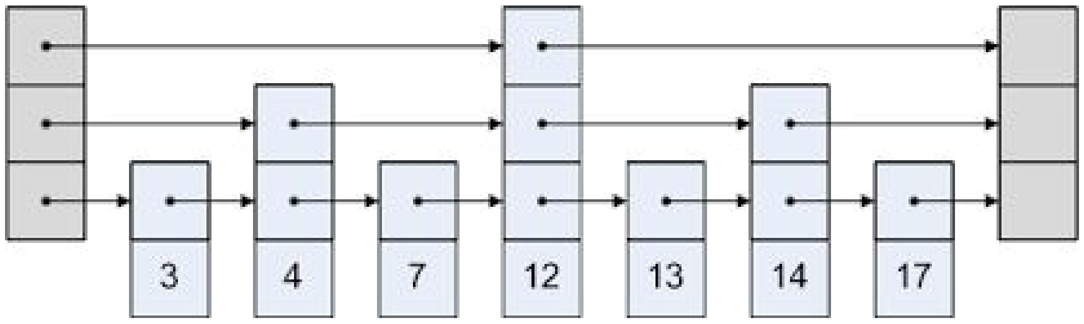
# Concurrent Skip List



**Dejan Grubisic**

Dec 16. 2020.

Comp522 Multi-core Computing, Rice University

https://github.com/dejangrubisic/ConcurrentSkipList

# ROADMAP

## INTRODUCTION

*In this report, we are going to discuss the challenges and solutions to the implementation of concurrent skip lists. Additionally, we are going to see the analysis of one particular custom implementation, created as a part of the COMP522 project.*

A general skip list is a data structure that implements a dynamic ordered list with average insertion time *O(logN)*, deletion time *O(logN)*, and searching time of *O(logN)* [1]. Differently than an ordinary list, skip lists have multiple next pointers per each node, that connect the list in multiple layers. Next pointers are organized in levels and their number is not the same for each node (Figure 1). Nodes present at the same level will be connected from left to right, sorted by their keys.

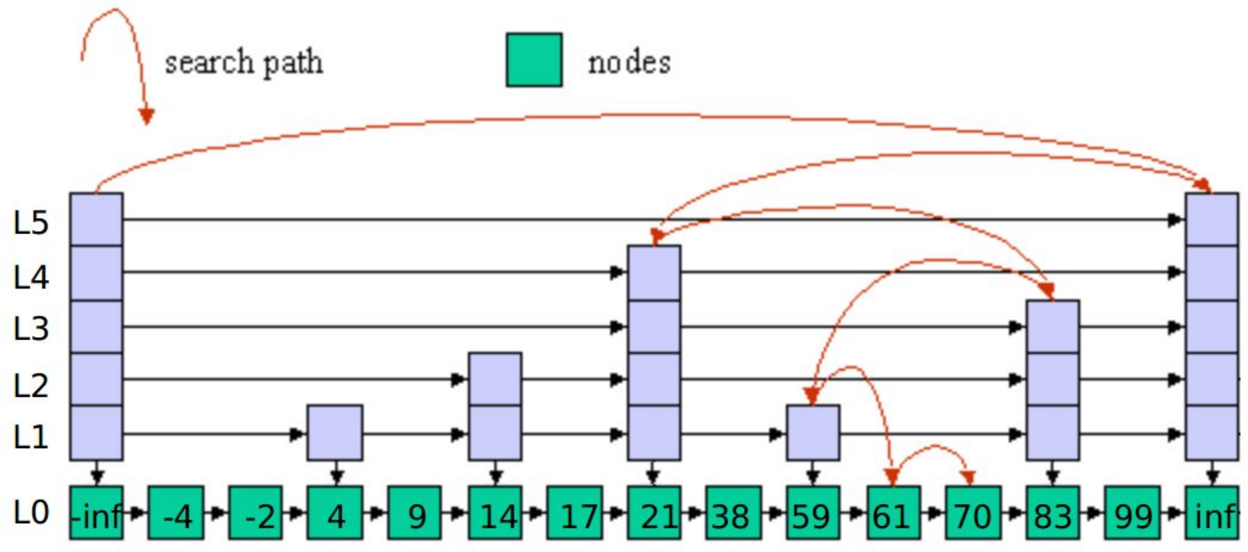


*Figure1. Skip list search for key 70*

To guarantee specified time complexity, it is enough to just randomly assign the height of each new node from 1 to the number of levels. The number of levels can be adjusted dynamically during the program and their number should be *logN*, where N is the number of nodes.

