Concurrent data structures for Unbounded Priority Queues

- Yash Chandarana and Bhavana Jain

Priority queues are fundamental data structures. From the operating system level to the user application level, they are frequently used as basic components. For example, the ready-queue that is used in the scheduling of tasks in many real-time systems, can usually be implemented using a concurrent priority queue. Consequently, the design of efficient implementations of concurrent priority queues is an active research area. We have implemented the two most referenced research papers -

1. G.C. Hunt, M.M. Michael, S. Parthasarathy and M.L. Scott. **An Efficient Algorithm for Concurrent Priority Queue Heaps.**
2. Lotan, I. and Nir Shavit. **Skiplist-Based Concurrent Priority Queues.**

The second paper implements the underlying skiplist data structure using locks and hence is a blocking algorithm. We have taken a forward step and used **lock-free skiplists** to implement the priority queue.

Fine Grained Locking over the Sequential Heap Implementation:

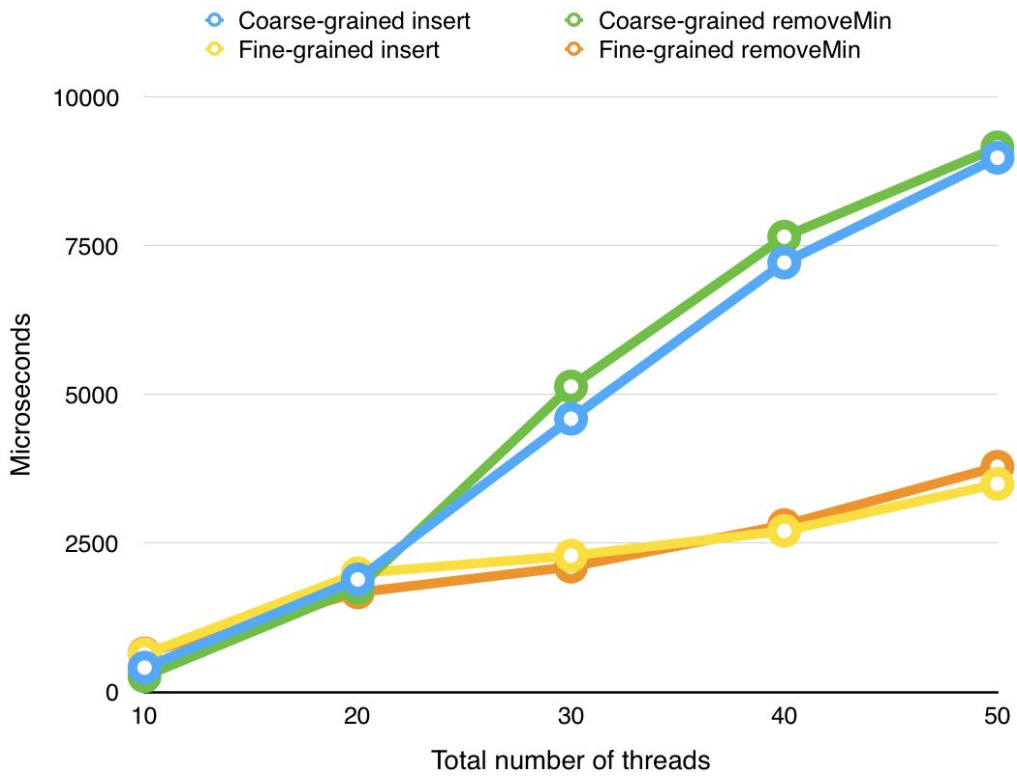
Fine grained heap is a concurrent linearizable version of the sequential heap data structure. We have used the array-based implementation of representing the heap since it is the most space efficient. This also means that the queue is bounded by size but that should not be a problem in any practical application since we can set the array size to whatever the max size the system on which the program is being executed allows.

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 and heaps. Empirical evidence collected in recent years 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.

