CS-1987-30

Evaluation of Concurrent Pools

David Kotz
Carla Schlatter Ellis
Department of Computer Science
Duke University

This pdf was scanned in 2020 from the original printed copy.
This TR was later revised and published at ICDCS, as follows:

David Kotz and Carla Ellis. Evaluation of Concurrent Pools. Proceedings of the International Conference on Distributed Computer Systems (ICDCS), pages 378–385. IEEE, June 1989.
doi:10.1109/ICDCS.1989.37968

Note that ICDCS paper includes a typographic error in Figure 7: the labels on the two curves should be interchanged, so that ‘balanced' labels the top curve, and ‘unbalanced' labels the bottom curve.

Evaluation of Concurrent Pools

David Kotz
Carla Schlatter Ellis

Department of Computer Science
Duke University
Durham, NC 27706

October 8, 1987

Abstract

In a parallel environment, requests for allocation and deallocation of resources or assignment of tasks should be served in a fashion that helps to balance the load and minimise the total parallel runtime. It is important to perform this allocation in a manner that preserves locality by avoiding remote references (and hence interference with other processes). Concurrent pools, as described by Manber, provide an appropriate data structure for addressing these goals. This paper evaluates the effectiveness of the pool structure under a variety of stressful workloads. It was found that the structure is highly successful at preserving the locality of pool accesses and that a simpler algorithm than that described by Manber may actually provide better performance.

1 Introduction

When subdividing a problem into tasks for processing in a parallel environment or dividing resources among many processors, it is often best to allocate the items (subproblems, resources, *etc.*) in a dynamic fashion, to balance the load among those processors. Frequently, it is not known in advance how to best allocate the items, or *elements*, and sometimes the elements themselves are dependent on the execution of the program. In particular, the order or location of the execution of the subproblems or the use of resources may not affect the overall solution. What is required, when it is not significant to the processor which element it receives, is a mechanism for distributing the elements to processors that keeps the amount of interference to a minimum.

This suggests that the data structure used to describe the available elements can profitably be implemented as a distributed data structure with components local to requesting sites. Here we assume an architectural model with some notion of local memory such as in a distributed system based on a local-area network (LAN) or various MIMD multiprocessors. In other situations, there may be some preferred (although not strictly required) assignment of elements to processors that could also be supported by this kind of distributed structure.

A *pool* is such 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.

This paper evaluates the effectiveness of the pool structure under a variety of stressful workloads. It was found that the structure is highly successful at preserving the locality of the pool accesses. Interestingly, however, a simple pool algorithm may provide better actual performance than a complex pool algorithm that encourages more locality. In the next section, we describe the concurrent pool data structure and the primary algorithms for its use. Then in Section 3 we outline the design and analysis of the experiments we performed on the pools structure. We describe two variations of pools used for comparison. This is followed by a discussion of the results of those experiments in Section 4. Finally, we present some conclusions and ideas for further research in Section 5.

2 The Pools Algorithm

The concurrent pool, described by Manber[4], partitions the elements of the pool into *segments*, one per processor. Each processor may then add and remove elements within its own local segment ideally without interference from the other processors. Only when it wishes to remove an element and its own segment is empty does it need to look elsewhere. The processor 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. 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 processors 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).

Manber's search algorithm attempts to keep non-local references and the number of potential collisions between processors to a minimum. To accomplish this, a tree is superimposed on the segments, each segment laying at a leaf of the tree. For convenience, assume that the number of leaves is a power of two so that the tree is a complete, balanced tree. Embedded in the tree is information that helps the processors 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 processor has an idea of the current round number, and each subtree (including leaves) has a counter that indicates 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, when the sibling's counter is less than its own, since the sibling subtree has not been marked as *empty* as recently as our subtree. In this case, it jumps directly to a leaf: its ***matching descendant***, the leaf in the sibling subtree that is symmetrically in the same position as the last leaf we 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 ours.

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 our subtree was, and we should re-examine our own subtree.

Thus, when a processor runs out of elements in its segment, it begins this search algorithm at the segment where it last found elements (except the first time, when it starts at its own segment). The entire search algorithm is given in Figure 2.