In order to perform a search, insert or delete the node with a specified key, we need to traverse to the skip list to the spot where it would be sorted. To do so, we are starting from the top-level and iterate to the right until we find a node with the higher key. Then we step back to the previous node in that level and perform the same operation on the one level below. Following this procedure, we will be in the right place when we come to level 0.

## PROBLEM-SOLVING

1. How to implement thread-safe insert
2. How to implement thread-safe delete

## 1 How to implement thread-safe insert

Let's see now how we could implement a thread-safe version of the skip list. With multiple threads, several things make this problem much harder. First, we need to find a way to keep the skip list sorted and consistent. If two threads are trying to insert the node in the same place, we have a race condition and one of the keys may not be inserted (Figure 2). To avoid this situation every next pointer must be atomic. This is still not enough. Once we make our new node point to the next node, we need to be sure that the previous node still points to the same place and if that is true, we can atomically direct the previous node to the new node. For this purpose, we can use a compare-and-swap (*CAS*) operation.

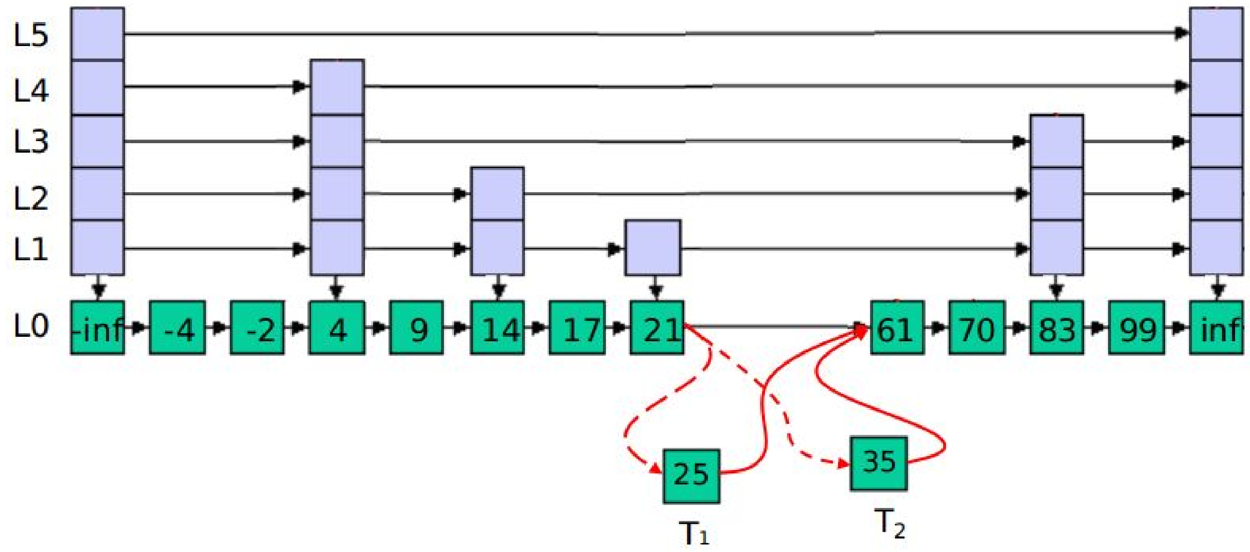


*Figure2. Insert with multiple threads - race condition*

But what happens if somebody else was inserted before us and our compare-and-swap fails? In that case, we need to traverse again on the same level, and to try again. Based on the key of the newly inserted node, three possible scenarios could happen:

1. If the new node key is greater than our node, we need to point our node to the inserted node and try *CAS* again
2. If the new node key is equal to our node, in this case, we can overwrite the value
3. If the new node key is less than our node, we need to check what is the next pointer of the inserted node, and if it is the same as the new node next, we can perform *CAS*.

