pcp report

by Harshit Patel

Submission date: 28-Apr-2019 10:50PM (UTC+0530)
Submission ID: 1120608684
File name: PCP_report.pdf (181.73K)
Word count: 1474
Character count: 6975

CS5300 - 04

Concurrent SkipList

Lock Based and Lock Free Concurrent Skiplist

Harshit Patel

CS16BTECH11017

Shubham Kumar

ES16BTECH11028

INTRODUCTION

We have chosen option 1 and implemented concurrent data-structures : lock-based concurrent skiplist and lock free concurrent skiplist.

Then we have performed several test to demonstrate the performance of these 2 data-structures and finally the graphs are shown.

Operations performed on skiplist:

- Insertion
- Deletion
- Searching

This concurrent skiplist design uses LazyList algorithm to implement the LazySkipList class.

Each level of Skiplist structure is a LazyList i.e. the contains() method is wait-free, does not contains lock s.

Optimistic fine-grained locking is used by the add() and the remove() functions.

Each node has its own Reentrant lock and a marked field.

This marked field is used to demonstrate the node's presence in the abstract set.

It is also helpful to know whether the node is logically remove or not.

To implement ***Re-entrant Lock*** in c++, we have used ***recursive locks*** : a lockable object which allows the same thread to acquire multiple level of ownership over the same lock.

We keep in mind this property while implementation:

Skip list property : Lower-level list always contain higher-level lists.

Uses of lock :

- If a node is added or removed, to prevent changes to occur in its neighbourhood during that period.
- Until a node is inserted into all the levels of the skip list, access to it is prevented.

Working of functions :

For inserting a node in the skip list it must be inserted into several levels. To do this, ***add()*** function calls ***find()*** , which returns the predecessors and successors of the node which we will inserts in to our ***preds[]*** and ***succs[]*** .

Locking is done to ensure no changes are performed on the predecessors while the addition of node is performed.

To ensure the skip list property until all the references(to its predecessor and successors) are set up, a node is not considered to be added. This property is ensured by a flag, ***fullyLinked***.

A thread must spin-wait until all the references are set.

remove() method uses ***find()*** method to ensure if the node with a specific key to be deleted is already present in the skip list or not. If it is present, it checks if it is ***unmarked*** and ***fullyLinked*** i.e. whether the victim is ready to be deleted.

If the node is present, remove() deletes the node logically by setting its mark bit.

The physical deletion of the victim is performed by locking all of its predecessors at each level and the victim node. It then checks if the predecessors are unmarked and are referring to the victi. Victim node is then spliced out.

contains() is wait free and it invokes find() function. This is done to locate the node which contains the target key. If this is successfully done, it checks whether the node belongs to the set. This is done by checking if it is fully linked and unmarked.

A Lock Free concurrent skiplist

It depends on the LockFreeList algorithm. Its each level is a lockFree List.

Compare and Set instruction is used for list manipulations.

Skiplist property is violated in this implementation. To overcome this, we consider that if there is a node with a key whose next reference is unmarked in the bottom-level list then that particular key is in the set. The bottom level lists define this abstraction(can be considered as abstract set).

FullyLinked flag is not used here.

Working of functions :

To implement add() and remove() function, we consider each level of the skip-list as a LockFreeList. To add() CAS is used to insert a node at a given level and to remove() a node we mark the next pointer of the node.

The find() method removes marked nodes are we encounter them.

The add() method is similar to the above implementation, only we (logically) add the new node(to be added) to the abstract set that we defined above by connecting it to the lowest or bottom level list and if succeeded then it is added to the higher levels.

In the remove() method if the node is present and if we encounter an unmarked node, it is marked starting from the topLevel. We logically remove all next pointers up to, but not including the lowest level pointer from the corresponding level list. This is done by marking them. Bottom level's next reference is marked once all the level except the bottom one has been marked. The item is removed from the abstract set if the marking is successful.

If a node is logically added to a list it is ready to be removed then.

Testing and Comparison graphs:

