An Efficient Algorithm for Concurrent Priority Queue Heaps*

Galen C. Hunt Maged M. Michael Srinivasan Parthasarathy Michael L. Scott

Department of Computer Science, University of Rochester, Rochester, NY 14627-0226
{gchunt,michael,srini,scott}@cs.rochester.edu

December 1994

Abstract

We present a new algorithm for concurrent access to array-based priority queue heaps. Deletions proceed top-down as they do in a previous algorithm due to Rao and Kumar [6], but insertions proceed bottom-up, and consecutive insertions use a bit-reversal technique to scatter accesses across the fringe of the tree, to reduce contention. Because insertions do not have to traverse the entire height of the tree (as they do in previous work), as many as $O(M)$ operations can proceed in parallel, rather than $O(\log M)$ on a heap of size M. Experimental results on a Silicon Graphics Challenge multiprocessor demonstrate good overall performance for the new algorithm on small heaps, and significant performance improvements over known alternatives on large heaps with mixed insertion/deletion workloads.

*This work was supported in part by NSF grants nos. CDA-8822724 and CCR-9319445, and by ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

```
void insert(int priority, int data, heap_t *heap)
{
  LOCK(heap->lock);
  i = ++heap->size;   ip = &heap->item[i];
  ip->pri = priority;   ip->dat = data;
  while(i>1)
  {
    parent = i>>1;   pp = &heap->item[parent];
    if(ip->pri < pp->pri)   SWAP(ip,pp);   /* swap items i and parent */
    else   break;
    i = parent;   ip = pp;
  }
  UNLOCK(heap->lock);
}

int delete(int *priority, int *data, heap_t *heap)
{
  LOCK(heap->lock);
  if(heap->size < 1){ UNLOCK(heap->lock);   return 0; }
  i=1;   ip = &heap->item[1];
  *priority = ip->pri;   *data = ip->dat;
  lp = &heap->item[heap->size--];
  ip->pri = lp->pri;   ip->dat = lp->dat;
  while(i <= heap->size>>1){
    min = MIN_CHILD(i);   /* index of child with higher priority */
    mp = &heap->item[min];
    if(ip->pri > mp->pri)   SWAP(ip,mp);   /* swap items i and min */
    else   break;
    i = min;   ip = mp;
  }
  UNLOCK(heap->lock);
  return 1;
}
```

Figure 1: Single-lock heap operations.

1 Introduction

The heap data structure is widely used as a priority queue [2]. The basic operations on a priority queue are *insert* and *delete*. *Insert* inserts a new item in the queue and *delete* removes and returns the highest priority (lowest numbered) item from the queue. A heap is a binary tree with the property that the value of the key at any node is less than the value of the keys at its children (if they exist). An array representation of a heap is the most space efficient: the root of the heap occupies location 1 and the left and right children of the node at location i occupy the locations $2i$ and $2i+1$, respectively. No items exist in level l of the tree unless level $l-1$ is completely full.

Many applications (e.g. heuristic search algorithms, graph search, and discrete event simulation [4, 5]) on shared memory multiprocessors use shared priority queues to schedule sub-tasks. In these applications, items can be simultaneously inserted and deleted from the heap by any of the participating processes. The simplest way to ensure the consistency of the heap is to serialize the updates by putting them in critical sections protected by a mutual exclusion lock (see figure 1 for C-like pseudo-code for insert and delete operations). This approach limits concurrent operations on the heap to one. Since updates to the heap typically modify only a small fraction of the nodes, more concurrency should be achievable by allowing processes to access the heap concurrently as long as they do not interact with each other.

