Comparing Concurrent AVL Trees

Nick Carlson, Paul Iwatake

Abstract

Summary and discussion of the implementation of lock and lockfree variants of AVL trees in two different programming languages, and analysis comparing its performance with related tree implementations.

Keywords: AVL Tree, Concurrency

1. Introduction

For our project, we set out to compare implementations of balanced tree data structures, or AVL trees. Initially, we wanted to compare an implementation created in an imperative language, versus a functional language. We planned to use Haskell, C++, Google Go, and Scala, but had to scale down to using two imperative languages, C++ and Go, instead, due to time constraints.

We had wanted to use functional languages due to their ease of paralelization, as they can be easier to prove correctness with, automatically put in parallel with some tools, and other characteristics that cater well to concurrency.

Various research has been conducted involving variants of non-blocking binary search trees [1], [2], [3], but we could not find many concurrent AVL trees implementations using locks freely available, bar one [3]. Obviously more research has focused on non-blocking trees in the interest of speed and efficiency. We also tried to incorporate some ideas from concurrent Red-Black Trees [4].

We did not seek to reinvent the wheel, so to speak, but to gain experience in concurrency using programming languages we were not entirely familiar with. As such, we started out with a coarse grain lock approach, then moved on to a fine grained lock, before attempting to apply non-locking implementations.

2. Related Work

Similar tree structures include the Skip List[3], non-blocking trees using Compare-and-Swap (CAS) constructs[2], and a concurrent relaxed balance AVL tree[1].

2.1. Skip List

The skip list is a probabilistic data structure essentially consisting of a hierarchal linked list where each node has many links, linking few nodes at the top of the hierarchy so that those nodes have "shortcut" links, and all the nodes at the bottom of the hierarchy. Nodes are assigned a level randomly at insertion time[3].

This is a popular (An implementation exists in the Java library, so that says something) quickly-searched data structure because of its simplicity to implement compared to a tree, where concurrent rebalancing can lead to significant bottlenecks[3].

2.2. Non-blocking Tree

One implementation of a tree data structure not using locks was found using CAS primitives. It uses multiword versions of the primitives load-link (LL), store-conditional (SC), and validate (VL), just multiword versions, denoted as LLX, SCX, and VLX respectively instead. These operate on "data-records", data structures that consist of a fixed number of mutable and immutable fields.

LLX is used to take a snapshot of a given data record, which may fail if overlapping with an SCX operation. SCX, as its name might imply, tries to atomically change a value of a mutable member of a data record, then make a data record "finalized" so that further LLX returns that status instead of a snapshot. Processes involved SCX and VLX must do an LLX that succeeds and produces a valid snapshot before they can proceed. VLX can then be used by the same process to see that the data record(s) have not changed since the last LLX call by the same process. In this way it obtains a snapshot of data-records, since it may also operate on a set.

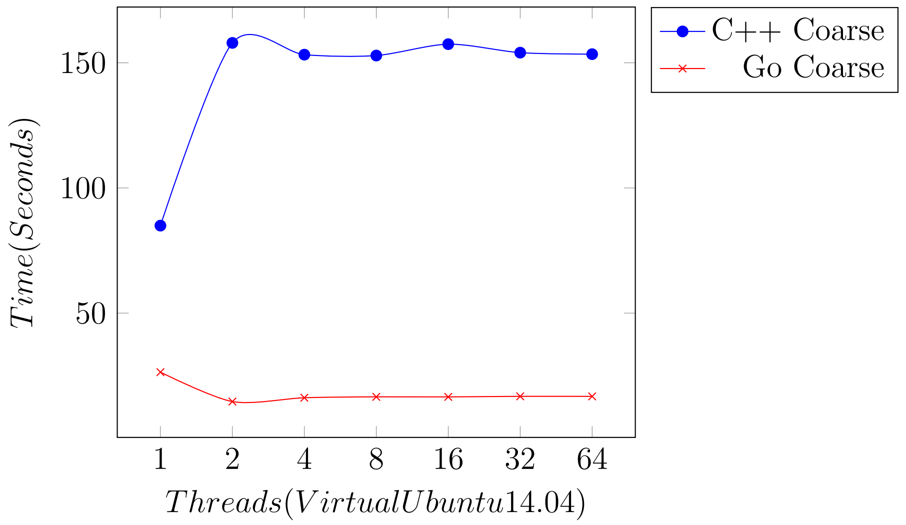
Using these primitives, it is possible to implement a chromatic tree that is provably correct, using a try-fail-retry strategy, that marks violations in the tree for cleanup later[2].

