LIST ACCESS PROCEDURES FOR THE ULTRACOMPUTER

Craig Williams
Paul F. Reynolds, Jr.

Computer Science Report No. TR-87-03
February, 1987

LIST ACCESS PROCEDURES FOR THE ULTRACOMPUTER

TR-87-03

Craig Williams
Paul F. Reynolds, Jr.

Department of Computer Science
University of Virginia

February, 1987

ABSTRACT

We describe a set of asynchronous procedures for accessing an unsorted linear linked list in shared memory. The procedures are designed for highly parallel computation on the ultracomputer and support search, insertion at the tail of the list, and deletion from anywhere within the list. The procedures maintain the consistency of the list without using high-level locks or any other kind of high-level critical section. No type of access locks out any other type of access, and any number of searches, insertions, and deletions can occur concurrently. Our simulation of the access procedures indicates that the time required for a list access can remain almost constant as the level of concurrency increases. We show that the procedures can be used to implement the wound-wait and wait-die protocols for accessing a database and as the basis for a space-efficient implementation of P and V operations that maintains FIFO order among waiting processes. Neither the database protocol nor the semaphore implementation itself introduces serialization. We also describe a parallel garbage collection algorithm based on reference counts which should collect garbage more promptly than marker-based algorithms and which can be used even when processes hold pointers into the shared structure exclusively in local memory.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The advent of highly parallel systems that communicate through shared memory has created a design problem: "How do we define data structures for highly parallel systems so that operations on the structures do not become bottlenecks?" Operations for accessing data structures typically involve some code that must be executed atomically to maintain the consistency of the structure. It is a simple matter to adapt an access operation defined for a uniprocessor for parallel execution if correctness is the only criterion --- place code that must be executed sequentially within a critical section or allow only a single server process direct access to the structure. In a system with only a few processors, the delay introduced by the restricted access will usually be tolerable since only a few processes can contend for access. As more processors are added, this straightforward adaptation of procedures designed for serial execution becomes unacceptable. If a data structure is frequently accessed by each of hundreds of processes, a critical section protecting the structure will tend to introduce a delay that lasts hundreds of times the time required to execute the code within the critical section. The effective use of data structures in highly parallel shared memory systems requires rethinking the design of data structures and the operations defined for accessing them.

In this paper we consider the problem of defining asynchronous, highly concurrent procedures for accessing a linear linked list in shared memory. Previous research concerning highly concurrent operations for accessing data structures in asynchronous, shared memory systems has focused on individual shared variables [Lam77, Pet83], heaps [Jon85, Yoo83], queues [GLR83, QuY84, Rud82], pools [Man86], and search trees. The search tree literature is the most extensive. Researchers have designed parallel access procedures for several varieties of search trees, including B-trees and variations of B-trees [BaS77, BiF85, KwW82, LeY81, Sam76], 2-3 trees [Ell80a], AVL trees [Ell80b], and binary search trees [KuL80, Man84, MaL84].

In all of the research cited above, the goal has been to increase the effective level of parallelism in accessing a shared data structure by reducing the prominence of the critical section(s) protecting the structure. In some cases the critical section is eliminated [Lam77, Pet83]. More commonly a higher effective level of parallelism is achieved by reducing the portion of the structure each critical section protects, the number of processes that may need to wait at a given critical section, or the execution time within the critical section.

1.1. Procedures for Accessing a Linked List

We propose a set of procedures for accessing a linked list that is free of high-level critical sections. The procedures are designed for a multiprocessor, such as the NYU Ultracomputer or IBM's RP3, in which the processing elements are connected to a bank of memory modules via a nonadaptive, multistage, switching and combining network. We describe some of the basic features of the ultracomputer in the next chapter. The ultracomputer architecture is attractive as an architecture for a highly parallel shared memory multiprocessor because the switching network is capable of combining memory access instructions issued by different processors when the instructions address the same shared variable. Without this combining capability, each access to a shared variable would introduce an implicit critical section. Multiple accesses to the same shared variable would be serialized, limiting the effective level of parallelism in a way not apparent on the face of the code. The combining capability also augments the processing power of the system by allowing the network itself to perform certain "fan-in" computations such as the addition of addends applied to the same shared variable.

We make no assumptions about the relative rate at which processes execute. Processes may execute asynchronously or in lockstep (but in the latter case a more efficient algorithm that takes advantage of the synchronous operation can probably be devised). We do not deal with issues of fault tolerance. We assume that no process fails while accessing the list.

The list access procedures support the following operations:

Insertion:
append an element to the tail of the list.

Deletion:
remove an element from anywhere in the list.

Search(target):
searching from the head, locate the first occurrence of an element of type "target" on the list.

Insertions are permitted only at the tail of the list, so the list generated by repeated execution of these procedures is unsorted. Note that the search procedure we define in chapter 3 not only looks for the first occurrence of an element of type "target" but also invalidates and attempts to delete the element it finds.

Insertions, searches, and deletions may occur concurrently and asynchronously. Excepting delays caused by network congestion, the only time a process must wait is when it deletes an element that another process is visiting. (Some kinds of searches also require a searcher to wait when it visits an incompletely inserted element.) We require that the list access procedures be suitable for use in a continuously operating program such as an operating system or database application, so provision must be made for returning deleted elements to the free space list. A deleting process waits until the other processes leave the deleted element before appending it to the free space list. Maintenance processes may be used to relieve user processes of the task of adding elements to the free space list and other housekeeping chores, but the use of maintenance processes is optional, and, we believe, not necessary for efficient operation.

1.2. Motivation for the list access procedures

The procedures we describe are intended for use in applications requiring the flexibility of a linked representation. A linked representation of a list allows space to be allocated dynamically for list elements and allows arbitrary elements to be deleted without a costly compacting operation. If the average length of the list is close to the upper bound on the list length and the list is treated as a queue or stack, an array representation is preferable. The procedures for inserting and deleting from a stack or queue are simpler because they do not need to interact with a search procedure and the random access to list elements permitted by the array representation facilitates parallel deletion.

We expect the list access procedures to be useful for lists containing elements of a small number of distinct types. If the elements are fungible, no search procedure is required. The list can be treated as a queue or stack and represented as an array. On the other hand, if there are a large number of distinct types of elements on the list, a search tree representation is probably necessary for acceptable performance. One way in which lists containing a small number of distinct types of elements may arise in practice is from hash table collisions. An application of the list access procedures to lists generated by hash table collisions in a data base application is described in Chapter 4.

We note that the results of a simulation of the list access procedures indicate that the average time for a list access increases as the number of processes increases when the list is searched from the head. When searches start from the insertion site, i.e., from the tail, the average time for a list access remains almost constant as the number of processes increases. We expect, therefore, that the list access procedures will be useful only in applications where elements that have been on the list a short time can be removed from the list before elements of the same type that have been on the list longer.

Even apart from specific applications, the design of concurrent access procedures for a linked list is interesting as an instance of the generic problem of designing data structures for highly parallel systems, as an exercise in coordinating parallel processes, and as a new implementation for a basic data structure.

1.3. Related work

Our research revealed no other critical-section-free algorithm for accessing a linked list suitable for continuous operation. In discussing the problem of finding data structures for a highly parallel operating system, Edler, Gottlieb, and Lipkis [EGL85] comment:

> We know of no algorithm for deleting items in a linked list without locking out some other accesses, or for assigning individual list items to different PEs without serialization. The desirable characteristics of linked lists must be found in other structures.

The only work we found which specifically addresses the problem of providing concurrent access to a linked list is due to Wyllie [Wyl79]. Wyllie notes that a process may delete an element from a list in which dummy elements alternate with real elements by simply linking the dummy elements which are the neighbors of the element to be deleted. The dummy elements insulate deleting processes from

interference. Even if other deleters are active nearby, the deletion will not disconnect the list because the dummy elements are not deleted. Since no provision is made for removing dummy elements or determining when elements removed from the list may be added to the free space list, the algorithm is unsuitable for continuous operation.

Wyllie also proposes a concurrent deletion procedure for synchronous execution that uses recursive doubling to locate the ends of each sublist of invalid elements. Deleting processes cooperate to reduce the time required to locate the ends of a sublist of invalid elements from a linear to a logarithmic function of the length of the sublist.

The work most closely related to our own is Kung and Lehman's research on concurrent access operations for binary search trees [KuL80]. Since a linear list is a special case of a tree, any set of access procedures for a tree, where the tree is represented as a linked structure and is not constrained to be balanced, can also serve as access procedures for a linked list.

The procedures described by Kung and Lehman protect the consistency of the tree by locking individual elements. Processes searching the tree can pass through the locks --- the locks restrict only strong searches that end at a locked element and contiguous deletions and insertions. When a deleting process finds its target element, it locks the parent of the element, then the element itself, and reexamines the target element to ensure that it was not removed before the locks were placed. Since the element pointed to by the parent pointer in the target element may have been deleted from the tree before the lock was secured, the process may have to lock and unlock several elements before it finds an ancestor of the target element that is still in the tree. The deleting process then adjusts pointers to route around the target element, marks it deleted, places it on a garbage collection queue, and unlocks both elements. Contiguous deletions are disallowed by the requirement that a deleting process lock both the target and the parent of the target element.

We note that when we simulated a modified version of Kung and Lehman's algorithm, we found that it performed poorly, i.e., the average time for an access increased markedly as the number of processes accessing the list increased. (The modifications we made to Kung and Lehman's algorithm were to pare it down to the procedures relevant to accessing a linked list, adapt it to the Ultracomputer,

and replace the insertion procedure with the nonlocking insertion procedure we propose.) This simulation is briefly described in appendix C. However, when we further modified the Kung and Lehman's algorithm to limit the number of processes that may need to wait at a given lock, we found that the algorithm performed better than the procedures we propose by a small constant. For both Kung and Lehman's algorithm, as modified, and the procedures we propose, the average time for a list access is a very slow growing function of the number of processes accessing the list. We were able to limit the number of processes that may have to wait at a given lock by requiring that a deleting process first invalidate the target element before attempting to secure any locks. This simple modification provides a procedure for selecting a single process from among several processes that might otherwise decide to attempt to delete the element and compete for locks. The simulation results suggest that it is not necessary to eliminate critical sections in shared data structures for highly parallel systems; limiting the number of processes that may wait at each critical section can also prevent a shared data structure from becoming a bottleneck.

Other related work includes that of Rudolf and the Ultracomputer group on the Ultracomputer queue and flexible queue [GGK83, Rud82], Manber on the concurrent pool [Man86], and Quinn and Yoo on concurrent queues and dequeues [QuY84]. This research is less closely related to our own because the data structures investigated are not intended to be searched. The deletion procedure defined for these data structures deletes the first element or an arbitary element and no provision is made for searching for an element with a specific data value. A common theme in their research is the tradeoff between the goals of limiting contention and the need for synchronization on the one hand and balancing workloads on the other. If each process maintains its own pool or queue, then every process has contention-free access to elements in its own pool (or queue), but the size of each pool may vary greatly. A process may become idle because there are no elements in its local pool although large pools of elements exist elsewhere in the system. Manber proposes procedures by which a process can locate and split off elements from another process's pool when its own is empty. Quinn and Yoo describe three different representations for a shared queue --- the linked array, the multiple queue, and the the parallel semiqueue --- each representing a different resolution of the tradeoff. These three representations all have serious draw-

backs, either in synchronization or space overhead or the potential for workload imbalance. Probably the best representation for a shared queue or pool is the flexible queue, designed for the Ultracomputer. We describe both the flexible queue and the Ultracomputer queue in Chapter 2.

1.4. Organization of this thesis

In Chapter 2 we describe the model of computation and the target architecture, the ultracomputer, for which we design list access procedures. We describe the list access procedures in Chapter 3 and give an informal correctness argument. A database application, a variant of an algorithm proposed by H.S. Stone, is described in Chapter 4. In the final chapter we summarize the thesis and propose topics for further research.

CHAPTER 2

THE ULTRACOMPUTER

In this chapter we describe the ultracomputer, the target architecture for the list access procedures we propose in chapter three. We also describe the ultracomputer queue and flexible queue, two highly concurrent data structures designed for the ultracomputer.

The ultracomputer is well documented, so our description will be brief. For a more detailed description, see [GLR83, GGK83, KrR85, Rud82]. We note that there are at least two different architectures referred to as the "ultracomputer", the shared memory ultracomputer represented by the NYU Ultracomputer and the RP3, and an ultracomputer without shared memory [Sch80]. We will always use "ultracomputer" to mean the shared memory version.

2.1. The Architecture

The ultracomputer is a parallel architecture designed to approximate the power of the paracomputer model of computation [Sch80]. The paracomputer allows each process to read or write any location in shared memory in a single step. The result of simultaneous writes to the same memory location is that the memory location takes on an arbitrary value from among those written to it. Fan-in limitations make the paracomputer a strictly theoretical model, a tool for reasoning about the complexity of parallel algorithms.

Unlike the paracomputer, the ultracomputer is realizable. (NYU and IBM are building ultracomputers.) The ultracomputer, however, only approximates the power of the paracomputer. Simultaneous access to memory is realized in the ultracomputer by fanning-in reads and writes through an Omega network. Note that we assume that the switches in an ultracomputer's Omega network are intelligent switches, capable of combining certain memory operations. A multiprocessor can be built around an Omega network that lacks any combining power, e.g., BBN's butterfly [Tho86].

As illustrated in figure 2.1, each memory module corresponds to the root of a complete tree in which the processors correspond to leaves and the switches to interior nodes. If two writes which address the same memory location collide at a switch in the network, the switch arbitrarily chooses one to forward. If two reads collide, the switch forwards only one, then satisfies both with the value returned. If a read and write collide, the read is satisfied with the value of the write and the write is forwarded. Thus every processor can simultaneously access a given memory location but the time for a memory access is a logarithmic function of the number of processors, not the single time unit assumed for the paracomputer model. A second way in which the ultracomputer is less powerful than the paracomputer is that processes cannot simultaneously address arbitrary memory locations. In theory, each memory module can hold only a single word of memory, but in practice, each module holds several words, only one of which can be accessed at a time.

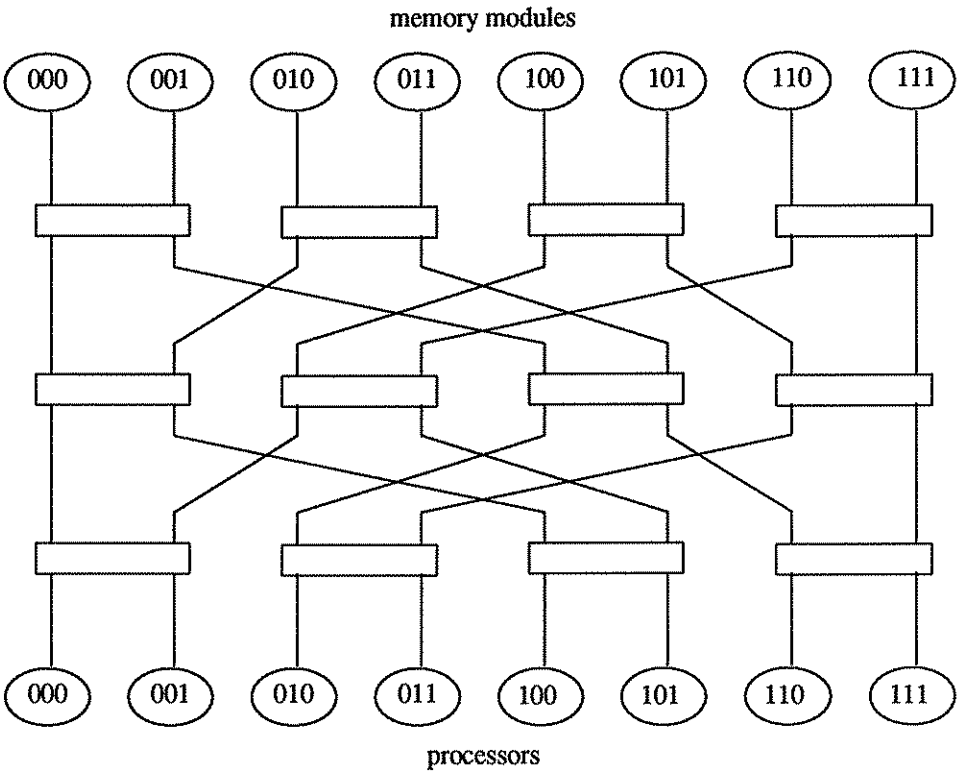
Although it takes logarithmic time to perform a read or write on an ultracomputer, it also takes logarithmic time to perform more complex memory operations, such as parallel prefix computations, which would also require logarithmic time on a paracomputer [KrR85]. (Note however that the ultracomputer operates asynchronously. There is no assurance that the operands will enter the network simultaneously or progress through the network at the same rate.) The source of this power to perform efficient combining of more complex operations is the switching network. The switches can be designed to do more than fan-in reads and writes. They can, for instance, be designed to manage swaps and add inputs.

The use of enhanced switches in the switching network allows the ultracomputer to support a fetch-and-* memory access operation, where * is some binary associative or ''pseudo-associative'' operation [KrR85]. Reads and writes are specific instances of this operation. We will describe only two other operations aside from the read and write operations: fetch-and-add, and swap. The fetch-and-add operation is used in the ultracomputer queue and flexible queue.

Execution of the fetch-and-*(V,e) operation is equivalent to atomic execution of the following statements:

```
temp := V;
V := *(V,e);
return temp,
```

Figure 2.1 Omega Network



where V is an integer variable and e an integer-valued expression. Note that read, write, and swap can each be expressed in the format of the fetch-and-* operation: read as fetch-and-#(V,x), and write and swap as fetch-and-$(V,e), where #(a,b) is defined to equal a, $(a,b) is defined to equal b, and "x" denotes an arbitrary value. The difference between write and swap is that the value returned from write is not used and so can be an arbitrary value. Swap(V,e) is thus equivalent to atomic execution of

```
temp := V;
V := e;
return temp,
```

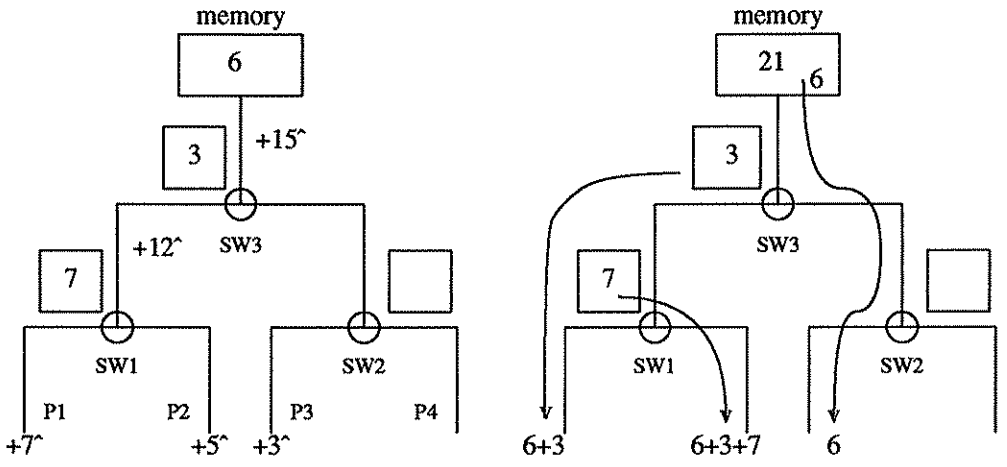
where V is a shared variable and e is an expression of the same type as V.

Fetch-and-* operations have the important property that they are combinable in the switching network. Combined execution of both homogenous (involving the same * operator) and inhomogenous (dif-

ferent * operators) operations obeys the serialization principle: concurrent execution of the operations produces the same effect as serial execution of the same operations in some arbitrary order. Operations that obey the serialization property have obvious value for concurrent computation: if every serial execution of the operations is correct, then concurrent execution of the operations is also correct. We can reason about parallel algorithms that access shared memory using operations that obey the serialization principle without considering all the many ways in which execution of the operations may interact in the switching network. Kruskal and Rudolph have formally demonstrated that the fetch-and-* operation as implemented on the ultracomputer obeys the serialization principle [KrR85].

