REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
Feb. 1992

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
A Distributed Data-balanced Dictionary Based on the B-Link Tree

5. FUNDING NUMBERS

6. AUTHOR(S)
T, Johnson, A. Colbrook

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
MIT/LCS/TR 530

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N00014-89-J-1988

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Many concurrent dictionary data structures have been proposed, but usually in the context of shared memory multiprocessors. In this paper, we present an algorithm for a concurrent distributed B-tree that can be implemented on message passing parallel computers. Our distributed B-tree (the *dB-tree*) replicates the interior nodes in order to improve parallelism and reduce message passing. We show how the dB-tree algorithm can be used to build an efficient, highly parallel, data-balanced distributed dictionary, the *dE-tree*.

14. SUBJECT TERMS
concurrenct dictionary data structures, message passing multiprocessor systems, balanced search trees B-link trees, Replica coherency

15. NUMBER OF PAGES
19

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

A Distributed Data-balanced Dictionary Based on the B-link Tree

by

Theodore Johnson
University of Florida
Gainesville, FL 32611
ted@squall.cis.ufl.edu

Adrian Colbrook
MIT Laboratory for Computer Science
Cambridge, MA 02139
colbrook@concerto.lcs.mit.edu

February 1992

Abstract

Many concurrent dictionary data structures have been proposed, but usually in the context of shared memory multiprocessors. In this paper, we present an algorithm for a concurrent distributed B-tree that can be implemented on message passing parallel computers. Our distributed B-tree (the *dB-tree*) replicates the interior nodes ir order to improve parallelism and reduce message passing. We show how the dB-tree algorithm can be used to build an efficient, highly parallel, data-balanced distributed dictionary, the *dE-tree*.

Keywords: Concurrent dictionary data structures, Message passing multiprocessor systems, Balanced search trees, B-link trees, Replica coherency.

Adrian Colbrook was supported in part by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, by an equipment grant from Digital Equipment Corporation and by a Science and Engineering Research Council Postdoctoral Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

We introduce a new balanced search tree algorithm for distributed memory architectures. The search tree uses the B-link tree [1] as a base, and distributes ownership of the nodes among the processors that maintain the tree. We also *replicate* the non-leaf nodes, to improve parallelism. If we apply a read-one, write-all consistency maintenance rule [2], then reading a node becomes cheaper and writing to a node becomes more expensive as we increase the degree of replication. We observe that a node close to the root is often read, but rarely written to, but a node close to a leaf is rarely read, but is (relatively) often written to. Therefore, our ownership rule is that if a processor owns a leaf, it owns a copy of all nodes from the root to the leaf. The root is replicated at every node, so operations can be initiated in a fully parallel manner, while the often-modified nodes near the leaf are only moderately replicated, reducing the cost of their maintenance. Restructuring in a B-tree occurs on the path from the root to the leaf. A processor that restructures a node also has a local copy of the parent, so that restructuring decisions can be made locally, eliminating the need for centralized control and permitting further parallelism.

If the keys of a distributed data structure are arbitrarily or statically allocated to the processors, then some processors might be required to store a disproportionate share of the keys. Such an imbalance might cause a processor to exhaust its storage space even though other processors can store many more keys. A distributed search structure needs to be able to perform *data balancing* in order to efficiently use the storage that is available in the system. We show how data balancing can be efficiently implemented using the flexible distributed B-link tree algorithms.

1.1 Search Trees

Search trees are widely used for fast implementations of dictionary abstract data types. A dictionary is a partial mapping from keys to data that supports three operations: *insert*, *delete* and *search*. For simplicity we will assume that the dictionary stores no data with the keys and so may be viewed as a set of keys. A number of useful computations can be implemented in terms of dictionary abstract data types, including symbol tables, priority queues and pattern matching systems.

The B-tree was originally introduced by Bayer [3]. The B-tree algorithms for sequential applications were designed to minimize the response time for a single query and the sequential algorithm for a single search operation on a balanced B-tree has logarithmic complexity. The improvement in the response time that may be achieved by a parallel algorithm for a single search can at best be logarithmic in the number of processors used [4]. Therefore, for parallel systems a more important concern is increasing system throughput for a series of search, insertion and deletion operations executing in parallel.

The large number of concurrent search tree algorithms presented in the literature [1, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] prevents a complete description of each in this paper. Instead, we discuss the common issues that are addressed by all of the algorithms.

All concurrent search tree algorithms share the problem of managing contention. Concurrency control is required to ensure that two or more independent processes accessing a B-tree do not interfere with each other. A common approach is to associate a read/write lock with every node in the search tree [19]. This causes data contention as writers block incoming writers and readers, and readers block incoming writers. The contention is severe when it occurs at higher levels in the search tree, particularly at the root (often termed a *root bottleneck* [20, 21]). Similar problems are caused by resource contention. In a shared-memory architecture all of the processes trying to access the same tree node will access the same memory module on the machine. Similarly, for message-passing architectures, the processor on which a node resides will receive messages from every processor trying to access the node. Resource contention is again most serious for the higher levels in the search tree. Node replication [17] reduces contention but requires a coherence protocol to maintain consistency. We present two algorithms for managing replicated copies of tree nodes.

Associated with the contention issue is the problem of ***process overtaking***. This may occur when a process that holds a lock selects the next node it wishes to access, releases its lock and attempts to acquire a lock for the next node. A second process may acquire a lock on the next node between the original process releasing the old lock and acquiring the lock for the next node. The second process can then update the node in such a fashion as to cause the first process to lock the wrong node when it eventually acquires the lock. To prevent this kind of process overtaking many algorithms have their operations use ***lock coupling*** to block independent operations [11, 22]. An operation traverses the tree by obtaining the appropriate lock on the child before releasing the lock it holds on the parent. B-link trees [1, 12, 13] eliminate the need for lock coupling. If the wrong node is reached at any stage the operation is able to recover. This reduces the number of locks that must be held concurrently and increases throughput. We use the B-link tree as a base for the dB-tree.

