Skiplist-Based Concurrent Priority Queues

Itay Lotan
Stanford University
and
Nir Shavit
Sun Microsystems Labortatories

This paper addresses the problem of designing scalable concurrent priority queues for large scale multiprocessors – machines with up to several hundred processors. Priority queues are fundamental in the design of modern multiprocessor algorithms, with many classical applications ranging from numerical algorithms through discrete event simulation and expert systems.

While highly scalable approaches have been introduced for the special case of queues with a fixed set of priorities, the most efficient designs for the general case are based on the parallelization of the *heap* data structure. Though numerous intricate *heap*-based schemes have been suggested in the literature, their scalability seems to be limited to small machines in the range of ten to twenty processors.

This paper proposes an alternative approach: to base the design of concurrent priority queues on the probabilistic *skiplist* data structure, rather than on a *heap*. To this end, we show that a concurrent *skiplist* structure, following a simple set of modifications, provides a concurrent priority queue with a higher level of parallelism and significantly less contention than the fastest known *heap*-based algorithms.

Our initial empirical evidence, collected on a simulated 256 node shared memory multiprocessor architecture similar to the MIT Alewife, suggests that the new *skiplist* based priority queue algorithm scales significantly better than heap based schemes throughout most of the concurrency range. With 256 processors, they are about 3 times faster in performing deletions and up to 10 times faster in performing insertions.

[0]A preliminary version of this paper appeared in the proceedings of the first *International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.
Keywords Large Scale Multiprocessors, Concurrent Data Structures, Priority Queues, Scalability.

1. INTRODUCTION

In recent years, we have seen a steady increase in the number of processors available on commercial multiprocessors. Machines with 64 processors [42], and ccNUMA architectures which scale to more than a hundred processors [40], are no longer found only in research labs. This increase in the availability of larger computing platforms, has not been met by a matching improvement in our ability to construct scalable software. If anything, it has made the difficulties more acute.

Priority queues are of fundamental importance in the design of modern multiprocessor algorithms. They have many classical applications ranging from numerical algorithms, through discrete event simulation, and expert system design. Though there is a wide body of literature addressing the design of concurrent priority queue algorithms for small scale machines, the problem of designing scalable priority queues for large machines has yet to be addressed.

This paper begins to tackle this problem by proposing an alternative approach: base the design of concurrent priority queues on the SkipList data structures of Pugh [30], rather than on the popular Heap structures found throughout the literature [3; 4; 9; 10; 16; 17; 23; 24; 26; 29; 34; 35; 36; 37; 43]. As we will show, this design shift, even in the simple form presented here, can produce significant performance gains.

The next three subsections in the introduction summarize the main points detailed in later sections of the paper.

1.1 Priority Queues

A priority queue is an abstract data type that allows n asynchronous processors to each perform one of two operations: an `Insert` of an item with a given priority, and a `Delete-min` operation that returns the item of highest priority in the queue[1]. We are interested in "general" queues, ones that have an unlimited range of priorities, where between any two priority values there may be an unbounded number of other priorities. Such queues are found in numerical algorithms and expert systems [25; 33], and differ from the bounded priority queues used in operating systems, where the small set of possible priorities is known in advance. The latter special case has scalable solutions applicable to large machines [39].

How does one go about constructing a concurrent priority queue allowing arbitrary priorities? Since for most reasonable size queues, logarithmic search time easily dominates linear one, the literature on concurrent priority queues consists mostly of algorithms based on two paradigms: search trees [18; 5] and heaps [3; 4; 9; 10; 16; 17; 23; 24; 26; 29; 34; 35; 36; 37; 43]. Empirical evidence collected in recent years [10; 17; 39] shows that heap-based structures tend to outperform search tree structures. This is probably due to a collection of factors, among them that heaps do not need to be locked in order to be "rebalanced," and that `Insert` operations on a heap can proceed from bottom to root, thus minimizing contention along their concurrent traversal paths.

[1]Though there are a variety of other operations, such as *merging* and *searching for the k-th item*, that can be added to priority queues based on SkipLists [32] and Heaps, such operations are outside the scope of this paper.

One of the most effective concurrent priority structures known to date is the heap-based algorithm of Hunt et al. [17], which builds on and improves other known heap based algorithms [3; 4; 23; 29; 35; 36; 43]. Its good performance is the result of several techniques for minimizing locking and contention: inserts traverse bottom up, only a single counter location is locked for a "short" duration by all operations, and a bit reversal scheme distributes delete requests that traverse top-down.

Unfortunately, as our empirical evidence shows, the performance of [17] does not scale beyond a few tens of concurrent processors. As concurrency increases, the algorithm's locking of a shared counter location, however short, introduces a sequential bottleneck that hurts performance. The root of the tree also becomes a source of contention and a major problem when the number of processors is in the hundreds. In summary, both balanced search trees and heaps suffer from the typical scalability impediments of centralized structures: sequential bottlenecks and increased contention.

1.2 The New Approach

The solution we propose in this paper is to design concurrent priority queues based on the highly distributed SkipList data structures of Pugh [30; 31]. Surprisingly, SkipLists have received little attention in the parallel computing world, in spite of their highly decentralized nature.

SkipLists are search structures based on hierarchically ordered linked-lists, with a probabilistic guarantee of being balanced. The basic idea behind SkipLists is to keep elements in an ordered list, but have each record in the list be part of up to a logarithmic number of sub-lists. These sub-lists play the same role as the levels of a binary search structure, having twice the number of items as one goes down from one level to the next. To search a list of N items, $O(\log N)$ level lists are traversed, and a constant number of items is traversed per level, making the expected overall complexity of an `Insert` or `Delete` operation on a SkipList $O(\log N)$.

In this paper we introduce the *SkipQueue*, a highly distributed priority queue based on a simple modification of Pugh's concurrent SkipList algorithm [31]. Inserts in the SkipQueue proceed down the levels as in [31]. For `Delete-min`, multiple "minimal" elements are to be handed out concurrently. This means that one must coordinate the requests, with minimal contention and bottlenecking, even though `Delete-min`s are interleaved with `Insert` operations.

Our solution is as follows. We keep a specialized delete pointer which points to the current minimal item in this list. By following the pointer, each `Delete-min` operation directly traverses the lowest level list, until it finds an unmarked item, which it marks as "deleted." It then proceeds to perform a regular `Delete` operation by searching the SkipList for the items immediately preceding the item deleted at each level of the list, and then redirecting their pointers in order to remove the deleted node.

We note that SkipQueue is fundamentally different from the bounded-priority SkipList-based priority queues described in [39]. Those queues work *only* for the special case where priorities are derived from a small predetermined set. They are based on a small SkipList structure whose elements are "bins," one per priority. Each "bin" contains many items of the same priority, with a specialized "delete-bin" [18] added to the structure to speed

up deletions. As a consequence, the key factors governing their performance are the contention and bottlenecks in the "bins," and not the efficiency of the operations on the SkipList as in the case of the more general SkipQueues.

SkipQueues have several notable advantages over prior heap and tree based schemes:

—Unlike in trees and heaps, all locking is distributed. There is no locking of a root or centralized counter.

—Unlike in search trees, balancing is probabilistic and there is no need for a major synchronized "rebalancing" operation.

—Unlike in heaps, `Delete-min` operations are evenly distributed over the data structure, minimizing locking contention.

—Unlike in heaps, there is no need to pre-allocate all memory since the structure is not placed in an array.