Concurrent execution of fetch-and-* operations is realized by the network switches, which combine operations addressed to the same memory location and fan-out the result returned from the combined operation. We will illustrate the operation of the switches by explaining how a switch combines fetch-and-add operations. Figure 2.2 shows the concurrent execution of three fetch-and-adds. The small circles in the figure represent switches and the boxes beside the circles represent buffers associated with the switches. Only the paths through the network that lead to the memory module containing the target memory location are shown. Switch 1 (SW1) combines fetch-and-add(V,7) and fetch-and-add(V,5) by forwarding fetch-and-add(V,12) and storing 7 (along with the return addresses for the operations) in its

Figure 2.2



buffer. Switch 3 then combines fetch-and-add(V,12) and fetch-and-add(V,3) by forwarding fetch-and-add(V,15) to the memory module and storing 3 in its buffer. The memory module adds 15 to 6, the current contents of V, and returns the value 6 to switch 3. Switch 3 routes the returned value to P3, the processor that originated the fetch-and-add of the value held in the buffer, adds the contents of the buffer to the returned value, 6, and routes the sum to switch 1. Similarly, switch 1 routes the returned value, 9, to P1, the processor that originated the fetch-and-add of the stored value, 7, and returns the sum of the returned value and the contents of the buffer to P2, the processor that originated the fetch-and-add(V,5). Note that the effect of the concurrent execution illustrated in the figure is the same as serial execution of fetch-and-add(V,3), fetch-and-add(V,7), and fetch-and-add(V,5), in that order, i.e., V is assigned V + 3 + 7 + 5, process 3, the process that originated the fetch-and-add(V,3), receives the initial value of V, and processes 1 and 2 receive the appropriate intermediate values. The serial ordering to which a particular concurrent execution corresponds is determined by the choices made by the switches in deciding which data to store in the buffers. For example, the choice by switch 1 to store 7 instead of 5 causes fetch-and-add(V,7) to precede fetch-and-add(V,5) in the serial ordering corresponding to the concurrent execution illustrated in figure 2.2. The procedure for combining fetch-and-adds readily generalizes to combining pairs, both homogenous and inhomogenous, of fetch-and-add, read, write, and swap operations.

Note that the procedure for combining operations imposes a restriction on the kind of routing algorithm that can be used in the network: only nonadaptive routing can be used. On the return trip, a memory operation must retrace the path that it took from the processor to memory so that the returned value can be adjusted by the contents stored in the buffers.

We assume that the target machine for the list access procedures we propose supports the fetch-and-add, swap, and fetch-and-complement operations as well as store and load. Two further assumptions that we make about the target machine are that each processor has a private local memory and that no processor can initiate a new memory operation until its previous operation is completed. Even a memory operation such as the store operation that does not explicitly return a value signals the processor to indicate completion of the operation. Both the RP3 and the NYU Ultracomputer support local memory and can enforce this restriction on accessing global memory.

2.2. The Ultracomputer Queue and Flexible Queue

Researchers at NYU have used the fetch-and-add operation as the basis for highly concurrent procedures for accessing a queue and a variant of a queue called the flexible queue [GLR83, Rud82]. We will briefly describe both the ultracomputer queue and the flexible queue.

Note that if a group of processes concurrently executes fetch-and-add(V,1), a unique value from the range a through a + n - 1 will be returned to each process, where "a" is the initial value of V and "n" is the number of processes executing the fetch-and-add operation. The queue access procedures use this property of the fetch-and-add operation to assign each process a unique index into an array representing the queue. Assume that the queue is represented by a circular array of size queueSize. When a process adds an element to the queue it executes a fetch-and-add on T, the index, mod queueSize, of the element that should receive the next entry. If n processes concurrently execute fetch-and-add(T,1), T will be updated appropriately and each process will write to the element indexed by the value returned from the fetch-and-add operation mod queueSize, a unique element in the range between the old tail and the new tail of the queue. Similarly, a deleting process executes fetch-and-add(H,1), where H mod queueSize is the index of the head of the queue. Two additional variables, LB and UB, which differ by no more than the number of processes currently accessing the queue, are maintained to record the lower and upper bound, respectively, of the number of elements in the queue. The variables are used to detect queue overflow and underflow. An inserting process first increments UB, inserts an element if the value of UB indicates that there is room in the queue, and then increments LB. The procedure for deleting an element is symmetrical. We note that we have omitted discussion of several essential features of the algorithm including the method for handling integer overflow of H and T, avoiding a race condition in checking for overflow and underflow, and handling the cell contention that can occur in spite of the use of the fetch-and-add operation to assign unique indices when one or more processes "goes to sleep" while accessing the queue.

The flexible queue is a simple extension of the ultracomputer queue designed to allow dynamic allocation of space for queue elements. An array of size n, where n is the number of processes that may access the list (or the desired level of concurrency), forms the backbone of the flexible queue. Each

element in the array points to the head and tail of a linked list. There is one linked list for each queue element. Each inserting and deleting process receives an index into the array as before by executing a fetch-and-add on either H or T. Instead of writing or reading to the queue element indexed by the returned value, the process adds or deletes an element from the list pointed to by that queue element. Since overflow is no longer possible, only LB, the lower bound used to detect underflow, is maintained. The lists are serially accessible. A semaphore protects each list. Since inserting and deleting processes access the list in a round-robin fashion and there are as many lists as there are accessing processes, insertions and deletions are spread evenly among the lists, keeping the list lengths balanced, and contention for a list infrequent. The flexible queue is thus an efficient representation for data structures in which all elements are fungible.

In the following chapter we propose a set of highly concurrent procedures for accessing a linked list. We assume that the system's free space list is represented by a data structure such as the flexible queue that supports highly concurrent access so that the need to allocate and deallocate space for list elements does not create a bottleneck.

CHAPTER 3

Description of a Concurrent List Algorithm

We describe a parallel list algorithm for the ultracomputer that allows inserting, searching, and deleting processes to access a shared list with a high degree of concurrency. We begin by describing the parallel insertion procedure. The insertion procedure is based on the ultracomputer "swap" operation and is especially well adapted to the ultracomputer architecture. Accept for the logarithmic delay through the combining network, the insertion procedure runs in constant time. The second section introduces the concurrent search and deletion procedures by describing the problems that must be solved in developing an algorithm that allows concurrent insertion, search, and deletion. These problems --- coordinating contiguous deletions and identifying elements that can be returned to the free space list -- motivate the more difficult aspects of the algorithm. In the final section, we develop the algorithm. We proceed in stages, starting with a number of simplifying but otherwise undesirable assumptions which we eliminate in succeeding stages. At each stage we argue for the correctness of the algorithm.

3.1. The List Insertion Procedure

The concurrent insertion procedure allows an arbitrary number of processes, executing concurrently and asynchronously, to insert elements into a shared list by appending them to the tail of the list. The procedure uses the message-combining power of the switching network to create progressively larger sublists from insertion requests that collide in the network. The procedure produces a list that is unordered except by time of insertion: if element r precedes element s on the list, i.e., is closer than s to the head of the list, then the insertion of r was started before the insertion of s was completed.

The insertion procedure is based on the "swap" operation. Assume that a process is required to insert a list of elements, < h...t >, onto the tail of the shared list, where h points to the head and t to the tail of the list that the process is required to insert. We define the insertion procedure as follows:

```
pred := swap(TAIL,t);
store(h^.b,pred);
store(pred^.f,h);
```

We assume that the elements appended to the shared list are already in global memory and that t^.f is nil initially. The variables "h", "t", and "pred" are local to the inserting process. "TAIL" is a global variable, initialized when the shared list is created to point to the list's header element and subsequently updated by inserting processes so that it continues to point to the tail of the list. The record fields "b" and "f" are link fields.

The process begins the insertion of the list < h...t > by using a swap to update TAIL with the value of t. the value returned by the swap is a pointer to the previous tail of the shared list or to the tail of a concurrently inserted list. The process completes the insertion by linking h to the element indicated by the returned pointer.

It is easy to see that the algorithm is correct when only one process at a time executes the swap operation. The algorithm is correct under concurrent execution because the swap operation on the ultra-computer obeys the serialization principle --- concurrent execution of the swap operation produces a result, i.e., a final value for TAIL and intermediate values returned to the inserting processes, that is the same as the result that would be produced by sequential execution of some arbitrary ordering of the same swap operations[KrR85].

Assume that two processes attempt to insert lists and their swap messages collide at a combining network switch. The switch, as described in [GGK83], resolves the collision by choosing one message to forward and it stores the other message in its buffer. When the value resulting from execution of the forwarded swap message returns to the switch, it is routed by the switch to the process that sent the stored message and the value specified in the stored message is sent to the process that sent the forwarded message. The arbitrary choice of which message to forward made by the switches in resolving collisions determines the order in which concurrently inserted elements will appear on the shared list. Each distinct set of choices corresponds to sequential execution of a different permutation of the swap operations. In effect, a switch concatenates lists associated with colliding swap operations into a larger list. The switch

forwards the pointer to the tail of the second sublist, i.e., the pointer to the tail of the new combined list, and stores the tail of the first sublist. When the switch receives back a value for TAIL, it sends that value to the process that contributed the head of the first sublist and sends the stored pointer to the tail of the first sublist to the process that contributed the head of the second sublist.

Figure 3.1 shows both of the two possible computations that can result from simultaneous execution of two given swap operations and the list that corresponds to each computation. As in figure 2.2, the small circles represent switches and the boxes adjacent to the small circles, buffers associated with each switch. The figure illustrates the concurrent insertion of two lists, both consisting of two elements. The process executing on PE1, P1, inserts the list consisting of elements at memory locations 68 and 15, and the process executing on PE2, P2, inserts elements at locations 86 and 73. In figure 3.1(a), the switch forwards 15, the tail of P1's list and stores 73, the tail of P2's list. When 27, the previous value of the tail, is returned to the switch, the switch sends 27 to P2 and sends the stored value, 73, to P1. The decision to forward the tail of P1's list and store the tail of P2's list thus gives the same result as the serial execution of P2's insertion followed by P1's insertion. In figure 3.1(b), the switch forwards the tail of P2's list and stores the tail of P1's list, giving the same result as serial execution of P1's insertion followed by P2's insertion.

For a larger example, assume that six processes execute the swap operation concurrently. Figure 3.2 shows one of the 6! possible ways in which the swap messages may be executed and the resulting list. The dashes that connect some of the list elements in figure 3.2 represent pending links. At any instant, the shared list may consist of a number of disconnected sublists because some processes may have completed the swap operation but not the operations which link the newly inserted elements onto the shared list. If no new insertions are initiated and no process fails, then it will eventually be true that all of the pending links will be realized and all of the sublists that make up the shared list will be joined into a single list.

The example in figure 3.2 demonstrates the insertion of single elements. The algorithm for inserting a single element is easily obtained from the algorithm for inserting a list of elements. A process inserts a new element by executing the following instructions, where "e" is a local variable pointing to

Figure 3.1

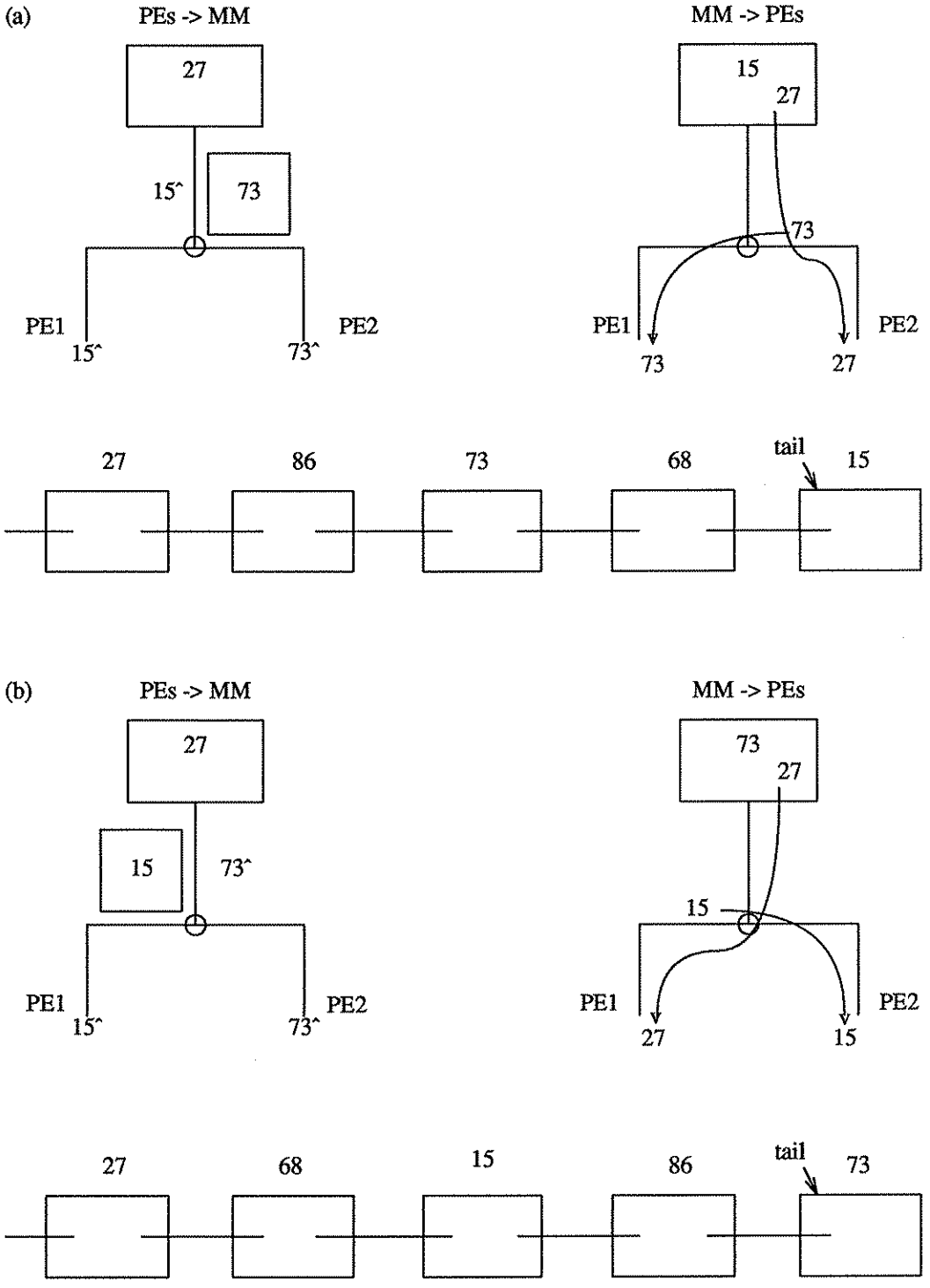
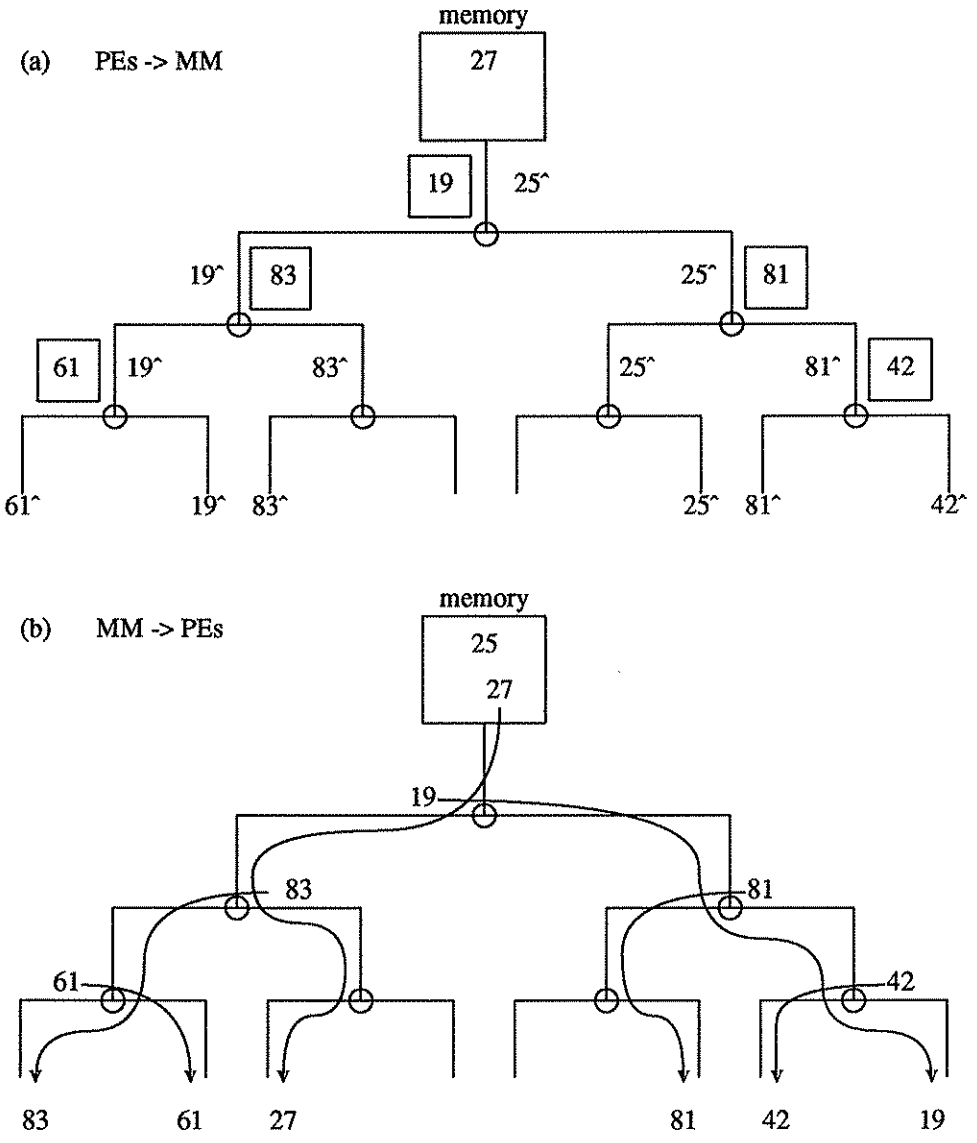

(a)
PEs -> MM
27
15^
73
PE1
15^
PE2
73^
MM -> PEs
15
27
73
PE1
73
PE2
27
27
86
73
68
tail
15
(b)
PEs -> MM
27
15
73^
PE1
15^
PE2
73^
MM -> PEs
73
27
15
PE1
27
PE2
15
27
68
15
86
tail
73

Figure 3.2



the new element:

```
pred := swap(TAIL,e);
store(e^.b,pred);
store(pred^.f,e).
```

It is hard to classify the insertion procedure unequivocally as of either constant or logarithmic complexity. Although the time required for each global memory reference increases logarithmically as the number of processors and thus the size of the network increases, the number of instructions required to insert an element into a shared list remains constant. At least one author has suggested that such algorithms can be classified as constant time algorithms on the grounds that delay through the combining network is comparable to delay through memory-address decoders and can, for practical purposes, be similarly ignored [Sto84]. The insertion procedure is logarithmic in that sublists are combined through fan-in. But the procedure maps so efficiently onto the ultracomputer that the combining is done in a constant number of global memory references as part of the process of routing messages to memory.

An application of the parallel insertion procedure, namely, a simple and efficient implementation of P and V operations on a binary semaphore that maintains FIFO queueing discipline among waiting processes, is described in Appendix A.

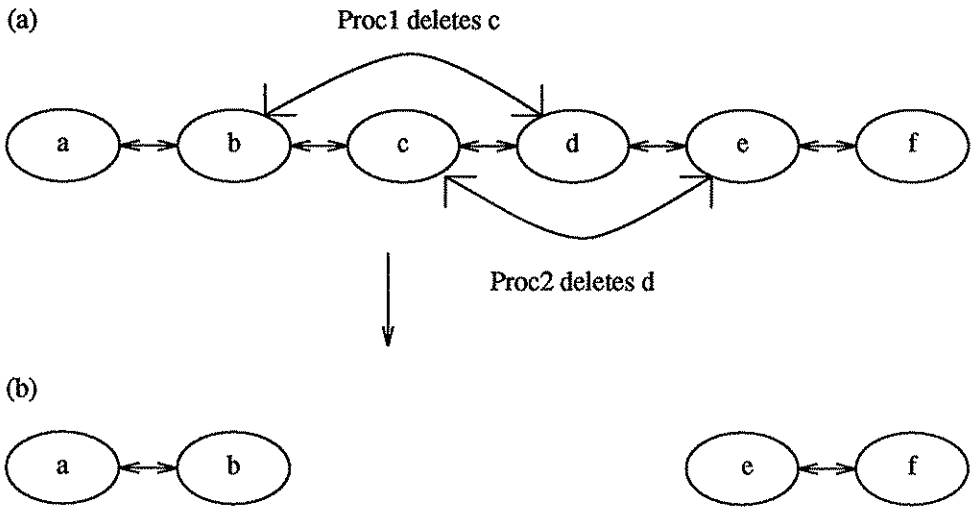
3.2. Deletions and the Concurrent List Algorithm

An algorithm for concurrent insertion, search, and deletion operations on a shared list must include a mechanism for coordinating contiguous deletions and must be able to determine when elements can be safely added to the free space list. If the algorithm is to achieve a high degree of concurrency, it must solve these problems in a way that requires little waiting. We describe each problem and briefly indicate the approach we take in solving it.

