Lock-free Concurrent Self-balancing Binary Search Tree

Ziyuan Chen (ziyuanc) and Ruogu Du (ruogud)

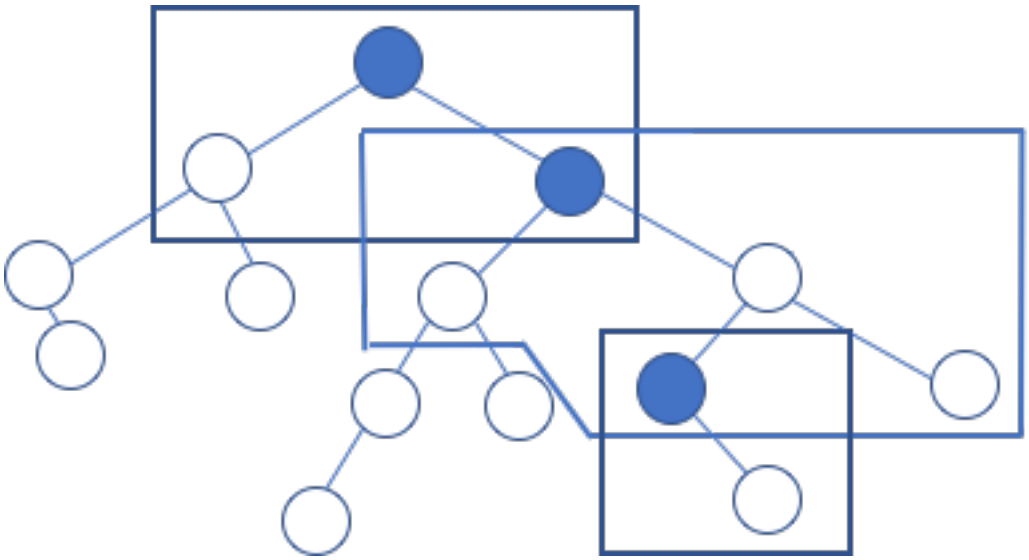

Table of Contents

1. Summary

We designed a lock-free algorithm for concurrent red-black trees and implemented it. The tree was then used as a C++ map container. We compared its performance under various degree of contention. We found that it outperformed both coarse-grained and fine-grained implementations under high contention.

Deliverables:

- A top-down window-based sequential red-black tree
- A templated class of a lock-free concurrent red-black tree
- Benchmark programs
- Lock-based (coarse and fine grained) red-black trees for comparison

Supported platform: X86-64

Supported operating system: Linux, macOS

2. Background

2.1 Key Data Structures

The key data structures here are our modified tree node and red-black tree, which will be explained in the Approach section. Our tree was written as a *templated class* so that it supports any data types as keys and/or values.

2.2 Key operations on data structures

The red-black tree must support:

- Search
- Insertion/Update
- Deletion
- Self-balancing

2.3 Algorithm inputs and outputs

The red-black tree is used as an ordered map to store and retrieve key-value pairs in $O(\log n)$ time.

2.4 Significance and challenges

Binary search trees are known to have notoriously high contention. This is because accessing and modifying certain keys frequently share at least part of the access path with other keys. The situation is even worse for self-balancing BSTs such as the red-black tree because their structures can change. We aim to devise and implement a variant of the red-black tree data structure and algorithm to reduce contention and achieve high scalability compared to current red-black trees.

3. Approach

3.1 Problems with the standard red-black tree

The very first step towards a lock-free red-black tree is to write a sequential tree that is parallelizable. The standard red-black tree is difficult to parallelize due to several reasons. First, all operations need to fix color violations in a bottom-up manner from the node(s) modified up to the tree root. This requires all nodes along the access path and their children and grandchildren to be protected. Often, this is equivalent to locking the entire tree. Second, a key can be stored at any level of the tree, so deleting the node involves fixing its subtrees which would create race conditions unless the entire tree is locked.

3.2 Modified algorithm for red-black trees

After preliminary research, we decided to implement the data structure and algorithm in Robert Tarjan's *Efficient Top-down Updating of Red-black Trees* [1]. There are two major differences between this modified version and the traditional red-black tree.

- All operations are window-based and top-down
- All values are only stored in leaf nodes.

We will explain the details of the data structure and algorithm and give reasons why it is more parallelizable than the standard tree.

3.2.1 Data Structure

The data structure is different from a standard red-black tree in that key-value pairs are only stored in leaf nodes called "external nodes". An external node has no children and is black. All nodes that are not external are called "internal nodes". An internal node has two children. Keys in internal nodes are used for directing the access path to an external node that stores the key-value pair; they do not represent data stored in the tree.

We can argue that this structure has asymptotically the same time complexity for search operations as the standard tree. This is because if we insert N keys, then this formulation will

have at most *2N-1* nodes, while the standard tree will have N nodes. The time complexity for search/traversal is *O(log(2N-1))* which is asymptotically *O(logN)*.

The same following constraints will be maintained in this data structure:

1. All external nodes are black
2. All paths from the root to an external node contain the same number of black nodes
3. The parent of any red node, if exists, is black
4. The root must be black

3.2.2 Search

Searching is very similar to that in a standard red-black tree. The only difference is that the operation only returns if (1) a node holding the key is found, and the node is an external node, or (2) an external node is reached, but the key in the node is not the key being searched. The operation returns the node's address in (1) and NULL in (2).

