text{pid}_1]$ and $M[\text{pid}_2]$, and if $\text{pos} = \text{high}$ then it computes the m upper keys among these

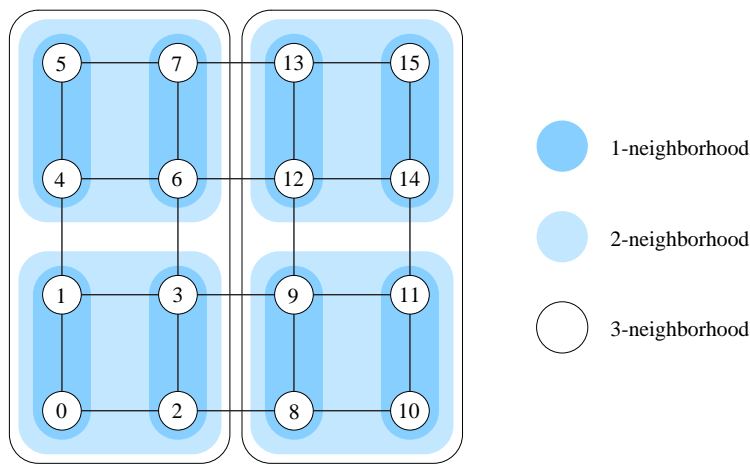


Figure 7: The ident-numbers and neighborhood sets for $P = 16$ processors.

keys. Finally, processor pid writes these keys in a sorted order to $M[\text{pid}]$. After all processors have executed the above program, the keys can be found in a sorted order in the global variables $M[0], \dots, M[P - 1]$.

Locality in the algorithm

The bitonic sorting 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 6. 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. This is explained in the following.

For a processor pid and an integer k with $0 \leq k \leq \log P$, we define the k -neighborhood of pid to include all processors whose highest $\log P - k$ bits of the ident-number are equivalent to the corresponding bits of pid . Figure 7 gives an example of the neighborhood sets for $P = 16$ processors. The neighborhood sets reflect the locality according to the access tree topology, that is, two processors that are included in the k -neighborhood of each other have distance smaller than or equal to k in the 4-ary access tree. All wires of a merging circuit of phase i are represented by processors that belong to the same i -neighborhood. In addition to the locality in the arrangement of the merging circuit, locality can be found inside each merging circuit, that is, any pair of processors that communicate in step j of

```

FOR  $i := 1$  TO  $\log P$  DO
   $\delta := 2^i$ ;
  buddy :=  $(\text{pid} \text{ div } \delta + 1) \cdot \delta - \text{pid} \bmod \delta - 1$ ;
  IF  $\text{pid} \bmod \delta < \delta/2$  THEN
    merge&split(pid, buddy, low)           /* phase  $i$ , step 1 */
  ELSE
    merge&split(pid, buddy, high);        /* phase  $i$ , step 1 */
  FOR  $j := 2$  TO  $i$  DO
     $\kappa := 2^{i-j}$ ;
    IF  $\text{pid} \bmod (2 \cdot \kappa) < \kappa$  THEN
      merge&split(pid, pid +  $\kappa$ , low)    /* phase  $i$ , step  $j$  */
    ELSE
      merge&split(pid, pid -  $\kappa$ , high); /* phase  $i$ , step  $j$  */

```

Figure 8: The bitonic sorting algorithm.

phase i are included in the same $(i - j + 1)$ -neighborhood. (In order to exploit this locality for the barrier synchronizations, too, we only synchronize subgroups of processors in the merge&split steps, that is, in step j of phase i a processor only synchronizes with the processors in its $(i - j + 1)$ -neighborhood.) We will show that the access tree strategy takes great advantage from the locality whereas the fixed home strategy does not take any advantage from this locality (apart from the local barrier synchronizations, which we also use for this strategy).