This implementation is the first attempt which allows, for a heap of size M, O(M) insertions in parallel as compared to O(log M) for all the previous implementations. On large heaps the algorithm achieves significant performance improvement. For small heaps it still performs well, but not as well as a coarse grained implementation. To check the point till

where coarse-grained performs better than fine-grained, we tested both and the results are plotted below:

Comparison of average execution time coarse-grained vs fine-grained:



From the graph we conclude, coarse-grained and fine-grained go together till number of threads is equal to around 20. After that point, fine-grained performs substantially better than coarse-grained. Hence, verified.

Overview of the implementation:
Each node in the heap has a Status which can have EMPTY (nodes with no data), AVAILABLE (nodes that are available for deletion), and a node that has been inserted and is being moved into its place is tagged with the thread id of the inserting thread.

In a delete operation, root is swapped with the “rightmost” node, so that heap remains full, and the element in the heap is stored so that it can be returned later. Then, the

thread doing delete swaps nodes on a path from root to a leaf (path is chosen by comparing priority of the 2 child of the current node) so that the ordering property of the heap is maintained.

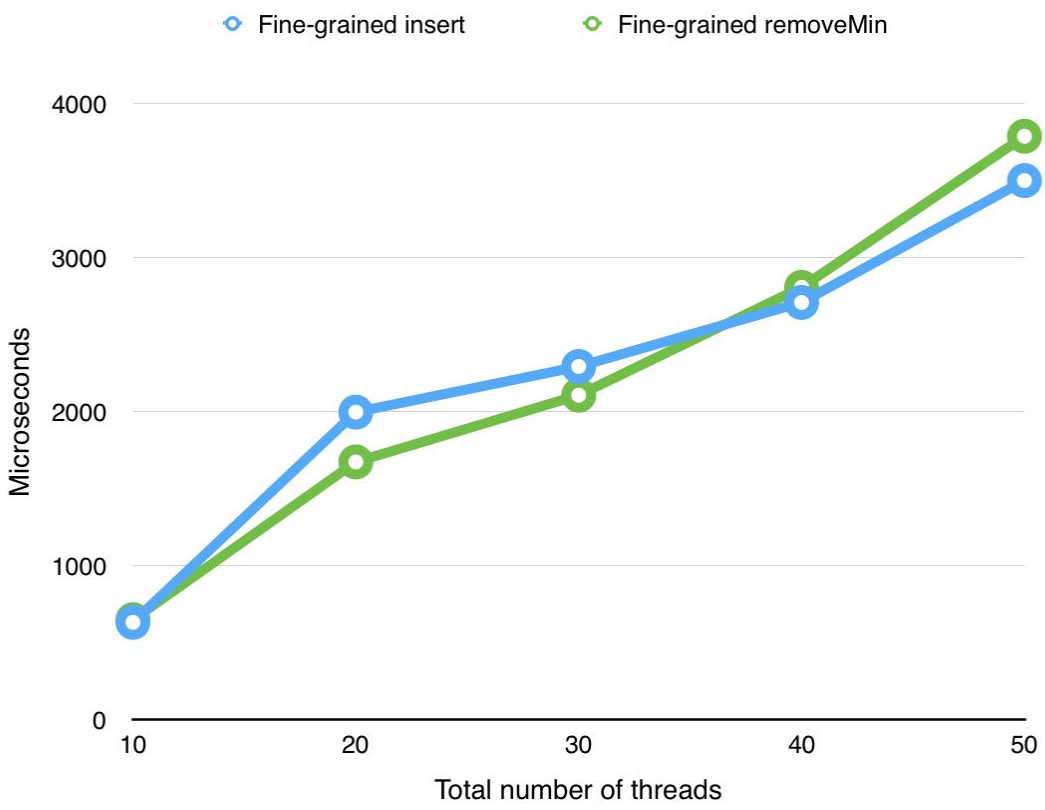
An insert operation inserts the data in the “rightmost” node, so that heap remains full, and it keeps swapping the new node with its parent as long as the priority of this node is more than that of the parent. In insert the Status tags are used as follows:

1. If the Status of parent is AVAILABLE, and the tag of current node is equal to the insert operation’s thread ID, no interference has occurred.

