# Efficient Algorithms for Dynamic Allocation of Distributed Memory

Tom Leighton *
Eric J. Schwabe †

Department of Mathematics and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

## Abstract

We consider the problem of dynamically allocating and deallocating local memory resources among multiple users in a parallel or distributed system. Given a group of independent users and a collection of interconnected local memory devices, we want to render the fragmentation of the memory resources irrelevant by allowing any user to allocate space for his or her purposes as long as there is space available anywhere in the system. In effect, we would like it to appear to the users as though they are allocating memory from a single central source, even though the space is distributed throughout the system.

Our goal is to devise an on-line allocation algorithm that minimizes both the fraction of unused space due to fragmentation of the memory and the slowdown needed by the system to service user requests. We solve this *distributed dynamic allocation problem* in near-optimal fashion by devising an algorithm that allows the memory to be used to 100% of capacity despite the fragmentation and guarantees that service delays will always be within a constant factor of optimal. The algorithm is completely on-line (no foreknowledge of user activity is assumed) and can accommodate any sequence of insertions and deletions by the users which does not violate global memory bounds.

Our results have applications in the domain of parallel disk allocation, where a set of independent workstations are interconnected to a collection of disks that are controlled by a central file server. The algorithm will allow each workstation to have exclusive access to an arbitrary amount of space on the disks provided only that the overall capacity of the disks is never exceeded by the cumulative demands of the workstations. The distributed dynamic allocation problem also arises in the context of parallel architectures with a distributed memory, and in certain queueing problems on fixed-connection networks.

*Supported by Air Force Contract OSR-89-02171, DARPA Contracts N00014-87-K-825 and N00014-89-J-1988, and Army Contract DAAL-03-86-K-0171. Author's email address: ftl@math.mit.edu.

†Supported by a National Science Foundation Graduate Fellowship, DARPA Contract N00014-89-J-1988, and a DARPA/NASA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland. Author's email address: schwabe@theory.lcs.mit.edu.

## 1 Introduction

In parallel and distributed computation, the total available memory is typically divided into many pieces to allow simultaneous accesses. The price we pay for this increase in speed is the fragmentation of memory — memory is stored in small amounts at many different locations. As a result, there may be times when local memory demands exceed the resources available at that point even though global memory bounds are not exceeded by the cumulative demands. It would be highly desirable to have general methods for reallocating memory from those places where it is not being used to those places where it is needed, thus guaranteeing that local memory needs can always be satisfied as long as global memory bounds are not exceeded.

In this paper, we consider the general problem of simulating a parallel or distributed machine with memories of varying sizes on one with memories of a fixed size (see Figure 1), and show that this is a useful model for problems that arise in practice. We model this situation as the following combinatorial problem, which we call the *distributed dynamic allocation problem*:



CH3062-7/91/0000/0470$01.00 © 1991 IEEE

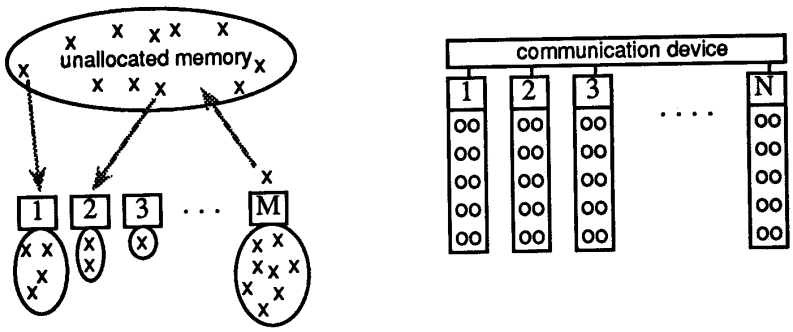


Figure 1: The virtual and real machines' views of memory. The number of x's is $p$; the number of o's is $qN$.

1. There are $M$ users, each of whom can delete and/or insert one block of private (i.e., non-shared) data into the distributed memory every $T$ steps. We refer to these blocks as *elements*, and elements inserted by the same user are said to be of the same *type*.

2. There are $N$ memory devices, each of which can contain up to $q$ of these elements. Each memory can perform one deletion and/or insertion per step.

3. The total number of elements stored in the distributed memory at any time is at most $p = (1-\sigma)qN$, where $\sigma$ is the fraction of wasted space in the system.

The problem is to devise an on-line algorithm for allocating and deallocating the available space that minimizes both the fraction of wasted space $\sigma$ which arises due to the fragmentation of the memory and the slowdown $T$ needed by the system to service the users' requests.