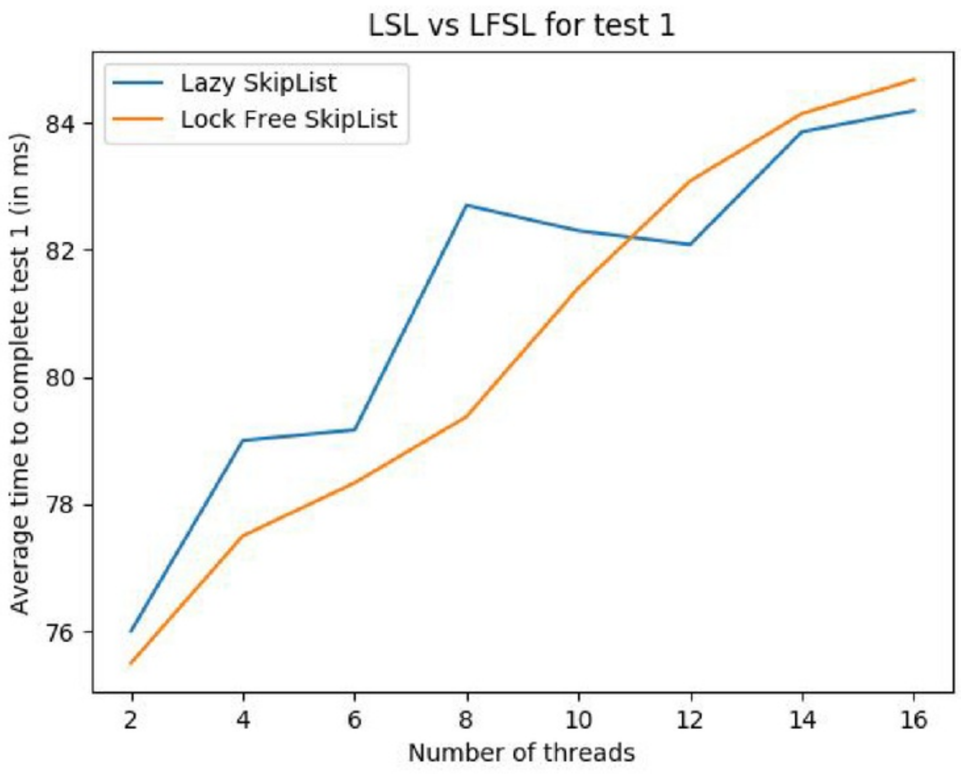
To test out the performance of the lock based and lock free concurrent skiplists we wrote three tests. The performance metric used is average time taken by a thread to complete the test. These tests are written in the file Application.cpp which invokes LazySkipList.hpp and LockFreeSkipList.hpp. While executing this file, three command line arguments are required. First being the number of threads, second being the number of operations to be performed (more about this is given below) and last being the usual lambda for exponentially distributed sleep time after each operation. In our testing we wrote a shell script that executes this application eight times with value of N ranging from 2 to 16 in increments of two, k being constant at

30 and lambda being constant at 10. To plot the graphs we wrote a plotter.py code.

For each test we first create N threads for Lock free skip list who have a shared object of the lock free class. After these threads finish their work and join and their average wait time is calculated, we create N threads again for Lock based skip list who have a shared object for lock based class between them. After finishing their work and joining we measure the average wait time.

Test 1:

In this test, first every thread adds k numbers into the skiplist. These numbers are generated at random and their modulo 100 value is used. The inserted number is also pushed back into a vector. After that each thread removes k/3 elements from the skiplist. Here we use the vector made during add operation by accessing one of its indices randomly and removing the element present at that index from the skiplist. This results in more number of successes while deleting. Finally every thread executes the contains operation on all of its inserted values. When these operations are executed there success or failure along with the time is written to log files logs_test1_lf.txt for lock free and logs_test1_lazy.txt for lock based. Average times are written to times1.txt. Graph created using plotter.py.