1.3 Empirical Performance Study

To the best of our knowledge, our work is the first attempt to empirically evaluate a SkipList based data structure on a large scale machine. In Section 5, we evaluated the performance of our SkipQueue algorithm in comparison to the most effective of former priority queue algorithms, the heap-based priority queue algorithm of [17]. As a comparison base, since a linked-list protected by a single lock had already been shown to perform rather poorly [17], we tested a simple *FunnelList* structure, a linked-list of items with a combining-funnel front-end [38] instead of a single lock. The combining-funnel [38] is a structure similar to a combining tree [15; 13], intended to allow high levels of parallelism in accessing the linked list.

We ran a collection of standard synthetic benchmarks [17; 39] on a simulated 256 processor ccNUMA multiprocessor architecture similar to the MIT Alewife [1]. The simulation was done on the well accepted Proteus platform of Brewer et al. [7]. Though this is not a real 256 node machine, we note that previous research by Della-Libera [11] has shown that with appropriate scaling, Proteus simulates a 32 node Alewife machine accurately for the kinds of data structures tested in this paper. Our benchmarks tested sequences of `Insert` and `Delete-min` operations on small and large queues. Our conclusions, presented in Section 5, are that the SkipQueue outperforms the heap-based algorithms throughout the concurrency range.

2. SKIPLISTS

The basic structure of the SkipList is as follows. Each item inserted into the SkipList is represented by a node (record) with a number of outgoing "forward" pointers (see Figure 1). The number of forward pointers a node has is referred to as the *level* or *height* of the node. The i-th pointer of a node points to the next element in the list whose height is at least i. The list is sorted and so a node points to elements ordered after it. The level of the node is chosen randomly when it is first inserted into the list, using geometric distribution derived via a pseudo-random number generation algorithm. The key idea in the design of SkipLists is that this probabilistic choice of height for a node will guarantee that the number of nodes participating in each level list will be exponentially decreasing

as one goes up the tree. To find item A, a processor starts with the highest level list, which should have an expected constant number of items, and searches for two adjacent items B and C such that $B \leq A \leq C$. The processor then follows the pointer of the next lower level in item B, again searching an expected constant number of items, and so on until the lowest level list is traversed and the item is found. Altogether, in a list of N items, $O(\log N)$ levels are traversed, and a constant number of nodes is traversed per level, making the overall complexity $O(\log N)$. Inserting a node into the list would mean to search for its place in the list and connect all of its levels. Deleting requires finding the element and disconnecting all of its levels.

In order for the SkipList to be used by a number of processors concurrently some modifications need to be made. As suggested in [31], locks should be introduced into the list. It is important to notice that when a processor disconnects the top level of some node, this does not in any way affect the correctness of the structure, only its performance. Therefore, processors can insert an element one level at a time from bottom to top and delete it a level at a time from top to bottom. A node is considered in the list if its level is at least 1 and considered removed from the list if its level is reduced to 0. This way only one level of a node needs to be locked at any given time when a node is inserted or deleted in front of it. The concurrent performance of the list benefits immensely from this feature. The detailed code for insertion and deletion operations appears in Section 6. They are performed as follows:

Insertion After randomly picking a level for the node, a processor searches for all the nodes in the list after which the new node should be inserted; one node for each level of the new node. The processor then locks the new node so no other processor can attempt to delete it while it is being inserted. Now the processor acquires the lock on the forward pointer of the first level node found earlier, and inserts the node at that level. The lock is then released and the lock of the second level node is now acquired, and so on, until pointers in all the levels of the node have been inserted.

Deletion To delete an item, a processor searches for the node in the list and remembers all the nodes immediately preceding it; one node for each level of the node. The processor then acquires the lock on the node to make sure it is not in the process of being inserted. In order to delete a node from level i, the processor needs to acquire two locks: one on the level i pointer of the node before the node, and one on the node's own level i pointer. The processor starts from the top level of the node and works its way down, each time acquiring two locks and removing the node from that level. It is important to note that one needs to be careful when removing a node from the list, since other processors might hold pointers to it. Thus, to prevent a node from being deleted prematurely, the processor deletes first the pointer going into the node, and only then redirects the forward pointer of the node to point to the node just before it. This way, other processes that held pointers to the node can still use them and need not be aware that the node they are using has already been deleted.

The potential advantage of SkipLists, which led us to try and use them as a basis for a priority queue in place of heaps, is that a processor needs to lock only a small part of the list at a time in order to insert or delete a node. Assuming that the insertions

and deletions are distributed more or less evenly throughout the structure, there should be only a few processors competing for the same lock at any given time, and since the operation they need to perform after acquiring a lock is very short (setting a pointer), locks are not highly contended.

3. SKIPLIST BASED PRIORITY QUEUES

This section explains how to create the *SkipQueue* data structure. The new operation one needs to add to the basic SkipList structure is a `Delete-min` operation. When there are several processors trying to delete the minimal node concurrently, the regular search methods no longer help. Two or more processes might compete for the same first node and the losers will be left empty handed. The key to our construction is the fact that the lowest level of the SkipList is really a regular linked list. The code for our `Delete-min` operation can be found in Figure 11.

As depicted in Figure 1, we modify the SkipList nodes so that each has a `deleted` flag, which is set to false when the node is first inserted into the list. When a processor wants to find the minimal node it starts traversing the bottom level of the *SkipQueue* until it finds a node whose flag is not yet set. It sets a flag marking that this node is logically deleted. What's left now is just to remove the node from the list. Since a processor already knows which node to remove, it uses the standard `Delete` operation of the SkipList. One is assured that no two processors will ever delete the same node since only one could have set its `deleted` flag. Processors use a register-to-memory `SWAP` operation to set the `deleted` flag. This allows any number of processors to search for a minimal node concurrently, competing for the first available node. The first processor to successfully `SWAP` false to true in the `deleted` flag of a node gets to delete that key, and the other processors move on down the list to try and find the next available node.

Though the above implementation should suffice in practice, we added a simple time-stamping mechanism to the code in order to assure a stronger ordering property among deleted values: each deleting processor returns the minimal node among those inserted *completely before* it began. As we prove in Section 4.2, the mechanism allows a processor

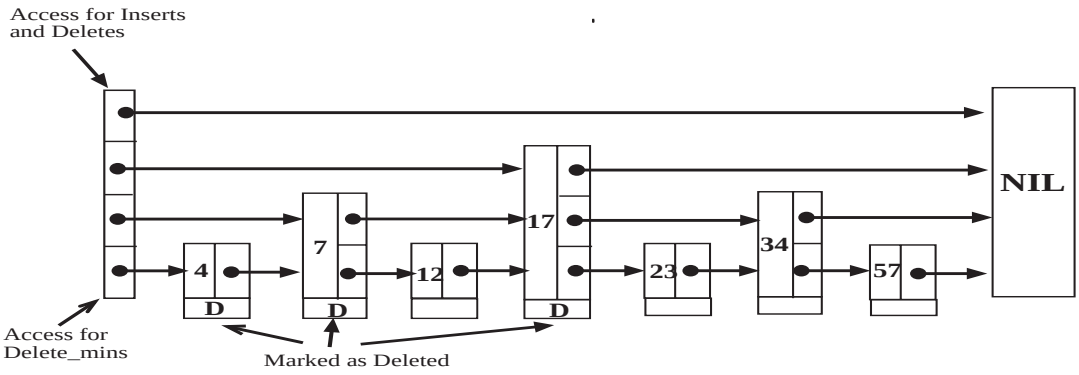

Fig. 1. The basic *SkipQueue* structure

to recognize and ignore nodes that were inserted cuncurrently with it. It works as follows. After a node is completely inserted into the SkipQueue, it acquires a time-stamp. A deleting processor notes the time at which it starts its search through the lower level of the SkipQueue, and only looks at nodes whose time-stamp is smaller then the time at which it started its traversal, effectively ignoring nodes inserted during its traversal.