3.2.1. Coordinating contiguous deletions

As illustrated in figure 3.3, if two or more processes, acting simultaneously, each attempt to delete one of a contiguous string of elements by linking the two immediately adjacent elements, a portion of the list may be lost. As a first-cut at a solution we could require a deleting process to mark the target element's neighbors as anchors for the deletion before linking them. We say that an element is an

Figure 3.3



"anchor" if it is marked ineligible for for deletion while a process deletes a neighboring element. If we can ensure that the target element's neighbors remain on the list until after the target element is deleted, then the deletion can be made without the risk that the list will become disconnected. This solution is flawed. Assume that the target element of a process cannot be marked as an anchor by any other process. If processes simultaneously attempt to delete adjacent elements, then no process will be able to mark both its anchors. If the processes persist in trying to mark their anchors, partial deadlock will result. Allowing processes to mark the target elements of other processes as anchors does not help. As was illustrated in figure 3.1, if anchors can be deleted the list can become disconnected, but if anchors cannot be deleted, then deadlock can still occur. Every process in a group of processes with contiguous target elements can mark its anchors but no process can complete a deletion because its target element is itself an anchor. If the processes avoid deadlock by releasing their anchors and trying again, then livelock can result.

We resolve this difficulty by using the order of elements in the list to define the order in which a process must acquire its anchors. We define the beginning anchor as the anchor closest to the head of the list and require that a process acquire its beginning anchor before claiming its ending anchor. We also bar a process from marking an element that is the target element for another process as an anchor. These rules for marking anchors coordinate the work of processes that are attempting to delete contiguous ele-

ments. Only the process that is attempting to delete the element closest to the head in a string of contiguous target elements can succeed in marking its beginning anchor. The other deleting processes fail to mark their beginning anchors, and abandon their deletion attempts, leaving their target elements marked invalid to indicate that they should be deleted from the list. After it marks its beginning anchor, the process that is deleting the first element in the string scans toward the tail of the list searching for the first element that is either valid or marked as a beginning anchor for another deletion and it marks that element as the endpoint for its deletion. When it links the beginning and ending anchors it deletes the entire string of contiguous invalid elements.

3.2.2. Identifying elements that can be disposed

The second problem we must solve is garbage collection. Since we want to be able to use the concurrent list algorithm in production code, we must define a procedure by which elements can be added to the free space list when they become garbage. Identifying garbage is a difficult problem because processes traversing the list may hold pointers to deleted elements. A garbage collection procedure must never err by misclassifying an element to which a process holds a pointer as garbage, but it must also ensure that all garbage is eventually identified and collected. The procedure should identify garbage promptly. A long lag between the time an element becomes garbage and the time it is collected implies wasted space.

Our algorithm for garbage collection uses reference counts to identify garbage and is a variant of the reference count garbage collection algorithm in [KuL80]. We have corrected an error in that algorithm that could result in a dangling reference and have eliminated an atomic read and write which would require locking list elements.

We have not used any of the algorithms for parallel garbage collection based on marking [Ben84, DLM78, Lam76, Ste75, Wad76]. Garbage collection algorithms based on marking collect garbage in batches and require more space than an algorithm that can detect when a unit of memory becomes garbage and return it to the free space list on an individual basis. Furthermore, the classic parallel garbage collection algorithms based on marking are inapplicable because they permit only two

processes, a single user process and a single garbage collector. Both Steele and Lamport have described algorithms to solve the more general problem in which multiple user processes and garbage collectors are permitted [Lam76, Ste75], but these algorithms would be a poor choice here. A process's pointers into the list are stored in local memory, inaccessible to the garbage collectors, which must be able to read all pointers to used elements. Furthermore, both Steele's and Lamport's algorithms require frequent synchronization among the user processes or between the user processes and the garbage collectors. Finally, use of a parallel garbage collection algorithm requires additional processes dedicated to the collection of garbage and a continual scanning through memory to find garbage. We do not wish to require the use of a parallel garbage collection algorithm in a program that may not otherwise need one.

3.2.3. Keeping the amount of waiting low

Handling contiguous deletions and identifying garbage can be greatly simplified if a process can freeze the state of a list element and require other processes to wait before accessing it. We require an algorithm that handles contiguous deletions and garbage collection while keeping the portion of execution time spent in waiting low. An algorithm may require waiting explicitly, by synchronization statements in the code, or implicitly, by requiring an arbitrary number of processes to use a system resource, such as memory bandwidth, that only a limited number of processes can use at any one time.

Ideally, no process will need to wait for another process. We have not found a "wait-free" algorithm for accessing a shared list, nor can we show that we have minimized waiting. But, by using the ultracomputer's capability for combining memory operations and by allowing searchers, deleters, and inserters to all access the list concurrently, we keep the expected amount of waiting to a low level.

There is one case in which the concurrent list algorithm described in this chapter explicitly requires waiting: before disposing of a deleted element, a process must wait until all other processes which are visiting or will visit the element have completed their work at the element. With the exception of this case, processes executing the algorithm described in this chapter are never explicitly required to wait. Note, however, that when searches begin from the tail instead of the head of the list, a searching process will need to wait if it reads a nil pointer to the next element before it encounters the header node. When

search begins at the tail, a searcher may visit an incompletely inserted element and need to wait until the insertion is completed. When search begins at the head, no searcher can see an incompletely inserted element.

Implicit waiting is reduced by the use of the ultracomputer. Without the combining power of the ultracomputer's switching network, a set of processes attempting to update the same variable would have to contend for access. The execution of each set of processes would be effectively serialized and the hot spot created by their contention could degrade the performance of the entire system [PfN85].

We take full advantage of the power of the ultracomputer by using only combinable memory operations. An earlier version of our algorithm used a conditional swap, a memory operation which is convenient but not, in general, combinable. The conditional swap is convenient because it allows a comparison and conditional write to be executed atomically. The removal of the conditional swap makes the algorithm more complicated and increases the number of memory references required for a list access, but it eliminates the potential for contention by processes executing a conditional swap on the same target variable.

The ultracomputer can combine requests directed toward the same variable, but it cannot combine requests with different target variables in the same memory module. We assume that references to addresses that are uniformly distributed among the memory modules will create only minimal contention in a well designed system, but we can not assume uniformly distributed addresses in the case of fields in the same record. Messages from processes attempting to access different fields in the same list element will have to be serialized at the memory module. One solution is to tolerate the problem. Since there are a limited number of fields in each list element, there are a limited number of messages that can contend for access to a given element, assuming all the requests directed toward the same field are combined. Note that some of the fields can be combined, reducing the number of fields, and thus the number of sets of contending processes. A second solution is to spread the fields of an element across the memory modules in much the same way as arrays are spread across memory in array processing systems. If the address of the first field in every element is randomly determined and if each of the remaining fields is stored in a different memory module at a fixed offset from the address of the first field, then a process

should be able to access any field in an element without any contention from other processes accessing the same element and with minimal contention from processes accessing the same memory module.

3.3. The Search, Deletion, and Excision Procedures

In the first section of this chapter we described a concurrent list insertion procedure. In this section we present the companion search, deletion, and excision procedures. We have been using the term "deletion" to mean the logical and physical removal of an element from the list. Henceforth we will reserve the term "deletion" to refer to execution of the deletion procedure and will use "excision" to mean the physical removal of an element from the list. The deletion procedure performs a logical removal and may or may not excise an element from the list.

We develop the concurrent list algorithm in five stages. In the first stage we concentrate on handling contiguous excisions. To simplify the explanation we assume that a conditional swap operation is available and that garbage collection is done in batches by a maintenance process as described in [KuL80]: Excised elements are added to a garbage collection list. The maintenance process periodically starts a new garbage collection list, waiting until all the processes active in the list at the time the new list is created complete their current list access before appending the old garbage collection list to the free space list. Deletion by user processes is "lazy", i.e., a user process deletes an element by simply marking it invalid. Maintenance processes, the "collectors", search the list and excise invalid elements.

At the second stage we eliminate the assumption that a conditional swap is available. The third stage introduces individual garbage collection and reference counts. The garbage collection list and the attendant maintenance process are eliminated. The collectors not only excise invalid elements but also add the elements they excise to the free space list when they become garbage. To simplify the introduction of reference counts, we initially allow an atomic read and write. We assume that a process can read a field in an element and increment the field indicated by the result of the read in a single atomic action. At the fourth stage we eliminate this atomic action. Finally, in the fifth stage we eliminate the collectors and require searchers and deleters to excise invalid elements.

Before describing the first stage of the concurrent list algorithm we describe the structure of the shared list and give a top-level description of the access procedures and the routines called by these top-level procedures.

The data structure

The valid elements in the shared list form a doubly-linked list with a header node that is a dummy element. The pointers to the head and tail of the list are global. Since the header node is never excised or moved, processes can keep a local copy of the head pointer, but the tail pointer must be accessed in global memory. The list is initialized by a single process. The process requests allocation of space in global memory for the header node, marks the node as valid, sets the header node's forward pointer equal to nil, and initializes the head and tail pointers to point to the header node.

We assume that every element has eight fields: a data field; a forward pointer; a back pointer; a status field; a census field; two intransit fields; and a switch field. The following is a Pascal style definition of the structure of a list element where the type "eptr" is defined as a pointer to a list element and the type "pIdRange" is an integer in the range 0 to the maximum number of processes that may access the list:

```
listElement = record
  data: dType;
  b: eptr;
  f: eptr;
  intransit: array[0..1] of pIdRange;
  switch: 0..1;
  census: pIdRange;
  status: statusVector;
  end;
```

The type of the data field is application dependent. If the data field cannot be accessed in an atomic action then some mechanism must be introduced to prevent searching processes from seeing inconsistent data. This mechanism could be a lock set by writing processes to block readers until completion of the write or a protocol such as that described in [Lam77, Pet83] which allows a reader to detect that it has read inconsistent data and to retry an unsuccessful read. We assume here that the data field can be read and written in a single atomic action. Element c's forward and back pointers are denoted c^.f and c^.b,

respectively. If c^.f = e we say that e is c's successor. If c^.b = e, then we say that e is c's predecessor. The predecessor of c is closer than c to the head of the list.

The status of an element has six dimensions:

(1) valid/ ~valid; An invalid element, denoted ''~valid'' in the status vector representing the element, is an element that has been logically removed from the list so that it can no longer satisfy a search. The element may or may not have been excised, i.e., physically removed from the list.

(2) bA/ ~bA; An element with status ''bA'' is a beginning anchor for an excision, i.e., it is the element immediately preceding the target element for an excision.

(3) EOL/ ~EOL; An element marked EOL has no successor or has an incompletely inserted successor. Recall that there may be several elements marked EOL at any given moment because of incomplete insertions.

(4) claimed/ ~claimed; The elements between a target element for an excision and the ending anchor for an excision are ''claimed'' elements. A target element is the element that a process seeks to remove from the list it when begins the excision. After marking its bA, an excising process scans from its target element for an ending anchor for the excision, claiming all invalid elements that are not beginning anchors or marked EOL. If the ending anchor is the target element's successor on the list, then the excision removes only the target element. If the ending anchor is not the target element's successor, then the excision removes the target element and one or more claimed elements. Since a process does not release its claim on an element after excising it, an excised claimed element remains claimed.

(5) gray/ ~gray; A gray element is or has been the target element for an excision. A target element is colored gray as the first step in excising the element. Since the excising process does not uncolor its target element after successfully removing it from the list, an excised target element remains gray.

(6) eA/ ~eA. An element with status ''eA'' is an ending anchor for an excision, i.e., it is the successor of the last element in a sublist of elements a process is excising.

Three of the dimensions listed, 2 through 4 above, can be collapsed into a single dimension which we will denote "red/~red". An element is red if it is either a beginning anchor, claimed, or marked EOL. The three dimensions can be collapsed into one because a positive choice along one dimension excludes a positive choice along either of the remaining two dimensions, e.g., a beginning anchor cannot simultaneously be marked either claimed or EOL. Note that we can represent the status of an element as a four bit boolean vector and that the operations on status vectors are actually operations on bit strings. For clarity we will sometimes prefer to specify whether a red element is a beginning anchor or marked EOL or claimed. For example, we may denote the status of a newly inserted element with the vector [valid,EOL,~gray,~eA], even though the vectors [valid,EOL,~gray,~eA] and [valid,bA,~gray,~eA] have the same physical representation and are indistinguishable to processes executing the algorithm.

We delay discussion of the remaining fields in an element, the census, intransit and switch fields, until we introduce individual garbage collection in the third stage algorithm. During the first two stages, when batched garbage collection is used, we require another link field in each element, used solely to link elements together on the garbage collection list. Since some process may need to reference the forward or back pointer of an element on the garbage collection list, these link fields cannot be overwritten and used to connect the garbage collection list.

Top-level description of the list access procedures

The concurrent list algorithm supports three operations on the list: insertion, search, and deletion. The insertion procedure permits insertion at the tail of the list only. The deletion procedure allows a process to invalidate an element at a known location anywhere within the list without first searching the list. An invalid element is logically, but not necessarily physically, removed from the list. Searchers may visit invalid elements but their search is never satisfied by an invalid element. A process using the deletion procedure must know that the target element is on the list and that it is valid and will remain valid until the process itself invalidates it. If another process can invalidate the target element, then the element can be freed and a subsequent attempt to use the deletion procedure can result in a dangling reference. A process must delete an element that can be invalidated by other processes by first searching the list for the element. The search procedure could have any of several possible specifications. In the procedure

described here, processes search the list starting from the head and invalidate the first occurrence of a valid element with a data value equal to a given target value. The search procedure can be readily modified to allow searches for a purpose other than deletion, searches from the tail of the list, or searches for all matching elements or for a specific element.

In the first few stages of the algorithm, as we develop it, the search and deletion procedures accomplish only a ''logical'' deletion, i.e., they only invalidate elements, leaving the job of excising them to maintenance processes. The maintenance processes that do the physical deletion of invalid elements from the list repeatedly execute the internal procedure ''Collect''. The collect procedure is very similar to the search procedure, but is a search for all invalid elements, not for the first matching valid element. The collect procedure calls the internal procedure ''AttemptExcise'' which in turn calls ''Excise''. When a collector encounters an invalid element it attempts to excise it. We refer to the invalid element that the collector is attempting to excise as the process's ''target''element The AttemptExcise procedure attempts to satisfy the conditions for an excise. If the attempt succeeds, the excise procedure is invoked. The excise procedure performs the physical removal of elements from the list and, if garbage is being collected on an individual basis, adds the excised elements to the free space list when they become garbage. In the fifth stage algorithm the collect procedure is eliminated as a separate procedure along with the maintenance processes that execute it, but its functions are incorporated into the deletion and search procedures executed by the user processes.

Note the distinction between invalidating, excising, freeing, and deleting an element. Invalidating an element logically removes the element from the list. Excising an element removes it from the list physically. An element is freed by placing it on the system free space list. Only invalid elements are excised and only excised elements are freed. A process deletes an element by executing the deletion procedure. In the first stages of the algorithm, deleting an element only invalidates it. In the fourth stage algorithm, deleting an element invalidates and may also excise and free the element.

3.3.1. First stage algorithm

Initially we assume that garbage collection is done on a batched basis by a maintenance process and that a conditional swap operation is available. Use of the batched garbage collection requires that processes "check in" before executing a list access and "check out" after completing a list access, but we will not illustrate this or any other detail of the algorithm relating only to the batched garbage collection.

Given these assumptions, procedures Insert, Delete, Search, and Collect are straightforward. The insertion procedure for the concurrent list algorithm is the same as the algorithm described in the first section of this chapter with two additional instructions. Before inserting an element a process marks the element valid and EOL, and after inserting the new element the process resets the EOL indicator in the element that was previously the last element in the list:

```
store(e^.status,[valid,EOL,~gray,~eA]);    -- mark element EOL
pred := swap(TAIL,e);
store(e^.b,pred);
store(pred^.f,e);
store(pred^.status,[-,~EOL,-,-]);          -- predecessor in no longer EOL
```

A "-" in a status vector denotes a position in the vector that does not participate in the operation. The instruction "store(pred^.status,[-,~EOL,-,-])" removes the red coloring from the element but otherwise does not change its status. Since a process may have received the address of the last element on the list as the element to which to link a new element, the last element on the list may be invalidated but it must not be excised from the list while it is the last element. As will be seen, the EOL indicator in the status field is a convenient way to ensure that the last element is not excised. Note that there may be several elements marked EOL since at any given moment several processes may have executed all but the last line of the insertion procedure.

The deletion procedure is trivial. A process deletes an element, c, known to be on the list by simply invalidating the element as follows:

```
store(c^.status,[~valid,-,-,-]);
```

The deletion thus accomplished is only a logical deletion. The element remains in the list until a collector

finds the element and succeeds in excising it.

The search procedure is also straightforward. Starting at the head node, a searcher visits every element in the list, reading the element's data value and attempting to invalidate any element it finds with a data value equal to its target value, until it succeeds in invalidating an element or until it encounters the end of the list:

```
c := fetch(H^.f);
while c <> nil do
  if fetch(c^.data) = target
    then if match(swap(c^.status,[~valid,-,-,-]),[valid,-,-,-])
          then exit;
  c := fetch(c^.f);
end while.
```

"Match" is a function that returns true if the two status vectors passed to it match, i.e., are identical in all positions not masked with an "-". When the searcher finds a element with a data field equal to its target it invalidates the element with a swap operation that returns the previous status of the element. If the previous status was valid then the process knows that it succeeded in invalidating the element and can exit. If the element was already invalid then the process must continue its search.

The search and collect procedures are very similar since the top-level collect procedure is also a search, in this case, for invalid elements. A collector starts at the head node's successor and visits every element, attempting to excise elements with status equal to "[~valid,~red,~gray,~eA]" until it encounters the end of the list:

```
c := fetch(H^.f);
while c NOT= nil do
  if match(fetch(c^.status),[~valid,~red,~gray,~eA])
    then attemptExcise(c);
  c := fetch(c^.f);
end while.
```

The collector does not attempt to excise an invalid element unless its status matches "[~valid,~red,~gray,~eA]".

When a collector finds an element, c, that is eligible for excision, it tries to excise the element by executing the AttemptExcise procedure. On return from AttemptExcise, c will remain unchanged if the attempt to excise c fails and will point to the ending anchor for the excision, if the attempt to excise succeeds.

We define AttemptExcise as follows:

```
if match(f&sOnMatch(c^.status,[~valid,~red,~gray,~eA], [-,-,gray,-]),
  [~valid,~red,~gray,~eA])                          try to color c gray
  then bAnchor := fetch(c^.b);
     if match(f&sOnMatch(bAnchor^.status,[-,~red,~gray,-], [-,bA,-,-]),
       [-,~red,~gray,-])                            try to mark a beginning anchor
       then excise(c,bAnchor)
       else store(c^.status,[-,-,~gray,-]).
```

The variable ''bAnchor'' is a local variable that points to the candidate beginning anchor for the excision. If a process executing AttemptExcise succeeds in marking the beginning anchor, it calls excise with bAnchor as a parameter.

The f&sOnMatch operation is a conditional swap. Execution of the operation f&sOnMatch(V,pattern,e) is equivalent to the atomic execution of the following procedure:

```
temp := V;
if match(V,pattern)
  then V := e;
return temp.
```

A process can excise the target element only if it can color the element gray and mark its predecessor as the beginning anchor for the excision. The requirement that an excising process first color the target element gray prevents more than one process from attempting to start an excision at the same node. Coloring an element gray also prevents its use as a beginning anchor by another process.

After coloring the target element gray, the excising process fetches the address of the target's predecessor. If the predecessor is neither red nor gray, the excising process marks that node as the beginning anchor for the excision. If the element is red, the excising process cannot rely on it to remain on the list. If the candidate beginning anchor is gray, then another process is attempting to excise it. In either case, the element should not be used as a beginning anchor.

A collector that marks the beginning anchor will succeed in excising its target element. The collector performs the excision as follows:

```
dptr := c;                                  -- remember address of target element
repeat                                      -- look for the ending anchor
   c := fetch(c^.f);
   eStatus := swap(c^.status,[-,-,-,eA]);
   if match(eStatus,[~valid,~red,-,-])
      then eStatus := swap(c^.status,[-,claimed,-,-];
until not match(eStatus,[~valid,~red,-,-]);  -- c is now the ending anchor
store(c^.b,bAnchor);                        -- link the anchors together
store(bAnchor^.f,c);
store(c^.status,[-,-,-,~eA]);               -- unmark the anchors
store(bAnchor^.status,[-,~bA,-,-]);
appendlist(dptr).
```

''Dptr'' and ''p'' are local variables and ''appendlist'' is a procedure which appends the list passed to it to the garbage collection list.

