ELSEVIER

Information Processing Letters 60 (1996) 151–157

Information Processing Letters

An efficient algorithm for concurrent priority queue heaps

Galen C. Hunt [1], Maged M. Michael *, Srinivasan Parthasarathy [1], Michael L. Scott [1]

Department of Computer Science, University of Rochester, Rochester, NY 14627-0226, USA

Received 30 March 1995; revised 2 September 1996
Communicated by G.R. Andrews

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 (1988), 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.

Keywords: Concurrency; Data structures; Priority queue; Heap; Bit-reversal; Synchronization

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 item from the queue. A heap is a binary tree with the property that the key at any node has higher priority than 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 [5,6]) on shared memory multiprocessors use shared priority queues to schedule sub-tasks. In these applications, items can be 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. 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

* Corresponding author. Email: michael@cs.rochester.edu.
[1] Email: {gchunt,srini,scott}@cs.rochester.edu.

PII S0020-0190(96)00148-2

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 [7] 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 (insertions in a traditional sequential heap proceed bottom-up). Jones [3] presents a concurrent priority queue algorithm using skew heaps. [2] 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.

2. The new algorithm

The new algorithm augments the standard heap data structure [2] with a mutual-exclusion lock on the heap's size and locks on each node in the heap. Each node also has a tag that indicates whether it is empty, valid, or in a transient state due to an update to the heap by an inserting process. Nodes that contain no data are tagged `EMPTY`. Nodes that are available for deletion are tagged `AVAILABLE`. A node that has been inserted, and is being moved into place, is tagged with the process identifier (*pid*) of the inserting process.

[2] Array based heaps can be considered as a binary tree that is filled at all levels except possibly the last level. In skew heaps this restriction is relaxed; the representative binary tree need not be filled at all the intermediate levels.

A delete operation in the new algorithm, as in the sequential algorithm, starts by reading the data and priority of the root of the heap and then replacing them with those of as rightmost node in the lowest level of the heap. Then, the delete operation "heapifies" the heap. It compares the priority of the root with that of each of its children (if any). If necessary, it swaps the root item with one of its children in order to ensure that none of the children has priority higher than the root. If no swapping is necessary the delete operation is complete; it returns the data that was originally in the root. Otherwise, the operation recursively "heapifies" the subheap rooted at the swapped child. To handle concurrency all these steps are performed under the protection of the locks on the individual nodes and a lock on the size of the heap. In each step of the heapify operation, the lock of the subtree root is already held. It is not released until the end of that step. Prior to comparing priorities, the locks of the children are acquired. If swapping is performed, the lock on the swapped child is retained through the next recursive heapify step, and the locks on the root and the unswapped child are released. Otherwise, all the locks are released, and the delete operation completes.

An insert operation starts by inserting the new data and priority in the lowest level of the heap. If the inserted node is the root of the heap, then the insert operation is complete. Otherwise, the operation compares the priority of the inserted node to that of its parent. If the child's priority is higher than that of its parent, then the two items are swapped, otherwise the insert operation is complete. If swapping was necessary, then the same steps are applied repeatedly bottom-up until reaching a step in which no swapping is necessary, or the inserted node has become the root of the heap. To handle concurrency, all these steps are performed under the protection of the locks and tags on the individual nodes and the lock on the size of the heap. In every step of the bottom-up comparison, the lock of the parent is acquired first, followed by the lock on the inserted node. After comparison and swapping (if necessary), both locks are released. Locks are acquired in the same order as in the delete operation, parent-child, to avoid deadlock. This mechanism requires releasing and then acquiring the lock on the inserted item between successive steps, opening a window of vulnerability during which the inserted item might be

swapped by other concurrent operations. Tags are used to resolve these situations.