Once we inserted a new node on level 0, we have a stable base for building the tower. Luckily, we can reuse our code and perform the same procedure. The only requirement is that we know which node on the upper level should point to our new node. We can collect this information for free during our traversal towards level 0, by saving pointers to all nodes on which we changed level (red rectangles in Figure 3).

## 2 How to implement thread-safe delete

At this point, we know how to insert in the concurrent skip list if only insertions are allowed. Let's check out what will happen if we also include deletions. Imagine what would happen if the node with key 14 is deleted before we successfully connect our new node to it. If the node is completely freed from memory, we would have a pointer to unallocated space in memory which would cause some issues.

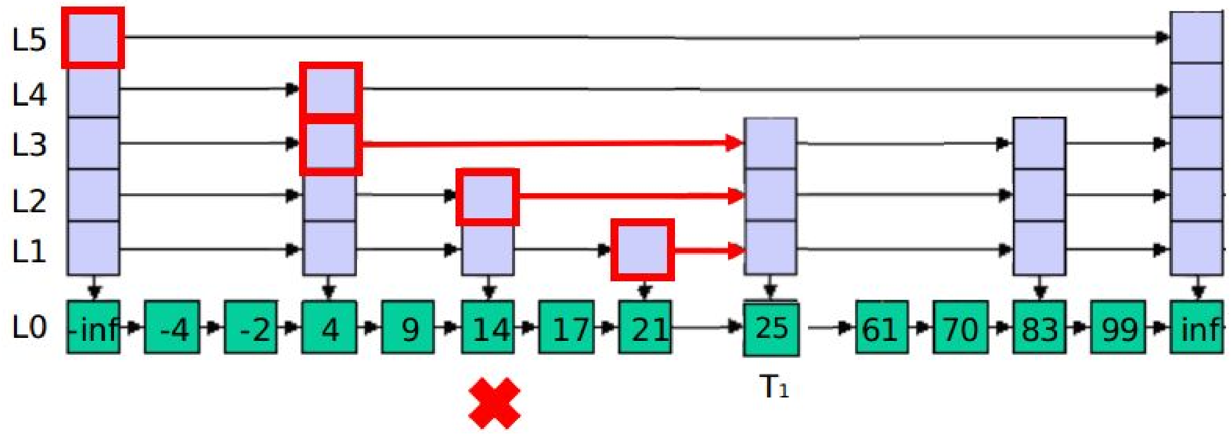


*Figure3. Building the tower for the new node*

To escape this situation, and completely free node 14, we would need to somehow to change all pointers to node 14 to its next nodes atomically. Furthermore, we would need to traverse to our node every time we want to build one level for our new node in order to find the right previous node to connect. As you can see, such implementation would include significant overhead and would be hardly implementable.

Instead of the previous solution, we can simply mark delete bit, which would say if a node is valid or not. In the case we are searching for some key and the node with that key has delete bit set, we are going to return *false*, but other than that we are going to treat deleted nodes the same as all other nodes for all other operations. By doing so, the only operation that can change the structure of the skip list is insert, and we can forget the problem from Figure 3.

On the other hand, the delete bit introduces a new problem of memory consumption. For applications that are constantly inserting and deleting new unique keys, the skip list could end up with a lot of nodes with a set deleted bit. The first solution to this problem could be to do implement sequential cleanup which would remove all deleted nodes. This approach would require locking the whole data structure for all other operations, which wouldn't be suitable for many applications.