The excising process first finds and marks the ending anchor for its excision and then links the two anchors together and appends the excised sublist to the garbage collection list. In its search for an ending anchor, the collector claims and skips over any element that is both invalid and not colored red. The process will excise any elements it claims, including its target element, when it links together the anchors for the excision. A collector finds the ending anchor for its excision by starting at the target element's successor and scanning toward the tail until it finds an element that is either valid or red. While looking for its ending anchor, the collector initially assumes that each element it encounters is the element it will use as its ending anchor. When the collector arrives at an element it marks the element as its ending anchor, using a swap operation. The collector then examines the value returned from execution of the swap operation to see if the element should be excised instead of being used as an ending anchor. If the returned value shows that the element is either valid or red, the collector was right in marking it as an end anchor. If the returned value shows an element that is neither valid nor red, then it can be excised. The collector marks the element as claimed, again using a swap operation. If the value returned from the execution of the second swap operation shows that the element was not colored red between accesses, then the collector succeeded in claiming the element with its second swap operation. The collector will continue its search for an ending anchor with the element's successor. (The collector leaves the ending

anchor bit set on the claimed element because the marking does no harm and unmarking requires work.) If the value returned from the execution of the second swap operation shows that the element was colored red between the execution of the first and second swap operations, then the collector stops its scan and uses the element as its ending anchor. The procedure for finding and marking the ending anchor for an excision is designed to work with the procedure for marking the beginning anchor for an excision in such a way that it is impossible for a given element to be both claimed by one process and marked as a beginning anchor by another. Note that the last element on the list is red so it can not be claimed. The last element will be marked as an ending anchor by any excising process that encounters it while looking for an ending anchor.

Code for the complete first stage algorithm can be found in Appendix B.

Correctness Argument: First Stage Algorithm

We say that the system of list access procedures described above is correct if it satisfies the following properties:

Prop1 An element is excised from the list only once and only if it is invalid.

Prop2 The list can only have elements added by the insertion procedure and can only have elements excised by the excise procedure.

Prop3 The list is always consistent. At any instant, the elements on the list appear on the search path from the head of the list in increasing order of insertion position counts.

Prop4 When a search terminates, either it has found the first currently valid instance of the item sought or there is no valid instance of the item on the list.

Prop5 Insertions and deletions always terminate. In the absence of further insertions, all excisions and searches terminate.

Prop6 The system does not deadlock.

An element is "excised" if there is no search path connecting it to the head after all pending insertions are completed. An element is "on the list" if it is currently accessible from the head, i.e., if there is a search path connecting it to the head. A search path is a path with edges defined by forward pointers. Since a search follows only forward pointers, an element is inaccessible in a search unless it can be reached from the head by following only forward pointers. Note that it is possible for an element to be neither excised nor on the list. Even if the element has been completely linked to its predecessor, the element will not be on the list unless its predecessor is on the list. Property 4 guarantees only that an

unsuccessful searcher has visited all elements accessible from the head of the list at the time the search terminates, not that it has visited all valid elements that have been inserted. We assume that the list access procedures are correct when executed sequentially, that the list is properly initialized, and that the processes are reliable --- no process fails while performing an access.

We do not attempt a formal proof that the system is correct. For all the properties except Prop1 we only sketch the outline of a proof. Prop1 is hard to establish because it involves the interaction of excising processes, the processes that have the most impact on the list structure. Since Prop1 is the least obvious property, we offer a more complete, though still informal, correctness argument for Prop1.

Prop1: correctness argument

We establish that only invalid elements are excised by showing that the beginning and ending anchors for an excision are both on the list at the time the excising process links its anchors and that all elements on the list between the beginning and ending anchors are invalid. The beginning and ending anchors are the elements the excising process marks bA and eA, respectively, before performing the excision. For the purpose of this argument we define an excision as the atomic action of redirecting the beginning anchor's forward pointer and we assume that no two excisions occur at the same instant. If the algorithm is correct when excisions can occur in any order and if no two excisions can interfere with each other by simultaneously using the same element as a beginning anchor, then the algorithm is also correct when excisions occur simultaneously.

Consider the action of a process, P1, in changing the forward pointer in its beginning anchor, bA, to point to its ending anchor, eA, at time t. We establish Prop1 by showing that the following properties hold at time t:

Prop1.1
: P1's beginning anchor, bA, is on the list.

Prop1.2
: There is a search path from bA to eA.

Prop1.3
: All the elements on the search path from bA to eA are invalid. The first element on the search path is Proc1's target element and all the elements on the search path from the target element to eA are claimed by P1.

Properties Prop1.1 through Prop1.3 establish that excisers remove only invalid elements from the list. If the beginning and ending anchors for every excision are both on the list at the time of the excision and if all the elements on the search path from the beginning to the ending anchor are invalid, then the action of redirecting the forward pointer in the beginning anchor to point to the ending anchor excises only invalid elements. Since the only other pointer changed by an excising process is the backward pointer in the ending anchor and changing a backward pointer cannot excise an element, excisers can remove only invalid elements from the list. Since by Prop2, only excisers can excise an element, only invalid elements can be excised. (Prop2 does not depend on Prop1 for its claim that only excisers can excise an element.) Note that Prop1.1 through Prop1.3 also establish that all excised elements are colored red or gray. The process responsible for excising an element colors the element before excising it. The process never uncolors the element and no process can reset a red or gray bit set by another process. Therefore, an excised element remains colored red or gray after it is excised.

We establish Prop1.1 through Prop1.3 inductively. If Prop1.1 through Prop1.3 are true for all previous excisions then they are true for the current excision.

Prop1.1: correctness argument

We claim that bA is on the list at time t. Assume, on the contrary, that bA is not on the list at time t. Since bA was found by a search from the head, if bA is not on the list at time t, bA is excised. By the inductive hypothesis, all elements excised up to time t are colored red or gray by the process responsible for excising them. Since bA was not colored red or gray at the time P1 marked it as its beginning anchor, bA can not have been excised before it became P1's beginning anchor. From our assumption that the excise procedure is correct when executed sequentially, we know that P1 does not excise bA while it is using bA as its beginning anchor. Therefore, bA must have been colored red or gray and then excised after it become P1's beginning anchor. But no process can color an element once a process marks it as its beginning anchor. We conclude that at time t bA is on the list. eA is on the search path from bA at time t.

Prop1.2: correctness argument

The element eA was encountered by following forward pointers from P1's target element, c. If c is on the search path from bA at time t and none of the forward pointers followed by P1 in finding eA has changed since read by P1 then eA is on the search path from bA at time t. None of the forward pointers followed by P1 can have changed since they were read by P1. If an element is not the last element on the list, its forward pointer can change only if it is marked as a beginning anchor. Element c could not have been marked by P1 as its target element if it was already marked as a beginning anchor and cannot subsequently be marked as a beginning anchor while it is colored gray. If any of the elements between c and eA encountered by P1 in its search for an ending anchor was already marked as a beginning anchor, the element would be used by P1 as its ending anchor, contradicting the assumption that eA is the ending anchor. Since P1 claims each element encountered in its scan for the ending anchor before reading the element's forward pointer and no process can mark a claimed element as its beginning anchor, no element encountered by P1 while looking for its eA could be marked as a bA after P1 reads its forward pointer. Therefore, none of the pointers read by P1 in finding eA have changed and there is still a search path from c to eA at time t.

We show that P1's target element, c, is on the list at time t by contradiction. Assume that c is excised at time t. Then c must have been excised after it was colored gray by P1. (The status of c was [˜valid,˜red,˜gray,˜eA] at the time P1 colored it and the status of all excised elements matches [-,red,-,-] or [-,-,gray,-].) Since no element can simultaneously be the target element for two different processes and c is P1's target element, c cannot have been the target element of the exciser which excised it. By P3, the first element in any excised sublist is the target element of the excising process so c's predecessor must have been excised with it. But c's predecessor, bA, is on the list. Therefore c's predecessor must have become bA at some time after c was excised and before P1 read c's backpointer to bA. The only time that the predecessor of an element on the list can change is when the element is marked as an ending anchor. But, by the inductive hypothesis, the ending anchors of all excisions up to time t have been on the list at the time of the excision.

Prop1.3: correctness argument

We have shown that there is a search path from bA to eA at time t that has not changed since P1 traversed it in finding eA. P1 colors c gray before time t and claims all other elements, if any, on the search path from bA to eA. Since P1 uses the first valid or red element it encounters as its ending anchor, all elements on the search path from bA to eA at time t are invalid. We must show that c is bA's successor at time t, i.e., that there are no elements between bA and c. By examination of the code for attemptExcise, bA is c's predecessor at some time after c is marked gray. Since insertions occur only at the end of the list and both bA and c remain on the list up to time t, bA must continue to be c's predecessor (and c, bA's successor) up to time t. Therefore, the first element on the search path from bA at time t is P1's target element.

We have shown that only invalid elements are excised. To complete the argument for Prop1, we show that an element can be excised only once.

An element can be excised only once because it can be excised by only one process and after it is excised it cannot be put back on the list. An element can be excised by only one process because an element cannot be excised unless it is bracketed by an excising process's beginning and ending anchors and only one process can bracket an element. If element e is on the search path between P1's beginning anchor, bA, and ending anchor, eA, then it is excised only by P1. By Prop1.1 and Prop1.2, no beginning or ending anchor of another process can be on the search path from bA to eA and neither bA nor eA can be on the search path between any pair of anchors belonging to another process. Therefore, no element between bA and eA is on the search path between the anchors of any other process. Since a process can excise only the elements between its anchors, no other process can excise any element on the search path from bA to eA.

Prop2: correctness argument

Since searchers do not change the list structure, Property 2 can be proven by showing that inserters cannot excise elements and excisers cannot add elements. An inserter cannot excise an element because each inserter receives a different element as a result of executing the swap operation on the tail variable to use as the predecessor for the element it is inserting. Since every inserter gets a unique predecessor, the value of the predecessor's forward link will be nil at the time the inserter changes it, i.e., no other

inserter will have already connected the element to a successor element. Since the link is nil, overwriting the link cannot excise an element. We can assume that no two inserters will receive the same element as a predecessor because no sequential ordering of the swap operations in the insert procedure can result in two inserters receiving the same predecessor and the swap operation on the ultracomputer obeys the serialization principle.

The only way an exciser could add an element to the list is by redirecting its beginning anchor to point to an element that is not on the list, We show in our discussion of P1 that the ending anchor for an excision is always on the list at the time the anchors are linked.

Prop3: correctness argument

The insertion position count of the header node is 1 and the insertion position count of element c, where c is not the header node, is 1 plus the insertion position count of c's predecessor when c is put on the list.

Neither searchers not inserters can change the order of elements on the list. Searchers cannot change the structure of the list and inserters can add elements only at the tail of the list. Excisers also maintain P3 since sequentially executing excisers cannot change the order of elements on the list and concurrently active excisers each operate within a different section of the list. As a corollary of P3, the search path contains no cycles. A cycle would have to contain at least one pair of ''out of order'' elements.

Prop4: correctness argument

Assume that a search terminates successfully at element c, but that at the time the search terminates, time t, there is a valid matching element, b, on the search path connecting c to the header node. The search must not have visited b since the search would then have terminated at b. If b is on the search path to c at time t but was not visited by the search before it arrived at c, then b was either inserted into the search path after the search passed b's position or the search skipped b. Since insertions can occur only at the tail of the list, b could not have been inserted into the search path ''behind'' a searcher. A search can skip over b if b is excised, i.e, if the forward pointer of a predecessor of b is redirected to point

to a successor of b, or if b's order in the list is changed, i.e, if a predecessor of b is moved so that the element become a successor of b. Since b is valid and only invalid elements are excised from the list, the search must have failed to visit b because b's order in the list changed. But by Prop3, the order of elements in the list cannot change. We conclude that a successful search has found the first currently valid matching element on the list.

If a search terminates unsuccessfully because it encounters a nil forward pointer before it finds a valid matching element, then we know, by the same reasoning, that there is no valid matching element on the list. An unsuccessful search will terminate at the end of the list unless at some time during the search a process destroys the search path from the searcher's current element to the end of the list by changing a forward pointer. Searchers cannot change pointers and inserters cannot change the pointer of any element on the list except for the last element on the list. An excising process changes only one forward pointer --- it redirects the forward pointer in its beginning anchor to point to its ending anchor. Since all the elements between the beginning and ending anchors are invalid, redirecting searchers from the beginning to the ending anchor cannot result in bypassing any valid elements. If a search path that includes all valid unvisited elements existed for each searcher before the excision, then it will exist after the excision. In particular, a searcher visiting an excised element will be able to find its way back to the list. We conclude that the system maintains P4.

Prop5: correctness argument

Prop5 is clearly true for insertions and deletions. Prop5 is true for searchers and excisers because the number of instructions executed at each element is bounded by a constant, the list contains a finite number of elements, and there is no cycle in the search path. We require the assumption that no further insertions are allowed in order to guarantee termination for searchers and excisers because both searchers and excisers scan toward the tail of the list, the insertion site for elements. A search will never terminate if no matching element is ever inserted and the rate of insertion matches or exceeds the rate at which the searcher traverses the list. Similarly, an excision near the end of the list will never terminate if all inserted elements are prompty invalidated and the rate of insertion matches or exceeds the rate at which the exciser scans the list for its ending anchor.

Prop6: correctness argument

We assume that the waiting implied by contention in the interconnection network and at the memory modules does not cause deadlock. The only waiting required by the algorithm itself is the waiting implied by use of the conditional swap. We have not specified how the conditional swap is implemented, but whether it is implemented in the memory module itself or by locking the memory location for the duration of a round trip to a processor, the conditional swap cannot cause a deadlock because no process can block the process executing the swap from completing execution of the swap and releasing the lock. Therefore, since the algorithm does not otherwise require any waiting, the system does not deadlock.

3.3.2. Second stage algorithm

In the first stage algorithm, we assume that a conditional swap is available. The conditional swap simplifies the presentation of the algorithm but is not, in general, combinable with other operations directed toward the same target variable, e.g., f&sOnMatch(V, [~valid,~red,~gray,~eA],[-,-,gray,-]) is not combinable with swap(V,[~valid,-,-,-,]). Since the combining network is unable to combine a conditional swap operation, use of the conditional swap implies waiting. In the second stage algorithm, use of the conditional swap is eliminated.

The two conditional swaps used in the first stage algorithm both occur in the AttemptExcise procedure. Only the AttemptExcise procedure changes in this version of the algorithm. The revised procedure for attempting to delete an element, c, is as follows:

```
eStatus := swap(c^.status,[-,-,gray,-]);
if match(eStatus,[~valid,~red,~gray,~eA])
   then bAnchor := fetch(c^.b);
        if match(swap(bAnchor^.status,[-,-,gray,-]),[-,-,~gray,-])
           then if match(swap(bAnchor^.status,[-,bA,~gray,-]),[-,~red,-,-])
                   then excise(c,bAnchor);
                        exit;
if match(eStatus,[-,-,~gray,-])
   then store(c^.status,[-,-,~gray,-]).
```

The first conditional swap in the original version of AttemptExcise, "f&sOnMatch(c^.status,[~valid,~red,~gray,~eA],[-,-,gray,-])", ensures that an element can be colored gray

only if its status is [~valid,~red,~gray,~eA]. We replace this conditional swap with an unconditional swap which colors the element gray and returns the previous status. If the value returned from the swap is [~valid,~red,~gray,~eA] or [~valid,~red,gray,~eA], then execution of the unconditional swap is equivalent to execution of the conditional swap: in the first case the process has succeeded in coloring its target element gray; in the second, the process colors an element gray that is not eligible as a target element but since the element was already gray and the process is informed that it was previously colored, the coloring is without effect. If the value returned from the swap operation does not match [~valid,~red,~gray,~eA] but does match [-,-,~gray,-], then the process has colored an element gray that is not eligible as a target element because it is claimed for excision by some other process or is marked as the beginning or ending anchor for an excision. The process must abandon the attempt to excise the element but must first indicate that it is no longer attempting to excise the element by uncoloring it. The process can unconditionally reset the gray bit because it knows that it was the process that colored the element and no process relies on an element remaining gray unless it colors the element itself.

The second conditional swap, "f&sOnMatch(bAnchor^.status,[-,~red,~gray,-],[-,bA,-,-])", ensures that only an element that is not red and not the target element for another excision can be marked as a beginning anchor. This conditional swap is more difficult than the first to eliminate. The correctness of the garbage collection procedure which will be described in the next section depends on maintaining the truth of the assertion that a target element cannot be used as an ending anchor, which is equivalent to asserting that the predecessor of a target element cannot change. If a process can tentatively mark an element as a beginning anchor before it knows that it can use the element as a beginning anchor, then it can mark an element which is the target element for another excision as its beginning anchor. If another process encounters that element while scanning for its ending anchor, it will use the element as its ending anchor.

To prevent a process from even tentatively marking another process's target element as a beginning anchor, we require that a process attempting to mark a beginning anchor first color the beginning anchor gray, using the swap operation instead of the store so it can know whether the coloring was successful. If the element was already gray, the coloring had no effect and the process abandons its attempt

to mark a beginning anchor. If the element was not gray, then the process knows that it succeeded in coloring the element gray and that the element can be used as a beginning anchor if the process can also succeed in coloring it red. The process executes a swap, marking the element as its beginning anchor and resetting the gray bit. If the element was already red, marking the element as a beginning anchor has no effect. If the element was not already red, the process has succeeded in marking its beginning anchor. As in the original version of AttemptExcise, a process can mark an element as a beginning anchor only if it is not red and not the target element for another excision.

3.3.3. Third stage algorithm

The batched garbage collection procedure which we have relied upon up until now uses space inefficiently. Since the garbage collector frees elements in batches and cannot free any element in a batch until all the processes active in the list at the time any element in the batch was excised have completed their current list access, there is likely to be a long average delay between the time an element becomes garbage and the time it can be reused by the system. In the next two sections, we describe how garbage can be detected and retrieved on an individual instead of a batched basis. In this section, we make the simplifying assumption that a process can read the switch field in an element and increment the intransit field indicated by the switch field in a single atomic action. The algorithm described in this section is practically identical to the algorithm described in [KuL80]. In the next section, we eliminate the requirement for an atomic read and write.

In the modified algorithm the collectors not only physically excise invalid elements from the list, but they also add the elements they excise to the free space list. The garbage collection queue and the maintenance process responsible for periodically appending the garbage collection queue to the free space list are eliminated --- elements are added directly to the free space list by the collectors as soon as the elements are known to be garbage.

Collectors identify garbage by means of reference counts. Reference counts are maintained by both searchers and collectors. As a searcher or collector traverses the list, it updates the reference counts of the elements it visits to reflect its presence in the list. When a collector excises a sublist of elements, it

decides when the sublist can be freed by reading the reference counts of the beginning anchor and the elements on the excised sublist.

Each element contains three reference count fields --- a census field and two intransit fields --- and a switch field for selecting between the intransit fields. The census count acts as a "check in" mechanism. The first action a searcher or collector takes upon visiting an element is to "check in" at the element by incrementing its census count. Since a process must obtain a pointer to an element before it can increment a field in the element, the census count cannot, by itself, give an accurate count of the number of processes that may reference the element. A count is needed of the number of processes intransit, i.e., the number of searchers that hold a pointer to the element but have yet to increment the element's census count. Each element records not only its own census count but a count of the number of processes intransit from it to the succeeding element. Since an element's forward pointer can change when it is the beginning anchor for an excision, each element actually keeps two intransit counts, one associated with the current, and the other with the immediately preceding value of its forward pointer. The element's switch is flipped each time the forward pointer is changed so that it indicates the intransit field that is associated with the current pointer. A searcher increments the current intransit field before it reads the forward pointer to the successor element and it decrements that same intransit field after it arrives at the successor element, i.e., after it has incremented the successor element's census count.

The new search procedure, redefined to maintain reference counts, is as follows:

```
{ s := fetch(H^.switch); inc(H^.intransit[s]) };
p := H;
c := fetch(H^.f);
while c <> nil do
   inc(c^.census);
   dec(p^.intransit[s]);
   if fetch(c^.data) = target
     then if match(swap(c^.status,[~valid,-,-,-]),[valid,-,-,-])
            then dec(c^.census);
                 exit;
   { s := fetch(c^.switch); inc(c^.intransit[s]) };
   dec(c^.census);
   p := c;
   c := fetch(c^.f);
   end while;
dec(p^.intransit[s]).
```

The braces, "{ }", enclose statements that must be executed atomically. The memory operations "inc(V)" and "dec(V)" are equivalent to f&add(V,1) and f&add(V,-1), respectively.

A process begins its search by incrementing the intransit count at the header node. At each subsequent element the searcher first increments the census count and then decrements the intransit count of the element it just left. If the element does not match the target or is invalid, the process leaves the element by incrementing the intransit count, decrementing the census, and fetching the pointer to the next element. The local boolean variable "s" is used to remember which intransit field was incremented so that the same intransit field can be decremented after the searcher arrives at the next element. If the element matches the target and the process succeeds in invalidating the element, then the searcher exits the search routine after first decrementing the census of the element it invalidated. The collect procedure, given in Appendix B, is modified in the same way as the search procedure, i.e., it is the same as the first version of the collect procedure except that it updates reference counts as it searches for an element eligible for excision.