Biswas and Browne [1] proposed a scheme that allows many insertions and deletions to proceed concurrently. Their scheme relies on the presence of maintenance processes that dequeue

sub-operations from a FIFO work queue. Sub-operations are placed on the work queue by the processes performing insert and delete operations. The work queue is used to avoid deadlock due to insertions and deletions proceeding in opposite directions in the tree. The need for a work queue and maintenance processes causes this scheme to incur substantial overhead. Rao and Kumar [6] present another scheme that avoids deadlock by using top-down insertions, where an inserted item has to traverse a path through the whole height of the heap. Jones [3] presents a concurrent priority queue algorithm using skew heaps. He notes that top-down insertions in array-based heaps are inefficient, while bottom-up insertions would cause deadlock if they collide with top-down deletions without using extra server processes.

This paper presents a new concurrent priority queue heap algorithm that addresses the problems encountered in previous research. On large heaps the algorithm achieves significant performance improvements over both the serialized single-lock algorithm and the algorithm of Rao and Kumar, for various insertion/deletion workloads. For small heaps it still performs well, but not as well as the single-lock algorithm. The new algorithm allows concurrent insertions and deletions in opposite directions, without risking deadlock and without the need for special server processes. It also uses a bit-reversal technique to scatter accesses across the fringe of the tree to reduce contention.

Section 2 presents the new algorithm and an analysis of its performance advantages. Section 3 presents experimental results on a Silicon Graphics Challenge multiprocessor. It compares the new algorithm to the single-lock algorithm and to the algorithm of Rao and Kumar, demonstrating performance improvements for a variety of workloads. Section 4 summarizes our conclusions.

2 The Algorithm

The new algorithm uses mutual exclusion locks on each node in the heap and on a variable that holds the number of items in the heap. Also, each node has a tag that indicates whether it is empty, valid, or in transient state due to an update to the heap by process *pid*. The tags serve to allow (bottom-up) insertions and (top-down) deletions to proceed in opposite directions without the need for a work queue or extra service processes [1]. The tags in the new algorithm allow a process to efficiently identify the item it is moving up or down the heap even if the item has been swapped by another process. For example, when a (top-down) delete operation swaps an item that is being inserted (bottom-up), the tags prevent the unlocked inserted item from being wrongly swapped by another parallel insert operation. Another advantage of the new algorithm is that unlike top-down insertions, bottom-up insertions do not necessarily have to traverse the whole height of the heap to complete the operation, thus reducing traversal overhead, and contention on topmost nodes.

In some definitions of heaps [2], all nodes in the last level of the heap to the left of the last item have to be non-empty. This is not required by priority queue semantics, or heap logarithmic complexity. In the new algorithm, we relax this restriction. Consecutive insertions traverse different sub-trees by using a bit reversal technique similar to that of an FFT computation [2]. For example, in the 3rd level of a heap (nodes 8-15, if the root is node 1), eight consecutive insertions would start from the nodes 8, 12, 10, 14, 9, 13, 11, and 15, respectively. Notice that for any two consecutive insertions, the two paths from each of the bottom level nodes to the root of the heap have no common nodes other than the root, thus reducing the contention on node locks. Similarly, consecutive deletions from the heap would follow the same pattern but in reverse order.

Since insertions in the new algorithm do not have to traverse the whole height of the tree, and since consecutive insertions have almost disjoint paths to the root, $O(M)$ heap operations can make progress concurrently. The bound on concurrency in Rao and Kumar's algorithm is $O(\log M)$.

```
void concurrent_insert(int priority, int data, heap_t *heap)
{
  LOCK(heap->lock);
  i = BIT_REVERSE(++heap->size);   ip = &heap->item[i];
  LOCK(ip->lock);
  UNLOCK(heap->lock);
  ip->pri = priority;   ip->dat = data;
  if(i == 1){ ip->tag = PRESENT;   i = 0; }
  else   ip->tag = pid;
  UNLOCK(ip->lock);
  while(i > 1){
    parent = i>>1;   pp = &heap->item[parent];
    LOCK(pp->lock);
    ip = &heap->item[i];
    if(pp->tag == PRESENT)   has_p_lck = 1;
    else if(pp->tag == EMPTY){ UNLOCK(pp->lock);   i = 0;   break; }
    else if(pp->tag == pid){ UNLOCK(pp->lock);   i = parent;   continue; }
    else{ UNLOCK(pp->lock);   has_p_lck = 0; }
    LOCK(ip->lock);
    if(ip->tag == pid){
      if(has_p_lck){
        if(ip->pri < pp->pri){
          SWAP(pp,ip);
          if(parent == 1){ pp->tag = PRESENT;   next = 0; }   else   next = parent;
        }else{ ip->tag = PRESENT;   next = 0; }
      }else   next = i;
    }else   next = parent;
    if(has_p_lck)   UNLOCK(pp->lock);
    UNLOCK(ip->lock);
    i = next;
  }
  if(i == 1){
    ip = &heap->item[i];
    LOCK(ip->lock);
    if(ip->tag == pid)   ip->tag = PRESENT;
    UNLOCK(ip->lock);
  }
}
```

