Exploiting Locality for Data Management in Systems of Limited Bandwidth

Bruce M. Maggs* Friedhelm Meyer auf der Heide** Berthold Vöcking**
Matthias Westermann**

Abstract

This paper deals with data management in computer systems in which the computing nodes are connected by a relatively sparse network. We consider the problem of placing and accessing a set of shared objects that are read and written from the nodes in the network. These objects are, e.g., global variables in a parallel program, pages or cache lines in a virtual shared memory system, shared files in a distributed file system, or pages in the World Wide Web. A data management strategy consists of a placement strategy that maps the objects (possibly dynamically and with redundancy) to the nodes, and an access strategy that describes how reads and writes are handled by the system (including the routing). We investigate static and dynamic data management strategies.

In the static model, we assume that we are given an application for which the rates of read and write acesses for all node–object pairs are known. The goal is to calculate a static placement of the objects to the nodes in the network and to specify the routing such that the network congestion is minimized. We introduce efficient algorithms that calculate optimal or close–to–optimal solutions for tree–connected networks, meshes of arbitrary dimension, and internet–like clustered networks. These algorithms take time only linear in the input size.

In the dynamic model, we assume no knowledge about the access pattern. An adversary specifies accesses at runtime. Here we devolop dynamic caching strategies that also aim to minimize the congestion on trees, meshes, and clustered networks. These strategies are investigated in a competitive model. For example, we achieve competitive ratio 3 for tree–connected networks and competitive ratio $O(d \cdot \log n)$ for d–dimensional meshes of size n. Further, we present an $\Omega(\log n/d)$ lower bound for the competitive ratio for on–line routing in meshes, which implies that the achieved upper bound on the competive ratio for meshes of constant dimension is optimal.

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 of limited bandwidth. In this paper, we consider the problem of placing and accessing shared objects that are read and written from the nodes in the network. The objects are, e.g., global variables in a parallel program, pages or cache lines in a virtual shared memory system, shared files in a distributed file system, or pages in the World Wide Web (WWW).

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 and links are the bottleneck in these systems, since 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., [7, 8]), redundant computation (see, e.g., [1, 2, 15, 21, 22, 24]) or slackness (see, e.g., [30]) the only way to bypass the bandwidth bottleneck is to reduce the communication load by exploiting locality.

The principle of locality is already known from sequential computation. Two kinds of locality are usually distinguished: temporal and spatial locality (see, e.g., [10]). *Temporal locality* means that in the near future, a program is more likely to reference those data objects that have been referenced in the recent past. This locality can be due to instruction references in program loops, or data references in working stacks. *Spatial locality* means that in the near future, a program is more likely to reference those data objects that have addresses close to the last reference. This is caused, e.g., by the traversal of data structures such as

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (email: bmm@cs.cmu.edu). Supported in part by the Air Force Materiel Command (AFMC) and ARPA under Contract F196828-93-C-0193, by ARPA Contracts F33615-93-1-1330 and N00014-95-1-1246, and by an NSF National Young Investigator Award, No. CCR-94-57766, with matching funds provided by NEC Research Institute and Sun Microsystems. This research was conducted in part while he was visiting the Heinz Nixdorf Institute, with support provided by DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen".

**Department of Mathematics and Computer Science, and Heinz Nixdorf Institute, University of Paderborn, 33095 Paderborn, Germany (email: {fmadh, voecking, marsu}@uni-paderborn.de). Supported in part by DFG-Sonderforschungsbereich 376, by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), and by DFG Leibniz Grant Me872/6-1.

arrays. A further kind of locality is specific to parallel and distributed systems, topological locality. *Topological locality* means that processors that are close together according to the topology of the interconnection network are likely to be interested in the same data objects. This locality can be due to a communication sensitive mapping of processes to processors. In this paper, we investigate data management strategies exploiting temporal and topological locality in order to increase the network utilization and therefore the performance of the system.

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

- How many copies of an object should be made?
- On which nodes should these copies be placed?

The access strategy specifies how read and write accesses are handled by the system. This includes answers to the following questions.

- How should consistency among the copies be maintained?
- How should access messages be routed through the network?

In this paper, we neglect the scheduling aspect of routing, because we want to concentrate on reducing the communication load which depends only on the routing paths. The following parameters are important measures for this communication load. For a given application, we define the *(absolute) load* of a bus or link to be the amount of data that must be transferred by this connection during the execution of the application. We define the *relative load* of a connection to be its load divided by its bandwidth. Finally, we define the *congestion* to be the maximum over the relative loads of all buses and links in the system.

Of course, it is useful to economize the total communication load, i.e., the sum over all bus and link loads. However, developing data management strategies that just aim to minimize the total load can produce communication bottlenecks in the network. Minimizing the congestion overcomes this drawback, because the congestion measure captures bottlenecks in the system. Therefore, it is a lower bound on the execution time of a given application. Moreover, several results on store–and–forward– and wormhole–routing (see, e.g., [8, 17, 23, 25, 28]) indicate that the congestion yields a good estimate for the execution time taken by coarse grained applications with high communication load. Hence, minimizing the congestion seems to be a good heuristic for efficient data management.

Our aim is to develop strategies for placing and accessing shared objects such that the locality inherent in an application is exploited in order to minimize the congestion. We model the network by a (hyper–)graph $G = (V, E)$ with node set V and edge set E such that the (hyper–)edges represent the links (or buses). The bandwidths of these connections are described by a function $b : E \longrightarrow I\!N$. The set of shared objects is denoted by X. We use two models to formalize information about the pattern of accesses to the shared objects of an application.

1.1 Description of the static model

In this model, we assume that we are given an application $\mathcal{A}$ from which the access rates from the nodes to the objects are known, i.e., the number of read or write accesses in $\mathcal{A}$ are described by the functions $h_r : V \times X \longrightarrow I\!N$ and $h_w : V \times X \longrightarrow I\!N$, respectively. We are interested in efficient algorithms calculating a static placement of the objects and specifying the routing paths such that the congestion is small.

The placement of the objects is allowed to be redundant, i.e., each object can be placed on several nodes. In order to maintain consistency among the different copies of an object, we assume that a write access to an object has to update all of its copies, whereas a read access to an object can be satisfied by one of its copies. Updates of copies are allowed to be done by multicasts, i.e., the node that wants to update a set of copies sends out one message that is transmitted along the branches of a tree to all the copies. The goal is to place the copies and to determine the routing paths (or trees in case of multicasts) in such a way that the congestion is minimized. For simplicity, we assume that the additional load due to a write or read access on each edge belonging to the respective routing path (or tree) is one. Alternatively, it is possible to weight the access rate functions h_r and h_w according to different cost measures, e.g., the h_r–values could be multiplied by some factor larger than one, because reads require to send additional request messages.

Note that the model described above is slightly restrictive in the sense that it fixes the update policy to a certain range. In particular, it does not include strategies that allow only a fraction of the copies to be updated in case of a write, which, e.g., is implemented in strategies using the majority trick introduced in [29]. However, all strategies using such techniques add time stamps to the copies. This requires that there is some definition of uniform time among different nodes. Since it is not clear how to realize this in an asynchronous setting, we initially restrict ourselves to strategies that update or invalidate all copies in case of a write. In a full version of this paper [20], we consider a more general model, including, e.g., strategies using the majority trick.

