ON MAINTAINING DYNAMIC INFORMATION IN A CONCURRENT ENVIRONMENT

by

Udi Manber

Computer Sciences Technical Report #535

February 1984

ON MAINTAINING DYNAMIC INFORMATION IN A CONCURRENT ENVIRONMENT

(Preliminary Version)

Udi Manber*

Department of Computer Science
University of Wisconsin
Madison, WI. 53706.

1. Introduction

This paper studies the amount of cooperation required for independent asynchronous processes to share a simple dynamic data structure. We define an abstract data structure, called a *concurrent pool*, consisting of a multiset with the operations *add* and *remove*. Concurrent pools are used for task sharing, resource management, garbage collection, interprocess communication, and more.

We present a scheme for designing efficient concurrent algorithms for the operations listed above. The efficiency is measured mainly by the number of non-local operations that a process may have to make (we also have to make sure that the local computation is efficient). In general, non-local operations may involve writing into a shared variable, locking, or waiting for a message, hence they introduce interference (or required cooperation). Our goal is to derive upper and lower bounds on the interference in the worst case.

The algorithms consist of two parts. The first part is an efficient sequential algorithm for another, more complicated, data structure, which includes a split operation. The second part is a concurrent algorithm to locate elements distributed among the processes. This is done by having (active) processes search throughout the system leaving marks indicating where they have already searched so that other processes will minimize their search. The search is based on a tree traversal, although the actual concurrent algorithm is more complicated due to the fact that elements are changed dynamically.

*This research was supported in part by the National Science Foundation under Grant MCS83-03134.

We also present a model of concurrent computation, based on a shared memory model, and prove lower bounds on the amount of interference. The gap between the upper bounds achieved by our algorithm and the lower bounds is small (a factor of $O(k^{\varepsilon})$, where k is the number of processes and $\varepsilon > 0$).

2. The problem

We want to develop an efficient data structure to represent *multisets*. We make no assumptions on the type of elements in the multisets. The data structure should support the following operations:

Add(M,x) : add the element x to the multiset M. Whether or not x is already in M has no effect on this operation.

Remove(M,y) : if M is not empty then choose an arbitrary element of M (any element will do), delete it from M, and assign it to y.

Data structures that support the operations listed above will be called *pools*. Pools can be used to store pointers to all waiting (independent) tasks in a multicomputer system. When a process (or processor) becomes available it removes a task from the pool; during its execution it may generate tasks, which are added to the pool. Pools can also represent available resources in a system, available blocks of memory, independent events in a simulation, etc.

Pools can be very efficiently implemented in a sequential environment by stacks or queues, in which case every operation takes constant number of steps. (Stacks and queues are even more powerful than what we need since we are satisfied with arbitrary order of removals and we do not require LIFO (or FIFO) ordering.) If there are only two processes we can use a double-ended queue and let each process add and remove from a different end. Each operation still takes constant time; moreover, the only time a process may interfere with the other one is when the queue is almost empty. In this case interference is unavoidable since the processes must compete for the same elements. Double-ended queues were in fact used in that way in the design of a parallel garbage collection algorithm with two processes, a list process, and a garbage collector [4]. This algorithm was generalized to many processes in [8]. A shared pool was used but there was no discussion on efficient implementation. Obviously only one process can add or remove from one end of a queue at any given time, so a better data structure is required in order to achieve a high level of concurrency. We assume that, although at some times the

multiset may be small or even empty, in general the number of elements involved is much greater than the number of processes. When the number of elements is small interference is unavoidable.

The processes perform add and remove operations in an arbitrary order and frequency. It is possible, for example, that only one process adds while all the others remove, or that all processes only add.

3. The model

A shared memory model, similar to the one described in [5], is used throughout the paper. Some comments on applications to distributed computation appear in section 6. The main reason we use a shared memory model is because of its simplicity. It allows us to gain some insight to the required cooperation of asynchronous processes that may be harder to get from a more complicated model. Lower bounds for this model should apply to other models with more expensive means of communication.

