Lock-free Red-black Tree Implementation with CAS

Shun Zhang* and Guancheng Li*

*{shunz, guanchel}@andrew.cmu.edu

1 Summary

In this project, we implemented a lock-free red-black tree based on [2], supporting multi-threads insert and remove. We also implemented a fine-grained version presented by [3], while this version only supports insert operation. After implementing them, we compared the performance of these two versions as well as the serial version. All of them are implemented in C++ on CPU.

Main work

1. Implemented sequential red-black tree
2. Implemented fine-grained red-black tree (only insert), which is our extra goal
3. Implemented lock-free red-black tree using CAS (First open-source fully functional implementation that scales)
4. Found that the *SpacingRule* and *MoveUpRule* introduced in [2] is unnecessary and derived a much simpler marker mechanism

In this report, we will cover lock-free implementation details in Section 5.1. We will not cover the conventional red-black algorithm and we will mainly focus on the important parts that are related to lock-free. We will also discuss about how we found the two rules introduced by Kim and Cameron [2] is unnecessary. In Section 5.2, we will introduce our fine-grained implementation of insertion on red-black trees. At the end, we compare the performance of our three version of insertion and deletion and show the results in graphs. According to the results, both our lock-free and fine-grained implementations are efficient and obtain linear speedup.

2 Previous implementations

As you may find, the remove function of lock-free red-black tree is hard to implement (more details later). At this moment, after we have done our own tests, we would like to find whether there are other implementations of lock-free red-black tree and compare those with ours. We do find some repositories, josecamacho8 and rutvij93, on GitHub that are trying to implement the lock-free red-black tree, but none of them succeed to implement a scalable and efficient remove function. One of them doesn't even have an efficient insertion algorithm. According to our tests, the rutvij93's version got a speedup smaller than 1 when doing pure insertion in multi-threading, which means this version only has negative effects when there are more threads! And the other one only works with two threads and 5 elements according to its README.

3 Background

A red-black tree is a kind of self-balancing binary search tree. Because of its good performance, it is widely used in different applications, such as Linux Kernel and C++ STL. Red-black tree's excellent performance owe to its balancing rules, while the rules are complex and thus hard to implement in parallel version. Our implementation references the algorithm in [2], which propose "flag" and "intention marker" to avoid deadlock during insert and remove operations.

4 Notations

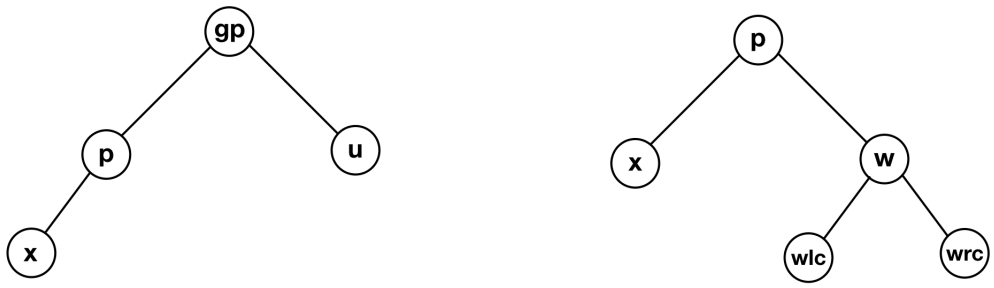
In this report, there are some common notations of nodes listed in Table 1. The notation x represents the node that has just been inserted during the insert phase or the node that replaced the removed node.

Notation	Meaning
x	the target node during insert/remove phase
p	the parent node of x
w	the sibling node of x
gp	the grandparent node of x
u	the uncle node of x, sibling node of p
wlc	the left child of node w
wrc	the right child of node w
flag	a boolean field in each node, true means the node is held by a thread
marker	an integer field in each node, the number represents the thread ID

Table 1: Notations of nodes

```
1    bool expected = false;
2    while (!root->flag.compare_exchange_weak(expected, true));
```

Figure 1: An example of using CAS to get the flag of a node(root is a node)



(a) Insertion Local Area

(b) Deletion Local Area

Figure 2: Local Areas for insertion and deletion

5 Implementations

5.1 lock-free version

5.1.1 Flag Field

Based on Ma's insertion algorithm presented in [4], we add an extra boolean field called "flag" to each node. When the flag of a node is set to true by a process, that process gains the control of this node. Since this implementation is lock-free, we use CAS operation from C++ atomic library to check and set flag fields. An example of using CAS is shown in Figure 1. If the flag value is already true, it means that some other process is operating on this node, and *compare_exchange_weak* will fail and return $false$. On the other hand, if the flag value is false at present, *compare_exchange_weak* will successfully set its value to *true* and return *true*. At that time, that process has obtained the access of this node.