1.2 Description of the dynamic model

In the dynamic model, there is no knowledge about the access patterns of an application in advance. We assume that an adversary specifies a parallel application running on the nodes of the network, i.e., the adversary initiates read and write requests on the nodes of the network. These accesses should be served by a dynamic data management strategy. If this strategy is randomized, then the adversary is assumed to be oblivious, i.e., the accesses to shared objects are not allowed to depend on the random decisions of the data management strategy.

We restrict the class of allowed applications specified by the adversary to data–race free programs, i.e., a write access to an object is not allowed to overlap with other accesses to the same object, and there is some order among the accesses to the same object such that, for each read and write access, there is a unique last recent write. Note that this still allows arbitrary concurrent accesses to different objects, and concurrent read accesses to the same object. A consistent data management strategy ensures that a read always returns the value of the most recent write. Write accesses are assumed to be object alterations rather than overwrites. Thus, we demand that none of them is ignored, even not in the case of immediately consecutive write accesses.

We are interested in developing on–line distributed data management strategies that minimize the congestion. These strategies are allowed to migrate, create, and invalidate copies of an object during execution time. Migration means that a copy is moved along a path through the network; creation means that a copy is duplicated, and the new copy is migrated; and invalidation means that a copy is deleted. Initially, one copy of each object is placed somewhere in the network. Arbitrary communication between neighboring nodes in the network is allowed. However, each communication increases the load on the involved edge. For simplicity, we assume that each object fits into one routing packet such that each migration of a copy along an edge increases the load of this edge by one. Also request, update, and invalidation messages are assumed to have size one. In a full version of this paper [20], we discuss also more complicated models including non–uniform message sizes and slice–wise accesses to larger objects.

We use the competitive ratio as a measure for the efficiency of a dynamic data management strategy. For a given application $\mathcal{A}$, let $C_{\mathrm{opt}}^{\mathrm{dyn}}(\mathcal{A})$ denote the minimum congestion expended by an optimal dynamic strategy having full knowledge of the parallel program specified by the adversary (including knowledge of all future accesses). A dynamic strategy is said to be *k–competitive* if it achieves congestion at most $k \cdot C_{\mathrm{opt}}^{\mathrm{dyn}}(\mathcal{A})$, for any application $\mathcal{A}$. A randomized strategy has to satisfy this bound with high probability[1].

Of course, the optimal strategy has an advantage over an on–line strategy, because it has full knowledge of the dynamic access pattern. Without the restriction to data–race free programs, the optimal strategy could, e.g., defer write accesses to later time steps in order to save repeated read accesses from the same node to the same object. This illustrates that the restriction to this class of programs is necessary in order to allow a more fair comparison of on–line against off–line strategies.

1.3 Our Results

We introduce 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. The strategies for trees are deterministic, and the others are randomized. All strategies aim to minimize the congestion by exploiting locality. To our knowledge this is the first analytic treatment of this problem.

We start with developing strategies for tree–connected networks. Here we introduce an efficient algorithm that calculates an optimal placement of the objects. We show that the placement minimizes the load on any edge of the tree and, therefore, also the total load and the congestion, regardless of the bandwidths. The sequential running time of the algorithm is linear in the input size, i.e., in time $O(|X| \cdot |V|)$. Moreover, the algorithm can be efficiently calculated in a distributed fashion by the processors of the underlying tree–network. We also develop a 3–competitive dynamic caching strategy. Note that both results are applicable to trees with arbitrary bus connections having non–uniform bandwidths. Hence, they are well suited, e.g., for Ethernet connected NOWs.

The situation on mesh–connected networks is much more complicated than the one on trees, because here there are several possible routing paths between every pair of nodes. In fact, we show by a straightforward reduction from PARTITION that the static problem is NP–hard already on a 3×3 mesh. The dynamic problem is even more complicated, since we have to solve an on–line distributed routing problem, including multicasting and data tracking. We describe a simulation strategy that solves the static and the dynamic problem simultaneously. The strategy is based on a randomized but locality preserving embedding of access trees into the mesh. Hence, it is called the *access tree strategy.* On the access trees, the static or dynamic strategy for trees is simulated. Consider an arbitrary mesh of size n and dimension d. Here the access tree strategy yields an efficient algorithm for static data placement on arbitrary meshes achieving optimal congestion up to an $O(d \cdot \log n)$ factor, w.h.p., that can be calculated sequentially in time linear in the input size. In the dynamic model, the access tree strategy achieves competitive ratio $O(d \cdot \log n)$. We give a corresponding $\Omega(\log n/d)$ lower bound for the competitive ratio for on–line routing in these networks, which implies that the upper bound is optimal for meshes of constant dimension.

Finally, we investigate internet–like clustered NOWs. We show that the access tree strategy is also suitable for this kind of topology. In particular, we show that a static placement with close–to–optimal congestion can be calculated in linear time. Further, the access tree can be used on these networks for efficient dynamic data management, e.g., for WWW pages. The characteristics of this topology and the results for it are described in more detail in Section 4.

In a full version of this paper [20], we show that most of the static results hold even in more general models capturing, e.g., the majority trick. Further, we show that the dynamic access tree strategy can be extended to handle non–uniform object sizes and slice–wise access to large objects. Due to space limitations also the proof for the NP–hardness of the static problem and the lower bound for on–line routing are shifted to the full version.

1.4 Related Work

The problem of distributing and accessing shared objects in networks is investigated in several papers. We give a brief overview among the most important or related work in this area.

The first approaches to solve the problem concentrated on modeling it by mixed integer programs and solving these programs efficiently by using heuristics. Here several models with different cost functions and constraints have been developed. The 1981 survey paper by Dowdy and Foster [9] gives an overview of this work.

Most theoretical work in the area of distributed data management concerns PRAM simulations. In [13], Karlin and Upfal present a probabilistic emulation on an N–node butterfly. Their algorithm emulates one step of an N–processor EREW PRAM in time $\Theta(\log N)$, w.h.p.. Ranade [27] showed how combining could be used to improve this result, i.e., he showed that a CRCW PRAM step can be emulated in the same time. Both strategies use random hash functions to distribute the memory cells uniformly among the processors. This scheme can also be adapted to other networks. For instance, this yields N–processor PRAM sim-

[1] Throughout the paper, the terms "*congestion C, with high probability*" or "*congestion C, w.h.p.*" mean "congestion $C + x$, with probability at least $1 - 2^{-\Omega(x)}$". Note that this induces that the expected congestion is $C + O(1)$.

ulations for the $\sqrt{N} \times \sqrt{N}$ mesh with slowdown $\Theta(\sqrt{N})$. Although this bound cannot be improved for general PRAM simulations, it is not satisfactory for applications including locality.

Awerbuch et al. investigate dynamic data management in arbitrary networks. In [4] they present a dynamic caching strategy that minimizes the total communication load up to a polylogarithmic factor. In [5] they adapt their scheme to systems with limited memory capacities. However, their strategies uses the concept of a global leader that eventually is involved in nearly any access issued by one of the processors. This shows the importance of considering the congestion rather than the total communication load.

