1989

Evaluation of Concurrent Pools

David Kotz
Dartmouth College

Carla Ellis
Duke University

Evaluation of Concurrent Pools

David Kotz Carla Schlatter Ellis
Department of Computer Science
Duke University
Durham, NC 27706

Abstract

The assignment of resources or tasks to processors in a distributed or parallel system needs to be done in a fashion that helps to balance the load and scales to large configurations. In an architectural model that distinguishes between local and remote data access, it is important to base these allocation functions on a mechanism that preserves locality and avoids high-latency remote references. This paper explores performance considerations affecting the design of such a mechanism, the Concurrent Pools data structure. We evaluate the effectiveness of three different implementations of concurrent pools under a variety of stressful workloads. Our experiments expose several interesting effects with strong implications for practical concurrent pool algorithms.

1 Introduction

One of the important problems to be solved in a parallel or distributed programming system is the assignment of resources or tasks of a computation to processor nodes. Often, the order and location of task execution or of the use of resources may not affect the overall solution. On the other hand, it does matter that the allocation of these elements be done in a dynamic and decentralized fashion, to balance the load among those processors and allow the allocation strategy to scale to larger configurations. A *mechanism* is needed for distributing elements to processors (or, in general, to processes) that keeps the amount of inter-process interference to a minimum.

Such a mechanism is particularly essential for an architectural model that distinguishes between local and remote memory access. Concrete examples of such architectures include non-uniform memory access (NUMA) shared-memory MIMD multiprocessors (e.g., the BBN Butterfly [1] and the IBM RP3 [7]), distributed memory multiprocessors (e.g., hypercube-based MIMD machines), and distributed systems based on local-area networks (LANs). In these systems, there are the additional requirements that the assignment mechanism should respect locality and avoid high-latency remote references or communication. This also becomes an issue whenever there is some preferred (although not strictly required) assignment of elements to processors.

This paper explores performance considerations affecting the design of such a mechanism based on an abstract data type known as *concurrent pools*. A pool is a collection of items, which grows and shrinks with the demands of the processes. A process may add an element to the pool or request an element from the pool at any time; the exact element removed from the pool is chosen arbitrarily (i.e., no ordering is enforced). A concurrent pool attempts to spread the elements out over the processors so that accesses are less likely to interfere with each other. The basic idea of the concurrent pool is to allow most operations to be done within the local components of the distributed data structure. Only when a request can not be satisfied locally does it become necessary to access remotely stored components.

Our focus is on evaluating effectiveness of several *implementations* of the concurrent pool concept. The requirements outlined above suggest that the data structure used to describe the available elements should be implemented as a distributed data structure with components local to requesting sites. We consider three algorithms that differ in the strategies used to locate remote elements when there are no elements available locally. Experiments performed on a BBN Butterfly multiprocessor under a variety of workloads show that the three implementations perform similarly well for light workloads, but that with stressful workloads it appears that a simple algorithm may provide better performance than a complex algorithm, designed to keep remote accesses to a minimum. In addition, we have found that implementations can benefit by taking into account information on the nature of the operations performed by each process to help balance the elements among processes that need them.

In the next section, we describe the concurrent pool and three concrete algorithms to implement it. Then in Section 3 we outline the design and analysis of the experiments we performed. This is followed by a discussion of the results of those experiments in Section 4. Finally, we present some conclusions in Section 5.

This research was supported in part by Burroughs Corporation.

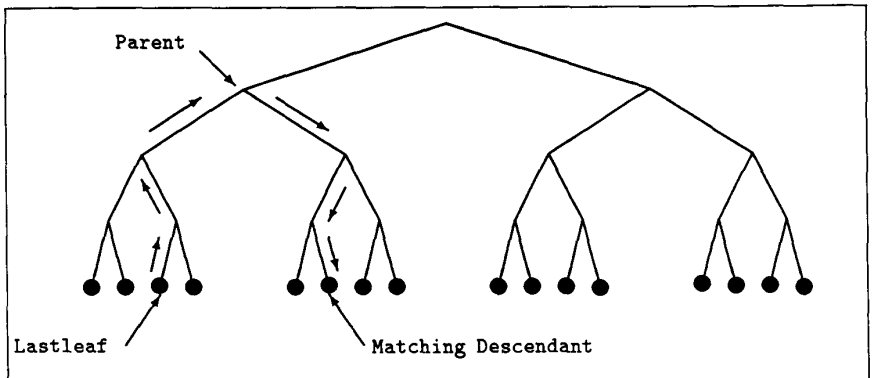

Figure 1: An example of the matching descendant for the node **LastLeaf** around a given ancestor **Parent**. This pool has 16 segments.