2. If the Status of parent is empty, then the inserted item is moved to root by some other delete operation and insert operation is finished.
3. If the Status of current node is equal to insert operation’s thread ID, then it was moved upward by some other delete operation.

Here, insertions don’t always have to traverse the whole height of the heap, so best-case time complexity on insert is $O(1)$. However, the worst-case is still $O(height = log\, M)$ where M is the size of the heap.
Time complexity for delete operation is also $O(height = log\, M)$ where M is the size of the heap.
This algorithm requires $1 + 3log\, M + (L + log\, P)\, M$ memory where
P = number of threads
M = size of the heap and
L = memory required by each lock (since each node has it’s own lock).

Comparison of the average Insert(x) and removeMin() time:



Skiplist-Based Concurrent Priority Queues

Introduction

The performance of the Fine Grained Locking heap algorithm presented above 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, balanced search trees and heaps suffer from the typical scalability impediments of centralized structures: sequential bottlenecks and increased contention. Hence Lotan, I. and Nir Shavit proposed a solution to design concurrent priority queues based on the **highly distributed SkipList** data structures.

The New Approach - SkipQueue

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*) .

Implementation:

```
bool findNode(int prio,skipListNode<X>* preds,skipListNode<X>* succs)
```

This function returns a vector of all nodes that should be just before priority = prio as predecessors, it populates *preds. Similarly, it populated *succs with the successors of priority = prio. This function also helps in physical deletion. It checks the marked field of each node and if it is set, it links the next field of previous node to the current's next. If there is a node in the list with priority prio, it returns true; else it returns false.

```
bool insert(int priority, X element = 0)
```

This function first chooses a maxLevel for the new node randomly (note: the generator is geometrically distributed to simulate coin flipping). It then calls the findNode function to get the list of predecessors and successors of the new node. It checks the return value of findNode, if it is true - it returns false (duplicate priorities aren't allowed). Else it sets the next field of the newNode to its successors at each level. It then tries to atomically swap the pred's next field to newNode after checking it currently points to the successor at

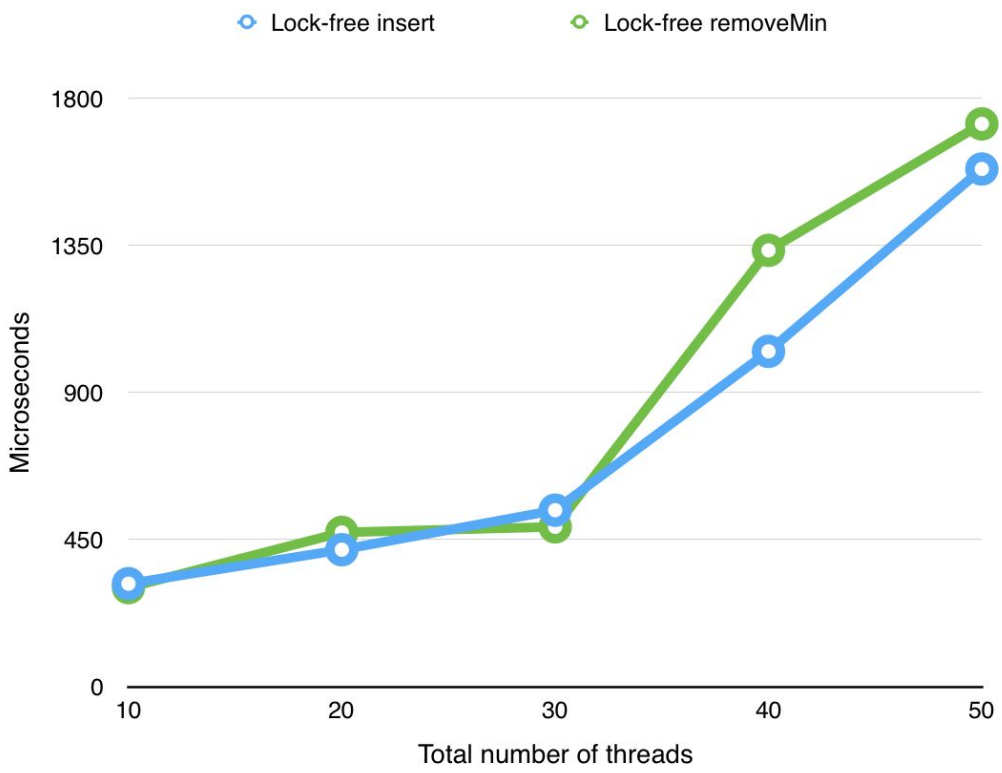
that level. This is done for all levels from 0 to topLevel of the newNode. In case of failure at CAS, it calls findNode to find the new pair of predecessor and successor list and retries the same operations.