1.2 Previous Work

Wang and Weihl [17, 18] have proposed that parallel B-trees be stored using ***Multi-version Memory***, a special cache coherence algorithm for linked data structures. Multi-version Memory permits only a single update to occur on a replicated node at any point in time (analogous to ***value logging*** [23, 24] in transaction systems). Our algorithm permits concurrent updates on replicated nodes (analogous to ***transition logging*** [23, 24]).

Ellis [25] has proposed algorithms for a distributed hash table. The directories of the table are replicated among several sites, and the data buckets are distributed among several sites. The hash table is a shallow search structure, so every update to the index structure must be distributed to every copy of the index. The distribution of updates can be limited in a distributed B-tree because it is a multilevel index structure. Also, the B-link tree is a very flexible data structure in which more sophisticated operations, such as range queries, can be easily implemented.

Peleg [26, 27] has proposed several structures for implementing a distributed dictionary. The concern of these papers is the message complexity of access and data balancing. However, the issues of efficiency and concurrent access are not addressed.

Colbrook et al. [28] have proposed a pipelined distributed B-tree, where each level of the tree is maintained by a different processor. The parallelism that can be obtained from this implementation is limited by the number of levels in the tree, and the distributed tree is not data-balanced.

The contribution of this paper is to present an efficient and highly parallel distributed B-tree algorithm (the *dB-tree*) that permits concurrent updates on a replicated tree node. We show how our distributed B-tree can be used to efficiently implement a highly parallel, data balanced distributed dictionary (the *dE-tree*) with good locality for neighboring leaves in the tree. In Section 2 we describe the dB-tree. In Section 3 we show how the dE-tree can be built from the dB-tree. Finally, conclusions are drawn in Section 4.

2 The dB-tree, a Concurrent Distributed B-tree

As a base for our distributed B-tree, we use the concurrent B-link tree [1, 12, 13]. The B-link tree algorithms have been found to have the highest performance among all existing concurrent B-tree algorithms [20, 21, 29]. Restructuring operations on B-link trees are performed one node at a time, so that the algorithms can be easily translated to a distributed environment. In a B-link tree, every node contains a pointer to its right neighbor. In the concurrent B-link tree algorithm, each node also contains a field, **highest**, which is the highest valued key that can be stored in the subtree rooted at that node. The B-link tree algorithms use the additional information stored in the nodes to let an operation recover if it misnavigates in the tree due to out-of-date information.

In the concurrent B-link tree algorithm described by Sagiv [12], insert operations place no more than one lock on the data structure at a time. Search operations start by placing an R (read) lock on the root and then searching the root to determine the next node to access. The search operation then unlocks the root and places a R lock on the next node. The search operation continues accessing the interior nodes in this manner until it reaches the leaf that could contain the leaf that it is searching for. If the operation reads a node and finds that the **highest** field is lower than the key it is searching for, it follows the right pointer of the node. When the search operation reaches the leaf that would contain the key that it is searching for, it searches the leaf for the key and returns success or failure depending on whether the key is or is not in the leaf.

An insert operation works in two phases: A search phase and a restructuring phase. The search phase on the insert operation uses the same algorithm as the search operation, except that it places W (exclusive write) locks on the leaf nodes. When the insert operation reaches the appropriate leaf, it inserts its key if the key is not already in the leaf. If the leaf is now too full, the insert operation must split the leaf and restructure the tree, as in the usual B-tree algorithm. The operations hold at most one lock at a time, so they must break the restructuring into disjoint pieces. So, when an insert operation finds that it must restructure the tree it performs a *half-split* operation (see figure 1). A half split operation consists of creating a new node (the sibling), transferring half of the node's keys to the sibling, and linking the sibling into the leaf list. In order to complete the split, the insert operation releases the lock on the node, locks the parent, and inserts a pointer to the sibling. If the parent becomes too full, the insert operation applies the same restructuring steps. Notice that for a period of time, there is no pointer in the parent to the sibling. Operations can still reach the sibling because of the right pointers and `highest` field.

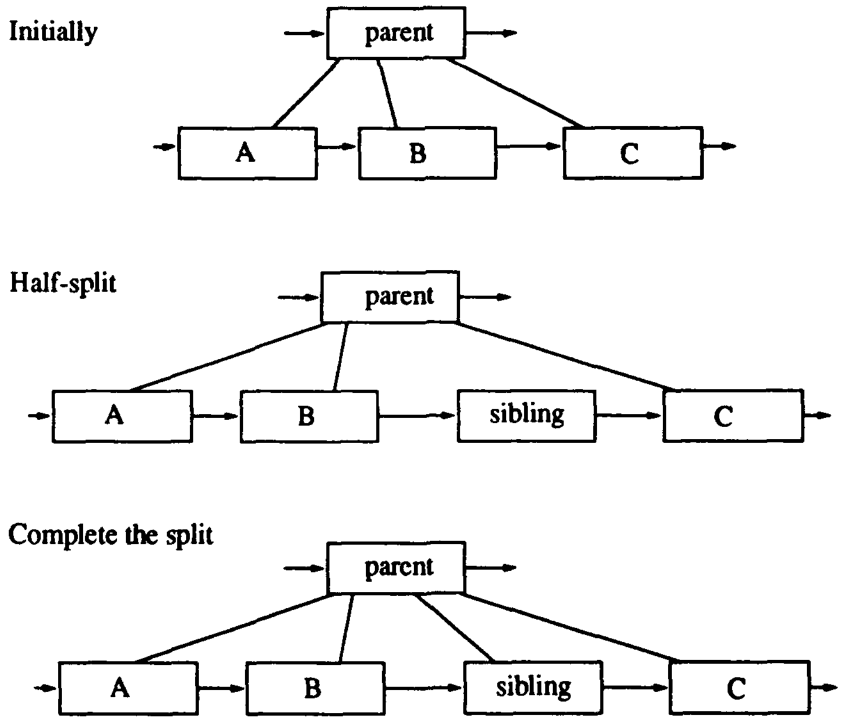


Figure 1: Half-split operation

The B-link tree in a shared memory multiprocessor does not easily support the deletion of nodes.