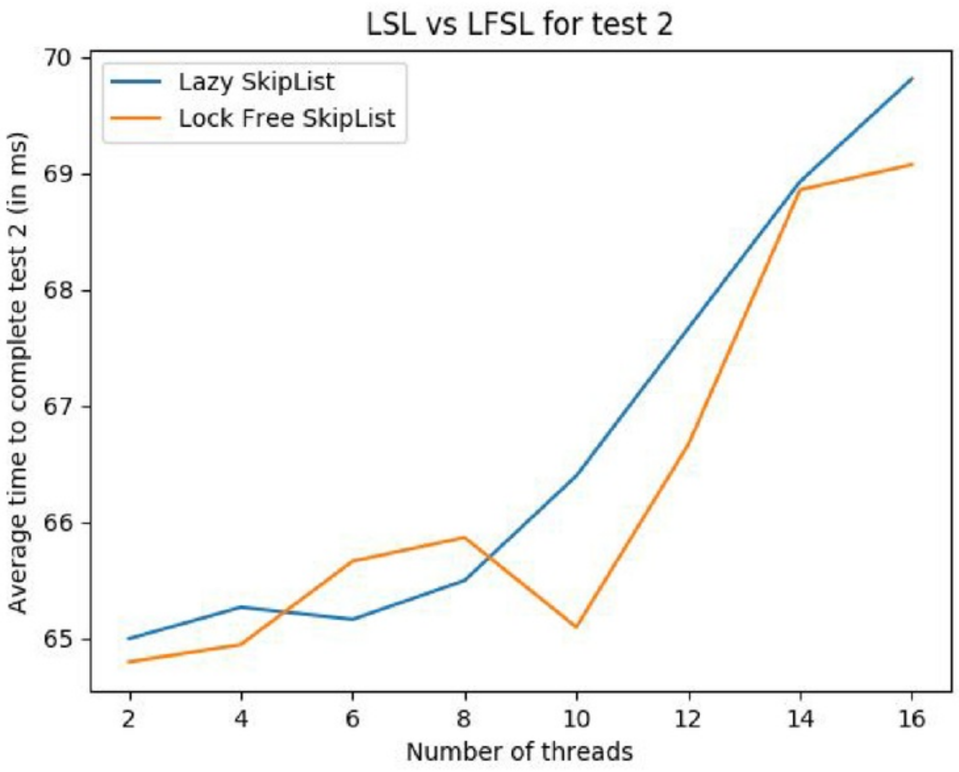


It can be observed from this graph that both these algorithms perform nearly equally when all the operations are executed by each thread with lock free performing well at lower values of N while lazy performing better at higher values of N. However, these results might change when run on a different machine at a different time based on the state of the machine.

Test 2:

In this test, every thread inserts 2*k elements into the skiplist if its thread id is less than N/2 and removes 2*k elements from the skiplist if its thread id is more than N/2. These elements are generated at random and the values used are taken modulo 15. This test leads to more failed removes at the start and more remove successes at the end. While for add success is kind of constant throughout the test. Log files: logs_test2_lf.txt for lock free and logs_test2_lazy.txt for lock based.

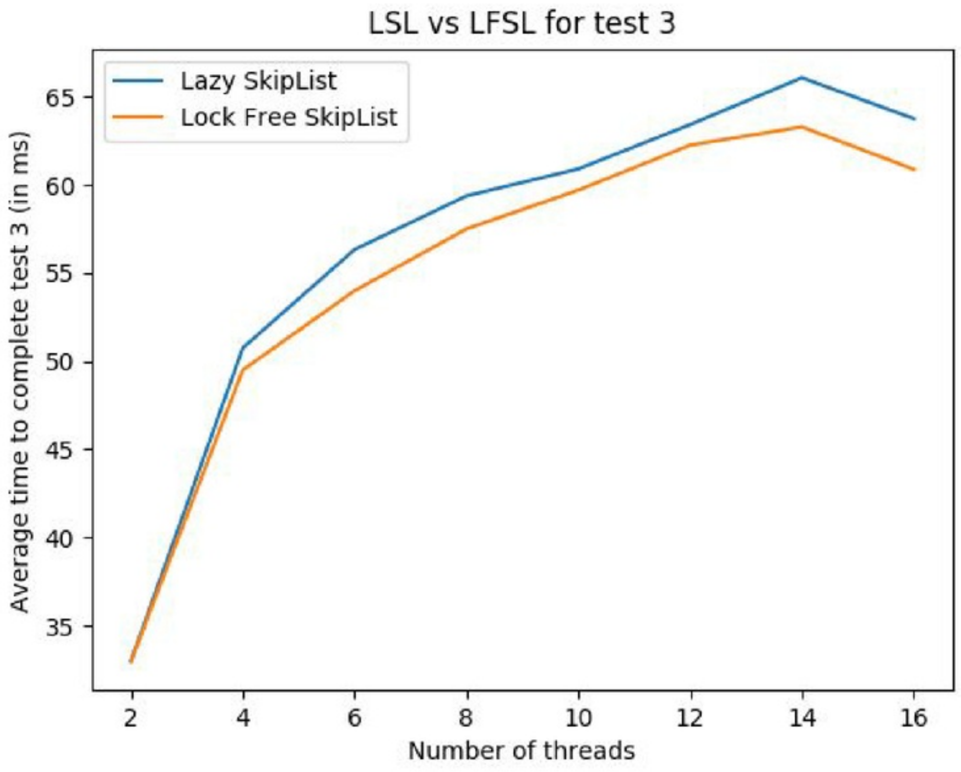
Average times file: times2.txt. Graph using plotter.py



As it can be observed from the graph that the lock free skiplist actually performs better for this test. However, lazy skiplist does not perform very poorly and its time is a little bit more than the lock free skiplist.

Test 3:

In this test, every thread inserts k elements into the skiplist if its thread id is less than N/2 and searches for 3*k elements from the skiplist if its thread id is more than N/2. These values are generated randomly modulo 30. This test represents the congestion of contains operation.
Log files: logs_test3_lf.txt for lock free and logs_test3_lazy.txt for lock based.
Average times file: times3.txt. Graph using plotter.py



For this test, the lock free skiplist performs a bit better than the lazy skiplist for all the values of N. This means that for an application that is search heavy, lock free skiplist should definitely be preferred over the lazy skiplist.

pcp report

ORIGINALITY REPORT

4%	0%	0%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to CSU, San Francisco State University Student Paper	4%

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		