Article title: Re-implementation of Lock-free Contention Adapting SearchTrees

Authors: Kevin Boyles[1], Hong Bao[2], Benjamin Le Heup[3], Triet Nguyen[4], Vo Tam[5]

Affiliations: University of Central Florida[1]

Orcid ids: 0000-0002-1176-4562[1]

Contact e-mail: kboyles@knights.ucf.edu

License information: This work has been published open access under Creative Commons Attribution License <http://creativecommons.org/licenses/by/4.0/>, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. Conditions, terms of use and publishing policy can be found at <https://www.scienceopen.com/>.

Preprint statement: This article is a preprint and has not been peer-reviewed, under consideration and submitted to ScienceOpen Preprints for open peer review.

Links to data: <https://github.com/kboyles8/lock-free-search-tree>

DOI: 10.14293/S2199-1006.1.SOR-.PPVIJDB.v1

Preprint first posted online: 01 April 2020

Keywords: lock-free, contention adaption, tree, MRLock, multi-core programming, treap, coarse-grained locking, memory management

Re-implementation of Lock-free Contention Adapting Search Trees

Boyles, Kevin

University of Central Florida
kboyles@knights.ucf.edu

Hong, Bao

University of Central Florida
baohong@knights.ucf.edu

Le Heup, Benjamin

University of Central Florida
bleheup@knights.ucf.edu

Nguyen, Triet

University of Central Florida
anhtel94@knights.ucf.edu

Vo, Tam

University of Central Florida
tam.vo99@knights.ucf.edu

April 1, 2020

Abstract

To explore multi-core programming, we re-implement the Lock Free Contention Adapting Search Tree. We follow the structure of the original, using immutable treaps as leaf nodes implemented with an array for better performance with memory caching. Memory leaks are prevented through pre-allocation of elements. We evaluate the performance of the LFCA tree and compare it to the previous MRLock version. The LFCA tree performs better in all cases with multiple threads.

Contents

1	Summary of Data Structure	1
1.1	LFCA Tree	2
1.2	MRLock Tree	2
2	Key Methods	3
2.1	Insert and Remove (Update)	3
2.2	Lookup	4
2.3	Range Query	4
3	Contention Adaptation	6
3.1	High-Contention Adaptation	6
3.2	Low-Contention Adaptation	6
4	Treap Leaf Nodes	7
4.1	Treap Operations	7
5	Memory Management	9
6	Issues	9
7	Performance Evaluation	10

List of Figures

1	Structure of the LFCA tree	1
2	Structure of the immutable treap	8
3	200000 Operations (I:25% R:25% L:50%)	11
4	200000 Operations (I:10% R:10% L:80%)	12
5	200000 Operations (I:0.5% R:0.5% L:99%)	12
6	200000 Operations (I:10% R:10% L:55% RQ:25%-10)	13
7	200000 Operations (I:10% R:10% L:55% RQ:25%-1000)	14
8	200000 Operations (I:10% R:10% L:55% RQ:25%-100000)	14

1 Summary of Data Structure

The Lock-free Contention Adapting Search Tree (LFCA Search Tree) [1] is composed of route nodes and base nodes. Route nodes allow searching through the tree, similar to Binary Search Trees (BSTs). These nodes contain a value and a pointer to the node's left and right child. All elements within the left subtree are less than or equal to the route node's value, and all elements within the right subtree are greater than the route node's value. Base nodes store a reference to an immutable data structure (in our case, a treap), which stores the values contained in the LFCA tree. The treap will be explained further in Section 4. Figure 1 shows the structure of the LFCA tree. The LFCA tree our report is based on supports lock-free insert, remove, and range-query, as well as wait-free lookup.

Figure 1: Structure of the LFCA tree

1.1 LFCA Tree

Our re-implementation of the LFCA tree follows [1] closely. We performed the re-implementation by taking the pseudocode from the paper and translating it from the C-like notation into C++ code. Some modifications were made to the code to simplify and integrate it into our existing code. These modifications are outlined below.

- C utilities used in the original implementation, such as the custom stack, now use the C++ standard library variants
- The node structs are combined into a single struct for simplicity
- Our custom immutable treaps are used instead of the treaps used by the original LFCA trees. There are significant differences in these structures, which are covered in Section 6.
- Range Queries store results in vectors instead of treaps. This is due to the fixed size limitation of our treaps.
- High contention adaptations (splits) are forced when a treap has reached the maximum size. This is again due to the fixed size treaps.
- Search order has been modified so that left children of route nodes can contain all values less than *or equal to* the route node's value, as opposed to strictly less than. This is due to the way the treaps are split.
- All nodes, result sets, and treaps are preallocated, fixing all memory leaks. See Section 5 for more information.