An insert operation tags the inserted item with its *pid*. In every step, an insert operation can identify the item it is moving up the heap even if the item has been swapped upwards by a deletion. In particular, tags are used in the following manner:

- If the tag of the parent node is equal to AVAILABLE and the tag of the current node is equal to the insert operation's *pid*, then no interference has occurred and the insertion step can proceed normally.
- If the tag of the parent node is equal to EMPTY, then the inserted item must have been moved by a delete operation to the root of the heap. The insert operation is complete.
- If the tag of the current node is not equal to the operation's *pid*, then the inserted item must have been moved upwards by a delete operation. The insert operation moves upward in pursuit of the inserted item.

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. Since this is not required by priority queue semantics, in the new algorithm we chose to relax this restriction to reduce lock contention, and thereby permit more concurrency. Under our relaxed model, consecutive insertions traverse different sub-trees by using a "bit-reversal" technique similar to that of an FFT computation [2]. For example, in the third level of a heap (nodes 8–15, where node 1 is the root), 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. This lack of overlap serves to reduce contention for node locks. Consecutive deletions from the heap follow the same pattern, but in reverse order. The relation between the indices of parents and children remains as it is in heaps without bit reversal. The children of node i are nodes $2i$ and $2i+1$, and the parent of node $i > 1$ is node $i/2$. Moreover, if a node has only one child, it is still $2i$, never $2i+1$.

Since insertions in the new algorithm do not have to traverse the whole height of the heap, they have a lower bound of $\Omega(1)$ time, while the algorithm due to Rao and Kumar requires $\Omega(\log M)$ time for insertions (top–down) in a heap of size M, as insertions have

```
record data_item
  lock := FREE; tag := EMPTY; priority := 0
record heap
  lock := FREE; bit_reversed_counter size; data_item items[ ]
define LOCK(x) as lock(heap.items[x].lock)
define UNLOCK(x) as unlock(heap.items[x].lock)
define TAG(x) as heap.items[x].tag
define PRIORITY(x) as heap.items[x].priority

procedure concurrent_insert(priority, heap)
  // Insert new item at bottom of the heap.
  lock(heap.lock); i := bit_reversed_increment(heap.size)
  LOCK(i); unlock(heap.lock); PRIORITY(i) := priority
  TAG(i) := pid; UNLOCK(i)

  // Move item towards top of heap while it has higher priority
  // than parent.
  while i > 1 do
    parent := i/2; LOCK(parent); LOCK(i)
    if TAG(parent) = AVAILABLE and TAG(i) = pid then
      if PRIORITY(i) > PRIORITY(parent) then
        swap_items(i, parent); i := parent
      else
        TAG(i) := AVAILABLE; i := 0
    else if TAG(parent) = EMPTY then
      i := 0
    else if TAG(i) ≠ pid then
      i := parent
    UNLOCK(i); UNLOCK(parent)
  enddo
  if i = 1 then
    LOCK(i)
    if TAG(i) = pid then
      TAG(i) := AVAILABLE
    UNLOCK(i)
```

Fig. 1. Concurrent insert operation. For conciseness, we treat *priority* as if it were the only datum in each data_item.

to traverse the entire height of the heap. In addition to reducing traversal overhead, the bottom–up insertion approach of the new algorithm reduces contention on topmost nodes.

