Skip Lists

Alec Bargas
Julian Domingo
Jacob Ingalls
John Starich
EE 360P Concurrent and Distributed Programming
Class: T TH 3:30 - 5:00 PM
Repo: github.com/JohnStarich/java-skip-list

Abstract

In concurrent applications with big data, the ability to modify large lists concurrently becomes critical. Using traditional, globally-locked, concurrent data structures we can achieve concurrency but, at the cost of modification speed as the entire structure must be locked during a modification. In this paper we explore creating fine-grained and lock-free skip-lists and compare their performance in the Java library.

Introduction

A skip-list is a data structure designed to allow for fast searching like a B-Tree, but also allow for fine-grained concurrency like a linked list. We implemented lock-free and fine-grained skip-lists, showing that we can get comparable performance with our implementations. Lock-free means that we use atomic actions instead of locks (or semaphores), we expect that this will give us a performance improvement as we will not have to perform lock arbitration. Fine-grained means that a small subset of the list will block other modifications, instead of the entire list blocking.

Table 1: Big-O of Linked List, Binary Tree, and Skip-List

Operation	Linked List	Binary Tree	Skip-List
Access	$\Theta(n)$	$\Theta(log(n))$	$\Theta(log(n))$
Search	$\Theta(n)$	$\Theta(log(n))$	$\Theta(log(n))$
Insert	$\Theta(1)$	$\Theta(log(n))$	$\Theta(log(n))$
Remove	$\Theta(1)$	$\Theta(log(n))$	$\Theta(log(n))$
Space Complexity	$\Theta(n)$	$\Theta(n)$	$\Theta(n\,log(n))$

In Table 1 we compare the various speeds of linked lists, binary trees, and skip lists. Linked lists are slow to access and search as we have to traverse each node in the list to get to the next. However, it is very fast to insert and remove a given node as we can get right to it and swap the pointers around. Binary trees are moderately fast all around, $\Theta(log(n))$, and a good in between in performance of linked list and arrays. However, the entire tree must be blocked off during an insert, delete, or modification. When we make a change to node n, n has a fairly good chance of moving to a different place in the list. This move forces the rest of the list to rebalance, and would force another process to start over from the new tree. A skip-list solves both of these issues by making modifications to the list like a linked list, but at the same time having a layered structure internally. From this, it is moderately fast and allows for fine-grained locking.

Implementation

A skip-list is a sorted linked list with several layers that enable searches to skip forward various distances in the list, as shown below [1]:

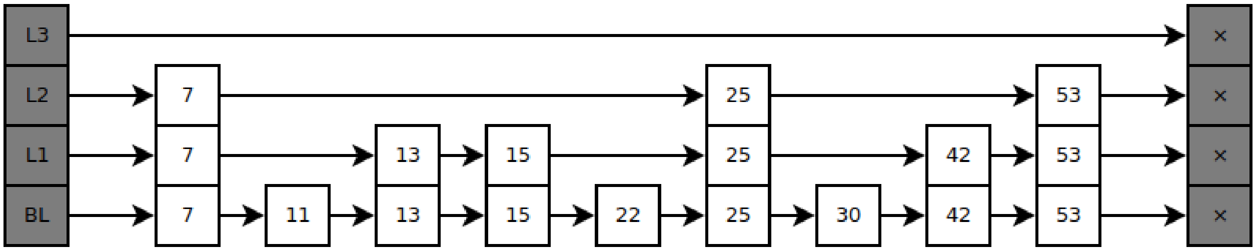

Figure 1: skip-list diagram

We implemented the probabilistic skip-list where the insertion of an element has some probability p that it will be inserted in the current level vs the next one. This allows for a probabilistically even distribution of links such that we can obtain $\Theta(log(n))$ insertion time.

Design Alternatives

In this paper we explore design decisions specific to the lock-free and fine-grained implementations of a skip-list. For the lock-free version, we made the decision to use Atomic References as opposed to Atomic updates to the nodes. For the fine-grained version we considered creating locks on the node, or the node/layer pair.

In the lock-free skip-list we debated on whether we should atomically update a boolean that says if the node is valid, or if we should use markable references to forward nodes. If we were to invalidate the entire node, we could quickly stop traversal through the list, as we do not need to iterate through the forward node array to determine if we should stop iterating. On the other hand if we mark the references, then we can stop before accessing a node that is being updated. Additionally, if we mark the reference, then we can traverse layers above the modification, removing an effective lock from a large portion of the list. We believe that using atomic, markable references is the better choice.

