Implementation and Performance Analysis of Concurrent AVL Trees

SUBMITTED BY,

Soumya Kumar (906211337)

Sharvari Chougule (906208386)

ABSTRACT

An implementation of concurrent AVL trees has been proposed that is fast, scalable and delivers high throughput. Two lock based AVL tree implementations have been considered. These include coarse grained locking and fine grained locking. The performance of Coarse grained and Fine grained AVL tree has been compared with lock based Red Black Trees for various thread counts and different read or write ratios. A proper analysis has been carried out for checking the correctness property of the concurrent AVL Tree algorithm. The verification of Amdahl's Law has also been carried out.

CHAPTER 1 INTRODUCTION

1.1 INTRODUCTION

In today's world the use of multi-cores has increased on a large scale. This makes it imperative for us to develop efficient and fast algorithms on data structures that provide us with a scalable multi-threaded access. One such widely used data structure is AVL trees. AVL trees are basically used for search intensive operation and where there are few insertions and deletions. An AVL tree is named for its inventors Adel'son-Vel'skii and Landiis. It is a balanced binary search tree. It's called as balanced because the heights of the left and right sub trees of any node in a balanced binary tree differ no more than by 1. An AVL tree can be searched as efficiently as a minimum height binary search tree. The AVL algorithm is a compromise. It maintains the binary tree whose height is close to minimum, but it is able to do so with far less work than would be necessary to keep the height of the tree exactly equal to minimum. The basic strategy of the AVL algorithm is to monitor the shape of the binary search tree. After every alteration to the binary tree you have to secure back the balance of the tree. In this project we implement a concurrent version of the AVL trees which will function in a multithreaded environment. The goal is to allow concurrency between number of readers and writer doing operations such as insertion, search and delete. We achieve this by using locking mechanisms which will prevent erroneous transactions in the tree. After implementation we compare the performance of coarse grained and fine grained AVL tree against lock based Red Black trees, measure throughput for various thread counts, different read or write ratios and contention scenarios. A Red Black tree is also a self balancing binary search tree where every node follows a set of rules.

1.2 AIM

- To develop a concurrent AVL tree with fine grained and coarse grained locking.
- To compare the performance against each other and verify Amdahl's Law.
- To conduct performance analysis with Red Black Trees.

1.3 BACKGROUND

In the past various concurrency algorithms have been implemented. Concurrent trees have been implemented using optimistic concurrency control as mentioned in the paper "A Practical Concurrent Binary Search Tree". Different approaches (mentioned in References section) have been used to get a more relaxed balance for concurrent AVL trees in order to reduce the high contention. The need for more efficient concurrent algorithms has increased with the boost in usage of multiprocessors.

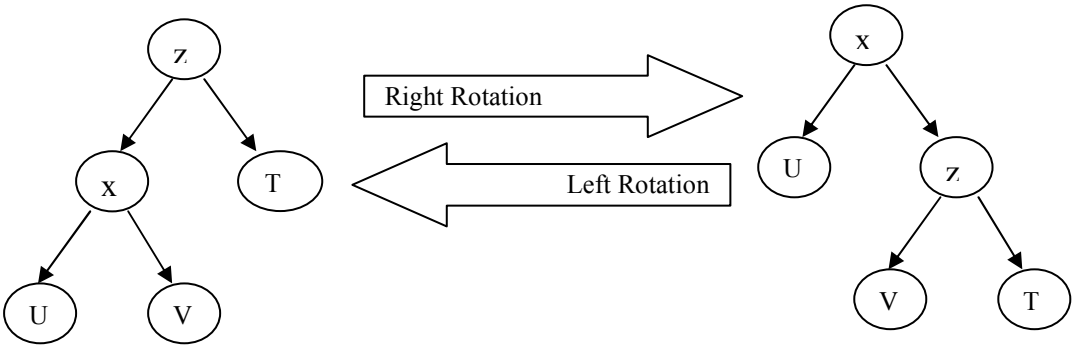
CHAPTER 2 ALGORITHM

2.1 OUR SOLUTION

- The AVL Trees are self balancing Binary search trees.
- In order to implement AVL Trees the same logic of insert/delete is used as in case of a BST. But after every insertion and deletion, rotations are performed on that tree so as to restore balance [Refer Fig 2.a]. The balance condition in this case is that the difference between the height of the left sub tree and the right sub tree should not be greater or less than 1. This is a basic AVL Tree algorithm.