We give here a brief description of the model. We assume a random access memory shared by many autonomous asynchronous reliable processes. A process is a state machine with its own local memory. In one atomic step a process can either read a variable of the shared memory, write into the shared memory, or lock part of the shared memory and change its own state (a more general operation, test and set, is allowed in [5]). Several processes can read the same variable at the same time; however, if a process writes into a shared variable then no other process can access that variable at that time. Hence some kind of locking is required. We use the regular notion of read and write locks (see for example [1,2]). A write lock gives exclusive access to its holder while a read lock only prevent writers from accessing the variable. We make no assumptions about the implementation of the locks. If a process is denied access to a variable that is locked by another process it will have to wait. We call such occurrences *collisions*. Our main goal in designing the algorithm is to minimize the number of collisions.

We believe that the definition of collision captures the simplest form of interference (or cooperation depending on the point of view) among asynchronous processes. This definition is also primitive enough to enable us to prove lower bounds. We do not consider in this paper issues of fairness or starvation. We think of the processes as being servers rather than customers.

We make no assumptions on the order in which the processes access the shared memory. Thus, it is possible that some processes are much "faster" than others or that a process "goes to sleep" for a period of time and "wakes up" later. We do not deal with issues of fault tolerance in this paper, although the algorithms can be made robust rather easily using standard techniques since processes depend very little on each other.

In order to prove lower bounds on the add and remove operations we have to define them precisely. We assume that for each element there is a unique variable that changes its value when the element is removed or added. Removing or adding an element may involve changing more than just one variable; however, the element is "formally" removed only when the unique associated variable has been changed. This assumption is required to rule out the possibility of two processes removing the same element concurrently.

4. The algorithm

The algorithm was designed to work best when all the processes have approximately the same behavior (i.e. it is designed for the average case). Surprisingly, it turns out that the algorithm is not far from optimal in the worst case as well (see section 5).

The basic idea of the algorithm is as follows. The multiset is initially partitioned among the processes. Every process maintains a segment of the multiset and performs the operations add and remove locally as long as possible. When a local segment becomes empty and the process wishes to remove it searches for a non-empty segment and performs the removal from it. Since removing an element from another process' segment may involve collisions we have to minimize such occurrences. For example, if few processes are adding and most processes only remove then the solution above is almost reduced to the sequential solution. We improve this simple solution in the following way. Once a process finds a non-empty segment and interrupts the "owner", it tries to take more than just one element at a time. Since the size of a local segment may vary substantially, we should not take a fixed number of elements but a fixed portion. This will improve the algorithm provided we can find a fast sequential algorithm to split a segment to two parts with sizes that differ by no more than a constant factor.

The algorithm consists of two separate parts. The first part is an efficient data structure and constant time sequential algorithms for adding, removing and splitting to two (approximately equal) segments. The second part is a scheme for finding a non-empty "donor".

4.1. Splitting a local segment

We use balanced binary trees where each node corresponds to an element of the multiset. Nodes are added level by level from left to right and are removed in the opposite order. One way of achieving constant time per add and remove is to maintain a pointer to the last inserted node at the bottom of the tree, and at each node pointers to its left and right "brothers", father, and two children. The father and children pointers are then updated as new nodes are added or removed. It is straightforward to perform add and remove in constant time. It seems, however, that in order to split the tree we need to update O(log n) pointers. We can reduce the O(log n) complexity to O(1) using the observation that except for the bottom level all the levels are full. In order to find out whether a node is at either the left or the right end of a level while we are adding or removing, it is sufficient to know what level (depth) it is. When we are adding (removing) and the level is full (empty) we start a new level (go higher up using the father pointers), keeping track of the level number and the number of nodes in that level. To split we need only to decrement the level number of the lowest level. This automatically makes the two children of the root two new roots. There are two ways to take care of the old root. We can either insert it at the bottom or maintain two types of trees, one with an additional root, call it a 2-roots tree, and the regular 1-root tree; splitting a 1-root tree forms a 1-root and a 2-roots trees, and splitting a 2-roots tree forms two 2-roots trees. In any case, we need to keep a pointer to the root. It is easy to see

that in the worst case we split to a third and two thirds.