3.2.3 Insertion/Update

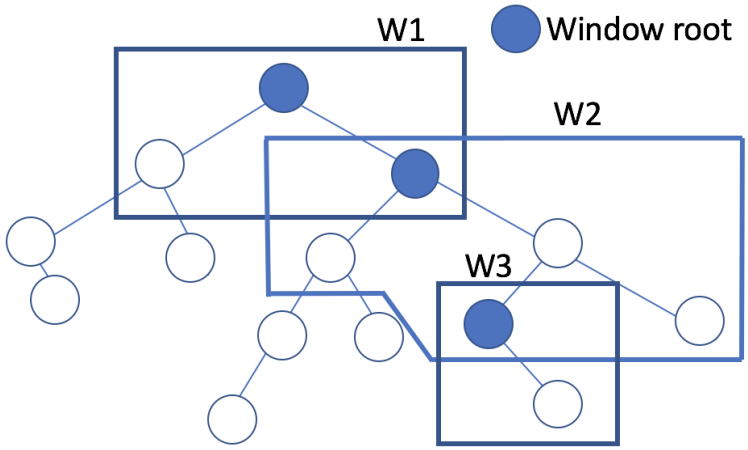

Figure 1 Illustration of Window-based Approach

In insertion, the operation starts from the root. It traverses down the tree following the access path for the key. The term "access path" is defined as the path an operation would take as if it's searching for the key. While it walks down the path, it operates based on the notion of a "window". A window is defined as a chain of nodes going downwards and their children and grandchildren. The following algorithm is used to define the extent of the window:

a. An external node is reached. Proceed to fix any possible color mismatches as in the standard bottom-up algorithm, all the way up to the root of the window, *not the root of the entire tree*.

b. A black node, say *y*, with a black child is reached. Set *y* as the new window root.

c. The following structure in figure 2 is reached. Fix the colors as shown,

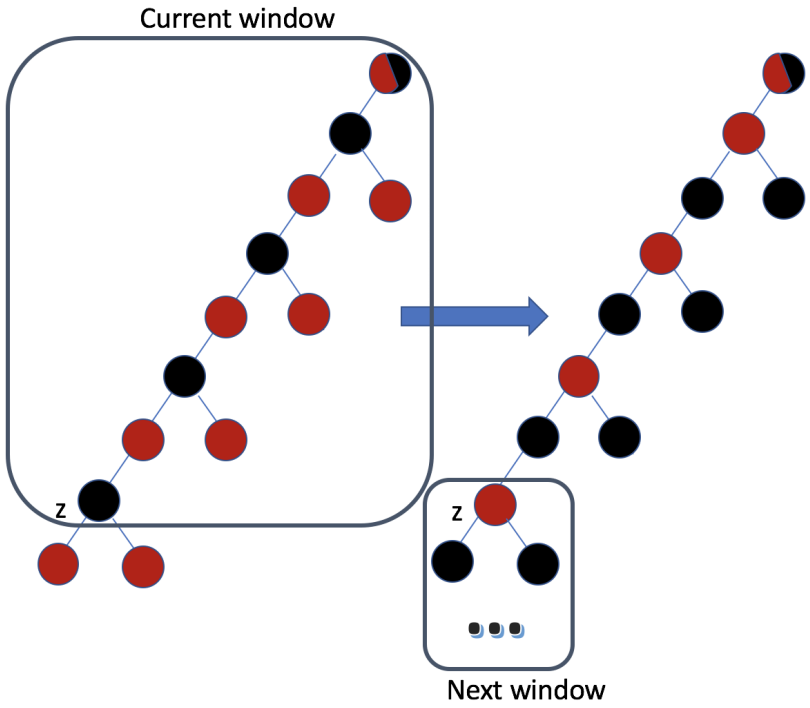

Figure 2 Case c for moving the window down the tree

Using the above algorithm, the operation only needs to fix color violations within the windows. Because the height of a window is at most 8, and even smaller in most cases, this algorithm significantly reduces the nodes that need to be protected when parallelizing this tree structure.

Within a window, we use the same algorithm as the standard red-black tree insertion to fix color violations up to the window root. At the root, one of the rotations in figure 3 is performed. After the rotation, the window (and the tree) is guaranteed to satisfy the color invariants.

