Performance Comparison and Analysis of Lock-base and Lock-free Concurrent SkipLists

Yixing Wang(yw9639)

Weiyu Zhu (wz4245)

November 19 2018

Abstract

In this paper, we show three different implementations of concurrent skip lists: *coarse-grained*, *fine-grained* and *lock-free*. Upon completing the implementations, we test the performance of each version, on three commonly used operations: **add**, remove and **contains**. Multiple test benchmarks with different parameters (*number of operation*,*number of threads*,*operation types*) are applied for a complete coverage and comparison. We also include the Java standard library **ConcurrentSkipListSet** in comparison as reference. The Runtime results show that our fine-grained and lock-free have similar performance, which is much better than the coarse-grained one and worse than Java standard library. This shows our implementations are basically workable, but still could be optimized.

1 Introduction

1.1 Background

A skip list, proposed by William Pugh in 1989[1], is a linked list that is sorted by key, and in which nodes are assigned a random height, up to some maximum height. A node in a skip list has a number of successors equal to its height. We consider a skip list as several layers of lists. The bottom(lowest) layer is an ordinary ordered linked list and each other layer is a sub-list of the list at the layer beneath it and there are exponentially fewer nodes of greater heights. Figure 1 illustrates a skip list's structure in which the keys are integers.

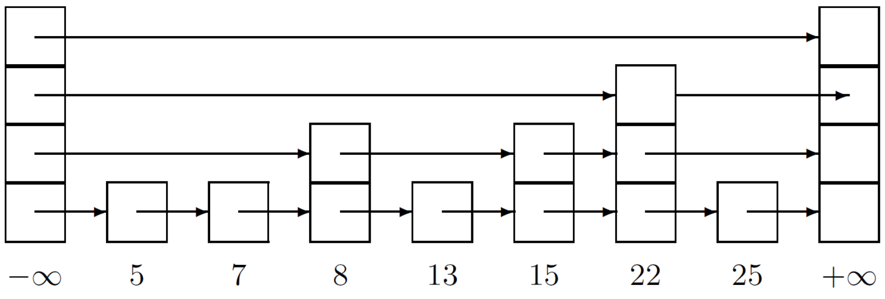

Figure 1: Internal Structure of a skip list[3]

A Skip list allows fast search within an ordered sequence of elements by searching first at highest layers, skipping over large numbers of smaller nodes and progressively proceeding to the next lower layer each time it encounters a node whose key is greater than or equal to the expected value until a node with the desired key is found, or else the bottom layer is reached. In this way, nodes can be effectively skipped, which leading to a better performance on searching with $O(logn)$ time complexity on average.

1.2 Description of Project

In this project, we implement three kinds of concurrent skip list supporting three basic operations - *add, remove* and *contains*:

1. Coarse-Grained Lock-based skip list.
2. Fine-Grained Lock-based skip list[3], presented by Maurice Herlihy, et al.
3. Lock-free skip list[4], presented by Herlihy and Shavits.

Then, we compare the performance of the three skip lists above, as well as one based on **Java** standard library *ConcurrentSkipListSet*, which is implemented based on *ConcurrentSkipListMap*. Several test benches with different number of operations and threads are applied for a complete comparison. Also, different operation types are considered.

2 Algorithms

2.1 Coarse-Grained Lock-based skip list

For Coarse-Grained implementation, we simply used a single lock to protect the entire list, which is, used **synchronized** around *add*, *remove* and *contains* functions. The implementation of these functions are the same as those of the non-concurrent skip list.

2.2 Fine-Grained Lock-based skip list

Our implementation of Fine-Grained skip list is based on the algorithm proposed by Maurice Herlihy, et al[3].We augment each node with a **marked** flag, which will make *remove* operations atomic, a **fullyLinked** flag, which is set to **true** after a node has been linked in at all it layers and setting of it is the linearization point of a successful *add* operation. Thus, a key exists in the set if and only if it's unmarked and fully linked.

1. **findNode(int key)**

   We first implement a helper method *findNode*, which searches exactly as in a sequential skip list, which will record the predecessors and successors of the expected node in each layer and the return the highest level at which node was found if it exists. Otherwise, it returns -1.

2. **add(int key)**

   In the *add* operation, we first run *findNode* to examine its existence, **marked** and **fullyLinked** flag. If the key indeed doesn't exist, then the thread tries to locks and validates all the predecessors obtained by *findNode*, if the thread succeed, the thread allocates a new node, links it between its predecessors and successors, and sets the **fullyLinked** flag of the new node. Finally, the thread releasing all its locks.