2 The Abstract Data Structure

The concurrent pool, described by Manber[5], partitions the elements of the pool into *segments*, one per processor. Each process may then add and remove elements within its own local segment ideally without interference from the remote processes. When it wishes to remove an element and its local segment is empty, it need to look elsewhere. The process then looks at the segments of other processors to find some elements that it may *steal*[†]. When it finds a non-empty segment it steals roughly half of the elements for its own segment and proceeds as before, unless there is only one element in the remote segment, in which case that element is taken immediately. By stealing half of the elements found at the non-empty segment rather than just enough to satisfy the immediate need, the searching process is trying to balance the available reserves and prevent its next request from also having to perform a search.

Thus there are two parts to the algorithm: one that defines the local segment manipulations and one that defines the segments to be examined when searching for elements to steal. The local segment manipulations may be done in many ways, depending on the semantics of the elements; Manber describes a method for arbitrary elements that requires constant time (i.e., $O(1)$) to add an element to a segment, to remove an element from a segment, or to split a segment.

Given a workload that generates a sufficiently high frequency of steals, the search algorithm becomes the dominant factor in the performance of the pool as a whole. It is during the rare but lengthy searches that processes interfere with one another and require the use of (presumably) slower non-local operations. The search strategy imposes some form of global organization upon the distributed segments, either implicitly or explicitly (e.g., a superimposed data structure). In this paper we will consider three search algorithms, one described by Manber and two simple algorithms we have designed for comparison.

[†]Although not addressed in this study, the problem of an add operation encountering a full segment (if there is a limit imposed) could be handled in a symmetric fashion, adding remotely to a segment with sufficient capacity.

2.1 The Tree Search Algorithm

Manber's search algorithm attempts to keep non-local references and the number of potential collisions between processes to a minimum. To accomplish this, a binary tree is superimposed on the segments, with each segment occupying a leaf of the tree. For convenience, we assume that the tree is full so that the number of leaves is a power of two. Embedded in the tree is information that helps the processes avoid subtrees that have recently been found to be devoid of elements (i.e., none of the leaves in that subtree have any elements). One complete traversal of the tree, in which each leaf is examined at least once, is called a *round*. Every process has an idea of the current round number in a counter, and each subtree (including leaves) has a counter indicating that it has been traversed completely and found to be empty in all rounds up to and including that round. When a process decides that a subtree is empty, it marks that subtree with its own round counter. If that subtree is the whole tree, the process increments its round counter and starts again at its leaf. Otherwise, by comparing its round counter with that of the subtree's sibling, a process can determine whether it should

1. descend into the sibling subtree to look for elements. It does this when the sibling's counter is less than its own, since the sibling subtree has not been marked *empty* as recently as the current subtree. In this case, it jumps directly to a leaf. As specified in [5], this leaf is the *matching descendant*, the leaf in the sibling subtree that is symmetrically in the same position as the last leaf visited in the subtree (see Figure 1).

2. move further up the tree, when the sibling's counter is equal to its own, since the sibling subtree has been marked *empty* as recently as the current one.

3. decide that it is behind, when the sibling's counter is greater than its own, update its own round counter, and start the new round again at its own leaf. In this case the sibling was marked *empty* before the current subtree was, and the process should re-examine its own subtree.

The round counters of the various subtrees must be accessed with locks protecting them so the examination and modification of the counters is done atomically. This is one source of inter-process interference in the tree search algorithm. Another source is the locking at the leaves where several processes may be waiting to perform an add, remove, or split operation.

The tree search algorithm is given below. Each process maintains three internal global variables: its current round counter (**MyRound**), the leaf containing the local segment (**MyLeaf**), and the most-recently-visited leaf node (**LastLeaf**). When a process issues a request on a processor that has run out of elements in its segment, it calls **TreeSearch(LastLeaf,** *nil*) (except the first time, when it calls **TreeSearch(MyLeaf,** *nil*)). Note that **MyRound** is initially 1 for all processes, and the round counter in each tree node is initially zero.

```
procedure TreeSearch(node, child)
    if node is a leaf then
        LastLeaf ← node;
        if node is non-empty then
            split half of node into MyLeaf;
            return one element from MyLeaf;
        endif
        TreeSearch(parent of node, node);
    else
        if either child's round counter is
           greater than MyRound then
            /* case 3 */
            MyRound ← higher round counter value;
            TreeSearch(MyLeaf, nil);
        else
            set round counter of child to MyRound;
            if other child's round counter is
               the same as MyRound then
                /* case 2, but there is no parent */
                if node = root then
                    increment MyRound;
                    TreeSearch(MyLeaf, nil);
                else /* case 2 */
                    TreeSearch(parent of node, node);
                endif
            else /* case 1 */
                TreeSearch(Match(LastLeaf), nil);
            endif
        endif
    endif
end TreeSearch.
```

2.2 The Linear Algorithm

Another possible search algorithm, which is much simpler than the tree algorithm, is a *linear* search. The linear algorithm starts looking at the segment where it last found elements, and travels from one segment to the next segment, as if they were arranged in a ring, until it finds a non-empty segment to split.