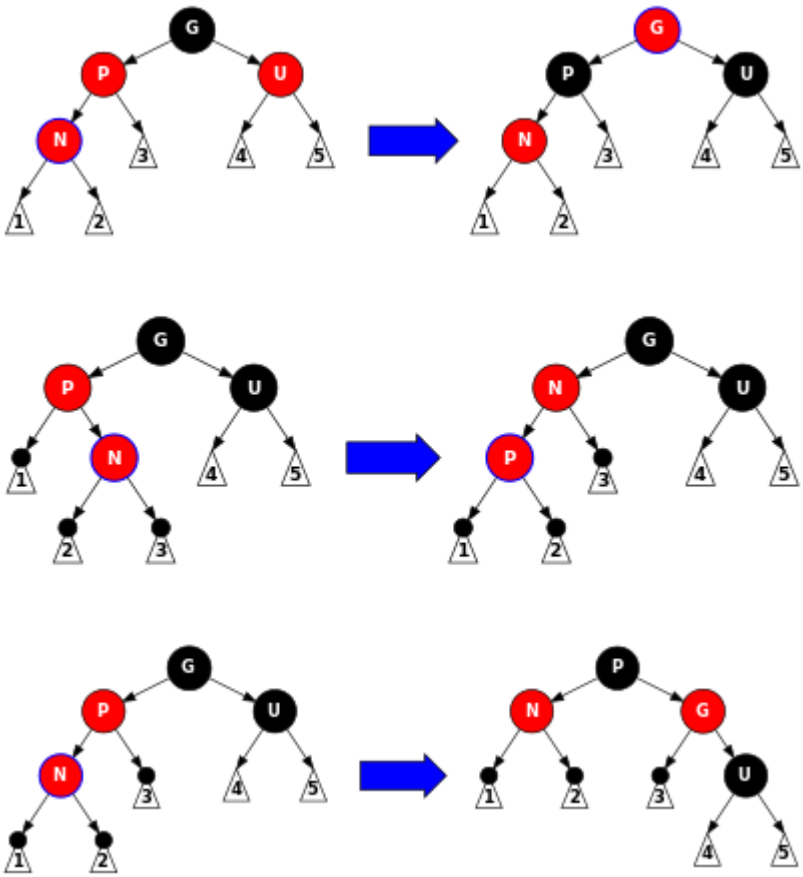

Figure 3 Rotations at Window Root [2]

3.2.4 Deletion

The top-down deletion process is pretty much conceptually similar to insertions/updates. To avoid the bottom-up color-fixing process enforced by the invariants of a red-black tree, we need to constrain all the fixing inside the current window. According to [1], this goal can be achieved by fixing only one situation during the top-down process, which is when the window includes three consecutive black nodes with all black children/grandchildren (shown in figure 5), we need to set the lowest node in the access path and its sibling to red and start the fixing process in the window, then move the window root to this node. When the window includes the leaf node to delete, traditional fixing process of red-black tree can guarantee the modification should terminate within the window.

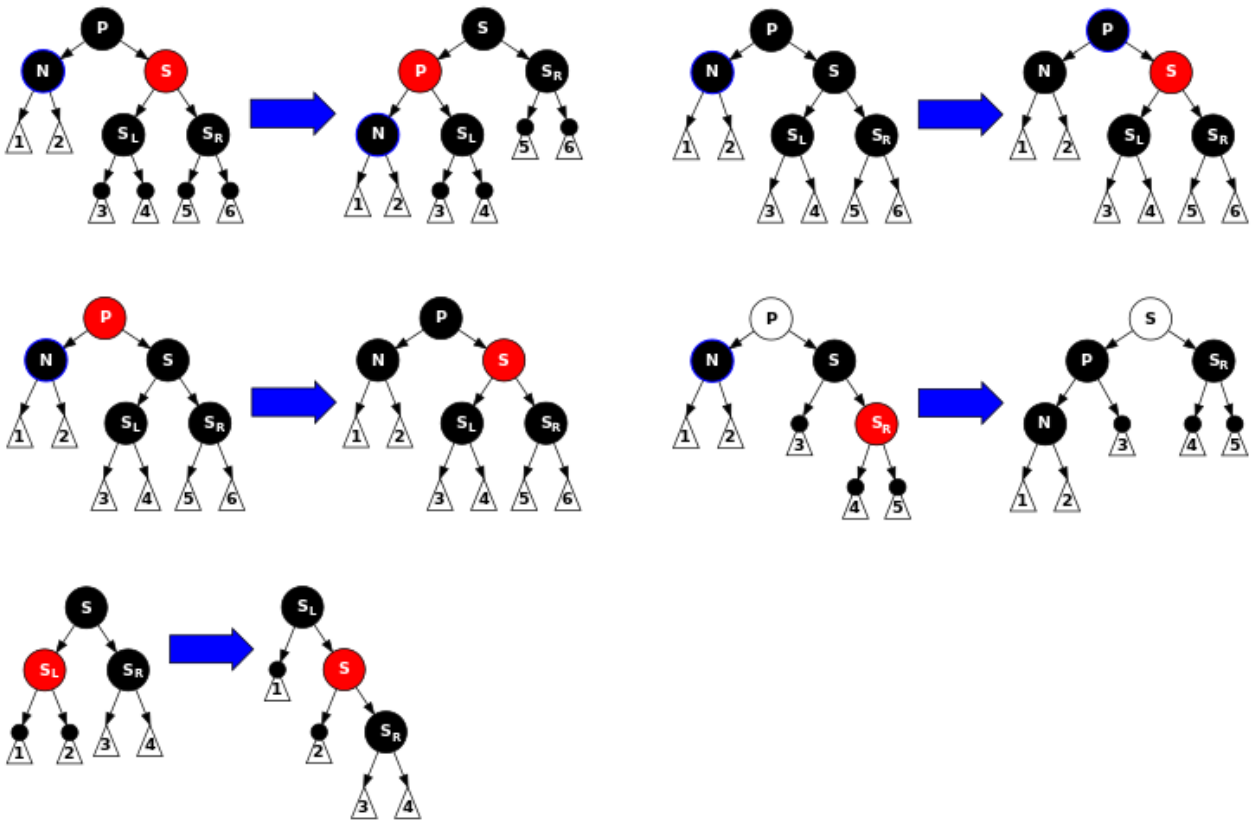

Figure 4 Rotations at Window Root [2]