3. **remove(int key)**

   In the *remove* operation, we also run *findNode* to examine its existence and check whether it's Ok-to-Delete by checking it **marked** and **fullyLinked** flag and it was found at its top layer. If the node satisfies the requirement, the thread locks it and verifies it's **marked** again, if succeed, the thread marks the node, which is logically deletes it. The rest of the procedure just accomplished the physically deletion.

4. **contains(int key)**

   The *contains* operation just calls *findNode* and check its **marked** and **fullyLinked** flag.

2.3 Lock-free skip list

Our implementation for *Lock-Free Skip List* is presented by Herlihy and Shavits[4]. In the lock-free skip list, since we are not using locks, we are not able to manipulate references at all levels simultaneously, thus to maintain the property that one list of some level is the sublist of all levels below it is impossible now. Thus we treat the bottom-level list as our abstract set. Hear, each node is a markable reference. To remove a node, we have two phases: *logically remove* and *physically remove.*

1. **findNode(int key)**

   When searching for a node with some specific node, we physically remove those marked nodes. If encountering an unmarked node with the exact value, we have found such a node. If not, just return *false.*

2. **add(int key)**

   We first call *findNode* to determine whether there is already a node with *key* in the skiplist. If we find such an unmarked node, just return *false* for *add* operation. If not, then create a new node with a random *topLevel.* Then, use **CompareAndSet** to make a check and link it to the bottom-level (or restart the call if something changes). The next step is to link the node in at higher levels.

3. **remove(int key)**

   Also call *findNode* to determine there is an unmarked node with the specific key in the bottom-level list. If such a node is found, we logically remove the associate key from the set by marking it. Also, other checks are applied for different conditions, as the node is already removed by other threads, or *next* has been changed.

4. **contains(int key)**

   This method is wait-free. Other than calling *find* to get the result, we simply traverse the SkipList, descending level-by-level, from the **head** of the list. When encountering marked nodes, just jump over them.

3 Performance Result

3.1 Compiling Environment

The benchmarking environment has the following specifications:

- jdk 1.8.0_191

- Intel Core i7-8565U processor

3.2 Results

When comparing the performance of different implementations, there are several issues to be considered: number of threads, number of operation, what types of operations are applied. And since keys are randomly generated within some range, the range of numbers might also influence the performance.

There are three different operations for a skip list: **add**, **remove** and **contains**. Here, we choose three different compositions of operations:

- 0 % **contains**, 50 % **add**, 50 %**remove**
- 70 % **contains**, 20 % **add**, 10 %**remove**
- 90 % **contains**, 9 % **add**, 1 %**remove**

We also try three different number & range of operations:

- **10000** operations with range **200**
- **10000** operations with range **2000**
- **100000** operations with range **20000**

Thus, we have totally 9 combinations of operation types and numbers. For each combination, we apply the test for different numbers of threads: 2,4,8,16,32. Five separate performance tests are made and the average run time is considered as the run time for a specific implementation of skip list, under this set of conditions.

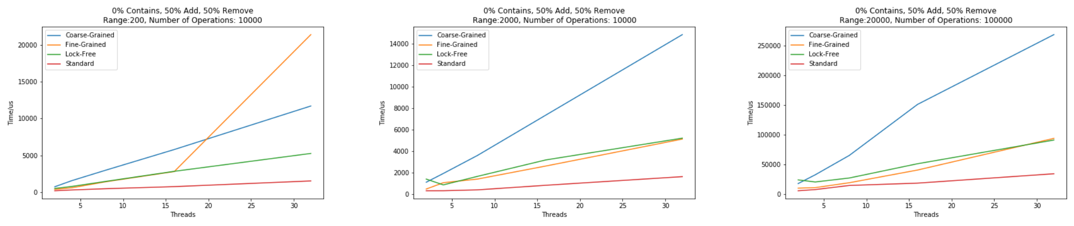

Figure 2: 0% **contains**, 50% **add**, 50% **remove**
Operations: 10000(range 200), 10000(range 2000), 100000(range 20000)

From Figure.2, we could clearly see conclude that the coarse-grained skip list always requires the longest time to complete the operations, and standard library **ConcurrentSkipListSet** always the shortest, which is same as expected. The fine-grained version and lock-free version have similar performance in most cases, especially for more operation numbers. Theoretically, the lock-free version skip list should have better performance, but since the existence of multiple corner cases, our lock-free implementation has a performance generally slight worse than fine-grained skip list. Generally, it requires longer time to complete the operations with more threads.