Lehman and Yao [1] recommend that nodes are never deleted. Sagiv [12] describes garbage collection algorithms for underfull nodes. Lanin and Shasha [13], and Wang [17] describe an algorithm that leaves stubs in place of deleted nodes. We show how nodes can safely be removed from the tree due to the lack of shared memory.

Our distributed B-tree algorithm builds on the concurrent B-link algorithms. An example dB-tree (distributed B-tree) is shown in figure 2. The leaves of the dB-tree are distributed among four processors. The interior nodes of the dB-tree are replicated among several processors. The rule for replicating the interior is that if a processor owns a leaf then it owns a copy of every node from the path from the root to the leaf, inclusive. Each processor on a level has links to both neighbors, not just the right neighbor. Keeping the nodes on a level in a double linked list simplifies replication control and the deletion of nodes (as will be discussed later), and also aids in downwards range queries.

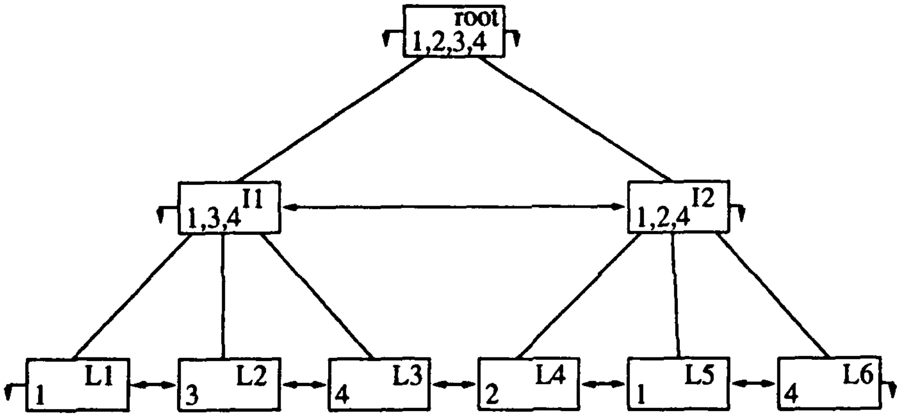


Figure 2: A dB-tree

The nodes of the dB-tree are distributed among the processors, which do not share a common memory space. Instead of naming the nodes with address, we will name them with *tags*. Each node has a unique tag. These tags can be generated by requiring each processor to keep a count of the objects that it creates. When a processor creates a node, it names it with the counter concatenated with the processor id. A pointer in a dB-tree node is a node tag together with a list of the processors that own a copy of the node. In order to access a node, the processor translates the tag to a local address through a translation table (organized as a hash table, for example).

The dB-tree has three *operations* defined on it: search(v), insert(v), and delete(v). These operations have the usual semantics. An operation on the dB-tree performs *suboperations* on the tree nodes in order to perform its computation. The suboperations are performed atomically by a processor on the processor's local data. Each processor has a queue of pending suboperations that need to be performed on its local nodes. A processor that helps to maintain the dB-tree accepts operations from other processors and performs the operation's suboperations. Conversely, a processor can send an operation to a different processor.

A search operation that originates at one of the participating processors starts by performing a ***node-search*** suboperation on the locally stored copy of the root. A search operation that originates at a processor that does not help maintain the dB-tree must transmit its request to one of the participating processors. The node-search suboperation is completely local: the node can be read in parallel at every processor that owns a copy of the node, and a node-search at one processor does not block an update of the same node at another processor. The node-search suboperation determines the next node to access. A copy of the next node on the search path may be stored locally, in which case the local copy should be used, or all copies may be stored remotely, in which case the processor must send the operation to a processor that stores a copy of the node.

As an example, if a search operation for a key stored in $L3$ originates in processor 3, then the operation reads processor 3's copy of the root and $I1$, then transmits a request to read $L3$ at processor 4. If a processor has a choice of sites to send the search operation to, it can make a choice based any reasonable criteria, such as locality or estimated load. For example, if a search request for a key stored at $L5$ originates at processor 3, processor 3 chooses between processors 1, 2, and 4 to service the request to search node $I2$. When the search operation reaches the correct leaf, it returns a **success** message to the originating processor if the key is in the leaf, and a **failure** message otherwise.

If an insert or delete operation does not cause restructuring, then it is executed in the same way as a search operation, except that the action on the leaf is to insert or delete a key. When restructuring is required, we can take advantage of the lack of shared memory - we know which processors have links to the nodes that we are restructuring. On the other hand, restructuring in the dB-tree is complicated by the need to keep all copies of a node consistent, and by the use of the doubly linked lists.

2.1 dB-tree Half-splits and Half-merges

The use of double links requires that splitting a node be carried out in three steps instead of two. The split procedure is shown in figure 3. When node B becomes too full, it splits into B and *Sibling*, each taking half of the keys that were stored in B. The *Sibling* node can be stored locally or transferred to a different processor. The next step is to make the leaf list consistent. Finally, a pointer to *Sibling* must be inserted into *parent*. The split procedure is carried out by performing suboperations on the nodes. First, a half-split suboperation is performed on node B. Next, a link-change suboperation is performed on node C. Finally, an insert suboperation is performed on *parent*. Note that there might be several copies of B, *Sibling* and *parent*. We will assume for now that modification suboperations are performed on all copies of a node atomically. Also note that the parent might have been split or merged, so that several search suboperations might have to be performed before the insert suboperation can be performed.