One example of a practical problem that fits into this framework is parallel disk allocation (see Figure 2). We have $M$ independent workstations connected to a central file server, which is in turn connected by a bus to $N$ independent disks of size $q$. (Since bus communication is in general at least an order of magnitude faster than disk I/O in bits per second, it is not unreasonable to consider the disks to be hooked up to the file server in parallel. For example, the network of workstations used by the Theory Group at M.I.T. has disks which can transfer data at 3 Mbits/second connected to a central file server by a bus which can operate at 22 Mbytes/second = 176 Mbits/second. Since there are fewer than 10 disks hooked up to this bus, their total data transfer rate is still far lower than what the bus can handle, and therefore we can consider them to be hooked up in parallel to the file server.) As long as there is space available on any of the $N$ disks, we want any of the $M$ workstations to be able to allocate it for its own use. When this space is later deallocated, it returns to the set of disk locations which are available for allocation. It is clear that this is just a restatement of the distributed dynamic allocation problem, where the elements of different types correspond to the blocks allocated by the workstations and the memories correspond to the disks. Therefore solutions to the distributed dynamic allocation problem will immediately yield algorithms for parallel disk allocation.

The problem of distributed dynamic allocation differs from previous work on the file assignment problem [3] in that the file assignment problem assumes that any processor can access the files written by any other, thus necessitating the use of considerable information about the pattern of future file accesses in order to achieve even heuristic results. By allowing the files written by a processor to be exclusively accessible to that processor until it deallocates the space, our strongest algorithm achieves performance within a constant factor of optimal in a fully on-line setting. Shared memory simulations [4, 5], on the other hand, often duplicate stored elements in order to reduce slowdown. This effectively makes only a fraction of the memory capacity available to store distinct elements. By keeping allocated space private to its owner until it is deallocated, we can insure that the available memory is filled to capacity with distinct elements (i.e., with no duplication). Of course, it is possible to build shared variable primitives in our model by sending messages between users, but it could involve a serious degradation in the near-optimal performance of our algorithms. Indeed, it is not possible to use 100%

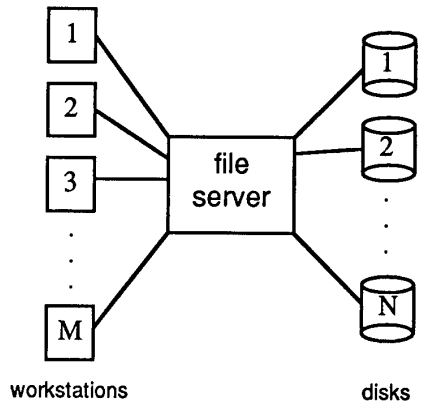


Figure 2: The parallel disk allocation problem. As long as there is space available on any of the disks, any of the workstations can allocate it.

of the available memory and allow arbitrary shared access between users.

As was mentioned before, the two natural measures of the efficiency of a solution to the distributed dynamic allocation problem are its slowdown $T$ and its fraction of wasted space $\sigma$. The slowdown is the number of steps needed to realize one deletion and/or insertion on the part of each user. It is clear that in the worst case, $T = \Omega(M/N)$, since some memory will contain $M/N$ elements of different types which could all be requested simultaneously. The fraction of wasted space $\sigma$ is the fraction of the total space in the $N$ memories which we cannot guarantee is filled with elements of the $M$ types; this implies that the total number of elements that can be stored in the distributed memory is $(1-\sigma)qN$. Our goal is to minimize both $T$ and $\sigma$. A near-optimal algorithm will be one which achieves $T = O(M/N + 1)$ and $\sigma = 0$ — slowdown within a constant factor of optimal and no wasted space.

In Section 2, we give algorithms which solve the distributed dynamic allocation problem. First we note that in the off-line case, where each element inserted carries with it a time-stamp denoting when it will next be requested, we can easily solve the distributed dynamic allocation problem near-optimally, with $T = O(M/N + 1)$ and $\sigma = 0$.

For the more interesting and natural on-line case, where we have no prior information about the sequence of insertions and deletions, we begin by giving a simple algorithm which solves the distributed dynamic allocation problem with slowdown $T = O(M/N + 1)$, but which wastes a constant fraction of the available space. This method can be extended to obtain an algorithm with $T = O((M/N + 1)\log q)$ and $\sigma = 0$. In fact, these two results are part of a general trade-off between slowdown and wasted space leading to algorithms with $T = O((M/N + 1)\log 1/\sigma)$ for any $\sigma > 0$. These algorithms all fall short of our goal, which is to simultaneously achieve $O(M/N + 1)$ slowdown and no wasted space.

