A CPP implementation of “A Fast Lock-Free Internal Binary Search Tree”

A Project for CSCI-B524 - Jayendra Khandare (jkhandar@umail.iu.edu)

Abstract

“A Fast Lock-Free Internal Binary Search Tree” was published in ACM in 2015 and was authored by A. Ramachandran and N. Mittal. The paper extensively talks about the algorithm and their experimental evaluation. In the paper, they have compared their implementation against that of some other and how they found out that this outperforms other by 35%(at 32 threads).
[ISBN : 978–1–4503–2928-6]

Introduction

The explanation of the algorithm given in the paper is well structured and easy enough to understand. It clearly says that,”this algorithm combines ideas from two other non-locking algorithms, namely those by Howley and Jones, and Natarajan and Mittal”, so when they compare this implementation with the one mentioned above it is bound to be better. As they have already compared the performance of this one with others, I decided to compare this non-locking algorithm with some locking algorithm. I also tried to get access to some other non-locking algorithms and make comparative analysis but was not successful in doing so.
Basically, the paper talks about internal binary search tree. A binary search tree has to follow these properties :

- the left subtree of a node contains only nodes with keys less than node’s key
- the right subtree of a node contains only nodes with keys greater than node’s key
- both the subtrees of a node are also binary search trees

Furthermore, a internal binary search tree in all internal and leaf nodes.

Overview of the Algorithm

Since a binary search tree[BST] has to allow three operations viz. Search, Insert, and Delete, this algorithm also talks extensively about them. Furthermore, it uses a function called seek in all of these three functions to create access path and keep track of anchor node.
The Insert function invokes the seek function and tries to insert a node using compare-and-swap[CAS] instructions as a child node of the last node in the access path. So, same as NM-BST(Natarajan and Mittal), this method uses only a single CAS operation.
The Delete function(named remove in this implementation) invokes seek and removes the node(if found) using CAS instructions, but that procedure depends upon how many children the target node has. As this implementation doesn’t consider garbage collection or memory deallocation, the procedure varies if the target node has no child in which case, it just simple

dereferences the target node at is parent. Furthermore, there are two types of deletions viz. simple(if node has a single child) and complex(if node has two children). Based on that, this method needs 4 CAS for simple delete and 7 CAS for complex deletes.
The Search function simply finds the node using seek and using the access path so created.
Also, the helping function is important as well, since it does all the cleaning tasks. When a CAS fails, depending upon the reason for failure, helping is performed along the last edge or the second to last edge.
The algorithm talks about the flow of execution in details. Also, the algorithm for major methods and some other auxiliary methods are given in the paper.

Algorithm	Number of objects allocated		Number of CAS performed	
	Insert	Delete	Insert	Delete
HJ-BST	2	Simple: 1 Complex: 1	3	Simple: 4 Complex: 9
NM-BST	2	0	1	3
THIS-BST	1	Simple: 0 Complex: 1	1	Simple: 4 Complex: 7

Implementation and Problems followed

As the algorithms were provided for all the major and auxiliary methods, it should have been an easy task, but it was not. The notations used in the algorithm for some tasks are beyond recognition. For example,
line 50:
<*,*,d,p,*>:=anchorRecord.node→child[RIGHT]

line 73:
<*,nKey> := node→mKey

line 212:
oldValue:=<0n,1i,0d,0p,node>

It was difficult to understand initially, hence I marked all the unknown notations and tried to generalize them. I found out that some of these were recurring like line 73, hence they can be put into some function and then executed. So, I created some functions of my own and declared them in “function_declarations.h”.
Also, the data structures have to be passed to threads with functions and each thread should run multiple times, hence I had to introduce a data structure “struct args” which I have declared in “ds_liabraries.h”.
The author talks about extremities by means of sentinel keys and nodes by means of R,S, and T. Hence I had to declare them and put the maximum key value possible (0x7FFFFFFF) in T node.

For CAS operations, I used the standard function compare_exchange_strong which is defined in “atomic.h”.
To be honest, the algorithm is easy to understand if you read the description of the algorithm rather than going to pseudo code directly. The algorithm could have been formulated better to explain important concepts flag settings, etc.
In section 4.2, author talks about the platform they used for experimental evaluation. It is mentioned that, they used Intel’s TBB Malloc as the dynamic memory allocator since it provides superior performance to C/C++ environment. I used the standard std::atomic for node structure. This might slow down the execution to certain level. Also, it is given that they used GNU Scientific Library for random number generation whereas I used standard random library. So, I can’t say that my implementation is exactly the same as the author used for evaluation.
Furthermore, author compared the results of their implementation with that of HJ-BST [Howley – Jones] and NM-BST [Natarajan - Mittal] which are also non-locking algorithms. I couldn’t find the code of these two algorithms initially. Later when I found them, it became a rather difficult task to edit them to fit into this implementation. So, finally I decided not to go on with that. Hence, I compared my implementation with basic locking internal binary search tree where I used the standard std::mutex for locking mechanism.

