Lock Bypassing: An Efficient Algorithm For Concurrently Accessing Priority Heaps

Yong Yan
HAL Computer Systems, Inc.
Campbell, California 95008
and
Xiaodong Zhang
Department of Computer Science
P. O. Box 8795
College of William & Mary
Williamsburg, Virginia 23187-8795

The heap representation of priority queues is one of the most widely used data structures in the design of parallel algorithms. Efficiently exploiting the parallelism of a priority heap has significant influence on the efficiency of a wide range of applications and parallel algorithms. In this paper, we propose an aggressive priority heap operating algorithm, called the lock bypassing algorithm (LB) on shared memory systems. This algorithm minimizes interference of concurrent enqueue and dequeue operations on priority heaps while keeping the strict priority property: a dequeue always returns the minimum of a heap. The unique idea that distinguishes the LB algorithm from previous concurrent algorithms on priority heaps is the use of locking-on-demand and lock-bypassing techniques to minimize locking granularity and to maximize parallelism. The LB algorithm allows an enqueue operation to bypass the locks along its insertion path until it reaches a possible place where it can perform the insertion. Meanwhile a dequeue operation also makes its locking range and locking period as small as possible by carefully tuning its execution procedure. The LB algorithm is shown to be correct in terms of deadlock freedom and heap consistency. The performance of the LB algorithm was evaluated analytically and experimentally in comparison with previous algorithms. Analytical results show that the LB algorithm reduces by half the number of locks waited for in the worst case by previous algorithms. The experimental results show that the LB algorithm outperforms previously designed algorithms by up to a factor of 2 in hold time.

General Terms: Aggressive Locking, Parallel Algorithm, Performance Evaluation, Priority Heap, Shared-memory System

---

This work is supported in part by the National Science Foundation under grants CCR-9400719 and CCR-9812187, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215 and by the Office of Naval Research under grant ONR-95-1-1239

1. INTRODUCTION

The heap is an important data structure used as a priority queue in many application areas, such as in operating system scheduling, discrete event simulation, graph search and the branch-and-bound method on shared-memory multiprocessors [Biswas and Browne 1993; Cormen et al. 1990; Jones 1986; Olariu and Wen 1991; Ranade et al. 1994]. In these applications, each processor repeatedly performs in a "access-think" cycle. Every processor executes its current event or subproblem, then accesses the shared priority heap to insert new events or subproblems generated by the processor. The shared heap is used to provide the most important events or subproblems for a set of processors to solve in parallel. Consistency and efficiency are two important characteristics to assess a concurrent operation on priority heaps. Consistency requires that operations should maintain the priority property of a heap, which is usually implemented with the help of some locking techniques. An efficient algorithm requires that a priority queue is operated in parallel by as many processors as possible, whose performance is mainly affected by the locking technique employed. Hence, the challenging issue is how to maximize the parallelism of accessing a priority heap while maintaining the consistency of the heap.

For a concurrent priority heap, enqueue (or insertion) and dequeue (or deletion) are two critical operations that need to be parallelized. An enqueue operation is responsible for putting a new data item at an appropriate place in the priority heap. A dequeue operation always removes the minimum of a minimum heap or the maximum of a maximum heap, and reconstructs the heap. To allow for more processors to operate on a heap, it is necessary to reduce the interference among operations as much as possible. In the ordinary sequential heap algorithm, dequeue operations manipulate the heap from top to bottom, while enqueue operations manipulate it from bottom to top. The reverse of the manipulation directions between enqueues and dequeues prevent both enqueues and dequeues from being parallelized.

To solve this problem, Biswas and Browne present a scheme in [Biswas and Browne 1987]. Their scheme decomposes an enqueue or a dequeue into a sequence of update steps at different levels of a heap, then schedules these steps to execute in parallel. However, this scheme causes substantial overhead, and even performs worse than the sequential access heap unless the heap size is very large. In [Rao and Kumar 1988], Rao and Kumar propose a top-to-bottom sequential insertion algorithm, then propose a concurrent algorithm for manipulating the priority heap. In their algorithm, both enqueues and dequeues start at the root of a heap (a heap is equivalent to a balanced binary tree), lock the current node, operate on the current node if necessary, and then select the next node to trace down the heap tree until the operations finish at an appropriate place to make the heap consistent. Although Rao and Kumar's algorithm, termed the RK algorithm hereafter, it is brief only needs to lock one node for an enqueue operation and three nodes for a dequeue operation, the lock granularity actually is the whole subtree rooted at the topmost node locked by an operation. The lock granularity may be too large to obtain good performance in parallel execution. Only those operations on different branches of the heap tree can proceed in parallel. In order to keep enqueues operating on

different subtrees of a heap tree, Ayani [Ayani 1991] proposes an algorithm to do this based on left and right directing rule. The proposed algorithm is called the LR-algorithm. But the LR-algorithm may not improve the RK algorithm much in practice, because dequeue operations still proceed randomly, depending on the current heap. The random order of the dequeue operations nullifies the benefit of separating parallel enqueue operations.

In this paper, rather than separate parallel operations on a heap, we propose more aggressive techniques to reduce locking granularity, hence reducing the interference among heap operations. Instead of blindly locking the current node, an enqueue operation exams the current node first, and then locks it when it is necessary. In this way, enqueue operations can bypass other locked nodes and go directly to the possible place where new data can be inserted. For a dequeue operation, it does not put the key gotten from one bottom node into the heap until the dequeue operation visits the last node on its heap tree walking path. The dequeue operation uses duplicated keys to eliminate unnecessary blocking on enqueue operations. The correctness of our LB algorithm is proved in terms of heap consistency and deadlock freedom. The proposed new algorithm, termed the LB (lock bypassing) algorithm, was, analytically and experimentally, evaluated in comparison with the LR algorithm and the RK algorithm. Analytical results show that the LB algorithm can reduce by half the numbers of locks waited for, in the worst case, by a hold operation. A hold operation consists of a pair of dequeue and enqueue operations. Experiments evaluated the three algorithms with respect to different thinking delay and heap data with different distributions. The experimental results showed that the average hold operation time of the LB algorithm increases significantly slower than that of the RK algorithm and that of the LR algorithm as the number of processors increases. When the number of processors increases up to 16, the LB algorithm outperforms the RK algorithm by 50% and the LR algorithm by 40% in hold operation time.

Section 2 reviews the basic characteristics of a priority heap and its sequential dequeue and enqueue operations. The RK algorithm and the LR-algorithm are reviewed in Section 2. We propose a more aggressive locking-based algorithm in Section 3. Its correctness is proven in Section 3. In Section 4, the performance of the LB algorithm is studied analytically. Section 5 experimentally evaluates the LB algorithm, the LR algorithm, and the RK algorithm on a cache coherent shared-memory system. Conclusions are drawn in Section 5.

2. PRELIMINARIES

2.1 Priority Heap

A priority heap [Cormen et al. 1990] is a complete binary tree with the priority property that the key value (hereafter called as the key or node key) at any node is not larger than (for a minimum heap) or not smaller than (for a maximum heap) the keys of its child nodes (if they exist). This is defined as the *consistency property* of a heap. In this paper we focus on the minimum heap (the ideas can also be used for other types of heaps). We maintain a *strict priority property*, which requires that a dequeue operation always return the current minimal data item in the heap.

An array $A[0 : n]$ is used to implement a heap. For a heap with n elements

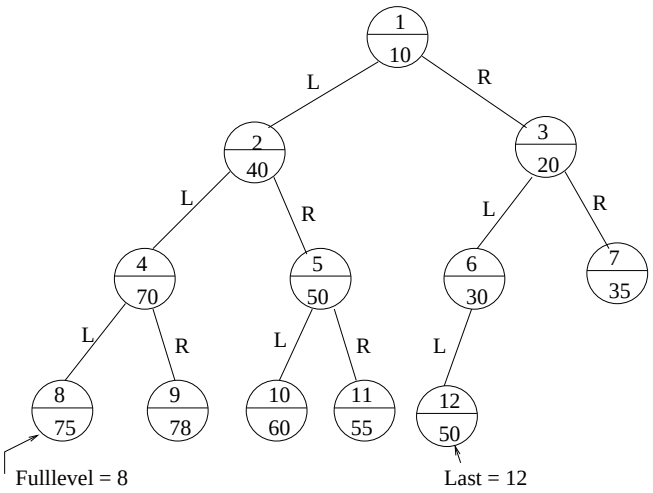


Fig. 1. **A minimum heap with 12 keys**. The upper half of each circle represents the node number, and the lower half represents the key value. The variable *Fulllevel* records the index of leftmost keys at the bottom level. The variable *Last* records the index of the last key in the heap.

as shown in Figure 1, an array of size n is used to hold the heap, where $node_1$ occupies location 1 and $node_i$ occupies location i. The left child of node i is found at location $2 * i$, and the right child of node i is found at location $2 * i + 1$. The parent of node i is $\lfloor i/2 \rfloor$. $A[0]$ is initialized to be MAXINT, the maximal integer value in a given machine, which will be used to simplify our LB algorithm. In order to focus on evaluating concurrent dequeue and enqueue operations, we initially allocate a sufficiently large array to hold our heap and we initialize empty locations with MAXINT.

2.2 Sequential Enqueue and Dequeue Operations

Enqueue and dequeue are two important operations supported for a heap. The enqueue operation inserts a new key in the heap and the dequeue removes the smallest key in the heap and returns it. The heap should maintain the priority property after an enqueue or a dequeue operation.

A dequeue operation first removes and returns the smallest key at the root node of the heap. It moves the last key pointed to by *Last* onto the root and compares it with the smallest child of the deleted root node. If the smallest child holds a key smaller than the current root, the new key of the root is exchanged with the smallest key. This procedure repeatedly goes down along the heap tree until reaching one descendant whose key is smaller than the keys of its children.