First, we consider the fixed home strategy. In each step, each processor pid reads a variable that is stored in its own cache and a variable that is stored in the cache of another processor. The first read access is served locally, whereas the second one induces the following communication overhead. Processor pid has to send a request message to the fixed home of the variable, the fixed home sends a request message to the processor holding the copy, this processor sends the requested data to the fixed home, and finally the fixed home sends the data to processor pid . The write access at the end of the step is served in a similar fashion but only requires the sending of small invalidation messages. It is easy to check that the average load per edge induced by the fixed home strategy in a step as described above is $\Theta(m \cdot \sqrt{P})$. Thus, the average load over all steps is $\Theta(m \cdot \sqrt{P} \cdot \log^2 P)$, which deviates by a factor of $\Theta(\log^2 P)$ from the minimal congestion of $\Theta(m \cdot \sqrt{P})$ that can be obtained for the given embedding of the bitonic sorting circuit.

When the 2-ary access tree strategy is used then most parallel merge&split steps take place in small, independent submeshes. The communication of a processor in step j of phase i only uses edges in the smallest submesh according to the decomposition hierarchy that includes the $(i - j + 1)$ -neighborhood of processor pid . The side lengths of this submesh are about $2^{(i-j+1)/2}$ such that the length of the routing path from processor pid to its merge&split partner is in $O(2^{(i-j)/2})$. Hence, the average communication load per edge in step j of phase i is $O(m \cdot 2^{(i-j)/2})$. Summing up over all these steps yields that the average load over all steps is

$$\sum_{i=1}^{\log P} \sum_{j=1}^i O\left(m \cdot 2^{(i-j)/2}\right) = O\left(m \cdot \sqrt{P}\right) . \quad (1)$$

If we assume that the 2-ary access tree strategy distributes the load relatively evenly among all edges, which is a reasonable assumption due to the randomized embedding of the access trees, then it achieves asymptotically optimal congestion. The same asymptotic results hold also for access trees with other constant degrees. However, the 2-ary access tree is the most promising as the 2-ary mesh decomposition corresponds exactly to the locality of the bitonic sorting circuit. The following experimental results will give an insight into how large the constant terms in the congestion bounds are and what the impact of the congestion on the execution time is.

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.)

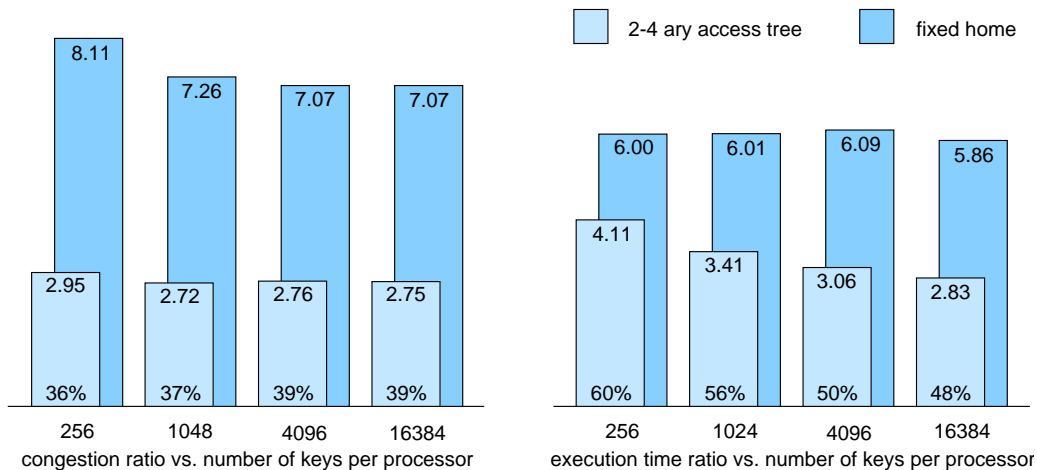


Figure 9: The graphics represent measured congestion and execution time for bitonic sorting on a 16×16 mesh, described as the ratio to the respective values of the hand-optimized strategy.

Figure 9 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.