A call to **LinearSearch(MyLeaf)** begins the first search, and later searches begin with the segment where elements were last stolen (**LinearSearch(LastFound)**).

```
procedure LinearSearch(segment)
    while segment is empty
        segment ← the next segment;
    end
    split off half of segment into my segment;
    LastFound ← segment;
    return an element from my segment;
end LinearSearch.
```

2.3 The Random Algorithm

Another simple algorithm chooses segments at random until it finds a non-empty segment to split.

```
procedure RandomSearch
    loop
        segment ← a random segment;
    while segment is empty;
    split half of segment into my segment;
    return one of the elements from my segment;
end RandomSearch.
```

3 Design of the Experiments

3.1 Parallel Processing Environment

The experiments have been performed on a Butterfly^{TM} Multiprocessor manufactured by Bolt Beranek and Newman[1]. The Butterfly is an MIMD machine in which all memory is physically local to a processor but accessible by all processors. There are, therefore, two levels of memory, from an individual processor's point of view: local and remote, with accesses to remote memory about 4 times slower than accesses to local memory[3]. Since the penalty for remote accesses on the Butterfly is not as great as in some architectures for which concurrent pools have been advocated, the cost of non-local operations is adjustable by a parameter in our experiments to allow us to emphasize the effects of the non-local operations. We have experimented with 16-processor pools on our 32-node Butterfly, with one segment and one process on each processor. Unfortunately, since a few of the 32 nodes are devoted to system tasks, a 32-segment pool cannot be properly simulated.

3.2 Search Algorithm Implementation

There are a number of issues in the implementation of the search algorithms. It is fairly easy to see (in all three algorithms) that a process may search for a long time, examining every segment possibly several times, before it finds any elements. This occurs when the pool is empty and elements

are being inserted relatively infrequently. If segments at all processors become empty and every process begins to look around the pool, livelock occurs. Since the pool is empty and none of them will add an element while looking for one, none of the processes will ever find an element. This is a difficulty that must be solved in any implementation of the algorithms. For simplicity, our implementations keep a shared count of the processes looking for elements. When any process discovers that all the processes involved in the pool operations are looking (and therefore no process might be adding), it aborts its operation. Note that this solution is based on a shared memory concept, and is not a full-fledged distributed termination algorithm.

In our initial experiments, we implemented the local segment operations as described in Manber[5]. It became evident in the preliminary results that the performance of the concurrent pool was driven primarily by the number and duration of steals. Therefore, we decided to concentrate the measurements on the search algorithm. We simplified the segments, representing them as a single counter that is atomically added to, subtracted from, or split in half (since the values of the elements do not matter to the simulation, we need only store the *number* of elements in each segment). This minimizes the time involved in segment operations, allowing the search time to dominate most of the measurements and simplifying analysis of the effects of the search schemes.

3.3 Workload

The workload presented to a pool may vary. Perhaps two of the most likely patterns of access are a random series of operations with some mix of additions and removals generated by each process, and a producer/consumer arrangement, in which some of the processes only add elements and the others only remove elements. Certainly, these represent two extremes, the former balancing the operations among the processes and the latter separating them completely.

In the *random* operations model, all processes perform the same mix of additions and removals. Each process chooses its next operation randomly to fit a predetermined overall *job mix*. All job mixes from zero to 100% add operations were tested, in steps of 10%. Clearly, job mixes of 50% or higher are *sufficient*, adding more elements than are removed. Job mixes of less than 50% adds are termed *sparse*.

In the *producer/consumer* operations model, the number and arrangement of producers were fixed during the test. All numbers of producers (from no producers through all producers) were tested. This fixed assignment of each process's role as either producer or consumer throughout an experiment is a simplifying assumption. In many real systems, the identity of the processes acting as producers may change dynamically over time. This assumption, however, allows us to capture the effect of different patterns. As we shall see, the *arrangement* of producers and consumers (with respect to the search pattern) proves to be significant.

3.4 Measurement and Analysis

It is clear that there are two algorithmic components that determine the overall performance of the pools structure: the segment manipulations and the search for elements. There is possible contention at both levels, as processes lock each other out of the data structures.