We also have to find the middle of the lowest level (or next to lowest level in case it is less than half full); removals and additions from the other half should start at that middle. We solve this problem by using arrays instead of a linked list representation for each level (see Figure 1). Since the size of each level is fixed the arrays will never have to be extended. Using arrays in this manner saves the "brother" pointers and one of the children pointers (if one child is known then the other one can easily be found). The children pointers are needed in order to find the two new roots after a split and the father pointers are needed to move up the tree when a level becomes empty.

Overall, we have an interesting data structure consisting of a linked list of variable sized arrays organized as a tree, and we have shown that it can be expanded and split, all in constant time. Obviously, a process that performs split, add, or remove must lock the tree with a write lock. One drawback for using arrays in this way is that it makes memory allocation more complex.

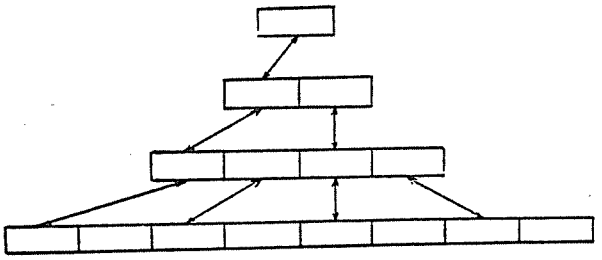

Figure 1: The data structure

4.2. Finding a non-empty segment

The problem of finding a non-empty segment is an interesting problem of locating resources in a dynamic concurrent environment. We want a solution that minimizes the amount of non-local access. In particular, processes should not have to update any kind of a global table while they perform local operations.

We first solve a very restrictive case of the problem and then show how to extend the solution to the general case. We assume that initially there is exactly one element in the local segment of each process. Furthermore, no additions are made; each process performs only removals. Let each process be associated with a leaf in a complete binary tree (for simplicity we assume that the number of processes is a power of 2). Each leaf initially contains the element associated with the process. Each node has a flag which signals that all the elements in the leaves in its subtree were removed, in which case we call the node *empty*. Each process starts the removals from its own leaf and traverses the tree according to the procedure described below (an example follows the procedure).

```
Procedure Traverse (node) ;
{ initially node is the process' own leaf }

if node is a non-empty leaf then
        remove the corresponding element ;
        mark node as empty ;
{ at this point node must be empty }
if node = root then terminate
 else if the father of node is empty then
        Traverse (father of node)
  else if node's brother is empty then
                mark the father as empty ;
                Traverse (father of node)
   else
```

{ half of the subtree is empty while the other half is not yet empty ; in this case the process goes directly to a leaf that is in the same relative position in the brother's subtree as the initial node is in the original subtree; we call this leaf the *matching descendant*. The idea of this step is to distribute all processes that may arrive together at the same internal node among the leaves. Since the tree is static such a step can be implemented in constant time using the binary representation of nodes in a balanced binary tree }

```
        Traverse (matching descendant of initial node)
;
```

Example 1: The tree is given in Figure 2. First consider the case of having only one process, P, initially at h, active. P traverses the tree in the following order:

h,d,i,d,b,e,j,e,k,e,b,a,c,l,f,m,f,c,g,n,g,o,g,c,a.

Let's start now with 3 proceeses: P at h, Q at i, and R at m, and assume that in each time unit one process completes the computation on one node. The following is an example of an order of execution. Each process is followed by the node it visits such that a bold node means that the node is marked as empty in this step.

P:**h**, P:d, Q:**i**, P:i, P:**d**, P:b, Q:d, Q:b, P:e, P:**j**, P:e, P:**k**, P:**e**, Q:e, Q:**b**, Q:a, P:**b**, P:a, P:c, R:**m**, R:f, R:**l**, R:**f**, R:c, P:l, P:**f**, P:c, R:g, R:**o**, R:g, R:**n**, R:**g**, P:g, P:**c**, Q:c, Q:**a**, P:a, R:c, R:a.