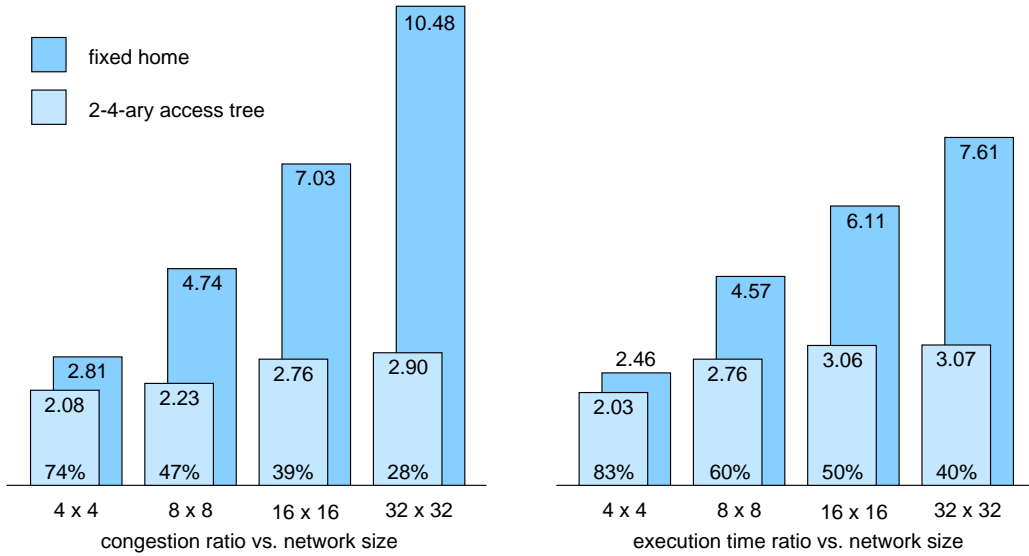


Figure 10: The graphics represent measured congestion and communication time for the bitonic sorting with 4096 keys per processor, described as the ratio to the respective values of the hand-optimized strategy.

Figure 10 illustrates the behavior of the strategies when scaling over the number of processors. The 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 $O(1)$. 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 the geometric sum given in Equation 1. Hence, we can expect that the congestion ratio of 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 [8] from the SPLASH II benchmark [39, 48], 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 in which the internal nodes are cells and the 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. A cell is determined to be far enough away if the following condition is satisfied:

$$d \cdot \vartheta > \ell ,$$

where ℓ is the length of a side of the cell, d is the distance from the body to that cell’s center, and ϑ is a user-defined accuracy parameter (ϑ is usually between 0.3 and 1.2, see [38]). 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. The variables for the bodies and the cells have a size of 88 and 194 bytes, respectively. Among other information, every cell has pointers to its children.

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 in 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 [39]). 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*: The Barnes-Hut tree is conceptually laid out in a two-dimensional plane, with a cell's children laid out from left to right in increasing order of child number. The cost of every body is profiled and stored with it; it corresponds to the number of bodies and cells that had to be opened for the force calculation in the last time step. (This method

works well because the system evolves slowly and the body distribution does not change too much between successive time steps.) As a result of Phase 2, a cell stores the total work associated with all the bodies it contains. The total work in the system is divided among processors so that every processor has a contiguous, equal range or *zone* of work. For example, a total work of 1000 units would be split among 10 processors so that zone 1–100 units is assigned to the first processor, zone 101–200 units to the second, and so on. Which costzone a body in the tree belongs to is determined by the total cost of an inorder traversal of the tree up to that body. In Phase 3, the processors traverse the tree in parallel and pick up the bodies that belong in their costzone. Hence, each processor is responsible for a contiguous interval of leaves of the Barnes-Hut tree.

Locality in the application

The costzones partitioning 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 inter-processor 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.

The partitions produced by the costzones partitioning scheme are contiguous in the tree, which, however, does not imply that the partitions are also spatially contiguous in the physical space. Sometimes a processor is assigned two or more physical regions. Besides these regions are not equally sized in all directions. However, if the number of bodies is much larger than the number of cells then the processors mostly pick up large cells from upper levels of the tree such that enough locality is obtained within each of these cells.

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 processors 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 GCell. We use the scaling rule proposed in [38]: If the number of bodies N is scaled by a factor of s , the time step duration Δt and the accuracy parameter ϑ must each be scaled by a factor of $1/\sqrt[4]{s}$. The idea behind these scaling rules is that all sources of error should be scaled so that their error contribution are about equal. We set

$$\Delta t = 0.25/\sqrt[4]{N} \text{ and } \vartheta = 7.2/\sqrt[4]{N} .$$

Hence, Δt ranges between 0.025 for 10,000 bodies and 0.016 for 60,000 bodies, ϑ ranges between 0.72 and 0.46. The decrease of Δt leads to an increase in the execution time as the number of time steps has to be increased in order to simulate a given period of time. As we will run simulations over a fixed number of time steps, the change of Δt does not have a great influence on our results. The decrease of ϑ , however, directly influences the execution time needed for each individual time step. Physicists usually simulate hundreds or thousand of time steps. The execution times for the individual time steps are already relatively stable after the simulation of the first two steps, because the load balancing is very fast well tuned. Hence, we only simulate 7 time steps, from which only the last 5 are included in the measurement.