The idea central to the pools structure is for processes to remain in their local segments as long as they have elements left, and to search remote segments only when necessary. When it is necessary to steal from another segment, a number of factors will determine the effectiveness of the steal: the number of segments examined before we find some elements and the amount of interference we find along the way (both affecting the *search time* directly), and the number of elements we are able to steal (affecting the length of time until the next steal).

Therefore, in addition to measuring the actual times for add and remove operations, the following measurements were taken from the simulation:

- the number of segments examined per steal
- the number of elements stolen per steal
- the percentage of *remove* operations that required a steal, in effect,
  the frequency of steal operations
- the size of each segment, over time

We began with the pool quite empty for the number of operations to be performed, forcing the processes to depend on elements added during the test. Thus, 5000 operations were performed on a pool initialized with only 320 elements.

For each workload, ten trials were performed and the measurements were averaged. In each trial, the pool was initialized and exercised under the given workload until all 5000 operations were completed. Rather than executing a fixed number of operations in each process, the processes performed operations until the combined total number of operations reached the desired amount.

3.5 Overall Impact of Assumptions

Taken together, the assumptions underlying the design of these experiments produce a stressful test of these algorithms. A continuous stream of requests are being generated by each process (as if no real computing is needed to generate new elements or use elements taken from the pool). This increases the activity in the data structure and therefore the potential for interference. The low initial fill of the segments quickly makes the job mix the prime factor in determining segment size. The simplification of segment manipulations and emphasis upon the search strategy helps distinguish among the algorithms (especially in cases where no additional penalty is artificially imposed on remote operations, which are otherwise relatively cheap on the Butterfly). However, this simplification has also eliminated some remote operations (common to all three search strategies)

such as the block transfer of stolen elements between processes.

Workloads experienced in real applications are not likely to be as stressful. Due to processing between accesses to the pool, fewer processes will be active in the pool simultaneously. The pool may tend to be more full, and the mixes more sufficient (at least in a well-tuned application). From preliminary experiments we found that all three implementations of the concurrent pool perform admirably under these conditions.

In addition, the workload may not be constant: the job mix, and perhaps the operations pattern, may change with time. It is easy to imagine an application which has an initial phase with more than sufficient adds (as the pool is filled), a stable phase, and a more sparse termination phase (as the pool is emptied). Our experiments have essentially examined these phases separately.

4 Results

4.1 Effect of Job Mix

Since steals require a significant amount of time, the performance is highly dependent on the amount of stealing involved. This is very evident in the performance of the random operations; the performance is much poorer with a sparse mix of adds and removes than when the mix is sufficient. As one would expect, no steals are performed with a sufficient mix, and, in fact, the performance generally levels off when more than 50% of the operations are adds.

In contrast, the producer/consumer model forces consumers to steal all of the elements they use, regardless of the ratio of adds and removes. Thus, steals are present at all job mixes, though most significant, of course, at sparse mixes. The performance of this model is similar to the random operations model above 50% adds, but is generally not as good at sparse job mixes. The average time for any operation, as it varies with job mix, is shown in Figure 2. Since the producer/consumer model was measured at each number of producers, the job mix was measured and the data was plotted on that scale. Using this approach, the sparse mixes of 1 to 4 producers (out of 16) all yield essentially the same mix of adds and removes (approximately 47% adds).

4.2 Balancing the producers

In the producer/consumer model, a certain fraction of the processes were producers and the remainder were consumers. The assignment of roles to processes turned out to have a significant effect on the performance for the pools structure. For example, consider the linear search algorithm.

In the linear algorithm, consumers looking to steal some elements will search the segments one by one as if they were arranged in a circle. If the producers are assigned to a contiguous portion of this cycle, then all consumers will encounter the same producer first (with the exception of some that may steal a few elements from another consumer). At this producer, the consumers will compete with each other for access to the rapidly diminishing segment. Once this segment is empty, they will all steal from the next segment. Intuitively, the consumers will remain in a tight bunch as they use the elements being produced— there is no incentive

