15-618 Final Report
Concurrent Binary Search Trees

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

December 2021

1 Project Web Page URL

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

2 Summary

We implemented 3 concurrent versions of binary search tree (BST) data structure, including single coarse-grained lock protected BST, fine-grained lock protected BST, and lockfree BST. We compared the performance of different concurrent BSTs when running different operation pattern with different tree size and thread numbers.

3 Background

A binary search tree (BST) is a data structure which allows data to be searched, inserted, and deleted 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. 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. As long as BST supports concurrency, the applications of BST can explore parallelism and speed up.

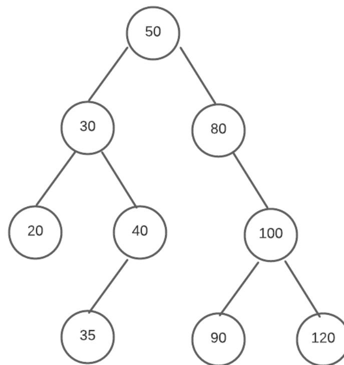

Figure 1: An example of a binary search tree.

In this project, we focused on find, insert, and erase three operations of the tree. We mainly want to parallelize the operations on a binary search tree, making it possible for multiple threads to operate on one tree concurrently without corrupting the tree. Dependencies occur when the program tries to read and modify same node connections. Therefore, we would have some synchronizations when one edge gets manipulated by multiple threads. Also, in concurrent BST, memory management can be a complex problem. Nodes cannot be freed immediately when they are removed from the tree since other threads may be accessing the same nodes. Therefore, we came up an idea on using Read/Write lock to implement the garbage collection mechanism.

4 Approach

C++ is used to implement solutions. STL mutex and atomic variables are used to ensure code is executed correctly in critical sections. Trees are implemented as template classes, and they are able to support all customized structures. But for the simplification, some of our operations only focused on integers and trees are not guaranteed can be used on types besides integers. We define the correctness of concurrent BST as insertion/deletion should not affect the traversal of other nodes in the tree.

4.1 Coarse Grained BST

This is the most basic BST. Find, insert, and erase are synchronized by using the single mutex. No two operations can be executed at the same time.

4.2 Fine Grained BST

The algorithm is implemented based on [1] in the reference section.

4.2.1 Find

The traversal is similar with the general BST. However, once the target is found, the algorithm checks whether the target is still the child of the parent node before it gets returned to avoid returning the node which has been slipped away. If this condition is met, the traversal will start from the parent of the child to find for the target again. Once the target is found, the parent node is locked to avoid other processes change the connection between the parent and the target node. If the target is not found, it returns the parent and the direction where the target should be appended to. Two routines are implemented for this step. One is a helper function which acts as the dependency for insert and erase functions. Another is the public find function for checking whether the target exists in the tree, and this functions will call the helper function to retrieve the result, and it will be responsible for releasing the lock of the node to allow other operations to make modifications to it.

4.2.2 Insert

Insert relies on find. The routine calls find to retrieve the parent node and the direction where the new node should be inserted to. At this point, the parent is locked and wait for modifications. Then the algorithm simply connect the new node to the parent and release the lock.

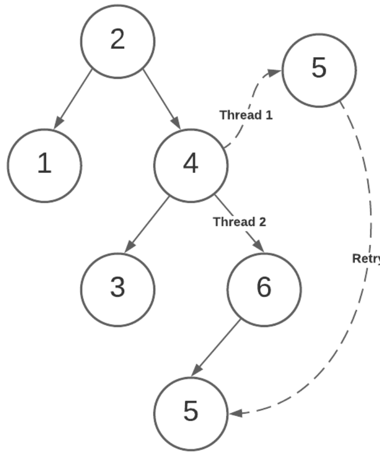

Figure 2: Fine Grained BST Insert example

4.2.3 Rotation

The rotation operation is needed to perform the deletion. While the rotation is performed, the original structure is not modified. Instead, the algorithm makes copies of rotated nodes and inject these nodes after the rotation into the tree so that other operations can still traverse in the right order before the rotation operation is done. While the rotation is being performed, edges which need to be changed are synchronized by using locks.