In contrast, a traditional enqueue operation works from the bottom to the top on a heap. To insert a new key into the heap, an enqueue operation initially adds the new key to the first empty node, i.e., node $Last+1$. It works up the tree exchanging with the parent node if the parent node has a larger key until the new key is placed at a node whose parent has a smaller key. Because a traditional sequential enqueue operation works in the opposite direction as a dequeue operation, both operations tend to deadlock in parallel execution. An alternative enqueue that can work from

```
PUBLIC struct node {
    int key;
} Heap[LEN]; /* LEN is large enough to hold a heap. */
int Fulllevel: /* index of the leftmost node on the bottom level. */
int Last; /* the index of the last node. */
Enqueue(nkey)
{     int i, j, k;
    Last = Last + 1;
    if (Last ≥ Fullevel * 2)
        Fulllevel = Last;
    j = Fulllevel/2; /* used to determine insertion path */
    i = Last - Fulllevel; /* the displacement of target */
    k = 1; /* current position in the insertion path */
    while (j ≠ 0){
        if (Heap[k].key > nkey)
            exchange(nkey, Heap[k].key);
        if (i ≥ j) {
            k = rson(k);
            i = i - j;
        }
        else
            k = lson(k);
        j = j/2;
    }
    Heap[k].key = nkey;
}
```

Fig. 2. **Sequential top-to-bottom enqueue algorithm**: `rson(k)` and `lson(k)` represent the right child and left child of node `k`: `exchange(key1, key2)` exchanges `key1` with `key2`.

top to bottom has been proposed by Rao and Kumar [Rao and Kumar 1988]. This algorithm adds a new key to the first empty node, termed the target node, then compares each ancestor node downwards on the insertion path from the root to the target node and exchanges the key at one ancestor node with the target node if the key at the target node is smaller than the key at the ancestor node. In Figure 1, the variable *Fulllevel* is used to determine the insertion path. This enqueue operation is helpful for implementing concurrent heap accesses. Hereafter, the enqueue operation refers to the top-to-bottom enqueue operation. The algorithm for the top-to-bottom enqueue operation is listed in Figure 2.

2.3 The RK Algorithm and the LR-algorithm

When processors access a shared heap in parallel, a lock should be used to maintain the consistency of the heap. The most conservative locking strategy is to lock the whole heap as a single data structure. This strategy just implements a serial-access scheme, resulting in the worst possible performance in parallel execution. Hence, reducing the locking granularity is the central task in parallelizing enqueue and dequeue operations.

In the RK algorithm, a window locking strategy is employed. A window is a portion of the heap. For an enqueue operation, it first locks the root of the heap, checking whether an exchange operation should be done between the new key and

the locked node. Then, it locks the next node along the insertion path from the root to the target, releases the current locked node, and makes a comparison between the two keys. This procedure will be repeated for each node on the insertion path. For a dequeue operation, it locks the root of the heap, removes the key at the root as a return key, moves the last node onto the root, and locks the two children of the root. If the root has a larger key than the minimum child (that has the smallest key), the key at the root is exchanged with the key at the minimum child and the locks on the root and the maximum child are released. The procedure is repeated from the minimum child until a descendant that has a larger key than its children nodes is met. The window size is one node for an enqueue operation and three nodes for a dequeue operation. Because no operation can bypass a locked node, the locking granularity of a lock is actually the whole subtree rooted at the locked node. For example, when an operation has locked the root of the heap, all the other incoming operations must wait at the root. When multiple enqueues operate on different parts of the heap, they must traverse the overlapping part of their insertion path in the order of their visits to the root. Hence, two operations can operate on the heap using maximal parallelism if and only if both operations have a smaller common part between their operating paths. In addition, the RK algorithm uses a state variable to resolve the interaction between an enqueue and a dequeue. For example, if the most recent enqueue operation is still in progress, a dequeue operation must wait for the last leaf node to have a non-transient key because what key will be finally inserted into a bottom node by an enqueue can only be determined when the enqueue opertion finishes its insertion procedure.

The LR-algorithm [Ayani 1991] is motivated by the possibility of directing enqueue operations to operate along different insertion paths as much as possible. When an enqueue operation is working on the left subtree of a heap the next enqueue operation is directed to work on the right subtree of the heap. The LR-algorithm lets multiple enqueue operations work on different parts of the heap as early as possible. For an enqueue operation, if the bottom level has q nodes, the new key should be added to the empty node $Fulllevel + b_1 b_2 \cdots b_m$ (here $b_m b_{m-1} \cdots b_1$ is the binary representation of $q-1$ and $m = \log(Fulllevel)$ is the depth of the bottom level and $Fulllevel$ is the index of the leftmost node at the bottom level.). The LR-algorithm uses a more effective approach to exploit parallelism in insertion operations. Because the working path of a dequeue operation depends on the heap, which cannot be pre-determined, the same dequeue algorithm as the algorithm is used in the LR-algorithm. However, in practice, enqueue operations interleave with dequeue operations in a heap. Because the working path of a dequeue operation depends on the heap, which cannot be pre-determined, only ordering enqueue operations can fail to guarantee a significant performance improvement over the RK algorithm. Our experimental results, reported in a later section, show that the LR-algorithm and the RK algorithm have no major performance difference.

3. LOCK BYPASSING ALGORITHM (LB ALGORITHM)

In this section the design motivation and idea of our lock bypassing algorithm are described. Then the actual algorithm is presented.

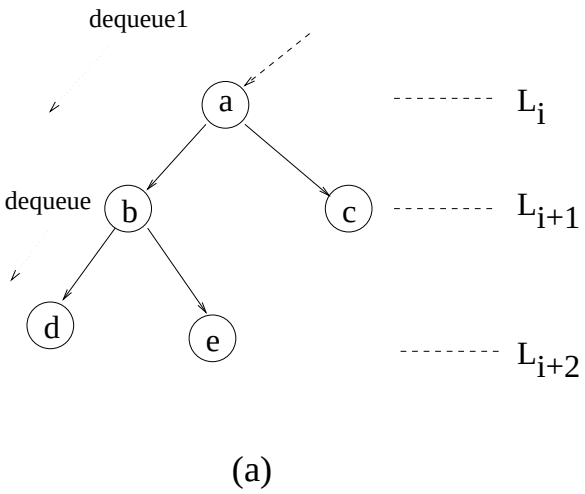


Fig. 3. (a). A dequeue follows another dequeue. (b) A dequeue follows an enqueue.

3.1 Motivation and algorithmic idea

In the design of a concurrent algorithm for accessing a priority heap, each processor actually executes traditional sequential enqueue and dequeue algorithms. The major work in designing a concurrent algorithm is to use lock primitives to coordinate the execution of multiple processors so that heap consistency can be guaranteed while high performance is being pursued. We let each processor execute the top-to-bottom enqueue and dequeue algorithms as described in the previous section. Here, we propose an aggressive way to use locks so that significantly more parallelism can be exploited while consistency is maintained.

The priority heap has a balanced tree structure. Its strict priority property requires that a dequeue operation always visits the root of a heap first to grab the minimal data in the heap. Although enqueue operations can be inverted to walk on a heap tree downwards as in the RK algorithm, the interference among parallel enqueues and dequeues are still serious. Locks are the necessary primitives used to maintain the consistency of a heap. Intuitively, when a dequeue or enqueue operation operates on a node, it is necessary for the operation to lock related nodes to guarantee heap consistency. However, it is unnecessary for a locked node to block incoming heap operations. If two operations do not need to operate on the same part of the heap they should not block each other. For example, in Figure 1, we assume that some operation has locked the root when an enqueue operation with new key of 40 is arriving. This new key is added as the right child, as node 13, of node 6, and will not be exchanged with any node along its insertion path from the root to the target. So the enqueue operation should bypass the lock at the root. An enqueue operation stops at a node if and only if it finds that the new key is smaller than the key of the node. Our LB algorithm is mainly motivated by this lock bypassing idea.

Bypassing locks in a heap really provides a good opportunity to exploit parallelism. Meanwhile, one must be careful with lock bypassing in order to maintain the

consistency of a heap and to avoid deadlock. For the two heap operations, dequeue and enqueue, there are four possibilities for lock bypassing: an enqueue bypasses an enqueue, an enqueue bypasses a dequeue, a dequeue bypasses an enqueue, and a dequeue bypasses a dequeue. Here we further analyze when the lock bypassing can be safely used.

First, because the strict priority property requires that a dequeue always returns the root of a heap, a dequeue operation actually cannot bypass any lock on the root. In addition, a top-to-bottom dequeue operation always first removes the root as the return key, then moves the key of the last node into the root node and sinks the key down the heap tree to a node that has two child nodes with larger keys. So, when two dequeue operations have the same sinking path, as in Figure 3(a) where dequeue1 wants to sink key a down and dequeue2 is sinking key b down along its right child, the late arriving dequeue operation, such as dequeue1, cannot decide which child's key should be chosen as the new key of node a before the dequeue2 operation has decided what is the new key of node b. It is a necessary condition for consistency to preserve the incoming order of dequeue operations on common paths. So, a dequeue operation cannot bypass another dequeue operation.

Secondly, if a dequeue operation meets an enqueue operation along its sinking path, as illustrated in Figure 3(b), the dequeue cannot decide which way to go before the enqueue finishes its comparison between the keys of node b and node e. So the dequeue cannot bypass the enqueue. The reason that a dequeue operation is unable to bypass other operations is that the sinking path of a dequeue operation is dynamically decided, which is affected by the results of other concurrent operations on its sinking path.