Benchmark

For this project, I used the code related to multi-threaded structure provided by C. Wailes for the assignments and build on that. Thanks for that.

So, the program needs following arguments:
[NUM_OF_THREADS] – number of threads you want the program to run with
[NUM_OF_OPS_PER_THREAD] – number of operations that you need a thread to perform
[INSERT_PERCENT] – Percentage of Insert operations you want the thread to perform
[SEARCH_PERCENT] – Percentage of Search operations you want the thread to perform
[DELETE_PERCENT] – Percentage of Delete operations you want the thread to perform

HOW TO COMPILE : g++ -pthread main.cpp
HOW TO EXECUTE : ./a.out [NUM_OF_THREADS] [NUM_OF_OPS_PER_THREAD] [INSERT_PERCENT] [SEARCH_PERCENT] [DELETE_PERCENT]

For benchmarking first, we first let all Insert operations happen, so that we have a BST to work with. Then We perform Search operations, then finally Delete Operations. I specifically did this to better understand which among these two operations(Search and Delete) takes more time and what takes so long.

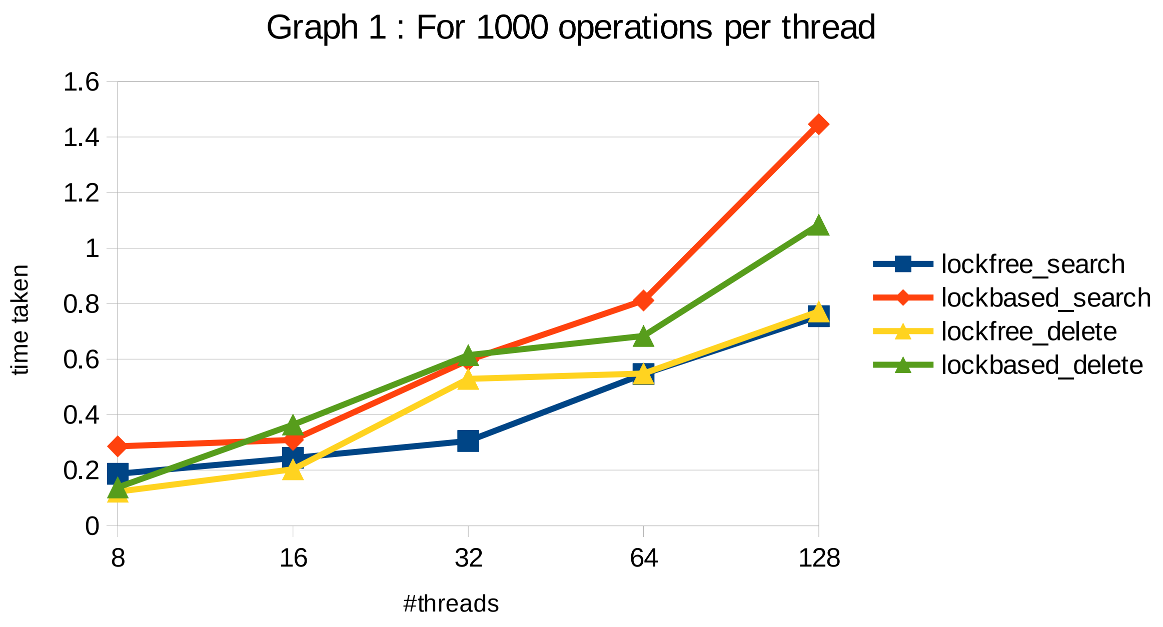
Performance Analysis

As you can see from the the graphs, lock-free methods are faster than lock-based methods. Specifically, the time difference between lock-free search and lock-based search is considerably