5.1.2 Local Area and Notations

To avoid contention, Ma proposed local area concept in [4] to make sure a process can finish its operation without being interrupted. Local areas for insert operation and remove operation are shown in Figure 2. To obtain the access of a local area, a process need to get all flags of nodes in the local area using CAS operations. And the order of obtaining flags is also important. As we learned from class, an inappropriate order of getting shared mutex may cause deadlock. Our order of getting flags for Insertion Local Area is $x \to p \to gp \to u$. And the Deletion one is $x \to p \to w \to wlc \to wrc$.

```
restart:
    do {
        root_node = root->left_child;
        expect = false;
    } while (!root_node->flag.compare_exchange_weak(expect, true));

    tree_node *y = root_node;
    tree_node *z = NULL;

    while (!y->is_leaf)
    {
        z = y; // store old y
        if (value == y->value)
            return y; // find the node y
        else if (value > y->value)
            y = y->right_child;
        else
            y = y->left_child;

        expect = false;
        if (!y->flag.compare_exchange_weak(expect, true))
        {
            z->flag = false; // release held flag
            usleep(100);
            goto restart;
        }
        if (!y->is_leaf)
            z->flag = false; // release old y's flag
    }
```

Figure 3: A hand-over-hand searching with back-off applied

5.1.3 Back-off Pattern

Since we use CAS operation instead of mutex locks here, we need some back-off stage when CAS operation fail. As shown in Figure 1, in some cases we use *while* loop with CAS operation, which is similar to a spin lock. But this may cause some blocking problems. For example, a process with some nodes' flags may keep trying on a new node, while some other processes want to pass through the nodes it already has. Thus we apply a "back-off pattern" in our code, which allows a process releases its possessed flags after a failed CAS. An example of searching target position hand-by-hand is shown in Figure 3. In this example, whenever a CAS operation fails, the process will release its flags(at most two flags here) and go back to *restart* to start from the root.

5.1.4 Lock-free Search

The lock-free search is very much similar to the fine-grained version of search, i.e. get flag hand over hand using CAS. If any CAS fails, then restart from the root node.

5.1.5 Blocking Lock-free

Before we step into lock-free insert and remove, we would like to mention that though this algorithm is called lock-free and it does use CAS instead of locks, it is still a blocking algorithm [5]. This is caused by the characteristics of red-black tree that in Case 2 of the remove phase [1], the target node moves up as well as the local area. And more importantly, before doing so, the node that should be deleted has already been removed from the tree. That is to say, it is impossible to restart from the very beginning of this remove operation and since we haven't finished the fixup yet, we couldn't even release the flag on the target node x. If we do so, the characteristics of red-black tree is totally broken and other threads are now working on a wrong red-black tree.

So, as a result, once we have the flag of target node x, we will never release it until the remove operation is done. Actually, we will not release any flag within our local area until new local area is obtained or the remove operation completes. This is blocking and an immediate consequence is that we have to deal with a potential deadlock situation shown in Figure 4. There are two threads, black and red, both doing a remove operation and both successfully hold the flags with their local area (wlc and wrc are outside the figure). However, since they will never release any flag in their local area and if, for example, thread (with color red) is in Case 2 of remove phase, its new local area is the 5 nodes within the red dash line. The new local area includes three nodes that are in the black thread's local area. Even if black area could release its flags on node p and w, as long as it still holds flag on node x, the two threads would end up in deadlock.

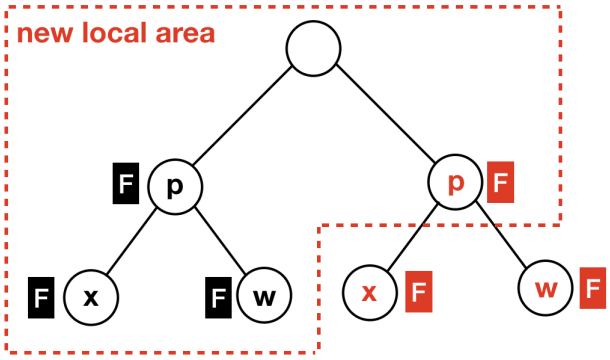


Figure 4: A deadlock situation caused by blocking

5.1.6 Markers

This deadlock problem is caused by the two threads being too close to each other. To prevent this from happening, Kim and Cameron [2] introduces markers to address this problem. It's

key idea is that thread must get four markers above node p before start removing. Markers are like loose 'flags' that a thread can put a flag on any node without a flag but markers can only be put on nodes without any markers or flags.

The four markers above the node p is shown in Figure 5 with label 'M'. In this case, a thread (with black color) has already set its markers and when another thread (with color red) intends to set its own markers on the blue node, it fails. This ensures that there is sufficient distance between the two threads.

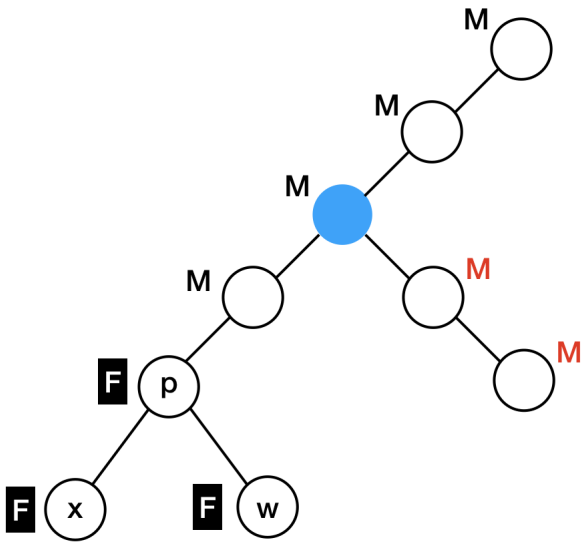


Figure 5: Markers that prevent two threads from being too close

5.1.7 Lock-free Insert

The pipeline of lock-free insert operation is shown in Figure 6. We assume the parent node is always the left child of the grandparent to simplify the graph. The operations in the lower right part are reverse if the parent node is the right child of the grandparent.

The blue part of it contains operations related to the local area. An inserting process must obtain all flags in its local area before it starts to modify the structure, and it also needs to release all flags after finishing all modification to avoid deadlock.