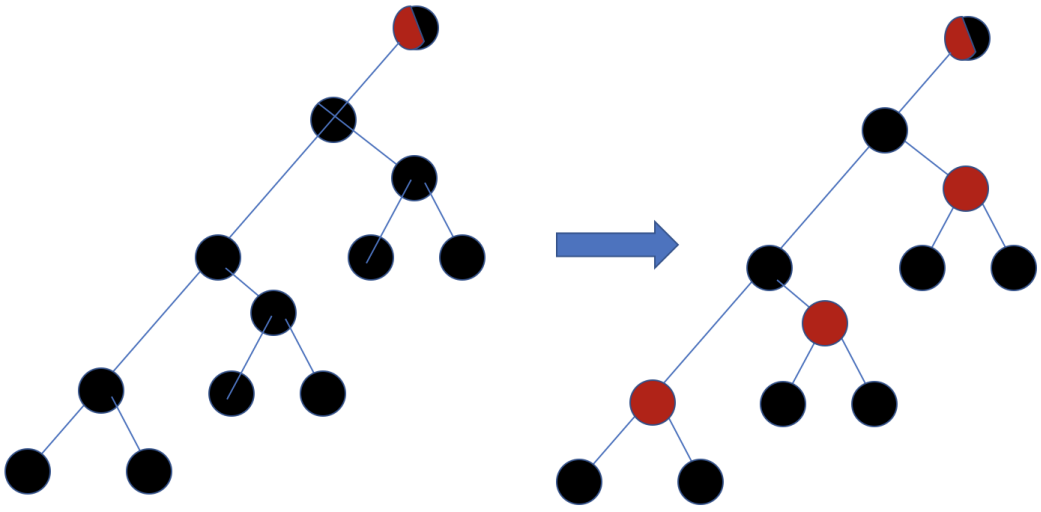

Figure 5 Three consecutive black nodes with black children

3.2.5 Proof of correctness and complexity

Tarjan [1] gave proof that the insertion and deletion algorithms mentioned above are (1) correct and (2) have *O(1)* amortized number of rotations for each operation. Therefore, the above

algorithms should asymptotically be as efficient as the standard red-black tree insertion and deletion algorithms.

3.3 Parallelization of sequential algorithm

3.3.1 New node structure

In order to update all fields in a tree node atomically using CAS, we modified the original tree node structure to consist of two nodes: (1) a pointer node, which only contains a pointer to its data node, and (2) a data node, which contains all other fields, such as *left and right pointers to the pointer node of the next tree node*, *the key and value,* and *ownership/operation information.* Using this modified structure, the tree node can have all its fields updated simultaneously by atomically modifying the pointer to its data node.

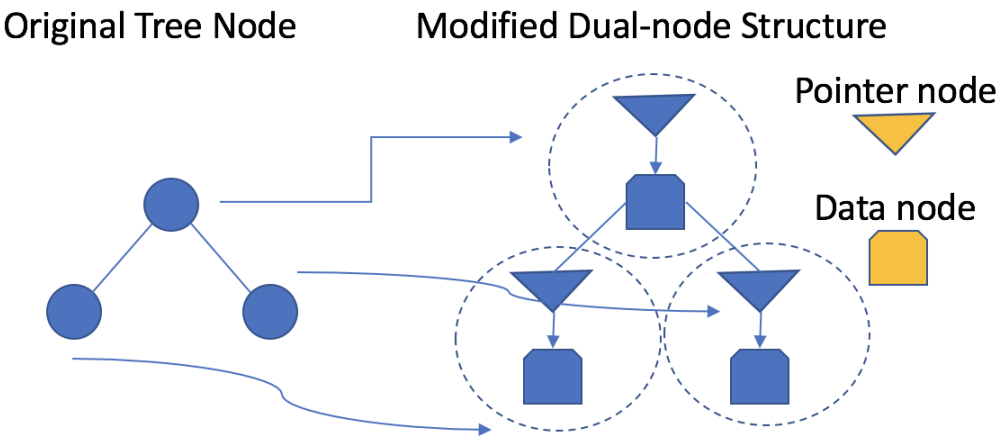

Figure 6 Dual-node structure

3.3.2 Compare-and-swap

std::atomic<>::compare_exchange_strong is used as our compare-and-swap instruction. This is the only primitive in our implementation. It requires hardware support, so our code will only work on supported platforms. We use this primitive operation to compare a new address of a data node of a pointer node to the recorded address of the same pointer node's data node to determine if some other process modified the window while the current process was executing its task.

3.3.3 Acquire ownership

When a process moves to a new window, it first needs to acquire the ownership of the window root. If the window root is not owned, the process creates a new data node, which is a copy of the original data node of the window root except for the ownership and operation information. The ownership is set to OWNED, while the current operation (INSERT or DELETE), the current key to be inserted/deleted, and the current value to be inserted if operation is INSERT, are written to the new data node. This new data node replaces the original data node in the window root using

compare-and-swap. If the data node pointer is changed before the swap, meaning that some other process must have acquired the ownership, this operation will fail. Then the current process will proceed to wait until it is able to acquire ownership. The current process returns from this acquire operation once ownership is successfully acquired.

3.3.4 Release ownership

Because our algorithm is window-based, once a process finishes its operation in its copy of the window, it will swap the top-most data node in the copy back to the window root node in the original tree by comparing to the address of the window root's data node when the process first acquired its ownership using compare-and-swap. If successful, then the window root node becomes FREE again and its ownership is successfully released. If not, some other process must have already swapped in a correct window and released ownership. Therefore, the current process will simply discard its copy of the window knowing that its operation has already been completed.

3.3.5 Search

As we will explain later, because of our parallel insertion and deletion algorithms, our search operation is guaranteed to not have any race conditions. It remains unchanged from the sequential algorithm. It does not require any synchronization or locking.