Starvation in the SkipQueue is theoretically possible but highly unlikely. A processor traversing the bottom of the list might always be beaten to the next available node by another processor. But the further down the list a processor is, the smaller the number of competitors it will encounter, so its chances of finding a yet-to-be-deleted node grows as it advances. In the worst case, it might reach the end of the list and return an `EMPTY` message. The fact that nodes are physically deleted from the list as a processor traverses the lowest level does not cause a problem, due to the order of pointer switches. A processor that is in the vicinity of a node that is being deleted, will not traverse it at all.

One more issue that requires attention is garbage collection. Unlike systems with built in garbage collection such as Lisp [41] and the Java™ programming language [14], our benchmarking system requires explicit garbage collection. Following Pugh's suggestion in [31], we note that it is safe to free the memory used by a particular node only after all the processors that were in the structure when the node was deleted, have already exited the structure. To be able to ascertain when such a condition is met, we again use a time-stamping mechanism. Each processor registers the time it has entered the structure in a special place in shared memory, and whenever a node is deleted, it is stamped with its deletion time. In the benchmarks in this paper, we assigned a dedicated processor to do all the garbage collection. Whenever a node is deleted, the node is put at the end of the garbage list of the deleting processor. The dedicated processor determines the time-stamp of the oldest processor in the list and then visits the garbage lists of all the processors. The processor looks at the deletion time of the first node of every list, and if it is earlier than the time-stamp of the oldest processor in the structure, it frees its memory. The dedicated processor will repeat this procedure as long as the structure exists. Clearly, if one desires, this garbage collection task can be split/shared among processors.

4. CORRECTNESS

We begin with a short formalization of a computation model which will serve as the basis of our proof. We then provide a specification and correctness proof.

4.1 The Computation Model

Our computation model follows [19]. A *concurrent system* consists of a collection of n *processors*. Processors communicate through shared data structures called *objects*. Each object has a set of primitive *operations* that provide the only means to manipulate that object. Each processor P is a sequential thread of control [19] which applies a sequence of operations to objects by issuing an invocation and receiving the associated response. A *history* is a sequence of invocations and responses of some system execution. Each history induces a "real-time" order of operations where an operation A *precedes* another

operation B if A's response occurs before B's invocation. Two operations are *concurrent* if they are unrelated by the real-time order. A *sequential history* is a history in which each invocation is followed immediately by its corresponding response. A *serialization* [19] of a collection of operations in a given history is a total ordering of those operations consistent with the real-time order induced by the concurrent history. In other words, serialized operations appear to take effect atomically at some point between their invocation and response. For the purposes of our proof, every shared memory location L of a multiprocessor machine memory allows every processor P_i to perform one of the following set of atomic operations (see [20; 21] for details):

. $\texttt{READ}_i(L)$ reads location L and returns its value.

. $\texttt{WRITE}_i(L, v)$ writes the value v to location L.

. $\texttt{SWAP}_i(L, v)$ writes the value v to location L and returns the value of L prior to this write.

We note that even though our algorithms use other abstract operations such as `lock` and `unlock`, these are all details of the implementation of the `Insert` and `Delete` operations on the SkipList, which are not necessary for our proof. The `READ` operation can be applied to the machine's shared clock location, in which case the clock's value at the time of the read is returned.

4.2 Specification and Proof

A *priority queue* object is a concurrent shared object which allows each processor P_i where $0 \leq i \leq n-1$ to perform two types of operations on the object: $\texttt{Insert}_i(r)$ and $\texttt{Delete\_Min}_i()$. The insert operation has input r from the range of `values`. The $\texttt{Delete\_Min}_i()$ operation returns a value or `EMPTY`. Without loss of generality, we will assume that all inserted and deleted values are unique.

We require that a *correct* implementation of a priority queue meet the following specification.

Definition 1. *For every* `Delete_Min` *operation in a finite or infinite history H, let I be the set of values inserted by* `Insert` *operations preceeding it in H. There exists a serialization of all* `Delete_Min` *operations in H, such that for each operation, if D is the set of values deleted by* `Delete_Min` *operations serialized before it, the value returned by the* `Delete_Min` *is the minimal element of $I - D$, or* `EMPTY` *if $I - D = \emptyset$.*

Lemma 1. *The SkipQueue data structure meets Definition 1.*

Proof. For any history H, let the serialization order on `Delete_Min` operations be defined as follows.

—Order each successful `Delete_Min` at the successful `SWAP(node1->deleted, TRUE)` of the deleted mark into a node, that is, the operation in which it completed its "logical" deletion (Line 5 of Figure 11).

—Order each unsuccessful `Delete_Min`, that is, one that returned `EMPTY` since it did not have a successful `SWAP(node1->deleted, TRUE)` in H, at its response point (its return instruction).

—Order each uncompleted `Delete_Min`, that is, one that was invoked but did not have a response or a successful `SWAP(node1->deleted, TRUE)` in H, at its invocation point (its first instruction).

The above is a serialization order since each `Delete_Min` operation is ordered between its invocation and response.

Each unsuccessful `Delete_Min` did not perform a successful `SWAP(node1->deleted, TRUE)` in H, so there is no returned value and we are done. It remains to be shown that each `Delete_Min` operation returned `EMPTY` only if $I - D = \emptyset$ and otherwise returned the minimum element x in $I - D$. We abuse notation and henceforth denote this operation, whether it returned x or `EMPTY`, as $\texttt{Delete\_Min}_x$.

We assume correct behaviour of the SkipList structure in SkipQueue based on Pugh's Concurrent Skiplist correctness proof in [31]. It follows from Definition 1 and the correctness of the SkipList, that every `Insert` operation with a value in I must have added a node with its respective input value to the lowest level of the SkipQueue prior to the invocation of $\texttt{Delete\_Min}_x$. Moreover, the `Insert`'s last executed instruction, which is a write of `node1->timeStamp`, must have preceded the $\texttt{Delete\_Min}_x$'s first instruction, which is a read of `time`. It follows that `node1->timeStamp < time`, so all nodes containing values in I will pass the $\texttt{Delete\_Min}_x$ operation's test in Line 4 of Figure 11.

$\texttt{Delete\_Min}_x$ traverses the lowest level list, marking and deleting the first unmarked element x it reads in the list, or `EMPTY` if none is found. In order to prove that either $I - D = \emptyset$ or x is the minimal element in $I - D$, consider all other elements y in I that were not returned by $\texttt{Delete\_Min}_x$. If y was not returned, it must be due to one of the following three situations:

(1) The element y was added to the list at a node preceding x (for a successful $\texttt{Delete\_min}_x$) or preceding the end of the list (for an unsuccessful one), yet $\texttt{Delete\_Min}(x)$ never read the pointer leading to the node containing y. It follows that the node must have been removed from the lowest level of the skiplist by some $\texttt{Delete\_min}_y$ operation. The marking of the node by $\texttt{Delete\_min}_y$ must have preceded the removal of the pointer to it at the lowest level of the SkipList, and since the pointer was not read, its removal, in turn, must have preceded the successful marking by $\texttt{Delete\_Min}_x$. Thus, $\texttt{Delete\_min}_y$ must have preceded $\texttt{Delete\_Min}_x$, and so y is in D, and not in $I - D$.