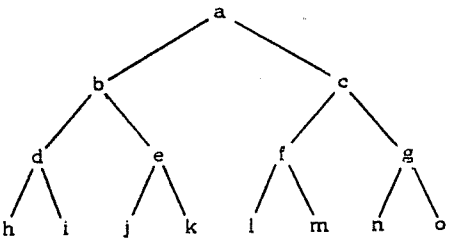

Figure 2: Example 1

To analyze the algorithm we count not only collisions but actual steps. Let S(k) denote the maximal number of steps taken by k processes traversing a tree with k leaves. The analysis of the algorithm relies on the following lemma.

Lemma 1: For any internal node v all processes that visit leaves in both subtrees of v visit the left subtree first, or they all visit the right subtree first.

Proof: Assume the contrary. Let P be a process that started in the left subtree of v and then visited the right subtree and let Q be a process that visited the subtrees in the opposite order. In order to visit both subtrees P and Q must have checked first if the root is empty. Let's assume that Q checked the root after P or at the same time as P. In this case, at the time Q checked the root P has already found the left subtree to be empty. Since Q has to check the left child before it goes down the left subtree it will find it empty and will not go down. □

Theorem 1: The total number of steps taken by k processes to remove k elements using procedure Traverse is $O(k^{1.59})$ in the worst case.

Proof: We use a recursive argument. Let T be a tree rooted at r, and assume, without loss of generality, that the first process that arrives at r started at the left subtree. By lemma 1 no process that started at the right subtree will visit the left subtree. Hence, the worst case of traversing T occurs when all the processes in the left subtree traverse it in the worst possible way, then they are dispersed to the bottom of the right subtree and then all processes traverse the right subtree together in the worst possible way. It is easy to see that we get the following recurrence relation for S(k):

$$S(2k) \leq 3S(k) + O(k),$$

which implies that

$$S(k) = O(k^{\log_2 3}) = O(k^{1.59}).$$ □

It is possible to improve the asymptotic behavior of the algorithm by using multiway trees and distributing the processes down from an internal node in an optimal way. One can achieve a complexity of $O(k^{1+\varepsilon})$ for any $\varepsilon > 0$. However, while this leads to an asymptotic improvement it is not a practical solution since for any realistic value of k, $\varepsilon \geq 0.5$. We describe the modifications below only for the purpose of comparing them to the lower bounds in section 5. Consider a 4-way tree. Using a similar traversal procedure and similar arguments as above one can show that the worst case occurs when the processes in one subtree traverse it in the worst possible way, then they move down to a second subtree and, with the processes in this subtree, traverse it in the worst possible way, then they all move to the third and then to the fourth subtree. This leads to the recurrence relation $S(4k) \leq (1+2+3+4)S(k) + O(k)$, which gives a slightly worse solution. However, if, instead of "sending" the processes from the first and second subtrees together to the third subtree, we "spread" them and send each to the remaining subtrees (the third and the fourth) we get the recurrence relation $S(4k) \leq (1+2+2+4)S(k) + O(k)$, which is slightly better (actually it gives the same bound as the binary case). In general, given an m-ary tree, it is not hard to show that we get the recurrence relation $S(mk) \leq O(m \log m)S(k) + O(k)$, which leads to $S(k) = O(k^{1+\varepsilon})$ where $\varepsilon = O(\log \log m / \log m)$.

So far we discussed a restrictive case. In the general case we divide the algorithm into *rounds*. We make sure that in each round all segments are visited by at least one "foreign" process seeking a donor. Since each visit splits the segment by at least a third we can have at most O(log n) rounds before all n elements are deleted. Notice that it is possible that the segment of one process contains all the elements in the multiset. Hence all segments must be searched and a complete round is indeed necessary.