Figure 11 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. The slightly super-linear increase of the execution time is due to the additional scaling of the accuracy parameter ϑ as described above, which, however, mainly influences the local computation time, only.

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

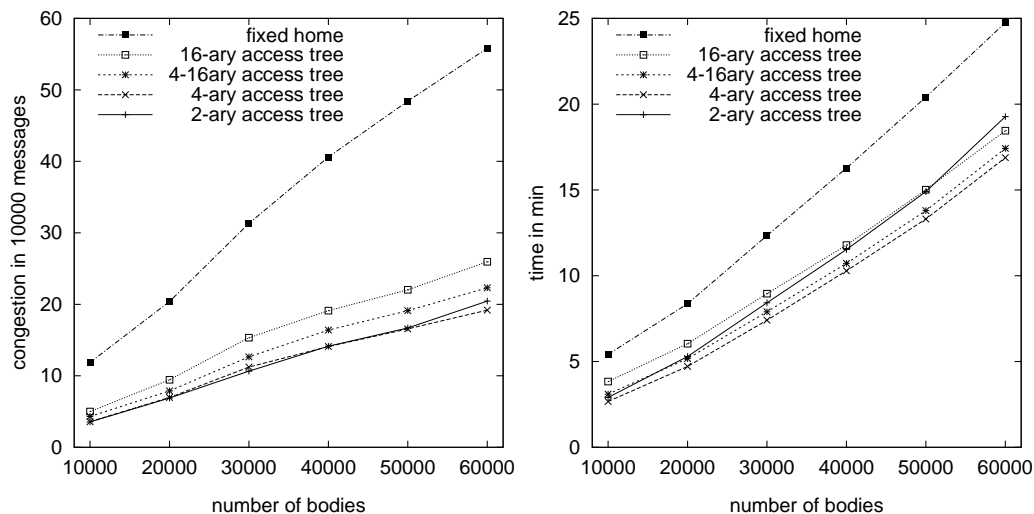


Figure 11: Congestion and execution time for the Barnes-Hut N -body simulation on a 16×16 mesh.

slower than the 4-ary although it has a very small congestion. As a result, the 4-ary strategy performs best among all strategies.

The order of the different access tree strategies concerning the execution time changes slightly if we run the N -body simulation on meshes that cannot be partitioned properly by a 4-ary decomposition. For example, the decomposition of a 12×16 mesh yields 1×2 submeshes on the decomposition level before final whereas the 4-16-ary decomposition ends with submeshes of size 3×4 . Since submeshes of size 1×2 produce more overhead for additional startups than saving for congestion, the 4-16-ary access tree strategy outperforms the 4-ary strategy slightly on the 12×16 mesh.

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 phase in further detail.