Better results are known for dynamic data management on tree–connected networks. Note that minimizing the total load in trees also minimizes the congestion. Here Bartal et al. [6] describe a randomized $\Theta(1)$–competitive dynamic strategy for trees, and Lund et al. [19] describe a deterministic but centralized strategy with same competitive ratio. (The competitive ratio given in both papers is 3. The ratio for the distributed strategy increases by a small factor if it is "translated" to our model that counts also request and invalidation messages.) Note that both strategies allow to access slices of objects, which cannot be handled by our dynamic caching strategy, which, however, is deterministic and distributed.

Several recent papers deal with the distribution of pages in the WWW. Plaxton and Rajaraman [26] show how to balance the pages among several caches by embedding a random cache tree for each page into the network. This balances the load well and ensures fast responses even for popular pages. Karger et al. [12] use a similar technique to relieve hot spots in the WWW. Note that the technique of embedding a random tree for each object is similar to our access tree strategy. The main differences to our approach are the following. The strategy in [26] uses a uniform embedding of the tree nodes onto the nodes in the Internet, which completely dissolves topological locality. The strategy in [12] pays attention to topological locality. In fact, they use a model similar to our Internet model. However, they consider the latencies instead of the bandwidths to be the main problem for data transmission in the Internet. Further, they consider only read accesses to the pages.

2 Data management on trees

In this section, we describe static and dynamic data management strategies for tree–connected networks. For simplicity, we assume that all connections in the network are point–to–point connections, i.e., links. These links are allowed to have arbitrary bandwidth.

Note, however, that all algorithms described in this section can also be applied to networks with more complicate bus connections. This is described in a full version of this paper [20]. In the following, the network is modeled by a graph $T = (V, E)$. Let $\mathrm{diam}(T)$ denote the diameter of this graph and $\mathrm{degree}(T)$ its maximum node degree.

The advantage of trees is that there is only one simple path between any pair of nodes. Thus, the placement strategy automatically defines the routing paths from the accessing processor to the respective copies. In particular, this means that the congestion is fixed as soon as the placement is specified. This make the analysis for trees much easier than the one for networks including cycles.

2.1 Static placement

We assume that we have some knowledge about the access frequencies from the nodes to the objects, i.e., we are given two functions $h_r : V \times X \longrightarrow I\!N$ and $h_w : V \times X \longrightarrow I\!N$ that describe the rate of read or write accesses, respectively, from the nodes in V to the objects in X. Each object can be placed statically in one or more nodes. A read access to an object can be satisfied by one of its copies, and a write access has to update all copies.

The following strategy, which we call the *nibble strategy*, places each object $x \in X$ independently from the other objects. Fix an object x. We use the following notations and definitions concerning the access rates to x. For a node $v \in V$, define $r(v) = h_r(v, x)$, $w(v) = h_w(v, x)$, and $h(v) = r(v) + w(v)$, and for a subtree $T' = (V', E')$, define $r(T') = \sum_{v \in V'} r(v)$, $w(T') = \sum_{v \in V'} w(v)$, and $h(T') = r(T') + w(T')$.

The nibble strategy places copies of x on some nodes that form a connected component including the *gravity center of* T. This gravity center depends on the access rates for x. Hence, each object has its own gravity center in the tree. The gravity center of our fixed object x is denoted by $g(T)$ and it is defined as follows. For $v \in V$, let $\mathcal{T}(v)$ denote the set of subtrees in which T is partitioned if v is removed from it. Consider the set of nodes v that satisfy the following condition:

$$\forall T' = (V', E') \in \mathcal{T}(v) : h(T') \leq h(T)/2 \ .$$

It is easy to check that this set is not empty. Suppose each node knows the $h(T')$ values. Then it can locally decide whether or not it belongs to the above set. We choose one arbitrary node from this set to be the gravity center $g(T)$, e.g., the one with the smallest index.

In the following, $g(T)$ is assumed to be the root of the tree T, which defines the parent and the children of each node. The subtree $T(v)$ rooted with $v \in V$ is defined to be the maximal subtree including v but not the parent of v. After we have fixed all these notations, the rest of the description of the nibble strategy is very simple:

> A node v gets a copy if and only if $h(T(v)) > w(T)$ or $v = g(T)$.

The nibble strategy can be calculated in time linear to the input size, which is $O(|V|)$ for each object. The most difficult part is to compute the $h(T')$ values for each node (with T' denoting the subtrees in which the tree is partioned when the node is removed). However, this can be done by a depth first search algorithm taking time $O(|E|) = O(|V|)$, for each object. Moreover, the placement can be calculated easily by the processors of the tree network. Here the $h(T')$ values can be computed in $2 \cdot \mathrm{diam}(T)$ rounds each of which takes time $O(\mathrm{degree}(T))$. Further, the computation for several objects can be pipelined, which gives time $O((|X| + \mathrm{diam}(T)) \cdot \mathrm{degree}(T))$ for the placement of all objects in X.

The following theorem shows that the nibble strategy calculates a mapping which yields minimum load on each edge. Of course, this also proofs that it minimizes the total load and the congestion regardless of the link bandwidths.

Theorem 2.1 *For any object x, the nibble placement strategy achieves minimum load for each edge of the tree.*

Proof. The nibble strategy places a copy on the gravity center for sure. The following lemma shows that this does not increase the load.

Lemma 2.2 *Any mapping S for x can be transformed into a mapping S^* without increasing the load on any edge such that the nodes that hold a copy build a connected component including $g(T)$.*

Proof. Consider the minimum Steiner tree connecting all copies of S. Let S' be the mapping in which every node of this Steiner tree holds a copy. Then transforming S to S' does not increase the load on any edge because each write access has to to update all copies of S, which means that each write access crosses each edge in the Steiner tree. If S' includes $g(T)$ then we are finished. Otherwise, we have to transform S' into S^* as follows.

The nodes holding the copies of S' build a connected component not including $g(T)$. Thus, one of the subtrees $T(v_1), T(v_2), \ldots$ includes all of the copies, where $v_1, v_2, \ldots$ denote the nodes incident to $g(T)$. Let $T_A = (V_A, E_A)$ denote this subtree, and let $T_B = (V_B, E_B)$ denote the remaining subtree, i.e., the subtree of T induced by the set of nodes in $T \backslash T_A$. Further, let $T_C = (V_C, E_C)$ denote the subtree of T_B induced by the set of the nodes holding a copy of S', let u denote the node in V_C which is closest to $g(T)$, and let P denote the simple path connecting u with $g(T)$. Figure 1 illustrates the situation.

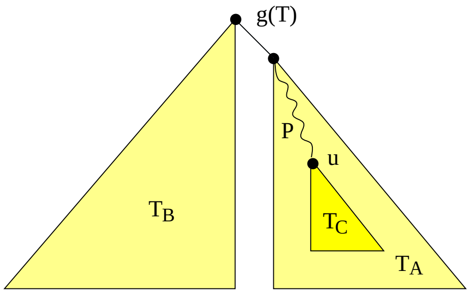

Figure 1: The subtrees T_A, T_B, and T_C, the nodes $g(T)$ and u, and the path P.