The AttemptExcise procedure does not change when reference counts are introduced, but the Excise procedure must be modified substantially since a process executing the excise procedure now has the additional responsibility of determining when the excised element or elements can be freed. The new Excise procedure is as follows, where changes are marked by double asterisks:

```
   dptr := c;                                   remember location of target
   repeat                                       look for ending anchor
      c := fetch(c^.f);
      eStatus := swap(c^.status,[-,-,-,eA]);
      if match(eStatus,[~valid,~red,-,-])
        then eStatus := swap(c^.status,[-,claimed,-,-]);
   until not match(eStatus,[~valid,~red,-,-]);
   store(c^.b,bAnchor);                         link anchors
   store(bAnchor^.f,c);
** s := f&Complement(bAnchor^.switch);
** inc(c^.census);
** store(c^.status,[-,-,-,~eA]);
** repeat                                      wait at beginning anchor until
** until fetch(bAnchor^.intransit[s]) = 0       processes are intransit
   store(bAnchor^.status,[-,~bA,-,-]);          to the excised sublist
** dec(dptr^.census);
   e := dptr;
** while e <> c do                             traverse excised sublist
**    repeat
```

```
**    until fetch(e^.census) = 0;
**    repeat
**    until fetch(e^.intransit[0]) = 0;
**    repeat
**    until fetch(e^.intransit[1]) = 0;
**    p := e;
**    e := fetch(e^.f);
** end while;
** store(p^.f,nil);
** disposelist(dptr).                          free excised elements
```

Recall that ''bAnchor'' is a pointer to the beginning anchor passed to Excise as a parameter. The memory operation ''f&Complement(V)'', where V is a boolean variable, complements V and returns the old value.

The new procedure is the same as the original procedure through the act of linking the anchors. After the anchors are linked, the excising process changes the switch setting in the beginning element so that other processes traversing the list will stop incrementing the intransit count associated with the old value of the forward pointer. In an earlier version, processes searching the list were required to wait at any element marked as the beginning anchor for an excision until the excising process linked together the anchors and unmarked the beginning anchor. The introduction of a second intransit count and a switch to indicate the count associated with the current pointer allows searchers and collectors to continue through beginning anchors. Two intransit fields are needed if searchers do not block at a beginning anchor because a separate count is needed for the number of processes intransit to each of the element's two successors, the old and the new successor. Although an element may have many successors while it is on the list, two intransit fields are sufficient because as long as there are processes that may have read the pointer to the old successor, i.e., as long as the intransit count associated with the old successor is not zero, the element remains marked as a beginning anchor and its forward pointer cannot be changed again.

After complementing the switch in the beginning anchor, the excising process increments the census of the ending anchor. This action gives the excising process a secure point from which to continue its search for invalid elements upon return to the collect procedure. When the excising process resumes execution of the collect procedure, its current element will have changed from the now excised target element to the element that was the ending anchor in Excise. After the anchors are unmarked in

Excise, they are subject to excision, and can be freed if their reference count is zero. By incrementing the census of the ending anchor before it is unmarked the process ensures that the element will not be freed before the process resumes its search. There is an additional reason for incrementing the ending anchor's census count that will be described below.

After incrementing the ending anchor's census, the excising process unmarks the element as an ending anchor. Before unmarking the beginning anchor, it waits until the intransit count associated with the beginning anchor's old successor, the target element, goes to zero. Since there are only two intransit fields, the process must prevent another process from marking the element as a beginning anchor, redirecting its forward pointer, and changing the switch setting while the process still needs to read the intransit field associated with its target element.

Since a process increments the intransit count associated with the current switch setting before reading the forward pointer and an excising process changes the switch setting only after it changes the forward pointer, the intransit count indicated by the old switch setting is an upper bound on the number of processes intransit from the element to the old successor. When the intransit count associated with the target element goes to zero the excising process will know that all searchers and collectors that saw the old pointer to the target element have arrived at the target element and incremented its census. Since it no longer needs the intransit count at the beginning anchor, it can now unmark the element as its beginning anchor.

After unmarking its beginning anchor, the excising process decrements the census count of its target element to avoid deadlocking itself. (The process incremented the census of the target element in the collect procedure before the call to AttemptExcise.) If the process waits for the target element's census to equal zero without first decrementing the census, it will wait forever. Alternatively, the process could treat the target element as a special case and wait for its census count to go to one instead of zero.

The excising process now traverses the list in the direction of search starting at the target element and waiting at each element first for the census count and then for the two intransit counts to equal zero. Note that it is possible for the census and intransit counts to increase while the process is waiting. The reference counts at an excised element, e, can increase because e may have been marked as an ending

anchor for a previous excision. Processes searching along an old excised sublist ending at e can arrive at e and increment e's census count.

Once the excising process sees e's census count go to zero, however, the count will remain zero. The excising process does not read e's census until after all processes that can arrive at e through e's current predecessor have done so. (We define e's current predecessor for this discussion as the beginning anchor if e is the target element and as the element preceding e on the excised sublist otherwise.) Since all the processes that can arrive at e from its current predecessor have arrived, the only processes that can increment e's census count are processes arriving from previously excised predecessors of e. But the census count at e cannot go to zero until all the searchers leave previously excised sublists ending at e. A process that excised a sublist ending at e must have incremented e's census count before unmarking e as the ending anchor for its excision and cannot decrement e's census count until it returns to the collect procedure. Since the process cannot return to the collect procedure until after all searchers have left the sublist it excised, the census count at e cannot go to zero until all the processes that can get to e from previously excised sublists ending at e have arrived. Since no processes can arrive at e from either e's current or e's previously excised predecessors after the process that excised e sees e's census go to zero, e's census will remain zero.

When the census and intransit counts of all the elements on the excised list have gone to zero, the memory held by the excised elements can be released. The excising process sets the forward pointer of the list element on the excised list equal to nil (in order to avoid disposing of the entire remainder of the list), and adds the excised sublist to the system's freelist. The procedure ''disposelist'' is assumed to be a procedure for appending a sublist of excised elements to the free space list. We will not describe the free space list except to note that the flexible queue described in [Rud82] would be a good representation.

Correctness Argument: Third Stage Algorithm

The modified algorithm is correct if it maintains correctness properties Prop1 through Prop6 and two additional properties:

Prop7 Only elements which have become garbage are added to the free space list.
Prop8 An element which has become garbage is eventually added to the free space list.

An element has become garbage if no process can ever access the element again. We assume that no process accesses an element, once the element has been inserted into the list, except through the list access procedures we define.

The changes introduced in the third stage algorithm do not affect Prop1 through Prop4. In this section we give an informal argument that the algorithm maintains Prop5 through Prop8. We discuss Prop7 first.

Prop7: correctness argument

We need to show that if an element is freed, then it must be garbage. An element, e, is freed if e is on the free space list. We consider each list access procedure and argue that no process executing the procedure can reference a freed element, i.e., if e is freed then no process can reference e. If no process can reference e, then e is garbage.

Prop7 Insert procedure: correctness argument

An inserting process receives a pointer to a list element when it performs the swap operation on TAIL. The element referenced by that pointer, the predecessor of the new element, must not be freed before the inserting process links it to the new element. The predecessor cannot be freed because it cannot be excised and an element must be excised before it can be freed. The predecessor cannot be excised because its red bit has been set since before it was inserted into the list and no process can delete an element colored red by another process. The inserting process resets the red bit in the predecessor element as its last reference to the element at the end of the insert procedure.

Prop7 Excise procedure: correctness argument

A process executing the Excise procedure references only its anchors and the intervening elements. We have already shown that the anchors remain on the list while they are marked as anchors and that the target element remains on the list up to the time that the anchors are linked. Any element that the process references while scanning for its ending anchor is also on the list. If an element is on the list and cannot

be excised or used as a beginning anchor by another process, then the element's successor is also on the list and cannot be excised. The excising process begins its scan at its target element, an element that cannot be excised except by the excising process itself. Before reading the forward pointer, the process claims the element for excision. Claiming the element prevents its use as a beginning anchor. After the element is claimed, its successor cannot change so if the element remains on the list, its successor remains on the list also. Since the excising process claims every element that it encounters during its scan other than the ending anchor, every element the excising process encounters must remain on the list until the process excises the elements itself by linking the anchors. Since an element must be excised before it can be freed and the excising process references only elements that are on the list through the time of the excision, an excising process cannot reference a freed element though the time of the excision.

After the excision, i.e., after it redirects the beginning anchor's forward pointer, the process references only its anchors and the elements it has itself excised and not yet freed. Since the excising process references its anchors only while they are still marked as anchors the anchors must be on the list and not freed at the time they are referenced. By Prop1, an element can be excised by only one process. No other process can have excised and freed an element excised by this process. We conclude that an excising process cannot reference a freed element.

Prop7 Search and Collect procedures: correctness argument

Searchers and collectors update reference counts in the same way so we consider them together. In this section we refer to both searchers and collectors as searchers.

The first reference a searcher makes to an element is an increment to the element's census field. Before decrementing the census field, it increments the current intransit field. Before decrementing that intransit field, it increments the census of the successor element. Thus a searcher always keeps a "toehold" into the list. It establishes its first toehold at the header node, an element known to be on the list, and then it never relinquishes a toehold until it has established the next or decided to stop its search.

The dangerous point for the searcher is the first reference to an element --- the increment to the element's census count. Once the element's census is incremented, then the searcher can be sure that the element will not be freed until after the searcher has finished referencing the element. We can show that a searcher can safely increment the census of the next element if the following assertion is true:

> Let b be an element, either deleted or on the list, that has
> not been freed. If a searcher increments b's current intransit
> count and then reads the pointer to b's successor, c, then
> when the searcher increments c's census count, c will not be freed.

If the assertion is true, then by simple induction no searcher references a freed element, since a search always begins at the header node and we know that the header node is not freed.

We argue that the assertion is true by considering each of six cases. At the time c's census is incremented, c and c's predecessor, b, can be excised or not excised as follows:

Case 1:	b is not excised	c is not excised;
2:	"	c is excised;
3:	b is excised	c is not excised;
4:	"	c was excised before b;
5:	"	c was excised at the same time as b;
6:	"	c was excised after b;

Since an element that is not excised cannot be freed, the assertion is trivially true for cases 1 and 3.

Case 2: b is not excised; c is excised

Case 2 occurs when b's forward pointer is redirected between the time the searching process reads b's forward pointer and the time it increments c's census. At the time the searcher increments c's census, c has been excised but b has not.

We have the following sequence of events:

```
P1: SEARCHER                P2: EXCISER

S1: {sw1 := b^.switch;
    inc(b^.intransit[sw1]}
S2: read b's f ptr to c
                            E1: redirect b's f ptr from c to d
S3: inc(c^.census);         E2: sw2 := f&Complement(b^.switch);
                            E3: wait til b^.intransit[sw2] = 0;
S4: dec(b^.intransit[sw1]);
                            E4: free c.
```

The assertion is true for case 2 if E4 cannot occur before S3. Since E4 cannot occur until after the condition in E3 is satified and the condition in E3 cannot be satisfied until after S4 is executed, E4 cannot occur before S3.

The claim that the condition in E3 cannot be satisfied until S4 is executed depends on the truth of the claim that sw2 = sw1, i.e., that the intransit field which the excising process, P2, is waiting to go to zero is the same as the intransit field incremented by the searcher, P1. Assume on the contrary that sw1 does not equal sw2. Then some other process, P3, complemented c's switch between the time P1 executed S1 and P2 executed E2. Since at the time a process changes an element's switch the element must be marked by that process as its beginning anchor, c was marked as P3's beginning anchor at some time between the execution of S1 and E2. Furthermore since an element cannot simultaneously be the beginning anchor for more than one process, P3 must unmark c as its beginning anchor before P2 can mark c as a beginning anchor and execute E2. But P3 could not have unmarked c as its beginning anchor until the intransit count indicated by sw1 was zero, i.e. until P1 executed S4. We conclude that no process changed the switch between execution of S1 and E2, i.e., s2 = s1, and that the assertion is true for case 2.

Case 4: b is excised; c was excised before b

Case 4 cannot occur. If c is excised before b but c was b's successor at the time the process read b's forward pointer, then b's forward pointer changed between the time the searcher read the pointer to c and the time it incremented c's census. Therefore b is a beginning anchor and on the list at some time after the searcher reads b's pointer to c. For case 4 to occur, b must be excised before the searching process increments c's census. Both b and c would then be excised, with c excised before b, at the time a searcher tries to increment c's census. But b cannot be excised while there are processes intransit from b

to c. Element B must remain marked as a beginning anchor until the intransit count associated with c becomes equal to zero. Since a searching process will not decrement b's intransit count until after it increments c's census, the intransit count at b associated with c will not equal zero until after all the searchers intransit to c have incremented c's census.

Case 5: b is excised; c was excised at the same time as b

If c is b's successor and b and c were excised at the same time, then there was some process, P1, which excised b and c when it redirected the forward pointer in its beginning anchor. If b has not been freed and b's intransit count has been incremented by a searching process then b cannot be freed until after the searcher decrements b's intransit count. Since the searcher must increment c's census before decrementing b's intransit, b cannot be freed until after c's census is incremented. By examination of the code for the excise procedure, P1 frees all the elements it excised at once. Since P1 cannot free b before c's census is incremented, it cannot free c either. Therefore, c is not freed at the time its census is incremented.

Case 6: b is excised; c was excised after b;

We described how case 6 can occur in our discussion of the stage 3 excise procedure. The ending anchor, c, for an excision can be unmarked and excised before all the searchers traversing the excised sublist arrive at c. Element c cannot be freed while there are still processes searching the excised sublist because the excising process responsible for the original excision increments the census of its ending anchor, c, and does not decrement it until after all the searching processes have arrived at c.

As an example, consider the two excisions illustrated in figure 3.4. Assume that the list initially consists of five elements, the header node, H, and elements a, b, c, and d, in the order given. Consider the following sequence of events:

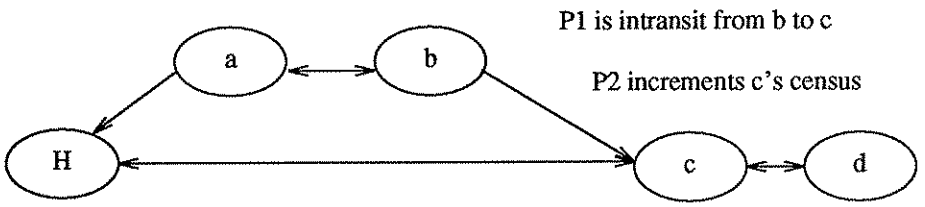
(1) A searcher, P1, suspends execution after incrementing b's intransit and decrementing b's census count;

(2) An excising process, P2, excises a and b using H and c as its anchors; P2 unmarks c as its ending anchor and begins waiting for searchers to leave the excised sublist;

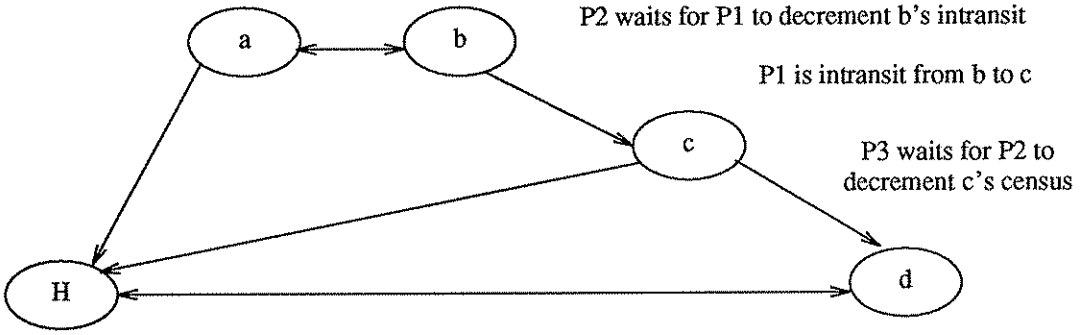
(3) Another excising process, P3, excises c, using H and d as its anchors.

Figure 3.4

(a) After P2 deletes a and b



(b) After P3 deletes c



P2 is now waiting at b for P1 to decrement b's intransit count and P3 is waiting at c for P2 to decrement c's census. P3 cannot free c until P2 completes execution of Excise and resumes its search for invalid elements. Since P2 cannot return from Excise until after all the searchers visiting a and b have arrived at c, no searcher can arrive at c from b and find that c has been freed. Note that if P2 had not incremented c's census count c could be freed by P3 before P1 increments c's census. We note that an error in the individual garbage collection algorithm described in [KuL80] is the failure to handle this case.

Prop7 AttemptExcise Procedure: correctness argument

A process executing the attemptExcise procedure references two elements, its target element and the predecessor of its target element, the candidate beginning anchor. The process increments the census of the target element in the collect procedure so the target element is not freed. We must show that the candidate beginning anchor cannot be freed. In outline, we argue that the predecessor of a target element cannot change. Therefore, if the candidate beginning anchor is excised, the target element must have

been excised with it. If the candidate anchor and the target element are excised together, then neither can be freed until the other can be freed. The target element cannot be freed because its census count is not zero. Therefore, the candidate beginning anchor cannot be freed.

The predecessor of a target element cannot change because a target element cannot be used as an ending anchor. If element c is a target element we know that c is invalid and that it is not claimed as a beginning anchor or marked as the end of the list since c's status had to be [~valid,~red,~gray,~eA] before c was marked as a target element, i.e., colored gray, and no process can mark a gray element as a beginning anchor. If c is invalid and not marked as a beginning anchor or marked as the end of the list, then if a process encounters c while scanning for an ending anchor, c's status will be invalid and not red. C cannot have been claimed by another element because that would imply that the scanning process has continued its scan through the beginning anchor of another process. Since c is both invalid and not red, the scanning process will claim c, not use c as its ending anchor. Note that the process will mark c and every other element it encounters during its scan as an ending anchor, but it will only use c as its ending anchor if it is either valid or red.

Prop5: correctness argument

The modified algorithm maintains Prop5. The insert procedure is unchanged. The search procedure is changed only by the addition of the constant number of memory references per element required to update the reference counts. If we assume that deadlock cannot occur, then excisions terminate, where an excision includes the work of freeing the excised sublist. Excisions terminate because searchers will eventually leave the excised section of the list, decrementing the reference counts as they go, allowing the excising process responsible for the excision to free the excised elements and terminate. The intransit count associated with the excised sublist at a beginning anchor for an excision will eventually go to zero because the excising process changes the switch setting before it waits for the count associated with the old switch setting to go to zero. Future searchers will see the new switch setting and increment the other intransit count.

Prop6: correctness argument

Assume that processes executing the stage 3 algorithm deadlock. Since searchers and inserters never wait, the circular waiting implied by deadlock must be among excising processes. An exciser waits for another exciser only when it is trying to free the element that was the ending anchor of the other process's excision. It is waiting because it cannot free this element until after the other exciser decrements the census count of its old ending anchor. Initially assume that three processes are deadlocked. Assume P1 is waiting for P2 to decrement the census of P2's ending anchor, c2, and that P2 is waiting for P3 to decrement the census of P3's ending anchor, c3, and that P3 is waiting for P1 to decrement the census of P1's ending anchor, c1. By examination of the code for the excise procedure, we contend that the following assertions are true:

D1: an exciser claims the elements it excises (and subsequently frees) before marking the ending anchor for its excision; and
D2: if an element is claimed for excision it cannot subsequently be marked as an ending anchor.

Given these assertions, we know that the following events occurred in the order given:

(1) P2 claims c3 for excision;
(2) P2 marks (and unmarks) c2 as its ending anchor;
(3) P1 claims c2 for excision;
(4) P1 marks (and unmarks) c3 as its ending anchor; and
(5) P3 claims c1 for excision.

We know that event 1 occurs before event 2 and event 3 before event 4 by assumption D1. Event 2 occurs before event 3 and event 4 before event 5 by assumption D2. When does P3 mark c3 as its ending anchor? By D1, this event must occur after event 5. But by D2 c3 must be marked as an ending anchor before event 1. We conclude that a deadlock involving three processes cannot occur.

Now assume that $n > 3$ processes are deadlocked. The addition of more processes into the set of deadlocked processes increases the length of the cycle but cannot introduce a new case, i.e., a different relationship among the deadlocked processes. Every deadlocked process is directly waiting on exactly one other process from among the set of deadlocked processes. Only excising processes can be deadlocked, and the only reason that one excising process, P1 waits on another, P2, is for P2 to decrement the census of P2's ending anchor. Since no two processes have the same ending anchor, a

deadlocked process can wait on at most one deadlocked process. Furthermore, for every deadlocked process, P1, there is at most one other deadlocked process directly waiting on P1 since P1's ending anchor can be in the excised sublist of at most one process. Given that each deadlocked process is waiting on exactly one other deadlocked process and that at most one process is waiting on any given deadlocked process, we can conclude that the deadlocked processes form a cycle. We can label the deadlocked processes 1 through n, so that Pi waits for Pj iff j = (i+1) mod n, and generalize the argument that a cycle of three processes implies a contradiction to a cycle of n processes.

Prop 8: correctness argument

An exciser frees every element it excises because the search path it follows in freeing elements is the same as the search path that existed at the time the anchors were linked. No other process can excise an element that was on the search path from the beginning to the ending anchor at the time the anchors were linked, because, by Prop1, an element can be excised by only one process. An exciser frees every element it excises before it terminates and, by Prop5, excisions terminate. Since every excised element is eventually freed and only excised elements can be garbage, every excised element is eventually freed.