Another observation is that the run time increases a bit as the range of keys gets larger. This is reasonable since the skip list could have a larger size, so longer time is required to find a node with specific key and complete the **add**/**remove** operation.

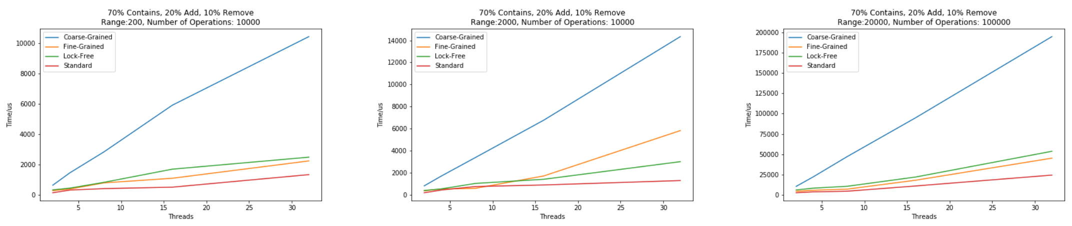

Figure 3: 70% **contains**, 20% **add**, 10% **remove**
Operations: 10000(range 200), 10000(range 2000), 100000(range 20000)

Similar conclusions could be drawn from Figure.3. The total run time is much shorter than the previous one since of different composition of operations. This time we have 70% operations as **contains**, which costs shorter time than **add** and **remove**.

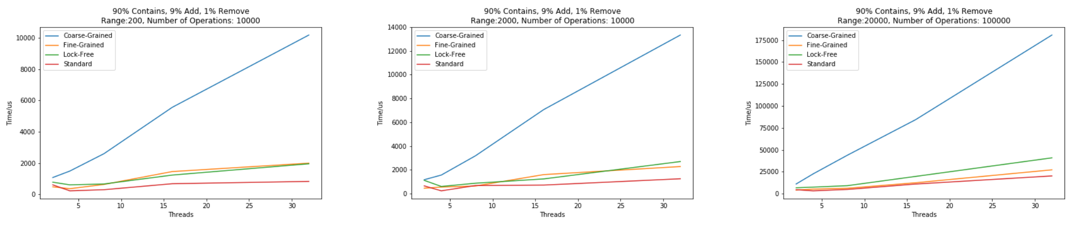

Figure 4: 90% **contains**, 9% **add**, 1% **remove**
Operations: 10000(range 200), 10000(range 2000), 100000(range 20000)

In Figure.4, we show the results of the last composition of operations: 90% **contains**, 9% **add** and 1% **remove**. All the results observed above still work for this set of graph. And even less total run time is required since of more **contains**.

4 Conclusion

From multiple tests with different parameters, we could conclude that our fine-grained and lock-free implementation have basically similar performance, either for less or more operations, or different compositions (portion of **contains**,**add** and **remove**) of operations. The fine-grained version is slightly better, but the stability is worse when number of operations is small and number of threads is large. (This is concluded from the weird rise in Figure.2 and 3) The laziness strategies in fine-grained implementation makes it better than coarse-grained version, which is obviously the slowest one. The lock-free version does not perform as well as expected; some corner cases might influence its efficiency. The **Java** standard library **ConcurrentSkipListSet** has the best performance, which is not affected a lot by number of threads.

Generally speaking, our implementations are working, and better than coarse-grained skip list, which is reasonable. Still some optimization could be applied to achieve better performance.

References

[1] Pugh, William. "Skip lists: a probabilistic alternative to balanced trees." Communications of the ACM 33.6 (1990): 668-677.

[2] Heller, Steve, et al. "A lazy concurrent list-based set algorithm." International Conference On Principles Of Distributed Systems. Springer, Berlin, Heidelberg, 2005.

[3] Herlihy, Maurice, et al. "A provably correct scalable concurrent skip list." Conference On Principles of Distributed Systems (OPODIS). 2006.

[4] Herlihy, Maurice, and Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2011.