Figure 2: Concurrent insertion.

Figures 2 and 3 present C-like pseudo-code for the insertion and deletion parts of the new algorithm, respectively. Initially, all locks are free, all node tags are set to `EMPTY`, and the number of elements in the heap is zero.

Instead of computing the bit-reverse ($O(n)$ time, where n is the number of bits to be reversed) for each operation on the heap, we use a bit-reverse counter with amortized time $O(1)$ for long sequences of increments only or decrements only. However alternating increments and decrements may result in $O(n)$ complexity. Figure 4 shows the bit-reversal routines.

3 Experimental Results

3.1 Methodology

We used a 12-processor Silicon Graphics Challenge multiprocessor to compare the performance of the new algorithm, the single-lock algorithm, and Rao and Kumar's algorithm. We tried the latter both with and without adding our bit-reversal technique.

```
int concurrent_delete(int *priority, int *data, heap_t *heap)
{
  LOCK(heap->lock);
  if(heap->size < 1){ UNLOCK(heap->lock);   return 0; }
  last = BIT_REVERSE(heap->size--);   lastp = &heap->item[i];
  LOCK(lastp->lock);
  UNLOCK(heap->lock);
  captive_pri = lastp->pri;   captive_dat = lastp->dat;  lastp->tag = EMPTY;
  UNLOCK(lastp->lock);
  i = 1;   ip = &heap->item[1];
  LOCK(ip->lock);
  if(ip->tag == EMPTY){ UNLOCK(ip->lock);   *priority = captive_pri;   *data = captive_dat;   return 1; }
  *priority = ip->pri;   *data = ip->dat;
  ip->pri = captive_pri;   ip->dat = captive_dat;   ip->tag = PRESENT;
  left = i<<1;
  while(left < MAX_SIZE){
    right = left+1;   lp = &heap->item[left];   rp = &heap->item[right];
    LOCK(lp->lock);
    if(lp->tag == EMPTY){ UNLOCK(lp->lock);   break; }
    else{ min = left;   mp = lp; }
    if(right < MAX_SIZE){
      LOCK(rp->lock);
      np = rp;
      if(rp->tag != EMPTY)
        if(rp->pri < lp->pri){ min = right;   mp = rp;   np = lp; }
      UNLOCK(np->lock);
    }
    if(mp->pri < ip->pri)   SWAP(ip,mp);
    else{ UNLOCK(mp->lock);   break; }
    UNLOCK(ip->lock);
    i = min;   ip = mp;   left = i<<1;
  }
  UNLOCK(ip->lock);
  return 1;
}
```

Figure 3: Concurrent deletion.

```
/* initially *counter = *reverse = 0,  *high_bit = don't_care */

void increment(int *counter, int *reverse, int *high_bit)
{
  if(*counter++ == 0){ *reverse = *high_bit = 1;  return; }
  bit = *high_bit>>1;
  while(bit){ *reverse ^= bit;  if(*reverse & bit) break;  bit >>= 1; }
  if(!bit)  *reverse = *high_bit <<= 1;
}

void decrement(int *counter, int *reverse, int *high_bit)
{
  *counter--;
  bit = *high_bit>>1;
  while(bit){ *reverse ^= bit;  if(!(*reverse & bit)) break;  bit >>= 1; }
  if(!bit){ *reverse = *counter;  *high_bit >>= 1; }
}
```

Figure 4: A bit-reverse counter.