3.3.4. Fourth stage algorithm

In the third stage algorithm we assumed that a process could read the switch field in an element and increment the intransit field indicated by the switch field as an atomic action. We now show that this atomic action can be eliminated.

Only the search and collect procedures need to be modified. The new search procedure is as follows:

```
** s1 := fetch(H^.switch);
** inc(H^.intransit[s1]);
** s2 := fetch(H^.switch);
** if s1 <> s2
**    then inc(H^.intransit[s2]);
   p := H;
   c := fetch(H^.f);
   while c <> nil do
      inc(c^.census);
      dec(p^.intransit[s1]);
**    if s1 <> s2
```

```
**        then dec(p^.intransit[s2]);
        if fetch(c^.data) = target
          then if match(swap(c^.status,[~valid,-,-,-]),[valid,-,-,-])
                 then dec(c^.census);
                   exit;
**      s1 := fetch(c^.switch);
**      inc(c^.intransit[s1]);
**      s2 := fetch(c^.switch);
**      if s1 <> s2
**        then inc(c^.intransit[s2]);
        dec(c^.census);
        p := c;
        c := fetch(c^.f);
        end while;
      dec(p^.intransit[s1]);
** if s1 <> s2
**    then dec(p^.intransit[s2]);
```

As before, changes are indicated by double asterisks. The collect procedure is modified in the same way as the search procedure.

A searcher reads the switch field twice, once before and once after incrementing the intransit field indicated by the first reading. If the two readings give different values for the switch, the process increments the remaining intransit field. After incrementing the intransit count, the searcher reads the pointer to the next element. If the current element is the last element on the list, the searcher decrements the intransit field or fields it just incremented and terminates. If the current element has a successor, the searcher first increments the census of the successor before decrementing the intransit field or fields it incremented before reading the forward pointer.

If we eliminated the atomic read and write without requiring the searcher to confirm that it has incremented the current intransit field a dangling reference could result, as follows:

```
SEARCHER                    EXCISER1
read b's switch = s;        redirect b's f ptr from c to d;
                            change switch from s to s';
                            wait til b^.intransit[s] = 0;
inc(b^.intransit[s]);       free c;
read b's f ptr = d;         EXCISER2
                            redirect b's f ptr from d to e;
                            change switch from s' to s;
                            wait til b^.intransit[s'] = 0;
                             free d;
inc(d^.census);
```

By the time the searcher increments b^.intransit[s], b's switch equals s'. Another exciser can now excise and free the current successor, d. This second exciser must wait for b^.intransit[s'] to equal 0 before freeing d. But, since the searcher incremented b^.intransit[s], not b^.intransit[s'], the exciser can free d and the searcher's subsequent reference to d will be erroneous. The error can occur because the searcher has not incremented the current switch before fetching the pointer to the successor element.

In the fourth stage algorithm we require that the searcher read the switch twice, and, if necessary, increment both intransit counts, to ensure the truth of the following assertion:

> Before a searcher reads the pointer in the current element, b,
> there exists a time, t, such that at time t
> the searcher has incremented intransit field s and
> s is the current value of b's switch.

The assertion is true in the third stage algorithm since a process reads the switch and increments the corresponding intransit field as an atomic action. The assertion is also true of the stage 4 algorithm. If s1, the value of the switch before incrementing an intransit count, and s2, the value of the switch after incrementing intransit[s1], are the same, then let time t be the time of the second read of the switch value. If s1 and s2 are different, then let time t be the time the process increments the second intransit count.

Given that the assertion is true for the stage 4 algorithm and further that no searcher decrements an intransit count in element, b, that it is currently visiting until after it increments the census of b's successor, a searcher can safely reference the next element. Between time t and the time the searcher decrements b's intransit field or fields, no exciser can free the element, d, that at time t is b's successor. For example, we could have the following sequence of events:

```
SEARCHER                        EXCISER
read b's switch = s;
increment b^.intransit[s];
read b's switch = s;
read b's f ptr = d;
                                redirect b's f ptr from d to e;
                                change switch from s to s';
inc(d^.census);                 wait til b^.intransit[s] = 0;
```

An exciser can excise d, but it cannot free d until b^.intransit[s] goes to zero and b^.intransit[s] cannot equal zero until after the searcher decrements it. Between the time a searcher increments b's current intransit field and the time it decrements the same field, at most one successor of b can be excised and that element cannot be freed. Since a searcher reads b's f pointer only after incrementing the current intransit field, it will not read a pointer to an excised element.

A searcher could, of course, ensure that the assertion is true by routinely incrementing both intransit counts before fetching the pointer to the next element. This solution would undermine Prop5. Given a steady stream of searchers through its beginning anchor, an exciser could wait forever for the intransit field associated with the pointer to the excised sublist to go to zero. In the solution we propose, the intransit count will eventually equal zero because the only time a searcher increments both intransit counts is when its two readings of the switch straddle a change in the switch. All searchers that arrive at the beginning anchor after the exciser changes the switch will increment only the intransit count associated with the new switch setting.

3.3.5. Fifth stage algorithm

In this final transformation of the list access procedures, we eliminate the maintenance processes. In some applications, we may prefer to leave the maintenance processes in the system. Maintenance processes relieve the user processes of work. But there are two drawbacks to the use of maintenance processes in the list access procedures. First, the use of maintenance processes to excise elements probably increases the amount of time an invalid element stays on the list and thus the number of elements a searcher must examine. If user processes excise the elements they invalidate, then the invalid elements would probably remain on the list for a shorter time than if the user processes left the invalid elements for maintenance processes to discover and excise. Second, the use of maintenance processes increases the amount of process management overhead.

We eliminate maintenance processes by simply transferring their work to the user processes. The collect procedure is eliminated and the work of checking for and excising invalid elements is added to the search and excise procedures. This transformation is simple enough to require no further informal expla-

nation. The code for the fifth stage algorithm is given in Appendix B.

3.4. Summary

We have described a set of list access procedures which allow concurrent access to a shared list in global memory. The procedures allow processes to insert elements at the tail of the shared list, to search the list, and to delete and excise elements from anywhere in the list. If maintenance processes are used to excise and free elements, then no user process must wait for any other process in accessing the shared list. If maintenance processes are not used, then a process must wait in only one case: a process that has excised an element waits until the element becomes garbage before putting it on the free space list. We have stated correctness criteria for the algorithm and argued informally that the algorithm is correct.

CHAPTER 4

A DATABASE APPLICATION

We describe a database application for the list access procedures described in chapter 3. The application is a variant of an algorithm proposed by H.S. Stone for implementing the wait-die and wound-wait protocols on a machine capable of executing the Ultracomputer fetch-&-add instruction [Sto84]. Stone's proposed implementation is based on fixed-size queues which must be large enough to accomodate worst-case demands for queue space and so is costly in its space requirements. Our list access procedures provide the basis for an implementation that is more efficient in its use of space.

4.1. Stone's Database Control Algorithm Stone proposes implementing the wait-die and wound-wait protocols of Rosenkrantz, Stearns, and Lewis [RSL78] using the Ultracomputer queue. The proposal does not alter the complexity of the protocol (O(N), where N is the number of concurrent lock requests). Stone's goal in proposing the use of the Ultracomputer is the elimination of unnecessary serialization. He contends that the implementation of the protocol should not itself introduce any serialization not required by inconsistent demands for access to the database.

The need to maintain consistency in the database requires a mechanism by which a process can ensure that no other process accesses records it uses in a transaction until the transaction is complete. The wait-die and wound-wait protocols assign a time-stamp to each process that needs to access the database. The time-stamps are used to resolve lock conflicts. In the wait-die protocol, if a process, P1, needs to lock a record locked by P2, P1 waits if P1 is older (has a lower time-stamp) than P2 and "dies", i.e., is rolled-back, if P1 is younger than P2. In the wound-wait protocol, P1 "wounds", i.e., rolls-back P2 if P2 is younger and holds a lock on a record required by P1 and waits if P2 is older. The ordering imposed by the time-stamps prevents deadlock. Furthermore, the time-stamps prevent starvation since a process that is repeatedly rolled-back will eventually become the oldest process.

A lock may lock a relation, a record, or a field in a record --- the choice is made by the designer of the database. Stone assumes that every unit of the granularity chosen by the database designer has a unique identifier. A request to lock a unit of the database names this identifier. The process requesting the lock searches the list of active lock requests for a conflict, i.e., a lock request with the same identifier. Hashing is used to reduce the expected length of the list that must be searched for a conflict.

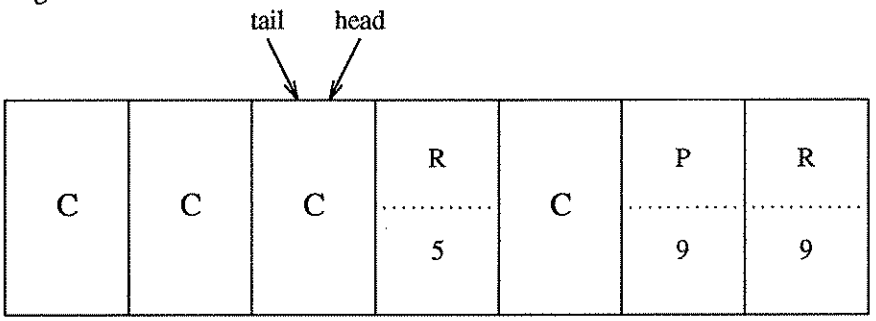
Stone proposes that each of the lists generated by hashing be represented as an Ultracomputer queue and that time-stamps be assigned using the fetch-&-add instruction. The proposals eliminate the serialization that is otherwise required to assign each process a unique time-stamp and maintain the consistency of the lists. A process secures a lock by first enqueueing a lock request record in the Ultracomputer queue to which the lock identifier hashes. The queue is treated as a circular array --- the pointers to the head and tail of the active portion of the queue wrap around the end of the array. If the lock request queue is full, the process busywaits until space is available. After enqueueing a lock request record, the process marks the record "active", reads and remembers the current positions of the head and tail, and examines the active lock requests between the remembered locations for a conflict. Conflicts are resolved in accordance with the protocol in use. When a process completes a transaction or is rolled-back it must release its lock requests. The procedure for releasing a lock request depends on whether the lock request record is the first record in the queue not marked "completed". If it is not the first such record, then a process can release the record by simply marking it "completed". If the record is the first record not marked "completed", then the process that enqueued the record is said to "own" the head pointer and is responsible for updating the pointer. The process advances the head pointer to point to the record one cell beyond its own record, marking all the cells over which it advances the pointer "empty". The process may need to advance the head pointer over more than one cell because one or more records marked "completed" may appear in the queue between the head pointer and the process's own record. This complication arises from the representation of a data structure that is essentially a list with a queue. Lock requests are not necessarily released in FIFO order. A cell between the head and tail of the queue may be unused because a lock was released out of order, but the cell cannot be reused until all the cells preceding it on the queue have been released and the head pointer advanced to point beyond it.

4.2. Space Complexity of Stone's Algorithm

Stone does not discuss the space requirements of his database control algorithm. No queue size is specified. The algorithm requires that a process wait if it needs to enqueue a lock request in a lock queue that is already full. Queue sizes can presumably be adjusted to reach an acceptable tradeoff between the space required for the queues and the time processes must wait. We contend, however, that if a process must wait, then deadlock is possible. Queues must therefore be large enough so that space for every lock request is guaranteed.

The potential for deadlock arises from the fact that space on the lock request queue is itself a resource needed to complete a database transaction. Locks are allocated using the time-stamp order but space on the lock queue is allocated on a first-come-first-serve basis. Since these orderings can be inconsistent, circular waiting is possible. Circular waiting can occur within a single queue or over several queues. Figure 4.1 illustrates how circular waiting can arise within a single queue. Although several cells have been marked "completed" (indicated by the "C's"), the queue illustrated is full. Process R is the owner of the head pointer because its lock request for unit 5 of the database is the first noncompleted entry in the queue. The queue will remain full until R completes its transaction or is rolled-back. Process P is waiting to enqueue a lock request. Initially, assume that the wound-wait protocol is used and that process P is older than R. Process R waits for process P because P has enqueued a lock request for unit 9, a unit also needed by R to complete its transaction that hashes to the same queue. Since process P is

Figure 4.1



P waits to enqueue a lock request

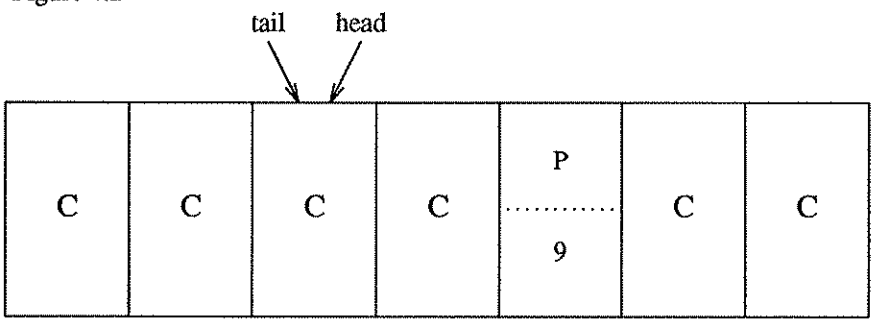
waiting for R to release space on the queue, processes P and R are deadlocked. The same figure illustrates deadlock in the wait-die protocol when we assume that process P is younger than process R.

A process can even deadlock itself, as illustrated by figure 4.2. Although all the cells except one are unused, the queue is full. Process P owns the head pointer. No space can open up on the queue until P completes or is rolled-back, but P is waiting (for process P) to release space.

Requiring that processes batch all requests destined for a given lock queue would solve both problems illustrated above. But deadlock could still arise over several queues, as illustrated in figure 4.3, and a requirement that requests be batched is not an acceptable solution in general because the identity of some records needed to complete a transaction may depend on the current value of other records. For such a transaction, a process would need to secure locks on some records before it could reliably identify other records needed to complete the transaction.

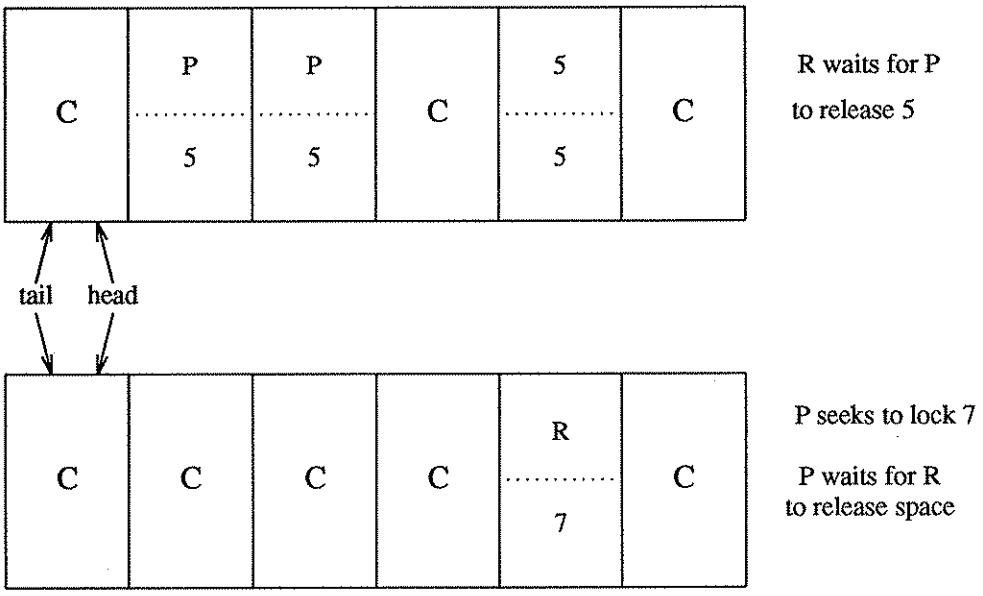
A more promising solution requires applying the same protocol to control access to space on the queue as is used to control locking. For example, if process P finds the lock queue full, it compares its time-stamp with the time-stamp of the owner of the head pointer, process R, and resolves the conflict in the usual way. Assuming the wound-wait protocol, if R is older than P, P waits as before, but if R is younger than P, R is rolled back, opening space on the queue. We note that this solution may require unnecessary rollbacks, i.e., rollbacks that are not required by lock conflicts (or anomalies in the protocol) but by the implementation of the protocol. Furthermore, the solution does not remove the possibility that

Figure 4.2



P waits to enqueue a lock request

Figure 4.3



a process may deadlock itself. Given a queue-based implementation of the lock request lists in which space on the queue is limited, we do not know a solution to this latter problem that does not raise the possibility of starvation.

If both deadlock and starvation are to be avoided, space on the queue must be effectively unlimited, i.e., every lock request must be guaranteed space on the queue. Since in the worst case every lock request hashes to the same queue, the algorithm requires space proportional to H * N * L, where H is the number of entries in the hash table, N is the maximum number of processes that may concurrently access the database, and L is the maximum number of locks that any one transaction can require.

4.3. A List-based Implementation

A more natural way to represent the groups of lock requests generated by hashing is with a list. Since lock requests may be released in an arbitrary order, special attention must be paid to circumventing the FIFO queueing discipline when a queue-based implementation is used. A list-based implementation, such as the implementation we propose, can, on the other hand, support deletions from arbitrary positions

within the list. More significantly, a list-based implementation is more reasonable in its demands for space. Since the size of a list, unlike the size of an ultracomputer queue, can change dynamically, the space required for a list-based implementation is proportional to the number of currently active lock requests plus H, the size of the hash table.

It is easy to see that the list access procedures described in chapter 3 provide the functionality needed for a list-based variant of Stone's database control algorithm. A process secures a lock by inserting a lock request record and searching the list for conflicts, resolving conflicts as dictated by the protocol in use. Roll-back and transaction completion requires release of lock request records. If the wait-die protocol is used, the release can be accomplished by the deletion procedure, i.e., without first searching for the record to be released. Assuming that the wounded process itself executes the roll-back, the deletion procedure can also be used in the wound-wait protocol.

We note that use of the list access procedures has one drawback as compared with Stone's ultracomputer-based algorithm --- a process searching the list for conflicts may have to wait while a process that is inserting a request between the head and the searching process's own request record completes its insertion.

Since insertions, deletions, and searches of the lock request lists can occur concurrently, the list-based variant of Stone's algorithm achieves the goal set by Stone of finding a way to implement the wait-die and wound-wait protocols in such a way that the protocol does not itself introduce serialization.

4.4. Summary

We have described an application of the list access procedures, namely, a list-based variant of Stone's implementation of the wound-wait and wait-die protocols for acquiring locks in a database. We contend that the list-based implementation is more natural and space-efficient than Stone's queue-based implementation of these protocols. Furthermore, it solves deadlock problems that we have identified in Stone's algorithm.

CHAPTER 5

CONCLUSIONS

5.1. Summary

We have considered the problem of providing highly concurrent access to a linked list in shared memory. This problem is a specific instance of the more general problem of designing data structures for highly concurrent systems so that access to the shared structure does not become a bottleneck.

We have proposed a set of asynchronous procedures for performing search, insert, and delete operations on a linked list. The procedures allow any number of processes to perform search, insert, and delete operations concurrently. The consistency of the list is maintained without introducing a critical section, and no access locks out other accesses. We have not eliminated waiting, but have limited it to two instances: an exciser must wait before freeing an element if other processes hold pointers to the excised element; and a searcher visiting an incompletely inserted element must in some cases wait until the insertion is completed.

The target architecture for the list access procedures is the ultracomputer. The ultracomputer architecture allows several processes to access the same memory location simultaneously, without serialization. This property of the ultracomputer has been used by other researchers to provide highly concurrent access to an array-based queue.

We believe that the list access procedures we have proposed will be useful in applications which require the flexibility of a linked list. We have described two such applications: an efficient critical-section-free implementation of P and V operations that maintains FIFO order among the waiting processes where the P operation is based on the insertion procedure; and an implementation of a database locking protocol which uses the full set of list access procedures we have defined.

We have also described a parallel garbage collection algorithm based on references counts. Use of the algorithm should cause garbage to be more promptly identified than it is by garbage collection

algorithms, such as marker-based algorithms, which collect garbage in batches. An additional advantage of the reference count algorithm we propose over marker-based algorithms is that it can be used even when processes hold pointers into the shared structure exclusively in local memory. The algorithm is very similar to a previously published algorithm, but corrects an error and eliminates an atomic read and write in that algorithm.

A simulation of the list access procedures we propose indicates that access time to the list is an almost constant function of the number of processes accessing the list, i.e., that performance does not degrade as the level of concurrency increases. A modification of an algorithm due to Kung and Lehman also achieves almost constant time access to the list.