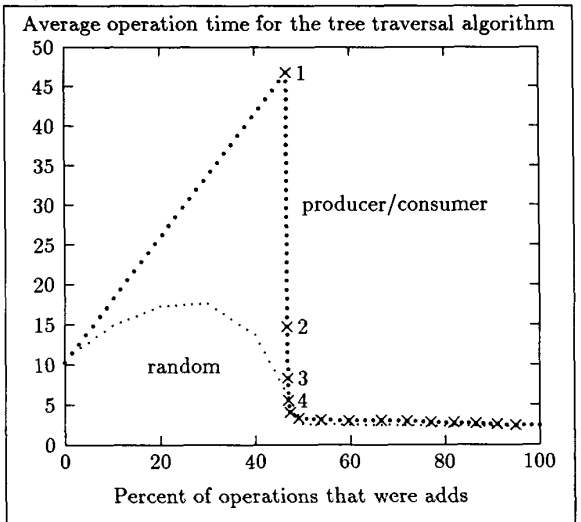

Figure 2: Average operation time (in msec) for the tree traversal algorithm, comparing the random and producer/consumer models. Some data points (×) are labeled with the number of producers.

for them to spread out to balance the load on the producers. There is increased interference between the processes as they collide at the producers' segments. The consumers will generally steal fewer elements, as successive accesses to a single segment halve the contents of that segment. This will mean the consumer will have to steal again much sooner. Thus, this *bunching* tends to significantly decrease performance. Figure 3 shows the size of each segment in a 16-segment pool over the time of a test using the linear search algorithm. Each processor recorded its segment size at strategic points in the program; these sizes were then plotted on the same time scale for comparison. A steal is obvious as a sudden drop in the size of one segment and a corresponding sudden increase in the size of another segment. The top eleven segments are those of consumers, the bottom five are segments of producers. It is clear that the producers are being stolen from in the order **0 1 2 3**, and producer **4** is never stolen from.

This effect also exists in the tree search algorithm, although the search pattern is more complicated, and information marking empty subtrees in the tree helps to steer processes away from empty producers. Figure 5 (in the same style as Figure 3) shows the segments of the pool while using the tree search algorithm; the effect is once again evident.

To correct this, the producers could be arranged in a balanced manner. The producers are arranged to be spread out as much as possible. For example, eight producers and

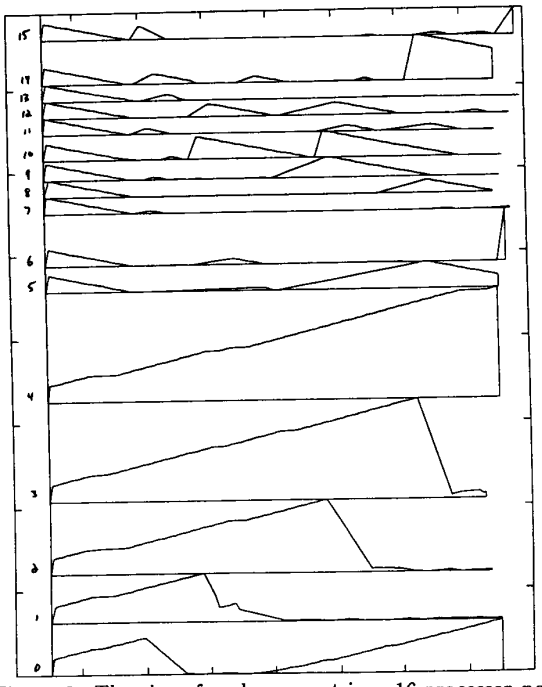

Figure 3: The size of each segment in a 16-processor pool while using the linear traversal algorithm with the producer/consumer model of operations. There are 5 producers and 11 consumers.

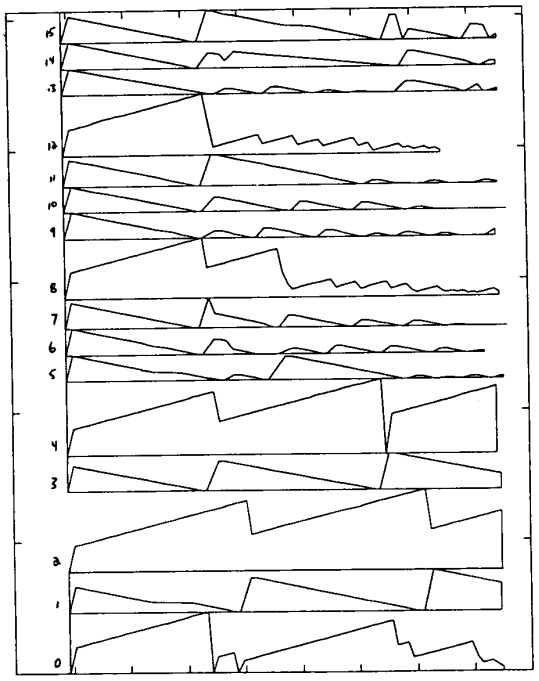

Figure 4: The size of each segment in a 16-processor pool while using the linear traversal algorithm with the 5 producers arranged in a more balanced fashion.