We modify the algorithm by replacing the flag in each node with a round counter. An internal node marked with a counter m implies that all of its leaves "donated" in round m. Processes run the traversal algorithm, starting with their own leaf, only when their local segments are empty and they are looking for a donor. Once they find a donor they continue their local operations until they empty their segments again, in which case they continue the traversal from the last node they visited. Round m is terminated when the root is marked m. When a process reaches the root it starts the traversal again from its leaf incrementing its local round number. If a Process finds a node with a later round number it simply updates its own counter.

Theorem 2: Let the multiset contain n elements that are divided among the processes in an arbitrary way. The total number of collisions resulting from k processes executing n removals and arbitrarily many additions is O(S(k) log n) in the worst case.

Proof: Assume first that no additions are performed. Every local segment is visited in a round, thus it is split by some process. It is not necessarily true that in a round of removals the total number of elements is cut by a constant. However, if at the beginning of a round the largest segment contains t elements then at the end of the round the largest segment will contain at most 2/3t elements. Hence, in the worst case O(log n) rounds are required to remove n elements. By Theorem 1 we get an upper bound of O(S(k) log n) on the number of steps in this case.

Additions are always performed locally; hence, the only time they can be involved in a collision is when another process (or processes) attempts to split the local segment. However, since we counted steps rather than collisions in the traversal algorithm, a split operation always contributes to S(k) whether it collides with the local process or not. As a result, performing arbitrarily many additions cannot cause an additional collision that was unaccounted for. □

If more than n removals are performed then obviously the worst case occurs when all the elements are removed and then repeatedly one element is added and all processes attempt to remove it. The algorithm is not very efficient for the case of multisets with very few elements. (The best solution in this case is probably to decrease the concurrency and let only few processes be active, assuming again that the processes are servers and that fairness is not an issue; in any case, this is beyond the scope of this paper.)

5. Lower bounds

In this section we describe the lower bound results on the number of collisions. Since the add operation in the algorithm described in the previous section does not cause collisions we ignore it for the lower bounds. We consider the following scenario: There are n elements in the multiset, and k processes, $P_1, P_2, \ldots, P_k$ are trying to remove all the elements. We do not count the time the processes spend locally per operation; we count only collisions. We use an adversary argument in the following way. Given a data structure and algorithms, we will produce a schedule for the processes that leads to many collisions. Let $x_{i1}, x_{i2}, \ldots, x_{in}$ be the sequence of elements P_i removes provided that all the other processes are not active.

Lemma 2: There exist i and j, $1 \le i,j \le k$, and r and s, $1 \le r,s \le \lfloor n/k \rfloor+1$, such that $x_{ir} = x_{js}$.

Proof: Using a simple pigeon hole argument: There are more than n elements in the k subsequences of size $\lfloor n/k \rfloor+1$, hence two of them must be equal. □

Theorem 3: Every algorithm for concurrent removal of n elements by k processes produces $\Omega(k \log n)$ collisions in the worst case.

Proof: Find i, j, r, and s that satisfy the conditions of lemma 2. Let P_i remove all the elements up to x_{ir} and then let P_j start removing its elements. If P_j does not change its order of removals as a result of the removals of P_i then we can cause a collision by letting them both attempt to remove x_{ir} (= x_{js}) at the same time. Otherwise, in order to know that P_i has been active, P_j must read a variable that P_i has written. Let w be the first such common variable. The execution sequences of P_i and P_j until they access w are independent. Hence we can let them arrive at w at the same time thus producing a collision. In any case, a collision occurs in the worst case before a portion of $O(1/k)$ of the n elements are removed. After the collision we "allow" all the processes to learn about it and change their removal orders accordingly (this probably causes more collisions but we do not count them). The number of remaining elements is now $n' = n(1-O(1/k))$. We now use the same argument repeatedly for n', n'' and so on. Overall, $\Omega(k)$ iterations are required to cut the number of elements by a constant factor (using the fact that $1/e \simeq (1-1/k)^k$). Hence, $\Omega(k \log n)$ iterations are required to remove all the elements and the theorem follows. □

In particular, theorem 3 implies that if k is a small constant then the algorithm in section 4 is optimal; for large k there is a gap of a factor of k^2 between the lower and upper bounds.