(2) The element y was added to the list at a node preceding x (for a successful $\texttt{Delete\_min}_x$) or following the end of the list (for an unsuccessful one), yet y was not deleted. Since the pointer to the `node1` containing y was read, it must be that the $\texttt{Delete\_min}_x$ failed on the `SWAP(node1->deleted, TRUE)` operation in Line 5 of Figure 11. This could happen only if some other $\texttt{Delete\_min}_y$ performed a succesful `SWAP(node1->deleted, TRUE)` before the `SWAP` attempt by $\texttt{Delete\_min}_x$ (recall that in our model, SWAP operations are atomic and thus totally ordered) and so $\texttt{delete\_min}_y$ preceded $\texttt{delete\_min}_x$ in the serialization order. Thus, y is in D, and not in $I - D$.

(3) The element y was added to the list but at a node following (reachable from) the node containing the returned value x of a successful $\texttt{Delete\_min}_x$. Since by

the correctness of the SkipList structure, elements in the SkipList are ordered in ascending order, this item y is greater than x, and so cannot be the minimum of $I - D$. Note that this cannot happen in case of an unsuccessful `Delete_min`$_x$ since all reachable nodes upto the end of the list are traversed.

It follows that there is no unreturned y in $I - D$ if `EMPTY` was returned, and no $y < x$ in $I - D$ if x was returned. □

We now examine the liveness properties of the SkipQueue algorithm. The proof that the `Insert` operation is terminating follows from [31] assuming that the locks used in the implementation are all fair. The marking part of the `Delete_Min` operation (the logical deletion part) is not guaranteed to terminate, but is non-blocking: a processor does not complete only if other processors repeatedly succeed in performing successful `delete_min` operations. Moreover, as explained in section 3, the further down the lower level of the SkipQueue that a deleting process advances, the better are its chances of finding a free element to delete. The second part of the `Delete_Min` operation (the removal from the list) is guaranteed to terminate as shown by [31].

5. PERFORMANCE RESULTS

To evaluate the performance of SkipQueues on large scale machines, we used a simulated 256 processor ccNUMA multiprocessor architecture similar to the MIT *Alewife* machine [1] of Agarwal et al. The simulation was conducted using the well accepted *Proteus*[2] multiprocessor simulator of Brewer et al. [6; 7]. Proteus simulates parallel code by multiplexing several parallel threads on a single CPU. Each thread runs on its own virtual CPU with accompanying local memory, cache and communications hardware, keeping track of how much time is spent using each component. In order to facilitate fast simulations, Proteus does not support complete simulation of the hardware. Instead, operations which are local (i.e. do not interact with the parallel environment) are executed uninterruptedly on the simulating machine's CPU and memory. The amount of time used for local calculations is added to the time spent performing (simulated) globally visible operations to derive each thread's notion of the current time. Proteus makes sure a thread can only see global events within the scope of its local time. Since actual machine instructions are counted for local operations, the quality of the code used to implement algorithms under Proteus can play an important part in determining the running time of the entire application.

Our benchmarking methodology was a variation of commonly used synthetic benchmark of [17; 39], in which processors alternate between performing some small amount of local work and accessing a priority queue. We used a short work period of 100 cycles between invocations, testing the data structures at high load levels. We chose not to present low load (long work period) benchmarks since they offer no surprises. As Figure 2 shows, as the work period is increased (lowering the load) the latency of operations goes down because of the lowered chances of concurrent access to shared locks in the data structure.

[2]Version 3.00, dated February 18, 1993.

Work Amount	`Delete_min` latency	`Insert` latency
100	190710	65699
1000	181442	45629
2000	149139	44904
3000	125111	32372
4000	107603	25823
5000	110975	36174
6000	69579	26462

Fig. 2. `Insert` and `Delete_Min` latency with different amounts of work with 256 processes and 1000 initial elements.

Processors accessing the data structure randomly chose whether to `Insert` a random priority item or apply a `Delete-min` operation. The priorities of inserted items were chosen uniformly at random, attempting to capture the most common priority queue access patterns.[3]

In each experiment, we measured *latency*, the amount of time (in machine cycles) it takes for an average `Insert` operation and an average `Delete-min` operation as well as the average time it takes to access the object. We also varied the initial size of the priority queue and the ratio of `Insert` operations to `Delete-min` operations.

Our benchmarks compared three data structures, all of which support insertion and deletion of arbitrary priorities.

Heap This is the priority queue implementation of Hunt et al. [17]. We choose this algorithm as representative of the class of heap based priority queue algorithms [3; 4; 8; 9; 10; 16; 17; 22; 23; 24; 26; 29; 34; 35; 36; 37; 43] since it was shown [17] to perform better than others under various `Insert/Delete-min` benchmarks. In this algorithm, there is a single lock that protects a variable holding the size of the array-based heap. All processors must acquire it in order to begin their operations, but it is not held for the duration of the operation. Rather, the heap's size is updated, then a lock on either the first or last element of the heap is acquired and then the first lock is released. In order to increase parallelism, insertions traverse the heap bottom-up, which reduces contention for the top-most nodes and avoids the need for a full height traversal in most cases. Deletions proceed top-down. Insertions also employ a novel bit-reversal technique that allows a series of insertion operations to proceed up the heap independently without getting in each other's way. As many as $O(N)$ operations can proceed in parallel on a heap of size N. The implementation is based on the code from the authors' *FTP* site, optimized for Proteus.

FunnelList This is a priority queue based on a simple sorted linked-list structure. Exclusive access to the list is controlled by a combining-funnel data structure [39], in order to increase parallelism and reduce contention. Combining-funnels are adaptive variants of combining trees [13; 15]. A funnel consists of a series of *combining layers*

[3]Though worst case adversarial insertion sequences can be devised for each of the tested data structures, we do not present such scenarios here. For most of these scenarios, the chance that they occur is relatively low (see [31] for a discussion for the case of SkipLists).

through which processors' requests are funneled in an attempt to combine. Processors that want to perform an operation on the list, be it `Insert` or `Delete-min`, enter the funnel and try to combine with other processors' requests. Every group of processors that combine requests is represented by one processor which exits the funnel and acquires a lock on the list. If the intent is insertion, the lucky processor traverses the list inserting the items of the processors it represents into their place in the list. If `Delete-min` is the operation, the representative processor cuts off as many items as it needs from the beginning of the list and distributes them to the processors it has combined with. The width and depth of the funnel adapt to the concurrency level on-the-fly.

SkipQueue This is the concurrent SkipList based priority queue described in Section 2. We note that the final design presented in Section 2 and used here is the result of experimentation with a series of different designs based on a concurrent skip-list. We tried using a funnel to regulate access of deleting processors at the bottom level of the SkipList. This funnel performed well in low contention but caused too much overhead when the concurrency level increased to 64 processors and more. In the end, we concluded that letting processors compete for the smallest element gives the best results, even when contention reaches 256 processors. In our experiments we assumed an upper bound on the maximal number N of items in the priority queue, making the maximal level of an element in the list be $\log N$. We note that there are more advanced methods known to set the maximal level, but we concluded that for the presented benchmarks, the performance gain is not significant enough to warrant more than this simple method [31].

5.1 Small Structure Benchmark

In our first benchmark we initialized all structures to contain 50 random elements. We then performed 70000 operations with an equal chance for `Insert` or `Delete-min`, and measured the average latency for both `Insert`s and `Delete-min`s, as well as an overall average latency. Figure 3 shows the results for this small structure benchmark. The two graphs at the top are the results for the whole concurrency range of up to 256 processors, with `Delete-min` operations on the left and `Insert` operations on the right. Below each graph is a closeup showing only a part of the scale.