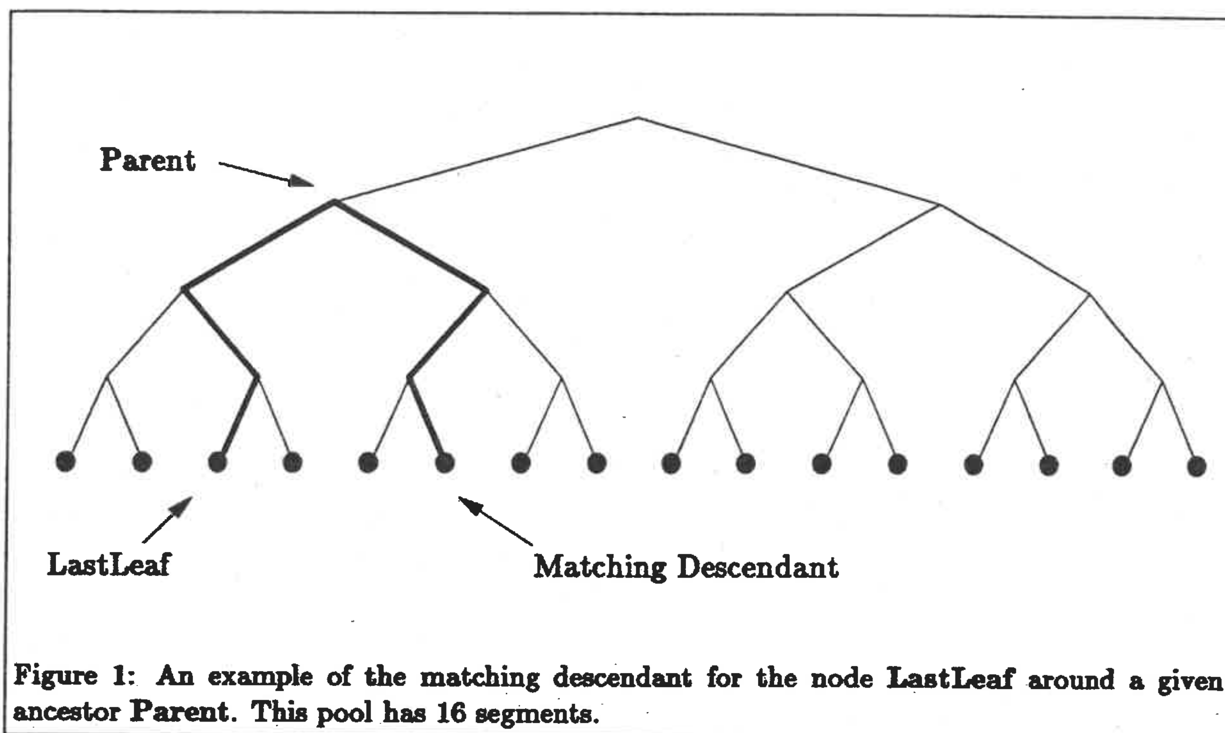

Figure 1: An example of the matching descendant for the node **LastLeaf** around a given ancestor **Parent**. This pool has 16 segments.

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.

3 Design of the Experiments

3.1 Parallel Processing Environment

The experiments were 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, the cost of non-local operations is adjustable by a parameter to allow us to emphasize the effects of the non-local operations. (The class of architectures Manber had in mind was distributed systems based on LANs.) We experimented with 16-processor pools on our 32-node Butterfly.

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

end TreeSearch.


MyRound is initially 1 for all processes.
All node round counters are initially 0.
The node passed to TreeSearch is initially my leaf, later is the leaf we last
searched (LastLeaf).
Recall that leaf nodes are segments of the pool.
```

Figure 2: The Tree Search Algorithm.

3.2 Algorithms for Comparison

To evaluate the performance of the *tree* search algorithm, it was compared with two other simple search algorithms, *linear* and *random*. 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. This algorithm is shown in Figure 3. The random algorithm chooses segments at random until it finds a non-empty segment to split. Figure 4 presents this algorithm.

```
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.


The segment passed to LinearSearch is initially my segment, later is the segment
we last searched (LastFound).
```

Figure 3: The Linear Search Algorithm.