We next consider the space requirements for algorithms under consideration. Let M be the maximum number of nodes in the heap, and P the maximum number of processes operating on the heap. Assume that each lock requires one bit of memory. The new algorithm requires 1 bit for the lock on the heap size variable, $3 \log M$ bit-reversal bits, and $1 + \log P$ lock and tag bits per node, for a total of $1 + 3 \log M + (1 + \log P)M$ bits of memory. The single lock algorithm requires 1 bit of memory for the single lock. Rao and Kumar's algorithm requires 3 bits per node for a total

```
function concurrent_delete(heap)
  // Grab item from bottom of heap to replace to-be-deleted
  // top item.
  lock(heap.lock)
  bottom := bit_reversed_decrement(heap.size);
  LOCK(bottom); unlock(heap.lock)
  priority := PRIORITY(bottom)
  TAG(bottom) := EMPTY; UNLOCK(bottom)

  // Lock first item. Stop if it was the only item in the heap.
  LOCK(1); if TAG(1) = EMPTY then UNLOCK(1)
  return priority

  // Replace the top item with the item stored from
  // the bottom.
  swap(priority, PRIORITY(1)); TAG(1) := AVAILABLE

  // Adjust heap starting at top. Always hold lock on item
  // being adjusted.
  i := 1
  while (i < MAX_SIZE / 2) do
    left := i * 2; right := i * 2 + 1; LOCK(left); LOCK(right)
    if TAG(left) = EMPTY then
      UNLOCK(right); UNLOCK(left); break
    else if TAG(right) = EMPTY
    or PRIORITY(left) > PRIORITY(right) then
      UNLOCK(right); child := left
    else
      UNLOCK(left); child := right

    // If child has higher priority than parent then swap.
    // If not, stop.
    if PRIORITY(child) > PRIORITY(i) then
      swap_items(child, i); UNLOCK(i); i := child
    else
      UNLOCK(child); break
  enddo
  UNLOCK(i)
  return priority
```

Fig. 2. Concurrent delete operation.

```
record bit_reversed_counter
  counter := 0; reversed := 0; high_bit := -1

function bit_reversed_increment(c)
  c.counter := c.counter + 1
  for bit := c.high_bit - 1 to 0 step -1
    c.reversed := not(c.reversed, bit)
    if test(c.reversed, bit) = TRUE then
      break
  if bit < 0 then
    c.reversed := c.counter; c.high_bit := c.high_bit + 1
  return c.reversed

function bit_reversed_decrement(c)
  c.counter := c.counter - 1
  for bit := c.high_bit - 1 to 0 step -1
    c.reversed := not(c.reversed, bit)
    if test(c.reversed, bit) = FALSE then
      break
  if bit < 0 then
    c.reversed := c.counter; c.high_bit := c.high_bit - 1
  return c.reversed
```

Fig. 3. A bit-reverse counter.

of $3M$ bits of memory. If bit reversal were added to Rao and Kumar's algorithm, it would require $3 \log M$ extra bits, for a total of $3 \log M + 3M$ bits of memory. The single lock algorithm is significantly more space efficient than the multiple lock algorithms. Rao and Kumar's algorithm requires less space than the new algorithm ($\Theta(M)$ for the former compared to $\Theta(M \log P)$ for the latter). In practice, however, bit packing results in false sharing in cache-coherent systems, and should therefore be avoided. If overhead bits for different nodes occupy different memory words, and if the number of processes operating on the heap does not exceed $2^n - 2$, where n is the number of bits per memory word, then the space overhead of the new algorithm is the same as that of Rao and Kumar's algorithm, except for three words for the bit-reverse counter.

Figs. 1 and 2 present pseudo code for the insert and delete operations 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.

Bit reversals can easily be calculated in $O(n)$ time, where n is the number of bits to be reversed. For long sequences of increments only or decrements only, we can improve this bound to an amortized time of $O(1)$ by remembering the high-order bit (see Fig. 3). Alternating increments and decrements may still require $O(n)$ time.

3. Experimental methodology

We use 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, in order to determine if it suffices to improve performance. For mutual exclusion we used test-and-test-and-set locks with backoff using the MIPS R4000 `load-linked`