The orange part of the pipeline is fixing up the tree structure after inserting the new node. This aims to make sure that the rules of red-black tree are followed.

5.1.8 Lock-free Remove

The whole pipeline of lock-free remove function is shown in Figure 7, where the white part is the same to sequential version and the colored part is lock-free related. Remove fix up is shown in Figure 8.

After we get the actual node to be deleted, we begin to setup for its local area as shown in Figure 2 (b). This is done by getting flags in the order of $x \to p \to w \to wlc \to wrc$ and

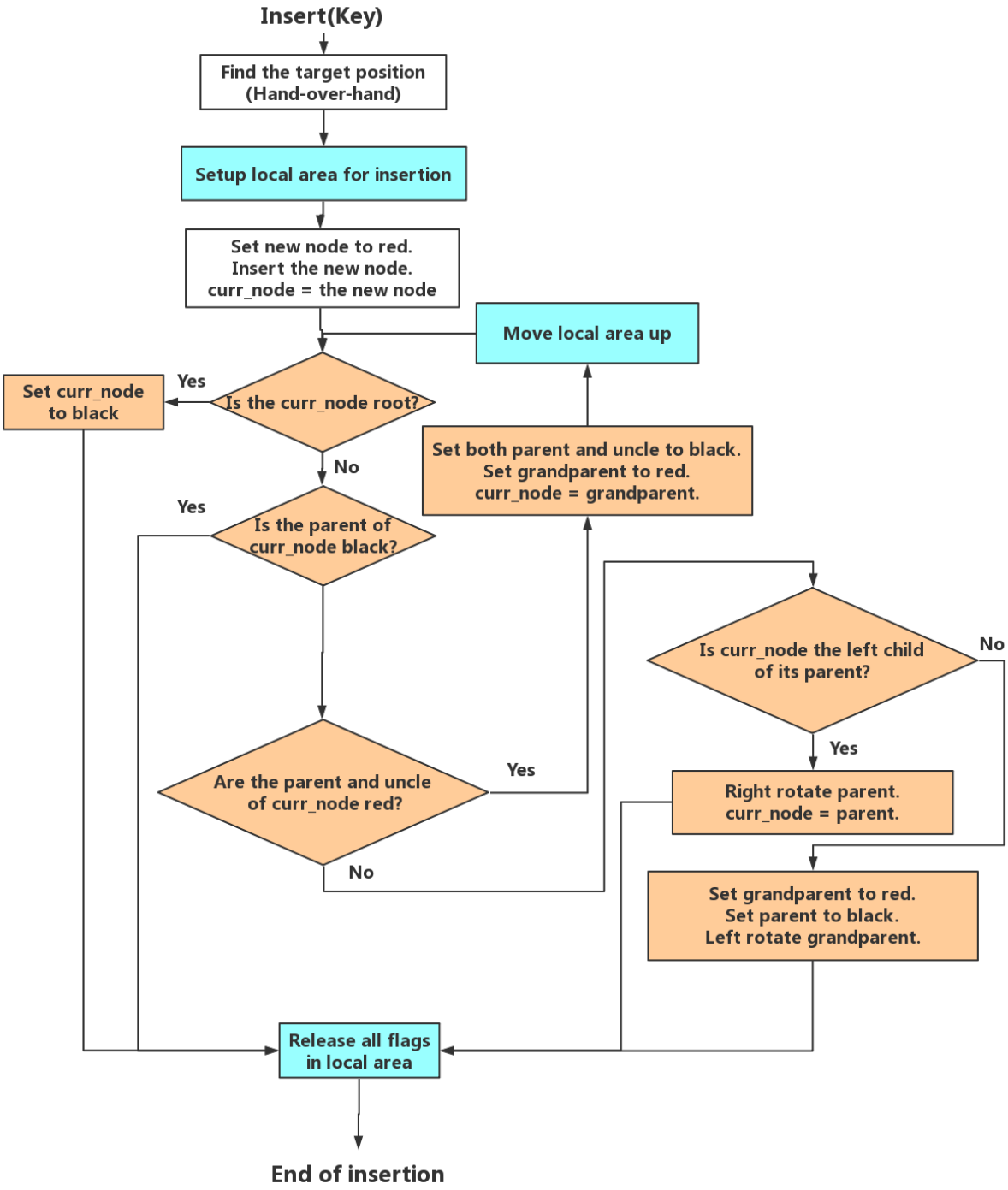


Figure 6: The pipeline of lock-free insert function

in some cases wlc and wrc do not exist when w is a null node. In those cases, the local area is degenerated into a smaller one with only $\{x, p, w\}$.

After we get flags of our local area, we start to set markers on the four ancestors of node p. The necessity of markers has already been discussed by Kim and Cameron [2]. Long story short, markers will prevent two threads from getting too close to each other and result in a deadlock situation shown in Figure 4.