Figure 12 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

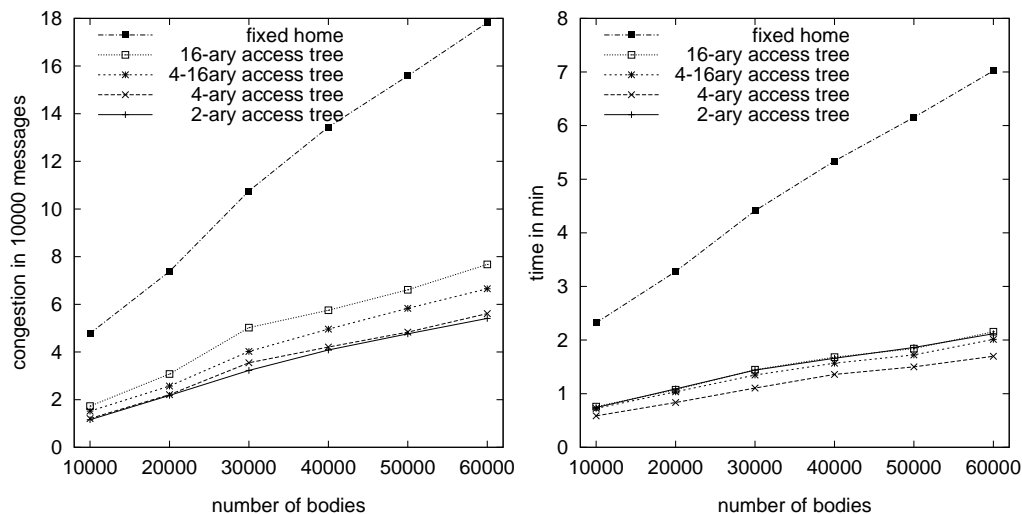


Figure 12: Congestion and execution time of the tree building phase on a 16×16 mesh.

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.

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.

What, however, is the reason for the steeper slope of the fixed home strategy's congestion and execution time? – Interestingly, the difference in the slope does not

occur in the simulation of the first time step in which the Barnes-Hut tree is built for the very first time. In the later time steps, each write access to a cell leads to the invalidation of all copies of the cell that has been created in the previous time step. The access tree strategy performs this invalidation more efficiently than the fixed home strategy since it uses multicast trees. A larger number of bodies leads to a larger number of cells whose copies need to be invalidated in the tree building phase, which explains the steeper slope of the fixed home strategy. Most of the copies to be invalidated are created in the force computation phase. This phase is considered next.

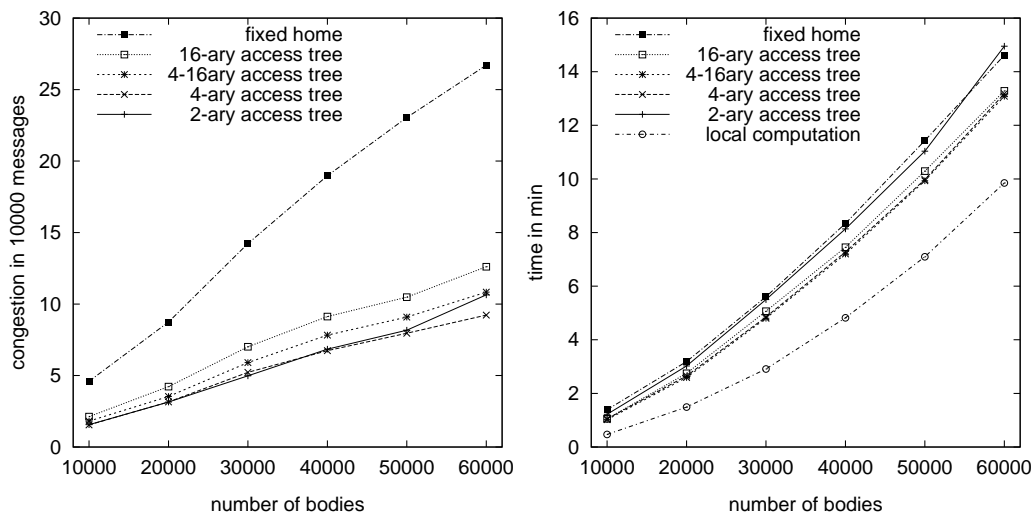


Figure 13: Congestion and execution time of the force computation phase on a 16×16 mesh.

Figure 13 depicts the congestion and the execution time of the force computation phase added over 5 time steps. The picture also 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. These copies have to be invalidated in consecutive phases like the tree building phase of the next step. (The time for the local computation has been measured by running the force computation phase twice, and measuring only the time needed for the second execution in which all required copies are already stored in the local caches.)

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, where the width of the neighborhood is defined by the parameter ϑ . 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.

Decreasing ϑ means that the neighborhood of a body's path grows. The curve for the local computation time is super-linear since we decrease ϑ with the number of bodies. The congestion, however, does not increase noticeable in a super-linear fashion as it only grows with the number of cache misses in this phase. This number is determined by the size of the union of the neighborhoods of the paths for all bodies computed by a processor. Interestingly, this union does not seem to grow in a super-linear fashion as the congestion does not either.

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 processors 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 $N = 200 \cdot P$. For simplicity, we use a naive scaling, i.e., we set $\vartheta = 2$, and $\Delta t = 0.025$, rather than decreasing this parameters by a factor of $\sqrt[4]{N}$.