Suppose $r(T_B) > w(T_A)$. Then placing additional copies onto all the nodes of the path P influences only the load on the edges of this path. On the one hand, each write access issued in T_A has to traverse this path now. On the other hand, the read accesses issued in T_B have not to traverse this path anymore. Thus, the load on every edge on the path is increased by $w(T_A) - r(T_B) < 0$, which means that adding copies to all nodes on P yields the strategy we are looking for, S^*.

Now assume $r(T_B) \leq w(T_A)$. Because $g(T)$ is the gravity center, $h(T_A) \leq h(T)/2 \leq h(T_B)$. Combining both equations yields

$$w(T_B) = h(T_B) - r(T_B) \geq h(T_A) - w(T_A) = r(T_A) \ .$$

Let S'' denote the mapping having only one copy which is placed on u. Then transforming S' to S'' only influences the load on edges in E_C. The load on these edges is increased by at most $r(T_A)$ and decreased by at least $w(T_B)$. This gives an increase of $r(T_A) - w(T_B) \leq 0$ on these edges. Therefore, changing S' into S'' does not increase the load on any edge.

Finally, we change S'' into S^* by moving the copy from u to $g(T)$. This effects only the load on edges of P, which is increased by at most $h(T_A)$ due to accesses issued in T_A and decreased by $h(T_B) \geq h(T_A)$ due to accesses issued in T_B. As a consequence, S can be transformed via S' and S'' into S^* without increasing the load on any edge. ∎

We show that an arbitrary mapping S of the copies can be transformed into the mapping computed by the nibble strategy without increasing the load on any edge. This yields the theorem.

First, transform S into a mapping S^* such that the set of nodes U holding a copy build a connected component including the gravity center $g(T)$. According to Lemma 2.2 this transformation can be done without increasing the load. The nibble strategy places a copy on a node $v \in V$ if and only if $h(T(v)) > w(T)$ or $v = g(T)$. We have to show that S^* can be transformed into the mapping computed by the nibble strategy without increasing the load on any edge. In particular, we have to show that

- one can add a copy to all nodes $v \in V \setminus U$ with $h(T(v)) > w(T)$, and
- one can remove the copies from all nodes $v \in U \backslash \{g(T)\}$ with $h(T(v)) \leq w(T)$.

without increasing the load on any edge.

Consider $v \in V \setminus U$ with $h(T(v)) > w(T)$. All nodes v' on the simple path from v to U (including v) satisfy $h(T(v')) \geq h(T(v)) > w(T)$ because U includes the gravity center. Adding a copy to all these nodes influences only the load on the edges on this path. In particular, the load on these edges is increased by at most $w(T) - w(T(v))$ because of write accesses issued by nodes not in $T(v)$ that have to traverse these edges after adding the copies. However, the load on these edges is decreased by at least $r(T(v))$ because of read accesses issued by nodes in $T(v)$ which can be satisfied now by the copy on v. Combining both effects, the load is increased by at most $w(T) - w(T(v)) - r(T(v)) = w(T) - h(T(v)) < 0$. This means adding a copy to all nodes $v \in V \setminus U$ with $h(T(v)) > w(T)$ does not increase the load on any edge.

Now consider $v \in U \setminus \{g(T)\}$ with $h(T(v)) \leq w(T)$. Let T' denote the subtree of $T(v)$ induced by the nodes holding a copy. Then each node v' in the subtree T' satisfies $h(T(v')) \leq h(T(v)) \leq w(T)$. Removing all copies from all nodes in T' influences only the load on edges in T'. On the one hand, the load on these edges is increased at most by $r(T(v))$ because of reads issued by nodes in $T(v)$. On the other hand, the load is decreased by at least $w(T) - w(T(v))$ because of writes issued by nodes not in $T(v)$. Thus, the load on these edges is increased by at most $r(T(v)) - (w(T) - w(T(v))) = h(T(v)) - w(T) \leq 0$. Hence, removing the copies from all nodes $v \in U \backslash \{g(T)\}$ with $h(T(v)) \leq w(T)$ does not increase the load on any edge, which completes the proof of Theorem 2.1. ∎

2.2 Dynamic placement

Here all placement decisions have to be made on–line, i.e., we have no knowledge about the access patterns beforehand. It is assumed that an adversary initiates the read and write accesses arbitrarily at execution time. A dynamic strategy is allowed to migrate, to create, and to invalidate, copies during the execution of the application. Initially, one copy of each object is placed somewhere in the network. Arbitrary communication between neighboring nodes in the network is allowed. However, each communication increases the load on the involved edge by one.

We describe a very simple caching protocol that invalidates all but one copy in case of a write. The accesses from node v to object x are handled in the following way:

- v wants to read x: v sends a request message to the nearest node u holding a copy of x. u sends the value of x to v, and a copy of x is created on each node on the path from u to v.

- v wants to write x: v starts an invalidation multicast to all nodes holding a copy of x. The nearest node u of these nodes sends a copy of x to v, and v updates this copy. (Thus, v is the owner of the only copy of x after the write.)

Note that the data tracking for this algorithm is very simple because the nodes holding the copies of an object always build a connected component in the tree. For each object x, each node is attached a signpost pointing to the last node that has updated x. (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 write. Note that this mechanism does not require extra communication, because only the signposts on nodes involved in the invalidation multicast has to be redirected. The number of signposts can be reduced by defining a root of the tree and canceling all signposts directed to the root. Then $\operatorname{diam}(T)$ signposts for each object are sufficient.

Theorem 2.3 *The dynamic caching strategy for trees is 3–competitive.*

Remark: In order to achieve a sequential consistent strategy according to the definition of Lamport [16], it is necessary to acknowledge the invalidations before updating a copy. Then the strategy becomes 4–competitive.

Proof. The accesses from nodes to each object x can be ordered by their occurrence. (Concurrent reads are ordered arbitrarily.) Then the sequence of accesses can be divided into *phases* such that the last access in each phase is a write and each phase includes only one write.

Fix one of these phases t. At the beginning of this phase there is only one copy. Let v denote the node holding this copy. Let U denote the connected component induced by the nodes holding a copy of x at the end of the phase. The additional load on all edges in U due to accesses to x in phase t is 3, because it is send one request message, one data message including a copy of x, and one invalidation message along each of these edges. The additional load on all other edges due to these accesses is 0.

Now consider the load of an arbitrary other consistent strategy. If phase t is the first phase, the the v is the node holding the initial copy of x. Otherwise, v is the last node that has issued a write access to x before phase t begins. In both cases, the value of the copy of x on v has to be propagated to the nodes that want to access x in phase t. This requires to send at least one message along every edge in U. This gives load 1 on each edge in U, which is 1/3 of the load of our strategy. ∎

3 Data management on meshes

In this section, we consider data management strategies for the mesh $M = M(m_1, \ldots, m_d)$, i.e., the d–dimensional mesh–connected network with side length m_i in dimension i. The number of processors is denoted by n, i.e., $n = m_1 \cdot \ldots \cdot m_d$. Each edge is assumed to have bandwidth 1. Thus, the relative and absolute load of an edge are identical. We describe static and dynamic strategies that aim to minimize the congestion. These strategies are based on the embedding of so called access trees into the mesh.