1.2 MRLock Tree

Our original locking re-implementation of the LFCA tree is included and used as a comparison for performance testing in section 7. This simplified re-implementation provides thread-safety for all operations through coarse-grained locking using the Multi-Resource Lock (MRLock) library [2]. Only one method can be called at a time by a single thread, and all other threads must wait. This is done by creating a single resource which represents the entire tree. All methods must acquire this resource before executing. A

wrapper class was created for using MRLock which uses the programming technique Resource Acquisition Is Initialization (RAII) [3]. The initialization of the wrapper class acquires the lock and its destruction releases the lock. This simplifies the use of the lock and avoids issues with unlocking when branching is involved, since the lock will always be released when the method is exited.

Coarse-grained locking is simple to implement and verify but causes high contention on the tree due to all operations needing the same lock. This contention is reduced in the lock-free implementation. The locked version does not have any contention-adapting properties, as this is irrelevant when no two threads can be accessing the tree at once.

2 Key Methods

Our re-implementation supports the same methods as the original: Insert, Remove, Lookup, and Range-Query. Insert and Remove follow a nearly identical algorithm and will be described as Update.

2.1 Insert and Remove (Update)

The update operation is used to perform inserts and removes on the tree. The operation proceeds as follows:

1. The tree’s route nodes are traversed using binary search until a base node is found.
2. The treap’s size is checked. If the treap is full, a high contention adaptation is attempted on the node, and the update method restarts. The update method does not guarantee that the adaptation succeeds, so multiple attempts might be needed.
3. The node is checked to see if it is replaceable. A node is replaceable if it is not in the middle of a join operation or a Range Query.
 - If the node is not replaceable, the update method sets a flag to indicate contention and attempts to help the other operation in progress, before restarting.

4. A new node is created with the updated treap (with value inserted or removed) and statistic value.
5. An attempt is made to replace the existing node with the new node. A failure indicates that another thread has performed a conflicting operation on this node. The contention flag is set, any ongoing operation is helped, and the method restarts.
6. On success, a contention adaptation is attempted based on the statistics value. See Section 3 for information about adaptations.

This operation is lock-free, since if the update operation fails, it is always because another thread has made progress. The use of compare and swap ensures correctness, as a node is only replaced if there is no ongoing operation using it. The update operation linearizes once it succeeds in replacing the found node with the updated node.

2.2 Lookup

The lookup operation traverses the tree's route nodes using binary search until a base node is found, then determines whether the value is located in the base node's treap. This operation is wait-free and always executes at a finite amount of time, which is based off the size of the tree and treap of the base node. The method linearizes once a base node is found, as the node cannot change. Because of the immutability of the nodes, the method is always correct for some history which it can be linearized to.

2.3 Range Query

The Range Query operation proceeds as follows:

1. The first node in the Range Query (which contains the low value) is found, keeping track of all route nodes passed in a stack.
2. If the node is replaceable, a copy of the node is created with info needed to help complete the Range Query. This info includes the low and high values, as well as a pointer to the result set for the Range Query. This new node functions similarly to a descriptor object. If the node is not replaceable, the ongoing operation is helped before restarting the operation.

3. An attempt is made to replace the original node with the new node. If this fails, the operation restarts.
4. The remaining base nodes for the Range Query are found in order, until the base node containing the high value is located. This is the stopping point for the Range Query.
 - (a) If the found node contains a result, it must be the result for the current range query which was completed by another thread. This result is returned. If the node is part of the current range query but does not have a result, it is skipped, as it has already been considered.
 - (b) If the node is replaceable, it is replaced with a copy containing the Range Query information. If the replacement fails, it is attempted again.
 - (c) If the node is not replaceable, it means there is an ongoing operation on the node. The operation is helped, and the loop to find the next base node repeats.
5. Once all nodes for the Range Query are found, they are looped through, collecting all values within the range. These values are stored into the result set object, which is shared across all nodes in the query, allowing any other helpers to see that the operation has been completed.
6. The result of the query is returned.

The Range Query operation is lock-free, as any failure during the process is due to another thread making progress. By marking each node in order, the operation “freezes” all needed nodes, creating a snapshot of the tree at some point in time. A range query on this snapshot is correct, as it is linearizable to a specific sequential history which created it. The linearization point of the operation is when the results are stored into the result set, making them available to any other thread. The marking of each individual node could be considered a linearization point as well, since they announce the in-progress Range Query to other threads, which are forced to help before completing their operation.

3 Contention Adaptation

Contention adaptations are performed whenever the contention statistic of a node exceeds the low threshold or the high threshold. Operations which succeed without conflict from another operation are considered uncontended and lower the contention statistic. If there is a conflict, the operation becomes contended, and the statistic increases. If a node was part of a Range Query affecting more than one base node, the statistic value is also decreased. High-contention adaptations are performed when the contention becomes too great and aims to reduce the contention of a node. Low-contention adaptations are performed on nodes that have low contention in order to reduce the depth of the tree.