The reason that simple deterministic algorithms have a hard time achieving this near-optimal performance is related to the properties of a graph derived from the configuration of elements in the memories. Construct a graph with $N$ nodes corresponding to the memories, where there is an edge between two nodes if their corresponding memories contain elements of the same type. We note that the number of different types contained in any memory is a lower bound on the slowdown of the algorithm, since it is the number of different elements which might have to be removed (one at a time) from a memory during the next set of user requests. Therefore the maximum number of different types of elements inhabiting any memory must be $O(M/N + 1)$ if we are to have any hope of achieving $O(M/N + 1)$ slowdown. Suppose that all the memories are full and that each memory contains the largest allowable number of types. If the resulting graph does not have the property that given any small set of nodes, the set of neighbors of those nodes includes sufficiently many nodes outside of the set, then no matter what algorithm is used there is some pattern of user requests which will incur more

than an $O(M/N+1)$ slowdown. We will show how to maintain an expansion property in this *shared type graph* which is sufficient to insure that all small sets have enough neighbors, leading to an algorithm that achieves $O(M/N+1)$ slowdown and 100% usage of the available memory. This solves the distributed dynamic allocation problem near-optimally.

In the statement of the distributed dynamic allocation problem, the $M$ types are effectively random-access memories and we assume unit-time communication between the $N$ memories. If we modify our communication model to have the $N$ memories at the nodes of a fixed-connection network, then solutions with $O(M/N+1)$ slowdown are no longer possible. However, if we restrict the access each user has to its elements to be FIFO, then we obtain the related problem of dynamically allocating queues in a fixed-connection network. In Section 3 we consider this problem and give deterministic algorithms for its solution with near-optimal slowdown and total space $qN = O(M \log M + p)$, where the asymptotic constant can be brought arbitrarily close to 1. We show that this space bound is optimal in the worst case for bounded-degree networks, but that it can be improved for average-case problems.

# 2 Solving the Distributed Dynamic Allocation Problem

In this section we give algorithms for the distributed dynamic allocation problem. We first note that near-optimal performance is easily achieved in an off-line setting. Next we describe a simple on-line approach which leads to a general trade-off between slowdown and wasted space. However, none of the resulting on-line algorithms simultaneously achieves constant slowdown and 100% usage of the available space. Finally, we describe certain conditions which near-optimal algorithms must satisfy, and demonstrate a near-optimal deterministic algorithm for the distributed dynamic allocation problem using an expansion property which is sufficient to satisfy these conditions.

Without loss of generality, we will throughout this section assume that the number of types is equal to $N$, the number of memories. When $M > N$, the only effect on our algorithms is that the slowdown degrades by the natural factor of $M/N$.

## 2.1 The Off-line Problem

First we observe that the off-line version of the distributed dynamic allocation problem, where the sequence of insertions and deletions is known ahead of time, can be solved near-optimally. In fact, it is sufficient for each inserted element to know when it will be deleted. The proof is omitted, but the algorithm works by guaranteeing that for any future time, at most three elements in each memory will next be requested at that time. This makes insertions of new elements straightforward, and although deletions require some load balancing to be done, they can be performed in constant time as well.

**Theorem 1** *We can solve the off-line version of the distributed dynamic allocation problem with $T = O(1)$ and $\sigma = 0$.* ■

## 2.2 The On-line Problem

In this section we consider the distributed dynamic allocation problem in the more natural on-line setting, where we are not given any information about future insertions and deletions, but only know which elements are being deleted and inserted at the current time step.

The following two lemmas suggest that the distributed dynamic allocation problem is more difficult to solve on-line than off-line. Lemma 1 shows that for any static division of $qN$ elements among the $N$ types, the elements can be assigned to the memories off-line in such a way that no memory contains more than two different types of elements. Lemma 2, on the other hand, shows that such optimal static assignments of elements to memories can not always be maintained dynamically — for any on-line algorithm, there is some sequence of insertions and deletions which results in at least three types of elements being placed into some memory if we maintain the system at 100% of capacity.

**Lemma 1** *Any set of $qN$ elements of $N$ types can be assigned off-line to $N$ memories of size $q$ such that no memory contains elements of more than two different types. In general, this is optimal.*

**Proof** Unless the number of elements of each type is a multiple of $q$, some memory will have to contain two types, so that if this can be achieved then it is in general optimal.