3.1 The static access tree strategy

The static access tree strategy is based on a hierarchical decomposition of M. We describe this decomposition recursively. Let i be the smallest index such that $m_i = \max\{m_1, \ldots, m_d\}$. If $m_i = 1$ then we have reached the end of the recursion. Otherwise, we partition M into two non overlapping submeshes $M_1 = M(m_1, \ldots, \lceil m_i/2 \rceil, \ldots, m_d)$ and $M_2 = M(m_1, \ldots, \lfloor m_i/2 \rfloor, \ldots, m_d)$. M_1 and M_2 are then decomposed recursively according to the same rules. Figure 2 gives an example for this decomposition.

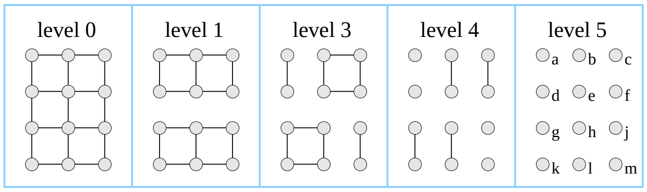

Figure 2: The partitions of $M(4,3)$.

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. Thus, $T(M)$ is a binary tree of height $O(\log n)$ into which the leaves correspond to submeshes of size one, i.e., to the processors of M. We define the root to be on level 0 of this tree, and all nodes whose parents are on level i are defined to be on level $i+1$. For each node v in $T(M)$, let $M(v)$ denote the corresponding submesh. Further, each edge e of $T(M)$ connecting a level i node u with a level $i+1$ node v is defined to be on level $i+1$ and $M(e) = M(v)$.

We interpret $T(M)$ as 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 an edge e of $T(M)$, define the bandwidth of e to be the number of edges leaving submesh $M(e)$. Figure 3 gives an example of a decomposition tree with bandwidths.

For each object $x \in X$, define an *access tree* $T_x(M)$ to be a copy of the decomposition tree $T(M)$. We embed the access trees randomly into M, i.e., for each $x \in X$, each interior node v of $T_x(M)$ is mapped by a random hash function $h(x, v)$ to one of the processors in $M(v)$, and each leaf v of $T_x(M)$ is mapped onto the processor in $M(v)$. (For simplicity we assume that the hash functions map in a truly random fashion, i.e., uniformly and independently.)

The remaining description of our data management strategy is very simple: For object $x \in X$, we simulate the

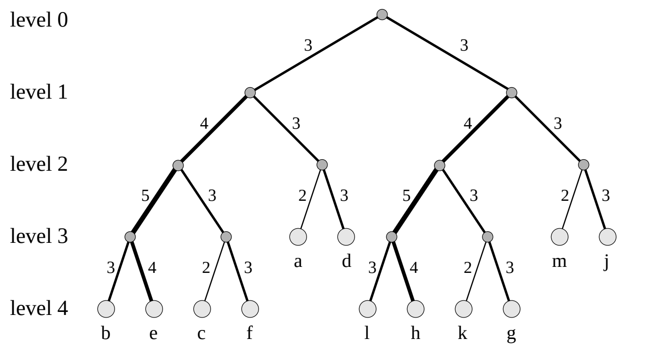

Figure 3: The decomposition tree $T(M(4,3))$. The node labels are equivalent to the ones in Figure 2. The edge labels indicate the bandwidth

optimal static strategy for trees described in section 2.1 on the access tree $T_x(M)$. 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.

The access static tree strategy yields a static placement of the objects onto the nodes and specifies the routing paths. This placement can be calculated in time $O(|X| \cdot |V|)$, because the optimal static placement of an object on its access tree can be calculated in time linear in the number of nodes of the tree, and this number is smaller than 2 times the number of nodes in the mesh. The following theorem shows that the access tree strategy achieves small congestion.

Theorem 3.1 *For any application on the mesh M of size n and dimension d, the static access tree strategy achieves congestion $O(C^{\text{stat}}_{\text{opt}}(M) \cdot d \cdot \log n)$, w.h.p., where $C^{\text{stat}}_{\text{opt}}(M)$ denotes the optimal congestion for the application in the static model.*

Proof. In order to prove the above result we require a lower bound and an upper bound on the congestion of the optimal strategy and the access tree strategy, respectively. Define $C^{\text{stat}}_{\text{opt}}(T(M))$ to be the best achievable congestion for the application when it is executed on the binary tree $T(M)$, under the assumption that each processor of M is simulated by its counterpart in $T(M)$. We give a lower bound that relates $C^{\text{stat}}_{\text{opt}}(T(M))$ to the optimal congestion on the mesh, and an upper bound that relates this value to the congestion achieved by our strategy. We start with the lower bound.

Lemma 3.2 $C^{\text{stat}}_{\text{opt}}(T(M)) \leq C^{\text{stat}}_{\text{opt}}(M)$.

Proof. For a given strategy on M with congestion C we have to describe a strategy on $T(M)$ with congestion at most C.

We simulate the strategy for M on $T(M)$, except for the routing. Instead, for the routing paths in $T(m,d)$ we use the unique shortest paths between the respective nodes. Let C' denote the congestion for a given application with the above strategy on $T(M)$. Let e denote an edge of $T(M)$ with relative load C'. Then the absolute load of e is $C' \cdot b(e)$ (remind that $b(e)$ is the bandwidth of e).

Now consider the same application on M. Any message that crosses e in $T(M)$ has either to leave or to enter the submesh $M(e)$ in M. The number of edges leaving $M(e)$ is $b(e)$. Thus, the load on one of these edges is at least $C' \cdot b(e)/b(e) = C'$, and hence, $C \geq C'$. ■

The following lemma gives an upper bound on the expected load of the access tree strategy. For an edge e of M, let $L(e)$ denote the load of e and $E[L(e)]$ the expectation of this value.

Lemma 3.3 *For any edge e of M, $E[L(e)] = O(\log n \cdot d \cdot C^{\text{stat}}_{\text{opt}}(T(M)))$.*

Proof. Let h denote the height of $T(M)$, and let $L_i(e)$ denote the load of e due to the simulation of edges on level i of $T(M)$, for $1 \leq i \leq h$. We show that $E[L_i(e)] = O(d \cdot C^{\text{stat}}_{\text{opt}}(T(M)))$, which yields the lemma since $h = O(\log n)$.

Fix i with $1 \leq i \leq h$. Let v be a node of $T(M)$ on level $i-1$, and let v_1 and v_2 be the children of v. Then the mesh $M(v)$ is partioned by the hierarchical decomposition into the submeshes $M(v_1)$ and $M(v_2)$. The nodes v, v_1, and v_2 are each mapped randomly to a node in $M(v)$, $M(v_1)$, and $M(v_2)$, respectively. $M(v_1)$ and $M(v_2)$ are submeshes of $M(v)$ that have at least 1/3 the size of $M(v)$. As a consequence, the expected load on an arbitrary edge e of $M(v)$ for sending a message between two nodes chosen randomly from $M(v)$ is at most 3 times larger than sending a message from a node chosen randomly either from $M(v_1)$ or $M(v_2)$ to a node chosen randomly from $M(v)$. Therefore, in the following we assume, for simplicity, that we are choosing v_1 and v_2 randomly from $M(v)$, rather than choosing v_1 from $M(v_1)$ and v_2 from $M(v_2)$.