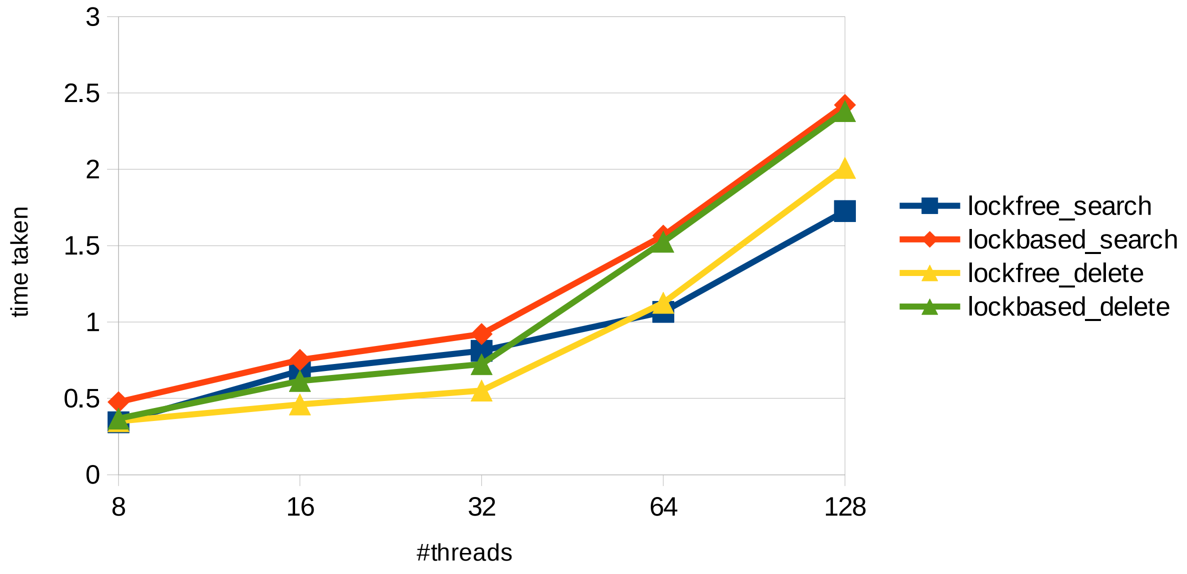
high as compared to the time difference between the second pair. It is fair to say that, this is because while searching for a node the lock-based method blocks the whole tree, hence decreasing the throughput.
Furthermore, we can also say that after a certain threshold, lock-free delete picks up the pace and takes up less time. I think at higher number of operations per thread, it will reach at par with lock-free searching. I couldn’t understand the reason behind it.

Conclusions

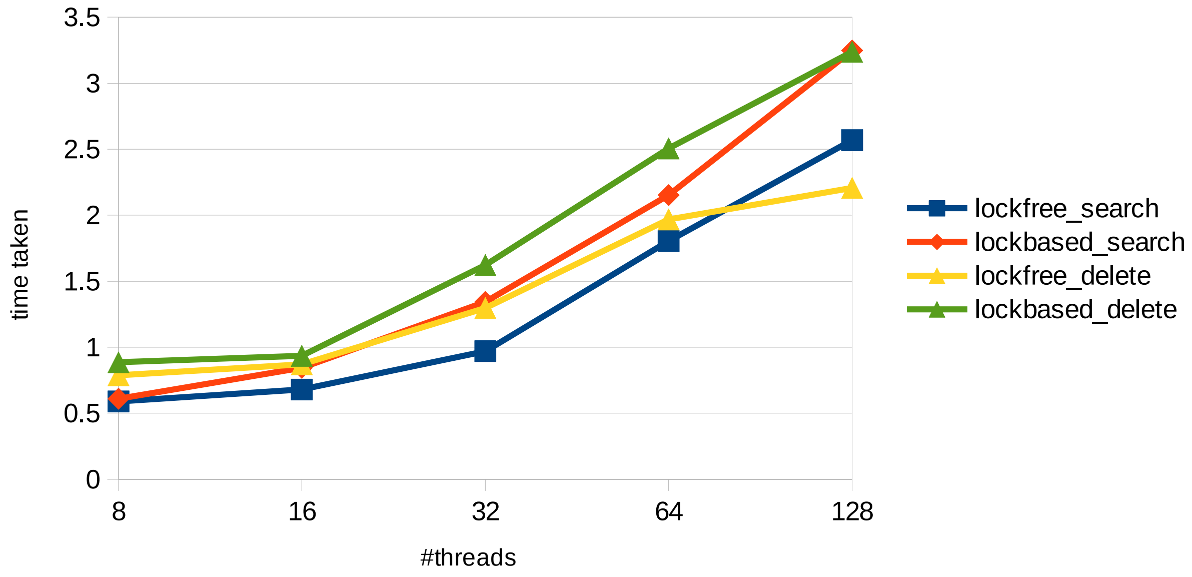
- The author could have simplified the pseudo code by explaining or mentioning the notations he used, so that the reader doesn’t have to go through the process of deciphering the meanings.
- The lock-free methods are always better than lock-based methods if number of operations is huge.
- The claims made by the author regarding comparison of this method with HJ-BST and NM-BST could be true, because this method tries to acquire the advantages of both these while trying to avoid all the drawbacks.



Graph 2 : For 5000 operations per thread



Graph 3 : For 10000 operations per thread


time taken
#threads
lockfree_search
lockbased_search
lockfree_delete
lockbased_delete