5.2. Topics for Further Research

We note several open questions and topics for further research:

- Verification

The correctness proof that we have presented for the list access procedures is informal. We would like to obtain a formal proof. One potential approach is the use of the automatic model checker described in [CES86]. This approach, based on branching-time temporal logic, is applicable only to finite-state systems. The list access procedures described in chapter three do not form a finite-state system, but the dual of those procedures, restricted to lists of a bounded length, can be modeled by a system of finite-state automata. (We use "dual" here in a sense similar to the way in which it is used in [LaN79]: memory locations in the shared memory model correspond to processes in the message-based model, and memory access operations to messages.) Assuming that the number of states in the global state transition graph is less than a few thousand, we could use Clarke, Emerson, and Sistla's model checking algorithm to verify that the dual of the list access procedures conforms with specifications written in temporal logic.

Another approach which we may wish to explore is the derivation of the list access procedures from specifications, as proposed by Chandy and Misra in [ChM86]. An advantage of this approach is that it may yield a simpler algorithm.

• Simulation

We have simulated several algorithms for accessing a shared list, including the list access procedures we propose, using a discrete event simulation on a uniprocessor. We were primarily interested in the simulation as a means of buttressing our informal correctness argument. A secondary goal was to be able to compare the list access algorithms in terms of the rate at which the time for a list access increases as more processes are added to the system. Our performance analysis, summarized in appendix C, is too informal to serve as the basis for anything but tentative conclusions about this rate of increase.

A second shortcoming of the simulation is that we did not model queuing delays caused by collisions in the network. A more realistic simulation would model the operation of the network.

• Improvements

We note several possibilities for improving the algorithm:

(1) Eliminate the census count field by merging it with one of the intransit count fields.

(2) Simplify the procedure for marking a beginning anchor by requiring that a process keep the intransit count of each element visited incremented until it knows that it does not need to use the element as a beginning anchor.

(3) Decrease the time an invalid element remains on the list by routing around each invalid element as it is claimed instead of waiting until the entire sublist of invalid elements is identified. (This improvement may also reduce the number of status bits required by the algorithm).

(4) Eliminate the requirement that an excising process wait by devising a means by which the last process to visit an excised element can know that it is the last process and should free the element.

• Extension to other data structures

We may be able to extend the list access procedures or the garbage collection algorithm we have proposed to other data structures. Candidate data structures include circular lists, trees, graphs, Fibonacci heaps [FrT84], self-organizing lists [HeH85], and sorted lists. Our list access algorithm may be easier to extend to circular lists and graphs than an algorithm, such as Kung and Lehmans's, based on locking,

since a cycle can cause an algorithm based on locking to deadlock. Extension to self-organizing lists could entail defining a procedure for performing local swaps on list elements. Extension to sorted lists would require the capability to insert elements at arbitrary locations in the list.

- Extension to other architectures

The ultracomputer's ability to combine memory operations that address the same memory location is intrinsic to the list access procedures. The concurrent list insertion procedure in particular is based on this feature of the ultracomputer, and we would expect the performance of the other list access procedures to degrade significantly if the execution of memory operations that are combinable on the ultracomputer is serialized. We may, nevertheless, be able to extend the list access procedures to other parallel architectures. A mapping of the algorithm onto a hypercube, such as the Connection Machine, is suggested by Hillis's proposal for implementing a butterfly structure on the Connection Machine [Hil85]. Since the nodes in the butterfly structure are processing elements, they could combine messages in the same way that memory operations are combined on the ultracomputer.

- Extension to the message based model

As we stated above, the list access algorithm we propose has a dual in which list elements in the original algorithm correspond to processes in the dual algorithm and execution of memory operations to message passing. Since algorithms that allow processes to leave and join computational structures of communicating processes without requiring global coordination may have many interesting applications, we would like to explore the dual of the list algorithm and the duals of extensions of the list algorithm to other structures.

APPENDIX A

An Implementation of P and V Operations

The insertion procedures described in section 3.1 can be used as the basis for a critical section free implementation of P and V operations on a binary semaphore which maintains FIFO queueing discipline among waiting processes and makes efficient use of space. The ultracomputer queue also supports an implementation of P and V operations which maintains FIFO order without requiring a critical section [Rud82]. But use of the ultracomputer queue requires allocation of a vector of length n for each semaphore, where n is the maximum number of processes that may perform a P operation on the semaphore. Also the ultracomputer queue based implementation may require a process executing the V operation to wait while a process executing a P operation inserts itself into the queue. The algorithm we describe requires no waiting.

We assume that there is an array of process control blocks in global memory, that each process control block or "PCB" contains a link field, and that each process knows the index into the array corresponding to its own PCB. A semaphore is an index into the PCB array and may be assigned any value from 1 to n or either of two special values, "redNil" or "greenNil." A semaphore is initially set equal to "greenNil" to indicate that there is no process in the critical section. We define the P and V operations as follows:

```
P(Sem): PCB[myId].link := redNil;
    pred := swap(Sem,myId);
    if pred <> greenNil
      then linkStatus := swap(PCB[pred].link,myId);
        if linkStatus = redNil
          then block;


V(Sem): tail := f&sOnEqual(Sem,myId,greenNil);
    if tail <> myId
      then next := f&sOnEqual(PCB[myId].link,redNil,greenNil);
        if next <> redNil
          then wakeup(next).
```

Execution of the memory instruction f&sOnEqual(V,a,b) is equivalent to atomic execution of the following sequence of instructions:

```
temp := V;
if V = a
  then V = b;
return temp.
```

This memory instruction is not available on the NYU Ultracomputer or the IBM RP3. Multiple f&sOnEqual instructions with the same target variable are not, in general, combinable, i.e., they cannot be executed concurrently in conformity with the serialization principle. A single f&sOnEqual instruction is, however, combinable with swap instructions. When an f&sOnEqual(V,a,b) collides with a swap(V,c), the switch resolves the conflict by returning c to the process executing the f&sOnEqual and forwarding swap(V,c) if c <> a and swap(V,b) if c = a. In this application there can be at most one process at a time executing an f&sOnEqual for any given semaphore or PCB link field so the limitation on combining colliding f&sOnEqual instructions is unimportant.

The P Operation

A process executing the P operation first sets the link field in its PCB equal to redNil. The process performs this step to initialize its link field for the V operation. The process next executes a swap on the semaphore. The semaphore functions in the same way as the TAIL variable in the list insertion procedure described in section 3.1. The semaphore "points to" the tail of the semaphore queue. The value returned from the swap instruction is thus the old tail of the semaphore queue. If it is greenNil, the process is at the head of the queue and can enter the critical section. If it is not greenNil, then the process enqueues itself by setting the link field in its predecessor's PCB to point to its own PCB, where the predecessor is the old value for Sem returned by the swap instruction on the semaphore. The process writes its index into its predecessor's PCB using a swap instruction instead of an ordinary store because it must examine the value returned by the swap, the "linkStatus". If the linkStatus is redNil, the process blocks. We expect the linkStatus to be redNil, but it may, due to an unusual interleaving, be the special value "greenNil". A greenNil signals the processor next in line that it can enter the critical section. The signal is left by a process executing the V operation when it knows that another process is in line for the critical

section but does not know the identity of the process.

The V Operation

A process executing the V operation first executes an f&sOnEqual(Sem,myId,greenNil) instruction. Recall that the semaphore functions as a pointer to the tail of the semaphore queue. If the process executing the V operation finds that it is itself at the tail of the queue, i.e., that Sem = myId, then there is no process waiting to enter the critical section. It removes itself from the queue and indicates that the queue is empty by setting the semaphore equal to greenNil. The process can exit the V operation. Since no process is waiting, there is no process to wake up. But if the value returned from the f&sOnEqual does not equal myId, the process must wake up the next process in line. Since a process performing the P operation writes the index of its PCB in the link field of its predecessor's PCB before blocking, a process executing a V operation can identify its successor by reading the link field in its PCB, unless the successor has not yet written to the link field. The process reads its link field using the instruction f&sOnEqual(PCB[myId].link,redNil, greenNil). If the successor has written its index in the link field, then the f&sOnEqual does not change the value of the link field --- the instruction merely returns the index of the next process in line for the critical section. If the value returned from the f&sOnEqual is not redNil, it is the index of the process that is next in line. It should now be clear why a process sets its link field to redNil as the first step in executing the P operation. If the link field is not initially set to redNil, then it cannot be determined whether it is garbage or the index of the next process in line. If the link field is not redNil, then the process executing the V wakes the process indicated by the returned value and exits the V operation. A value of redNil in the link field indicates that the next process in line has executed the swap instruction in the P operation but has not yet written its index into the link field. If the value of the link field is redNil, execution of the f&sOnEqual instruction sets the link equal to greenNil and returns the value rednil. The process can exit the V operation without waking any process even though it knows another process is in line because it has left a signal for that process that it should not block. As we have seen, if a process executing the P operation receives a value of greenNil back from its swap on its predecessor's link field it enters the critical section instead of blocking.

Note that this solution is incorrect if the block and wake up instructions are implemented so that a process can block even if it has just received a wakeup signal. If a process receives a wake up signal while it is awake, the signal should function as a credit capable of offsetting one block instruction, as described in [HwB84].

The implementation of the P and V operations for a binary semaphore we have described maintains FIFO order among waiting processes without requiring a critical section or any waiting for execution of the P and V operations themselves. The solution requires space proportional to s + n, where s is the number of semaphores and n is the number of processes in the system. The ultracomputer queue described in [Rud82] also provides a critical section free implementation of P and V operations which maintains FIFO ordering but requires space proportional to s * n and can require waiting.

APPENDIX B

The Concurrent List Access Procedures

In the following procedures, status vectors are represented as bit vectors. For example, the status of a valid element which is a beginning anchor is represented as (1100). An "X" is used instead of a hyphen to denote masked positions.

First Stage Algorithm

```
    var H,T: eptr,local;

Procedure Initialize;
   var e: eptr,local;
begin
newGlobal(e);
store(e^.f,nil);
store(e^.status,1100);
store(H,e);
markCacheable(H);
store(T,e);
end;

Procedure Insert(e:in eptr);
-- The value of e^.f is assumed to be nil.
   var pred: eptr,local;
begin
store(e^.status,1100);
pred := swap(T,e);
store(e^.b,pred);
store(pred^.f,e);
store(pred^.status,X0XX);
end;

Procedure Delete(c:in eptr);
begin
store(c^.status,0XXX);
end;

Procedure Search(target:in dType);
   var c: eptr,local;
begin
c := fetch(H^.f);
while c <> nil do
  if fetch(c^.data) = target
    then if match(swap(c^.status,0XXX),1XXX)
```

```
            then exit;
    c := fetch(c^.f);
  end while.
  end;

  Procedure Collect;
     var c: eptr,local;
  begin
  c := fetch(H^.f);
  while c <> nil do
    if match(fetch(c^.status),0000)
      then attemptExcise(c);
    c := fetch(c^.f);
  end while.
  end;

  Procedure AttemptExcise(c:in out eptr);
     var bAnchor: eptr,local;
  begin
  if match(f&sOnMatch(c^.status,0000,XX1X),0000)
    then bAnchor := fetch(c^.b);
        if match(f&sOnMatch(bAnchor^.status,X00X,X1XX),X00X)
          then excise(c,bAnchor)
          else store(c^.status,XX0X);
  end;

  Procedure Excise(c:in out eptr;bAnchor:in eptr);
     var eStatus:statusType local;
        dptr:eptr,local;
  begin
  dptr := c;
  repeat
    c := fetch(c^.f);
    eStatus := swap(c^.status,XXX1);
    if match(eStatus,00XX)
      then eStatus := swap(c^.status,X1XX);
  until not match(eStatus,00XX);
  store(c^.b,bAnchor);
  store(bAnchor^.f,c);
  store(c^.status,XXX0);
  store(bAnchor^.status,X0XX);
  appendlist(dptr);
  end.
```

Second Stage Algorithm

```
  Procedures
    Initialize, Insert, Delete, Search, Collect, and Excise:
    see First Stage Algorithm

  Procedure AttemptExcise(c:in out eptr);
```

```
   var bAnchor: eptr,local
      eStatus: statusType,local;
begin
eStatus := swap(c^.status,XX1X);
if match(eStatus,0000)
  then bAnchor := fetch(c^.b);
      if match(swap(bAnchor^.status,XX1X),XX0X)
        then if match(swap(bAnchor^.status,X10X),X0XX)
             then excise(c,bAnchor);
                  exit;
if match(eStatus,XX0X)
  then store(c^.status,XX0X).
end.
```

Third Stage Algorithm

```
Procedure
 Delete:
 see First Stage Algorithm
Procedure
 AttemptExcise:
 see Second Stage Algorithm

Procedure Initialize;
   var e: eptr,local;
begin
newGlobal(e);
store(e^.f,nil);
store(e^.status,1100);
store(e^.intransit[0]) := 0;
store(e^.intransit[1]) := 0;
store(H,e);
markCacheable(H);
store(T,e);
end;

Procedure Insert(e:in eptr);
-- The value of e^.f is assumed to be nil.
   var pred: eptr,local;
begin
store(e^.census,0);
store(e^.intransit[0],0);
store(e^.intransit[1],0);
store(e^.status,1100);
pred := swap(T,e);
store(e^.b,pred);
store(pred^.f,e);
store(pred^.status,X0XX);
end;

Procedure Search(target:in dType);
```

```
   var s: switchType,local;
      p,c: eptr,local;
begin
{ s := fetch(H^.switch);
inc(H^.intransit[s]) };
p := H;
c := fetch(H^.f);
while c <> nil do
  inc(c^.census);
  dec(p^.intransit[s]);
  if fetch(c^.data) = target
    then if match(swap(c^.status,0XXX),1XXX)
          then dec(c^.census);
               exit;
  { s := fetch(c^.switch);
  inc(c^.intransit[s]) };
  dec(c^.census);
  p := c;
  c := fetch(c^.f);
  end while;
dec(p^.intransit[s]);
end;

Procedure Collect;
   var s: switchType,local;
      p,c: eptr,local;
begin
{ s := fetch(H^.switch);
inc(H^.intransit[s]) };
p := H;
c := fetch(H^.f);
while c <> nil do
  inc(c^.census);
  dec(p^.intransit[s]);
  if match(fetch(c^.status),0000)
    then attemptExcise(c);
  { s := fetch(c^.switch);
  inc(c^.intransit[s]) };
  dec(c^.census);
  p := c;
  c := fetch(c^.f);
  end while;
dec(p^.intransit[s]);
end;

Procedure Excise(c:in out eptr;bAnchor:in eptr);
   var eStatus: statusType,local;
      dptr,p,e: eptr,local;
      s: switchType,local;
begin
dptr := c;
repeat
 c := fetch(c^.f);
 eStatus := swap(c^.status,XXX1);
```

```
  if match(eStatus,00XX)
    then eStatus := swap(c^.status,X1XX);
until not match(eStatus,00XX);
store(c^.b,bAnchor);
store(bAnchor^.f,c);
s := f&Complement(bAnchor^.switch);
inc(c^.census);
store(c^.status,XXX0);
repeat
until fetch(bAnchor^.intransit[s]) = 0
store(bAnchor^.status,X0XX);
dec(dptr^.census);
e := dptr;
while e <> c do
 repeat
 until fetch(e^.census) = 0;
 repeat
 until fetch(e^.intransit[0]) = 0;
 repeat
 until fetch(e^.intransit[1]) = 0;
 p := e;
 e := fetch(e^.f);
 end while;
store(p^.f,nil);
disposelist(dptr);
end;
```

Fourth Stage Algorithm

```
Procedure
 Delete:
 see First Stage Algorithm
Procedure
 AttemptExcise:
 see Second Stage Algorithm
Procedures
 Initialize, Insert, and Excise:
 see Third Stage Algorithm

Procedure Search(target:in dType);
   var s1,s2: switchType,local;
       p,c: eptr,local;
begin
s1 := fetch(H^.switch);
inc(H^.intransit[s1]);
s2 := fetch(H^.switch);
if s1 <> s2
  then inc(H^.intransit[s2]);
p := H;
c := fetch(H^.f);
while c <> nil do
```

```
  inc(c^.census);
  dec(p^.intransit[s1]);
  if s1 <> s2
    the dec(p^.intransit[s2]);
  if fetch(c^.data) = target
    then if match(swap(c^.status,0XXX),1XXX)
          then dec(c^.census);
             exit;
  s1 := fetch(c^.switch);
  inc(c^.intransit[s1]);
  s2 := fetch(c^.switch);
  if s1 <> s2
    then inc(c^.intransit[s2]);
  dec(c^.census);
  p := c;
  c := fetch(c^.f);
  end while;
dec(p^.intransit[s1]);
if s1 <> s2
  then dec(p^.intransit[s2]);
end;

Procedure Collect;
   var s1,s2: switchType,local;
      p,c: eptr,local;
begin
s1 := fetch(H^.switch);
inc(H^.intransit[s1]);
s2 := fetch(H^.switch);
if s1 <> s2
  then inc(H^.intransit[s2]);
p := H;
c := fetch(H^.f);
while c <> nil do
  inc(c^.census);
  dec(p^.intransit[s1]);
  if s1 <> s2
    then dec(p^.intransit[s1]);
  if match(fetch(c^.status),0000)
    then attemptExcise(c);
  s1 := fetch(c^.switch);
  inc(c^.intransit[s1]);
  s2 := fetch(c^.switch);
  if s1 <> s2
    then inc(c^.intransit[s2]);
  dec(c^.census);
  p := c;
  c := fetch(c^.f);
  end while;
dec(p^.intransit[s1]);
if s1 <> s2
  then dec(p^.intransit[s2]);
end;
```

Fifth Stage Algorithm

```
Procedure
  AttemptExcise:
  see Second Stage Algorithm
Procedures
  Initialize,
  Insert, and Excise:
  see Third Stage Algorithm

Procedure Delete(c:in eptr);
begin
inc(c^.census);
if match(swap(c^.status,0XXX),1000)
  then attemptExcise(c);
dec(c^.census);
end;

Procedure Search(target:in dType);
   var s1,s2: switchType local;
      eStatus: statusType local;
      c,p,temp: eptr local;
begin
s1 := fetch(H^.switch);
inc(H^.intransit[s1]);
s2 := fetch(H^.switch);
if s1 <> s2
  then inc(H^.intransit[s2]);
p := H;
c := fetch(H^.f);
while c <> nil do
  inc(c^.census);
  dec(p^.intransit[s1]);
  if s1 <> s2
    then dec(p^.intransit[s2]);
  repeat
    temp := c;
    eStatus := fetch(c^.status);
    if match(eStatus,1XXX)
      then if fetch(c^.data) = target
           then eStatus := swap(c^.status,0XXX);
              if match(eStatus,1XXX)
                then if match(eStatus,1000)
                       then attemptExcise(c);
                     dec(c^.census);
                     exit;
      else if match(eStatus,0000)
             then attemptExcise(c);
  until (c = temp) or (c = H);
  if c = H
    then dec(c^.census);
       exit;
  s1 := fetch(c^.switch);
```

```
    inc(c^.intransit[s1]);
    s2 := fetch(c^.switch);
    if s1 <> s2
      then inc(c^.intransit[s2]);
    dec(c^.census);
    p := c;
    c := fetch(c^.f);
    end while;
  dec(p^.intransit[s1]);
  if s1 <> s2
    then dec(p^.intransit[s2]).
```

APPENDIX C

Simulation Results

We simulated the list access algorithm we describe in chapter 3 and several alternative algorithms for accessing a shared list. We had two goals in designing the simulation. First, we wanted to buttress the informal correctness argument for our list access algorithm empirically by watching for errors over many simulation runs. Second, we wanted to explore the performance of our list access algorithm to discover how the level of concurrency affects its performance and to compare its performance with that of alternative algorithms. No errors in the list access procedures were discovered from the simulation. This appendix summarizes what we learned from the simulation about the performance of the list access algorithms we simulated. We emphasize that the results we report here are tentative.

6. Description of the simulation

We simulated variations of three concurrent algorithms for accessing a list:

ParList the concurrent list access algorithm described in chapter 3;

BiLock a concurrent list access algorithm based on Kung and Lehman's algorithm for the manipulation of binary search trees [KuL80]; and

LockList a list access algorithm that allows concurrent searches but requires that deleting and inserting processes gain exclusive control of the list.

We simulated three versions of ParList: the stage 5 version with search from the head; the stage 5 version with search from the tail; and the stage 2 version with search from the tail. We simulated the stage 2 version (batched garbage collection) in order to compare BiLock with a comparable version of ParList. Kung and Lehman's algorithm provides for batched garbage collection only, so we were unable to compare BiLock directly with a stage 5 version of ParList. (As noted in chapter 3, the version of Kung and Lehman's that uses individual garbage collection contains an error.) Two versions of BiLock were simulated. The first version, BiLock1, is the closer of the two versions to the algorithm described in [KuL80]. It differs from the original algorithm described by Kung and Lehman principally in that it uses the inser-

tion algorithm we have described to achieve concurrent insertion and the combinable memory operations provided by the ultracomputer to avoid serializing accesses to the same global variable. BiLock2 is a further modification in which a process invalidates the element it intends to excise before attempting to acquire any of the locks necessary for an excision. This modification decreases the potential for contention at locks. Locklist was simulated to provide an essentially sequential algorithm against which to compare the concurrent algorithms. We simulated both a search from the head and a search from the tail version of LockList.