Let k_j denote the side length of $M(v)$ in dimension j. W.l.o.g. we assume $k_1 \leq k_2 \leq \ldots \leq k_d$. Let size$(M(v))$ denote the number of nodes in $M(v)$. Further, let out$(M(v))$ denote the number of edges leaving $M(v)$, and let out$_j(M(v))$ denote the number of edges of dimension j leaving $M(v)$. Then the order of the partitions in the hierarchical decomposition ensures that we have not divided the mesh according to dimension j if $k_j < k_d/2$. Thus, $\text{out}_j(M(v)) = 0$ if $k_j < k_d/2$. Otherwise,

$$\text{out}_j(M(v)) \leq \frac{2 \cdot \text{size}(M(v))}{k_j} \leq \frac{4 \cdot \text{size}(M(v))}{k_d} \quad .$$

Thus, $\text{size}(M(v)) \geq k_d \cdot \text{out}(M(v))/4d$.

Consider an edge e from $M(v)$. Let j denote the dimension of e. The probability that a dimension–by–dimension order path between two nodes chosen randomly from $M(v)$ crosses e is at most

$$\frac{k_j}{2 \cdot \text{size}(M(v))} \leq \frac{k_d}{2 \cdot \text{size}(M(v))} \leq \frac{2d}{\text{out}(M(v))} \quad .$$

(This can be seen easily as follows. Add wrap around edges to $M(v)$ in each dimension. Then $M(v)$ is a torus, and all edges in dimension j look the same. The number of these edges is size$(M(v))$, and the expected number of these edges that are traversed by the random path is $\lfloor k_j/4 \rfloor$. Therefore, the expected load on one of these torus edges is at most $k_j/(4 \cdot \text{size}(M(v)))$. The expected load for the mesh edges is at most twice this value.)

Now consider the tree edge $\{v_1, v\}$. The bandwidth of this edge is at most out$(M(v))$. Thus, the maximum number of messages that are transmitted along this edge is at most $C^{\text{stat}}_{\text{opt}}(T(M)) \cdot \text{out}(M(v))$. The edge $\{v_1, v\}$ is simulated by a dimension–by–dimension order path in $M(v)$. As a consequence, the expected load on edge e for simulating edge $\{v_1, v\}$ is at most

$$C_{\text{opt}}^{\text{stat}}(T(M)) \cdot \text{out}(M(v)) \cdot \frac{2d}{\text{out}(M(v))} = 2d \cdot C_{\text{opt}}^{\text{stat}}(T(M)))$$

under the assumption that v and v_1 are chosen randomly from $M(v)$. Of course, the same holds for the simulation of edge $\{v_2, v\}$. Thus, $E(L_i(e)) = O(C_{\text{opt}}^{\text{stat}}(T(M)) \cdot d)$, and hence, $E[L(e)] = O(\log n \cdot d \cdot C_{\text{opt}}^{\text{stat}}(T(M)))$. ■

In order to complete the proof of Theorem 3.1, we have to show that the maximum load over all edges does not deviate too much from the expected load of a fixed edge.

Consider edge e of the mesh M. Let L denote the load of e. We color the edges of the access trees with three colors $\{1, 2, 3\}$ such that all incident edges have different colors. For $1 \leq j \leq 3$, let $E_{x,j}$ be the set of tree edges of $T_x(M)$ with color j. For $\ell \in E_{x,j}$, let $A_j(\ell)$ be a random variable that is 1 if the dimension–by–dimension order path that simulates ℓ crosses e, and that is 0, otherwise. Further, for $\ell \in E_{x,j}$, let $K(\ell)$ denote the load of ℓ in the access tree $T_x(M)$. Then define

$$L_j := \sum_{x \in X} \sum_{\ell \in E_{x,j}} A_j(\ell) \cdot K(\ell) \ , \qquad (1)$$

for $1 \leq j \leq 3$, i.e., $L = L_1 + L_2 + L_3$. The coloring yields that the $A_j(\ell)$'s are independent, for fixed j. Therefore, L_j is a sum of weighted independent random variables. Let κ denote the write contention, i.e., the maximum number of write accesses to the same object. Then the static strategy for trees guarantees that for every $\ell \in E_{x,j}$, $K(\ell) = O(\kappa)$. As a consequence, the maximum weight in the sum of random variables in equation 1 is at most $O(\kappa)$. Applying a Chernoff–Hoeffding bound [11] to this sum yields that L_j deviates by at most $O(\kappa \cdot \log n)$ from $E[L_j]$, w.h.p., for $1 \leq j \leq 3$.

Since $L = L_1 + L_2 + L_3$, it follows $L = E[L] + O(\kappa \cdot \log n)$, w.h.p.. Now, applying Lemma 3.2 and Lemma 3.3 yields $L = O(\log n \cdot (d \cdot C_{\text{opt}}^{\text{stat}}(M) + \kappa))$, w.h.p.. Further, we have $C_{\text{opt}}^{\text{stat}}(M) \geq \kappa/d$ because each write access has to update all copies and each copy is placed statically at a node with degree d. Thus, $L = O(\log n \cdot d \cdot C_{\text{opt}}^{\text{stat}}(M))$, w.h.p.. Summing over all edges yields that the same bound holds for the congestion. This completes the proof of Theorem 3.1. ■

3.2 The dynamic access tree strategy

We can easily adapt the access tree strategy described above to the dynamic model by simulating the dynamic instead of the static tree strategy on the access trees. This gives congestion $O(C_{\text{opt}}^{\text{dyn}}(M) \cdot d \cdot \log n + \kappa \cdot \log n)$, w.h.p., where $C_{\text{opt}}^{\text{dyn}}(M)$ denotes the optimal congestion for a given application in the dynamic model, and κ denotes the write contention. In the static model, the optimal congestion was at least κ/d. From this we could deduce $O(d \cdot \log n)$–competitiveness. Unfortunately, in the dynamic model we cannot apply this bound. In fact it is easy to construct counterexamples in which the optimal congestion is much smaller than κ/d. In order to achieve $O(d \cdot \log n)$–competitiveness in the dynamic model the access trees have to be remapped dynamically when too many write accesses are directed to the same variable. This yields the following result.

Theorem 3.4 *The dynamic access tree strategy is $O(d \cdot \log n)$–competitive, for meshes of size n and dimension d.*

Due to space limitations we omit the proof for this theorem. It can be found in a full version of the paper [20].

4 Data management on clustered networks

A *clustered network* $G = (V, E)$ is a network that consists of several small subnetworks, i.e., *clusters*, that are organized hierarchically. The *cluster tree* $T(G)$ describes this hierarchical structure. The internal nodes of $T(G)$ correspond to the clusters of G, and the leaves correspond to the *user processors*. (In the following, we interpret these processors as clusters of size 1.) Each pair of clusters that is connected by one or more edges in G is also connected by an edge in $T(G)$. The bandwidth of this edge in $T(G)$ is the sum of the bandwidths of the corresponding edges in G. For instance, NOWs are usually organized as clustered networks. Figure 4 depicts a possible topology for a wide-area NOW.