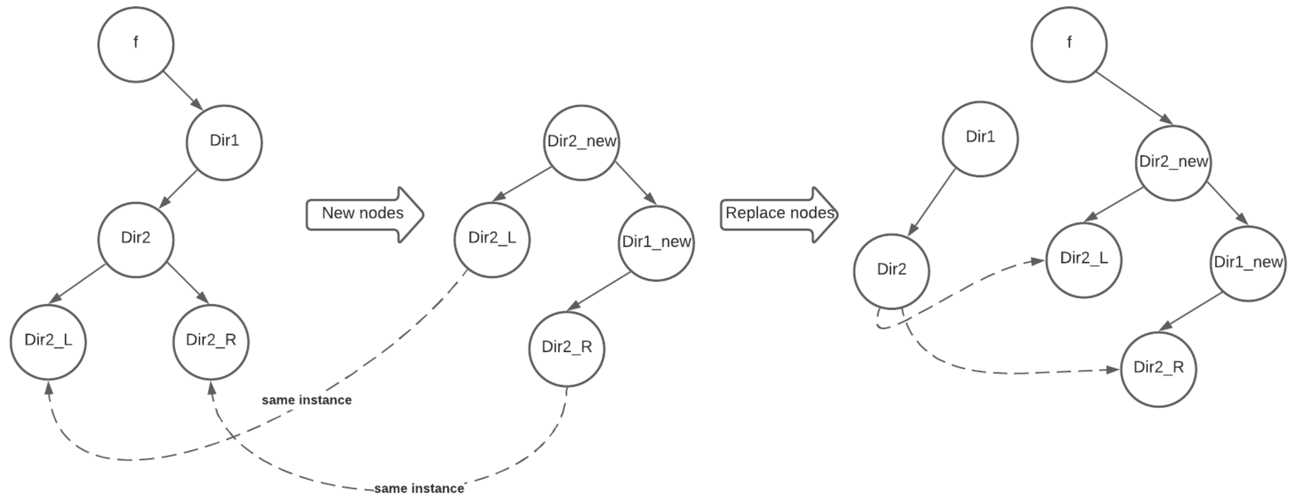

Figure 3: Fine Grained BST Rotation example

4.2.4 Erase

Erase relies on the find and rotation operations. The algorithm first use find to locate the node. Then it tries to rotate the node which needs to be removed until the node have only one incoming edge and one outgoing edge. Then the node's parent will be reconnected with node's child to make the node disappear from the tree. Modifications to the parent node will be synchronized using the lock. Since it could be possible that other operations are trying to access the erased node, we could not free this node immediately. Instead, we would mark the node as freed, and have a back pointer to point to the original parent of the node and make other operations

to resume from that point. And the node which needs to be freed will be pushed to the retire list which is local to each thread.

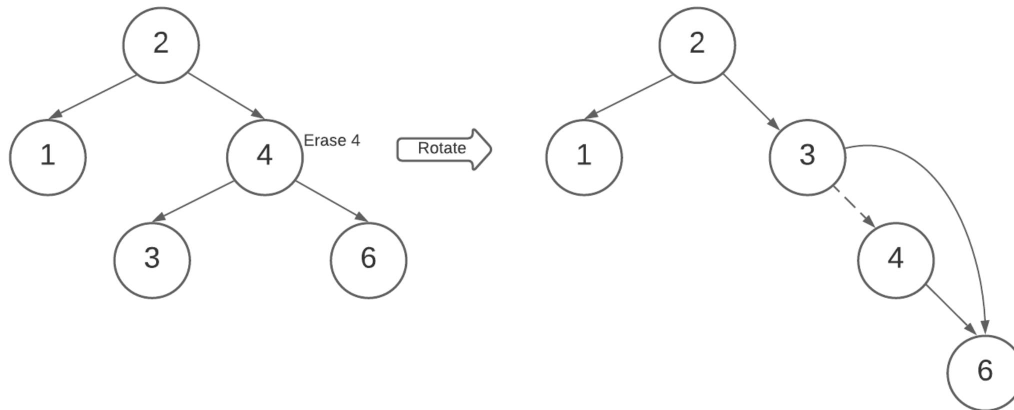

Figure 4: Fine Grained BST Erase example

4.3 Lock Free BST

The algorithm is implemented based on [2] in the reference section. Lock free tree gets rid of locks. Instead, it uses C++ atomic class to ensure edge modifications are atomic. One property of this tree is that all data are stored on leaves. Two bits are taken from node address to indicate whether the edge between parent and child is being erased.