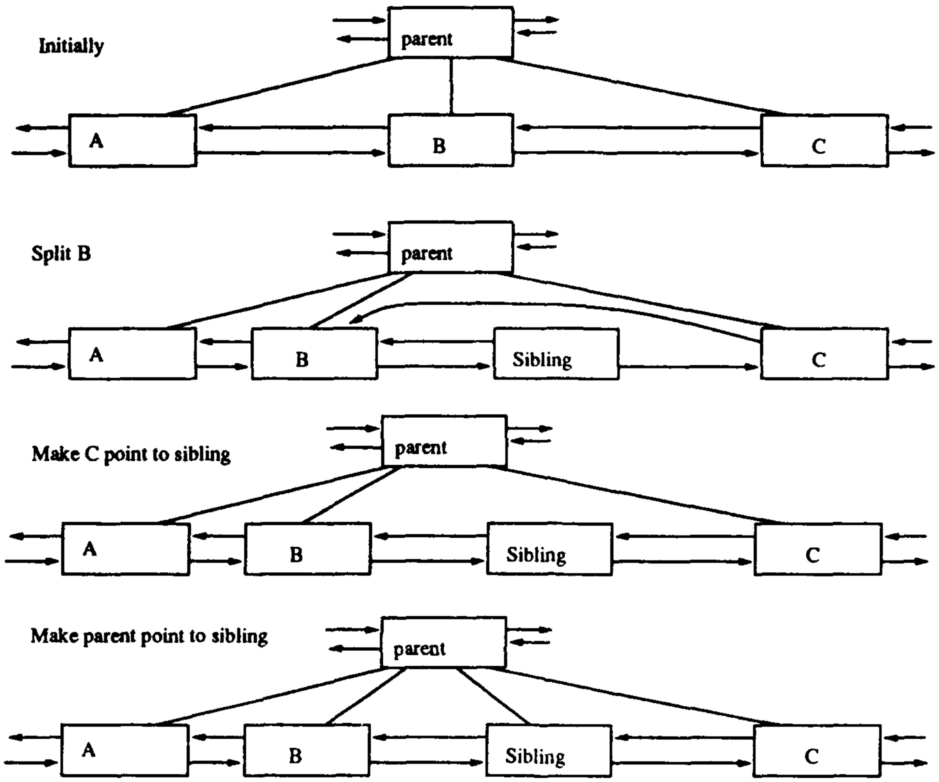


Figure 3: A dB-tree half-split

After a node is merged into another, or becomes empty, it must be removed from the tree. Figure 4 shows the procedure for removing a merged node from the data structure. The first step in deleting a node is to mark it as deleted, so that operations that navigate to it can recover their path. If the keys in *B* were moved to *A*, then a marker in node *B* should be set to indicate that operations that read node *B* should read node *A* next. The neighbors of the deleted node must be made to point to each other. Next, the parent's pointer to the deleted node is removed. The processors that own the parent and the neighboring nodes must inform the deleted node's owner when they have changed or removed their pointers. Deleted nodes cannot be disposed of until it is determined that there are no remaining references to them in the system.

When a processor splits or deletes a node, it must inform neighboring processors of the link changes. If two neighboring nodes are deleted at the same time, telling the deleted neighbor to change its link will not leave the tree in a correct state. Therefore, we require that a node restructure-lock the neighbors

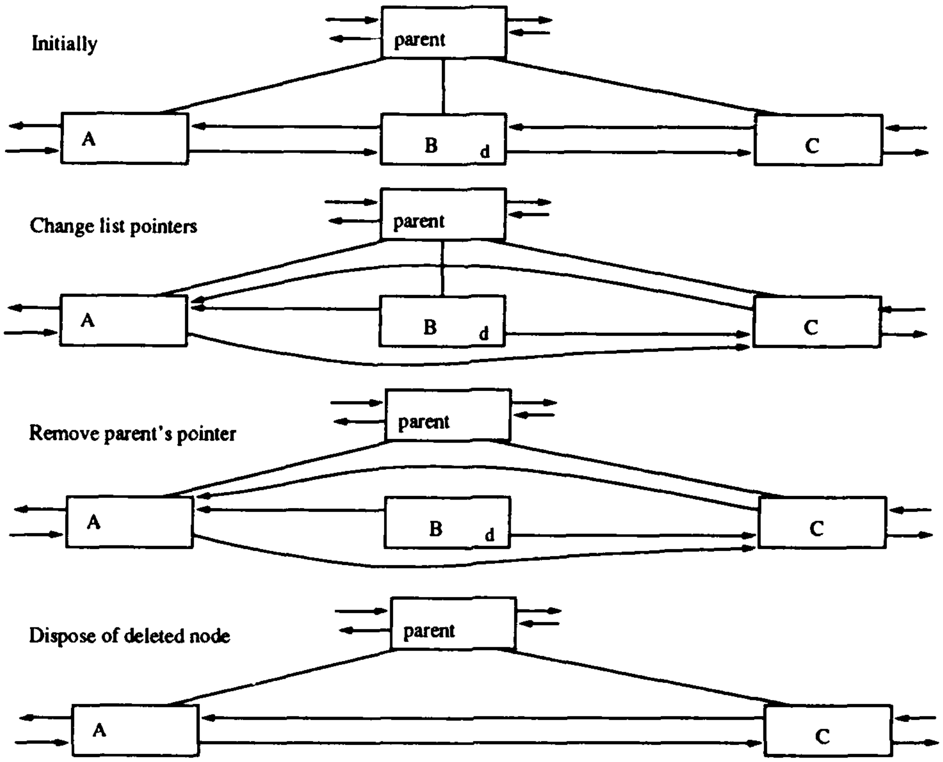
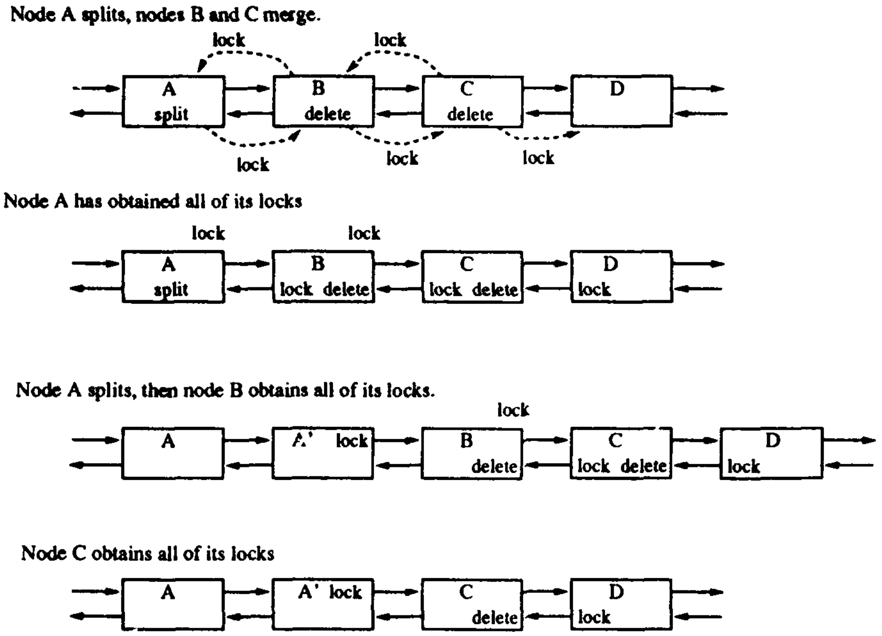