```
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.
```

Figure 4: The Random Search Algorithm.

These algorithms will help to contrast the overhead incurred by the superimposed tree directly against any savings due to a reduced number of remote segment accesses, as well as to help isolate interference effects that may arise from the regularity of either the linear or tree search algorithms.

It is fairly easy to see (in all three algorithms) that a processor 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 new elements are being produced relatively infrequently. In fact, if all processors empty their segments and begin 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 processors will ever find an element. This is a difficulty that must be solved in any implementation of the algorithms. For simplicity, our implementations of the above algorithms keep a count of the processes looking for elements. When any processor discovers that all the processors involved in the pool operations are looking (and therefore no processor 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[4]. It became evident in these 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 as much as possible, representing them as a single counter that is atomically added to, subtracted from, or split in half. This minimizes the time involved in segment operations, allowing the search time to dominate most of the measurements and making the effects of the search schemes become more evident.

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 at each processor, and a producer/consumer arrangement, in which some of the processors only add elements and the others only remove elements. Certainly, these represent two extremes, the former balancing the operations among the processors and the latter separating them completely.

In the *random* operations model, all processors perform the same mix of additions and removals. Each processor 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 processor's role as either producer or consumer throughout an experiment is a simplifying assumption. In many real systems, the identity of the processors 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 proves to be significant.

3.4 Measurement and Analysis

It is clear that the primary components of the overall performance of the pools structure can be attributed to the two algorithmic components: 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. By using a very simple mechanism for segment manipulations, we minimize the interference at this level and concentrate on the search effectiveness and interference.

The central idea 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

The segmentation of the concurrent pool provides very well for the locality of the operations, and in reasonable situations with a fairly full pool and sufficient add operations, the pool performs admirably, with steals from remote segments being rather rare.[1] To better understand the search

[1] How rare, of course, depends on the job mix and pattern of operations, as will be seen. With *sufficient* mixes, typically 0-20% of the removes are steals.

algorithm and the effects of different workloads, we began with the pool quite empty for the number of operations to be performed, forcing the processors 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: 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 on each processor, the processors performed operations until the combined total number of operations reached the desired amount.