5 Appendix Data

90% Contains, 9% Add, 1% Remove

Table 1: Range: 200 Number of Operations: 10000 (ns)

threads	Coarse	Fine	lock-free	standard
2	1.0723627E7	4786173.0	7753409.0	6062472.6
4	1.48620888E7	3599319.8	6047976.0	2228125.6
8	2.59644726E7	6303256.0	6602642.8	2946365.4
16	5.55967372E7	1.45398784E7	1.23008548E7	6773309.4
32	1.017870656E8	1.99189646E7	1.95202618E7	8258623.8

Table 2: Range: 2000 Number of Operations: 10000 (ns)

threads	Coarse	Fine	lock-free	standard
2	1.17348786E7	4607590.0	1.12077652E7	
4	1.56621662E7	5553556.2	6060210.2	6570976.8
8	3.18448664E7	6449865.2	8792522.8	6570976.8
16	7.0701872E7	1.60379414E7	1.24106568E7	6570976.8
32	1.33354855E8	2.28087834E7	2.70097548E7	6570976.8

Table 3: Range: 20000 Number of Operations: 100000 (ns)

threads	Coarse	Fine	lock-free	standard
2	1.109992638E8	4.09616556E7	6.76988578E7	4.54138976E7
4	2.265124566E8	5.28645278E7	7.52332786E7	3.29138976E7
8	4.383243406E8	6.21838554E7	9.15379124E7	4.86083464E7
16	8.430761746E8	1.248403338E8	1.971070964E8	1.10166184E8
32	1.8080728438E9	2.722134036E8	4.088156552E8	2.028340954E8

70% Contains, 20% Add, 10% Remove

Table 4: Range: 200 Number of Operations: 10000 (ns)

threads	Coarse	Fine	lock-free	standard
2	6487596.8	2942355.8	3435747.2	1535692.0
4	1.46438206E7	3996891.2	4578802.6	3303737.4
8	2.84601134E7	8039019.4	8391867.0	4265845.6
16	5.9163877E7	1.10764754E7	1.70568012E7	5211503.4
32	1.043608424E8	2.24937698E7	2.49685398E7	1.34715666E7

Table 5: Range: 2000 Number of Operations: 10000 (ns)

threads	Coarse	Fine	lock-free	standard
2	8284943.8	3791165.6	4062484.4	2020652.8
4	1.71511822E7	5420518.4	5530115.6	4493366.2
8	3.3954142E7	6036666.8	1.0467524E7	7672085.6
16	6.776075E7	1.72319918E7	1.42259956E7	9075562.0
32	1.434267614E8	5.83179458E7	3.0180352E7	1.3072145E7

Table 6: Range: 20000 Number of Operations: 100000 (ns)

threads	Coarse	Fine	lock-free	standard
2	1.070924356E8	4.6942701E7	6.36338024E7	2.89122774E7
4	2.23754647E8	5.92731656E7	8.51877744E7	3.974108E7
8	4.736807258E8	7.3419174E7	1.084534514E8	4.79290726E7
16	9.485326776E8	1.825032694E8	2.217384152E8	1.120291244E8
32	1.945290627E9	4.534854052E8	5.37888935E8	2.455593758E8

0% Contains, 50% Add, 50% Remove

Table 7: Range: 2000 Number of Operations: 10000 (ns)

threads	Coarse	Fine	lock-free	standard
2	7314405.2	3796306.2	5039397.0	1888951.6
4	1.56102464E7	5909694.8	7736651.2	2955618.4
8	2.9899471E7	1.41417928E7	1.46329226E7	4679660.4
16	5.7848304E7	2.79511974E7	2.8265594E7	7500596.4
32	1.169384654E8	2.13857623E8	5.25502334E7	1.52179186E7

Table 8: Range: 2000 Number of Operations: 10000 (ns)

threads	Coarse	Fine	lock-free	standard
2	1.11613972E7	4844981.2	1.40925466E7	3245546.2
4	1.93232756E7	1.0726197E7	8699170.0	3270118.4
8	3.60778118E7	1.41071454E7	1.66222178E7	4070504.0
16	7.35506696E7	2.62286972E7	3.18893834E7	8333778.8
32	1.484340818E8	5.12966558E7	5.21517374E7	1.63592266E7

Table 9: Range: 20000 Number of Operations: 100000 (ns)

threads	Coarse	Fine	lock-free	standard
2	1.735473794E8	9.58826142E7	2.353275978E8	5.22929998E7
4	3.20822161E8	1.048082774E8	2.001573012E8	7.21299206E7
8	6.476465472E8	1.873395116E8	2.671698932E8	1.415318468E8
16	1.5068224534E9	4.011959006E8	5.073421642E8	1.805063658E8
32	2.6829222254E9	9.34819299E8	9.066834522E8	3.387645786E8