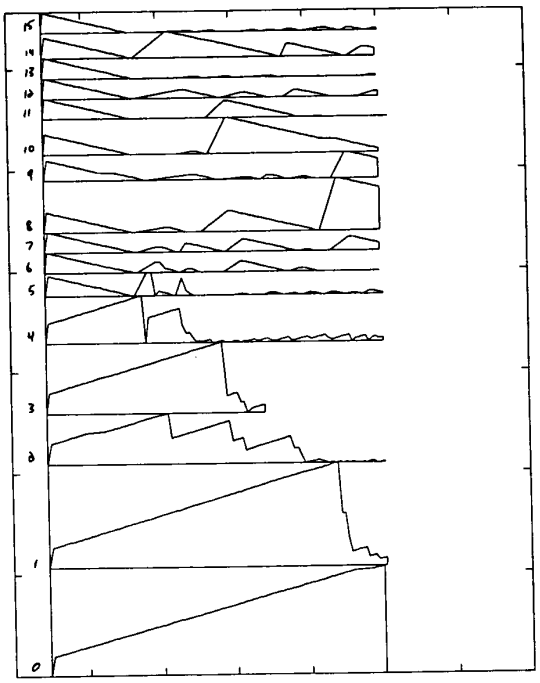

Figure 5: The size of each segment in a 16-processor pool while using the tree traversal algorithm with the producer/consumer model of operations. There are 5 producers and 11 consumers.

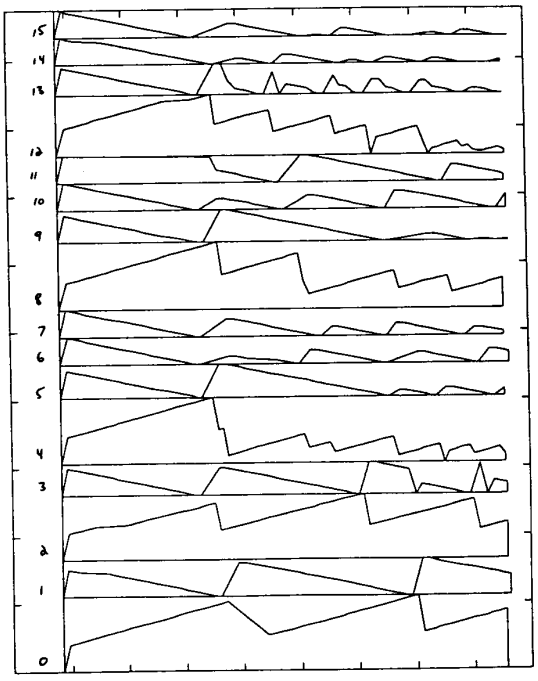

Figure 6: The size of each segment in a 16-processor pool while using the tree traversal algorithm with the 5 producers arranged in a more balanced fashion.

eight consumers would be arranged in an alternating fashion. Although this means that they may have to search a little more after depleting a segment, the reduction in interference should be worth the effort. The Figures 4 and 6 show the effectiveness of balancing the producers in the linear and tree algorithms, respectively: note that the segments of all producers (processes **0 2 4 8 12**) are accessed.

The most significant effect that balancing has on performance is in the number of elements stolen on each steal. By spreading out the producers, forcing the consumers to steal from all producers rather than one at a time, each steal is likely to find a greater number of elements. In Figure 7 the improvement due to balancing is extremely clear: this figure compares the number of elements stolen with each steal as the job mix varies from 0% adds to 100% adds.

Balancing the producers consistently lowered the average time for add operations, remove operations, and steals. These improvements are due primarily to the reduced interference at the segments, by spreading the stealers out over the producers. The frequency of steals decreased with the balancing, due to the increased number of elements stolen with each steal. There was, however, no consistent significant difference in the number segments examined for each steal; since the algorithm causes the consumer to look first where it last found elements, it will usually find elements very quickly (immediately, as it turns out, for five or more producers).

It is useful to look at the random search algorithm: since all segments are stolen from equally, one would expect no "bunching" effect. The graph of segment sizes showed no evidence of bunching, and balancing did not significantly affect the performance of the random search algorithm.