To obtain an average value of a particular measurement, say, the time required for an *add* operation, the time used by all add operations on a particular processor was recorded and divided by the number of add operations on that processor. This average was then averaged over all processors performing the operations. This trial average was then averaged over the ten trials to obtain the final average add operation time for the given workload.[2] The average value was then plotted as the job mix varied. Thus, we compute the average for a measurement m of an operation as follows:

$$\hat{m} = \frac{1}{N_{trials}} \sum_{trials} \left\{ \frac{1}{N_{procs}} \sum_{procs} \left[\frac{1}{N_{ops}} \sum_{ops} m \right] \right\}$$

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 at each processor (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 processors.

Workloads experienced in real applications are not likely to be as stressful. Due to processing

[2] An alternative would be to total the measurement over all processors and over all trials, and divide by the total number of measurements:

$$\hat{m} = \frac{\sum_{trials} \sum_{procs} \sum_{ops} m}{N_{trials} N_{procs} N_{ops}}$$

between accesses to the pool, fewer processors 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).

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 5. 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%).

4.2 Balancing the producers

In the producer/consumer model, a certain fraction of the processors were producers and the remainder were consumers. The assignment of roles to processors 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,

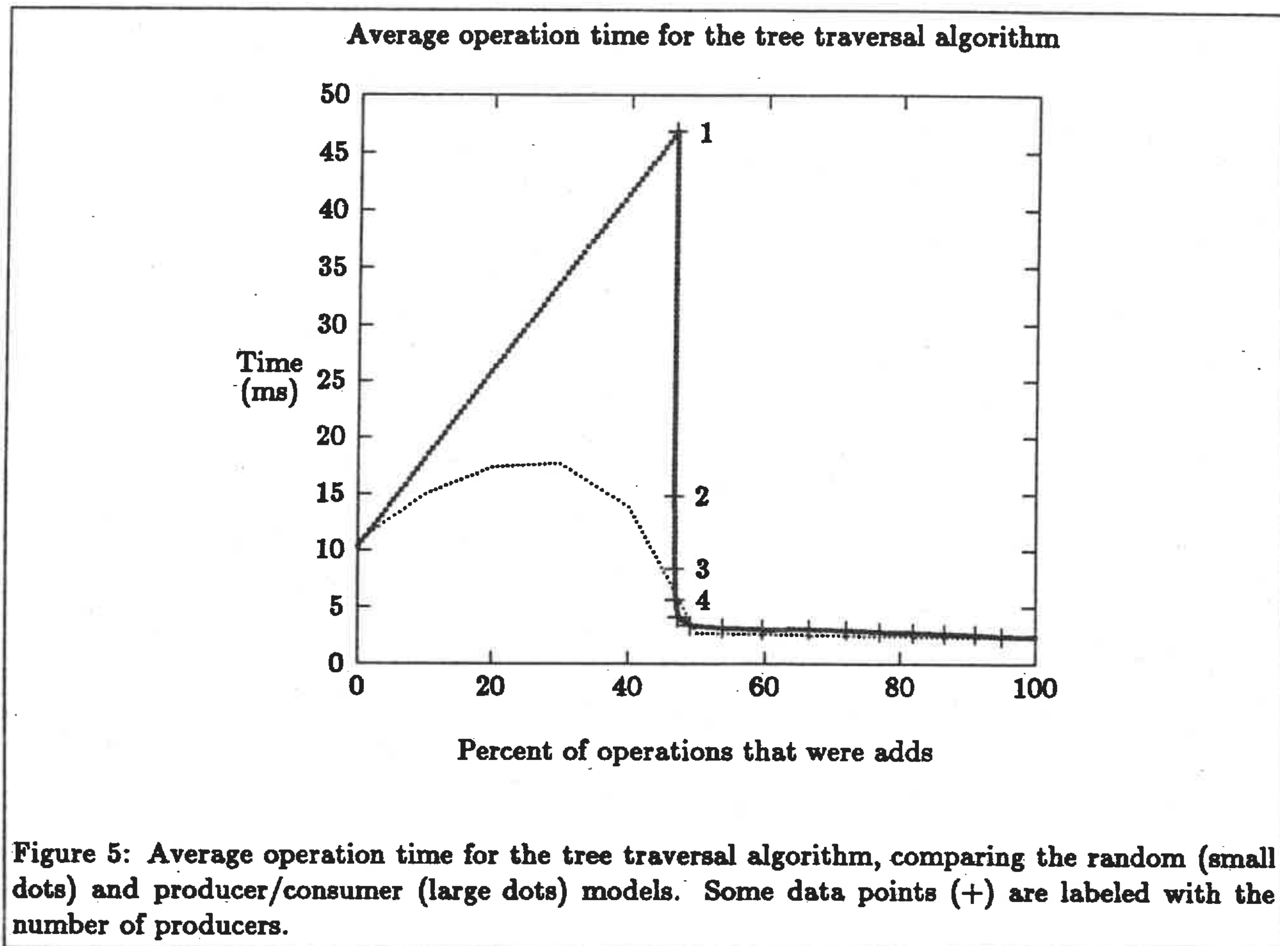

Figure 5: Average operation time for the tree traversal algorithm, comparing the random (small dots) and producer/consumer (large dots) models. Some data points (+) are labeled with the number of producers.

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 for them to spread out to balance the load on the producers. There is increased interference between the processors 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 6 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 com-

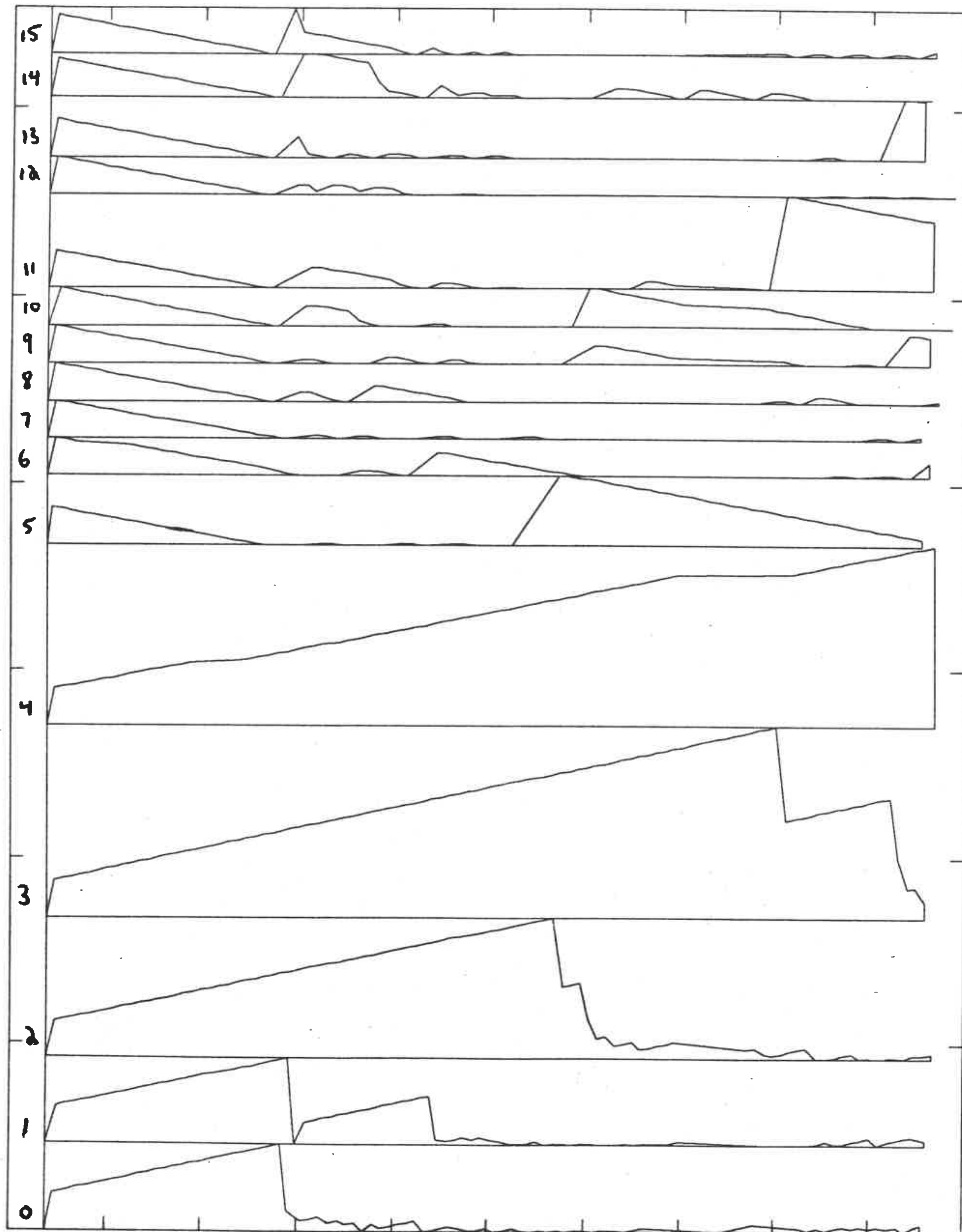

Figure 6: The size of each segment in a 16-processor pool while using the linear traversal algorithm with the producer/consumer model of operations. There are 5 producers and 11 consumers.

plicated, and information marking empty subtrees in the tree helps to steer processes away from empty producers. Figure 7 (in the same style as Figure 6) 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 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 8 and 9 show the effectiveness of balancing the producers in the linear and tree algorithms, respectively: note that 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 10 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.

Of course, balancing the producer/consumer arrangement is a practical management policy only when the role played by a processor 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.

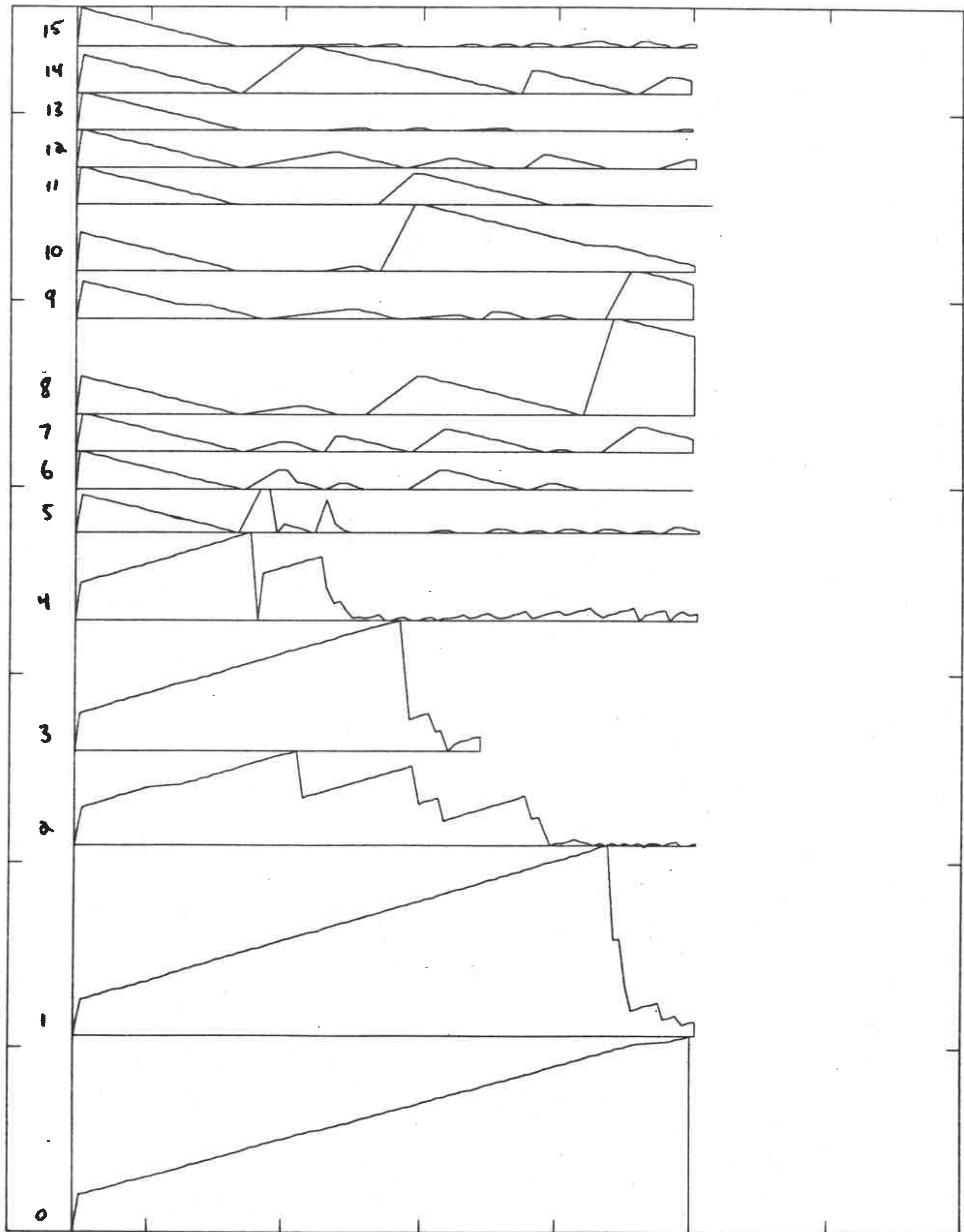

Figure 7: The size of each segment in a 16-processor pool while using the tree traversal algorithm with the producer/consumer model of operations. There are 5 producers and 11 consumers.