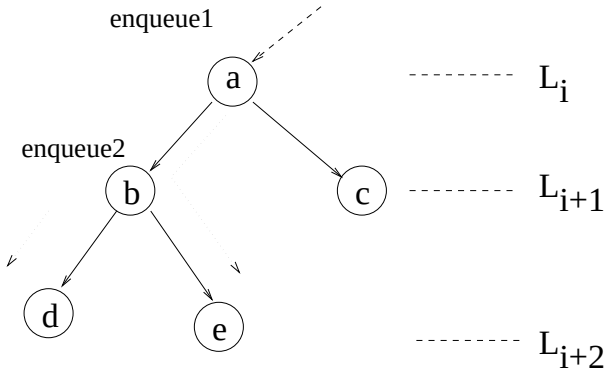
Bypassing can be achieved from enqueue operations. The walking path of an enqueue operation along a heap tree can be predetermined, it is from the root to the next available empty node on the lowest level. What an enqueue operation does is to insert a new key into the empty node first, then exchanges it downwards with some nodes along its walking path if the empty node holds a smaller key than that of the compared node. The necessary and sufficient condition for an enqueue operation to stop sinking and to do an exchange is when it meets a node holding a smaller key than the new key to be inserted. In the third case that an enqueue operation follows another enqueue operations, as described in Figure 4(a) where node b is assumed to be locked by enqueue2 and enqueue1 has walking path $a \rightarrow b \rightarrow e$, enqueue1 can bypass node b if it finds that the new key to be inserted is larger than the key in node b. This bypass is safe even if enqueue2 will exchange its new key with node b. The comparing relation between the new key of the enqueue2 and the key in node b only has the following two results:

—enqueue2's new key is smaller than node b's key: Because enqueue1's key is larger than node b's key, it is also larger than enqueue2's new key. So, node b is certainly not the place for enqueue1's key to be inserted.

—enqueue2's new key is not smaller than node b's key: Node b's key will be changed by enqueue2. Node b is still not the place for enqueue1's key to be inserted.

In addition, if enqueue1 finds it has a smaller key than node b it must stop here to check whether enqueue2 will insert an even smaller key into node b.

The last case we consider is an enqueue operation following a dequeue operation.



(a)

Fig. 4. (a). A enqueue follows another enqueue. (b) A enqueue follows an dequeue.

This is illustrated in Figure 4(b) where the dequeue operation locks node b and the enqueue operation has the walking path $a \rightarrow b \rightarrow e$. If the enqueue finds it has a larger key than node b, it can bypass b and go to node e. When the enqueue still has a larger key than node e, it can confirm that its bypassing node b is safe because the dequeue operation always moves the minimal key in its children (d and e) into node b, which is certainly smaller than the enqueue's key. However, when the enqueue finds that its key is smaller than node e's key, it stops at node e to do an exchange. After the exchange, the enqueue operation must make another comparison back with node b's key, because when node b is the root where it holds the minimum of a heap, the dequeue operation may finish moving the minimal key in its children into b while the enqueue goes to compare with node e. If, in this case, the enqueue holds a key smaller than all other existing keys except in the root, the enqueue should put its key as the new root in node b. Because we allow both the dequeue and the enqueue to proceed in parallel on the overlapped part along their walking paths, the enqueue will put its key in node e, not in root b, so the backing check is necessary for consistency.

The above descriptive analysis has determined when it is illegal to bypass a lock. In the design of our lock bypassing algorithm, the emphasis is on how to introduce locks to preserve the execution order of a dequeue operation from existing operations in a heap and how to allow an enqueue operation to bypass the locks on its walking path in a heap. Exploiting as much parallelism as possible, with the guarantee of heap consistency, is fundamental principle in the use of locks.

3.2 Lock bypassing algorithm

Besides using a lock bypassing technique, our LB algorithm also incorporates some advantages of previous work described in [Ayani 1991] and [Rao and Kumar 1988]. The enqueue algorithm of our LB algorithm is a LR top-to-bottom enqueue algorithm with the implementation of a lock bypassing technique. The dequeue algorithm of the LB algorithm follows the traditional top-to-bottom dequeue algo-

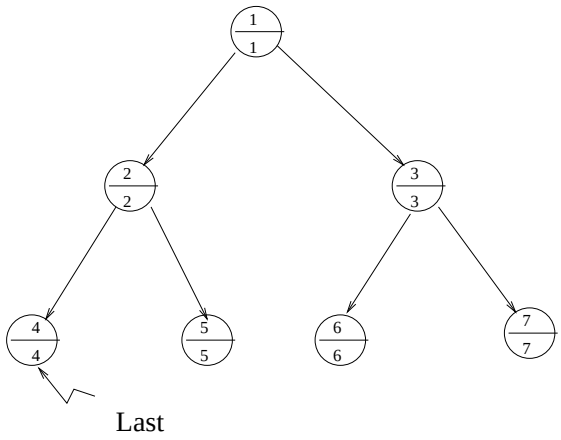


Fig. 5. Insertion order of the top-to-bottom enqueue algorithm (Figure (a)) and LR top-to-bottom enqueue algorithm (Figure (b)). In Figures (a) and (b), each node is represented as a cycle with two labels. The upper half is the node number (or node position), and the lower label represents the enqueue ordering number.

rithm with the incorporation of locks to guarantee the execution order of dequeues relative to other heap operations.

The difference between a LR top-to-bottom enqueue algorithm and the top-to-bottom enqueue algorithm shown in Figure 2 is shown in Figure 5(a) and (b). The insertion order in the top-to-bottom enqueue algorithm is the same as the node order, e.g., the next insertion node in a heap is the node indexed by $Last + 1$. ($Last$ indexes the last node in the heap). However, the insertion order of the LR top-to-bottom enqueue algorithm is different from the node order for the purpose of separating the insertion paths of two consecutive enqueue operations as early as possible. For example in Figure 5(b) where the last enqueued data is placed in node 4, the next enqueue operation should put its data item in node 6, not in node 5 so that the last two enqueue operations only have the common root in their enqueue paths. The calculation of enqueue node positions is described in Section 2.3.

In order to calculate the enqueue position, three global variables are introduced: $Fulllevel$, $Last$ and $level$. Variable $Last$ gives the total number of nodes in a heap. $level$ records the total levels of a heap, which equals $\lfloor \log(Last) \rfloor$. Variable $Fulllevel$ indexes the leftmost node on the bottom level. The 3 global variables, together, control a heap. Because enqueue or dequeue operations initially need to update a heap based on these 3 variables, a lock, named $Lastlock$, is used to enforce mutually exclusive access to these variables. In addition, a lock is introduced for each heap node so that data exchanges are made during the heap tree walking procedures of enqueue and dequeue operations to maintain consistency of the heap. There are two variables associated with each heap node i: $Heap[i].state$ and $Heap[i].sstate$, which are used to coordinate enqueue and dequeue operations.

The state variable $Heap[i].state$ has one of the following values, representing the different states of the node key:

—**PRESENT**: A key exists at the node;

—**PENDING**: An enqueue operation, which will ultimately insert a key at the node, is in progress;

—**ABSENT**: No key is present at the node.

A heap node initially is in the **ABSENT** *state* (an empty node). When an enqueue operation is trying to insert a key in an empty node, called the target node, it constantly compares its new key with that of the nodes along its heap walking path and does an exchange if necessary. Before the enqueue operation reaches the target node, the key of the target is not determined. To prevent a dequeue operation from trying to use the key of the target node during this time, the state variable of the target is set to **PENDING** to inform the dequeue that the key in the target is temporarily not available so that the dequeue can delay itself. When an enqueue inserts a stable new key into the target node, the state variable of the target is set to **PRESENT** so that a dequeue can safely use it.

Although the *state* variable in a node can inform a dequeue when a key in the node is available, a dequeue should wait for the *state* variable to change from **PENDING** to **PRESENT**. This will cause significant idle time for a dequeue operation. When a dequeue operation gets the minimal key in the root of a heap, it rearranges the heap by moving the key in the last node into the root and then sinks the key along the appropriate heap tree branch. Because a dequeue operation does not require a specific key to move into the root, it does not need to wait for the last node to have a stable key. In other words, it does not need to wait for the change of the *state* of the last node from **PENDING** to **PRESENT**. Hence, variable *sstate* in a node is used for a dequeue operation to inform an enqueue that it wants the key to be inserted by the enqueue operation. The *sstate* variable gets one of the following two values:

—**WANTED**: A dequeue operation is waiting for the key;

—**FREE**: No dequeue operation is waiting for the key.

When a dequeue operation is waiting for the key in a node that is going to be inserted by an enqueue, it sets the *sstate* variable to **WANTED**. The enqueue operation always checks the *sstate* variable of its target. When it sees that the *sstate* variable in the target has value of **WANTED**, it stops its enqueue procedure to insert its key into the target so that the dequeue operation can get this key as quickly as possible.

The above data structure setting allow us to present our parallel enqueue and dequeue algorithms. The algorithms are given in Figure 6 and Figure 7 where `rson(k) = 2k + 1` gives the right child node of node `k`, `lson(k) = 2k` gives the left child node of node `k`, `MAX(i)` and `MIN(i)` calculate respectively the indexes of the larger child node and smaller of node `i`. `mutex_lock` and `mutex_unlock` are lock and unlock primitives. `mutex_trylock` is a nonblocking lock primitive (which is provided in most commercial thread libraries in slightly different formats), which tries to get the lock. If it fails, a non-zero error is returned; otherwise, zero is returned for success.