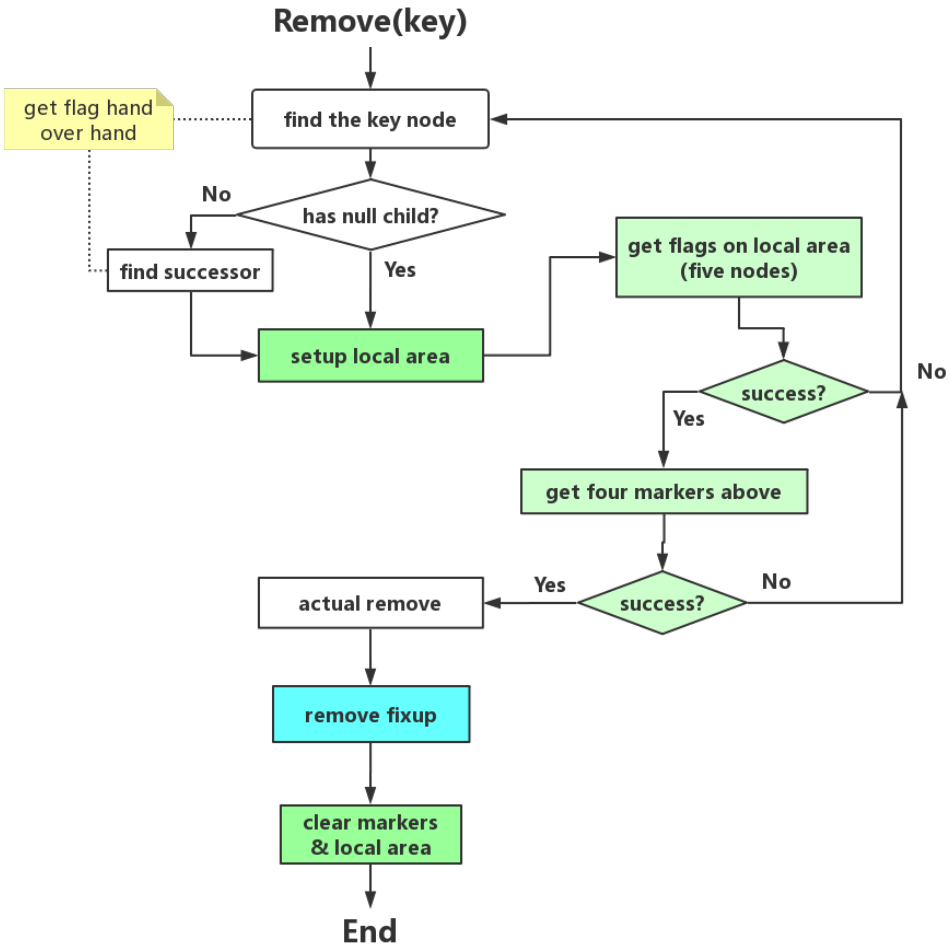


Figure 7: The pipeline of lock-free remove function

After the node has been actually removed from the tree, if the replaced node's color is black, then fixup is needed to maintain the property of red-black tree [1]. The pipeline of the remove fixup is shown in Figure 8. 'Case1 fix up' and 'Case3 fix up' are mainly responsible for the change in local area.

However, according to Kim and Cameron [2], these two fixup case should also be responsible for changes in markers. Doing so will result in 'double marker' problem shown in

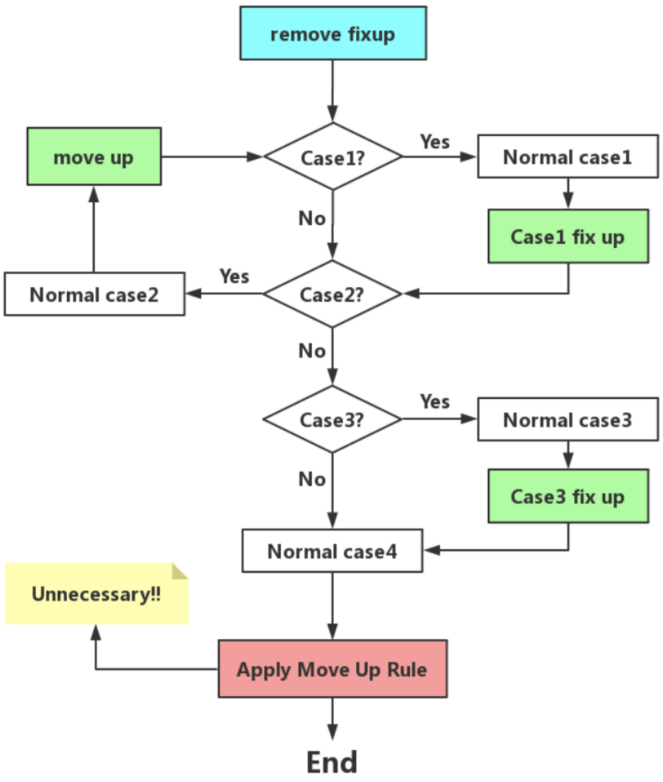


Figure 8: The pipeline of remove fix up

Figure 9. So Kim and Cameron [2] introduces *SpacingRule* and *MoveUpRule* to address this problem.

This is OK under their assumption that markers should always follow the relationship of parent-child. That's why the top black marker in Figure 9 goes to the blue node after rotation.

Simplified markers mechanism We found the above assumption is **unnecessary** and not in line with the spirit of lock-free. So we come up with a much simpler mechanism about markers that a thread should try to set four markers every time it performs 'move up' (Figure 8) and only continue when the four markers are successfully set. By doing so, the safe distance between threads are also reserved because of the markers and the two fixup cases in Figure 8 becomes much easier that they are only responsible for changes in flags and they will clear any markers in the local area.

Such simplification is confirmed by our tests that without *SpacingRule* and *MoveUpRule*, our lock-free implementation also works and is the same efficient as the one with two rules implemented. So, we can say that the red part in Figure 8 is unnecessary as well as the *SpacingRule*.