Figure 4: A dB-tree half-merge

who must change their pointers before they tell the neighbor to change the pointer. A node may be split or deleted only after all of the required restructure-locks have been obtained. Placing a restructure-lock on a node delays restructuring only; searches and non-restructuring inserts and deletes may continue to execute. Since delete operations must change both of their neighbor's nodes, there is the possibility of a deadlock. To break the deadlock, a request to change a left-link has priority over a request to change a right link. If a no le requests a restructure-lock on its right neighbor and receives a restructure-lock request from the right neighbor, it delays the response to the request. If a node requests a restructure-lock from its left neighbor and receives a request from the left neighbor, the node grants the restructure-lock. This assumes that lock requests from neighboring processors preempt the operation of the processor.

An example of the locking protocol is shown in figure 5. Node A decides to split and nodes B and C decide to delete at approximately the same time. Node A sends a lock request to its right neighbor (node B), and nodes B and C send lock requests to both neighbors. Only node A obtains all of the required locks, so node A splits, modifies the list, and passes the lock request to the new sibling A'. Node A' grants the lock request from node B, and so node B deletes, modifies the list, and passes the lock request from node C to node A'. Finally, node C obtains all of its locks so it may be deleted from the list.


Node A splits, nodes B and C merge.

Node A has obtained all of its locks

Node A splits, then node B obtains all of its locks.

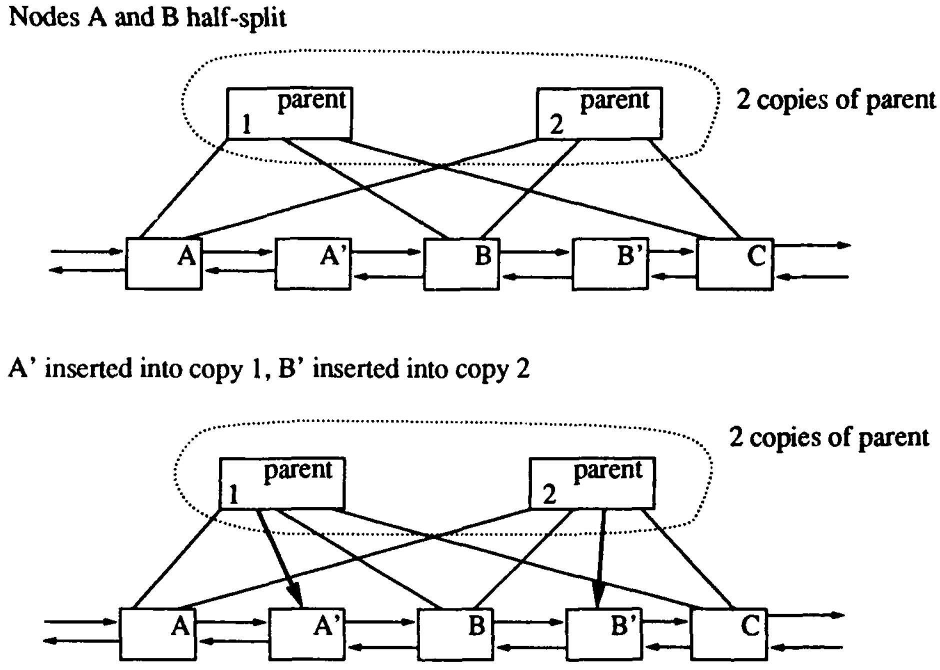
Node C obtains all of its locks

Figure 5: Locking for Splits and Merges

A processor does not need to explicitly restructure-lock the neighboring leaves, since the restructure-lock request is implicit in the request to change the link. A restructure-lock placed by a link-change request due to a split operation is released as soon as the link is changed. A restructure lock placed by a merge operation lasts until the deleted node is removed from the tree. This requirement ensures that a node may be safely deallocated when no tree links point to the node.

2.2 Integrating Concurrency Control with Replica Coherency

In the above discussion, we have assumed that modifications to nodes occur atomically. We can achieve the atomic updates by requiring that modifying suboperations lock every copy of the modified node before performing the update and block all reads and updates on the node, by using one of the well-known algorithms for managing replicated data [2, 30]. However, we can maintain our replicated nodes with far less synchronization and overhead. First, observe that it is not necessary to distribute the contents of the node on every modification, it is only necessary to distribute the modification itself. Second, the tree is never left in an incorrect state, so that search suboperations do not need to be blocked. Third, many of the modifying suboperations commute. Several authors have studied the issue of concurrency control on abstract data types when some operations may commute [31, 32, 33, 34, 35], but in the context of a transaction processing system. Our algorithms are similar to those described by Ellis [25] to maintain the replicated directories of the distributed hash tables in so that it is not always necessary that all suboperations are performed in the same order at all nodes. For example, insert suboperations may be performed out-of order. In the example shown in figure 6, nodes A and B have split, and pointers to the new nodes (A' and B') need to be inserted in the pointers to the parents. During the time from when a sibling is created to when a pointer to the sibling is inserted into the parent, the sibling can still be reached by the search, insert, and delete suboperations by the right and left pointers. So, if two inserts to the parent node are pending, it does not matter which is performed first, so that they can be performed in a different order in different copies of the parent node.