For mutual exclusion we used test-and-test-and-set locks with backoff using the MIPS R4000 `load-linked` and `store-conditional` instructions. On small-scale multiprocessors like the Challenge, these locks have low overhead compared to other more scalable locks.

To evaluate the performance of the algorithms under different levels of contention, we varied the number of processes in our experiments. Each process runs on a dedicated processor in a tight loop where it repeatedly updates a shared heap. Thus, in our experiments the number of processors corresponds to the level of contention. We believe these results to be comparable to what would be achieved with a much larger number of processes, each of which was doing significant real work between queue operations. In all experiments, processors perform equal workloads.

We studied the performance with workloads of insertions only, deletions only, and various mixed insert/delete distributions. We also varied the initial number of full levels in the heap before starting time measurements to identify performance differences with different heap sizes. For the mixed insert/delete experiments we used workloads of 200,000 heap operations. Experiments with smaller workloads are too fast to time. In these experiments, the heap size remains almost constant as the number of insertions and deletions are equal and processors alternate performing insertions and deletions. We also ran experiments with 100,000 insertions only, and with 100,000 deletions only on a 17-level-full heap. Inserted values were chosen randomly and uniformly on the domain of 32-bit integers.

All the C programs for the different algorithms were compiled with the highest optimization level, and were carefully hand-optimized. For the multiple-lock algorithms we changed the data layout to reduce the effect of false sharing, but we did not apply this optimization to the single lock algorithm as it does not need it and it would only produce unnecessary overhead. Therefore, in our experiments we were using the best version of each algorithm, thus guaranteeing fair evaluation. The programs are accessible by anonymous ftp to `ftp.cs.rochester.edu/pub/packages/concurrent_heap`, or by contacting any of the authors.

3.2 Results

Figures 5 and 6 show the time taken to perform 100,000 insertions and deletions, respectively, on a heap with 17 full levels. Figure 7 shows the time taken to perform 10,000 sets of 10 insertions and 10 deletions on an empty heap. Figures 8 and 9 show the time taken to perform 100,000 insert/delete pairs on a 7-level-full heap and a 17-level-full heap, respectively.

In the case of insertions only without deletions (figure 5), the single-lock and the new algorithm have better performance because insertions do not have to traverse the whole height of the tree (as they do in Rao and Kumar's algorithm), and most inserted items settle in the two bottom-most levels of the heap. In effect, insert operations for the single-lock algorithm in this case are fast enough that greater potential for concurrency in the new multi-lock algorithm does not matter much.

In the case of deletions only without insertions (figure 6), most deletions have to traverse the whole height of the tree. Therefore, the delete operation traverses many nodes, and the multi-lock algorithms outperform the single-lock algorithm. Because deletions in the new algorithm proceed top-down in essentially the same manner as in Rao and Kumar's algorithm, the two algorithms display very similar performance.

In the case of alternating insertions and deletions on an initially empty heap (figure 7), the average height of the heap ranges from 3 to 5. The single-lock algorithm outperforms the other algorithms because it has low overhead and there is relatively little opportunity for the multi-lock algorithms to exploit concurrency on very small heaps. Comparing the new algorithm with that of Rao and Kumar, we find that the new algorithm yields better performance because it suffers less