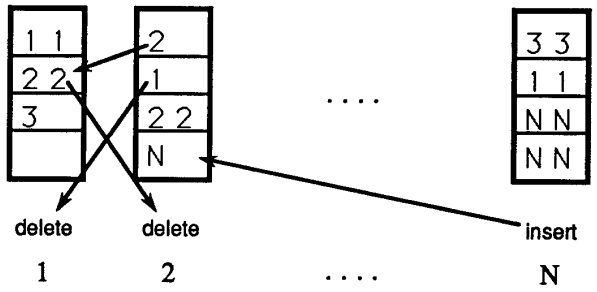


Figure 3: A simple on-line algorithm. Each memory is divided into $c$ dedicated blocks, and all insertions and deletions are done at a type's partial block.

Let a set of $qN$ elements of $N$ types be given. Divide the types into two sets: $S$, consisting of all types with fewer than $q$ elements, and $L$, consisting of all types with at least $q$ elements. We assign the elements to the memories as follows.

Choose the smallest type in $S$, and put all of its elements in some empty memory. Fill the rest of the memory with elements from the smallest type in $L$; if this type now has fewer than $q$ elements remaining, move it from $L$ into $S$. Repeat until $S$ is empty. If at this point both $S$ and $L$ are empty, then we will have filled every memory with elements of exactly two different types and will have assigned every element to some memory. If not, then we are left with some set of empty memories and some set of types in $L$, all having at least $q$ elements. It is easy to assign these remaining types to the memories so that each memory contains elements of at most two types. ■

**Lemma 2** *For any on-line algorithm for the distributed dynamic allocation problem which achieves $T = O(1)$ and $\sigma = 0$, there is a sequence of insertions and deletions that causes elements of three different types to be assigned to some memory.*

**Proof** Consider an on-line algorithm that always assigns at most two different types to each memory. Insert $3qN/8$ elements each of types 1 and 2, followed by $q/3$ elements each of types 3 through $3N/4+2$ (which we call the *small* types). The total number of elements is $qN$, so that all the memories are full.

Since each memory contains elements of at most two types, each memory can contain elements of at most one of the small types. It follows that at least $N/2+4$ of the memories contain all $q/3$ of the elements of some small type. Without loss of generality, assume that $N/4+2$ of these memories are filled the rest of the way with elements of type 1. In order for these memories to be completely full, they must collectively contain at least $(2q/3)(N/4) = qN/6$ elements of type 1, leaving at most $3qN/8 - qN/6 = 5qN/24$ elements of type 1 in the remaining $N/2-4$ memories.

Next we delete one element of the small type contained therein from $N/4-2$ of the memories being considered, and insert a single element of each of the unused $N/4-2$ types. These new elements must end up in distinct memories that currently contain only elements of type 1, except for possibly a constant number of elements of other types. However, there can be at most $5N/24 + o(N)$ such memories; this yields a contradiction. Therefore some memory must receive three types. ■

These lemmas suggest that the natural on-line version of the distributed dynamic allocation problem is more difficult than the off-line version. However, we will show later that slowdown within a constant factor of optimal and 100% usage of the available space can be achieved on-line as well.

The following theorem shows how to achieve constant slowdown while wasting only a constant fraction of the available space.

**Theorem 2** *For any constant $c > 0$, we can solve the distributed dynamic allocation problem with $T = O(c)$ and $\sigma = 1/c$.*

**Proof** The following algorithm achieves the desired performance by dividing each memory into a constant number of 'dedicated blocks', which are allocated to particular types as needed (see Figure 3).

Let $c$ be a constant, and divide each memory into $c$ blocks of $q/c$ memory locations. When the first element of a type is inserted, it is placed in one of the

unassigned blocks, and no other types are permitted to store elements in that block. Further elements of that type are stored in that block until it is full, at which point the next insertion will cause another block to be assigned to that type. At any point in the algorithm, each type has assigned to it some number of blocks that are full of elements of that type, and at most one block that is only partially full of such elements. When an element is deleted, it is removed from whichever block it is stored in, and the resulting hole is filled by moving an element from the partially filled block of that type. When a type's partial block is emptied, that block is moved back to the set of unassigned blocks, and can be reassigned to any type. Since each memory can be storing elements of at most $c$ different types, it is easy to verify that the slowdown of the algorithm is $O(c)$.