Address potential livelock As a result of the blocking property mentioned in Section 5.1.5, we will have a potential livelock situation shown in Figure 10. In this situation, the black thread is trying to move up by getting the four markers above. Meanwhile, there is

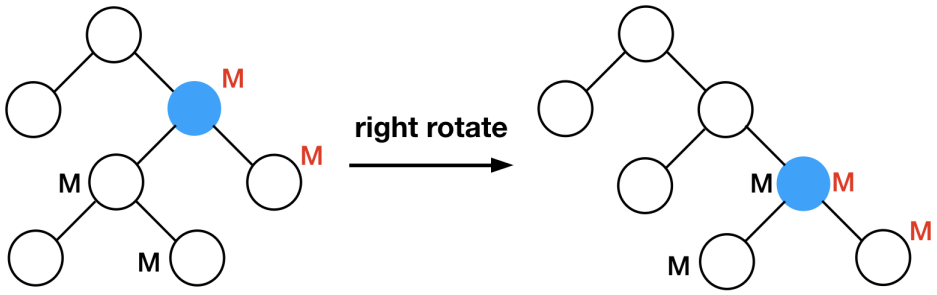


Figure 9: Double marker problem caused by rotation

another green thread trying to find the green node by getting flags hand over hand. And from the figure we can see that the green thread will never get to the green node until the black thread completes. So the green thread is wasting time trying to do so, which will possibly cause the black thread failing to get the markers. And if there is another thread, with color blue, also start doing so. They will possibly end up in a temporary livelock because there is still a chance when the black thread succeed to set the markers. However, since the green thread and blue thread are doing useless work, we decided to let the green and blue threads back-off a little period of time (100us). It turns out to be very helpful and 100us is set arbitrarily and could be further explored.

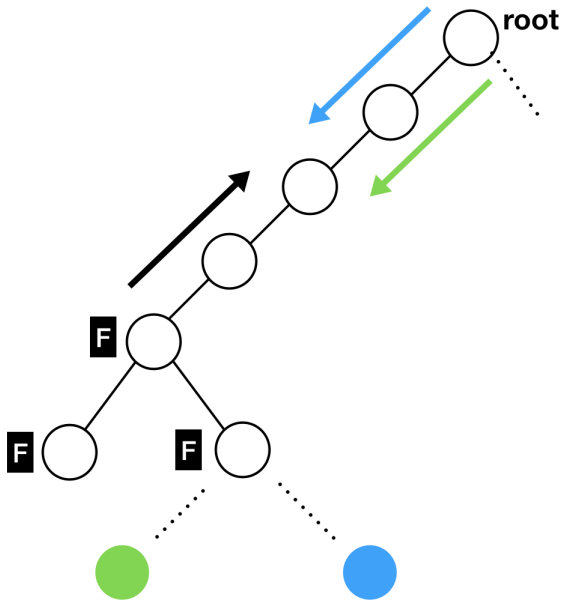


Figure 10: Potential livelock due to blocking

5.2 Fine-grained version

We also implemented a fine-grained version according to [3]. This paper introduces a parallel insertion algorithm on red-black trees. This algorithm also adds an extra mutex on each

node and obtains all needed locks before modifying the structure. The needed locks of single insertion are similar to the local area mentioned above. But there is still some difference: the fine-grained version only take locks before it is going to manipulate those nodes, while the lock-free version sets up local area and maintain it until it finishes all operations. Thus we think the fine-grained version may be more efficient than the lock-free version since the contention may be lower.

The fine-grained insertion includes two parts:

1. search the position for insertion using hand-by-hand locking.
2. while the tree still needs to be restructured:
   (a) lock all the nodes involved in the restructuring.
   (b) restructure the tree.
   (c) unlock all the nodes that have been locked.

As mentioned in 5.1.2, the order of locking and unlocking is significant and may cause deadlock. Thus in this algorithm, all lock operations are performed in a top-down fashion. On the other hand, all unlock operations are performed in a down-top fashion to avoid deadlock.

[3] doesn't present an algorithm for remove operation. As mentioned in 5.1.8, deletion may cause deadlock or starvation since the operating node may go up. If two processes are going up at the same time and build their own local area for deletion, deadlock may occur. Thus we add "intention marker" and *SpacingRule* in lock-free version. But for the fine-grained version, it is hard to avoid deadlock with only locks. In our opinion, this is why there is no code or paper of fine-grained version red-black tree supporting deletion at present.

6 Results

We ran some tests on our serial version, our lock-free version as well as our fine-grained version. We used GHC machines for our tests, each of which has an eight-core, 3.2 GHz Intel Core i7 processor enabling Hyper-Threading Technology. Thus we used at most 16 threads in tests.

Aside from purely testing insertion or deletion, we want to know the performance in actual work, which contains not only insertion and deletion but also computations. So we let threads *sleep* for a short period after each operation to simulate computations and we got another set of test results.

Note that we used the one-thread version of lock-free algorithm as the serial version. We have compared the performance between our serial version and our lock-free version runs on one thread. And we found that they have almost the same performance, which means that the overhead of extra operations in lock-free version can be ignored. The only source of overhead is the contention between threads as some threads may need to wait until they get the flags they want.

6.1 comparing serial version and lock-free version

We compared the serial version and lock-free version and got some conclusions. The speedups of insertion and deletion of different sleep time are respectively shown in Figure 11 and Figure 13.

6.1.1 Both insertion and deletion of lock-free version gain linear speedup

From Figure 11 and Figure 13, we can see that the speedup remains linear and is sometimes slightly higher than the linear speedup. This means that our implementation is efficient.