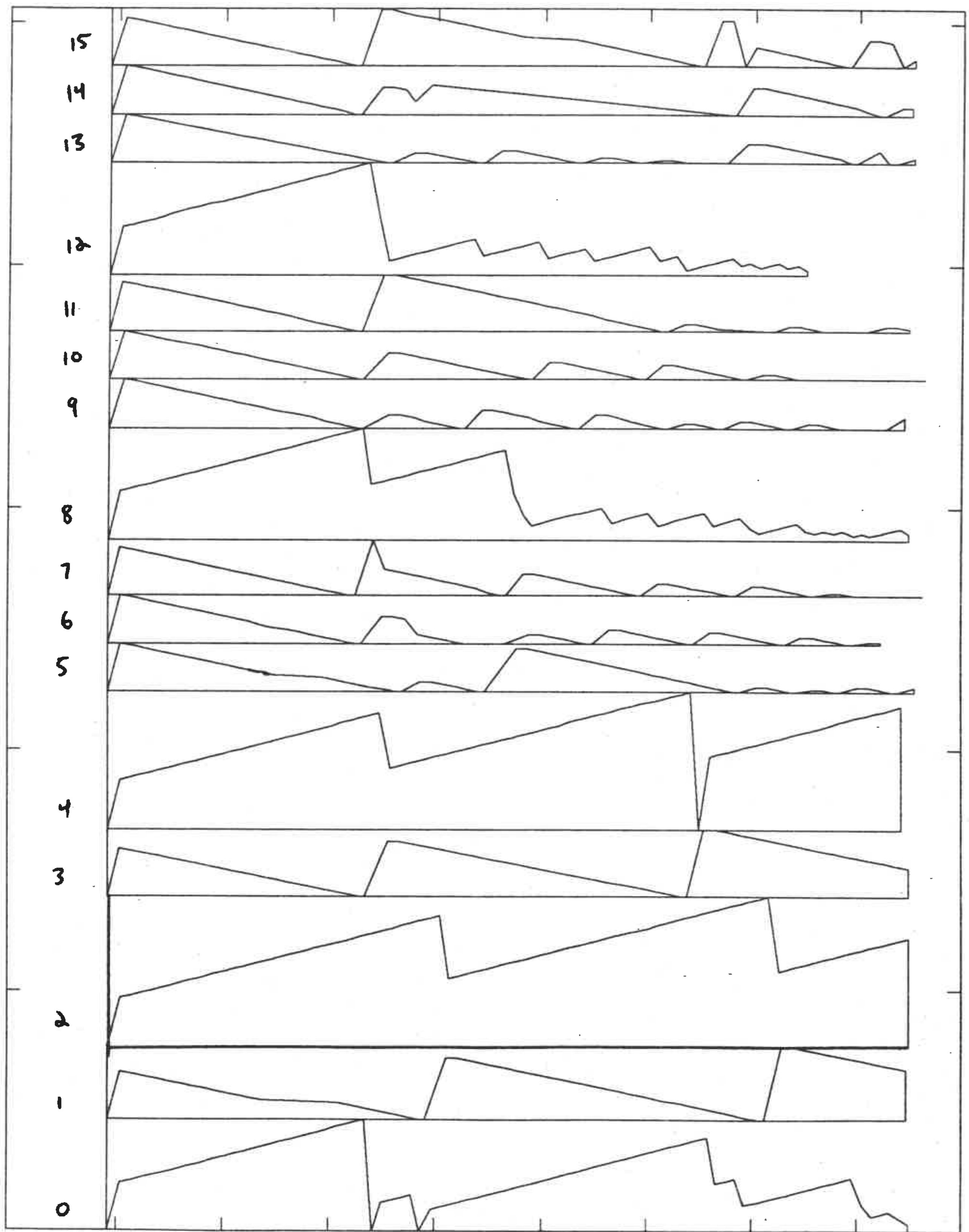

Figure 8: The size of each segment in a 16-processor pool while using the linear traversal algorithm with the 5 producers arranged in a more balanced fashion.

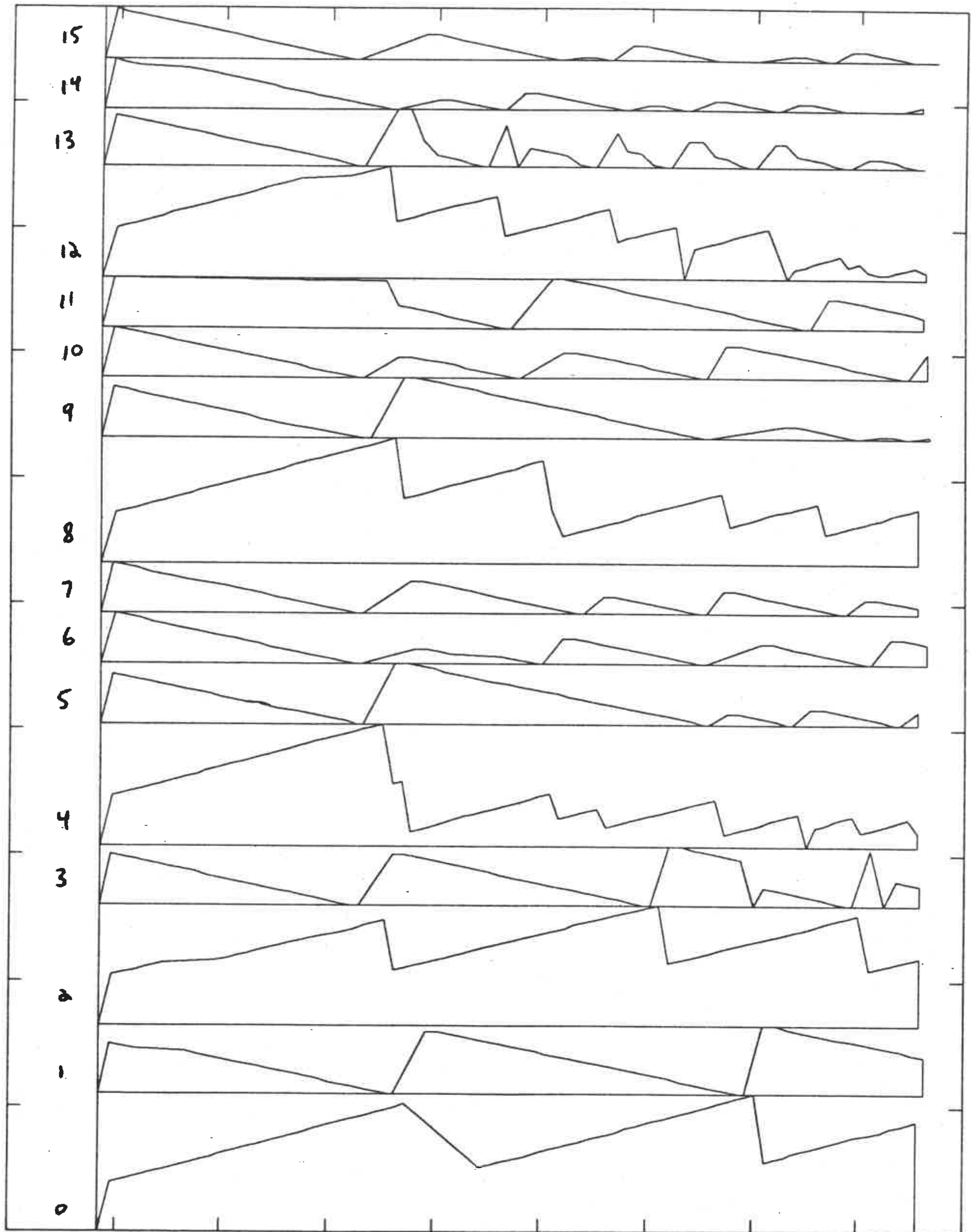

Figure 9: The size of each segment in a 16-processor pool while using the tree traversal algorithm with the 5 producers arranged in a more balanced fashion.

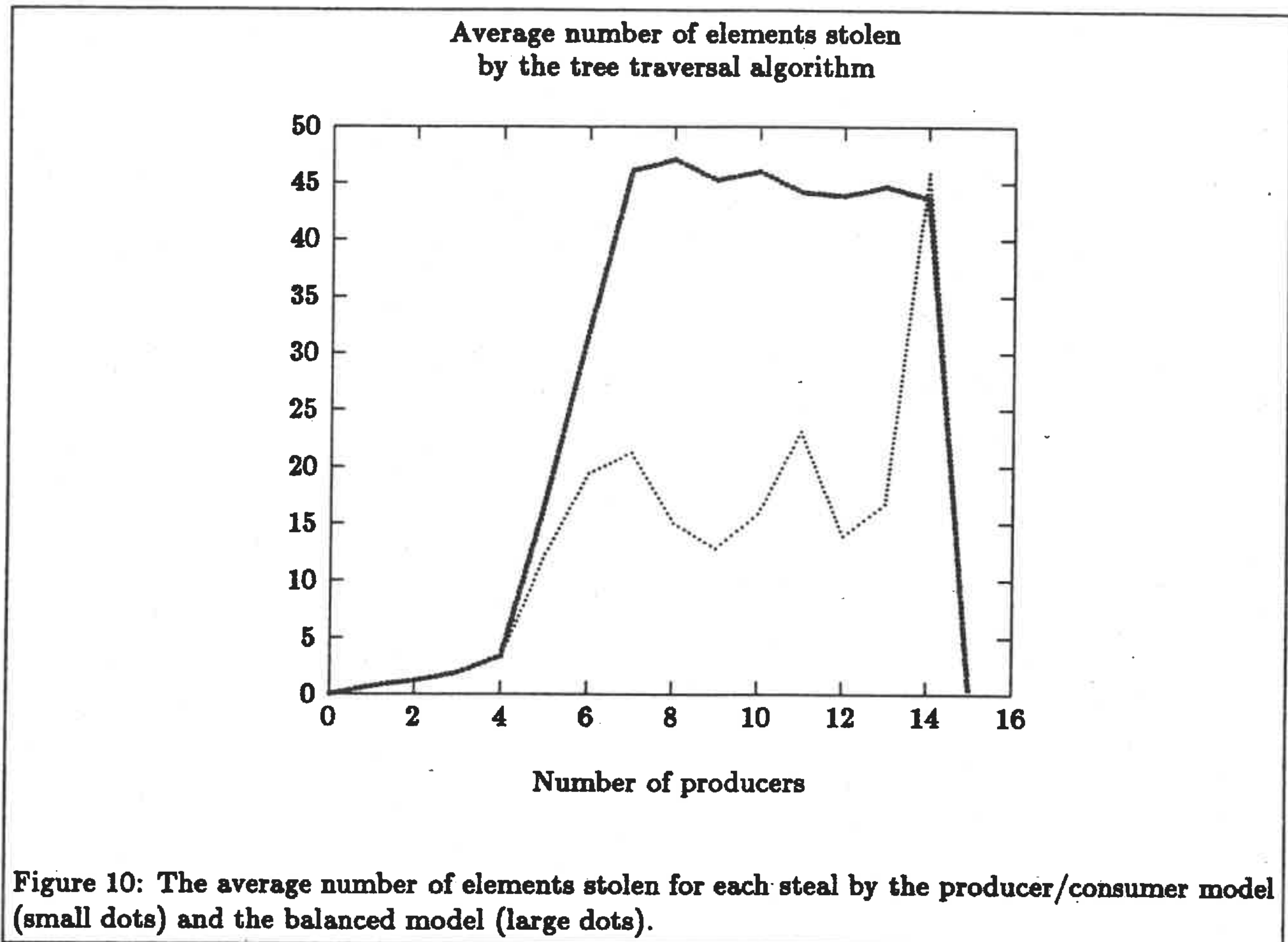

Figure 10: The average number of elements stolen for each steal by the producer/consumer model (small dots) and the balanced model (large dots).

4.3 The Tree Search Algorithm

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

(remove, split, test-for-emptiness) 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 did 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. Indeed, a recent 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 to be quite good; no mention was made of any implementation of the more complex tree search algorithm.

5 Conclusions

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 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 out of sixteen), 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 Manber (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 far 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.

Although 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, it is not clear that the tree search algorithm is useful. The linear or the random search algorithm may suffice and provide better performance.

In general, it is advantageous 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.