In the enqueue algorithm, an enqueue operation first competes for the lock *Lastlock*, trying to get the right to update the heap. When it gets the lock, it

```
PUBLIC struct heapnode {
    int key, state, sstate;
}Heap[LEN]; /* Define a heap */
int Fulllevel, Last, Level = -1;
mutex_t Mutex[MAX_HEAP_SIZE], Locklast;
void Parallel_enqueue(float nkey)
{       int target, j, k, offset, i = 0;
1        mutex_lock(&Locklast);
2        if ((++Last) >= (Fulllevel<<1))
3            { Fulllevel = Last; Level++;}
4        j = Fulllevel/2; offset = Last - Fulllevel;
5        for (k=0; k<Level; k++){ /* compute position of last */
6            i = (i<<1)|(i&0x0001); /* element */
7            offset = offset>>1;
8        }
9        target = Fulllevel + i;
10       Heap[target].key = nkey;
11       Heap[target].state = PENDING; Heap[target].sstate = FREE ;
12       mutex_unlock(&Locklast);
13       k = 1;
14       while (j != 0){
15           if (Heap[target].sstate == WANTED) break;
16           if (nkey < Heap[k].key){
17               while ((t = mutex_trylock(Mutex+k))!= 0&&Heap[target].sstate != WANTED);
18               if (Heap[target].sstate == WANTED){
19                   if (t == 0) mutex_unlock(Mutex+k);
20                   break;
21               }
22               if (nkey < Heap[k].key)
23                   if (nkey >= Heap[k/2].key || k == 1){
24                       exchan(&nkey,&Heap[k].key);
25                       mutex_unlock(Mutex+k);
26                   }else{ mutex_unlock(Mutex+k);
27                       k /= 2; j *= 2;
28                       continue;
29                   }
30               else mutex_unlock(Mutex+k);
31           }
32           if (i >= j){
33               k = rson(k); i -= j;
34           }else k = lson(k);
35           j /= 2;
36       }
37       Heap[target].key = nkey; Heap[target].state = PRESENT;
}
```

Fig. 6.

```
PUBLIC struct heapnode {
    int key, state, sstate;
}Heap[LEN]; /* Define a heap */
int Fulllevel, Last, Level = -1;
mutex_t Mutex[MAX_HEAP_SIZE], Locklast;
int Parallel_dequeue()
{       float least, newkey;
        int i, j, k, ikk, offset = 0;
1       mutex_lock(&Locklast);
2       if ( last == 0){ /* empty heap * /
3           mutex_unlock(&Locklast); return(-1);
4       }
5       j = Last - Fulllevel;
6       for (m=0; m<Level; m++){
7           offset = (offset<<1)|(j&0x0001);
8           j = j>>1;
9       }
10      j = offset + Fulllevel; /* Find the last inserted node */
11      if (--Last < Fulllevel){
12          Fulllevel = Fulllevel<<1;
13          Level--;
14      }
15      Heap[j].sstate = WANTED;
16      while (j != 1 && Heap[j].state == PENDING);
17      least = Heap[1].key;
18      newkey = Heap[j].key
19      Heap[j].sstate = FREE;
20      Heap[j].key = MAXINT; Heap[j].state = ABSENT;
21      if (j == 1){
22          mutex_unlock(&Locklast);
23          return(least);
24      }
25      mutex_unlock(&Locklast);
26      i = 1;
27      mutex_lock(Mutex+i);
28      mutex_lock(Mutex+lson(i)); mutex_lock(Mutex+rson(i));
29      k = MIN(i); ikk = MAX(i);
30      while (newkey > Heap[k].key){
31          exchan(&Heap[i].key, &Heap[k].key);
32          mutex_unlock(Mutex+ikk); mutex_unlock(Mutex+i);
33          i = k;
34          mutex_lock(Mutex+lson(i)); mutex_lock(Mutex+rson(i));
35          k = MIN(i); ikk = MAX(i);
36      }
37      exchan(&newkey, &Heap[i].key);
38      mutex_unlock(Mutex+i); mutex_unlock(Mutex+lson(i)); mutex_unlock(Mutex+rson(i));
39      return(least);
}
```

Fig. 7.

calculates the target node where the new is to be inserted in statements 2 to 9. It sets the *state* variable of the target node to **PENDING** to inform incoming dequeue operations, and sets the *sstate* variable of the target node to **FREE**. Then it releases the *Lastlock* lock. Starting from statement 14, instead of blindly locking each node step-by-step on its insertion path, the enqueue operation checks whether a dequeue operation is waiting for its key. If so, it breaks to statement 37 to set its key into the target node directly and change the *state* variable of the target to **PRESENT**. If not, it compares its new key with the root. The procedure is repeated to bypass all locks along its insertion path from the root to the target node until the enqueue operation at statement 16 meets a node, denoted by j, with a larger key value than its new key. Then it first enters a spinning test loop at statement 17 to check whether it can hold the lock on node j, or a dequeue operation is waiting for its key. If it finds a waiting dequeue operation, it releases its lock if it has gotten it and breaks to statement 37. Otherwise it rechecks whether its new key is smaller than that of the locked node j at statement 22, because it is possible for other enqueue operations holding larger keys to first lock node j and change the key in node j during the period from its first check at statement 16 to its holding the lock of node j at the spinning test loop. If not, it will continue to walk down the heap along its insertion path. Otherwise, it checks with the parent node before it can exchange its new key with node j because it is possible its parent has a key temporarily placed by a dequeue operation which does not satisfy heap consistency. If the enqueue operation finds that the parent node holds a smaller key than its new key, the enqueue operation releases the current node and goes back to the parent node to repeat its insertion procedure. Otherwise, it exchanges its new key with the key of node j, and then releases the locks and repeats this procedure down the insertion path. When the enqueue procedure reaches the target node at the bottom level, it changes the state of the target node to **PRESENT** so that the waiting dequeue operation can proceed. From the enqueue algorithm, the following property is given.

PROPERTY 1. *An enqueue operation can change the key in a heap node if and only if it holds the Lastlock lock (from statements 1 to 12) or the lock on the heap node (at statement 24), or the heap node is the target node of the enqueue operation with* **PENDING** *state (at statement 37).*

In the dequeue algorithm, a dequeue operation also first competes for the lock *Lastlock*. When it gets the lock, it calculates the last heap node and adjusts the heap control parameters in statements 2 to 14. It sets the *sstate* variable of the last heap node to **WANTED**, then enters a spinning test loop at statement 16 to check whether the last heap has a stable key. It gets out of the spinning loop when the *state* variable of the last heap node has value **PRESENT**. Then, it copies the minimal key in the root to the variable *least* and removes the key in the last heap node into variable *newkey* as its new key, so that its heap tree walking procedure gets close to an enqueue algorithm. The dequeue operation gets into its heap tree walking procedure at statement 27, which starts from the heap root. Along the heap tree walking path, a dequeue operation first acquires the locks on the current node and its two children nodes. It compares its new key with the smallest key of its children nodes. If the dequeue operation has a larger new key, it walks down the

heap tree along the child node with smallest key. This procedure is repeated until the dequeue operation reaches a node where its two child nodes have larger keys than that of the dequeue operation. Then the new key is finally inserted into the heap. From the dequeue algorithm, the following two properties can be derived.

PROPERTY 2. *A dequeue operation can remove the key in a heap node if and only if it holds the Lastlock lock (at statement 17 or 18).*

PROPERTY 3. *A dequeue operation can change the key in a heap node if and only if it holds the locks on the node and its two child nodes (at statement 31 or 37).*

The major point that distinguishes our dequeue algorithm from the previous dequeue algorithm in [Rao and Kumar 1988] is that instead of directly removing the key in the last heap node into the root and sinking it down the heap tree, our dequeue algorithm removes the key in the last heap node in a temporary variable as its new key and compares the new key with the children of a current node. The new key of a dequeue operation occurs in the heap only when the dequeue operation finds an appropriate insertion node. This approach helps exploit parallelism in the lock bypassing procedure of enqueue operations.

For example, in Figure 1 where a dequeue operation is going to return key 10 in the root and an enqueue operation with new key of 37 follows the dequeue operation, if the dequeue operation directly removes key 50 in the last heap node 12 into the root, the enqueue operation is always blocked by the dequeue operation when it walks down the tree along path $node_1 \rightarrow node_3 \rightarrow node_6$ because the enqueue operation always finds that the keys locked by the dequeue operation are larger than its key 37. However, if the key of the last node is not directly removed into another heap node temporarily, the enqueue can bypass any nodes locked by the dequeue operation provided that the new key 50 of the dequeue has not been inserted in the heap. This helps the enqueue operation to arrive at its insertion position, node 12, quickly.

3.3 Correctness analysis of LB algorithm

In order to avoid the intricacy of formal proof methods, such as Petri Net and temporal logic, we show the correctness of our parallel dequeue and enqueue algorithm by using more intuitive arguments.

In the heap tree walking procedure of the dequeue operation presented in the example of the last section, the current node locked by it has an invalid key. For example, when the dequeue operation locks the root, it copies out the key of the root as its return key. Although the root still has its original key 10, this key is actually a deleted key. Moreover, when the dequeue operation walks down to node 3 from the root, it copies the key of node 3 into the root. This time, the key in node 3 is a duplicate of the root key. When the dequeue inserts its new key into an appropriate node, there are no invalid keys existing in the heap. The duplicated keys and invalid key facilitate the exploitation of parallelism by delaying the insertion of the new key of a dequeue operation as late as possible since the new key coming from the bottom level usually has a large key. Invalid keys only occur in those heap nodes locked by the dequeue operation, which cannot be returned as a valid key by another dequeue operation (by Property 2). One concern that should

be addressed is whether the existence of invalid keys would violate the consistency of a heap because the key to be inserted by a dequeue operation is not seen by any other bypassing enqueues until the key is inserted into a final, appropriate heap node. Owing to this, it is possible for an enqueue operation holding a smaller key to have bypassed a dequeue operation holding a larger key. The following statements refer to the example above (see Figure 1). When the dequeue operation with new key of 50 finishes poping node 6's key into node 3 and releasing the locks on node 3 and node 7, if the enqueue operation with a key of 37 following the dequeue operation has bypassed node 6 because its key is larger than the invalid key 30 in node 6, the enqueue is going to insert its key into node 12. By Property 1 and Property 2, it is known that the dequeue operation cannot insert its new key 50 into node 6 concurrently with the insertion of the new key of the enqueue into node 12. If the dequeue finishes its insertion in node 6 before the enqueue does, the enqueue operation will find the existence of a larger key in its parent by one-step back-checking so that it restarts its heap tree walking procedure from its parent node. Otherwise, the dequeue operation will be responsible for moving the new key 37 of node 12 into node 6, and putting its key 50 into node 12. This guarantees the consistency of the heap.