6.1.2 Both insertion and deletion get more speedup for short computation time

Comparing data in Figure 12, we found that the speedup of 16 threads is higher than the one without sleep time when sleep time is less than or equal to 0.000001 seconds. The reason of this phenomenon may be that short sleep time hides the contention among threads and increases the efficiency.

The similar pattern can be found in Figure 14 and the effect is even better than insertion. From our aspect, deletion for 0.000001s sleep time has more obvious growth because of its higher contention than insertion. The local area of a deletion is larger and often moves up among the tree, which has higher probability to interfere with other threads. So when there is some sleep time, its contention can be completely hidden and it gains a better speedup than insertion.

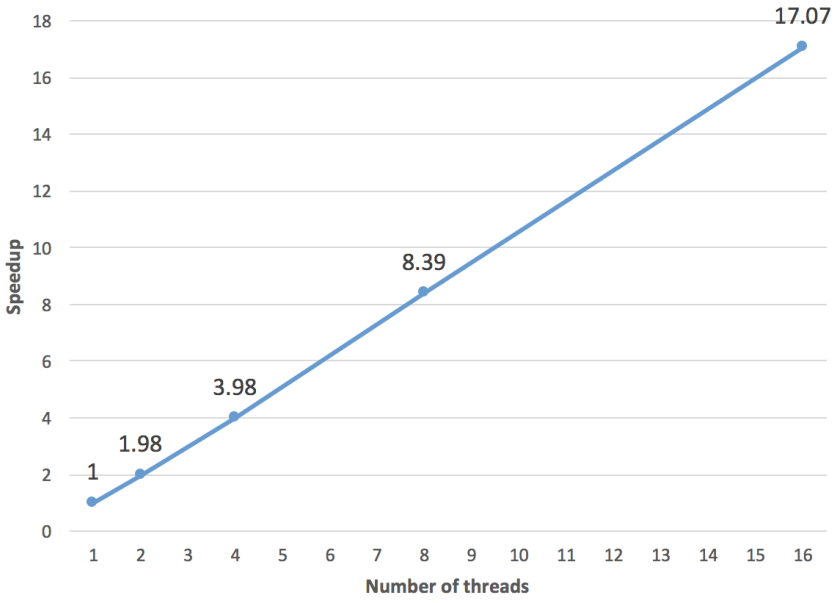


Figure 11: The speedup of 100,000 lock-free insertion(no sleep time)

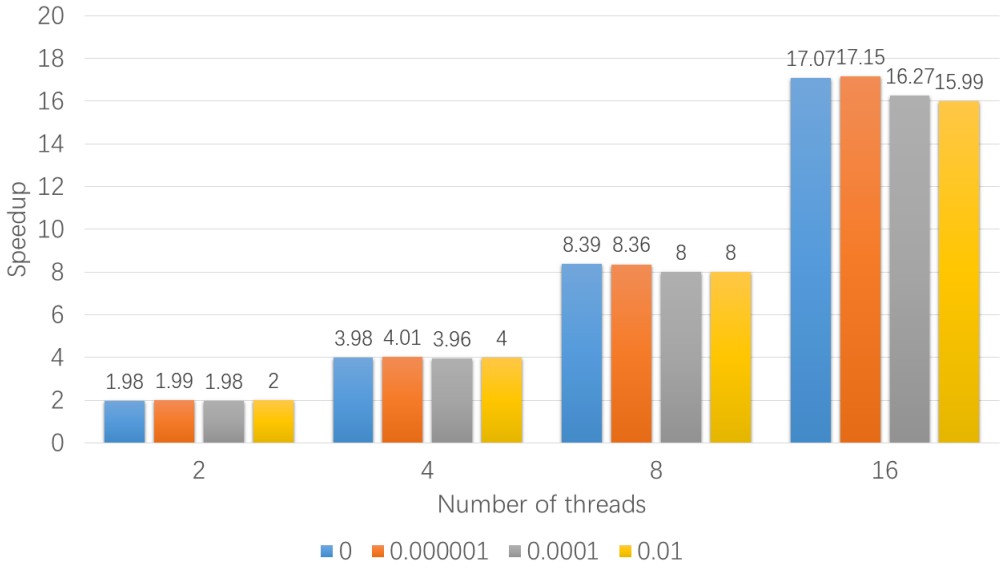


Figure 12: The speedup of 100,000 lock-free insertion for different sleep time(sec)

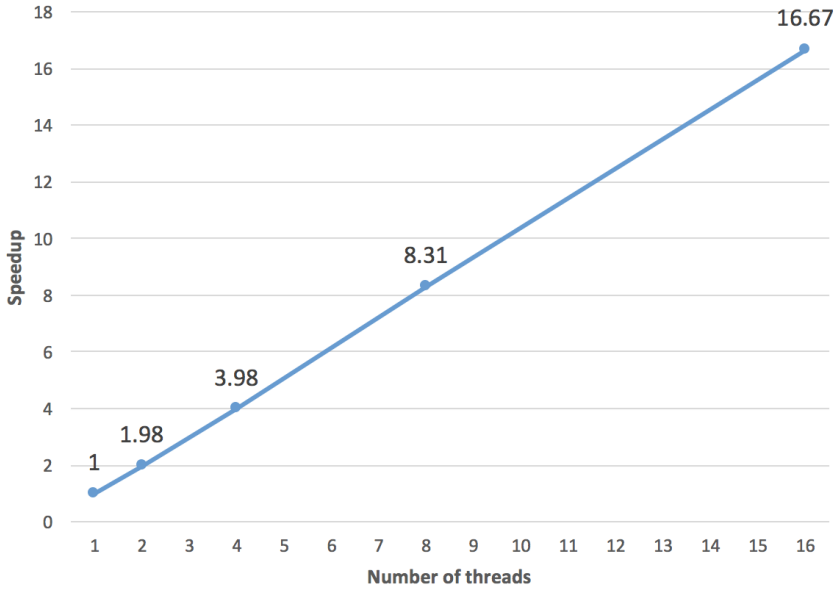


Figure 13: The speedup of 100,000 lock-free deletion(no sleep time)

