Pardis21 Lab3
Report

Buonomo Fanny, Alagappan Sriram

October 13, 2021

Contents

1 Introduction

The goal of this lab was to make us implement a free-lock SkipList, and familiarize us with different ways of checking if a data structure is linearizable or not. We started by creating the list, populating them and then exposing it to different operations (add, remove, contains). We tried 3 different ways to see if the SkipList is linearizable. First, we used a global lock, then we created separated, local histories, either merged at the end of the execution or on the fly. In this report, we will describe and analyse our results.

2 Populating the list and first tests

We created 2 lists. The first one was filled with an uniform law, with 10^6 items $\in [0; 10^6]$. The expected mean is $\frac{10^6}{2} = 500000$ and the expected standard deviation is $\frac{10^6}{\sqrt{12}} \simeq 288675$. We filled a second list, using a normal law of mean $\frac{10^6}{2} = 500000$ and standard deviation $\frac{10^6}{\sqrt{12}} \simeq 288675$, in order to have a similar distribution.
We obtain the following results, which are close enough to the expected values.

	Mean	Standard deviation	Number of items
Expected	500000	288675	1000000
Uniform distribution	500136	252564	574060
Normal distribution	500459	288643	631765

Then we exposed the list to 4 different scenarios, with 10^6 operations:

- 90% adding, 5% removing, 5% contains
- 50% adding, 50% removing, 0% contains
- 50% adding, 25% removing, 25% contains
- 10% adding, 10% removing, 80% contains

The figures 1 and 1 shows the execution times for the uniform and the normal lists, for different numbers of threads. We see that the results are similar for both lists. We observe the best executions times are obtained for N=12. N=2 is too small and there are too many operations. With N=30, we have more conflicts between the thread, and the execution is slower. We thought it would be the same for N=46.
We observe that the scenario *0.1;0.1;0.8* is usually the fastest. Since the contains method does not change the content of the list, if we have a majority of it, we should not have too many conflicts.
We observe that the two scenarios *0.9;0.05;0.05* and *0.5;0.25;0.25* are usually the slowest. In both cases, we have a majority of add/remove. These methods have to try all over again when there is a conflict between 2 threads, slowing down the execution.

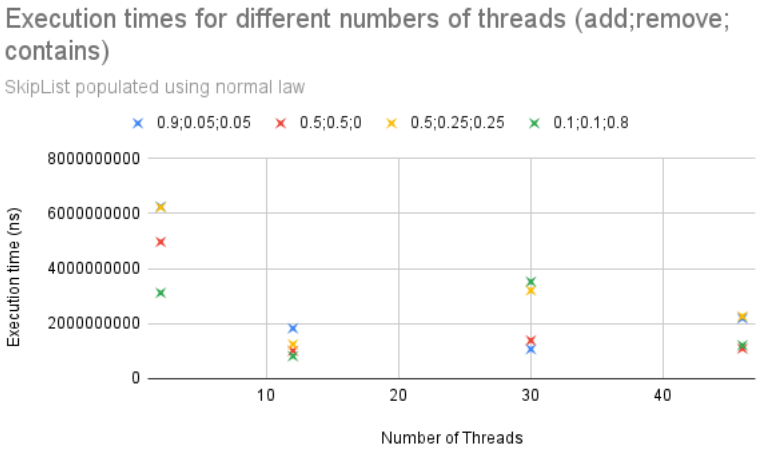

Figure 1: Execution times, for the 4 scenarios, versus the number of threads, with the SkipList populated with the normal law

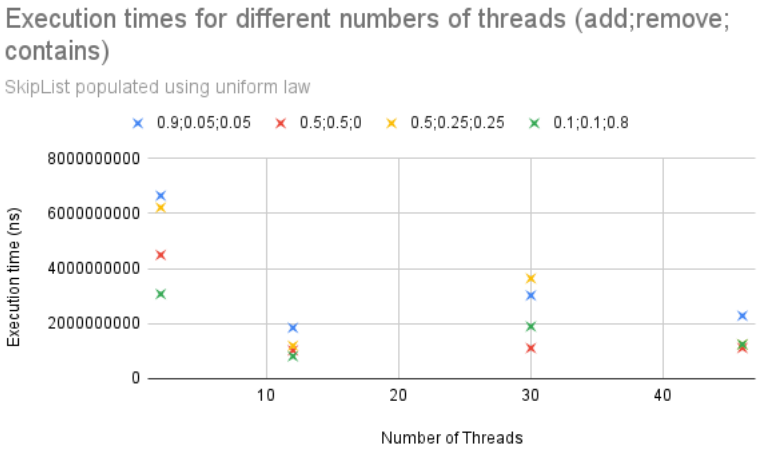

Figure 2: Execution times, for the 4 scenarios, versus the number of threads, with the SkipList populated with the uniform law

3 Linearization points

In order to check if the lock free Skip List is linearizable, we can construct the history of the execution, and check that it follows sequential consistency (cannot remove an item not in the list, cannot add an item already in the list, ...). To do so, we can sample the time right after the linearization point. We use 3 methods to do so, and the results are presented below.