Elements can be inserted until there are no unassigned blocks remaining. At this point, there are at most $N$ blocks that are partially full; all the rest are completely full of stored elements. Therefore this algorithm wastes at most $N(q/c)/qN = 1/c$ of the available space. ■

We can achieve 100% usage of the available space by applying this method recursively to the partial blocks. This leads to an algorithm which achieves $O(\log q)$ slowdown and 100% usage of the available space, but by varying the depth of the recursion we obtain the following trade-off between slowdown and wasted space (stated without proof):

**Theorem 3** *For any $\sigma > 0$, we can solve the distributed dynamic allocation problem with $T = O(\log 1/\sigma)$.* ■

In order to see why it is difficult to achieve constant slowdown and 100% usage of the available space simultaneously, consider the graph whose nodes are the $N$ memories and where there is an edge between two nodes if their corresponding memories contain elements of the same type. Recall that in order to have a constant-slowdown algorithm, we must have at most a constant number of different types in any memory. If each memory contains the largest possible number of different types, then the only allowable rearrangements of stored elements (i.e., those with $O(1)$ slowdown) consist of removing at most a constant number of elements from each memory and placing them in other memories which already contain elements of the same type. Each such deletion and insertion effectively induces a unit of flow along the corresponding edge in this *shared type graph* (see Figure 4).

In general, a set of $k$ insertions and deletions induces a network flow problem in the shared type graph with $k$ sources and sinks, where the sources and sinks are not paired — flow can go from any source to any sink. If the smallest cut separating the sources and sinks is of size $o(k)$, then there is no way to perform the desired insertions and deletions with constant slowdown, as this would imply the existence of a maximum flow with value $\Theta(k)$ and violate the max-flow min-cut theorem. Therefore for some $\alpha < 1$, every set $S$ of $k \leq \alpha N$ nodes in the shared type graph must have $\Theta(k)$ of its neighbors outside of $S$ in order for all possible sets of insertions and deletions to be achievable in constant time. In other words, a constant-slowdown algorithm which achieves 100% usage of the available space is only possible when the shared type graph maintains the property that for every small set of nodes, sufficiently many of its neighbors are not among its members.

It is not obvious how to maintain such a property in the shared type graph, as over the course of the algorithm edges may be removed and new edges added as elements are deleted and inserted. We will show how to maintain an expansion property in the shared type graph which is sufficient to satisfy the desired condition, thus demonstrating a constant-slowdown algorithm for the distributed dynamic allocation problem which wastes no space. This algorithm solves the distributed dynamic allocation problem near-optimally.

**Theorem 4** *We can solve the distributed dynamic allocation problem with $T = O(1)$ and $\sigma = 0$.*

**Proof Sketch** Assume that we begin with an assignment of elements to memories such that at most eight different types are contained in any memory. Without loss of generality, assume that all the memories are full. Given such an assignment, we will perform $\Theta(q)$ sets of insertions and deletions by the users, each with constant slowdown, while transforming this assignment into another one with only eight types per memory. Iterating this procedure will yield the desired algorithm for the distributed dynamic allocation problem.

Let the *capacity* of an edge in the shared type graph be the smaller of the numbers of elements of the shared type stored at its two incident nodes. This is the maximum number of elements we can 'pass along' that edge while guaranteeing that the edge will still exist in the shared type graph. We will explain how to, in $q/10$ steps, build a shared type graph for the new assign-

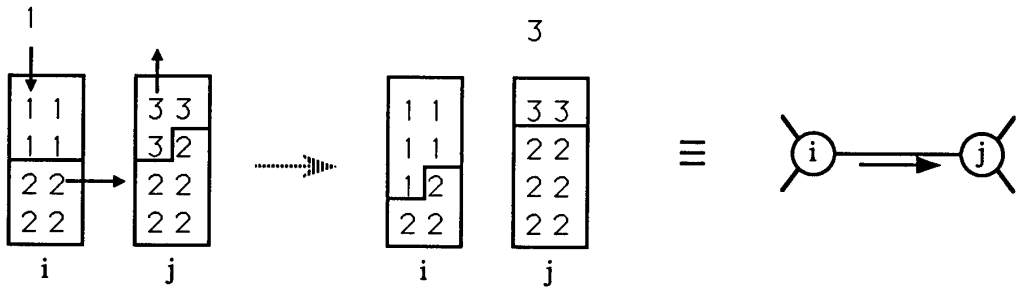


Figure 4: Inserting and deleting elements induces flow along edges in the shared type graph.

ment which contains an expander with enough capacity on each edge to solve $\Theta(q)$ flow problems induced by sets of insertions and deletions by the users. Once this shared type graph is constructed, we need only show how to move the remaining elements from the old assignment to the new assignment in $O(q)$ steps, as we will have sufficient capacity on the expander edges to handle with constant slowdown any insertions and deletions which could take place during that time.

We note that for some positive constant $\alpha < 1$, a 3-regular bipartite graph made up of three random matchings has, with high probability, an expansion property which is sufficient to satisfy the required condition for the shared type graph [1]. We choose such a graph with $N$ nodes as the underlying structure for our shared type graph, and 3-color its edges.