How is removeMin carried out?
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 has not been set. It sets the flag marking that this node is logically deleted. The processor now uses the delete operation to physically remove it from the list.
No two processors will ever delete the same node since only one could have set its deleted flag. Why? 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.

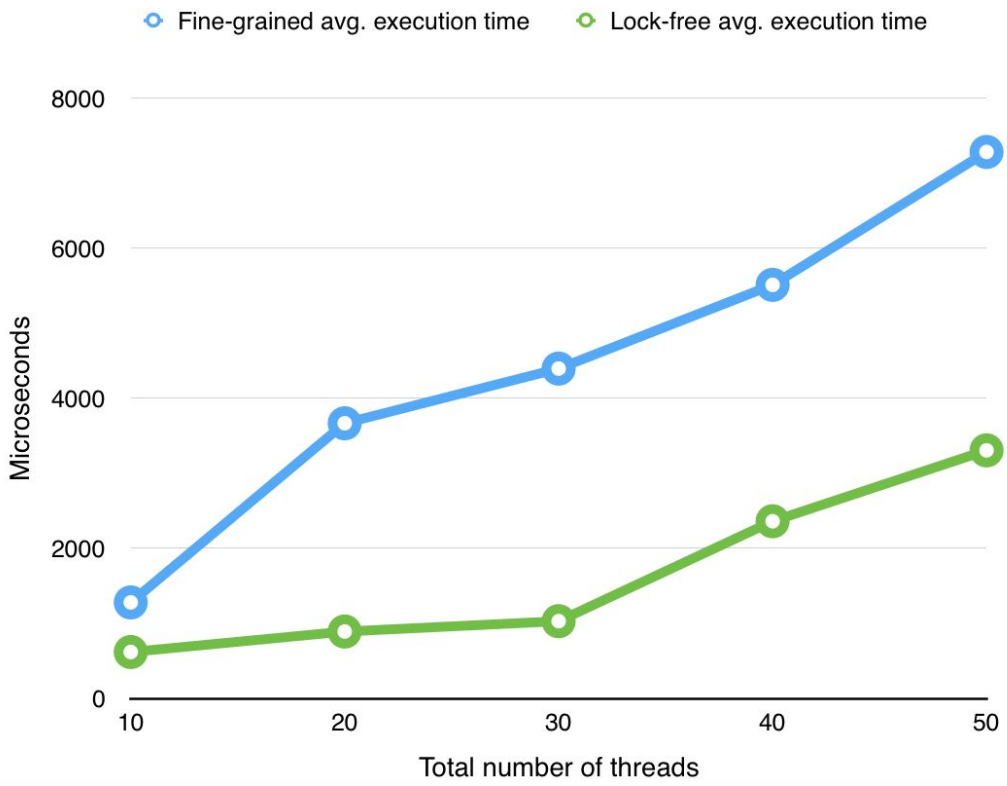
```
skipListNode<X>* findAndMarkMin()
```
This function traverses the bottomLevel to find the minimal node whose flag has not been set. Once it finds such a node, it tries to atomically set its deleted flag. If it succeeds it returns the node; else traverses further down and tries again. In the process, if it reached tail, it returns NULL.

```
bool remove(int priority)
```
This function sets the marked field of the node with priority field = priority. It then calls findNode(...) to remove the node physically.