These observations reaffirm the value of experimental algorithm analysis, especially in parallel environments. Implementations on real parallel machines can pinpoint the true bottlenecks and identify counterproductive "improvements" in proposed algorithms.

5.1 Further work

Testing on larger configurations and with different architectures still remains to be tried. The pool may not need to be filled to such a low level to obtain noticeable effects.

Other data structures that can benefit from attempts to preserve locality, or which may be modified to preserve locality, may be very useful in a variety of architectures in which different parts of the data structure have different access costs, such as the Butterfly, hypercube–based multiprocessors, or LAN-based distributed systems.

It would be interesting to devise an algorithm that allows a process desiring an element but that can find none to make this known so that any element additions occurring soon are somehow communicated directly to that process. Perhaps, when empty, the pool would fill with *requests* for elements that could later be served directly by new additions. (This concept is analogous to the dual-queue data structure supported by the Butterfly's Chrysalis^{TM} operating system[1].) This would prevent the processes from cycling continuously in the pool, looking for elements.

References

[1] BBN Laboratories, Inc., "Butterfly Parallel Processor Overview", June, 1985.

[2] Raphael Finkel and Udi Manber, "DIB — A Distributed Implementation of Backtracking", *ACM Trans. on Programming Languages and Systems*, pp. 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] Udi Manber, "On Maintaining Dynamic Information in a Concurrent Environment", *SIAM J. Computing*, pp. 1130-1142, November 1986.