3.3.6 Insertion and deletion

Parallel insertion/deletion is modified from the sequential algorithms in the following way, shown in figure 7:

(1) Assume the current process is p1. Every time a new window root node x for window W is chosen, it tries to acquire the ownership of x. If x is already owned by other processes, it helps that process before continuing. It tries to acquire the ownership again after helping.

(2) After the ownership of x is successfully acquired, p1 continues its own insertion/deletion. It does so by following the sequential algorithm until it reaches the end of W. For each node in W, it looks at their ownerships and helps them if they are already owned. Then, when p1 reaches the end of the current window, and there is color violation, it fixes the color in the window by duplicating the window into W', then swaps the pointer to the data node of the the window root x by its counterpart in the duplicated window W'.

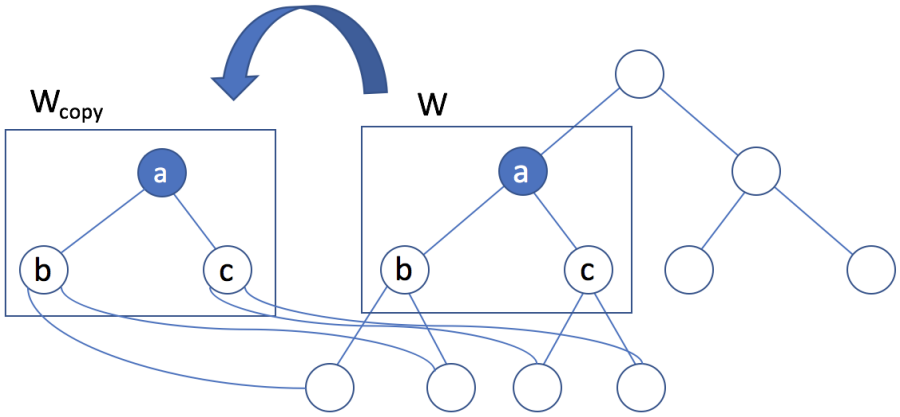

Figure 7 Window copying

This parallel algorithm is non-blocking because even if a process p holding the ownership of a node is crashed or stuck, other processes will finish its operation on its behalf. After the helper process returns from helping, the node's ownership will be released, so that if p finishes later, it will simply fail the compare-and-swap. All the work it did will be in vain, which is a waste of computing resources, but this is a tradeoff to ensure the non-blocking property of the algorithm.

Because of window copying, the tree structure remains unchanged until the window operation is completed and the new window is installed. In addition, window installation is atomic via compare-and-swap, Therefore, the search operation is guaranteed to not run into any race conditions. It is also guaranteed to be sequentially consistent within a process, i.e. if the same process inserts a key K before searching for K, then search is guaranteed to return the node containing K. The same holds for deletions. Consequently, the search operation is the same as the sequential algorithm.

3.3.7 Technologies

Our project was entirely in C++ 11. The libraries <atomic> and <vector> were used to implement our data structure. We targeted any machines with X86-64 Architecture, which is guaranteed to support single-word compare_and_exchange in C++ std::atomic.

4. Results

4.1 Benchmark

Our benchmark program uses pthreads to launch a mix of parallel insertions (33.3%), searches (33.3%) , and deletions (33.3%). The total number of keys to be inserted, searched, and deleted are five hundred thousand (500k) for each operation type. The nodes for each type of operation

are equally divided among pthreads. Therefore, the total test case size is 1.5 million (1.5M) keys. Both the keys and the values are integers.

The measured execution time is the *wall-clock* time for launching the pthreads, executing the tasks, and joining them at the end. The correctness of the tree is verified at the end, but the verification time is not part of the execution time.

The baseline for the lock-free as well as the reference implementations was their respective single-threaded execution time of the benchmark program.

We ran the benchmark on the following platform:

Operating System: macOS 10.13.1

CPU: Intel Core i7 7700HQ

Number of Cores: 4

Number of physical threads supported: 8

Clock speed: 2.80 GHz

4.2 Reference implementations

4.2.1 Coarse-grained locking

The red-black tree with coarse-grained locking is implemented by locking a single mutex at the beginning of every insertion/deletion/search to the tree, and unlocking when every operation is finished.

4.2.2 Fine-grained locking

The fine-grained locking concurrency control is implemented based on our window-based algorithm. A mutex is added to every node in the. When an operation is being performed, we need to always lock the root node in the current window. When the window moves down, we need to first lock the next window root before releasing the lock at the current window-root. The fixing process remains the same with the top-down algorithm.