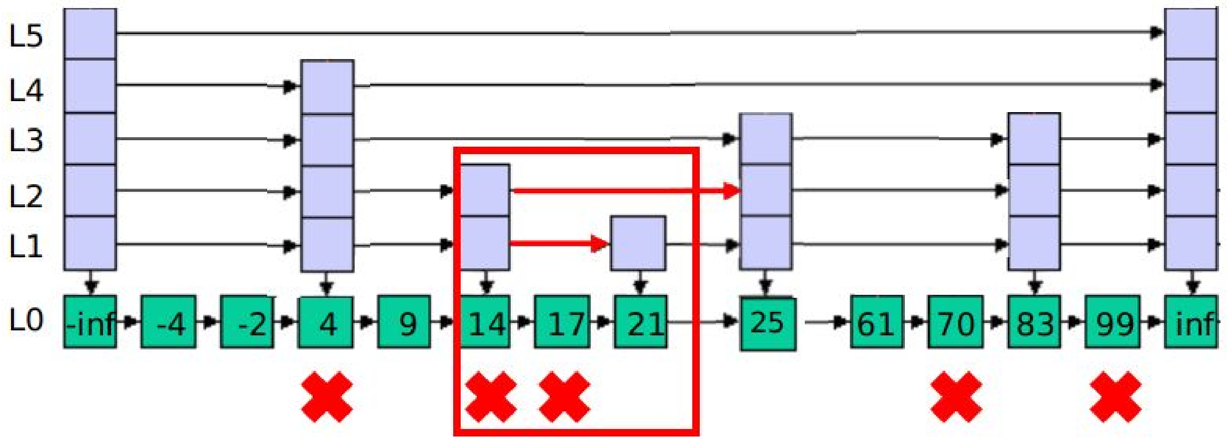


*Figure4. Removing nodes with delete bit set with node lock*

Another option is to perform a cleaning operation on a part of the skip list by locking one node, which would transitively restrict the access to all next nodes with a lower tower up to the first higher node than the locked node. To make sure that there are no other nodes that are inserting in the same part, we can create a shared array, which will contain what key are threads currently working.

For different implementations of a concurrent skip list check the algorithms described in [2].

*Note:* For the custom implementation of the skip list implemented as a part of the *COMP522* project, sequential deletion suffice, and no locking was used.

## EXPERIMENTS

For measuring performance over the congestion in the concurrent skip list, I created three different modules that test:

1. 100% insertions (Series1)
2. 100% deletions (Series2)
3. 50% insertions, %50 deletions (Series3)

In the first experiment, threads are taking random numbers in the range from 0 to 255 and insert them into the empty skip list. In the second experiment, each thread is deleting unique values until the skip list becomes empty. In the third experiment, each thread is getting a unique set of random numbers from the same range and they perform one insertion and one deletion of each number in an interleaved fashion. For each of these modules, test the performance of the execution of 51.2k 102.4k 204.8k 409.6k819.2k 1638.4k operations with 1, 2, 4, 8, 16, 32 threads. The experiments are based on the execution of ppc64le on llnl.cs.rice.edu (Table 1). The obtained results are shown in Figure 5.

| Experiments Architecture | |
|---|---|
| Architecture: | ppc64le |
| Byte Order: | Little Endian |
| CPU(s): | 160 |
| On-line CPU(s) list: | 0-159 |
| Thread(s) per core: | 4 |
| Core(s) per socket: | 20 |
| Socket(s): | 2 |
| NUMA node(s): | 6 |
| Model: | 2.2 (pvr 004e 1202) |
| Model name: | POWER9, altivec supported |
| CPU max MHz: | 3800.0000 |
| CPU min MHz: | 2300.0000 |
| L1d cache: | 32K |
| L1i cache: | 32K |
| L2 cache: | 512K |
| L3 cache: | 10240K |