As one can see, the structures maintain their size: about 100 elements throughout the experiment. When concurrency is low (less than 16 processors), the FunnelList performs the best since it has the simplest implementation and the adaptive funnel structure is still very small so it does not incur latency penalties. But as concurrency increases, the size of the funnel increases and the delays in traversing and combining in the funnel become a more dominant factor. The ability of the SkipQueue to distribute the load and the relatively small amount of coordination done per operation becomes dominant, and its performance becomes superior to the other structures. Looking just at the insertion latency, SkipQueues do better than FunnelLists with as little as 8 processors, and the difference grows to a fourfold difference at 256 processors. The Heap structure is slower than SkipQueues throughout the concurrency range, and by 256 processors it is 10 times slower.

The picture is somewhat different when we compare `Delete-min` latencies. This is the weak point of the SkipQueues, because there could potentially be heavy reading at

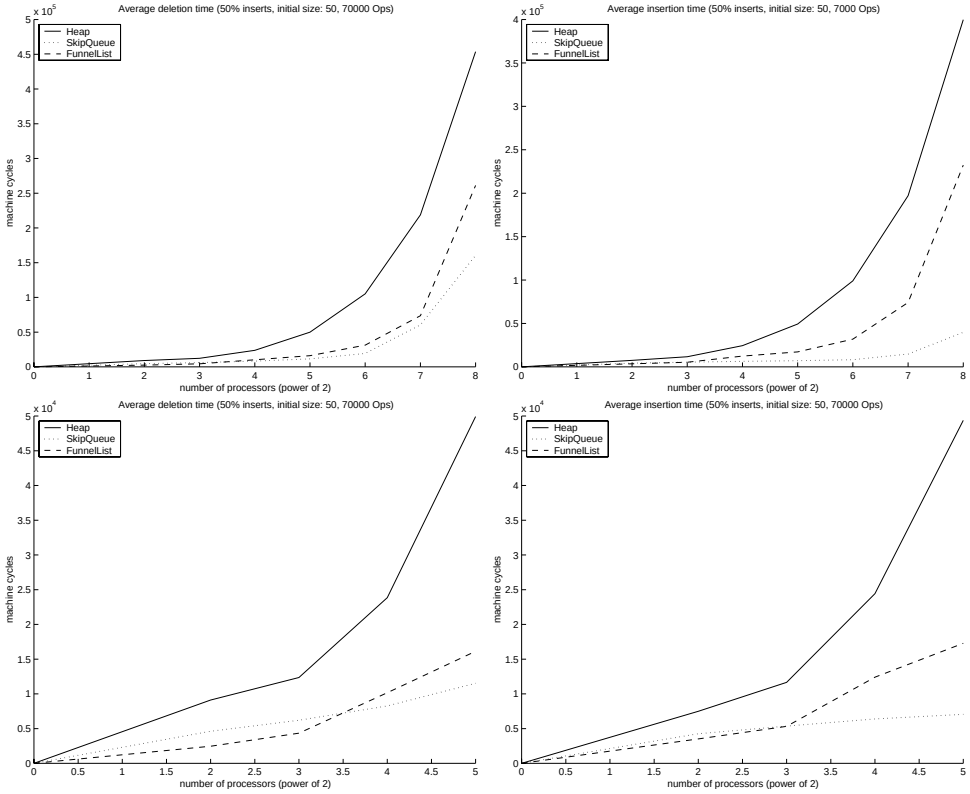

Fig. 3. The small structure benchmark.

the lowest level of the list, which creates contention and slows things down. We can see that SkipQueues become better than FunnelLists only when concurrency rises above 16 processors, and as we increase the concurrency beyond that, the gap increases only slightly. SkipQueues still perform deletions significantly better than Heaps, almost 3 times better at 256 processors.

5.2 Large Structure Benchmark

Our second experiment measured the performance of the three priority queue implementations when the data structure contains a large amount of elements: about 1000 items at all times. Figure 4 shows the results of the large structure benchmark. As before, the results for `Delete-min` operations on the left and for `Insert` operations on the right.

The large size of the data structure exposes the inadequacy of the FunnelList structure whose latency per operation is linear in the number of items in the funnel-protected linked-list. The other two algorithms have logarithmic dependency on the number of items and are thus only slightly influenced by the increase in the structure size. They are only 5 percent slower while the structure is initially 20 times larger. With 256

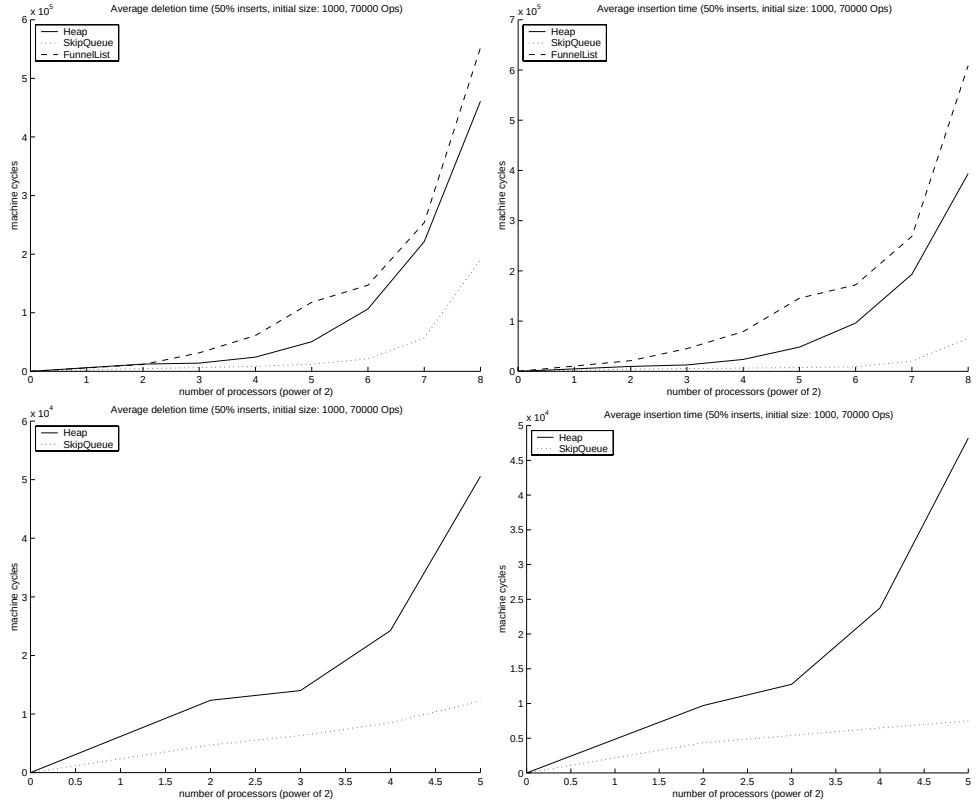

Fig. 4. The large structure benchmark.

processors, SkipQueues are 2.5 times faster on deletions and as much as 6.5 times faster in performing insertions.

5.3 Large Structure Benchmark With 70 Percent Deletions

In the third experiment we varied the ratio of insertions to deletions in favor of the deletions, by biasing the coin flip so that the simulation would choose to do a `Delete-min` operation 70 percent of the time. We started out with 27000 initial items in each structure and performed a total of 60000 operations on each. In this benchmark, the size of the structure decreases gradually throughout the simulation until it reaches a size of about 3000 elements at the end. We excluded FunnelLists from this benchmark since the results of the Large Structure Benchmark show that they perform miserably when the structure is large. Figure 5 shows the results of the large structure benchmark. As before, `Delete-min` operations are on the left and `Insert` operations on the right.