The above argument shows that the bypassing of enqueue operations over dequeue operations is safe in our enqueue and dequeue algorithms. For the bypassing of enqueues over enqueues, it is impossible for an enqueue with smaller key to bypass a lock held by an enqueue with larger key because an enqueue stops at a heap node to acquire its lock if and only if the enqueue finds a larger key in the held node. In this case, the following enqueue operation with a smaller key also must stop at this point. So, the bypassing of enqueues over enqueues is safe. Moreover, properties 1, 2, and 3, show that the update on each heap node is atomic, i.e., only one operation can update a heap node at a time. From these points, the following conclusion can be derived.

CONCLUSION 1. *The consistency of a heap is guaranteed by our parallel dequeue and enqueue algorithms.*

Besides consistency, deadlock freedom is another requirement for the correctness of our concurrent enqueue and dequeue algorithms. For any real application, the number of events generated is finite. So, we can assume the total number of enqueue and dequeue operations to be finite.

For the parallel enqueue algorithm given in Figure 6, when it grabs lock *Lastlock* at statement 1, it will release it in a finite time at statement 12. So enqueue operations do not block one another on the *Lastlock* lock. The following conclusion can be derived.

CONCLUSION 2. *An enqueue operation is indefinitely blocked by the Lastlock lock only when the Lastlock lock is indefinitely held by a dequeue operation.*

Among statements 13 to 37 in Figure 6, the spinning test loop at statement 17 is another possible place for an enqueue to be blocked. An enqueue operation spins indefinitely at statement 17 only when no dequeue operation is waiting for its key and it cannot get its requested lock. Here, we assume an enqueue always can get out of the spinning loop. (Later, we will show that this is true by considering a

dequeue operation.) If an enqueue gets out of the spinning loop by finding a waiting dequeue operation, it will jump directly to the end and finish its enqueue procedure. So, we only need to consider the situation that an enqueue operation gets out of the spinning loop by acquiring its requested lock and no waiting dequeue is found. From statements 22 to 30 in Figure 6, we can see that the enqueue operation finishes all its operations and releases its held lock in a finite time. Combining this with its execution behavior on the *Lastlock* lock, we can conclude the following.

CONCLUSION 3. *An enqueue operation never holds a lock indefinitely.*

When an enqueue gets its requested lock and reaches statement 22, it may walk down the heap tree or walk back to its parent node. If an enqueue always walks down the heap, it finally reaches its target insertion node and finishes its insertion procedure because a heap has a finite number of heap nodes. When it walks back to its parent node, it must have found that its parent has held a larger key after the enqueue had bypassed it at statement 23. Assume the current node locked by the enqueue operation is k, its parent node is $Parent(j)$ ($= \lfloor j/2 \rfloor$) and $newkey$ is the new key in the enqueue operation. Walking back has the following condition:

$$(newkey < heap[j].key) \wedge (newkey < heap[Parent(j)].key). \quad (1)$$

This can happen only when a dequeue was going to change the key of the parent while the enqueue operation was bypassing the parent node because an enqueue only puts a smaller key in a node when it is going to change the key of the node. However, a dequeue operation never walks back to its parent in its heap tree walking procedure. So, an enqueue operation only does backward walks for a finite number of times because the total number of dequeue operations is finite. Finally, an enqueue operation will walk down the heap to reach its target insertion node to finish the enqueue procedure. Combining the above analysis, we can derive another important conclusion as follows.

CONCLUSION 4. *If an enqueue operation can acquire the Lastlock lock at statement 1 and locks at the spinning loop of statement 17 in a finite time, it will finish its enqueue procedure in a finite time.*

Now we analyze the dequeue algorithm (shown in Figure 7). A dequeue operation enters the global heap updating section by acquiring the *Lastlock* lock at statement 1. If it can get out of the spinning loop at statement 16 in a finite time, it releases the *Lastlock* lock in a finite time. A dequeue operation is waiting at the spinning loop when the key to be removed is being held by an enqueue operation which must be executing the enqueue algorithm among statements 14 to 37. By Conclusion 3, the enqueue operation being waited for by the dequeue operation can only be blocked by the dequeue operations that execute the dequeue algorithm in statements 26 to 39. However, from the dequeue algorithm, if a dequeue operation can acquire its requested locks among statements 26 to 39 in a finite time, it will release its locks in a finite time and walk down the heap tree. A dequeue operation only visits a heap node one time. In a heap tree structure, two dequeue operations never lock each other and, by Conclusion 4, an enqueue never blocks a dequeue. The enqueue operation waited for by the dequeue operation at the spinning loop will put a stable key into its target node to get the dequeue operation out of the spinning loop in a finite time. So, we have the following conclusion.

CONCLUSION 5. *A dequeue operation never holds a lock indefinitely, and if it can get its requested locks in a finite time, it will finish its dequeue operation in a finite time.*

By Conclusions 3, 4, and 5, when the total number of enqueues and dequeues is finite, each enqueue or dequeue will finish in a finite time. This shows that our parallel enqueue and parallel dequeue algorithms are deadlock free.

4. ANALYTICAL PERFORMANCE ANALYSIS

Expressing the performance of a parallel program precisely in mathematical form is difficult, and is sometimes impossible, because there are a set of complicated factors to be formalized. These range from parallel architectures and system software to applications. In this section, we approximately analyze the average number of locks that a hold operation (a pair of enqueue and dequeue operations) would wait for in the worst case during its execution for our LB algorithm, the LR algorithm, and the RK algorithm. The analytical results provide upper bounds on the performance of the algorithms.

We use the following assumptions to restrict our analysis:

(1) Heap node keys follow a uniform distribution.

(2) All processors issue their hold operations simultaneously and all the operations in a heap walk down the heap tree synchronously, i.e., if some operations can walk down the heap tree to the next level from the current level, they will get to the next level simultaneously. This assumption sets the largest contention on the heap node to give the worst case performance.

(3) The enqueue algorithm or the dequeue algorithm is simplified as a downward walking procedure along a heap tree.

(4) For simplification, a heap is assumed to have a complete balanced tree (the results are approximately valid for an incomplete tree). Let N be the total number of nodes in the heap. Its height is $\log(N+1)$, denoted by h. (in this paper, all the logs have base of 2.) In a heap, the level number of each node is labeled downwards from the root with level number of 1. The bottom level nodes have level number of h.

In addition, we let P be the total number of processors in a shared-memory system.

From the view point of an algorithmic abstraction, the dequeue operations in the LB algorithm, the LR algorithm, and the RK algorithm have the same procedure: removing a key from one bottom node and making comparisons with the nodes along its downward walking path from the root. A dequeue operation can walk down to next level from a current node only when its key is not smaller than child nodes of the current node. The key of a dequeue operation comes from one bottom node. So, its key is at least larger than the keys of the other $h-1$ nodes on the path from the bottom node to the root. Combining this with assumption 1, the probability for the key of a dequeue operation to be larger than the key in any heap node is

$$p_b = \frac{N+h-1}{2\times N} = \frac{N+\log(N-1)-1}{2\times N}. \tag{2}$$

When a dequeue operation occurs at level i (> 1), it has a key that is larger than those of the children of the nodes passed, which are a total of $2(i-1)$ nodes. The probability for a dequeue operation to reach level i is $p_b^{2(i-1)}$. On level i, there are 2^{i-1} nodes. The probability, denoted by $p_{deq}(i)$, for a dequeue operation to occur at a specific node on level i is

$$p_{deq}(i) = \frac{p_b^{2(i-1)}}{2^{i-1}}. \tag{3}$$

At level $i = 1$, all the dequeue operations compete for the root. When the number of processors is P, in the average case half of them issue dequeue operations. By equation (3), the average number, denoted by $Num_{deq}(i, P/2)$, of dequeue operations occurred on a node at level i (when $i > 1$) is

$$\begin{aligned} Num_{deq}(i, P/2) &= \sum_{j=1}^{P/2} j \times p_{deq}(i)^j \\ &= \frac{p_{deq}(i)[1 - p_{deq}(i)^{P/2}]}{[1 - p_{deq}(i)]^2} - \frac{P \times p_{deq}(i)^{P/2+1}}{2 \times [1 - p_{deq}(i)]} \end{aligned} \tag{4}$$

When $i = 1$, $Num_{deq}(1, P/2) = P/2$.

The enqueue operations in the LB algorithm, the LR algorithm, and the RK algorithm have very different execution procedures. In the following, we analyze each of them separately.