Since each memory has at most eight types, we can choose some set of $q/8$ elements of the same type from each memory. $q/10$ of these elements from each memory will be used to construct the expander edges in the shared type graph. Take half of these sets, and divide each one into two sets of $q/20$ elements. We now have three collections of $N/2$ sets of elements, one of sets of size $q/10$ and two of sets of size $q/20$. Assign the $N/2$ sets of each of the three collections to the $N/2$ edges of one of the three color classes. Dividing the elements of each set evenly between the two nodes incident to its edge assigns a total of $q/10$ elements of at most three different types to each node of the graph. The resulting shared type graph contains the desired degree-3 expander, and each edge of the expander has capacity at least $q/40$.

This procedure takes $q/10$ elements from each memory and puts the same number back into each memory. Therefore the movement of the $q/10$ elements from each memory to their desired locations can be scheduled to take only $q/10$ deletions and insertions from each memory. Thus in $q/10$ steps we can construct an expander in the shared type graph with capacity $q/40$ on each edge. By choosing the constant slowdown to be sufficiently large, we can insure that the types needed for this construction are not depleted by deletions during the $q/10$ steps used to construct the expander.

Given this expander, we perform a set of insertions and deletions by the users as follows. Mark the memories from which elements are being deleted as sinks. For each element of type $i$ being inserted, mark memory $i$ as a source. (Without loss of generality, we assume that the same number of elements are being deleted as are being inserted, and that no source is also a sink.) Since no source or sink has greater than constant multiplicity, the resulting flow problem can be solved in the expander graph with constant flow along each edge. Recall that the sources and sinks in this problem are not paired, so that flow can go from any source to any sink. Delete the requested elements from the sinks, and then for each unit of flow along an edge, move one element of that edge's shared type between its two incident memories in the direction of the flow. Finally, insert the new elements to the sources. It is clear that this procedure will require only a constant number of deletions and insertions from each memory, and that it will reduce the capacity on any expander edge by at most a constant amount.

Thus the shared type graph we have just constructed is sufficient to perform $\Theta(q)$ sets of insertions and deletions by the users with constant slowdown. Furthermore, by making the constant slowdown sufficiently large, the expander edges will be built quickly enough that there is always sufficient capacity for the next set of insertions and deletions by the users.

After we have constructed the expander graph for the new assignment, we have roughly $9q/10$ elements remaining in each memory from the old assignment. After holding aside enough of the remaining elements of each type to satisfy any future deletions, we choose a mapping of the remaining elements to the memories such that at most two additional types are assigned to each memory in the new assignment (see Lemma 1). Since the same number of elements will be removed from and put into each memory during this stage of the transformation, a coloring of the corresponding regular bipartite graph will yield a schedule for moving these elements to their positions in the new assignment using the same number of memory operations as there are elements to be removed from each memory. After every constant number of memory operations in this schedule, we pause and perform one set of insertions and deletions by the users, using the held-aside elements of each type to keep the deletions from affecting the scheduled movement of elements from the old assignment to the new assignment. When this is done, the remaining few elements in the old assignment are divided into $2N$ subtypes of roughly equal size and given positions in the new assignment by repeatedly deleting one element of each type and inserting the elements of subtype $j$ into memory $j \bmod N$, again pausing every constant number of steps to perform one set of insertions and deletions by the users.

Upon completion, we have moved all $q$ elements in each memory in the old assignment to their positions in the new assignment while performing $\Theta(q)$ sets of insertions and deletions by the users. The total number of types in each memory in the new assignment is at most eight: three from the expander construction, one from the insertions of type $i$ to memory $i$, two from the bulk movement of elements from the old assignment to the new assignment, and two from the 'cleaning up' at the end. By choosing the constant in the $\Theta(q)$ to be sufficiently small, we can insure that the expander edges are built quickly enough to handle the insertions and deletions by the users, and that there is sufficient total capacity on the expander edges to handle all $\Theta(q)$ sets of insertions and deletions by the users which must be performed during the transformation from the old assignment to the new assignment.