3.1 Global lock

The first method consists of using a global lock, around the linearization point and the sampling, making the two actions atomic. While it helps us to see that the free lock Skip List is, indeed, linearizable, it slows down the execution a lot. In figure 3, we compare the execution times for the list without the global lock, and the list with. We saw before that the execution times for the uniformly and the normally distributed lists are the same, so we took the average between the 2 execution times. Each symbol corresponds to a different scenario. The blue series are without global lock, while the red series have the global lock.

Figure 3: Comparison of execution times with, and without, global lock

We can observe that the red symbol is almost always above the blue symbol. Since all threads share the same global lock, each action will take longer. But, in the other hand, it reduces the probability of conflicts, where add/remove have to try all over again. It could explain why the difference in the time execution is not always significant.

3.2 Merging logs after execution

The first approach to logging without a global lock is to have each thread record its own history, and then merge the histories of all the threads at the end, ordering them by the timestamp. To test this logging method, we ran operations on the skip list that was distributed evenly between add and remove operations. This was repeated 10 times and executed on 46 threads. We started the integer value range at 10^7, wrote the logs, and verified sequential ordering. From the integer value ranges 10^7 to 10^5, the logs indicated no races. By 10^4, races were prevalent in every execution we ran.

3.3 Merging logs using a Multi Producer, Single Consumer

The second approach uses a multi producer, single consumer (MPSC) queue, where there are multiple SkipList threads that process the SkipList operations and produce logs, an a single consumer thread that orders and stores them. This logging method was tested exactly as above, with a 46 thread execution service running add and remove operations, each at a 0.5 probability. The integer range started at 10^7. From integer value ranges 10^7 to 10^4, the logs indicated no races. By 10^3, races were prevalent in every execution we ran. We compared the previous 2 implementations by measuring the average execution time during this test which was generated by running this experiment 10 times.

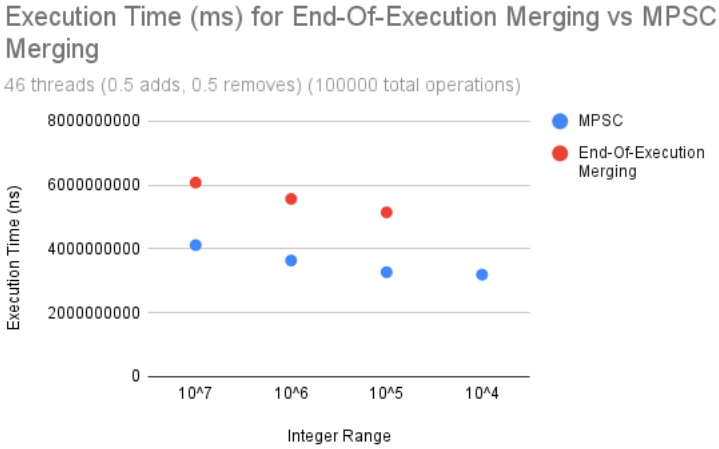

Figure 4: Comparison of execution times between two approaches to lock-free logging

The average execution time for both methods decreases as the integer range decreases. The end of execution merging method takes, on average between

the first 3 integer ranges, 1920.58 ms longer to execute than the MPSC method. There is no entry for End-Of-Execution merging at 10^4 as races were encountered at that range for this method.

4 Conclusion

To determine if a data structure is linearizable, or not, a global lock combined with time sampling seems to be, intuitively a good way. When we use a lock, we ensure that the history we write will follow sequential consistency, if the data structure is linearizable and if the linearization have been correctly identified. But it slows down the execution, and if the data structure is slow, or not optimised, testing it several times, with different scenarios and a big enough amount of data can be time and resource consuming.
A solution to that problem is that we do not use a global lock. But we have to keep in mind that the global history built that way may contain sequential inconsistency, because the linearization points and the following time sampling does not represent an atomic action. Using this method with a large enough range of data should reduce the number of conflict and allow use to test correctly the data structure.