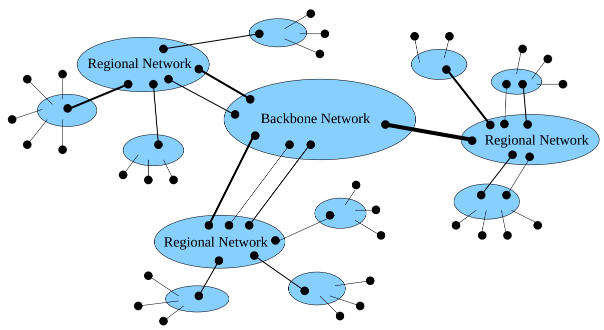

Figure 4: The topology of a wide area network of workstations.

The nodes inside the clusters are connected in an arbitrary fashion. However, we assume that communication between nodes of the same cluster is less expensive than communication between nodes of different clusters, which is just the basic idea behind any kind of clustering. This means we assume that the links connecting nodes in distinct clusters are the bandwidth bottlenecks of the system. This property can be formalized as follows. Consider a cluster $K = (V_K, E_K)$. Define the weight $w(v)$ of a node $v \in V_K$ to be the sum of the bandwidths of its incident edges leaving K, and the weight $w(U)$ of $U \subseteq V_K$ by $w(U) = \sum_{v \in U} w(v)$. Further, for $U \subseteq V_K$, let $\text{cut}(U)$ denote the sum of the bandwidths of the edges connecting the processors in U with the processors in $V_K \backslash U$. Then the *xflux* (cross flux), defined by

$$\alpha(K) \ = \ \min_{U \subseteq V_K} \frac{\text{cut}(U)}{\min\{w(U), w(V_K \backslash U)\}} \ ,$$

is a good measure for the bandwidth bottleneck of K. If $\alpha(K) \ll 1$ then there is a bottleneck inside cluster K. We assume that $\alpha(K) = \Omega(1)$, for any cluster K.

For each object $x \in X$, define the *access tree* $T_x(G)$ to be a copy of the cluster tree $T(G)$. Each interior node a of $T_x(G)$ is mapped randomly to one of the processors in the associated cluster $K = (V_K, E_K)$, i.e., a is mapped with probability $w(v)/w(V_K)$ onto node $v \in V_K$. Analogously to the mesh strategy, we simulate the optimal static or dynamic strategy for trees on these access trees. The static placement for each object can be calculated in time $O(|V|)$. However, the selection of the routing paths is more complicated than the one for meshes. It requires an initialization of each cluster, which takes time polynomial in the size of the cluster. The routing will be described in the proof for the following theorem.

Theorem 4.1 *Consider an application running on the user processors in a clustered network G of size n. Let δ denote the maximum number of edges that leave the same cluster, let γ denote the maximum degree in the cluster tree $T(G)$, and let κ denote the write contention, i.e., the maximum number of write accesses to the same object. Further, let $C_{\rm opt}(G)$ denote the optimal congestion for the application when it is executed on G. Then the access tree strategy achieves congestion*

$$O(\log\delta\cdot C_{\rm opt}(G)+\gamma\cdot\kappa\cdot\log n)\ ,$$

w.h.p.. If the clusters in G can be represented by planar graphs or by constant genus graphs, then this result improves to

$$O(C_{\rm opt}(G)+\gamma\cdot\kappa\cdot\log n)\ ,$$

w.h.p..

Proof. Consider a cluster $K=(V_K,E_K)$. Let $\alpha=\alpha(K)$ denote the xflux of K and $\delta=\delta(K)$ denote the number of edges that leave cluster K. We define the congestion of K to be the maximum relative load of all edges adjacent to nodes in K. Let $C_{\rm opt}=C_{\rm opt}(K)$ denote the optimal congestion of K. We show, for cluster K, that the strategy achieves congestion $O(\log\delta\cdot C_{\rm opt}/\alpha+\delta\cdot\kappa\cdot\log n)$, w.h.p.. If K is planar or of constant genus, then this result improves to $O(C_{\rm opt}/\alpha+\delta\cdot\kappa\cdot\log n)$, w.h.p.. This yields the above theorem. (Furthermore, it shows that the influence of a "bad" xflux is only small and local to the respective cluster.)

First, we describe the path selection strategy for the routing along the edges that leave cluster K. Suppose K' is a cluster neighboring to K. Let u and u' be the cluster tree nodes that represent K and K', respectively, and let e_T be the edge that connects u and u'. Furthermore, let $e_1,\ldots,e_k$ denote the edges in G represented by e_T. We send a message that traverses e_T in $T(G)$ along edge e_i with probability $p(e_i)=b(e_i)/b(e_T)$.

The following lemma, gives an upper bound on the expected relative load of edges that leave cluster K.

Lemma 4.2 *Let e be an edge of G that connects a node in cluster K with a node in one of the neighboring clusters. Then the expected load on e is at most $C_{\rm opt}$.*

Proof. Let e_T denote the edge in $T(G)$ that corresponds to e. Our strategy simulates the optimal strategy on the access trees. This strategy is equivalent to the optimal strategy on the cluster tree $T(G)$. Thus, the load on e_T is the optimal load achievable in $T(G)$. This load is at most $C_{\rm opt}\cdot b(e_T)$.

Therefore, the expected absolute load for edge e is at most $C_{\rm opt}\cdot b(e_T)\cdot p(e)=C_{\rm opt}\cdot b(e)$. As a consequence, the expected relative load on e is at most $C_{\rm opt}$. ∎

Now we have to describe the path selection strategy for the routing inside cluster K. Consider the following multicommodity flow problem. Let $v_1,\ldots,v_\delta$ denote the nodes incident to the edges that leave cluster $K=(V_K,E_K)$. We have δ^2 commodities $\ell_{i,j}$ with $1\le i,j\le\delta$. The source of commodity $\ell_{i,j}$ is v_i, its sink is v_j, and its demand is $w(v_i)\cdot w(v_j)/W$ with $W:=\sum_{k=1}^{\delta}w(v_k)$. We solve this commodity flow problem on K with respect to the capacities, i.e., the bandwidths, of the edges in K. The result is a multicommodity flow in which the demand of each commodity is satisfied up to a factor q. That is, for each commodity $\ell_{i,j}$, there is a flow of size $q\cdot w(v_i)\cdot w(v_j)/W$ from v_i to v_j.

Lemma 4.3 *In the general case, $q=\Omega(\alpha/\log\delta)$. If K is a planar graph or a constant genus graph then $q=\Omega(\alpha)$.*

Proof. First we consider the general case. According to [3] any multicommodity flow problem can be satisfied up to a factor $O(S/\log k)$ with S denoting the minimum cut ratio and k denoting the number of commodities. It is easy to check that the minimum cut ratio of our multicommodity flow problem is α. Further, the number of commodities is δ^2. Therefore, the maximum flow can be satisfied up to a factor $q=\Omega(\alpha/\log\delta)$.