4.3.1 Seek

The algorithm traverses the tree until it encounters a leaf which holds the same value as the key. If the key does not exist, the seek function returns the leaf whose key is most likely to match with the given key.

4.3.2 Find

The function simply check whether seek function returns a node whose key is same as the given key.

4.3.3 Insert

Insert operation first calls seek to find the location where the node should be inserted to. If the key has existed, nothing will be done. If the key does not exist, two new nodes will be created in this step. One node is the internal node, another node will be the leaf which stores newly added key. The reason to have the internal node is to make the actual data store on the leaf. The key of the internal node will be the maximum among the old leaf key and the new key. Then the old leaf and the new leaf will be inserted to left and right of the internal node based on their key values.

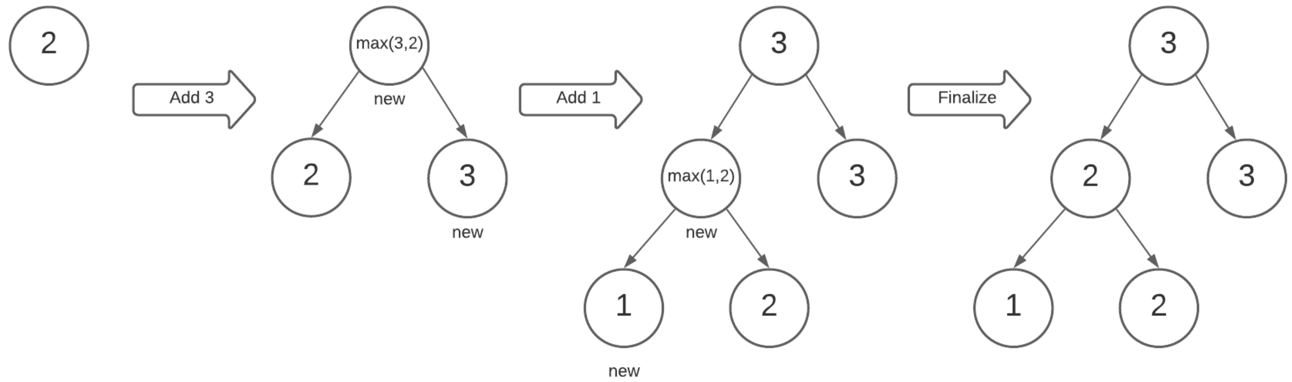

Figure 5: Lock Free BST Insert example

4.3.4 Erase

As oppose to insert operation, the erase operation removes two nodes for each call. One node is the internal node, and another node is the leaf node which stores the key. In this case, we call the leaf node which needs to be removed as child, and the other child of the internal node will be called as sibling. "Flag" bit is used for marking the edge between internal node and the child is being modified. "Tag" bit is used for marking the edge between internal node and the sibling is being modified. Erase operation simply reconnect the parent of the internal node to the sibling. Therefore, the internal node and the leaf is isolated from the tree.

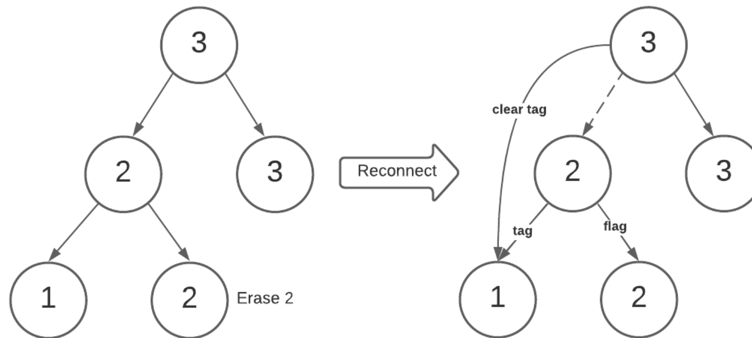

Figure 6: Lock Free BST Erase example

4.4 Garbage Collection

Nodes which are being accessed by other operations cannot be freed immediately. Instead, we have a local retire list for each thread. When one node needs to be freed, it gets pushed onto this list. When the size of the list exceeds a threshold value, the garbage collection routing will wait until current read and write operations complete and block all later operations get executed. Once the previous reads/writes finish, GC starts executing. It would try to free all nodes in the local retire list. Once GC completes its work, it gives away the permission and let later operations keep executing their works. By having an global lock and a read/write counter, this mechanism can be implemented easily.

