15-618 Project Proposal
Concurrent Binary Search Trees

Yifan Cao (Andrew ID: yifanca2), Yahui Liang (Andrew ID: yahuil)

November 2021

1 Project Web Page URL

`https://yahuiliang.github.io/15618-yahuiyifan-team/`

2 Summary

We want to implement 3 - 4 concurrent version of binary search tree (BST) data structures, including BST protected by single coarse-grained lock, fine-grained lock protected BST, lock free version BST and a transactional memory version BST if we have time. We will do experiments and measure the performance of each version on different workloads and concurrent thread count to analyze the pros and cons of each implementation and compare their performance.

3 Background

A binary search tree (BST) is a data structure which allows data to be searched, find, and delete with a time complexity O(log(n)). In a BST, the most important property is that a parent node will always have key larger than the key of its left sub-tree and smaller than the key of its right sub-tree. Three operations are supported in BST, insertion, deletion, and searching. When inserting, the location where the new node will be inserted need to guarantee that the property of BST still holds. After deletion, the BST may need to be adjusted to make sure the property still holds. In BST, data can be stored with hierarchies so that the modification to one sub-tree would not affect other sub-trees, though the structure of current sub-tree may be changed.

Binary search trees have lots of applications. Sorting can be performed using a binary search tree by inserting all elements and then perform in-order traversal. Another important application is database, which can use B-tree to sort data. Although not exactly using BST, B-tree is just a generalization of BST allowing each parent node having more than two child nodes. We want to implement concurrent BST because as long as BST supports concurrency, the applications of BST can explore parallelism, so the potential benefits can be large. We want to compare different implementations of concurrent BST to know what is the most efficient implementation or which implementation is the most suitable for a certain workload pattern so that the applications of BST can choose the best implementation to speedup more when parallelized.

4 Challenge

One of the most important problems to be solved when using concurrent data structures is how to ensure the data integrity when multiple threads are trying to access or modify the same part of the data structure. Solving this problem can be challenging in binary search tree since deleting a node will not only impact the node itself, its parent node and its children node. The structure of the sub-tree containing that node may be affected since in binary search tree since we need to maintain the correct order between a parent node and its left and right child node. When using different methods to ensure the data integrity, efficiency is also needed to be considered and

can be another challenging part. The overhead of operations used to maintain data integrity must not over-shade the speedup brought by parallelization. Potential problems brought by locks and concurrency, such as deadlock, must be dealt with, which may also be challenging.

Another possible challenging part in our project is how to generate different workloads for experiments. It is likely that a certain version of concurrent BST perform better than other versions when using a particular workload pattern (e.g. more inserts than deletes). If we want to explore the pros and cons of each version, we may need to come up with a few workload patterns, which may be challenging.

5 Resources

We found three papers relate to fine-grained lock protected BST, lock free version BST, and transaction memory BST.

Kung, H. T., & Lehman, P. L. (1979). Concurrent manipulation of binary search trees . Carnegie-Mellon University, Dept. of Computer Science.

Natarajan, A., & Mittal, N. (2014). Fast concurrent lock-free binary search trees. SIGPLAN Notices, 49(8), 317–328. https://doi.org/10.1145/2692916.2555256

Bronson, N. G., Casper, J., Chafi, H., & Olukotun, K. (2010). A practical concurrent binary search tree. SIGPLAN Notices, 45(5), 257–268. https://doi.org/10.1145/1837853.1693488

We will use GHC machines and PSC machine. We plan to implement different versions of BST and verify their correctness on GHC machines. To test their performance when using different number of threads, we may need to use PSC machine as it has more cores. Tests on how different workload affects the performance of different versions of BST will probably be carried on GHC machines as well.

6 Goals and Deliverables

6.1 Plan to achieve

- Implement the tree with coarse-grained lock supported
- Implement the tree with fine-grained lock supported
- Implement the tree without locking mechanism
- Implement the tree with software transaction memory supported
- Generate different insertion and deletion workloads
- Verify the correctness of three implementations
- Carry out experiments on different implementations of BST with different thread count and workloads
- Analyze performance, pros and cons of each BST implementation

In order to verify the correctness, we want our data to be consistent before and after a group of manipulations. Also, we do not want deadlocks appear in our implementation, we would have a timeout mechanism to detect whether deadlock happens. All details about these concurrent trees have been specified in papers that we found. Therefore, we are confident that we can implement them.

6.2 Hope to achieve

If we are on the right track, and finish above features earlier, we would like to add

- fine-grained tree balancing
- lock-free tree balancing
- STM tree balancing

However, if we did not have enough time to implement all 4 trees, we want first 3 of 4 tree implementations to be done, i.e. BST protected by coarse-grained lock, fine-grained lock, and lock-free BST.

6.3 What to show

We plan to show several speedup graphs during our post session. The speedup graphs will be about Speedup of different BST implementations vs. thread count when using different workloads. There will be one graph for each workload. These speedup graphs can help people know which BST performs best under different scenarios.

6.4 Expectations

Either fine-grained concurrent BST, lock-free BST, and STM BST implementations should perform better in speedup than coarse-grained concurrent BST, as well as the sequential version of BST manipulations. Different insertion, deletion, searching workloads may lead to different implementations outperforming others.

7 Platform Choice

We are going to use C++ to implement our data structures. Our team members are more experienced with using C/C++ rather than some high-level languages like Java, Python, and etc. Also, C++ allows us to manage memory on our own, and this is usefull when the paper explicitly specifies an garbage collection mechanism.

8 Schedule

Week	Date	Tasks
1	11.1 - 11.7	Proposal Literature review Coarse-grained lock version BST
2	11.8 - 11.14	Fine-grained lock version BST Lock-free version BST
3	11.15 - 11.21	Lock-free version BST Transactional memory version BST Milestone report
4	11.22 - 11.28	Transactional memory version BST Experiment workload generation
5	11.29 - 12.5	Experiments and analysis
6	12.6 - 12.10	Final report Poster