Figure 14 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.

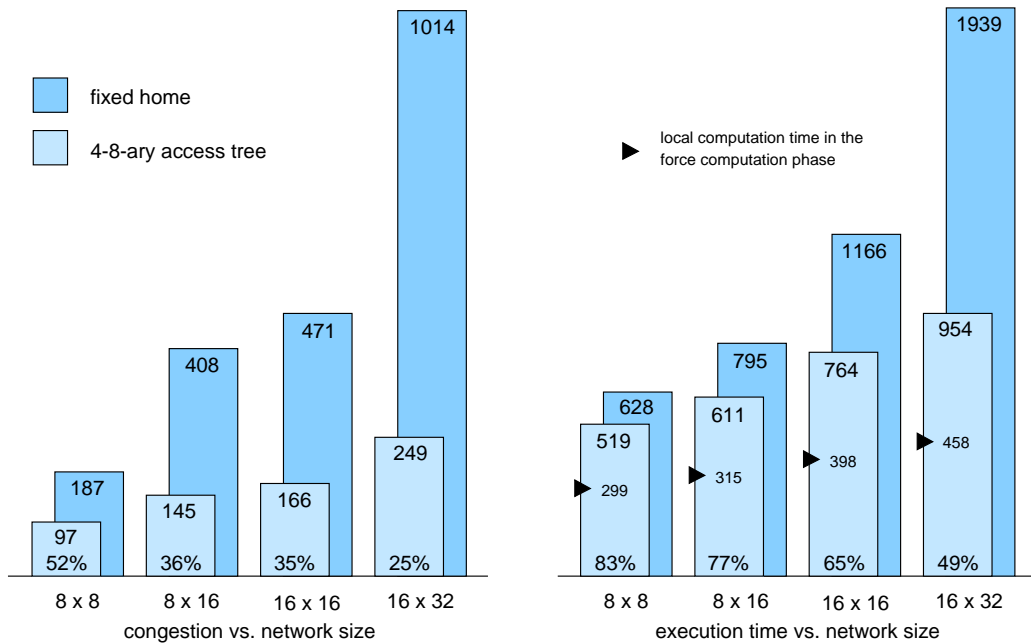


Figure 14: The graphic represents measured values for the Barnes-Hut N -body simulation for different numbers of processors P . The number of bodies is increased with the number of processors, that is, $N = 200 \cdot P$. The congestion is measured in units of 1000 messages. The time is measured in seconds.

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 on three standard applications of parallel computing. In all experiments for each of these applications, the access tree strategy clearly outperformed a standard strategy using fixed homes as it produces significantly less congestion.

Of course, the access tree strategy cannot win against hand-optimized message passing strategies for oblivious applications like matrix multiplication or bitonic sorting that can use full knowledge of the access pattern. Nevertheless, the comparisons with these strategies were very illuminating. Without using any knowledge of the application, the access tree strategy achieves execution times that are reasonably close to those of the hand-optimized message passing strategies. This shows that

the access tree strategy tends to adapt automatically the locality inherent in an application.

The more relevant applications for a dynamic data management service, however, are non-oblivious as, e.g., the Barnes-Hut N -body simulation. Since there exists no “optimal off-line strategy” for these applications, we compared the access tree strategy only with the fixed home strategy. As the network grows, the access tree strategy becomes more and more superior. One reason for the efficiency of the access tree strategy is its simplicity. Another reason is its data tracking mechanism that produces only very small overhead for bookkeeping. We believe, however, that the small congestion produced by the access tree strategy is the most important reason for its efficiency.

Our experimental results show that, apart from the congestion, the startup costs 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 these trees become flatter. On the positive side, this decreases the number of startups required to serve the requests. On the negative side, however, a higher degree may increase the congestion. For 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 costs 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 be efficiently adapted to other networks, too. Some examples are given in [27, 32, 45], including Internet-like clustered networks, fat-trees, hypercubic networks, and even completely connected networks. All that is needed for the implementation of the access tree strategy is a hierarchical network decomposition that possesses properties similar 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, K. Johnson, D. Kranz, J. Kubiawicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum. The MIT Alewife machine: A large-

- scale distributed-memory multiprocessor. In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, New York, NY, USA, 1991.
- [2] C. Amza, A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. ThreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 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(1):67–104, 1998.
 - [7] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 503–513, 1990.
 - [8] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324(4):446–449, 1986.
 - [9] K. E. Batcher. Sorting networks and their applications. In "AFIPS" *Conference Proceedings 32*, pages 307–314, 1968.
 - [10] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON)*, pages 528–537, 1993.
 - [11] 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.