5 Results

5.1 Experiment Setup

We conducted experiments on Pittsburgh Supercomputing Cluster (PSC) under shared memory mode. The technical specifications are listed below:

- CPU: 2x AMD EPYC 7742 (2.25-3.40 GHz, 2x64 cores per node)
- RAM: 256 GB
- Cache: 256 MB L3 cache, 8 memory channels
- Local Storage: 3.84 TB NVMe SSD

To compare the performance of each BST implementation, we measured the time each BST implementation needed to finish the program under the same experiment setting. To compare the performance of different versions of BST under difference circumstances, we conducted experiments with the following three parameters:

- Number of threads. We conducted experiments using 1, 2, 4, 8, 16, 64, 128, and 256 threads. The number of threads used is related to the degree of parallelism and also the degree of contention [2].
- Operation patterns. We carried out experiments using 7 different operation patterns, as listed below:
  1. Pure insert. Only insert operations are performed and timed.
  2. Pure erase. After inserting some data, then only erase operations are performed and timed.
  3. Pure search. After inserting some data, then only search operations are performed and timed.
  4. Contention simulated. Multiple threads will try to perform insert, erase, and search operations on same nodes concurrently. All operations are timed.
  5. Write dominance. 50% operations are insert operations and 50% operations are erase operations. Both operations are modifying the tree.
  6. Mixed. 20% operations are insert operations, 20% operations are erase operations, and the other 60% operations are search operation. The pattern is a most likely pattern in daily use of BST.
  7. Read dominance. 10% operations are insert operations and 90% operations are search. This pattern mainly reads the tree instead of modifying the tree.

  The last three patterns are developed from the idea in [2].
- Problem size. The problem size can be an important factor in parallelizing problem. In our experiment, we use 25,000, 50,000, 75,000, and 1,000,000 tree nodes to explore the impact of problem size.

5.2 Experiment Results

We timed each experiment, and plot bar plots of time to compare the performance of each implementation of binary search tree. We selected results of using 1 thread, 64 threads and 256 threads to be presented here. Results using other thread numbers are similar to result of 64 threads and 256 threads.

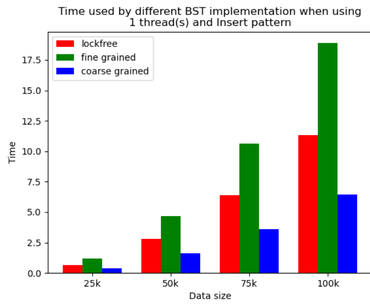

Figure 7: Time used by different BST implementations when using 1 thread and pure insert pattern

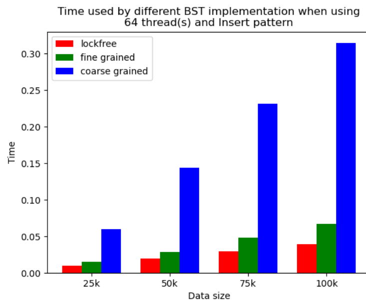

Figure 8: Time used by different BST implementations when using 64 threads and pure insert pattern

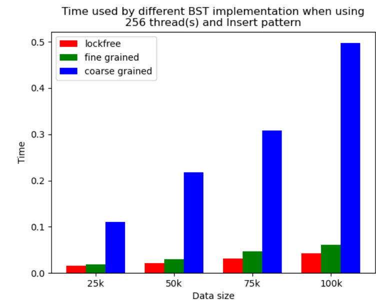

Figure 9: Time used by different BST implementations when using 256 threads and pure insert pattern

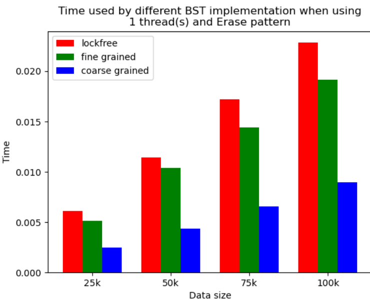

Figure 10: Time used by different BST implementations when using 1 thread and pure erase pattern

