Aryan Soman
Jerry Xu
15418 Project Proposal
Lock-free AVL Tree
URL: `https://github.com/aryansoman/lock-free-AVL-tree.git`

Summary

An AVL tree is a balanced binary search tree. We plan to implement lock-free and lock-full AVL trees and compare their performance on real-world use cases. We also hope to implement a wait-free variant of an AVL tree or other types of lock-free binary search tree and make the same comparisons.

Background

Binary search trees (BSTs) are useful for storing and accessing ordered data in $O(\log n)$ time, and as such are common implementations of ordered sets and maps. These implementations are almost exclusively *balanced*–that is, upon a modification, additional operations are done to ensure that branches of the tree remain $O(\log n)$ depth throughout any sequence of operations.

An AVL tree is a type of balanced BST that maintains balance through the following invariant: After each operation, for every node, the height of its two children may differ by at most 1, where the height of a node n is the maximum length of a path from n to a leaf node in the subtree rooted at n. This invariant is maintained through operations called rotations that are done after the modification operation that created a violation of the invariant. A key characteristic of rotations is that they maintain the ordering property of the BST, ensuring that search is still possible in $O(\log n)$.

We may encounter many cases where multiple processes want to access a single data store implemented as a balanced BST, which is why we are interested in making a thread-safe implementation. The most straightforward way is to close off critical sections with locks, but lock-free algorithms are also possible. In fact, they are usually more difficult to implement than their lock-full counterparts, but can provide better parallel throughput. We are interested in whether we can observe this effect in AVL trees.

The Challenge

The main challenge of this project will be to create a lock free data structure for AVL trees. We can start by implementing with fine grain locks while avoiding common pitfalls like deadlock scenarios. In our lock-free implementation, we will need to ensure correctness is maintained through insertion and deletion operations. Finally, our lock-free implementation needs to have reasonable performance and ideally would offer significant speedup in comparison to the lock-full AVL tree. Although we

won't be coming up with the algorithms ourselves, parsing the papers on the lock-free trees is still also a challenge, since in class we only talked about lock-free queues which are much simpler.

Resources

We will start coding from scratch. However, we will implement the algorithms described in one or more of the following papers:

- "A General Technique for Non-Blocking Trees" (Brown et al 2014)
- "Lock-free Self-adjusting Binary Search Tree" (Sueki 2017)
- "A Fast Lock-free Internal Binary Search Tree" (Ramachandran and Mittal, 2015)
- "A Relaxed Balanced Lock-Free Binary Search Tree" (Singh et al, 2021)
- "A Methodology for Implementing Highly Concurrent Data Objects" (Herlihy 1993)
- "Concurrent Wait-Free Red Black Trees" (Natarajan et al 2013)
- "Lock-Free Concurrent Tree Structures for Multiprocessor Systems" (Tsay and Li, 1994)

Since the project focuses on a low-level data structure, we won't require any computing resources beyond the GHC machines, which should be more than sufficient.

Goals and Deliverables

Baseline:

- Implementation of a thread-safe lock-full AVL tree with insert, delete, and search operations. Operations are pretty easy to implement, and slapping a lock on top of critical sections should take much more time.
- Implementation of a thread-safe lock-free AVL tree with insert, delete, and search operations. Lock-free variant should be more difficult than lock-full, but there has been a lot of research already on lock-free balanced BSTs that is hopefully accessible with a bit of focused reading.
- Unit tests for both variants supporting correctness.
- A report detailing the implementations, arguments for their correctness, and a operation throughput comparison for a variety of thread counts and workloads.

- A poster including explanation of our project, diagrams showing tree operations as a quick refresher, as well as graphs of our throughput comparison on the various workload/thread count combinations. A data structure project is probably not conducive to a demo; all the important metrics should just be in our graphs. We are hoping to learn whether lock-free data structures really provide performance improvement over lock-full counterparts, and support/refute this in our throughput analysis.

Stretch:

- Implementation of bulk operations (split, join) for lock-full and lock-free AVL trees. These operations are still quite simple sequentially, so the lock-full algorithm should be doable, but we found nothing about them in the lock-free tree literature, so they will be much more difficult to implement. Also, we're not sure that they make sense for our use case.
- Lock-free and lock-full implementation of other types of balanced BSTs (red-black, splay), with performance comparisons to all other implemented trees. The tree algorithms are about the same as the AVL tree, except in the case of splay it would be better to do a lazy-splay tree to make it more amenable to parallel execution.
- Wait-free AVL tree, which provides per-thread progress guarantee that lock-free does not. Probably much more difficult to implement that the other two goals, but still possible by extending the lock-free structure via a method like the one shown in "A practical wait-free simulation for lock-free data structures" (Timnat et al, 2014).
- Any stretch goals implemented will also be included in the report and poster.

Platform Choice

We will code in C++ because it enables low-level manipulation of pointers and memory and relatedly is quite fast, making it a great language to implement common data structures like sets and maps. The implementation should be pretty platform-independent as long as there is a CAS operation, which is ubiquitous. We will not require computational resources beyond our own machines and the GHC machines.

Schedule

Nov 9 - 12
Additional research into AVL trees.
Begin implementation for lock-full AVL tree.

Nov 13 - 19
Finish implementing search, insertion, and deletion for lock-full AVL tree + unit tests.

Nov 20 - 30
Finish implementing search, insertion, and deletion for lock-free AVL tree + unit tests.

Dec 1 - Dec 3
Optimize lock-free and lock-full trees. Complete throughput comparison. Write results in report.

Dec 4 - 13
Stretch goal or slack days in case one of the implementations takes more time, accounting for sickness, etc.

Dec 14 - 17
Finish final report and create poster presentation.

Dec 18
Presentation day.