In the RK algorithm all the enqueue operations simultaneously entering the heap, in total $P/2$ enqueue operations on average, will have the same heap tree walking path from the root to level $h - \log(P/2)$. So, the number of enqueue operations contending for a node above level $h - \log(P/2) + 1$ on their insertion paths is $\frac{P}{2}$. Below level $h - \log(P/2)$, the number of enqueue operations contending for each node along the insertion paths will reduce by half each time they walk down a level. Hence, the number of enqueue operations contending for one node on level i ($> h - \log(P/2)$) along the insertion paths is $(P/2)/(2^{i-h+\log(P/2)})$. Consequently, the number of enqueue operations contending for one node on any level i along an insertion path is calculated as

$$Num_{enq-rk}(i, P/2) = \begin{cases} \frac{P}{2} & \text{if i} \leq h - \log(P/2), \\ \frac{P/2}{2^{i-h+\log(P/2)}} & \text{otherwise.} \end{cases} \tag{5}$$

By equations (4) and (5), on one node of level i along an insertion path, there are $Num_{deq}(i, P/2) + Num_{enq-rk}(i, P/2)$ operations contending for the lock of the node. In the average case, the number of locks waited for by an enqueue operation at level i is $Num_{deq}(i, P/2)/2 + Num_{enq-rk}(i, P/2 - 1)/2$. (Note that the number of other enqueue operations is $Num_{enq-rk}(i, P/2 - 1)$.) Hence, the total number of locks waited for by an enqueue operation during its heap tree walking procedure is

$$NLocks_{enq-rk} = \frac{\sum_{j=1}^{\log N}(Num_{deq}(j, P/2) + Num_{enq-rk}(j, P/2 - 1)}{2}. \tag{6}$$

For a dequeue operation in the RK algorithm it meets an enqueue operation only when its walking path overlaps with the insertion path of the enqueue operation. Because the total number of enqueue operations occurring at any level of the heap

is $P/2$, the average number of enqueues occurring on a node at level i is $(P/2)/2^{i-1}$. So, the average number of locks waited for by an dequeue operation at level i is $(Num_{deq}(i, P/2-1) + (P/2)/2^{i-1})/2$. Therefore, the total number of locks waited for by a dequeue operation during its heap tree walking procedure is

$$Nlocks_{deq-rk} = \frac{\sum_{j=1}^{\log N}(Num_{deq}(j, P/2-1) + P/2^j)}{2}. \quad (7)$$

By equations (6) and (7), the number of locks waited for by a hold operation in the RK algorithm is

$$Hold_{locks-rk} = NLocks_{enq-rk} + Nlocks_{deq-rk}. \quad (8)$$

In the LR algorithm, enqueue operations separate their insertion paths on the top part of the heap. On a node at level i along the insertion path of an enqueue operation, the number of concurrent enqueue operations is $(P/2)/2^{i-1}$ when $1 \leq i \leq \log(P/2)$, and 1 when $i > \log(P/2)$, which can be expressed as:

$$Num_{enq-lr}(i, P/2) = \begin{cases} \frac{P}{2^i} & \text{if } 1 \leq i \leq \log(P/2), \\ 1 & \text{otherwise.} \end{cases} \quad (9)$$

By equations (4) and (9), along the insertion path an enqueue operation will wait for $(Num_{deq}(i, P/2) + Num_{enq-lr}(i, P/2-1))/2$ operations' locks at level i. So, the total number of locks waited for by an enqueue operation during its heap tree walking procedure is

$$NLocks_{enq-lr} = \frac{\sum_{j=1}^{\log(N)}(Num_{deq}(j, P/2) + Num_{enq-lr}(j, P/2-1))}{2}. \quad (10)$$

For a dequeue operation in the LR algorithm, it has the same execution procedure as that in the RK algorithm. So, the total number, denoted by $Nlocks_{deq-lr}$, of locks waited for by a dequeue operation during its heap tree walking procedure is

$$Nlocks_{deq-lr} = Nlocks_{deq-rk}. \quad (11)$$

By equations (10) and (11), the number of locks waited for by a hold operation in the LR algorithm is

$$Hold_{locks-lr} = NLocks_{enq-lr} + Nlocks_{deq-lr}. \quad (12)$$

In the LB algorithm, enqueue operations not only separates their insertion paths on the top part of the heap like that in the LR algorithm, it also uses a bypassing technique to reduce the number of locks waited. Here, the number of concurrent operations on a node at level i is the same as that in the LR algorithm, which is $Num_{deq}(i, P/2) + Num_{enq-lr}(i, P/2)$. If an enqueue finds that its key is larger than the key of the current node, it will bypass the current node regardless how many other operations are contending for the current node. The probability for an enqueue operation to bypass the current node is 1/2. When the enqueue operation needs to lock current node with probability of $1 - 1/2$ $(= 1/2)$, it must wait for the operations preceding it releases the lock on the current node. The average number of operations preceding the enqueue operation is $(Num_{deq}(i, P/2) + Num_{enq-lr}(i, P/2-1))/2$, among which the number of dequeue operations is $Num_{deq}(i, P/2)/2$, and the number of enqueue operations is

$Num_{enq-lr}(i, P/2-1)/2$. Each dequeue operation must acquire the lock on the current node before it can walk down the heap tree. Each enqueue operation acquires the lock on the current node with probability 1/2. Hence, the average number of locks waited for by an enqueue operation at level i is

$$Nlocks_{enq-lb}(i) = \frac{2 \times Num_{deq}(i, P/2) + Num_{enq-lr}(i, P/2-1))}{8}. \quad (13)$$

The total number of locks waited for by an enqueue operation during its heap tree walking procedure is

$$NLocks_{enq-lb} = \sum_{j=1}^{\log(N)} Nlocks_{enq-lb}(j). \quad (14)$$

For a dequeue operation, it cannot bypass any lock. It must wait for the operations preceding it to release the lock on the current node. Similar to the above analysis, on a node along the walking path of the dequeue at level i, the average number of other dequeue operations is $Num_{deq}(i, P/2-1)$ and the average number of enqueue operations is $(P/2)/2^{i-1}$. So, the average number of operations preceding the dequeue operation is $(Num_{deq}(i, P/2-1) + (P/2)/2^{i-1})/2$. Hence, the total number of locks, denoted by $Nlocks_{deq-lb}$, waited for by a dequeue operation during its heap tree walking procedure is

$$NLocks_{deq-lb} = \frac{\sum_{j=1}^{\log(N)}(Num_{deq}(i, P/2-1) + P/2^{i+1})}{2}. \quad (15)$$

By equations (14) and (15), the number of locks waited by a hold operation in the LB algorithm is

$$Hold_{locks-lb} = NLocks_{enq-lb} + Nlocks_{deq-lb}. \quad (16)$$

Based on equations (8), (12), and (16) the numbers of locks waited for by the 3 algorithms are compared in Tables 4 and 2 respectively for two different heap sizes. The two tables present very similar comparison results. The number of locks waited for by the LB algorithm is just half of that waited by the LR algorithm and one-third of that waited for by the RK algorithm in most cases. The number of locks waited for is a reasonable metric to evaluate the parallelism exploited by a concurrent algorithm for accessing a priority heap. This analytical result shows the effectiveness of the lock bypassing technique in exploiting parallelism. However, we should note that some reduction in the number of locks waited for by an algorithm would not necessarily generate the same reduction in total execution time. Besides parallelism and heap size the execution time of an algorithm is also affected by the computation granularity of each node and by the architecture of the parallel computer system. In next section, we will compare the 3 algorithms on a parallel system.

5. EXPERIMENTAL PERFORMANCE ANALYSIS

To provide a more complete understanding on our LB algorithm and its performance advantage over previous algorithms, we compare the LB algorithm with the LR algorithm and the RK algorithm on a real parallel computing environment.

procs.	2	4	8	16	32	64	128
RK	1.57	7.41	17.80	36.23	68.66	125.09	221.45
LR	1.57	7.66	12.93	24.29	47.70	95.10	190.46
LB	0.79	3.12	5.95	11.79	23.63	47.47	95.28

Table 1. Comparison of the average number of locks waited for by the 3 algorithms investigated. Here the heap has 255 nodes.

procs.	2	4	8	16	32	64	128
RK	1.57	7.91	19.31	39.74	76.19	140.64	253.07
LR	1.57	8.16	13.43	24.80	48.22	95.66	191.08
LB	0.79	3.24	6.08	11.92	23.77	47.62	95.46

Table 2. Comparison of the average number of locks waited for by the 3 algorithms investigated. Here the heap has 512 nodes.

This section describes our experimental environment, experimental method and measurement results.

5.1 Experimental environment and method

We conducted our experiments on one hypernode of a HP/Convex Exemplar S-Class. This is a cache coherent non-uniform memory access (ccNUMA) system. Each hypernode of the Exemplar has 16 PA-8000 processors, each having a peak performance of 720 Mflops. Each processor has a 1 Mbyte instruction cache and a 1 Mbyte data cache. Processors in a hypernode are interconnected by a crossbar so that each processor in a hypernode has uniform memory access time. We implemented our LB algorithm, the LR algorithm, and the RK algorithm using the Exemplar's CPS thread library, which is similar to the POSIX thread standard. The Exemplar machine actually provides very powerful parallel compiler optimizations. In order to minimize the effect of optimizations, we masked all compiler optimizations in the implementation of the algorithms and made all algorithms have the same parallel threads creation and execution procedure.

In the real application of parallel priority heap operation algorithms, enqueue and dequeue operations interleave their execution. So, the enqueue or dequeue algorithm cannot be evaluated independently. Reference [Chung et al. 1993] proposes a useful evaluation model, called the hold model. In the hold model, a hold operation consists of an enqueue operation and a dequeue operation. Each parallel processor executes a number of hold operations to access the priority heap. In our experiments, we used the hold model to comparatively evaluate the LB algorithm and the other two algorithms. The average hold time is the major performance metric used in our experiments.