Figure 2.a: ROTATIONS IN AVL TREE:

U, V and T are sub trees of the tree rooted with z (on the left side) or x (on the right side):
Order of the keys: keys(U) < key(x) < keys(V) < key(z) < keys(T)



- We implement **search/contains** function, **insert** function and **delete/remove** function.
- Now the concept of coarse grained and fine grained locking are introduced.
- A re-entrant lock is used for locking purposes as it provides synchronization with greater flexibility. It gives lock to current thread and blocks all other threads.
- In a coarse grained lock, a single lock is acquired by the threads in order to mutate the AVL tree. The thread in possession of the lock is allowed to perform operations such as insert, delete, search etc.
- In fine grained locking, the data structure is split into pieces and each piece has its own lock. Multiple threads can acquire locks on different nodes in the tree depending on a set of conditions. This tree uses hand over hand locking, each node contains a reentrant lock that is acquired before any accesses or modifications are performed.
- In fine grained locking, a Relaxed Balance AVL Tree approach is implemented. The idea with this approach is to uncouple the rebalancing in the tree in order to achieve higher throughput and speedup. A set of insert/delete operations are performed and late the balance of the tree s restored. This approach is against the rigid balanced trees in which after every insert/update, the rebalancing is performed. This improves performance of algorithm during high contention scenarios.

CHAPTER 3 PROPERTIES OF ALGORITHM

3.1 COARSE GRAINED LOCKING AVL TREE

3.1.1 LINEARISABILITY

The algorithm is linearisable. The linearization point for insert(), delete() and search() is tree tree.root.getLock().

3.1.2 MUTUAL EXCLUSION

The algorithm guarantees mutual exclusion. This property says that only one thread will have the access to the critical section at a time. No two threads can enter the critical section together. Consider the implementation in the insert class run method:

```
public void run() {
   synchronized(tree){
   for (int i = 0; i < PER_THREAD; i++){
      int RandInt = ThreadLocalRandom.current().nextInt(1, n);
      if(tree.root!=null){
          tree.root.getLock();
      } catch (InterruptedException e) {
         e.printStackTrace();
       }}
      tree.root = tree.insert(tree.root, RandInt);
      if(tree.root!=null)
      tree.root.ReleaseLock();
   }
  }
}
```

Figure: 3.a

For every thread executing the run method must acquire the lock on the root node of the AVL tree, goes into the critical section and then it performs the insert operation. The tree balancing operations are handled inside the critical section itself. We are using a re-entrant lock so no other thread can acquire the lock on the root node and perform insert(). Therefore, unless the lock is released by previous thread, the current thread cannot acquire the getLock() on root. Hence we achieve mutual exclusion.

3.1.3 DEADLOCK FREEDOM

The algorithm provides deadlock freedom. At least one thread always gets access to the critical section and progresses. Consider the implementation in Figure 3.a. For multiple threads trying to acquire the lock one will always succeed.

3.1.4 STARVATION FREEDOM

The algorithm does not guarantee starvation freedom. Suppose a thread acquiring the lock might go to sleep and the other threads might starve waiting in that thread to release the lock.

3.2 FINE GRAINED LOCKING AVL TREE

The following properties have been verified through experimental results.

The code for insert function is given below: Figure: 3.b

```
while (true) {
                        parentNode = curNode;
                        compare = curNode.data.compareTo(data);
                        if (compare > 0) {
                                curNode = curNode.left;
                        } else if (compare < 0) {
                                curNode = curNode.right;
                        } else {
                                curNode.unlock();
                                return false;
                        }

                        if (curNode == null) {
                                break;
                        } else {
                                curNode.lock();
                                parentNode.unlock();
                        }
                }

                if (compare > 0)
                        parentNode.left = newNode;
                else
                        parentNode.right = newNode;
                parentNode.unlock();
        }
```

3.2.1 LINEARISABILITY

The algorithm is linearisable at the point where the currNode.lock() is acquired. Refer Figure 3.b

3.2.2 MUTUAL EXCLUSION

The algorithm guarantees mutual exclusion. No two threads are present in the critical section at the same time.