3.1 High-Contention Adaptation

High-contention adaptations are performed on nodes with high contention. A base node is split into two base nodes with roughly half of the elements of the original. The old base node is converted into a route node with a search value equal to the split value. Splitting nodes improves the performance of the LFCA tree by distributing the contention on the original node into two nodes instead.

3.2 Low-Contention Adaptation

Low-contention adaptations are performed on nodes with low contention. When a node is selected, it's closest sibling node is found. This neighboring node has values which are just greater than or just less than the values in the current node. The neighbor node is merged with the main node, and the tree is shifted to account for the change. This involves a two-step process which first marks the nodes for joining, and then completes the merge. The marking of the nodes can fail, which aborts the adaptation. The complete operation is always guaranteed to succeed, since any conflict in completing the join indicates that another thread completed the join operation instead. This optimizes the structure of the LFCA tree for performance on Range Queries, as the number of nodes that need to be considered are reduced.

4 Treap Leaf Nodes

The leaf nodes of our search tree are implemented using treaps. A treap is a data structure that combines the properties of binary search trees (BSTs) and heaps. BST ordering allows for fast lookups, while heap ordering is used to balance the tree. Nodes are given random weights which, on average, result in a treap that is balanced. The balanced treap has a depth of $O(\log n)$, which allows for $O(\log n)$ lookup. Our treaps have a maximum size of 64, which was originally thought to be the size of the treaps in [1].

All operations on the treap are immutable operations, meaning that the original treap is not modified. Instead, new treaps are created and returned, which contain the modifications.

An unusual property of these treaps is that they are implemented using an array. Rather than nodes being linked to one another by pointers, the nodes are linked using local array indices. By keeping the nodes nearby in memory, multiple accesses to the same treap will be likely to result in many cache hits. This increases the performance of lookups. Another benefit is that the treaps can easily be copied without needing to allocate and link every node. Only the contents of the node array need to be copied. This property is less important for the locked version of the LFCA tree, but will be very important for the lock-free version, as a copy of the treap will be made for every modification due to the immutability. Figure 2 provides a graphical representation of the structure of the treap.

4.1 Treap Operations

The following is a list of operations supported by the treaps. These operations were implemented based on pseudocode from [4].

- Lookup: Determines if a value is in the treap.
- Insert: Inserts a value into the treap. The value is inserted into the last free index in the node array.
- Remove: Removes a value from the treap. In order to restore the node array structure, the last node in the array is transferred into the location where the removed node was, adjusting any references to the node as needed. The transfer is a constant-time operation with little overhead.

Figure 2: Structure of the immutable treap

- Range-query: Returns all values in the treap within a range.
- Split: Splits the treap into two approximately equal-sized treaps. The treaps are split at the median of the contained values, giving a *left* and *right* treap. These contain all values less than or equal to the median, and all values greater than the median, respectively.
- Merge: Merges two treaps (a *left* and *right* treap) together into a new treap. The combined size of the two treaps must not be larger than the maximum size of a treap. All values in the *left* treap must be smaller than the values in the *right* treap.

The split and merge operations are performed using a special node called the *Control Node*. When splitting a treap, this node is inserted into the treap with the median value and pulled up to the root. The left and right children of the *Control Node* become the *left* and *right* treaps. When merging two treaps, the process is performed in reverse. The *Control Node* is added as

the root of the new treap, and its left and right children are the *left* and *right* treaps. The control node is moved down to become a leaf node, and cut off.

The process for moving nodes up and down and the details for maintaining BST and heap ordering when inserting and removing are common to the treap data structure, and will not be discussed.

5 Memory Management

Memory management in concurrent systems is difficult, and can lead to memory leaks or crashes. This is due to the difficulty of knowing when an object is safe to be deleted. At any moment, there may be other threads that are accessing the object. To avoid these issues in our testing, we preallocate all elements before beginning a test and free them after, when we are sure the nothing is still using them. To simplify this process, a Preallocatable class was designed.

This class is used to give any other class the ability to be preallocated. To use it, the other class simply needs to extend Preallocatable. Static methods are added to the child class allowing for preallocation, deallocation, and distribution of the preallocated elements. Distribution is made thread-safe through an atomic counter which is incremented every time an element is retrieved. This creates a bottleneck at the atomic access to that counter, but due to the infrequency of allocations compared to other work and the high cost of allocations in general, this is an improvement. This also avoids the need to pass around arrays of preallocated elements to each thread using them, or significantly modify the existing data structure.

The Preallocatable class is designed using the Curiously Recurring Template Pattern (CRTP) [5] to give the Preallocatable class the type of the base class. This allows the Preallocatable parent class to create and use instances of the base class.

6 Issues

Our re-implementation has several issues, which are covered here.