Thus we have shown how to perform $\Theta(q)$ sets of insertions and deletions by the users with constant slowdown while preserving the invariant that at most eight different types are stored in any memory. During intermediate stages of the procedure there may be more than eight different types residing in each memory, but there will never be more than twice that number (eight each from the old and the new assignments), so the number of types in each memory will always be at most a constant. By repeating this procedure as necessary, we can solve the distributed dynamic allocation problem near-optimally with constant slowdown ($T = O(1)$) and no wasted space ($\sigma = 0$). ■

We should note that all of the algorithms given in this section use global control for their allocation strategies. Considering again the motivating example of parallel disk allocation, we see that this is not a major drawback, as it is common for there to be a central file server which stores global information about the locations of each users' blocks of data and makes decisions about where to allocate new blocks. It would be interesting, however, to see if there exist equally efficient algorithms which use only local control.

## 3 Allocating Priority Queues in Networks

Placing the $N$ memories at the nodes of a fixed-connection network rather than allowing arbitrary unit-time communication between them makes constant-slowdown simulations of random-access virtual memories impossible, due to communication latency in the network. However, if we restrict our $M$ virtual memories to be linear arrays, each accessed through a single port, then efficient simulations are once again possible.

In this section we consider the problem of allocating $M$ linear arrays with total size bounded by $p$ on an $N$-node degree-$d$ network where each node contains a linear array of fixed size $q$. This problem arises in practice as the problem of maintaining priority queues of varying length in fixed-connection networks. As an example, consider the problem of routing a permutation of $n$ packets on an $n$-node network. Before and after the routing each processor has precisely one packet, but depending on the algorithm used to route the packets, large numbers of packets might converge at a few nodes during intermediate stages of the routing. For example, up to $\Omega(\sqrt{n})$ packets might have to be queued at a single node if an oblivious routing strategy is used [2]. Although we could handle these local space needs by allocating a factor of $\Theta(\sqrt{n})$ excess capacity to each local memory, it would be terribly wasteful to do so since we would only be using a total of $n$ out of $\Theta(n^{3/2})$ space overall at any point in time. It would be wiser to reallocate space across the

network from those nodes which are empty to those maintaining long queues.

We give deterministic, local-control algorithms to perform such allocation on either a butterfly or hypercube network. Both algorithms achieve constant slowdown and an arbitrarily high constant fraction usage of the total memory $qN$ when $p = \Omega(M \log M)$. For the butterfly algorithm, we have $N = O(M \log M + p)$, $q = O(1)$, and $d = O(1)$. For the hypercube algorithm, we have $N = O(M + p/\log M)$, $q = O(\log M)$, and $d = O(\log M)$.

In addition, we prove a lower bound which implies that any bounded-degree network on which we can solve the dynamic linear array allocation problem must have total memory $qN = \Omega(M \log M)$, even if $p = o(M \log M)$ and even if randomized algorithms are used. Therefore for any $p$, the allocation algorithm for the butterfly network is optimal for the entire class of bounded-degree networks. However, all of the algorithms discussed in this section are optimal in the sense that the total storage needed for the simulation is only slightly larger than the trivial lower bound of $p$ when $p = \Omega(M \log M)$, and in the sense that the simulations incur only constant slowdown. We also show that if the distribution of elements among the $M$ linear arrays is random, then we can allocate the $M$ linear arrays with total space $O(M \log\log M + p)$, with high probability.

### 3.1 Allocation Algorithms

In this section we describe a simple network with $N = O(M \log M)$ and $q = O(1)$ which can allocate $M$ stacks with $M \log M$ total elements, thus achieving constant-fraction usage of the available memory with constant slowdown. All of the algorithms described at the beginning of this section are extensions of this basic algorithm. The network consists of $M$ upper linear arrays of $5 \log M$ processors attached to the inputs of a back-to-back butterfly network, and $M$ lower linear arrays of size $\log M$ attached to its outputs. The $M$ ports are at the end processors of the upper linear arrays.

The network maintains the stacks by running in *cycles* of $O(\log M)$ steps. Each of the $M$ stacks will have some of its elements stored in its upper linear array, and the remainder stored in some subset of the lower linear arrays. In each cycle, $\log M$ stack operations are executed on the upper linear arrays, and then adjustments are made on each linear array in an attempt to keep the number of elements it contains between $2 \log M$ and $3 \log M$. The remainder of the elements will be stored (in blocks of size $\log M$) in some set of adjacent lower linear arrays. In addition, the sets of adjacent lower linear arrays used by the ports will be in the same left-to-right order as the ports, and will be packed to the left. The procedure for adjusting the upper linear arrays is as follows (illustrated in Figure 5): First the upper linear arrays calculate which ones need to send and which ones need to retrieve blocks of elements. Then in three stages they retrieve the needed blocks, adjust the locations of the elements in the lower linear arrays, and send blocks down to be stored in the lower linear arrays.