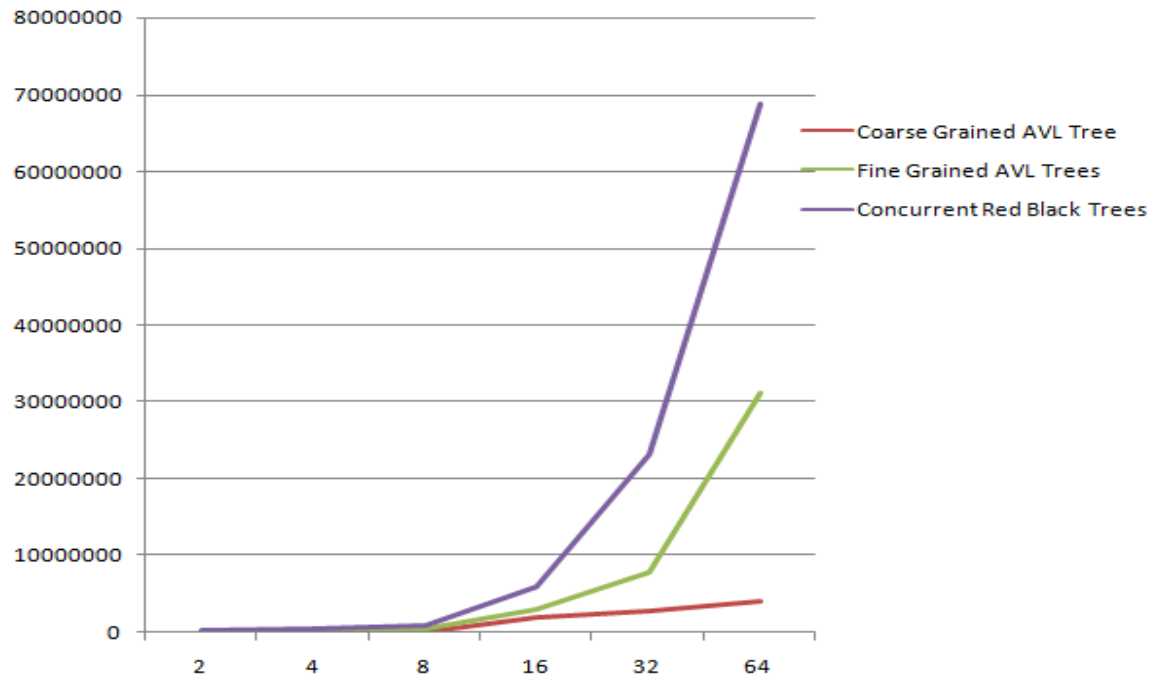
3.2.3 DEADLOCK FREEDOM

The algorithm does not provide deadlock freedom. At some point the algorithm gets deadlocked. But it is rare.