Nodes A and B half-split

A' inserted into copy 1, B' inserted into copy 2

Figure 6: Lazy inserts

Not all suboperations on a node can be performed in an arbitrary order. Consider, as one example, two copies of a node that is full. There are two suboperations that are pending on the node, an insert and a delete. Suppose that the insert is performed before the delete at the first copy, and the delete is performed before the insert at the second copy. The first copy will decide to split the node, and the second copy will not. The problem is not that inserts and deletes must be ordered, rather the problem arises because the split suboperation creates a new object. The creation and deletion of nodes are synchronization points. At the synchronization points the copies of the object must come to an agreement on the order of suboperations on the nodes.

We can categorize suboperations on nodes as being either ***synchronizing*** or ***lazy*** according to whether they do or do not require synchronization when they are performed. Table 1 lists the possible suboperations on the nodes and whether they are synchronizing or lazy. Performing an insert is a lazy suboperation, while performing a delete can be considered either lazy or synchronizing. While deletes can commute among themselves and among insert suboperations,[1] the owners of a deleted node must be notified of the edge deletion, and the deletion of the edge should not be arbitrarily delayed. We call the delete suboperation *lazy/synchronizing* to indicate that while the node updates do not need to be synchronized, some process needs to be informed when all updates have been performed. The link-change suboperations are also lazy/synchronizing. Split and merge suboperations require synchronization, because they create and destroy objects. A boundary change suboperation occurs when a leaf node transfers some of its keys to a neighboring leaf and is a lazy/sychronizing suboperation.

Creating a new copy of a node requires synchronization, because the new copy must be informed of all changes to the node. The neighboring nodes need to have the distribution lists on their edges updated, which can be accomplished by a variant of the link-change suboperation. Deleting a copy of a node can be lazy, if the following method is used. When an suboperation on a node arrives at a processor, the processor looks up the node's tag. If the processor finds the tag, it performs the suboperation. If the processor cannot find the tag, it sends the operation back to the requesting processor. When a processor receives a returned request, it notes that the returning processor no longer stores a copy of the node.

insert	lazy
delete	lazy/synchronizing
boundary change	lazy/synchronizing
split	synchronizing
merge	synchronizing
link change	lazy/synchronizing
join replication	synchronizing
leave replication	lazy

Table 1: Suboperation commutivity.

2.3 Replication Concurrency Control Algorithms

We present two algorithms for managing the replicated copies of a node. Both use the idea of a *birth processor*, the processor which is responsible for managing the replication of the object.

The first algorithm [17] is simply to send all modifying suboperations on a node to the birth processor, and the birth processor decides on a serial order in which the operations are to be performed. If the operation is lazy, the birth processor performs it locally, then relays the operation to all copies of the node. If the operation is synchronizing, the birth processor requires an acknowledgment of the operation,

[1] The insert and delete of the same object can commute if the delete is remembered and kills the insert

and delays further operations until all responses are received. If the parent level requires restructuring, the birth processor searches for the parent, then sends the modifying suboperation to the parent's birth processor.

The second algorithm is to allow non-birth processors to perform lazy operations locally and relay the operation to the other copies, but to send the synchronizing operations to the birth processor. When the birth processor receives a synchronizing operation, it locks all copies of the node. When a non-birth processor receives a lock from the birth processor, it delays all further operations on the node. When the birth processor receives all responses to the lock from all nodes, it will (if we assume network message ordering) have received all operations on the node, and can make a decision about which synchronizing operations need to be performed. For example, if an insert and a delete are performed on different copies of a full node, then the insert will trigger a request for a split, a synchronizing operation. The birth processor will lock all nodes and perform all lazy operations relayed from remote nodes as they arrive. After all lock responses are received, the birth processor will see that no action needs to be taken, and will release the locks. Note that in this algorithm, a node may continue to receive insert suboperations after it is too full. The node must temporarily store the overflow keys in overflow buckets until the birth processor performs the split.

When a new node is created (i.e, a node is split), a birth processor needs to be designated for the new node. The birth processor should be the processor that will own a copy of the new node, and which is the birth processor for the fewest nodes among all those who will own a copy of the new node. If the birth processor gives up its copy of the node, it must elect a new birth processor, and distribute the decision to all owners on the node (a synchronizing operation).

2.4 Parent Pointers

Requiring every node to contain a pointer to its parent simplifies the task of finding parents and splitting the root [28, 17]. Parent pointers can be maintained in the following manner: When a node splits, some of the node's children are transferred to the new sibling. When a processor receives the split suboperation from the birth processor, it splits the node locally, then makes all locally managed children of the new sibling point to the new sibling. Since owning the child implies owning the parent, every existing child of the new sibling will be made to point to the sibling. If we use merge-at-empty on the interior nodes, merging a node does not affect any children.

3 Distributing the Data : The dE-tree

A simple implementation of a distributed dictionary is to statically divide the key range among the processors and direct operations to the processor that manages the operation's key. The problem with the simple approach is that the processors will not be data-balanced. Even if each insert is equally likely to be directed to each processor, the processor with the most keys will hold considerably more keys than the average [36]. Data-balancing is necessary in order to distribute the request load, prevent processors from being required to devote a disproportionate share of their memory resources to the dictionary, and prevent out-of-storage errors when storage is available on other processors.

Ellis' algorithm [25] performs data-balancing whenever a processor runs out of storage. Peleg [26, 27] has studied the issue of data-balancing in distributed dictionaries from a complexity point of view, requiring that no processor store more than $O(M/N)$ keys, where M is the number of keys and N is the number of processors. In practice, this definition is simultaneously too strong and too weak, because it ignores constants and node capacities.