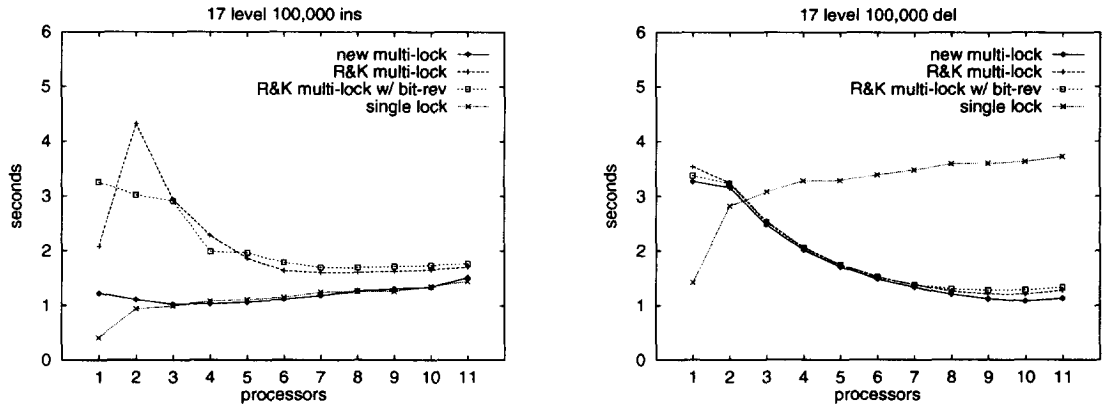

Fig. 4. Performance results for (a) 100,000 insertions and (b) 100,000 deletions.

and `store-conditional` instructions. On small-scale multiprocessors like the Challenge, these locks have low overhead compared to other more scalable locks [4].

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 that 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 are equally loaded. We studied the performance under 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 experiments we used workloads of around 100,000 to 200,000 heap operations. Experiments with smaller workloads are too fast to time. Inserted item priorities were chosen from a uniform distribution on the domain of 32-bit integers.

The sources for all the algorithms were carefully hand-optimized. For example in the multiple-lock algorithms we changed the data layout to reduce the effect of false sharing. This was not applied to the single lock algorithm as it does not support concurrent access; aligning data to cache lines would only increase the total number of cache misses. We believe we have implemented each algorithm as well as is reasonably possible, resulting in fair comparisons.[3]

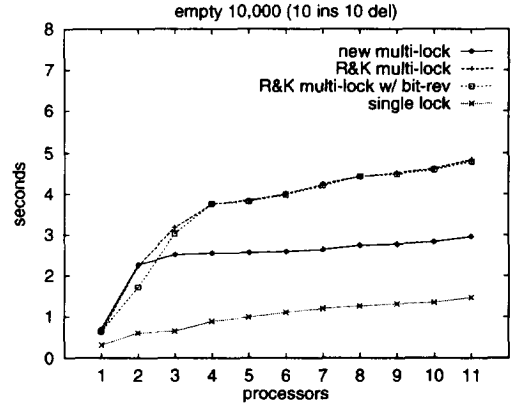

Fig. 5. Performance results for 10,000 sets of 10 insertions and 10 deletions on an empty heap.

Figs. 4(a) and 4(b) show the time taken to perform 100,000 insertions and deletions on a heap with 17 full levels. Fig. 5 shows the time taken to perform 10,000 sets of 10 insertions and 10 deletions on an empty heap. Figs. 6(a) and 6(b) show the time taken to perform 100,000 insert/delete pairs on a 7-level-full heap and a 17-level-full heap.

In the case of insertions only (Fig. 4(a)), 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

[3] The programs are accessible at ftp://ftp.cs.rochester.edu/pub/packages/concurrent_heap