In different applications the heap data (i.e., the keys in heap nodes) have different distributions. In our experiments, three kinds of distributions – uniform distribution, exponential distribution, and geometric distribution – are used to generate enqueue keys to investigate algorithmic applicability spanning wide range of applications. In addition, the issue rate of hold operations is controlled by a spinning loop between two consecutive hold operations. Three inter-hold operation delays,

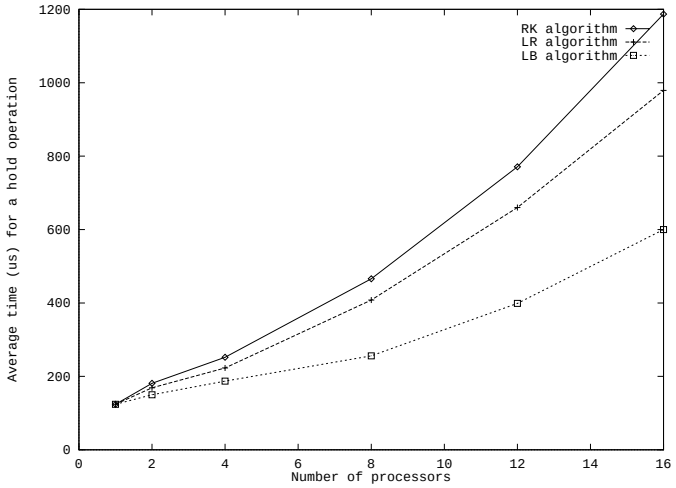


Fig. 8. Average hold operation times for the three algorithms where the heap data follows a uniform distribution and the inter-hold operation delay is 0 μs.

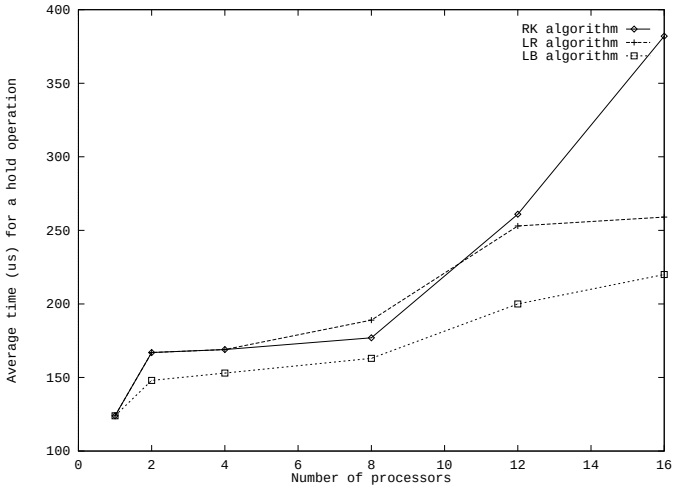


Fig. 9. Average hold operation times for the three algorithms where the heap data follow a uniform distribution and the inter-hold operation delay is 160 μs.

0 μs, 160 μs and 640 μs are measured. Each priority heap usually has three distinguished phases: growing up, stable, and shrinking. The first and the last phases usually take a short time. So, we focus our performance evaluation on the stable phase where the size of a priority heap changes a limited amount. Initially, the heap has 300 nodes. Each processor executes 10000 hold operations. The average hold time is reported.

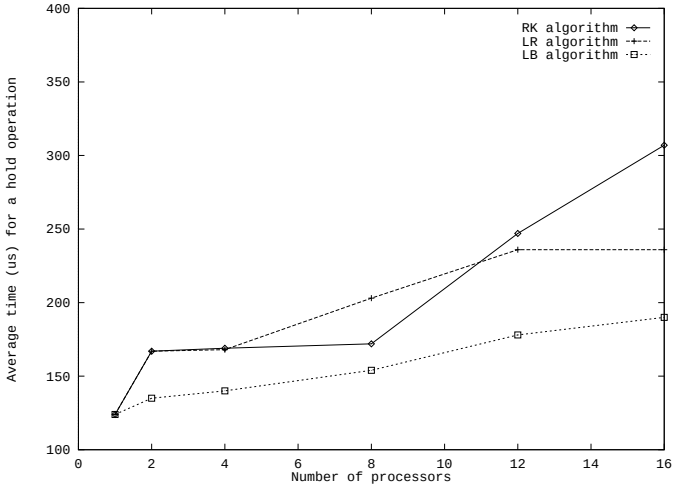


Fig. 10. Average hold operation times for the three algorithms where the heap data follows a uniform distribution and the inter-hold operation delay is 640 μs.

5.2 Experimental results

In Figures 8, 9, and 10, the average hold times of the three algorithms over uniform key distribution and different inter-hold operation delays are reported. When inter-hold delay is 0 (as illustrated in Figure 8), i.e., the heap has highest contention, the LR algorithm is slightly better than the RK algorithm. When the number of processors increases up to 16, the average hold time of the LR algorithm is about 20% less than that of the RK algorithm. This performance improvement of the LR algorithm over the RK algorithm shows the effectiveness of directing enqueue operations to different insertion paths. Compared with the LR algorithm and the RK algorithm the LB algorithm has achieved significant improvement in terms of the average hold time. When the number of processors increases up to 16, the LB algorithm has achieved a 50% reduction in average hold time over the RK algorithm, and a 40% reduction over the LR algorithm. These illustrate the opportunity for the bypassing technique to exploit parallelism in heap operation while the keys of heap nodes follow a uniform distribution. While inter-hold operation delay is increased from 0 μs to 160 μs and then to 640 (us), the average hold time curves for the three algorithms all drop significantly. This is shown in Figure 9 and Figure 10. For example, when the system has 16 processors, the maximal average hold time of RK algorithm drops from 1200 μs for an inter-hold time of 0 μs to 380 μs for an inter hold time of 160 μs, and to about 300 μs for an inter hold time of 640 μs. By studying the effect of increasing inter-hold time on the performance difference among the three algorithms, we can observe that the average hold time curve of the LR algorithm crosses that of the RK algorithm. When the number of processors is 8, the RK algorithm outperforms the LR algorithm. Only when the number of processors is increased to 16 does the LR algorithm outperform the RK algorithm by 30% in average hold time. This shows that only separating enqueue insertion paths cannot achieve significant improvement when heap contention is low. This

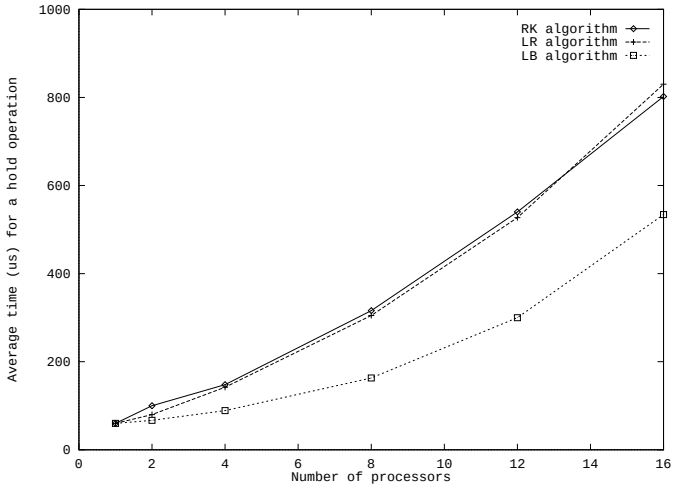


Fig. 11. Average hold operation times for the three algorithms where the heap data follows an exponential distribution and the inter-hold operation delay is 0 μs.

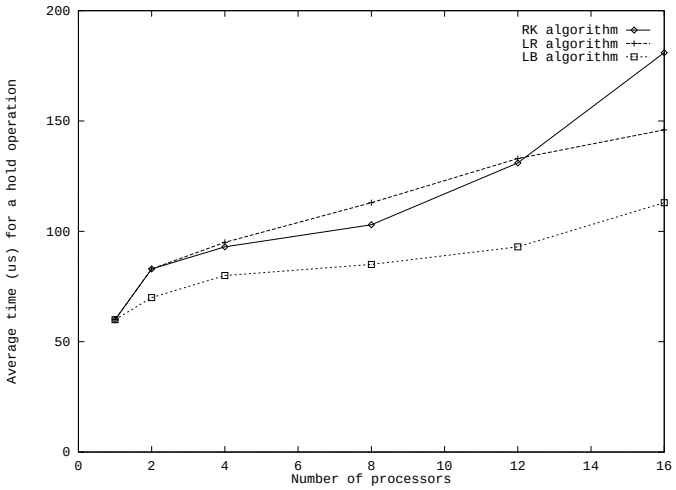


Fig. 12. Average hold operation times for the three algorithms where the heap data follow a exponential distribution and the inter-hold operation delay is 160 μs.

also shows that the effect of dequeue operations on enqueue operations is significant. However, the LB algorithm is observed to achieve consistent improvement over the LR and the RK algorithms in average hold time when contention on the heap is decreased. This shows the effectiveness of the lock bypassing technique in reducing the interference between enqueue operations and dequeue operations.

In Figures 11, 12, and 13, we show the average hold times of the three algorithms over an exponential key distribution and different inter-hold operation delays. When the inter-hold delay is 0 μs shown in Figure 12, the LR algorithm has nearly the identical average hold time curve to that of the RK algorithm. This is

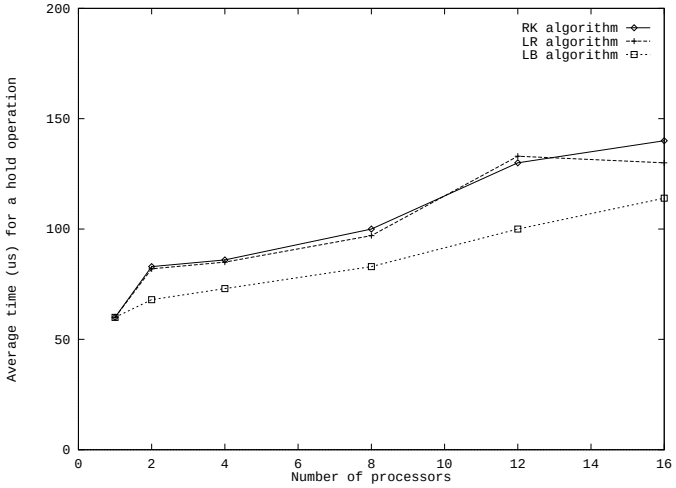


Fig. 13. Average hold operation times for the three algorithms where the heap data follow a exponential distribution and the inter-hold operation delay is 640 μs.

different from the uniform key distribution case. Comparing the effects of uniform distribution and exponential distribution, we can see that exponentially distributed data does not help the separation of enqueue operations to exploit parallelism. However, the LB algorithm still improves performance over both of the LR and the RK algorithms in average hold time by 25% when the number of processors is 16. This shows that the combining use of the lock bypassing technique with the enqueue insertion path separation technique yields good scalability over a wide range of applications. When inter-hold operation delay is increased further as illustrated in Figures 12 and 13, the LR algorithm has a slight improvement over the RK algorithm only when the number of processors is 16. Our LB algorithm still maintains about a up to 20% improvement of the average hold time over the LR and the RK algorithms.