6. Extensions and further research

Each local tree in our algorithm acts as a stack. The multiset as a whole does not follow a LIFO ordering since a local tree can be split and be assigned to a slow process. We have generalized the algorithm so that each local tree acts as a queue, making it more adaptable to some applications.

It is possible to support processes that are allocated and destroyed dynamically. We need to be able to insert and delete leaves from the global tree concurrently with the other operations. Efficient concurrent algorithms for insertions and deletions in external binary search trees are described in [7]; general concurrent binary search trees are discussed in [3,6]. These algorithms can be easily adapted to this application. If a dynamic tree is used then it may not be possible to find the matching descendant of a node (see algorithm Traverse in section 4.2) in constant time. However, the search for matching descendants need not interfere with the removals or additions operations; it only interferes with inserting or deleting a process.

We are also working on adapting the algorithm to distributed computation where processes reside at remote sites. The algorithm described in section 4 was designed so that most actions take place locally. The only exceptions are accessing the global tree to find a non-empty segment (which we minimize), and splitting a local tree. The splitting can be done locally; then, assuming each site has several processors, either we keep both parts in the same site or shipped one part to the requesting site with minimal interference with the processors doing other local computation. All the lower bounds still hold; in this case they also apply to the amount of communication.

Several open questions remain.

First, can one improve the tree traversal algorithm? Is it possible to adapt the algorithm to a message passing model in an efficient way?

We defined collisions as being either read-write or write-write conflicts. If we count only write-write conflicts, is it possible to improve the upper bounds or (more likely in our opinion) prove the same lower bounds?

We have not considered probabilistic algorithms in this paper. There are several examples where probabilistic algorithms are proven to be better than deterministic algorithms, especially in parallel computation [9,10]. At first, it seems that partitioning the elements into several stacks and letting each process choose at random which stack to remove from may lead to a better algorithm. However, a simple analysis shows that this is not the case and that choosing elements simply at random leads in general to an inferior solution. This does not rule out, of course, the possibility of improving the upper bounds with a more sophisticated probabilistic algorithm.

7. Conclusions

We presented in this paper a design of an abstract concurrent data structure. We proved upper and lower bounds on the amount of interference among competing processes under a shared memory model. Abstract data structures have proven to be very helpful in the design of sequential algorithms. Having an arsenal of data structures and efficient implementations makes it easier to formalize problems and to solve them. We hope that the same will be true for concurrent computation.

References

[1] C. Ellis "Concurrent Search and Insertion in AVL Trees", *IEEE Transactions on Computers*, Volume C-29, September 1980, pp. 811-817.

[2] J.N. Gray "Notes on Database Operating Systems", in *Lecture Notes in Computer Science*, Volume 60, 1978, pp. 393-481.

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

[4] H.T. Kung and S.W. Song. "An Efficient Parallel Garbage Collection System and its Proof of Correctness", In *18th Annual Symposium on Foundation of Computer Science*, October 1977, pp. 120-131.

[5] N. A. Lynch and M. J. Fischer, "On Describing the Behavior and Implementation of Distributed Systems", *Theoretical Computer Science*, Volume 13, 1981, pp. 17-43.

[6] U. Manber and R.E. Ladner, "Concurrency Control in a Dynamic Search Structure", *ACM Symposium on Principles of Database Systems*, Los Angeles, March 1982, pp. 268-282.

[7] U. Manber, "Concurrent Maintenance of Binary Search Trees". To Appear in *IEEE Transactions on Software Engineering*.

[8] I.A. Newman and M.C. Woodward, "Alternative Approaches to Multiprocessor Garbage Collection", In *1982 International Conference on Parallel Processing*, Bellaire, Michigan, August 1982, pp. 205-210.

[9] M. O. Rabin, "N-Process Mutual Exclusion with Bounded Waiting by 4 $\log_2$N-Valued Shared Variable, *Journal of Computer and System Sciences*, August 1982, pp. 66-75.

[10] M. O. Rabin, "The Choice Coordination Problem", *Acta Informatica*, 17(1982), pp. 121-134.