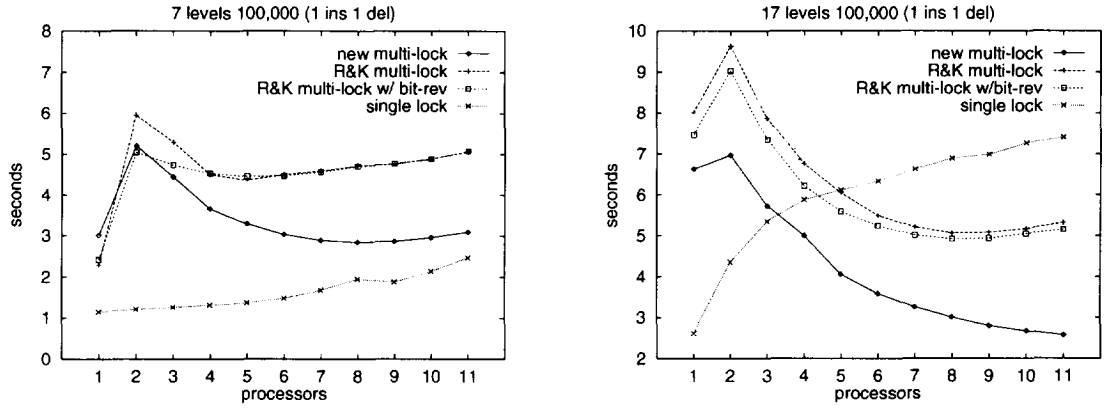

Fig. 6. Performance results for (a) 100,000 insert/delete pairs on a 7-level-full heap and (b) 100,000 insert/delete pairs on a 17-level-full heap.

Kumar's algorithm), and most inserted items settle in the two bottom-most levels of the heap. 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 help much.

In the case of deletions only (Fig. 4(b)), the multi-lock algorithms outperform the single-lock algorithm. This is because most deletions have to traverse the whole height of the tree and may not traverse the same path each time. As a result, the concurrency permitted in the multi-lock algorithms is higher and outweighs the overhead of locking, since there is little contention along the paths. Deletions in the new algorithm proceed top-down, similar to deletions in Rao and Kumar's algorithm; therefore the two algorithms display similar performance.

In the case of alternating insertions and deletions on an initially empty heap (Fig. 5), the height of the heap is very small. The single-lock algorithm outperforms the other algorithms because it has low overhead and there is little opportunity for the multi-lock algorithms to exploit concurrency. Comparing the new algorithm with that of Rao and Kumar, we find that the new algorithm yields better performance as it suffers less from contention on the topmost nodes of the heap. Note that after several insert/delete cycles, the items remaining in the heap tend to have low priorities, 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 due more to reduced contention for the topmost nodes of the tree (due to opposite directions for insertion and deletion) than to shorter traversals.

In the case of alternating insertions and deletions on a 7-level-full heap (Fig. 6(a)), the heap height remains almost constant. The single-lock algorithm outperforms the others due to 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 (Fig. 6(b)), the larger heap height makes concurrency, rather than locking overhead, the dominant factor in performance. The multi-lock algorithms therefore perform better than the single-lock algorithm. As in the case of the empty and 7-level-full heaps, new insertions tend to have higher priorities than the items already in the heap, and tend to 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 ac-

cesses without forcing insertions to proceed top-down [7], 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. Empirical results, comparing the new algorithm, the single-lock algorithm, and Rao and Kumar's top-down insertion algorithm [7] on an SGI Challenge, show that the new algorithm provides reasonable performance on small heaps, and significantly superior performance on large heaps under high levels of contention.

Acknowledgments

We thank Greg Andrews and the anonymous referees for their useful comments that improved both the quality and conciseness of this paper. 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).

References

[1] J. Biswas and J.C. Browne, Simultaneous update of priority structures, in: *Proc. 1987 Internat. Conf. on Parallel Processing*, St. Charles, IL (1987) 124–131.

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

[3] D.W. Jones, Concurrent operations on priority queues, *Comm. ACM* **32** (1) (1989) 132–137.

[4] J.M. Mellor-Crummey and M.L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Trans. Comput. Systems* **9** (1) (1991) 21–65.

[5] J. Mohan, Experience with two parallel programs solving the travelling salesman problem, in: *Proc. 1983 Internat. Conf. on Parallel Processing* (1983) 191–193.

[6] M.J. Quinn and N. Deo, Parallel graph algorithms, *ACM Comput. Surveys* **16** (3) (1984) 319–348.

[7] V.N. Rao and V. Kumar, Concurrent access of priority queues, *IEEE Trans. Comput.* **37** (12) (1988) 1657–1665.