Our dB-tree provides us with the flexibility to perform data balancing among the processors, for the leaves of the tree can be distributed among the processors in an arbitrary fashion. If a processor splits a node, it can assign the newly created node to the processor that owns the fewest leaves.

Data balancing using the above mechanism is expensive, because every restructuring requires communication among the processors, and causes the internal nodes to be widely replicated. One way to reduce the communication costs is to try to have neighboring leaves stored in the same processor. We define an *extent* to be a maximal length sequence of neighboring leaves that are owned by the same processor. When a processor decides that it owns too many leaves, it first looks at the processors who own neighboring extents. If the neighbor will accept the leaves, the processor transfers some of its leaves to the neighbor. If no neighboring processor is lightly loaded, the heavily loaded processor searches for a lightly loaded processor and creates a new extent.

Figure 7 shows a four processor dB-tree that is data balanced using the extents. Notice that the extents have the characteristics of a leaf in the dB-tree: they have an upper and lower range, are doubly linked, accept the dictionary operations, and are occasionally split or merged. We can transform the extent-balanced dB-tree into the *dE-tree*: the distributed extent tree. Each processor manages a number of extents. The keys stored in the extent are kept in some convenient data structure. Each extent is linked with its neighboring extent. The extents are managed as the leaves in a dB-tree. When a processor decides that it is too heavily loaded, it first looks at the neighboring extents to take some of its keys. If all neighboring processors are heavily loaded, a new extent is created for a lightly loaded processor. The processor that manages the new extent can chosen according to a number of heuristics that examine factors such as data load, processor capacity, communication locality, and degree of replication. The creation and deletion of extents, and the shifting of keys between extents in the dE-tree correspond to splitting and merging leaves in the dB-tree.

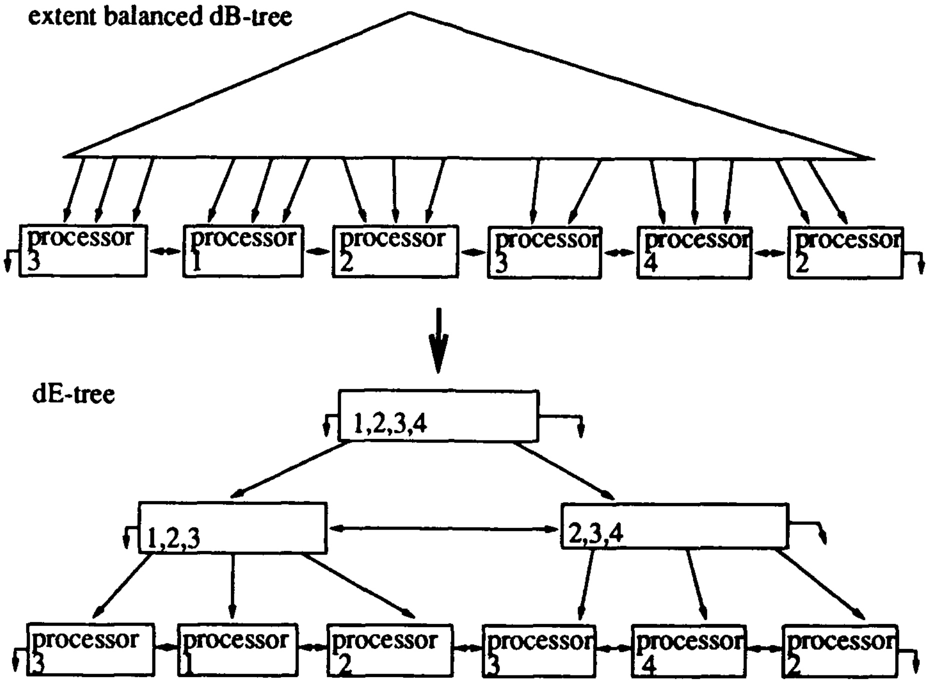


Figure 7: The dE-tree

3.0.1 Propagating Load Information

Information about the load on processors must be distributed throughout the system so that informed decisions for creating and placing new extents can be made. The design of a load propagation mechanism must address the tradeoff between propagating information quickly to all processors, which can require significant communication bandwidth to distribute the load information, and propagating information either too slowly or to too few processors, which can result in load imbalance and processors sitting idle because the system is slow to react to changes in the load on individual processors.

In addition, many load-balancing techniques become unstable in large systems. Instability can arise because information is propagated too slowly, so that many processors may think another is lightly loaded long after it becomes heavily loaded and continue to create tasks there. It can also arise because decisions to create new extents are made quickly when the load on another processor changes, so that many new extents are moved to a processor that suddenly becomes lightly loaded. To avoid these problems, information must be propagated reasonably rapidly through the system, but the number of processors that will react quickly to a change in load on a particular processor must be limited.

Suitable load propagation mechanisms include probing [37], gradient methods [38], processor pairing [39, 40], drafting algorithms [41] and bidding algorithms [42]. The ultradiffusive algorithms of Lumer and Huberman [43], which uses a hierarchical control structure, are particularly well suited when the number of processors is large.

4 Conclusions

We present a distributed dictionary based on the B-link tree, the dB-tree. The interior nodes of the tree are replicated in order to allow many processors to read the same node and thus increase parallelism. The degree of replication decreases as one descends from the root, in order to control the cost of maintaining replica coherency. Update operations on the interior nodes never block operations in their search phases and can proceed concurrently on the same replicated node in some cases, further increasing the parallelism of the tree. We describe two efficient algorithms for maintaining coherent replicas.

We use the flexible dB-tree to construct an efficient data-balanced dictionary, the dE-tree. The dE-tree assigns key ranges to processors, and a processor may maintain several key ranges. We use the dB-tree to allow the key ranges to be created, deleted, or modified without centralized control.

We plan to implement and experimentally analyze the dB-tree and the dE-tree, investigate data and load balancing strategies, and investigate methods for supporting transaction concurrency control and recovery.

5 Acknowledgments

We thank William Weihl for his comments and suggestions on this work.