As can be seen, SkipQueues are up to 2.5 times faster than Heaps in performing deletions at 256 processors. Deletion latency increased in both SkipQueues and Heaps because of the larger number of deleting processors. On the other hand insertion time for

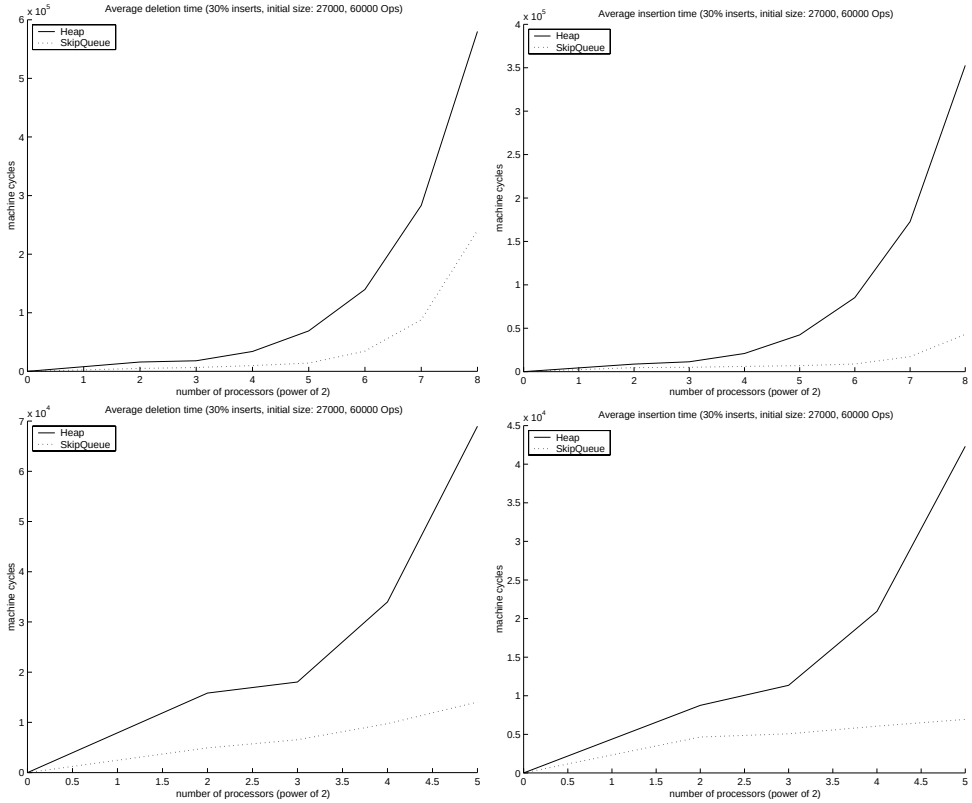

Fig. 5. The large structure benchmark with 70 percent deletions.

SkipQueues improved a little since there where less processors trying to `Insert` at the same time, but insertion in the heap structure suffered from the extra contention at the root area caused by the increased number of deletes. Heaps can handle insertions much better than they handle deletions. Insertions are distributed across the wide (bottom) part of the heap, while deletions are concentrated at the root. Therefore extra insertions would be spread out and have little impact, but extra deletions would also slow down inserting processors which happen to get close to the root of the heap. Overall the extra deletions slowed down both SkipQueues and Heaps but their effect on heaps was markedly worse.

In conclusion, The FunnelList is the most effective method when concurrency is low (up to 16 processors) and the size of the structure is small, but deteriorates in performance as soon as the structure becomes too large. The SkipQueue scales significantly better than the Heap algorithm throughout the concurrency range, and outperforms the FunnelList at concurrency levels of more than 16 processors.

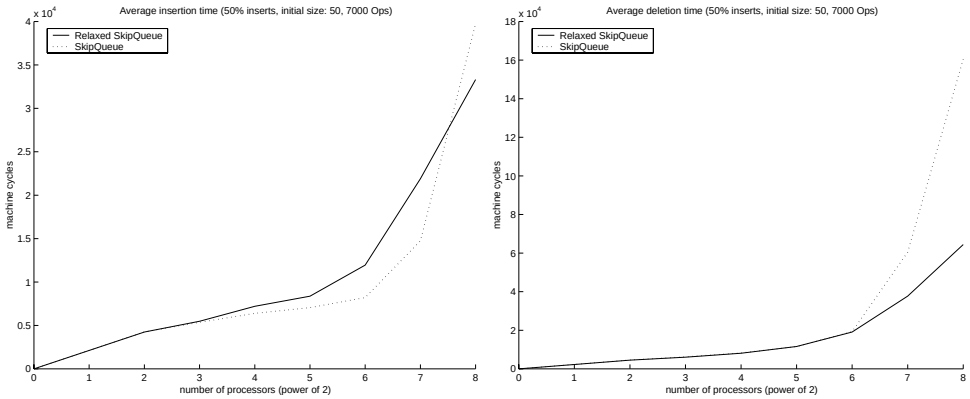

Fig. 6. SkipQueue vs. Relaxed SkipQueue for small structure.

5.4 Relaxed SkipQueue

We now examine the effect of removing the order-preserving time-stamp mechanism from the `Insert` and `Delete_min` operations. In other words, `Insert` operations will not need to mark items with a time-stamp, and `Delete_min` operations will not need to test the time-stamp of items they are about to delete.

With the order-preserving mechanism removed, one gets a new *relaxed* SkipQueue specification that allows a `Delete_Min` to return items inserted concurrently with it. Formally, for any `Delete_Min` operation d, we define a set I, as before, to include all elements whose `Insert` operations preceded d. We define a set D to include all elements returned by `Delete_min` operations that preceded or were concurrent with d. We also define a set I_C to include all elements inserted by `Insert` operations that were concurrent with d. The `Delete_Min` operation d is guaranteed to return an item from $\min(I - D)$ or a smaller item from I_C. In other words, if an item smaller than the minimum of $I - D$ is found among concurrently inserted entries, this item will be returned.

We tested the relaxed SkipQueue under the above three benchmarks. As seen in Figures 6, 7, and 8, both versions of the SkipQueue behave more or less the same up to a concurrency level of 32 processors. When concurrency is higher than 32, the relaxed versions performs deletions faster than the regular version: up to twice as fast in the best case. We also notice though, that there is a matching slowdown of the relaxed SkipQueues in performing insertions in these high concurrency levels.

We believe the slower `Insert`s are actually a side effect of the faster `Delete_Min`s. Bearing in mind that processes choose at random (a flip of a virtual coin) whether to insert or delete, when processes complete their deletions faster (as in the case of the realxed SkipQueues), there are more of them at each given moment trying to perform an insert. There is thus more contention on the locks of the list and the latency of the `Insert`s increases.

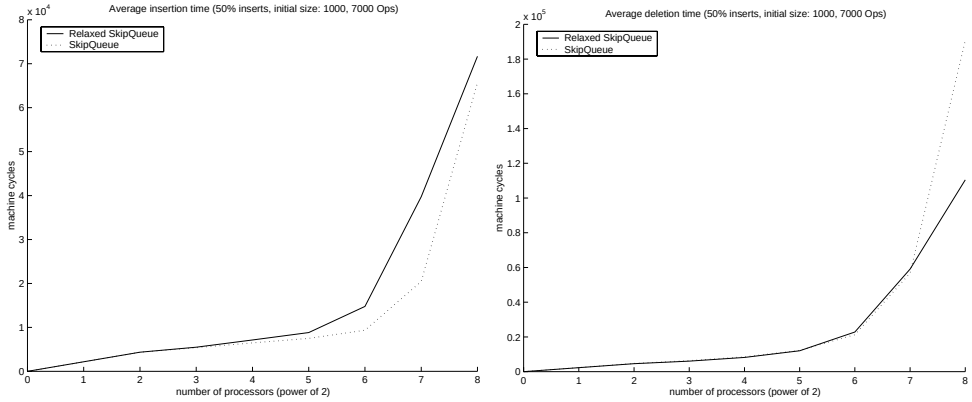

Fig. 7. SkipQueue vs. Relaxed SkipQueue for large structure.