To ensure the layers of the fine-grained skip-list are sublists of lower layer lists, modifications to the skip-list should only occur once all locks are obtained for nodes needing modification. As a result of the internally layered structure of a skip-list, locks are retrieved for all predecessor nodes up to and including the highest layer of occurrence of a node to add or remove. The predecessor nodes point to either the correct location for a new node (add() operations), or to the node to remove (remove() operations). In our implementation, each node is associated with a single lock that locks the node at every layer. An alternative implementation could lock each layer of a node separately. Once a thread finishes their modification, the thread unlocks all locks belonging to it, allowing other threads to acquire those locks or locks passed them. This implementation guarantees deadlock freedom; when a thread locks a node with a search key k it will never acquire a lock on a node with a search key $>= k$. From an implementation perspective, this means locks are acquired from the lowest layer upwards. Furthermore, concurrent modifications are guaranteed as long as there aren't overlapping search key values. It would be more difficult to ensure deadlock freedom if each node had separate locks on each layer, because multiple threads could have access to different parts of the node at the same time. On the other hand, the implementation is blocking; it prevents other threads from completing operations on the skip-list which don't have the locks. This results in a significant time/memory overhead: a thread must retry its operation until it can successfully acquire the lock(s) it needs, and every node must have their own instance of a ReentrantLock. If we were to lock each layer of a node separately, the node would not be blocked for as long and processes could access different parts of a node at the same time.

Performance Comparison

In figures 3 and 4 in Appendix B, we show the performance of our two implementations as compared to Java's built-in *java.util.concurrent.ConcurrentSkipListSet*. As we can see, we achieve similar performance to Java, with some minor differences. For example, we can see that our Fine-grained solution is slower on average than a lock-free implementation like java's.

Lock-Free

We expected a lock-free solution to be faster than a locked solution as less time would be spent in lock arbitration. However, our implementation is significantly slower than both Java's and our fine-grained alternative. We found that a lock-free implementation of a skip-list has many edge cases we must consider, as we must allow for an insert, delete, and iteration to occur at the same time on the same node. We believe our slowdown to be due to our code designed to prevent deadlocks, however we still seem to have deadlocks. For the rest of this paper we will use Java's implementation as the reference point, as java used a lock-free implementation [2].

Fine-Grained

The fine-grained implementation performs worse the the built in implementation, a lock-free implementation. We see that on average our implementation is slower, and seems to have a low $\Theta(n)$ coefficient. Our argument is that all operations in a skip-list are $\Theta(log_2(n))$.

We examined the performance of the skip-list by varying the number of rows modified and the number of threads in contention. From the data we gathered we could see that the performance of the data structure followed closely with that of Java's implementation. We ran our tests ten times, ignoring the first few runs to allow the JVM to compile and optimize the code. Then, we increased the number of threads running the test. Each thread performs insert and remove tasks that duplicate or preempt each other. This allows us to examine the performance gain in the locking mechanism alone.

Conclusion

We implemented both a lock-free and a fine-grained version of a skip-list. We used AtomicBooleans in our lock-free version and a ReentrantLock on each node in our fine-grained version to ensure mutex. Both of our implementations were slower than the ConcurrentSkipListSet from the Java library. The lock-free skip-list was difficult to implement due to the structure of a skip-list. It was difficult to ensure deadlock freedom while making a change because there are multiple nodes that can be affected by each change. Although it only performed on a small set of nodes, the lock-free version was significantly slower than both of the other implementations. The fine-grained list underperformed when there was little concurrency, but as the number of threads operating on the list increased, the performance of our version came very close to the ConcurrentSkipListSet.

Bibliography

[1] Ticki. (2016). Skip Lists: Done Right. Available: https://ticki.github.io/blog/skip-lists-done-right/.

[2] Pediaview. (2017). Java ConcurrentMap. Available:
https://pediaview.com/openpedia/Java_ConcurrentMap.

Appendix A: Educational Material

Skip-List Data Structure

A skip-list is a sorted linked list with several layers that enable searches to skip forward various distances in the list, as shown below:

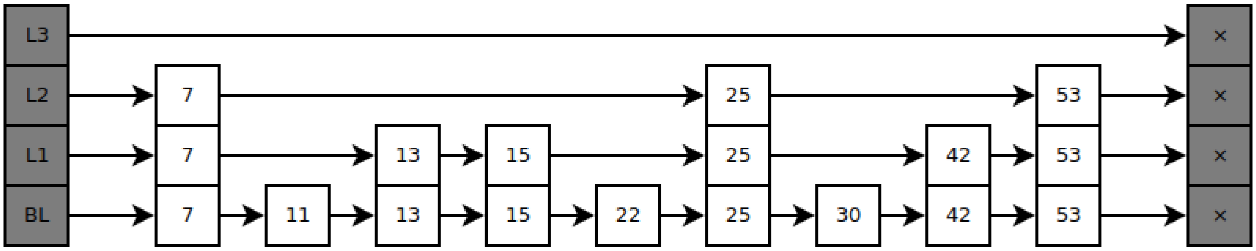

Figure 2: skip-list diagram

At the lowest layer (BL), a skip-list looks very similar to a sorted linked list. As you progress to higher layers, fewer of the elements in the list are included. If an element is present in a given layer, it is also present in all layers below that layer, forming a column references to the next nodes in a single node.

Searches in the skip-list begin at the highest layer and progress down the hierarchy. The search progresses through each layer until the algorithm either finds the target, or a value that is larger than the target. If the search finds that the target is not present in a given layer, the search travels down a layer in the column of the largest value that is not greater than the target. The search continues in the new layer and repeats the process until either the target is found or the element is not found in the lowest layer. This type of search and structure results in $\Theta(log(n))$ searches.