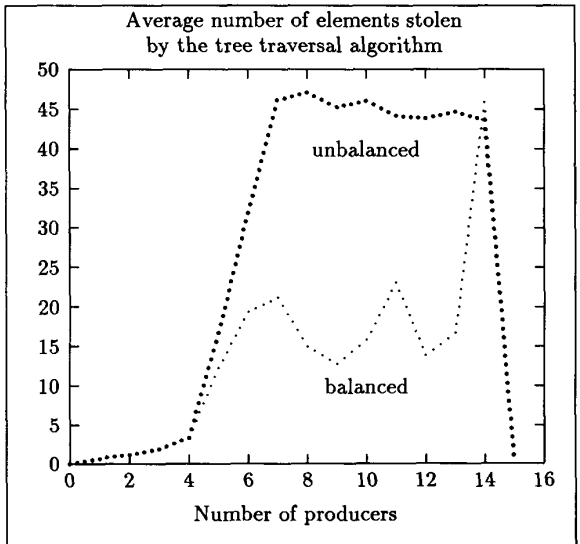

Figure 7: The average number of elements stolen for each steal by the producer/consumer model and the balanced model.

Of course, balancing the producer/consumer arrangement is a practical management policy only when the role played by a process can be determined and remains fixed (at least for a long period compared to the cost of reassignment). However, even in dynamically changing situations, this information about the impact of different arrangements can be used to understand performance variations.

4.3 Comparison of Algorithms

The tree search algorithm tends to have similar, though slightly slower, times for operations when compared with the linear and random search algorithms in the balanced producer/consumer operations pattern. It compares much less favorably, however, under the random operations pattern, when the job mix is sparse. For job mixes with more than 50% adds the three algorithms are nearly identical. This is directly related to the existence of steals in removal operations when the job mix is sparse.

The tree algorithm, however, examines many fewer segments in the course of a steal than do either the linear or random algorithms, and it also tends to steal more elements. In the Butterfly model and our implementation, the overhead of traversing the tree (and its locks) is comparable to the segment access time. One might suppose that in a different architecture, where there is a higher penalty for remote accesses, the tree search algorithm would be superior.

To simulate a higher-cost remote access architecture, delays were added to each remote operation (attempt to steal from a segment) and to each access of nodes in the superimposed tree (remember that this tree must reside somewhere, centrally or distributed; in any case it is likely to be remote for most of the processors). We tried a variety of different delays from 1 μsec per operation to 100 msec per operation (typical undelayed segment operation times are approximately 70 μsec for add operations and 110 μsec for remove operations). We found that the tree algorithm never performed better than either of the two other search algorithms; in fact, as the delay increased all three algorithms converged to very nearly identical performance graphs, both for the random operations model and the balanced producer/consumer model.

It seems, therefore, that the complexity of the tree search algorithm does not pay off in the actual performance of the pools data structure. Simpler search algorithms, such as the linear and random search algorithms, may suffice.

4.4 Using pools in an application

Perhaps the most common application of the pools data structure is the scheduling of dynamically-created tasks. Each process may be removing tasks, processing them, and producing new tasks that are put into the structure. An example of such tasks are the nodes to be expanded in a game tree. It does not matter which process expands each node, but there is no reason to share nodes with another

process until the local collection has been depleted.

In order to determine the impact of using various forms of concurrent pools in actual applications, we have adapted an existing parallel program that plays three-dimensional tic-tac-toe. This is a program using the minimax algorithm [4] for the game tree, with a central work list containing unexpanded nodes of the tree. To examine the first three moves of a 4 by 4 by 4 game requires examining 249,984 board positions. In the modified version, each position is placed in a pool when it is generated. Processors repeatedly pull a position from the pool and possibly generate new positions to put in the pool.

All three pool search algorithms performed similarly, as expected with a sufficient mix. Speedups for the application were nearly linear (14.6–15.4 with 16 processors). This suggested that the mechanism provided adequate load balancing in distributing tasks (board positions) to processors while experiencing little contention for the components of the structure. The original version that used a stack with a global lock for the work list was 40% slower and had worse speedup (only 10.7 for 16 processors).

There is additional evidence of the effectiveness of using the simple forms of concurrent pools in real applications. A paper by Finkel and Manber[2] describes an implementation on a distributed system of an application that relies heavily on a concurrent pools data structure for load balancing. They used, essentially, the linear and random search algorithms and found the performance of their applications to be quite good. The more complex tree search algorithm has apparently not been incorporated into their system [6].

5 Conclusions

All versions of concurrent pools seem to provide very good performance, in that they provide for a great deal of locality and avoid inter-process collisions. When pushed to their limit (i.e., nearly empty pools), the structure still performs admirably although slight variations in workload and access patterns can have a large effect on performance.

We tested implementations of the pools data structure with three different patterns of operations (random, producer/consumer, and balanced producer/consumer) under a full range of job mixes in order to examine the effect of the workload on the performance of the data structure. As long as the job mix remains at least sufficient (i.e., at least as many adds as removes) the performance is very good, with steals being very rare. If sparse (essentially, less than 50% adds in the random case or only a few producers), the performance depends highly on the success of steal operations.