In Figures 14, 15, and 16, the generation of heap node keys follows a geometric distribution in each processor. When the inter-hold operation delay is 0 μs, the LR algorithm shows an insignificant improvement over the RK algorithm. The maximal improvement is only about 6%. When the inter-hold operation delay is further enlarged to reduce contention on the heap operations, the average hold time curve of the LR algorithm crosses that of the RK algorithm. Similar to above observation, the LB algorithm has significant improvement in average hold time over both of the LR and the RK algorithms. Although the improvement of the LB algorithm over the LR and the RK algorithm decreases with the increase in inter-hold time delay (from Figure 14 to Figures 15 and 16), the LB still has about 25% improvement in hold time. A different result from that under uniform and exponential distributions shows that the LR algorithm even performs worse than the RK algorithm when the number of processors was 4 and inter-hold time delay was 160 μs.

In conclusion, if one considers the effect of different distributions on the performance of the three algorithms shown in Figure 8 throught Figure 16, the LB

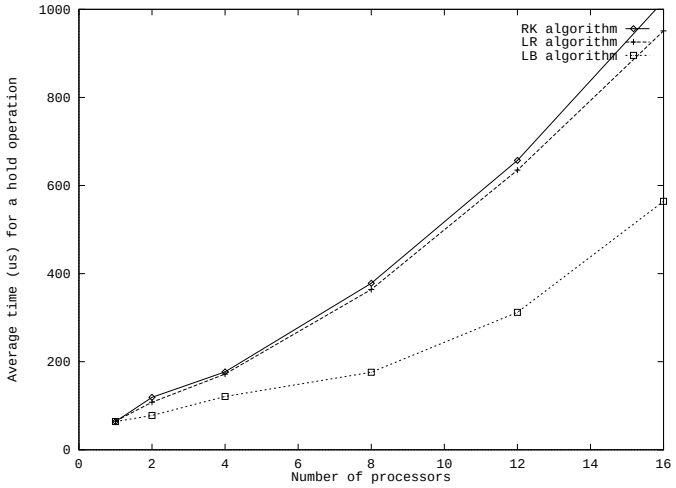


Fig. 14. Average hold operation times for the three algorithms where the heap data follows a geometric distribution and the inter-hold operation delay is 0 μs.

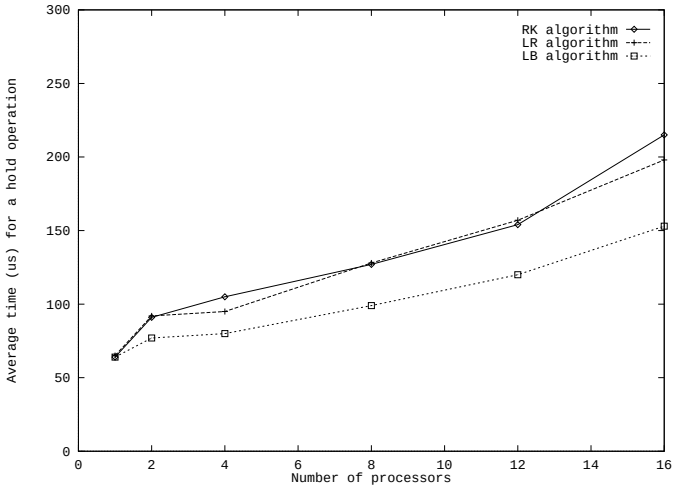


Fig. 15. Average hold operation times for the three algorithms where the heap data follow a geometric distribution and the inter-hold operation delay is 160 μs.

algorithm shows a much better applicability to a wide range of applications than the LR and the RK algorithms, and better scalability when contention on the heap increases. The LR algorithm shows a significant improvement in average hold time over the RK algorithm only when heap node keys are generated in a uniform distribution.

6. CONCLUSION

In this paper, we studied the features of enqueue and dequeue operations for concurrently accessing a priority heap. We proposed a lock bypassing technique to

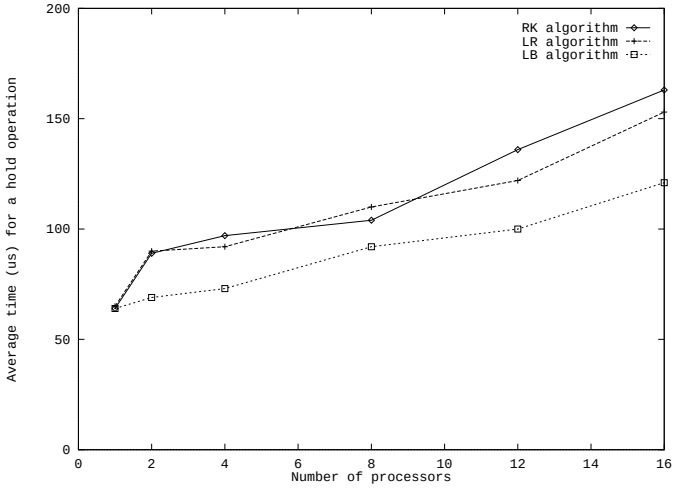


Fig. 16. Average hold operation times for the three algorithms where the heap data follow a geometric distribution and the inter hold operation delay is 640 μs.

allow concurrent enqueue operations to minimize the interference of locks on their insertion paths. By combining the lock bypassing technique and the enqueue path separation technique proposed in [Ayani 1991], we designed concurrent enqueue and concurrent dequeue algorithms: the LB algorithm. The correctness of the LB algorithm was presented, and the performance of the LB algorithm over previous algorithms was evaluated both analytically and experimentally.

From our evaluation, the following conclusions can be drawn:

—Compared with previous algorithms, the LB algorithm reduces the number of locks waited for in the worst case by half.

—The LB algorithm achieves a consistent performance improvement over the LR and the RK algorithms under three tested heap key distributions: uniform distribution, exponential distribution and geometric distribution. The maximal improvement in average hold time of the LB algorithm over the LR algorithm and the RK algorithm can be up to 50%. When contention on the heap decreases, the LB algorithm still outperforms the LR algorithm and the RK algorithm significantly.

—The LR algorithm only achieves a consistent performance improvement over the RK algorithm in a high contention heap with uniformly generated heap node keys. When a heap has exponentially generated keys or geometrically generated keys, the LR algorithm has performance similar to the RK algorithm. This shows that just separating enqueue insertion paths cannot significantly improve performance over the RK algorithm for a wide range of applications.

—In comparing the performance results of the LB algorithm and the LR algorithm, the lock bypassing technique is shown to be very effective in exploiting the parallelism of concurrently accessing a heap.

The algorithm is presented using the spining lock, which does not guarantee fairness. Because lock bypassing does not rely on a specific locking technique, the

fairness of this algorithm can be guaranteed by using the other kind of synchronization techniques, such as queue-based synchronization primitives. In addition, the cache in modern computers also has certain effect on the efficient implementation of a heap. Some useful research results can be found in reference [LaMarca and Ladner 1996].

ACKNOWLEDGMENTS

We appreciate Samir Das bringing this topic to our attention. We thank Wei Xie for his effort in debugging the workstation implementation of the algorithms. We are grateful to Neal Wegner for reading the paper and his constructive comments. Specially, we would like to thank Greg Astfalk for his valuable suggestions and comments, help, and support of this work. In addition, we thank Hewlett-Packard's Scalable Computing Division for us to use the Exemplar S-Class to test the algorithms.

REFERENCES

AYANI, R. 1991. Lr-algorithm: concurrent operations on priority queues. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing* (Oct. 1991), pp. 22–25. IEEE Computer Society Press.

BISWAS, J. AND BROWNE, J. C. 1987. Simultaneous update of priority structures. In *Proceedings of International Conference on Parallel Processing* (Aug. 1987), pp. 124–131. CRC Press.

BISWAS, J. AND BROWNE, J. C. 1993. Data structures for parallel resource management. *IEEE Transactions on Software Engineering 19*, 7 (July), 672–686.

CHUNG, K., SANG, J., AND REGO, V. 1993. A performance comparison of event calendar algorithms: an empirical approach. *Software–Practice and Experience 23*, 10 (Oct.), 1107–1138.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* The MIT Press, McGraw-Hill Book Company.

JONES, D. W. 1986. An empirical comparison of priority-queue and event-set implementations. *Communications of ACM 29*, 4 (April), 300–311.

LAMARCA, A. AND LADNER, R. E. 1996. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics (electronic publication) 1.*

OLARIU, S. AND WEN, Z. 1991. Optimal parallel initialization algorithm for a class of priority queue. *IEEE Transactions on Parallel and Distributed Systems 2*, 4 (Oct.), 423–429.

RANADE, A., S. CHENG, E. D., JONES, J., AND SHIH, S. 1994. Parallelism and locality in priority queues. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing* (Oct. 1994), pp. 490–496. IEEE Computer Society Press.

RAO, V. N. AND KUMAR, V. 1988. Concurrent access of priority queues. *IEEE Transactions on Computers 37*, 12 (Dec.), 1657–1665.