Article title: Re-implementation of Lock-free Contention Adapting Search Trees Using MRLock

Authors: Kevin Boyles[1], Bao Hong[2], Benjamin Le Heup[3], Triet Nguyen[4], Tam Vo[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Preprint statement: This article is a preprint and has not been peer-reviewed, under consideration and submitted to ScienceOpen Preprints for open peer review.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-.PPIV7TZ.v1

Preprint first posted online: 23 March 2020

Keywords: coarse-grained locking, multi-core programming, lock-free, MRLock, tree

Re-implementation of Lock-free Contention Adapting Search Trees Using MRLock

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

March 23, 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. The initial re-implementation uses coarse-grained locking with MRLock to provide thread-safety. This will be extended to be lock-free in a future publication.

Contents

1	Summary of Data Structure	1
2	Key Methods	2
2.1	Insert	2
2.2	Remove	3
2.3	Lookup	3
2.4	Range Query	3
3	Treap Leaf Nodes	3
3.1	Treap Operations	5

List of Figures

1	Structure of the LFCA tree	1
2	Structure of the immutable treap	4

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 3. 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

Our simplified re-implementation of the LFCA tree 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 provides high contention on the tree due to all operations needing the same lock. This contention will be reduced when the tree is converted into a lock-free version in the future.

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. It also only allows storing integers in the tree, but could be extended to store arbitrary data.

2 Key Methods

Our re-implementation supports the same methods as the original: Insert, Remove, Range-Query, and Lookup.

2.1 Insert

The insert operation traverses the route nodes using binary search until a base node is found. The value is inserted into the treap the base node is pointing to.

After adding the value to the treap, the size of the treap is checked. If the treap's size is above a threshold (≥ 64 in our case), the treap is split in two. The base node is converted into a route node whose value is the splitting point of the treap. The left and right children of the new route node are the left and right side of the split treap, respectively.

2.2 Remove

The remove operation traverses the route nodes using binary search until a base node is found. The value is removed from the treap the base node is pointing to, if it exists.

After potentially removing the value from the treap, a check is made to determine whether to perform a merge. If the base node's sibling (the other side of the base node's parent route node) is also a base node, then the size of the two treaps is checked. If the combined size of the two treaps is below a threshold (< 32 in our case), then the two treaps are merged together. The parent route node is converted into a base node which points to the new treap.

2.3 Lookup

The lookup operation traverses the route nodes using binary search until a base node is found, then determines whether the value is located in the base node's treap.

2.4 Range Query

The range-query operation traverses the route nodes using a depth-first search to locate all base nodes that may contain values within the range. For each of these nodes, the referenced treaps are searched for all values within the range. These values are combined into a single vector and returned.

3 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 the size used 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. This is not necessary for the locked version of the LFCA tree, but will be required for a lock-free implementation.

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.

Figure 2: Structure of the immutable treap

3.1 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.
- **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.

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).