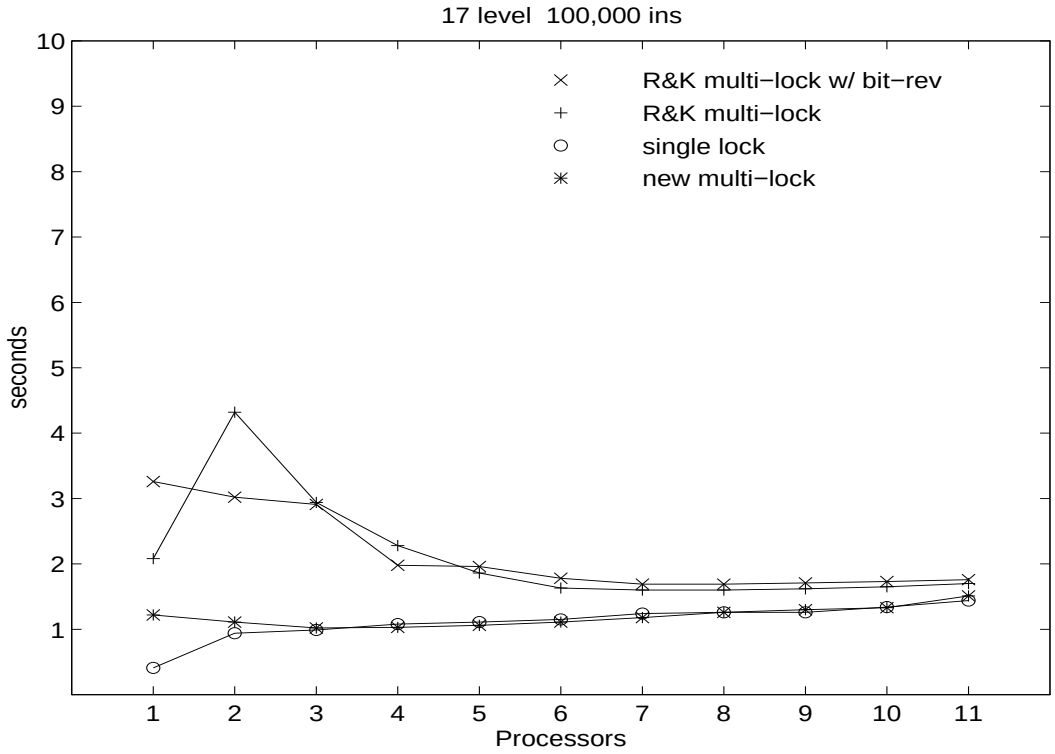

Figure 5: Performance results for 100,000 insertions.