4.3 Execution time and speedup

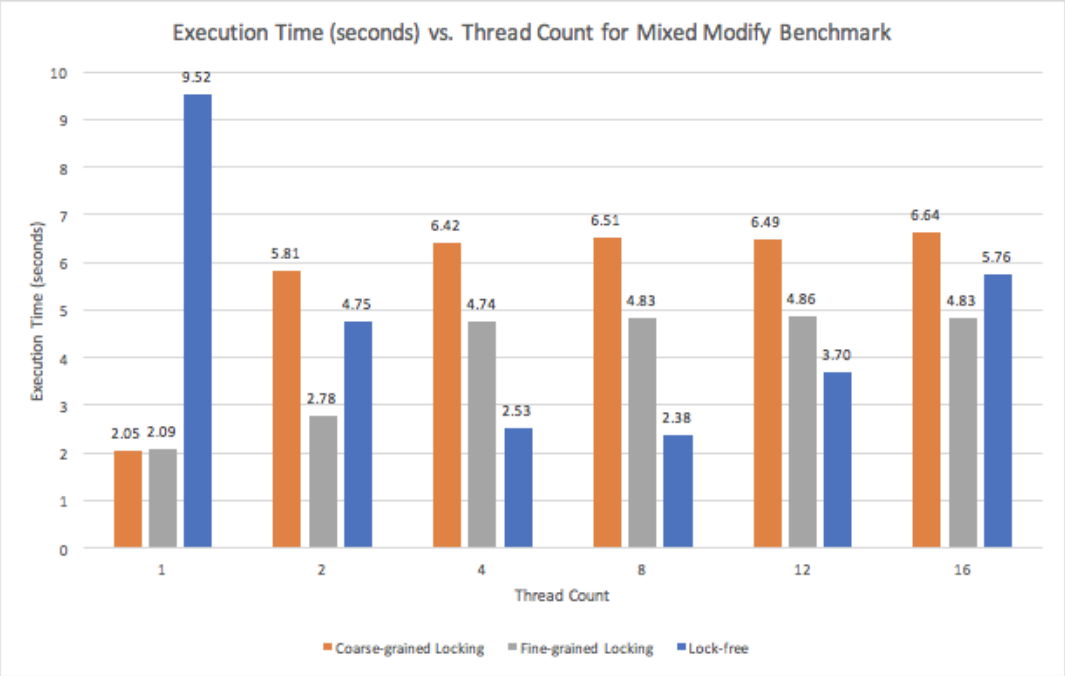

Figure 8 Execution Time

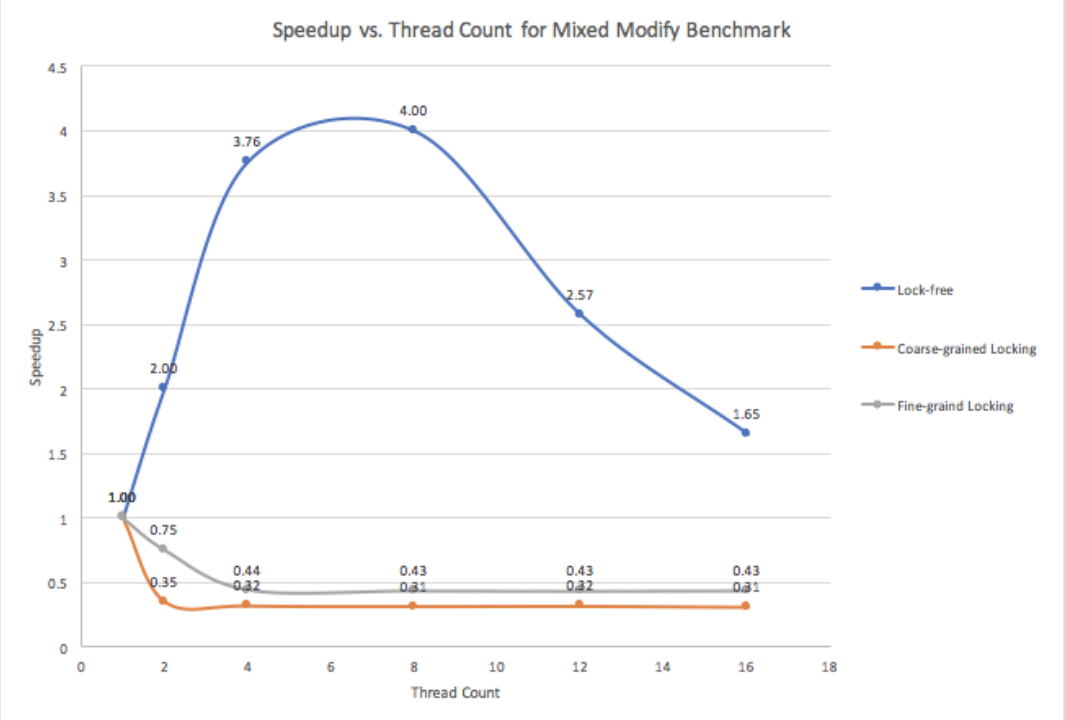

Figure 9 Speedup

4.4 Memory usage

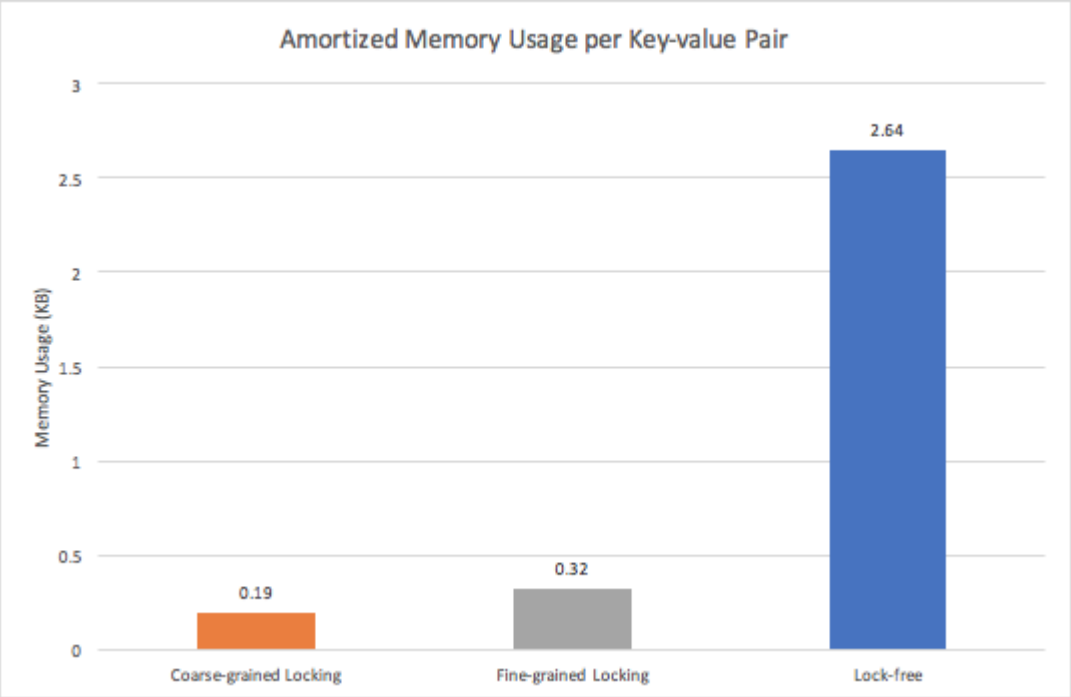

Figure 10 Memory usage

5. Analysis and discussion

5.1 Execution time

In figure 8, the lock-free implementation presents a much higher execution time compared to references on single thread. When thread number increases, the execution time of the lock-free implementation decreases at the beginning and outperforms both references at 4 and 8 threads. It starts to increase again at 12 threads and becomes close to the references at 16 threads.

The higher execution time on single thread can be explained by the overhead of our window-based approach, which requires copying and CAS when an operation is being performed. The decrease in execution time can be explained by the amortized workload of each thread decreases as the thead number grows. The later increase could be caused by the higher contention when threads compete for ownership for window roots, and the cost for more context switches when thread number exceeds the processor number.

5.2 Speed up

Figure 9 shows the lock-free implementation scales better than the references. It is reasonable because most of the operations happen near the bottom of the tree. Thus contention for the

window roots near the top shouldn't be too high as windows slide rapidly at this stage. When a window slides to the bottom, contention should even decrease because less threads should be competing for the same nodes.

The reason why the coarse-grained locking implementation cannot scale is pretty clear. For this implementation, multithreading not only cannot help with concurrency, but may increase the cost for locking and unlocking.