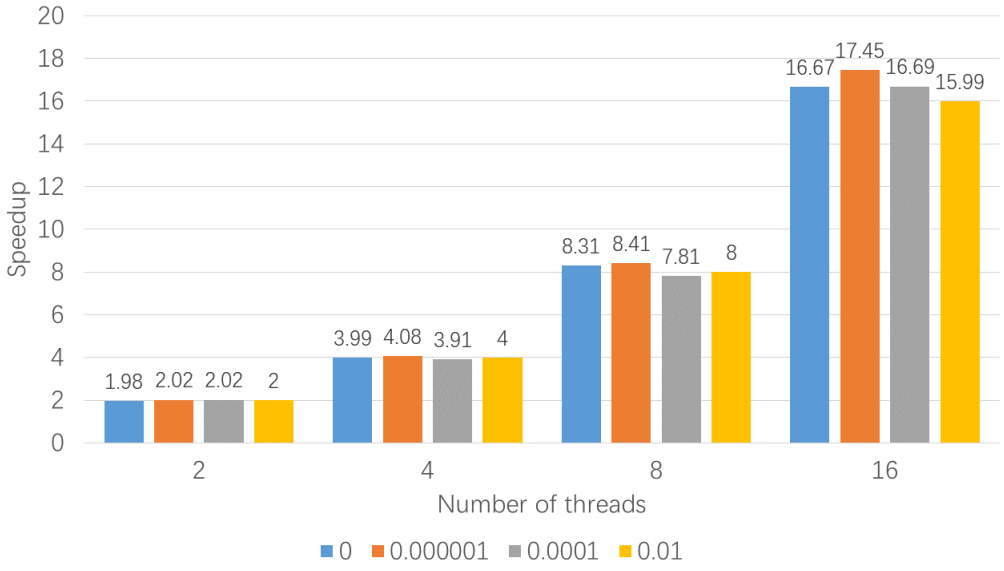


Figure 14: The speedup of 100,000 lock-free deletion for different sleep time(sec)

6.2 comparing lock-free version and fine-grained version

As for the comparison between lock-free version and fine-grained version, we could only compare the performance of insertion since we don't have an implementation of fine-grained deletion. The speedups of fine-grained version are shown in Figure 15.

6.2.1 Fine-grained insertion also has a linear speedup

From Figure 15, we can see that the speedup is linear as well. Sometimes it gets a superlinear speedup, which is similar to the lock-free version. This result proves the above statement that the fine-grained version is slightly more efficient than the lock-free version. Both two versions have similar mechanism to gain access of needed nodes while the "get-before-modify" fashion of the fine-grained version seems to be better than the "get-all-in-advance" fashion of the lock-free version.

6.2.2 Fine-grained insertion's execution time is slightly shorter than lock-free version for short sleep time

The results of execution time of both fine-grained version and lock-free version are shown in Figure 16 and Figure 17.

As we can see in Figure 16, the execution time of fine-grained version is shorter than the one of lock-free version for different thread numbers when there is no sleep time. But in Figure 17, the execution time of two version is almost the same. Our conjecture is that for short or no sleep time, the contention of the lock-free insertion cannot be hidden by sleep, which causes it slower than the fine-grained version. As the sleep time increases, it becomes able to completely hide the contention difference of two versions so the execution time is nearly the same.

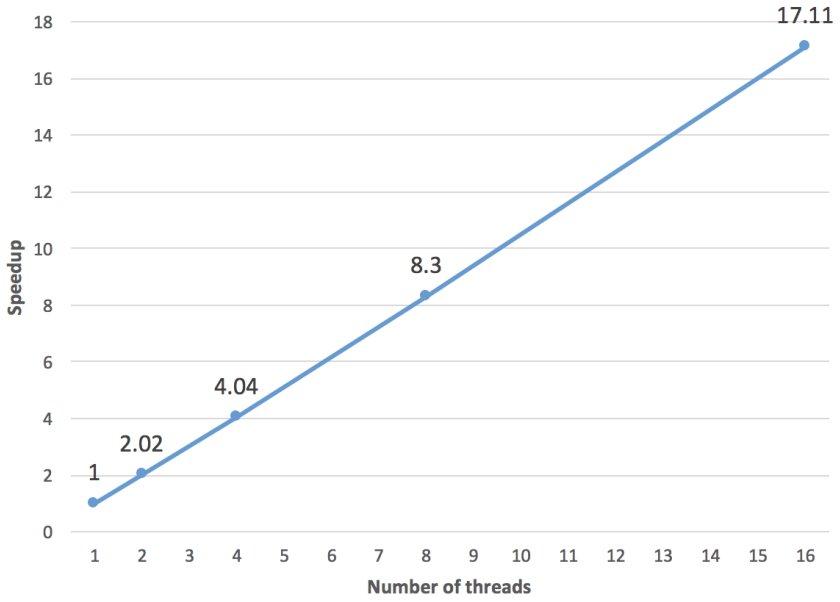


Figure 15: The speedup of 100,000 fine-grained insertion(no sleep time)