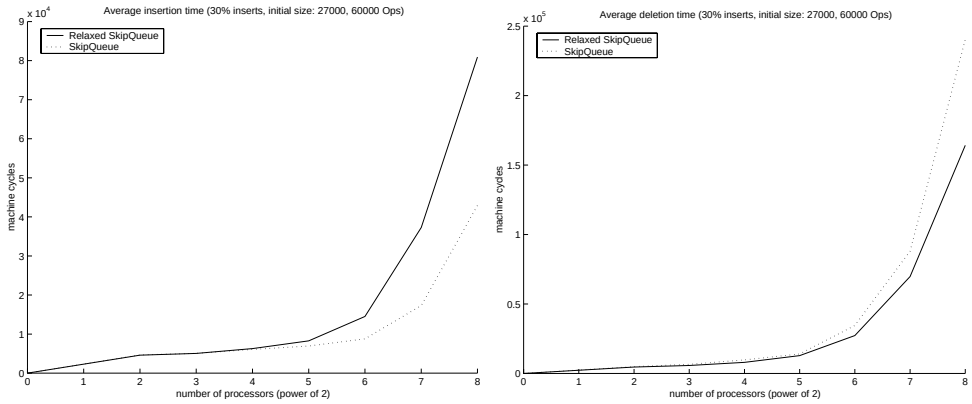

Fig. 8. SkipQueue vs. Relaxed SkipQueue for large with 70 percent deletion.

6. PSEUDO CODE FOR THE PRIORITY QUEUES

This section describes the code of our SkipQueues in detail. Our implementation is based on the SkipList implementation in [31]. The code for the auxiliary procedures and for the `Insert` is identical, and our changes are in the `Delete_Min` procedure which uses the `Delete` operation for SkipLists provided in [31]. We note that for compatibility with earlier *C*-based SkipList implementations, the interface of the actual implemented code differs slightly from the specification of Section 4.2. An inserted item in the `Insert` procedure is actually a pair of `key` and `value)`, where comparisons are done on the key and the value is just the stored item. The `Insert` procedure returns a success code. The `Delete_min` operation returns the deleted item's `value` in a designated memory location, and returns a notification of success or a possible `EMPTY` SkipQueue.

Figure 9 describes the auxiliary procedures we use. The `getLock` procedure is used

```
 node_t * getLock(node_t * node1, key_t key, int level)
 {
1   node2 = node1->next[level]
2   while (node2->key < key) { // Look for the node with the largest
3      node1 = node2           // key smaller than the key we're
4      node2 = node1->next[level] // searching for.
5   }
6   lock(node1, level) // Lock the node.
7   node2 = node1->next[level]
8   while (node2->key < key) { // Something changed before locking.
9      unlock(node1, level)    // Unlock node.
10     node1 = node2           // Get the next node in the queue.
11     lock(node1, level)      // Lock it.
12     node2 = node1->next[level]
13  }
14  return node1
 }

 int randomLevel()
 {
1   int l = 1
2   while (random() < p)
3      l++
4   if (l > queue->maxLevel)
5      return queue->maxLevel
6   else
7      return l
 }
```

Fig. 9. Code for auxiliary procedures. `getLock` is used to lock the node with the largest key smaller than `key`. `randomLevel` picks the maximal level of newly created node.

by a processor to acquire a lock on the node which has the largest key which is smaller than the key it is searching for. The node is assumed to be somewhere in front of (i.e. reachable from) the node a processor currently holds, and so it attempts to lock only a certain level of the node. In lines 1–6, a processor searchs for the node and gets a lock on a specified level. It then makes sure (lines 7–13) that no new node with a key closer to its search key was inserted while it was locking. If one was inserted, the processor moves the lock to this new node and checks again. In our benchmarks, processors used semaphores provided by the *Proteus* simulator to implement locks (in the code of SkipQueue and FunnelList). More efficient lock implementations are known in the literature.

The `randomLevel` procedure calculates the maximal level for a newly created node. It does so (lines 1–3) by tossing a coin and incrementing a counter as long as the toss was successful. With the first failure the tossing ceases. This provides a geometric distribution of the results. In lines 4–8 a processor checks that the result is not bigger than the maximal allowed value. It then returns the maximal value.

Figure 10 describes the `Insert` procedure. To insert a value with a given key, a processor first searches and saves an array containing all the "preceding" nodes, ones after which the new node would be inserted at each level (lines 1–9). It then acquires a

```
 int Insert(key_t key, value_t value)
 {
1   node1 = queue->Head                // Search from the queue head
2   for (i = queue->maxLevel; i > 0; i--) {  // search all levels.
3      node2 = node1->next[i]
4      while (node2->key > key) {      // Find the place at this
5         node1 = node2                // level in which to
6         node2 = node2->next[i]       // Insert the new node.
7      }
8      savedNodes[i] = node1   // Save the location that was found.
9   }
10  node1 = getLock(node1, key, 1)
11  node2 = node1->next[i]
12  if (node2->key == key) {
13     node2->value = value;
14     unlock(node1, 1)
15     return UPDATED
16  }
17  level = randomLevel()      // Generate the level of the new node.
18  newNode = CreateNode(level, key, value)
19  newNode->timeStamp = MAX_TIME;  // Initialize the time stamp.
20  lock(newNode, NODE)        // Lock the entire node.
21  for (i = 1; i <= level; i++) {
22     if (i != 1)             // level 1 is already locked
23        node1 = getLock(savedNodes[i], key, i)
24     newNode->next[i] = node1->next[i]       // insert the new node
25     node1->next[i] = newNode                // into the queue.
26     unlock(node1, i)
27  }
28  unlock(newNode, NODE)      // Release the lock on entire node.
29  newNode->timeStamp = getTime();  // Set the time stamp.
30  return INSERTED    // The insertion was successful.
 }
```

Fig. 10. Code for inserting a node into the queue.

lock on the first level preceding node. If the key searched for already exists in the queue, then this locking prevents the node from being deleted by another processor while the inserting processors are trying to update it (line 10). If the key is already in the queue, the inserting processor just updates the node's value and returns. Otherwise, it calulates a new maximal level and creates a new node to be inserted into the SkipQueue (lines 12-18). It locks the new node to prevent it from getting deleted while the insertion is in progress. Then, in lines 20–25, the processor goes from bottom to top through all the levels at which the new node should be connected and connects it into the queue. At every level, the processor first acquires a lock on the node, then makes the connection, and finally releases the lock. All that is left now is for the processor to release the lock on the new node (line 26).

The code for `Delete_Min` operation appears in Figure 11. A processor starts at the bottom level of the head node and advances through the bottom list until an unmarked node is found (lines 1–7). It uses a register-to-memory swap operation (denoted `SWAP`

in the code) to check if a node is unmarked, while marking it at the same time. If no unmarked node was found, it returns. Otherwise, it notes the key of the node it marked and sets out to delete it from the queue (lines 8–13). The processor searches for the location of the node at all the levels in which it appears. It saves pointers to the preceding nodes in each level (lines 15–22). Then, the processor makes sure it has a pointer to the node it wants to delete, and attempts to lock it in order to make sure that it is already completely inserted (lines 23–26). The processor now proceeds to the top level of the node and gets a lock on both its preceding node at this level and on its own top level. The processor removes the node at the top level, releases the locks, and moves on to perform the same series of operations on all lower levels (lines 27–34). Once this sequence is complete, the node is no longer reachable from the SkipQueue, and the processor releases the lock and adds the node to its garbage list (lines 35-36).