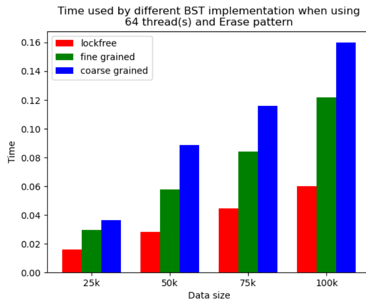

Figure 11: Time used by different BST implementations when using 64 threads and pure erase pattern

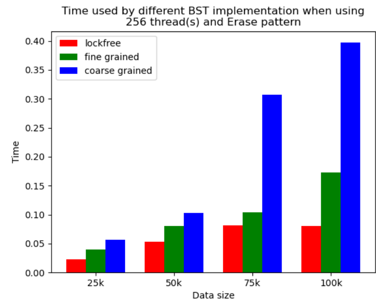

Figure 12: Time used by different BST implementations when using 256 threads and pure erase pattern

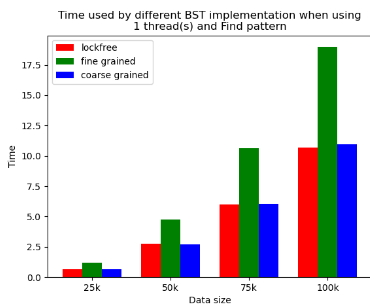

Figure 13: Time used by different BST implementations when using 1 thread and pure search pattern

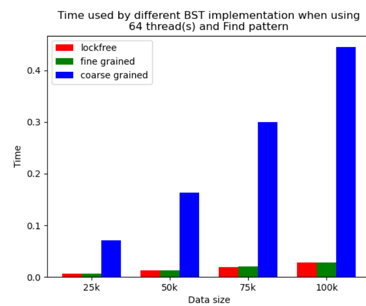

Figure 14: Time used by different BST implementations when using 64 threads and pure search pattern

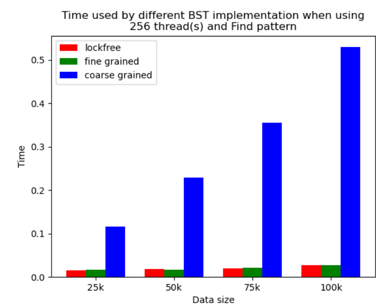

Figure 15: Time used by different BST implementations when using 256 threads and pure search pattern

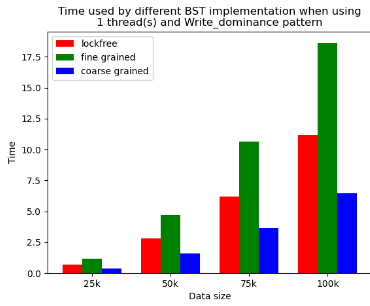

Figure 16: Time used by different BST implementations when using 1 thread and write dominance pattern

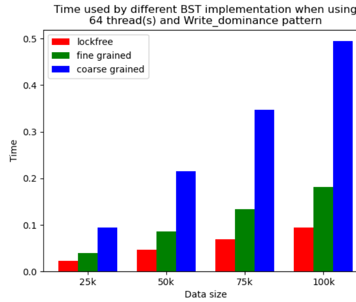

Figure 17: Time used by different BST implementations when using 64 threads and write dominance pattern

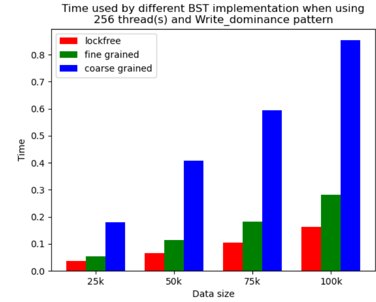

Figure 18: Time used by different BST implementations when using 256 threads and write dominance pattern

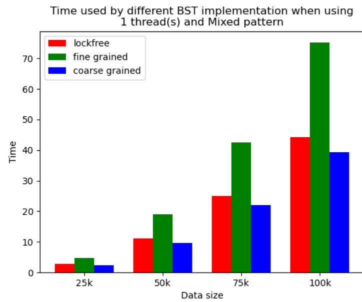

Figure 19: Time used by different BST implementations when using 1 thread and mixed pattern