The reason why the fine-grained locking scales poorly is rather vague. Intuitively, the contention it encounters should be similar to the lock-free implementation. We surmise this behavior could be caused by context switches, because locking and unlocking could lead to more context switches when threads sleep and wake. To prove this hypothesis, we conducted deeper analysis which will be introduced below.

5.3 Memory Usage

Our implementation consumes more memory than lock-based implementations. As shown in figure 10, for 500,000 key-value pairs and 1,000,000 insertion and deletion operations, our lock-free implementation consumed about 2.64 KB of memory per key-value pair compared to 0.32 and 0.19 KB for coarse-grained and fine-grained locking respectively. This was due to the window-based algorithm. A simple analysis yields that the space complexity of our algorithm is $O(K)$, where K is the number of insertion and deletion operations. Every time a process executes an operation, it duplicates a series of windows where color violations occur. We cannot immediately release the memory because some other processes could still be working on the nodes released by a process. We optimized memory reclamation as much as possible without causing concurrency issues, but still leaving behind a significant portion that was not reclaimed. We intended to devise a memory reclamation scheme using hazard pointers, but due to the complexity of the reclamation algorithm and limited time, our algorithm was not able to reclaim all memories at the time the this report.

5.4 Deeper analysis

In previous analysis, we deduced that context switching could explain why our implementation had better speed-up over lock-based implementations under high contention. We measured the number of context switches in each implementation under various thread counts. The benchmark program and the problem size remained the same.