Part of this work was performed at Tel-Aviv University with funding from the Israel Science Foundation under grant 03610882, the Israeli Ministry of Science, and NSF grants 9225124-CCR and 9520298-CCR.

REFERENCES

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991. Also as MIT Technical Report MIT/LCS/TM-454, June 1991.

[2] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[3] R. Ayani. Lr-algorithm: concurrent operations on priority queues. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing* pp. 22-25, 1991.

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

[5] J. Boyar, R. Fagerberg and K.S. Larsen. Chromatic Priority Queues. Technical Report, Department of Mathematics and Computer Science, Odense University, PP-1994-15, May 1994.

[6] E.A. Brewer, C.N. Dellarocas. PROTEUS *User Documentation*. MIT, 545 Technology Square, Cambridge, MA 02139, 0.5 edition, December 1992.

[7] E.A. Brewer, C.N. Dellarocas, A. Colbrook and W.E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. MIT Technical Report /MIT/LCS/TR-561, September 1991.

[8] Vincenzo A. Crupi, Sajal K. Das, Maria Cristina Pinotti Parallel and Distributed Meldable Priority Queues Based on Binomial Heaps. In *Proceedings of the 1996 International Conference on Parallel Processing*, Vol. 1 1996: 255-262

[9] Sajal K. Das, Maria Cristina Pinotti, F. Sarkar. Distributed Priority Queues on Hypercube Architectures. In *International Conference on Distributed Computing Systems (ICDCS)* 1996: 620-628

[10] N. Deo and S. Prasad. Parallel Heap: An Optimal Parallel Priority Queue. In *The Journal of Supercomputing*, Vol. 6, pp. 87-98, 1992

[11] G. Della-Libera. Reactive Diffracting Trees. Master's Thesis, Massachusetts Institute of Technology, 1997.

[12] G. Della-Libera. Dynamic diffracting trees. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, July 1996.

[13] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.

[14] J. Gosling, B. Joy, G. L. Steele Jr. The Java™ Language Specification. *Addison-Wesley*, 2550 Garcia Avenue, Mountain View, CA 94043-1100, 1996.

[15] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.
343-364.

[16] Q. Huang. An Evaluation of Concurrent Priority Queue Algorithms. Technical Report, Massachusetts Institute of Technology, MIT-LCS/MIT/LCS/TR-497, May 1991.

[17] G.C. Hunt, M.M. Michael, S. Parthasarathy and M.L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. In *Information Processing Letters*, 60(3):151–157, November 1996.

[18] T. Johnson. A Highly Concurrent Priority Queue Based on the B-link Tree. Technical Report, University of Florida, 91-007. August 1991.

[19] M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[20] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems, 13*, 1, 124–149, January 1991.

[21] Herlihy, M. A Methodology For Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems* 15(5): 745-770, November 1993.

[22] B. Le Cun, B. Mans and C. Roucairol. Comparison of Concurrent Priority Queues for Branch and Bound Algorithms. Technical Report RR-MASI-92-65, MASI, Universite Paris 6, Oct. 1992.

[23] Carlo Luchetti and M. Cristina Pinotti. Some comments on building heaps in parallel. In *Information Processing Letters*, 47(3):145-148, 14 September 1993

[24] B. Mans. Portable Distributed Priority Queues with MPI. In *Concurrency: Practice and Experience*, 10(3):175-198, March 1998.

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

[26] Optimal Parallel Initialization Algorithms for a Class of Priority Queues. In *IEEE TRansactions on Parallel and Distributed Systems*, Vol. 2, No. 4, October 1991.

[27] Maria Cristina Pinotti and Geppino Pucci. Parallel Priority Queues. *Information Processing Letters*, 40(1):33-40, 11 October 1991.

[28] G.H. Pfister and A. Norton. 'Hot Spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(11):933–938, November 1985.

[29] Sushil K. Prasad and Sagar I. Sawant. Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP 95)*, 1995.

[30] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Communications of the ACM*, 33(6):668–676, June 1990.

[31] W. Pugh. Concurrent Maintenance of Skip Lists. Technical Report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, CS-TR-2222.1, 1989.

[32] W. Pugh. A Skip List Cookbook. Technical Report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, CS-TR-2286.1, July 1989 (Revised June, 1990).

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

[34] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and Locality in Priority Queues. In *IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, October 1994

[35] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers* 37, 1657-1665, December 1988.

[36] P. Sanders. Fast priority queues for parallel branch-and-bound. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 379-393, Lyon, 1995. Springer.

[37] P. Sanders. Randomized Priority Queues for Fast Parallel Access. In *Journal of Parallel and Distributed Computing*, 49(1), 86 - 97, 1998.

[38] N. Shavit and A. Zemach. Combining funnels. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 61–70, Puerto Vallarta, Mexico, June 28th – July 2nd 1998.

[39] N. Shavit and A. Zemach. Concurrent Priority Queue Algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 113-122, Atlanta, GA, May 1999.

[40] `http://www.sgi.com/origin/2000/index.html`

[41] G. L. Steele Jr. Common Lisp. Second Edition, *Digital Press, Digital Equipment Corporation,* 1990.

[42] `http://www.sun.com/servers/hpc/products/hpc10000.html`

[43] Y. Yan and X. Zhang. Lock Bypassing: An Efficient Algorithm for Concurrently Accessing Priority Heaps. *ACM Journal of Experimental Algorithmics*, vol. 3, 1998. `http://www.jea.acm.org/1998/YanLock/`

```
 int Delete_Min(value_t * value))
 {
 1   time = getTime();  // Mark the time at which the search starts.
 2   node1 = queue->head->next[1] // Start search at start of first level.
 3   while (node1 != queue->tail) {     // Search until end of queue.
 4      if (node1->timeStamp < time) { // Ignore all nodes that were
                                       // inserted after search began.
 5         marked = SWAP(node1->deleted, TRUE) // Swap the flag value.
 6         if (marked == FALSE)         // An unmarked node was found,
 7            break                     // so end the search.
 8         node1 = node1->next[1]       // Move to next node.
 9      }
10   }
11   if (node1 != queue->tail)  { // We found an unmarked node
12      *value = node1->value     // save its value
13      key = node1->key          // and its key. 11   } 12   else
14      return EMPTY             // No node was found in the queue.
15   node1 = queue->head        // Start the search from the head.
16   for (i = queue->maxLevel; i > 0; i--) // Search all levels.
17      node2 = node1->next[i]
18      while (node2->key > key) { // Find the place at this
19         node1 = node2           // level in which the node
20         node2 = node2->next[i]  // with the key is located.
21      }
22      savedNodes[i] = node1   // Save the location that was found.
23   }
24   node2 = node1
25   while (node2->key != key)  // Make sure we have a pointer
26      node2 = node2->next[1]  // to the node with the key.
27   lock(node2, NODE)   // Lock the entire node to be deleted.
28   for (i = node2->level; i > 0; i--) {
29      node1 = getLock(savedNodes[i], key, i)  // Lock this level on
30      lock(node2, i)  // the node to be deleted and node before it.
31      node1->next[i] = node2->next[i] // Remove the node from the
32      node2->next[i] = node1          // queue.
33      unlock(node2, i)      // Release the locks on this level at
34      unlock(node1, i)      // the deleted node and node before it.
35   }
36   unlock(node2, NODE)   // Release the lock on entire node.
37   PutOnGarbageList(node2)    // Put the node on the garbage list.
38   return DELETE      // Delete was successful.
 }
```

Fig. 11. Code for deleting the smallest node from the queue.