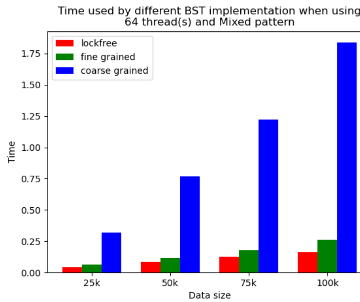

Figure 20: Time used by different BST implementations when using 64 threads and mixed pattern

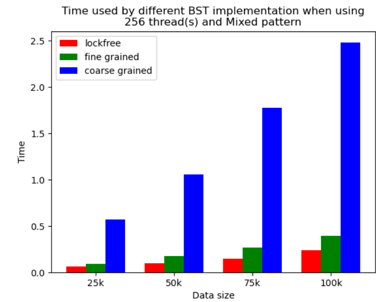

Figure 21: Time used by different BST implementations when using 256 threads and mixed pattern

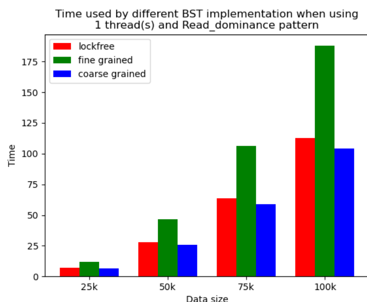

Figure 22: Time used by different BST implementations when using 1 thread and read dominance pattern

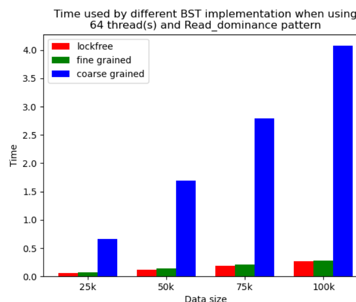

Figure 23: Time used by different BST implementations when using 64 threads and read dominance pattern

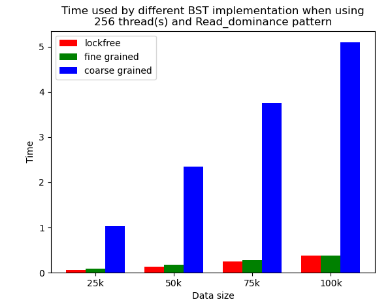

Figure 24: Time used by different BST implementations when using 256 threads and read dominance pattern

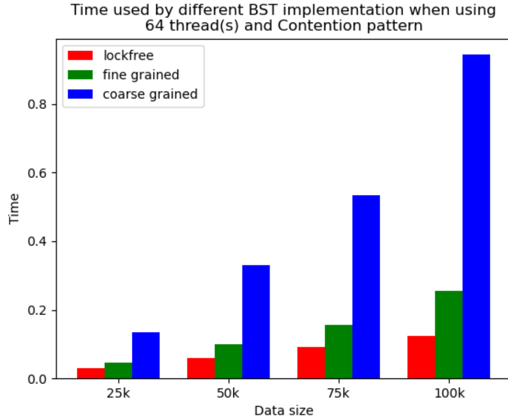

Figure 25: Time used by different BST implementations when using 64 threads and contention simulated pattern

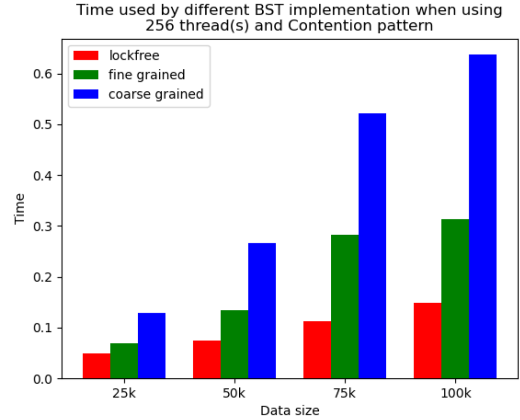

Figure 26: Time used by different BST implementations when using 256 threads and contention simulated pattern

5.3 Conclusion and Analysis

After carrying out the experiments, we draw several conclusions and analyze the reason.

5.3.1 Single thread performance

The performance comparison plot when using single thread is shown in the first column of plots in Section 5.2. From the plots, we can see that when there is only 1 thread, the coarse-grained lock protected BST is the fastest, and then lockfree BST is faster than fine-grained lock protected BST in most operation patterns except for the pure erase pattern.