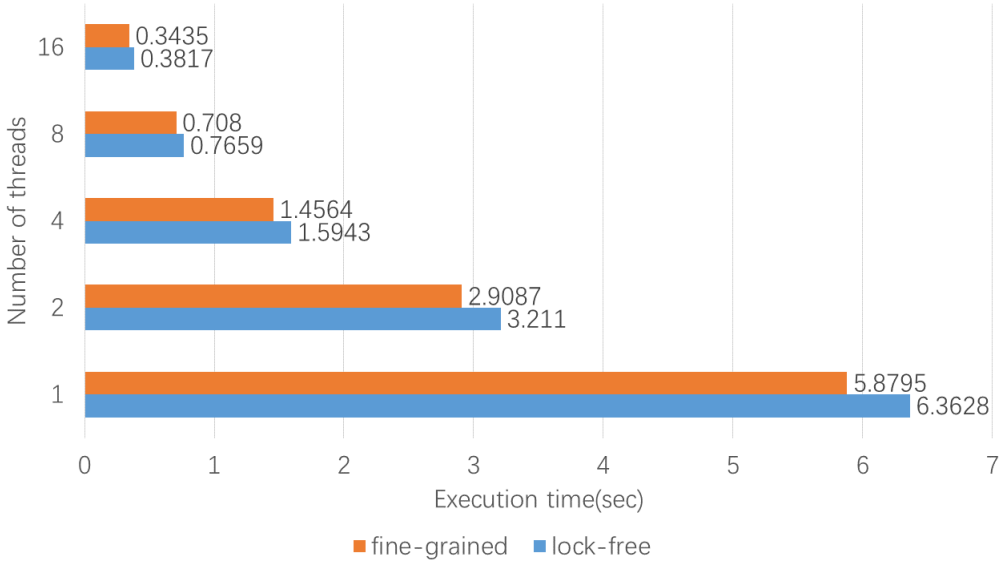


Figure 16: The execution time of 100,000 insertion(no sleep time)

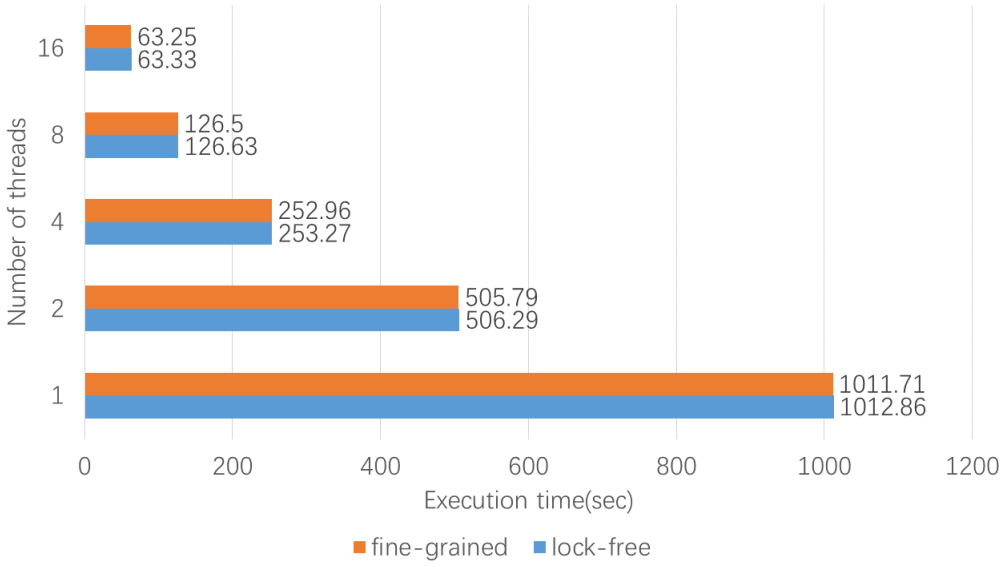


Figure 17: The execution time of 100,000 insertion(sleep time=0.01s)

7 Work Distribution

The work performed by each team member is listed in Table 2 and the overall distribution of the two team members is **50%-50%**.

Team members	**Main work**
Shun Zhang	1. implemented sequential red-black tree 2. implemented lock-free red-black tree (remove & search) 3. found the two rules unnecessary and conduct tests to confirm that 4. analyze results, write report and make poster
Guancheng Li	1. implemented lock-free red-black tree (insert) 2. implemented fine-grained red-black tree (insert) 3. do various experiments on the efficiency and scalability on the three versions 4. analyze results, write report and make poster

Table 2: Work distribution of the team members

References

[1] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to algorithms[M]. MIT press, 2009.

[2] Kim J H, Cameron H, Graham P. Lock-free red-black trees using cas[J]. Concurrency and Computation: Practice and experience, 2006: 1-40.

[3] van Breugel F. Concurrent Red-Black Trees[J]. Assignment, January, 2015.

[4] Ma J. Lock-Free Insertions on Red-Black Trees[J]. Master's thesis. The University of Manitoba, Canada (October 2003), 2003.

[5] Natarajan A, Savoie L H, Mittal N. Concurrent wait-free red black trees[C]//Symposium on Self-Stabilizing Systems. Springer, Cham, 2013: 45-60.