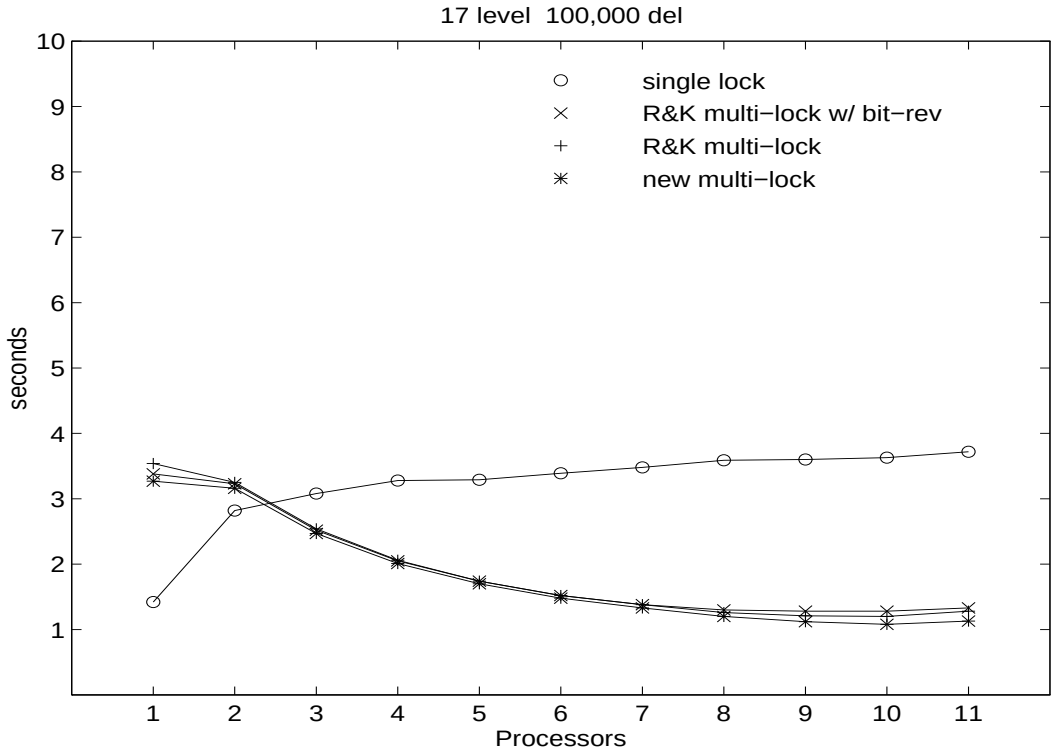

Figure 6: Performance results for 100,000 deletions.

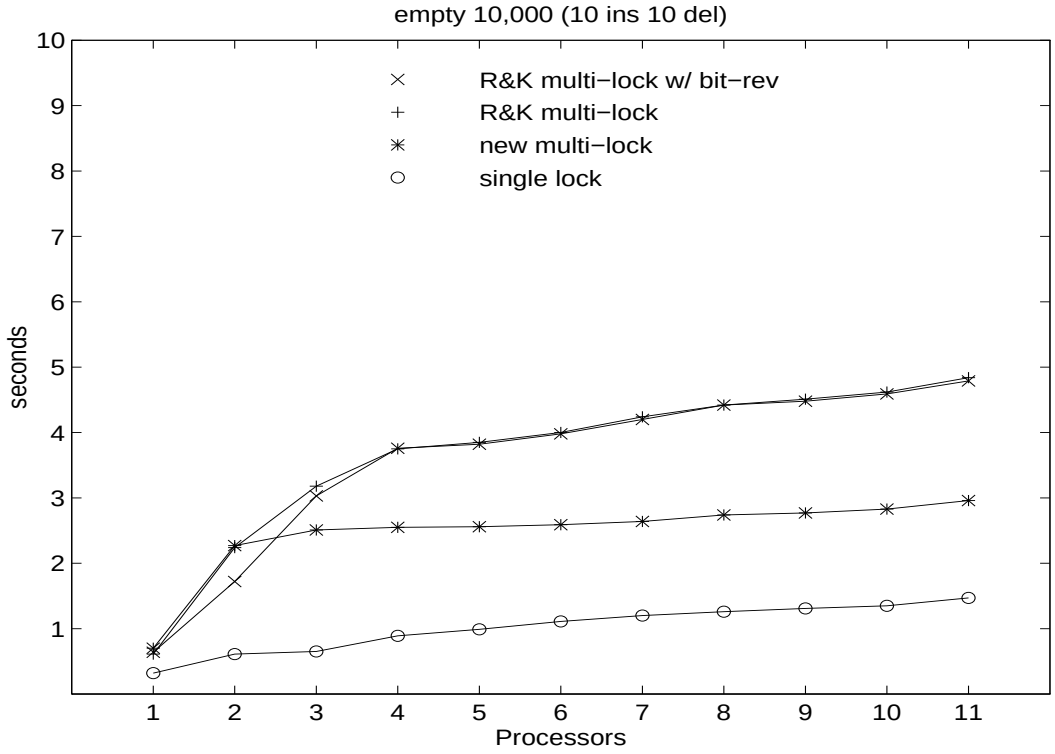

Figure 7: Performance results for 10,000 sets of 10 insertions and 10 deletions on an empty heap.