It is easy to show that the total running time of this adjustment procedure is $O(\log M)$ steps so that the allocation algorithm has constant slowdown, and that the network achieves constant-fraction usage of the total memory. This fraction can be brought as close to 1 as desired by making the lower linear arrays longer. Furthermore, this network can be simulated with constant slowdown by an $M \log M$-node butterfly or an $M$-node hypercube, and the algorithm can easily be extended to handle general linear arrays and the case where $p = \Omega(M \log M)$.

### 3.2 Lower Bounds on Total Memory

In this section we give lower bounds on the total space needed to solve the linear array allocation problem on networks, considering both deterministic and randomized algorithms. In proving our lower bounds, we use an off-line formulation of the problem called the path-finding problem, where we are given a final linear array length for each port in the network and the goal is to find disjoint paths of the appropriate lengths from the $M$ ports in the network. The main lower bound theorems for deterministic and randomized algorithms are as follows (stated without proof):

**Theorem 5** *Let $N = N(M)$ and $q = q(M)$, and assume $p = p(M) \geq (qN)^{\epsilon}$ for some $\epsilon \in (0, 1]$ and sufficiently large $M$. If $qN \log d = o(M \log M)$, then there does not exist any family of $N$-node degree-$d$ networks with deterministic path-finding algorithms for $M$ ports and $p$ total elements.* ■

**Theorem 6** *Let $N = N(M)$ and $q = q(M)$, and assume $p = p(M) \geq (qN)^{\epsilon + \frac{1}{2}}$ for some $\epsilon \in (0, \frac{1}{2}]$ and sufficiently large $M$. If $qN \log d = o(M \log M)$, then there does not exist any family of $N$-node degree-$d$ networks with randomized path-finding algorithms for $M$ ports and $p$ total elements where we cannot find some input on which the algorithm fails with probability exponentially close to 1.* ■

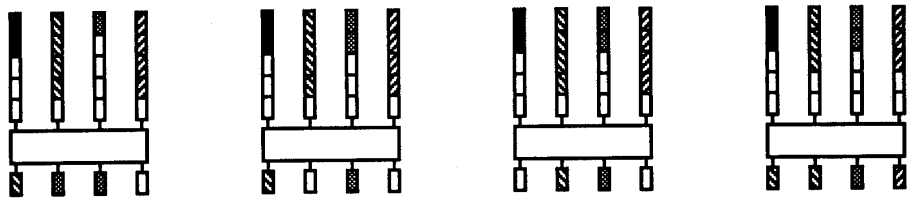

Figure 5: Adjusting the lower linear arrays at the end of a cycle.

The consequence of these theorems is that in order to allocate $M$ linear array memories on a fixed-connection network with no more than a constant fraction wasted space, using either a deterministic or a randomized algorithm, we must have $qN \log d = \Omega(M \log M)$. For bounded-degree networks, we therefore have $qN = \Omega(M \log M + p)$, which is achieved by the algorithm for the butterfly network given in Section 3.1, thus showing that it is optimal for the class of bounded-degree networks. Furthermore, we note that the fraction of the total memory being wasted can be made as small as desired.

### 3.3 Improved Algorithms for the Average Case

We have shown that space-efficient allocation of $M$ linear arrays can be performed when $p = \Omega(M \log M)$, and that when we restrict our attention to bounded-degree networks, this lower bound is optimal in the worst case. We now show that a substantial savings in memory can be achieved in the average case (i.e., when the assignment of elements to linear arrays is random) by a network and algorithm which will not work on every assignment of elements among the linear arrays, but will work with arbitrarily high probability over the space of all possible assignments. More specifically, we demonstrate a space-efficient algorithm for linear array allocation on a network with total memory $O(M \log\log M)$ which will correctly allocate $M$ linear arrays with high probability when the elements inserted into the network are assigned randomly to the linear arrays.

By dividing the $M$ linear arrays into groups of $\log M$ linear arrays each, we have that with high probability, each group of linear arrays will receive no more than $\Theta(\log M \log\log M)$ elements. Thus we can use our previous construction on each group of $\log M$ linear arrays to construct $M/\log M$ disjoint networks, each of which will solve the allocation problem for its group of linear arrays with total memory $O(\log M \log\log M)$. Together they form a bounded-degree (disconnected) network with $qN = O(M \log\log M)$, which with high probability will solve the average-case allocation problem with constant slowdown and constant-fraction wasted space. This network can be simulated on a butterfly or hypercube with total memory $O(M \log\log M)$.

## References


[1] S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in a nonblocking network. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 149–158, 1990.

[2] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 338–344, 1982.

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

[4] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations on PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[5] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, 1987.