We wanted to discover from the simulation how the performance of ParList, BiLock, and LockList varied with the level of concurrency. The parameters to the simulation include the following:

the number of processes accessing the list;
the number of distinct data values;
the variability in the time required for a global access;
the proportion of inserts and searches in the job mix;
seeds for the random number generators; and
the number of list accesses to be completed by the simulation.

We also experimented with nonuniform distribution of data values, slow and fast classes of processes, and variations in the amount of time a process executes between list accesses.

The simulation was designed to measure performance in terms of the number of references to global memory required for a list access. This statistic seems to be a reasonable statistic to focus on as a single best indicator of performance because we would expect that local processing would make a relatively insignificant contribution to the time required for list access operations and that accessing global memory would account for almost all the time required for a list access. The simulation result on which we focused was, therefore, the average number of references to global memory made by a generic list access. (The simulation produced many additional statistics, including the average success rate for a search, the average and maximum list lengths, and the number of global references spent waiting.)

We simulated the algorithms on a uniprocessor using an event-driven simulation. An event in the simulation corresponds to a single reference by a process to global memory and possibly some local computation by the same process. Simulating an event includes scheduling the next global reference for the process. We did not simulate the operation of the network. Events scheduled to occur at the same time

were executed serially in an order determined by the order in which the events were added to the simulation calendar. A simulation in which a global reference was assumed to take one unit of time therefore corresponded to round-robin execution of the processes with one global reference per burst.

7. Summary of results

The results which we report are all based on simulations in which we varied the level of concurrency and held the other parameters constant. We have also experimented with changing these other parameters. We believe (though more systematic experiments should be done) that changes in these parameters do not affect the validity of the observations we make below about the response of the algorithms' performance to changes in the level of concurrency.

In the experiments on which we report, we assumed the following values for the simulation parameters:

Number of data values: 4
Units of simulated time per global reference: 1
Proportion of list accesses that are inserts: 48%
Time spent between list accesses: 0

The four data values were randomly assigned according to a uniform distribution as the value for new elements and as the target for searching processes. We assumed that every global reference took the same amount of time, that the probability that a list access was a search or an insert was almost equal, and that processes existed only to access the list.

Figures 1 and 2 summarize the results of our experiment. The figures show for each algorithm the number of global references required for a generic list access as a function of the number of processes accessing the list. Figure 1 shows the performance of the search from the head and the search from the tail versions of both ParList (the stage 5 algorithm) and LockList. Figure 2 shows the performance of BiLock1, BiLock2, and the comparable version (stage 2) of ParList. Only search from the tail versions of these algorithms were simulated. Note that the performance graphs in figures 1 and 2 are incomparable. The algorithms shown in figure 1 collect garbage on an individual basis. Those shown in figure 2 use batch garbage collection and so the cost of garbage collection is not counted in computing the time for a

list access.

Each data point shown in figures 1 and 2 represents a single simulation run. We made the runs lengthy (20,000 list accesses) to capture the long-term rather that the startup performance of the algorithm, but only one run was conducted at each level of concurrency per algorithm. (We did choose a few data points for which we set up multiple runs. The results of this informal experiment indicate that figures 1 and 2 would not change significantly if we set up more runs per data point.)

We were frequently surprised by the simulation. We make the following observations about the simulation results:

(1) Locklist and ParList behave very similarly when searches start at the head of the list.
(2) Parlist achieves almost constant time access when searches start at the tail.
(3) BiLock1 performs worse than the comparable version of ParList.
(4) BiLock2 performs better than the comparable version of ParList by a small constant.

Since access to the list in LockList is blocked whenever a process is modifying the list by inserting or deleting an element, we were not surprised that LockList failed to achieve constant time list access, but we had expected that ParList would perform better than LockList, i.e., that the number of global references required per list access would increase at a slower rate than for LockList as the number of processes accessing the list increased. We explain this behavior by noting that the factor that dominated the time for a list access was the time spent by searching processes looking for a valid element that matches their target, and that when searches start from the head, all the searching processes tend to be searching along the same short (three or four element long) sublist. Since we would expect the time required for a search to increase with the number of competitors the searcher has for a valid matching element, and since the number of competitors is a direct function of the number of processes accessing the list (given that searching processes search the same sublist), the time for a list access can be expected to increase with the level of concurrency.

We predicted (after a period of consternation over the previous result) that if searches started at the tail instead of at the head of the list, ParList's behavior would improve, i.e., the number of global references per list access would be a constant or slowly growing function of the number of processes access-

ing the list. We expected that searching processes would tend to be spread through the list by the concurrently occurring insertions, instead of being bunched together by concurrently occurring excisions, as happened when searches started from the head of the list. If searches start at the insertion site, searchers would initially tend to see new list elements, elements not seen by other searchers with the same target, because new elements are being added at the place searches begin. If searches start at the head, searches would begin at a well-traveled section of the list and tend initially to visit elements that are already invalid. Furthermore, new searchers would join the pack of old searchers because excisions will have removed the initial section of the list visited by the old searchers, in effect creating a "shortcut" for the new searchers. We further expected that the number of competitors each searcher would have in the part of the list it searches, and thus the average number of elements visited in a search, would tend to be a constant function of the number of processes accessing the list, and that the performance of the algorithm would not degrade as the level of concurrency increased. The simulation confirmed this expectation. When searches start from the tail, the average number of elements visited per search is a very slowly growing function of the number of processes accessing the list. A display version of both the search from the head and the search from the tail helped confirm that searchers clump together when they start at the head and that they tend to spread through the list when they start at the tail of the list.

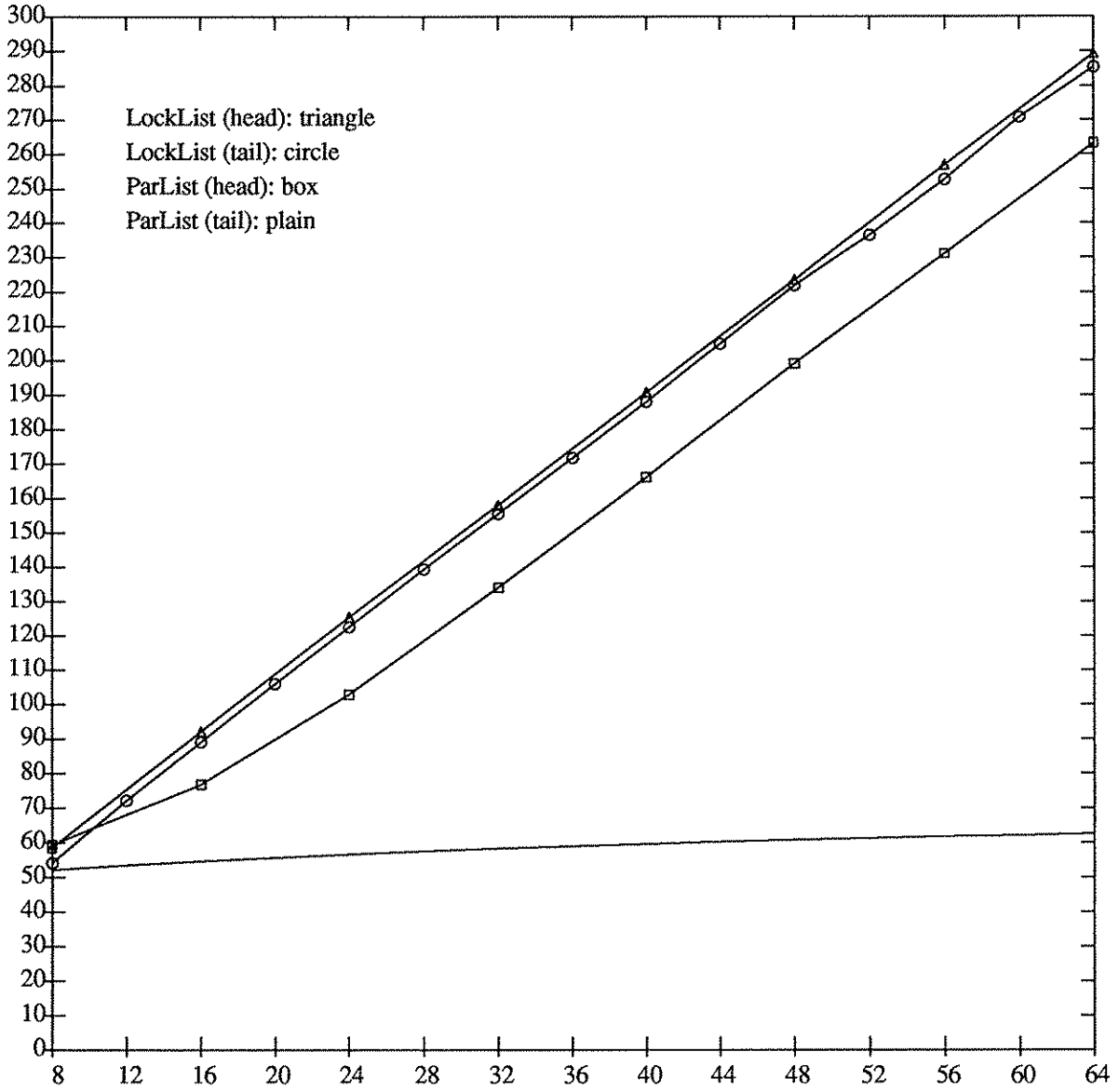
BiLock1's performance, as shown in figure 2, degrades as the level of concurrency increases because the algorithm requires that an excising process obtain a lock on the target element and then on that element's predecessor. Many different processes may attempt to excise the same element (though only one will succeed). Since the number of processes that may attempt to excise a given element is a function of the number of processes accessing the list and the wait to obtain locks is a function of the number of processes that are attempting to excise an element, it is not surprising that BiLock1's performance degrades with increases in the level of concurrency. Note, however, that the scales on the y axis in figures 1 and 2 are different --- BiLock1's performance does not degrade as significantly as that of LockList or the head search version of ParList.

Finally, we note that BiLock2 achieves almost constant time access to the shared list in spite of the presence of critical sections (locks that must be acquired) in the procedure for excising an element. We

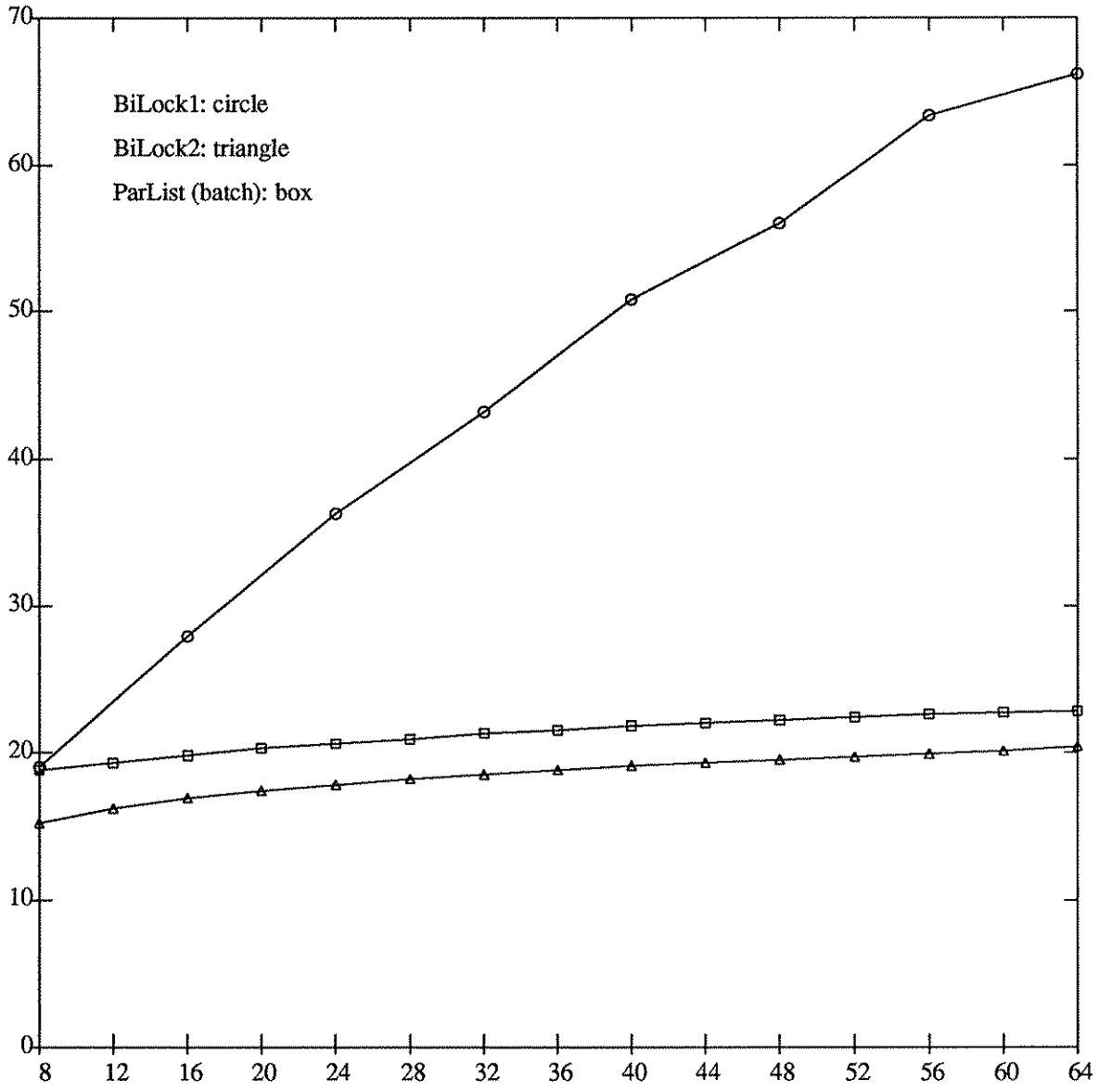
speculate that BiLock2 achieves this performance because at most two processes can contend for any given lock.

8. Evaluation of the simulation

As we note in chapter 5, further work is required before we can state anything but tentative results. We need to explore more systematically the effect of changes in parameters other than the number of processes accessing the list and to increase the number of experiments each data point represents. We also need to modify the simulation so that it simulates the operation of the network. The current simulation does not have the power to model queueing delays in the network.



Performance of LockList and ParList:
Global references per list access as a function of the number of processes
Figure 1



Performance of BiLock and ParList
Global references per list access as a function of the number of processes
Figure 2

REFERENCES

References

[BaS77] R. Bayer and M. Schkolnick, Concurrency of Operations on B-trees, *Acta Informatica 9*, 1 (1977), 1-21.

[Ben84] M. Ben-Ari, Algorithms for On-the-Fly Garbage Collection, *ACM Transactions on Programming Languages and Systems 6*, 3 (July, 1984), 333-344.

[BiF85] A. Biliris and M. B. Feldman, Concurrent Insertions in Multiway Dynamic Structures, *Proceedings of the 19th Annual Conf. on Info. Sciences and Systems* , Baltimore, MD, March, 1985, 289-294.

[ChM86] M. Chandy and J. Misra, An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection, *ACM Trans. on Programming Languages and Systems 8*, 3 (July 1986), 326-343.

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Trans. on Programming Languages and Systems 8*, 2 (April 1986), .

[DLM78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten and E. F. M. Steffens, On-the-Fly Garbage Collection: An Exercise in Cooperation, *Communications of the ACM 21*, 11 (November 1978), 966-975.

[EGL85] J. Edler, A. Gottlieb and J. Lipkis, Operating System Considerations for Large-Scale MIMD Machines, Ultracomputer Note #92, Courant Institute, New York City, New York, December 1985.

[Ell80a] C. Ellis, Concurrent Search and Insertion in 2-3 Trees, *Acta Infomatica 14*, (1980), 63-86.

[Ell80b] C. Ellis, Concurrent Search and Insertion in AVL Trees, *IEEE Trans. on Computers 29*, 9 (September 1980), 811-817.

[FrT84] M. L. Fredman and R. E. Tarjan, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, *Annual Symposium on Foundations of Computer Science 25*, (October, 1984), 338-346.

[GLR83] A. Gottlieb, B. D. Lubachevsky and L. Rudolph, Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors, *ACM Transactions on Programming Languages and Systems 5*, 2 (April 1983), 164-189.

[GGK83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer --- Designing an MIMD Shared Memory Parallel Computer, *IEEE Transactions on Computers 32*, 2 (February 1983), 175-189.

[HeH85] J. H. Hester and D. S. Hirschberg, Self-Organizing Linear Search, *Computing Surveys 17*, 3 (September 1985), 295-312.

[Hil85] W. D. Hillis, *The Connection Machine*, The MIT Press, 1985.

[HwB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, 1984.

[Jon85] D. W. Jones, *Concurrent Operations on Priority Queues*, (manuscript), April, 1985.

[KrR85] C. P. Kruskal and L. Rudolph, A General Look at the Fetch-and-Add, in *The Architecture of Parallel Computers*, C. P. Kruskal (ed.), (manuscript), 1985.

[KuL80] H. T. Kung and P. L. Lehman, Concurrent Manipulation of Binary Search Trees, *ACM Transactions on Database Systems 5*, 3 (September 1980), 354-382.

[KwW82] Y. S. Kwong and D. Wood, A New Method of Concurrency in B-Trees, *IEEE Trans. on Software Eng. 8*, (May 1982), 211-222.

[Lam76] L. Lamport, Garbage Collection with Multiple Processes: An Exercise in Parallelism, *Proceedings of the 1976 International Conference on Parallel Processing* , Detroit, Michigan, 1976, 50-54.

[Lam77] L. Lamport, Concurrent Reading and Writing, *Communications of the ACM 20*, 11 (November 1977), 806-811.

[LaN79] H. C. Lauer and R. M. Needham, On the Duality of Operating System Structures, *Operating Systems Review 13*, 2 (April 1979), 3-19.

[LeY81] P. L. Lehman and S. B. Yao, Efficient Locking for Concurrent Operations oon B-Trees, *ACM Trans. on Database Systems 6*, 4 (December 1981), 650-670.

[Man84] U. Manber, Concurrent Maintenance of Binary Search Trees, *IEEE Trans. on Software Eng. 10*, 6 (November 1984), 777-784.

[MaL84] U. Manber and R. E. Ladner, Concurrency Control In a Dynamic Search Structure, *ACM Trans. on Database Systems 9*, 3 (September, 1984), 439-455.

[Man86] U. Manber, On Maintaining Dynamic Information in a Concurrent Environment, *SIAM J. Comput. 14*, 4 (November, 1986), 1130-1142.

[Pet83] G. L. Peterson, Concurrent Reading While Writing, *ACM Transactions on Programming Languages and Systems 5*, 1 (January 1983), 46-55.

[PfN85] G. F. Pfister and V. A. Norton, *Proceedings of the 1985 Conference on Parallel Processing* , St. Charles, IL, August 20-23, 1985, 790-797.

[QuY84] M. J. Quinn and Y. B. Yoo, Data Structures for the Efficient Solution of Graph Theoretic Problems on Tightly-Coupled MIMD Computers, *Proceedings of the 1984 International Conference on Parallel Processing* , Columbus, Ohio, 1984, 431-438.

[RSL78] D. Rosenkrantz, R. Stearns and P. Lewis, System-level Concurrency Control for Distributed Data Bases, *ACM Transaction on Database Systems 3*, 2 (1978), 178-98.

[Rud82] L. Rudolph, *Software Structures for Ultraparallel Computing*, PhD Thesis, New York University, February 1982.

[Sam76] B. Samadi, B-Trees in a System with Multiple Users, *Inf. Process. Letters 5*, 4 (October 1976), 107-112.

[Sch80] J. T. Schwartz, Transactions on Programming Languages and Systems, *Communications of the ACM 2*, 4 (October 1980), 484-521.

[Ste75] G. L. Steele, Multiprocessing Compactifying Garbage Collection, *Communications of the ACM 18*, 9 (September 1975), 495-508.

[Sto84] H. S. Stone, Database Applications of the FETCH-AND-ADD Instruction, *IEEE Transactions on Computers 33*, 7 (July 1984), 604-612.

[Tho86] R. H. Thomas, Behavior of the Butterfly Parallel Processor in the Presence of Memory Hot Spots, *Proceedings of the 1986 International Conference on Parallel Processing* , Columbus, Ohio, 1986, 46-50.

[Wad76] P. L. Wadler, Analysis of an Algorithm for Real Time Garbage Collection, *Communications of the ACM 19*, 9 (September 1976), 491-500.

[Wyl79] J. C. Wyllie, *The Complexity of Parallel Computations*, Ph.D. Thesis, Cornell University, Ithaca, New York, 1979.

[Yoo83] Y. B. Yoo, *Parallel Processing for Some Network Optimization Problems*, Ph.D. Thesis, Washington State University, Pullman, Washington, 1983.