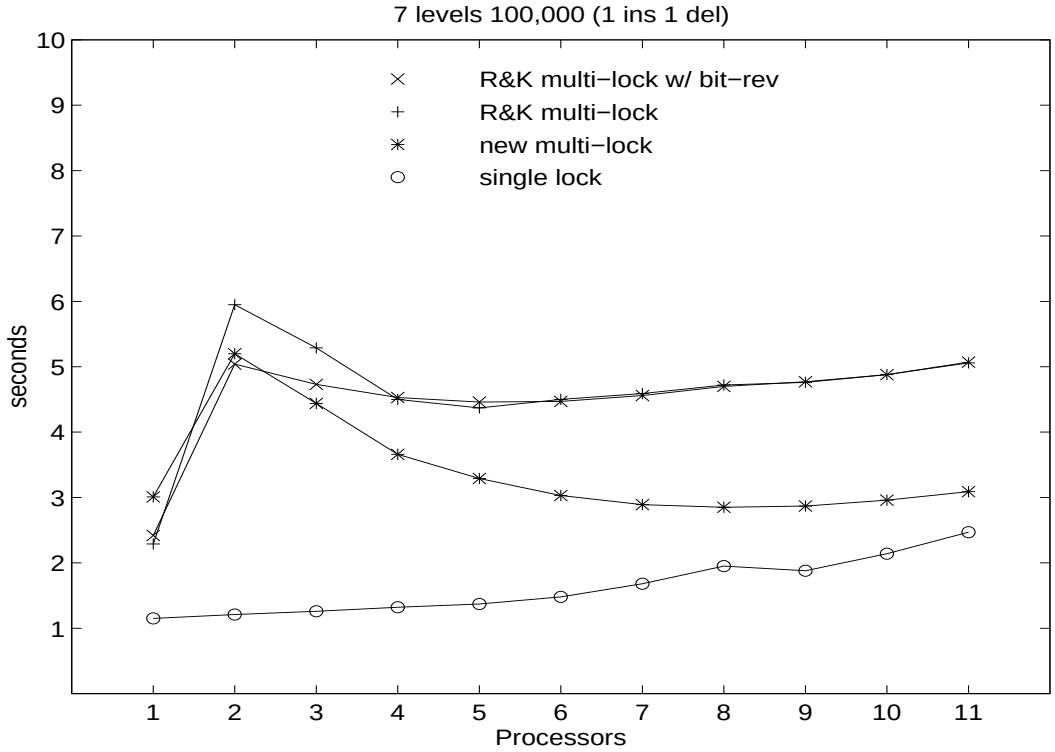

Figure 8: Performance results for 100,000 insert/delete pairs on a 7-level-full heap.