Comparison of average Insert(x) and removeMin() time:

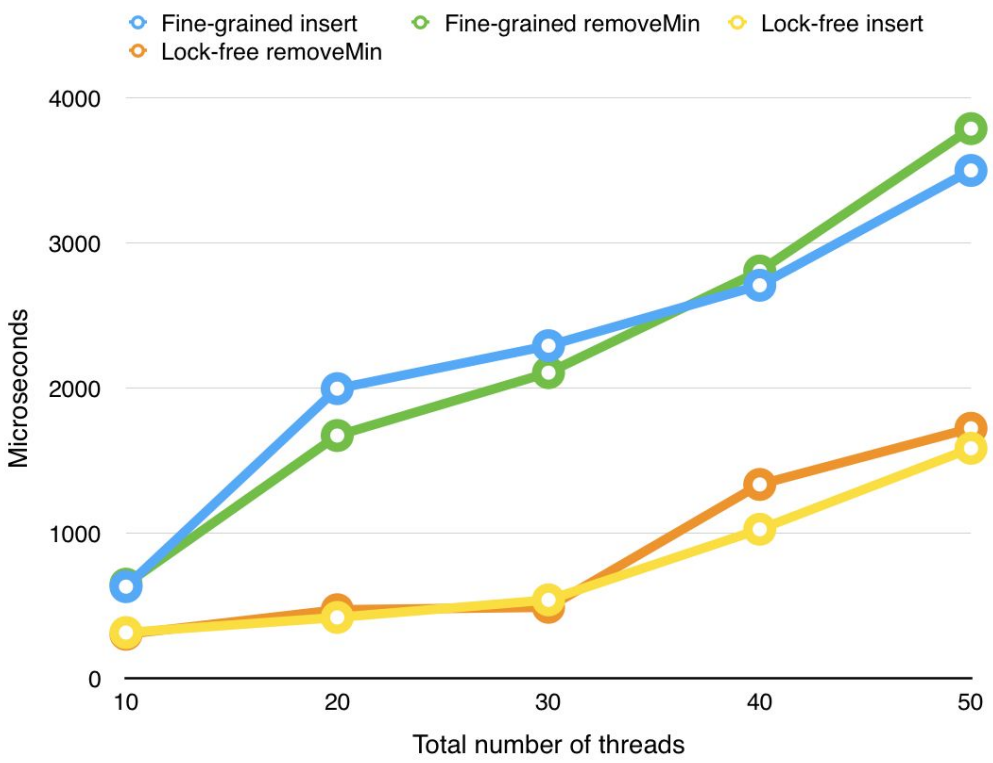


Comparison of average execution times of Fine-grained locking based priority queue vs Lock-free skiplist based priority queue:



There is a substantial reduction in average execution time when we switch to lock-free skiplists based priority queues.

Comparison of average Insert(x) and removeMin() in fine-grained locking based priority queues and lock-free skiplists based priority queue:



Why skiplists work better than heap-based or tree-based structures?

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. This is in contrast to the strict structural and ordering property of the heaps. 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. This way only one level of a node needs to be locked at any given time when a node is inserted or deleted. The concurrent performance of the list benefits immensely from this feature.

Also, assuming that the insertion 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.

Other advantages:

- 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.

Major Contributions:

- Implementation of move - copyable templated Reentrant Locks in C++ for the fine-grained heaps.
- Implementation of templated AtomicMarkableReference class in C++.
- Using atomic RMW primitives with high consensus number instead of locks to implement lock-free concurrent priority queue.