References

[1] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[2] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[3] R. Bayer. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] M.J. Quinn. *Designing efficient algorithms for parallel computers*. McGraw-Hill, New York, 1987.

[5] R. Bayer and M. Schkolnick. Concurrency of Operations on B-trees. *Acta Informatica*, 9:1–22, 1977.

[6] R.E. Miller. Multiple access to B-trees. In *Proceedings of the 1978 Conference on Information Sciences and Systems*, pages 400–408, Johns Hopkins University, Baltimore, MD, 1978.

[7] C. Ellis. Concurrent search and insertions in 2-3 trees. *Acta Informatica*, 14(1):63–86, 1980.

[8] Y. S. Kwong and D. Wood. A New Method for Concurrency in B-trees. *IEEE Transactions on Software Engineering*, SE-8(3):211–222, May 1982.

[9] G. Lausen. Integrated concurrency control in shared B-trees. *Computing*, 33:13–26, 1984.

[10] D. Shasha and N. Goodman. Concurrent search tree algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.

[11] Y. Mond and Y. Raz. Concurrency control in B+-trees using preparatory operations. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 331–334, 1985.

[12] Y. Sagiv. Concurrent Operations on B-Trees with Overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, October 1986.

[13] V. Lanin and D. Shasha. A Symmetric Concurrent B-Tree Algorithm. In *1986 Proceedings Fall Joint Computer Conference*, pages 380–386, November 1986.

[14] D. Shasha, V. Lanin, and J. Schmidt. An Analytical Model for the Performace of Concurrent B Tree Algorithms. Ultracomputer note 124, New York University, July 1987.

[15] A. Biliris. Operation specific locking in B-trees. In *Symposiun on the Principles of Database Systems*, pages 159–169. ACM SIGACT-SIGART-SIGMOD, 1987.

[16] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. Research Report RJ 6864, IBM, 1989.

[17] P. Wang. An in-depth analysis of concurrent B-tree algorithms. Master's thesis, MIT Laboratory for Computer Science, 1991.

[18] W.E. Weihl and P. Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, 1990.

[19] V. Lanin, D. Shasha, and J. Schmidt. An analytical model for the performance of concurrent B-tree algorithms. Technical report, Ultracomputer Laboratory, New York University, 1987.

[20] T. Johnson and D. Shasha. A Framework for the Performance Analysis of Concurrent B-tree Algorithms. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, April 1990.

[21] T. Johnson. *The Performance of Concurrent Data Structure Algorithms*. PhD thesis, NYU Dept. of Computer Science, 1990.

[22] L.J. Guibas and R. Sedgewick. A dichromatic framework for balancing trees. In *Proceedings of the 19th Annual IEEE Computer Society Symposium on the Foundations of Computer Science*, pages 8–21, 1978.

[23] W.E. Weihl. The impact or recovery on concurrency control. Technical Report MIT/LCS/TM-382b, MIT Laboratory for Computer Science, 1989.

[24] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, Massachusetts, 1986.

[25] C.S. Ellis. Distributed data structures: A case study. *IEEE Transactions on Computing*, C-34(12):1178–1185, 1985.

[26] K.Z Gilon and D. Peleg. Compact deterministic distributed dictionaries. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 81–94. ACM, 1991.

[27] D. Peleg. Distributed data structures: A complexity oriented view. In *Fourth Int's Workshop on Distributed Algorithms*, pages 71–89, Bari, Italy, 1990.

[28] A. Colbrook, E.A. Brewer, C.N. Dellarocas, and W.E. Weihl. An algorithm for concurrent search trees. In *Proceedings of the 20th International Conference on Parallel Processing*, pages III138–III141, 1991.

[29] V. Srinivasan and M. Carey. Performance of B-tree concurrency control algorithms. Technical Report Computer Sciences Technical Report 999, University of Wisconson-Madison, 1991.

[30] D.K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Annual ACM Symposium on Operating System Principles*, pages 150–150. ACM, 1979.

[31] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[32] T.P. Ng. Using histories to implement atomic objects. *ACM Transactions of Computer Systems*, 7(4):360–393, 1989.

[33] W.E. Weihl and B. Liskov. Implementation of resilient, atomic data objects. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, 1985.

[34] P.M. Schwartz and A.Z. Spector. Synchronizing abstract data types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[35] W.E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[36] K. Donovan. Performance of shared memory in a parallel computer. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):253–256, 1991.

[37] Christos G. Cassandras, James F. Kurose, and Don Towsley. Resource contention management in parallel systems. Radc-tr-89-48, Rome Air Development Center (RADC), April 1989.

[38] F.C.H. Lin and R.M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, SE-13:32–38, 1987.

[39] R.A. Finkel, M.H. Solomon, and M. Horowitz. Distributed algorithms for global structuring. In *Proceedings of the National Computer Conference*, pages 455–460, 1979.

[40] R.M. Bryant and R.A. Finkel. A stable distributed scheduling algorithm. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 314–323, 1981.

[41] L.M. Ni, C.W. Xu, and T.B. Genfreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, SE-11:32–38, 1985.

[42] K. Hwang et al. A unix-based local computer network with load balancing. *IEEE Computer*, pages 55–64, 1982.

[43] E. Lumer and B.A. Huberman. Dynamics of resource allocation in distributed systems. Preprint (submitted to IEEE Transactions on SMC), Xerox Palo Alto Research Center, March 1990.

OFFICIAL DISTRIBUTION LIST

DIRECTOR 2 copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Boulevard
Arlington, VA 22209

OFFICE OF NAVAL RESEARCH 2 copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. Gary Koop, Code 433

DIRECTOR, CODE 2627 6 copies
Naval Research Laboratory
Washington, DC 20375

DEFENSE TECHNICAL INFORMATION CENTER 2 copies
Cameron Station
Alexandria, VA 22314

NATIONAL SCIENCE FOUNDATION 2 copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

HEAD, CODE 38 1 copy
Research Department
Naval Weapons Center
China Lake, CA 93555