Now suppose K is a planar graph or of constant genus. Then we can translate the above multicommodity flow problem into a uniform multicommodity problem in which each node sends the same amount of data to each other node such that the flow and the cut ratio in both problems is nearly equivalent. According to [14], the flow in uniform multicommodity factor can be satisfied up to a factor $q=O(S)$ with S denoting the minimum cut ratio. Applying, $S=O(\alpha)$ yields the lemma. ∎

The maximum flow for cluster K can be calculated efficiently by a randomized approximation scheme in time $O(\delta^2\cdot|V_K|\cdot|E_K|\cdot\log^4|V_K|)$, see [18]. Alternatively, it can be calculated deterministically based on linear programming. Let $f_{i,j}:E_K\to I\!R$, represent the flow of commodity $\ell_{i,j}$ on the respective edges. If the flow on edge $e=(x,y)$ is oriented from x to y, then $f_{i,j}(x,y)$ is positive; otherwise it is negative. We use these flow values to determine the routing paths. Let v be a node of K. Let $F_{i,j}(v)$ be the sum of flows that leave v. Let $(v,y)\in E_K$ be an edge with $f_{i,j}(v,y)>0$. Then each message with source v_i and destination v_j that arrives at node v is sent with probability $f_{i,j}(v,y)/F_{i,j}(v)$ along edge (v,y).

Lemma 4.4 *Let e be an edge of G that connects two nodes of cluster K. Then the expected relative load on e is at most $2\cdot C_{\rm opt}/q$.*

Proof. Suppose each node v_i wants to send $q\cdot w(v_i)\cdot w(v_j)/W$ data units to node v_j, for $1\le i,j\le\delta$. Then the expected load on each edge e is equal to its bandwidth. This means, the expected congestion in K is at most 1.

Now consider the messages that arrive in the cluster. According to Lemma 4.2 the number of messages that arrives from outside at node v_i, with $1\le i\le\delta$, is at most $C_{\rm opt}\cdot w(v_i)$ since $w(v_i)$ is equivalent to the sum of the bandwidths of the incident edges leaving K. Each message is sent to a random destination chosen from $v_1,\ldots,v_\delta$ such that the probability for v_j to be the destination is $w(v_j)/W$. As a consequence, for $1\le i,j\le\delta$, the expected amount of data for incoming messages sent from node v_i to node v_j is at most $C_{\rm opt}\cdot w(v_i)\cdot w(v_j)/W$. Analogously, we can prove that the same bound holds for the messages that leave the cluster. As a consequence, the expected relative load on each edge is at most $2\cdot C_{\rm opt}/q$. ∎

Combining the results of the Lemmas 4.2, 4.3, and 4.4 yields that the expected load for each edge in K is $O(\log\delta \cdot C_{\mathrm{opt}}/\alpha)$ in the general case, and it is $O(C_{\mathrm{opt}}/\alpha)$ if K is planar or of constant genus. In order to complete the proof of Theorem 4.1, we have to show that the maximum relative load over all edges in K does not deviate too much from the expected load of a fixed edge.

Consider edge e of the cluster K. Let L denote the load of e. Then we have to show that $L = E[L] + O(\gamma \cdot \delta \cdot \log n)$, w.h.p.. Let L_x denote the load of e due to the accesses to objects in $x \in X$. The data management strategy for trees ensures that the load on each edge of the cluster tree due to accesses to x is $O(\kappa)$. Further, each edge e is involved in the simulation of at most γ different cluster tree edges. Therefore, $L_x = O(\gamma \cdot \kappa)$. This means that L_x is the sum of $O(\gamma \cdot \delta)$ not necessarily independent 0–1–random variables. We add some dummy variables so that L_x is the sum of exactly $\tau = O(\gamma \cdot \kappa)$ random variables $A_1(x), \ldots, A_\tau(x)$, for every $x \in X$. Then

$$L = \sum_{x \in X} \sum_{i=1}^{\tau} A_i(x) = \sum_{i=1}^{\tau} \sum_{x \in X} A_i(x) \ .$$

The variables $A_i(x)$ in the sum $S_i = \sum_{x \in X} A_i(x)$ are independent, for $1 \leq i \leq \tau$. Applying a Chernoff–Hoeffding bound [11] to this sum yields that it deviates by at most $O(\log n)$ from $E[S_i]$, w.h.p., for $1 \leq i \leq \tau$. Since $L = \sum_{i=1}^{\tau} S_i$, it follows $L = E[L] + O(\gamma \cdot \delta \cdot \log n)$, w.h.p., which completes the proof of Theorem 4.1. ∎

References

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

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

[3] Y. Aumann and R. Rabani. An $O(\log k)$ approximate min–cut max–flow theorem and approximation algorithm. *SIAM Journal on Computing, to appear*, 1997.

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

[5] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. In *Proc. of the 7th ACM Symp. on Discrete Algorithms (SODA)*, pages 574–583, 1996.

[6] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. of the 24th ACM Symp. on Theory of Computing (STOC)*, pages 39–50, 1992.

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

[8] R. Cypher, F. Meyer auf der Heide, C. Scheideler, and B. Vöcking. Universal algorithms for store-and-forward and wormhole routing. In *Proc. of the 26th ACM Symp. on Theory of Computing (STOC)*, pages 356–365, 1996.

[9] D. Dowdy and D. Foster. Comparative models of the file assignment problem. *Computing Surveys*, 14(2):287–313, 1982.

[10] A. J. van de Goor. *Computer Architecture and Design.* Addison Wesley, 1994.

[11] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1989/90.

[12] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*, pages 654–655, 1997.

[13] A. Karlin and E. Upfal. Parallel hashing–an efficient implementation of shared memory. In *Proc. of the 18th ACM Symp. on Theory of Computing (STOC)*, pages 160–168, 1986.

[14] P. Klein, S. A. Plotkin, and S. Rao. Excluded minors, network decomposition, and multicommodity flow. In *Proc. of the 25th ACM Symp. on Theory of Computing (STOC)*, pages 682–690, 1993.

[15] 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, Jan. 1997.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, c-28(9):690–691, 1979.

[17] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17:157–205, 1994.

[18] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Science*, 50:228–243, 1995.

[19] C. Lund, N. Reingold, J. Westbrook, and D. Yan. On-line distributed data management. In *Proc. of the 2nd European Symposium on Algorithms (ESA)*, 1996.

[20] B. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for networks of limited bandwidth. Technical Report tr-rsfb-97-042, University of Paderborn, 1997.

[21] F. Meyer auf der Heide. Efficiency of universal parallel computers. *Acta Informatica*, 19:269–296, 1983.

[22] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15(1):106–119, Feb. 1986.

[23] F. Meyer auf der Heide and B. Vöcking. A packet routing protocol for arbitrary networks. In *Proc. of the 12th Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 291–302, 1995.

[24] F. Meyer auf der Heide and R. Wanka. Time-optimal simulations of networks by universal parallel computers. In *Proc. of the 6th Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 120–131, 1989.

[25] R. Ostrovsky and Y. Rabani. Universal $O(\text{congestion} + \text{dilation} + \log^{1+\epsilon} n)$ local control packet switching algorithms. In *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*, pages 644–653, 1997.

[26] C. G. Plaxton and R. Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *Proc. of the 37th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 570–579, 1996.

[27] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Science*, 42:307–326, 1991.

[28] C. Scheideler and B. Vöcking. Universal continuous routing strategies. In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 142–151, 1996.

[29] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34:116–127, 1987.

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