The reason why coarse-grained lock protected BST is the fastest is that when there is only one thread, every operation is serialized naturally. In this case, coarse-grained lock has smallest overhead since it only locks at the beginning of each operation and release the lock after each operation. Fine-grained lock protected BST involves more locking and unlocking operations since it will lock the nodes that are being modified, and there will be more than one pair of locking and unlocking per operation. Therefore, it has the largest overhead. The exception is that lockfree BST is a little bit slower than fine-grained lock protected BST in the pure erase pattern. The possible reason is that the implementation of lockfree BST also involves a lot of flagging and tagging edges and compare-and-swapping, so the overhead can also be very large.

5.3.2 Multi-thread performance

When the number of threads is larger than 1, the general trend is that lockfree BST is faster than fine-grained lock protected BST, and fine-grained lock protected BST is faster than coarse-grained lock protected BST, which meets our expectation.

It is clear that coarse-grained lock protected BST is the slowest because it does not allow two operations to access the tree concurrently, so when using coarse-grained lock protected BST, basically all the operations are serialized. Therefore, it will be very slow and actually does not benefit a lot from parallelism. In fine-grained lock protected BST, every node has its own lock, and only nodes that are currently being modified will be locked, so it is truly benefiting from parallelism. However, fine-grained locks bring quite large overhead due to frequent locking and unlocking. Lockfree implementation is avoiding the large overhead by using compare-and-swap instructions before modifying an edge in the tree but not locking all nodes accessed, so the overhead will be smaller while still benefiting from parallelism, and thus lockfree implementation is the fastest.

The difference between coarse-grained lock protected BST and other two types of BST is very large when performing pure search, mixed, and read dominance pattern operations. The reason is that in these three patterns,

reading the tree is the dominating operation but not modifying the tree, so fine-grained lock protected BST and lockfree BST is experiencing relatively small overhead, and there will be much less time spent on garbage cleaning. When erase operations are rare, the retire list will be filled up more slowly, and garbage collection happens less frequently. When garbage collecting, the operations on tree will be temporarily blocked, and longer the total execution time. Therefore, less garbage collection means even faster execution. Therefore, in these three operation patterns, fine-grained lock protected BST and lockfree BST will be overwhelmingly faster.

As opposed to pure search, mixed, and read dominance pattern operations, other patterns involve more writing operations, including inserting and erasing. As explained above, in these situations, there will be more lock overhead and/or more garbage collection. Although fine-grained lock protected BST and lockfree BST are still much faster than coarse-grained lock protected BST, the time difference is smaller.

5.3.3 Contention

The time plots for the contention simulated pattern when using 64 threads and 256 threads are shown in Fig. 25 and Fig. 26. From the figures, we can see that lockfree BST is still the fastest, followed by fine-grained lock protected BST, and coarse-grained lock protected BST. However, comparing with experiment result in Fig. 20 and Fig. 21, we can see that the time difference is obviously smaller. The operation pattern of the mixed pattern and contention simulated pattern are very similar. The difference is that in contention simulated pattern, we intentionally let multiple threads accessing or modifying same set of tree nodes, but in mixed pattern, each thread will access or modify different tree nodes. Although in the setting of binary tree, contention can still occur even when threads are accessing different set of nodes as they may share common parents, but the chance will be much smaller. We can see that contention does impact the performance. Coarse-grained lock BST is impacted the least as it is suffering from contention all the time. The other two types are impacted more as their performance are closer to the performance of coarse-grained lock BST, but they are still faster.

6 References

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

[2] 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

7 Work Distribution

Yahui Liang: 50%

- Coarse-grained lock protected BST implementation
- Fine-grained lock protected BST insert operation implementation
- Lockfree BST erase operation implementation
- Garbage collection implementation
- Pattern generator (Write-only, Read-only, Contention)
- Testing and experiments
- Report

Yifan Cao: 50%

- Fine-grained lock protected BST erase operation implementation
- Lockfree BST insert operation implementation

- Pattern generator (Write-dominance, Read-dominance, Mixed)
- Automatic simulating script
- Testing and experiments
- Result plotting and analysis
- Report