- [12] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, MA, USA, 1992.
- [13] R. Cabrera Dantart, I. Demeure, P. Meunier, and V. Bartro. Phosphorus: A tool for shared memory management in a distributed environment. Technical Report 95D003, École nationale supérieure des télécommunications, Paris, France, 1995.
- [14] 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 (SOSP)*, pages 152–164, 1991.
- [15] 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.
- [16] D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauer. Fast parallel sorting under LogP. In *Portability and Performance for Parallel Processing*, chapter 4, pages 71–98. John Wiley & Sons, Inc., New York, NY, USA, 1994.
- [17] 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.
- [18] 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.
- [19] B. D. Fleisch, R. L. Hyde, and N. C. Juul. Mirage+: A kernel implementation of distributed shared memory for a network of personal computers. *Software—Practice and Experience*, 24(10):887–909, 1994.
- [20] 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, Norwell, MA, USA, 1990.

- [21] D. E. Knuth. *The Art of Computer Programming, Volume 3*. Addison–Wesley Longman Publishing Co., Redwood City, CA, USA, 1998.
- [22] R. R. Koch, F. T. Leighton, B. M. Maggs, S. B. Rao, A. L. Rosenberg, and E. J. Schwabe. Work-preserving emulations of fixed-connection networks. *Journal of the ACM*, 44(1):104–147, 1997.
- [23] 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.
- [24] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17(1):157–205, 1994.
- [25] D. E. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford University, 1991.
- [26] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.
- [27] 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.
- [28] F. Meyer auf der Heide. Efficiency of universal parallel computers. *Acta Informatica*, 19(3):269–296, 1983.
- [29] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15(1):106–119, 1986.
- [30] F. Meyer auf der Heide and B. Vöcking. A packet routing protocol for arbitrary networks. In *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 291–302, 1995.
- [31] F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Provably good and practical strategies for non-uniform data management in networks. In *Proceedings of the 7th European Symposium on Algorithms (ESA)*, pages 89–100, 1999.

- [32] F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Caching in networks. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 430–439, 2000.
- [33] F. Meyer auf der Heide and R. Wanka. Time-optimal simulations of networks by universal parallel computers. In *Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 120–131, 1989.
- [34] R. Ostrovsky and Y. Rabani. Universal $O(\text{congestion} + \text{dilation} + \log^{1+\epsilon} N)$ local control packet switching algorithms. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, pages 644–653, 1997.
- [35] A. Saulsbury and A. Nowatzky. Implementing simple COMA on S3.MP, 1995. Presentation at the 5th Workshop on Scalable Shared Memory Multiprocessors.
- [36] C. Scheideler and B. Vöcking. Universal continuous routing strategies. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 142–151, 1996.
- [37] R. Sedgewick. *Algorithms (2nd ed.)*. Addison–Wesley Longman Publishing Co., Boston, MA, USA, 1988.
- [38] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 26(7):42–50, 1993.
- [39] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1992.
- [40] 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.
- [41] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
- [42] A. J. van de Goor. *Computer Architecture and Design*. Addison-Wesley, 1994.

- [43] B. Vöcking. *Static and Dynamic Data Management in Networks*. PhD thesis, Universität Paderborn, Dec. 1998.
- [44] 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.
- [45] M. Westermann. *Caching in Networks: Non-Uniform Algorithms and Memory Capacity Constraints*. PhD thesis, Universität Paderborn, Nov. 2000.
- [46] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Data Base Systems*, 22(4):255–314, 1997.
- [47] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Transactions on Data Base Systems*, 16(1):181–205, 1991.
- [48] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH–2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd ACM International Symposium on Computer Architecture (ISCA)*, pages 24–36, 1995.
- [49] 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)*, pages 4–13, 1997.
- [50] 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)*, pages 272–283, 1997.