We found that an unfortunate arrangement of producers in the pool can lead to *bunching* of the processes in the pool, causing a lot of inter-process interference and reducing performance. By rearranging the producers in a more balanced manner, the performance can be improved drastically.

Since steals are so important to the performance of the structure, the algorithm used to search the segments to find elements is also important. When the more complex tree approach was compared against two simple alternatives, a linear search and a random search, the operation times in the tree search algorithm did not compare favorably for steal-intensive workloads, even though the tree search algorithm examines fewer segments in its searches. This held even when delays were added to simulate a more loosely-coupled architecture, where remote access times tend to be higher.

The concurrent pool structure is advantageous for applications that require access to a pool of arbitrary items, particularly if they can benefit from the locality that is provided by the pool. When the workload is heavy, the implementation of concurrent pools becomes important. In this case, the tree search algorithm does not appear to be useful, since the linear or the random search algorithm may suffice and provide better performance.

In general, it is worthwhile to make efforts to preserve locality in distributed data structures; significant improvements in performance may be obtained in certain situations. On the other hand, our experiments have shown that this need not always be the case: certain architectures, data structures, or process activity patterns may not warrant the extra complexity required to achieve strong locality.

There are several possible extensions of this work. For example, how might concurrent pools be modified so that searching processors leave hints in the pool, and elements added by another processor can be directed to the searching process. How might pools be extended to handle distinguishable elements? Concurrent pools are well suited to non-uniform memory access (NUMA) machines, including distributed systems. Are there lessons to be learned from this data structure that can be applied to other concurrent data structures when used in a NUMA environment?

References

[1] BBN Advanced Computers. *Butterfly Products Overview*, 1987.

[2] Raphael Finkel and Udi Manber. DIB — A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[3] Mark Holliday. Private communication regarding his timings on the Butterfly at the Duke University Department of Computer Science, July, 1987.

[4] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1984.

[5] Udi Manber. On maintaining dynamic information in a concurrent environment. *SIAM Journal on Computing*, 15(4):1130–1142, November 1986.

[6] Udi Manber, 1987. Personal communication.

[7] G. Pfister, W. Brentley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

Errata:
Evaluation of Concurrent Pools

David Kotz Carla Schlatter Ellis
Department of Computer Science
Duke University
Durham, NC 27706

International Conference on Distributed
Computing Systems
June 1989

The labels on the curves in Figure 7, page 384, were reversed in the published paper. The corrected Figure 7 is shown below.

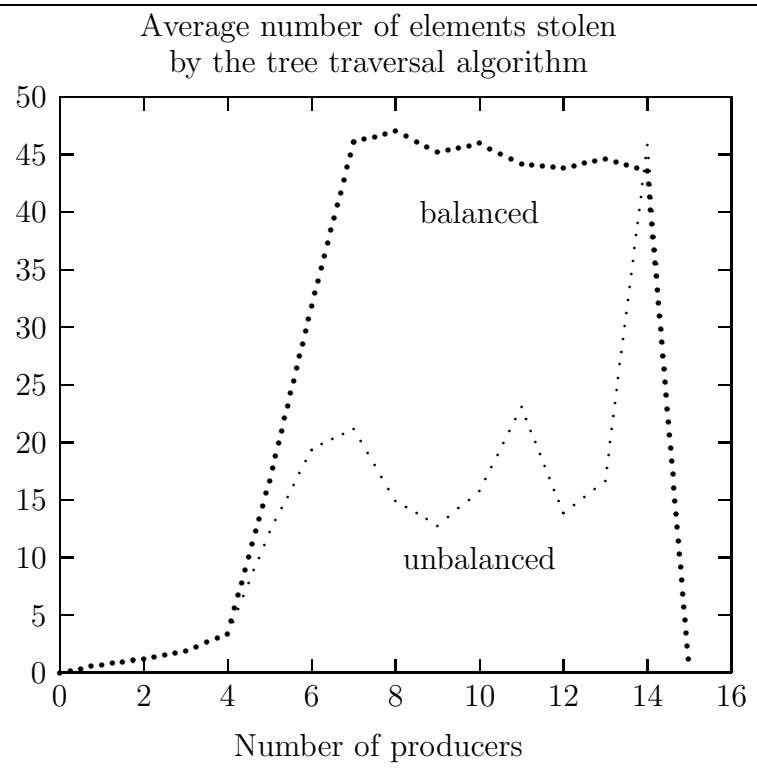

Figure 7: The average number of elements stolen for each steal by the producer/consumer model and the balanced model.