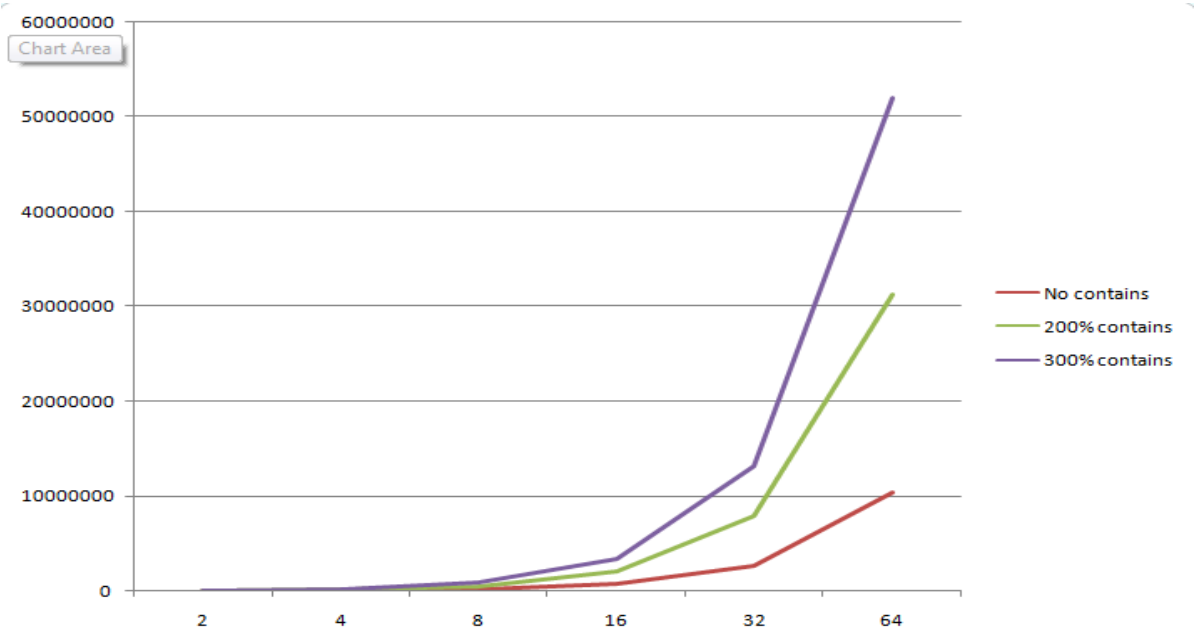
3.2.4 STARVATION FREEDOM

The algorithm does not guarantee starvation freedom. Suppose a thread acquiring the lock might go to sleep and the other threads might starve waiting in that thread to release the lock.

[1] Throughput Vs No. of threads for Coarse Grained, Fine Grained AVL tree and Concurrent Red Black tree. [All readings have been taken on Rlogin.]



[2] Throughput Vs No. of Threads as the Contains Ratio increases for Concurrent AVL {All readings have been taken on Rlogin.]



CHAPTER 5 CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSION

- Amdahl's law states that the speedup increases with increase in parallelization. This can be verified from the above performance analysis where throughput increases with the increasing thread count. Thus, we have verified Amdahl's law.
- After analyzing the performance of Coarse grained locking AVL Trees, Fine Grained locking AVL Trees and Locked Red Black Trees, we conclude that the throughput (T) of each one of the following can be arranged as follows:

T (Concurrent Red Black Tree) > T (Fine Grained AVL) > T (Coarse Grained AVL)

- Fine Grained AVL tree gives better performance than Coarse Grained AVL because fewer locks are acquired and there is relaxed balance and low contention.
- The Concurrent Red Black Tree have a more relaxed balance than the Concurrent AVL Tress, hence it has a better performance due to fewer rotations per insert/delete operation.
- The Concurrent Red Black tree is faster for insertion and removal operations than Concurrent AVL trees.
- The performance of the Concurrent AVL Trees increases when the number of search operations increase, and the number of insert/delete operations reduce. This is because the number of successive rotations reduces.
- Thus, for search intensive tasks the Concurrent AVL Trees must be preferred as they are more strictly balanced thereby providing faster lookups.
- Hence we have implemented and discussed the Concurrent Trees in this project.

5.2 FUTURE WORK

In future the lock based AVL tree algorithms can be improved by achieving starvation freedom of both coarse grained and fine grained approach. We can also improve the efficiency of the algorithm for increasing number of insert and delete operations.

CHAPTER 6 REFERENCES

[1] Concurrent Rebalancing of AVL Trees, A Fine Grained Approach, L. Bouge, J. Gabbarro, X Messegeur, N Schabanel, European Conference on Parallel Processing, May 1997

[2] A Practical Concurrent Binary Search Tree, N G. Bronson, J Casper, H Chafi, and K Olukotun, Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming, pages 257 268, 2010, http://ppl.stanford.edu/papers/ppopp207-bronson.pdf

[3] The Performance of Concurrent Red Black Tree Algorithms, S. Hanke, Proceedings of the 3rd International Workshop on Algorithm Engineering, LNCS, Volume 1668/1999,286-300, http://www.springerlink.com/content/2q914552q2515859/

[4] Concurrent Search and Insertion in AVL Trees, C. S. Ellis, IEEE Transactions on Computers, vol.C-29, no.9, pp.811-817, Sept. 1980.

[5] Parallel algorithms for red–black trees, Park, Heejin, and Kunsoo Park, Theoretical Computer Science 262.1-2 (2001): 415-435.

[6] Report on Parallel algorithms for Red Black Trees, Z. Xue , May 1997

[7] Data Abstraction and Problem Solving, F. M. Carrano, T.M. Henry, Walls and Mirriors, 7th Edition, 2017.