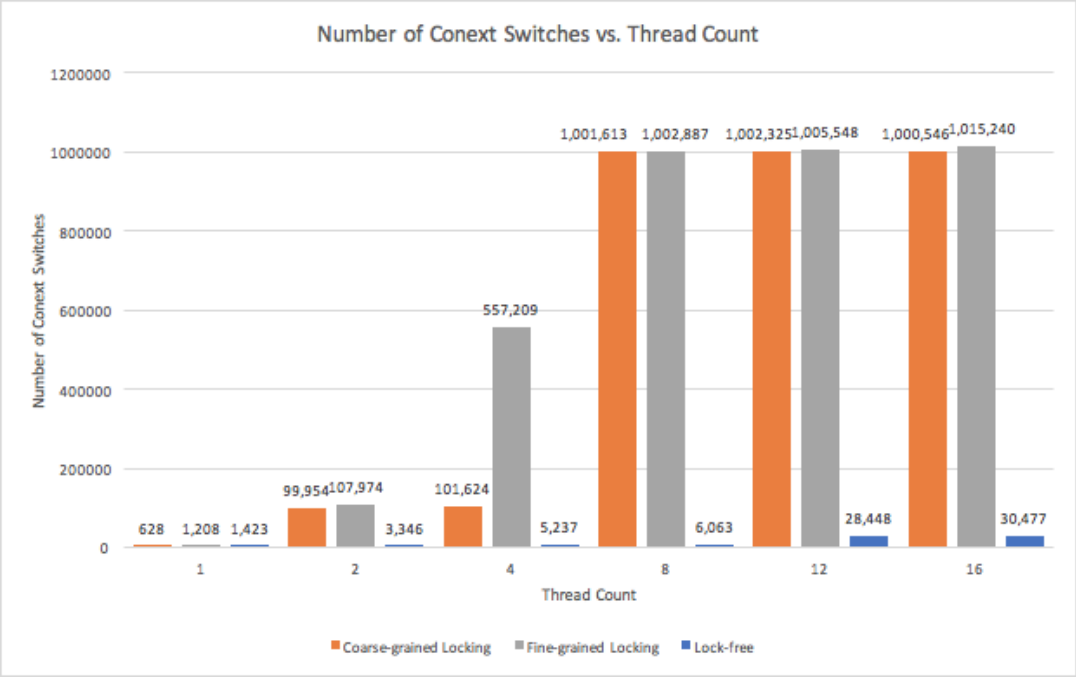

Figure 11 Number of context switches

As we can see in figure 11, all three implementations had little context switching under 1 thread. However, the number of context switches for lock-based implementations increased drastically at 2 and 4 threads. Most notably, fine-grained locking had more context switches than coarse-grained locking at 4 threads. This was because the fine-grained locking implementation acquires locks much more frequently. When there were 8 or more threads, both coarse and fine-grained locking had around 1,000,000 context switches while the lock-free implementation only had a few thousand. This was because the total number of physical threads supported on the platform (Intel Core i7 7700HQ) was 8, and at least 1 thread was dedicated to other processes such as the OS. When the thread count exceeded 8, we started to have more threads than the number of physical threads supported, resulting in a steep increase in context switches. Our lock-free implementation also had a significant increase in the number of context switches, but the number was relatively little (about 1/20) compared to lock-based implementations. This was because in our lock-based implementation, all threads are constantly doing work, while in lock-based operations, threads that are waiting to acquire locks are swapped out by the operating system. Based on our measurements, we conclude that hypothesis that the lock-free implementation scaled better than lock-based reference implementations because of context switching is true.

6. Conclusion

We achieved all the original goals in our project proposal by implementing a lock-free concurrent red-black tree. We started from a modified sequential algorithm then parallelized it.

We performed benchmarks and performance analysis, and found that our implementation scaled better than the reference implementations. Upon deeper analysis, we concluded that the scalability was achieved at the expense of higher overhead due to window and node copying and higher memory consumption.

In summary, our data structure has the following desirable properties:

1. Lock-free
2. High scalability
3. Search operations are fast and requires no atomic or synchronization instructions
4. Low contention compared to the standard red-black tree thanks to window-based approach
5. Templated class - supports any data type as key and/or value

It has the following drawbacks compared to a lock-based red-black tree:

1. High space complexity *O(#number of insertions and deletions)*
2. Overhead due to window copying

Our benchmark result was optimal at 4, 8, and 12 threads, which are the typical numbers of threads one would use on a modern CPU, such as the Core i7. Therefore, we conclude that our lock-free data structure is practical for a large number of parallel programs. In practical software running on typical CPUs, it would significantly outperform lock-based implementations because of its high scalability.

7. Extra work

There exist many optimizations for our lock-free algorithm. For example, the amount of extra memory required can be reduced using a separate *record* data structure to store the values rather than storing them in external nodes [3]. The external nodes only need to store points to their *records* and the pointers can be updated using compare-and-swap. Therefore, the values themselves won't need to be copied. We attempted to implement this scheme but were not able to finish due to the deadline.

Also, we discovered a way to achieve wait-freedom, guaranteeing thread-level progress. We intended to have a process help other processes finish to reduce the amount of time wasted when it has no work to do. We implemented this mechanism by storing in owned nodes the operation type, the key to be inserted/deleted, and the value to be deleted if applicable. Using the same compare-and-swap scheme for the data node of a window's root, we could achieve this mechanism. However, the actual implementation brought forth many details in ensuring

correctness and avoiding race conditions. Although we were able to finish the implementation, we were not able to make it bug-free. It produced a tree containing all the key-value pairs correctly, but had incorrect color invariants resulting in an unbalanced tree with suboptimal performance.

8. List of work done by each student

The following items were completed with work divided equally between the two team members.

1. Research on concurrent red-black tree algorithms.
2. Research on lock-free red-black tree variants.
3. Implement sequential window-based top down red-black tree.
4. Implement functions to verify tree correctness
5. Parallelize red-black tree with window-copying and CAS.
6. Implement baseline concurrent red-black trees.
7. Implement test cases
8. Measure multiples versions of concurrent red-black trees with the lock-free version.
9. Analyze the result and propose hypothesis
10. Deeper analysis
11. Conclusion

Works Cited

[1] R. Tarjan, "Efficient Top-down Updating of Red-black Trees," 1985. [Online]. Available: ftp://ftp.cs.princeton.edu/reports/1985/006.pdf. [Accessed 16 Nov 2017].

[2] Wikipedia, "Red-black Tree," [Online]. Available: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree. [Accessed 20 Nov 2017].

[3] A. Natarajan, "Concurrent Wait-Free Red Black Trees," 2014. [Online]. Available: https://pdfs.semanticscholar.org/5afc/38731711a6516640e360c8f8394707bbff19.pdf. [Accessed 15 Nov 2017].

[4] J.-J. Tsay and H.-C. Li, "Lock-free concurrent tree structures for multiprocessor systems," in *Parallel and Distributed Systems*, 1994.

[5] P. Chuong, F. Ellen and V. Ramachandran, "A Universal Construction for Wait-Free Transaction Friendly Data Structures," *SPAA' 10,* June 2010.