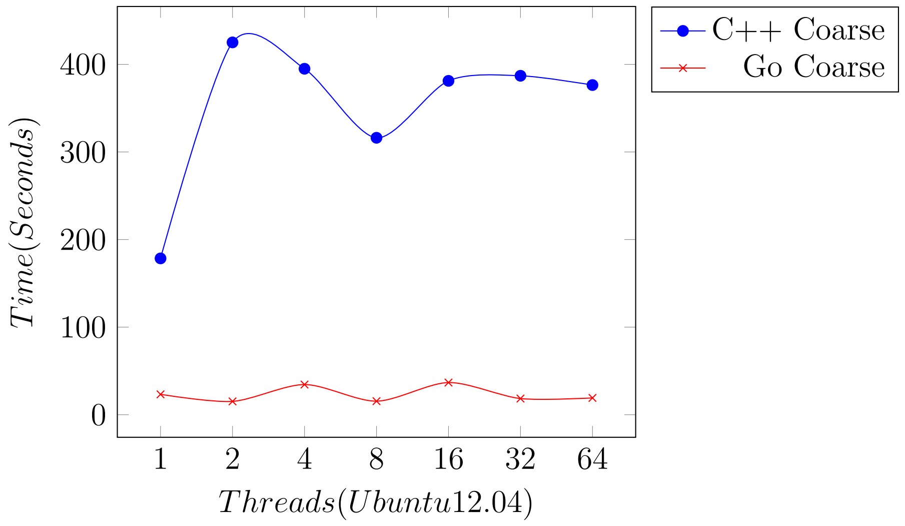
2.3. Relaxed Balanced AVL Tree

Yet another closely related structure is the relaxed balanced AVL tree. It is an optimistic structure, implemented in such a way that reads and writes have specific atomic regions that can be known ahead of time. To do this, it utilizes hand-over-hand locking, mutations that rebalance on their own, and deletions that sometimes leave behind a routing node[1]. A variant of this uses a linearizeable 'clone' operation to help enforce consistency.

An idea this uses that is worth explaining is that of the external tree. This is in contrast with an internal tree, which is the usual AVL tree. In an external tree, all actual values in the tree are stored on leaves. Any non-leaf (internal) node is a 'routing' node, to aid in traversal only. This introduces some overhead with the extra nodes and the traversal of them, but it allows for much less complicated deletes, which can incur serious rebalancing issues. The relaxed balanced AVL tree uses a partially external tree structure[1].

3. Performance





Testing was done on a virtual machine Ubuntu 64-bit 14.04 Linux system (2-Core 3.5 GHz processor) in order to utilize pthreads, and on a regular Ubuntu 32-bit 12.04 Linux system (2-Core 2.53 GHz processor). Testing procedure was to start with a tree of size 1, and to undertake an average workload doing 100 million operations, inserting possible values from 0 to 1 billion, with average workload defined as 90 % contains, 9 % insert, and 1 % delete. Tests were conducted 5 or more times for each number of threads, then averaged. Performance was compared by calculating runtime in seconds (So fewer seconds means better performance).

12.04 Ubuntu testing may have a minor descrepancy in the Go data, since the tests were done discontinuously, so there is apparently a spike (notable at 4 and 16 threads) in runtime due to initialization overhead for the program. But, this is still not enough to explain away the huge difference in performance between Go and C++. At times, Go seems to outperform C++ by more than a factor of ten. This is a huge difference. Possible reasons for this are that C++ pthreads are starting to show their age, or that Go's goroutines act significantly differently than regular threads. It is also quite possible our implementation for Go was flawed in some way, but this is not clear.

We found two projects kind enough to make their code freely available on GitHub, so it would make comparison testing quite simple, given additional time.

4. Our Approach

Most of the ideas for the approach come from concurrent red-black trees[4]. It uses a variable to keep track of the version of the nodes. If the version changes while the node is trying to traverse, that means that the nodes' pointers have changed and the operation would need to start again from the top. The version of the node updates whenever there is an add performed or a rotation performed.

So for contains operations, it would never have to obtain a lock, merely store a local copy of the node at the beginning of its call and check the version of the node as it tries to advance. Insert and Delete will both still need to lock to function correctly. Now to handle the rotate function, every node maintained a parent pointer that it would use to climb-up the tree. As the nodes rotate, they lock, their versions update, and a Boolean is set to marked. This Boolean allows inserting and deleting threads to know that there is a rotation going on and to return to the top of the tree and start again.

4.1. Difficulties In Implementation

Other than the abandoning the functional language implementations due to time, other problems were most of the functionality of C++ that we had researched and thought would be available to us, were not. The C++ 2011 libraries such as thread and chrono were not available and alternatives such as boost did not work either. Furthermore, the research we based our tree off of required the use of Compare and Swaps, something that could not be done in C++ without making it ourselves or getting the newer libraries to work.

5. Conclusion and Future Work

To summarize, comparing concurrent AVL trees in different languages proved an interesting task. In the future, it would be interesting to go back and actually finish the original comparison we sought, which was comparing functional language and imperative language AVL tree concurrency.

References

[1] N.G. Bronson, J. Casper, H. Chafi, and K. Olukotun, A Practical Concurrent Binary Search Tree. *PPoPP '10: Proceedings of the 15th ACM*

SIGPLAN Symposium on Principals and Practice of Parallel Programming, 2010.

[2] T. Brown, F. Ellen, and E. Ruppert, A General Technique for Non-blocking Trees. *PPoPP '14: Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014.

[3] B. Wicht, Binary Trees Implementations Comparison for Multicore Programming. *Switzerland HES-SO University of applied science*, 2012.

[4] J.H. Kim, H. Cameron, and P. Graham, Lock-Free Red-Black Trees Using CAS. *Concurrency and Computation: Practice and Experience*, 2006.