The largest issue was out misunderstanding of the immutable treaps used in the original LFCA tree. We designed a treap which is entirely a fixed size, assuming that the “leaf nodes” referred to were the treaps themselves.

However, the original implementation uses treaps that have arbitrary size, with the leaf nodes of those treaps being fixed size instead. This causes issues with how our treaps interacted with the contention adaptation. First, we are forced to split treaps that become max size, whereas the original treaps can continue growing. Second, it becomes very difficult for the contention statistic of a node to fall low enough to be considered for merging, since treaps are constantly needing to be split due to filling up. This reduces the efficiency of the original algorithm.

We left out one call to randomly adapt a single node involved in a Range Query. We did not want to introduce random number generation to the LFCA tree, and were not sure how useful it was. It may have played a role in helping to reduce contention on Range Queries across multiple base nodes.

Treaps still use random number generation. Performance evaluation of the data structure should not include random number generation, but this was not resolved. To fix this and get a better estimate of the performance of the concurrency, random values should be preallocated within the treap class and distributed, similar to how the treaps themselves are preallocated.

7 Performance Evaluation

Performance evaluation was performed on the re-implementation of LFCA trees as well as the MRLock version using a laptop with an 8-core Intel i7-8550U CPU. 200000 random operations were performed on an initially empty tree with varying weights using 1 to 32 threads for the LFCA tree, and 1 to 8 threads for the MRLock tree. It was not possible to perform evaluation of the MRLock tree with more than 8 threads due to a limitation of the library. The library limits the number of threads to the hardware concurrency of the machine, which was 8 in testing. Attempting to exceed this limit causes crashes and hangs. The weights used were the same weights used in the performance evaluation of [1]. Weights are denoted with an abbreviation for the operation, a colon, and the percentage of that operation. Operations are Insert (I), Remove (R), Lookup (L), and Range Query (RQ). For Range Queries, the size of the query is listed following a dash after the percentage.

Figures 3, 4, and 5 show performance evaluation for varying weights of single-element operations (Lookup, Remove, and Insert). In all cases, the LFCA tree outperforms the MRLock tree. For high amounts of lookups, the MRLock tree shows slight improvements with increasing threads, but

degrades after 2-3 threads due to high contention. The LFCA tree shows consistent performance increases with increasing threads, up to the 8 core limitation of the hardware. After that, performance flattens out with some fluctuation. With 99% lookup operations, performance flattens earlier, at around 4 threads.

Figure 3: 200000 Operations (I:25% R:25% L:50%)

Figure 4: 200000 Operations (I:10% R:10% L:80%)

Figure 5: 200000 Operations (I:0.5% R:0.5% L:99%)

Figures 6, 7, and 8 show performance evaluation which includes Range Queries. Operation weights remain constant with the size of the Range Queries varying. In all cases, MRLock trees outperform LFCA with a single thread. This is likely due to the overhead of the lock-free algorithm. From 2-3 threads, the LFCA tree shows significant performance improvements, with minor performance improvements up to 8 threads. After 8 threads, performance flattens out with some fluctuation. MRLock trees seem to have a slight performance increase for 2 threads, but quickly degrade in performance after that.

Figure 6: 200000 Operations (I:10% R:10% L:55% RQ:25%-10)

Figure 7: 200000 Operations (I:10% R:10% L:55% RQ:25%-1000)

Figure 8: 200000 Operations (I:10% R:10% L:55% RQ:25%-100000)

References

- [1] K. Winblad, K. Sagonas, and B. Jonsson, “Lock-free contention adapting search trees,” in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, Vienna, Austria: Association for Computing Machinery, 2018, pp. 121–132, ISBN: 9781450357999. DOI: 10.1145/3210377.3210413. [Online]. Available: <https://doi.org/10.1145/3210377.3210413>.
- [2] D. Zhang, B. Lynch, and D. Dechev, “Fast and scalable queue-based resource allocation lock on shared-memory multiprocessors,” in *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, ser. OPODIS 2013, Nice, France: Springer-Verlag, 2013, pp. 266–280, ISBN: 9783319038490. DOI: 10.1007/978-3-319-03850-6_19. [Online]. Available: https://doi.org/10.1007/978-3-319-03850-6_19.
- [3] T. Dalling. (May 17, 2012). Resource acquisition is initialisation (raii) explained, [Online]. Available: <https://www.tomdalling.com/blog/software-design/resource-acquisition-is-initialisation-raii-explained/> (visited on 03/22/2020).
- [4] Algorithm Tutor. (2019). Treaps (with implementation in c++), [Online]. Available: <https://algorithmtutor.com/Data-Structures/Tree/Treaps/> (visited on 03/22/2020).
- [5] E. Bendersky. (May 17, 2011). The curiously recurring template pattern in c++, [Online]. Available: <https://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c> (visited on 04/01/2020).