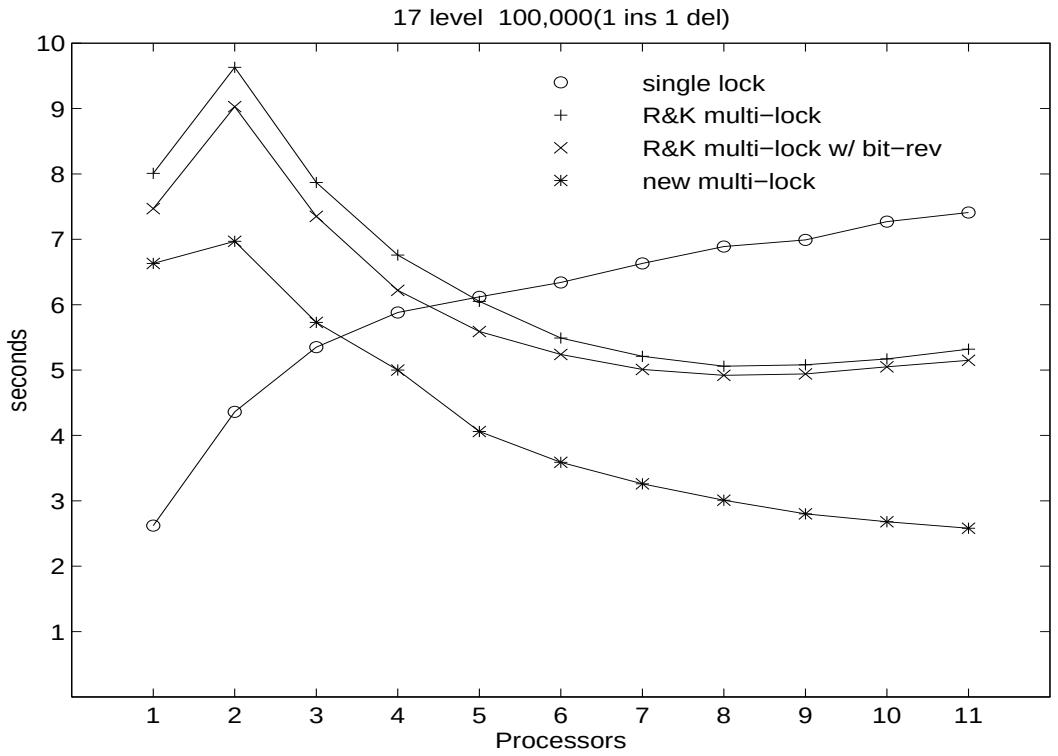

Figure 9: Performance results for 100,000 insert/delete pairs on a 17-level-full heap.

from contention on the topmost nodes of the heap. Note that after several insert/delete cycles, the items remaining in the heap tend to be of very low priority, so new insertions have to traverse most of the path to the root in the new algorithm. This means that the performance advantage of the new algorithm over that of Rao and Kumar in this case is more because of reduced contention for the topmost nodes of the tree (due to opposite directions for insertion and deletion) than because of shorter traversals.

In the case of alternating insertions and deletions on a 7-level-full heap (figure 8), the height of the heap remains almost constant. The single-lock algorithm continues to outperform the others because of its low overhead, but the difference between it and the new algorithm narrows as the level of contention increases, since 7 levels provide the new algorithm with reasonable opportunities for concurrency. Rao and Kumar's algorithm suffers from high contention on the topmost nodes.

In the case of alternating insertions and deletions on a 17-level-full heap (figure 8), the large height of the heap makes concurrency, rather than locking overhead, the dominant factor in performance. The multi-lock algorithms consequently show improved performance over the single-lock algorithm. As in the case of the empty and 7-level-full heaps, most new insertions tend to have higher priorities than the items already in the heap, and thus eventually settle near the top of the heap. In spite of this, the new algorithm outperforms that of Rao and Kumar because of reduced contention on the topmost nodes.

4 Conclusions

We have presented a new algorithm that uses multiple mutual exclusion locks to allow consistent concurrent access to array-based priority queue heaps. The new algorithm avoids deadlock among

concurrent accesses without forcing insertions to proceed top-down [6] or introducing a work queue and extra processes [1]. Bottom-up insertions reduce contention for the topmost nodes of the heap, and avoid the need for a full-height traversal in many cases. The new algorithm also uses bit-reversal to increase concurrency among consecutive insertions, allowing them to follow mostly-disjoint paths.

We compared the performance of the new algorithm, the single-lock algorithm, and Rao and Kumar's top-down insertion algorithm [6] on a 12-node SGI Challenge multiprocessor. The results show that the new algorithm provides reasonable performance on small heaps, and significantly superior performance on large heaps under high levels of contention.

References

[1] J. Biswas and J. C. Browne. Simultaneous Update of Priority Structures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 124–131, August 1987.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[3] D. W. Jones. Concurrent Operations on Priority Queues. *Communications of the ACM*, 32(1):132–137, January 1989.

[4] J. Mohan. Experience with Two Parallel Programs Solving the Travelling Salesman Problem. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 191–193, 1983.

[5] M. J. Quinn and N. Deo. Parallel Graph Algorithms. *ACM Computing Surveys*, 16(3):319–348, September 1984.

[6] V. N. Rao and V. Kumar. Concurrent Access of Priority Queues. *IEEE Transactions on Computers*, 37(12):1657–1665, December 1988.