As an example, let us search for the element 30 in the above list. First we look at the highest level, L3. There are no nodes in this layer yet, so we move to L2. The first node is 7, which is less than 30, so we continue searching through L2. We do this until we reach 53, which is the first element in L2 that is greater than 30. We then move down into L1 in the column that came before 53, which is 25. We then go from 25 to 42 in L1 and see that 42 is greater than 30. So we move down to BL in the 25 column. We search BL until we reach 30, and return that we found the target. If we were instead searching for 27, the process would be the same until the search in the BL layer. After moving to the BL layer, we move to the next node which is 30 and notice that 30 is greater than 27. Since we are in the lowest layer, which includes all elements in the skip-list, we can conclude that 27 is not in the skip-list.

Adds and removes are extensions of searches (so they are also $\Theta(log(n))$ operations). For the add method, a search is conducted for the value to be added. If the target is found, the add returns without modifying the skip-list. If it is not found, a node with the target value is added immediately before the first node that is larger than the target on the lowest layer. The algorithm then randomly decides whether to include the new node in the next highest layer or not. This promotion continues until either it is not chosen to be promoted to the next layer or it reaches the highest layer. This will ideally result in a Gaussian distribution with only a few nodes in the highest layers.

To remove a node, a search is conducted for the value to be added. If the target is not found, then remove returns without modifying the skip-list. If it is found, the node is marked as removed, but is not actually deleted from the structure. A node that is marked as removed is no longer considered as included in the list for further operations. The reference to the deleted node in the preceding node is then replaced with the reference to the node that followed the deleted node.

Fine-Grained Skip-List

In order to create a concurrent version of a skip-list, we first used ReentrantLocks to ensure mutex on alterations. Modifications to the skip-list only occur once all locks are obtained for nodes needing modification. These locks come from the nodes that precede the node to be modified on each level since their next node pointers might be updated. While making modifications to a node, a fullyLinked boolean is set to false and is reset once the modifications are done (at which point the obtained locks will also be released). Once a node is removed, the markedForRemoval boolean is set to true, telling other operations to ignore the node. Atomic values for the list size and number of layers in the list are updated as needed and ensure consistent values.

Lock-Free Skip-List

A lock-free version of the skip-list can use AtomicBooleans for the fullyLinked and markedForRemoval conditions described above to ensure mutex. Additionally, the nodes do not have ReentrantLocks. While making changes to the list, CAS on fullyLinked is called for the node on each layer that precedes the node to be changed/added. Similarly, when a node is removed, CAS is called on the markedForRemoval boolean.

Appendix B: Performance Data Graphs

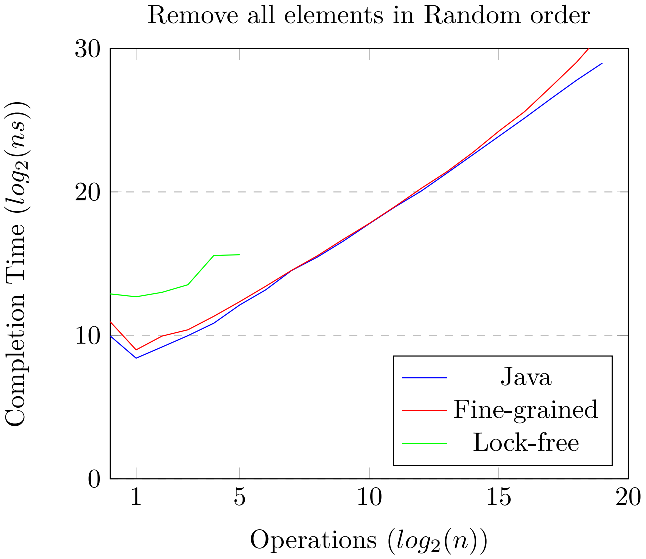

Figure 3: Remove Random

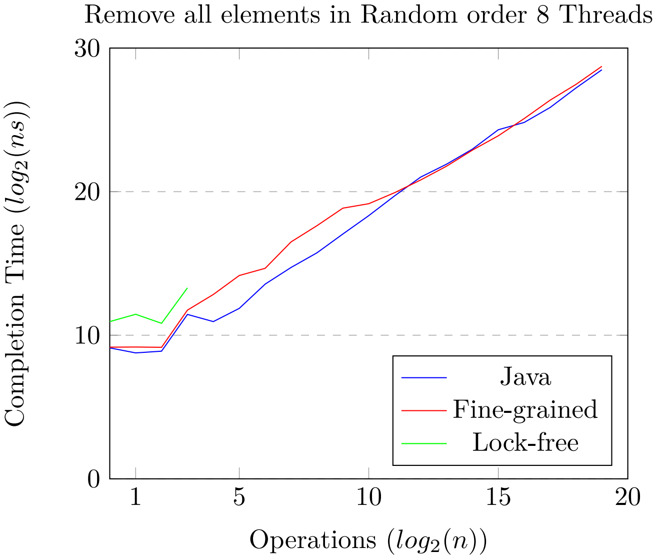

Figure 4: Remove Random 8 Threads