*Table1. Architecture characteristics of experiment machine*

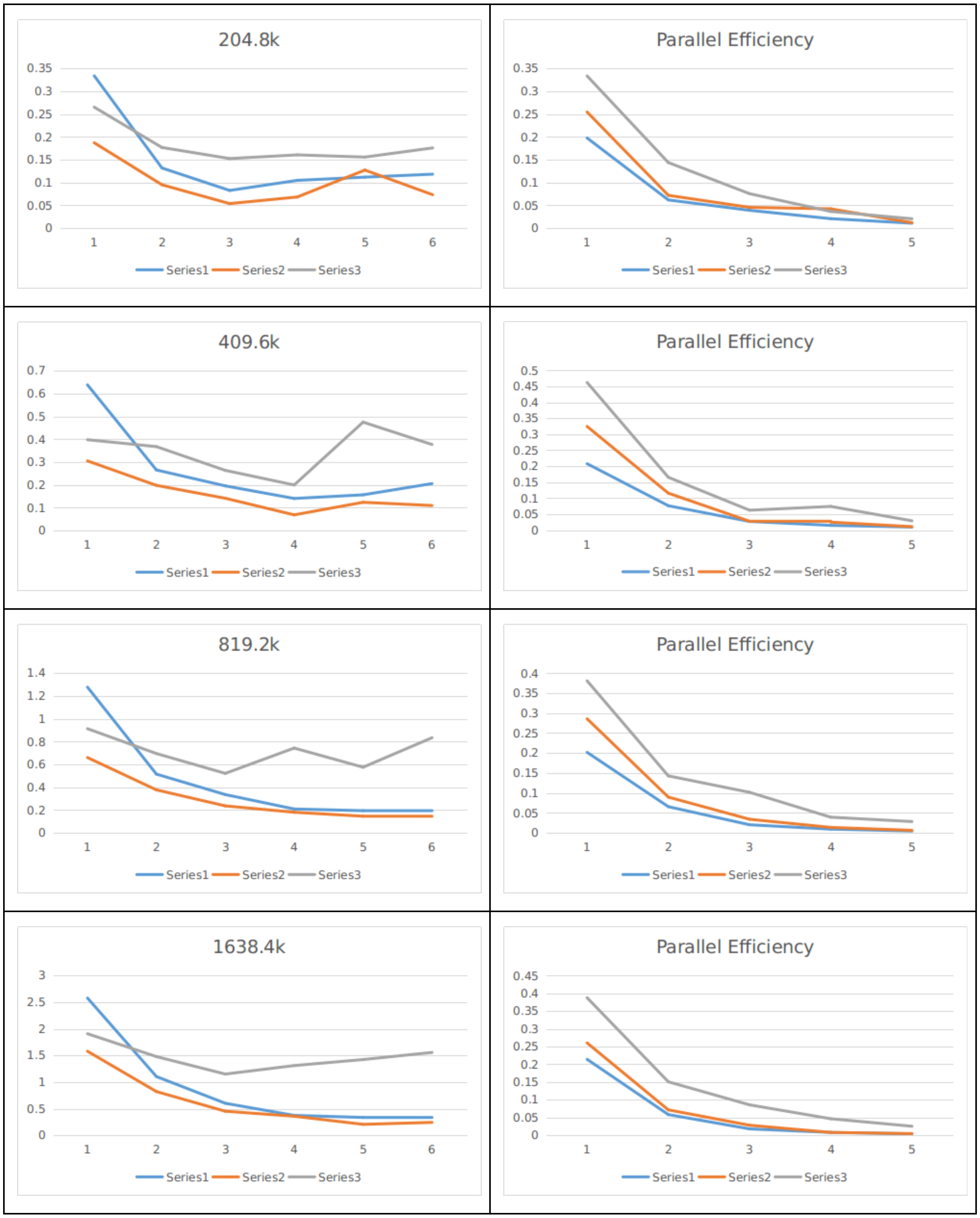


*Figure 5. Experiment results (elapsed time and parallel efficiency)*

From the obtained results, we can see that serial implementation needs the most time for insertion, followed by 50%insertion-50%deletion and finally deletion. On the other hand, parallel execution spends the most time on 50%insertion-50%deletion, while insertion and deletion take a similar time. Although parallel efficiency is not good, parallel implementation is still better than serial and could be improved by decreasing the number of compare-and-swap operations. If we don't need the number of non-deleted nodes in the skip list we can get rid of one compare-and-swap on every delete, which could save some time.

## CONCLUSION

In this report, I am trying to describe challenges and solutions for concurrent skip list implementations. In contrast to a sequential version, a parallel version has many challenges that include race condition on an insert, building the tower, and all the problems that arise with node deletion. The implementation of a concurrent skip list adjacent to this report is still in the development phase and requires further engineering. So far, the current implementation correctly implements insertions and deletions while providing better execution time than serial implementation. For large scale use, some additional work is required.

## REFERENCES

[1] Skip Lists: A Probabilistic Alternative to Balanced Trees

[2] https://dl.acm.org/doi/10.1145/2688500.2688501