♦ Abstract

Binary Search Tree is an efficient data structure where search, insertion and deletion takes O(logn) time complexity. The project implements different versions of concurrency in a BST using three different locking mechanisms.
Depth First traversal of any tree is of O(n) time complexity. Use of multiple threads for a traversal can make the traversal faster. The proposed algorithm traverses the tree using multiple threads for different subtrees of a binary tree. The individual traversals from different threads are joined to produce a pre-order traversal of the tree.

♦ Concurrency patterns

ReadWriteLock BST: This implementation uses a pair of associated locks for read-only operations and write operations. Ideal situation to use a ReadWriteLock is when the reads are frequent compared to the writes.

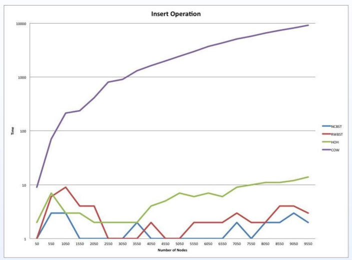
Hand-Over-Hand BST: With fine-grained locking, multiple locks are used in a set sequence to lock the smallest possible part of the BST that the current thread needs to operate on. This gives other threads the opportunity to work in parallel on other parts of the tree.

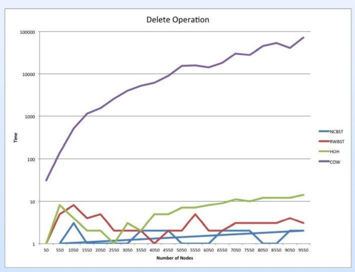
Copy-on-write BST: This creates a separate (private) copy of the BST and redirects the task to making changes to the private copy to prevent its changes from becoming visible to all the other tasks.

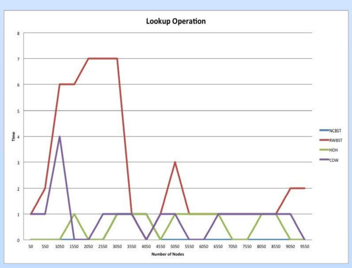
♦ Concurrent Read-Writes

In the implementation of *RWBST*, read-write-locks ensure parallel reads by multiple threads but not parallel write or parallel read-and-write by multiple threads.
In *COWBST*, copy-on-write allows parallel read-and-write, where a stale value of BST is read. However, a parallel write is not allowed. In *HOHBST*, hand-over-hand locking ensures parallel read as well as parallel write operation on a single BST by multiple threads.

♦ Results: Concurrent BST







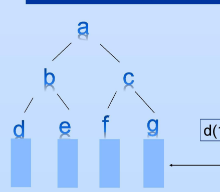
♦ Proposed Algorithm: Parallel DFT

```
Cardiality := numProcs
childQ.enqueue(node)
LOOP FOR node.cardinality != 1
   node := childQ.deque()
    if(node.left != null && node.right != null){
         cardinality = cardinality/2
    }
    node.left.cardinality = cardinality
    node.right.cardinality = cardinality
    childQ.enqueue(node.left)
    childQ.enqueue(node.right)
    parentQ.enqueue(node)
END LOOP

For each node in ChildQ:
   spawn threads for DFT

LOOP FOR parentQ =! empty
    node = parentQ.dequeue()
    print the node
    if(node.cardinality >2) {
            node = parentQ.dequeue()
            print the node
    }
    else {
            node = parentQ.dequeue()
            print the node
            if(node.left != null && node.right != null)
            {         childQ.dequeue()
                      print DFT of subtree
                            returned by thread
            }
            childQ.dequeue()
             print DFT of subtree
                   returned by thread
     }
END LOOP
```

Child and Parent Queue structure. cardinality = 4



Parent Queue:

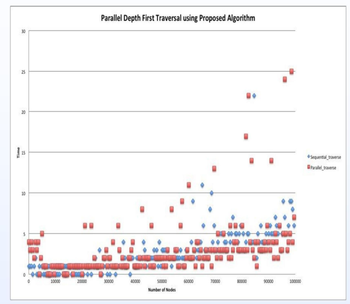
a(4) | b(2) | c(2)

Child Queue:

d(1) | e(1) | (f(1) | g(1)

Subtree Threads spawned

♦ Results: Parallel DFT



♦ Future Scope

Current implementation of Copy-On-Write copies the whole data structure in the process. A more elegant and optimized way of implementation is given in [1]. In the approach, each node that is traversed is copied and pointed to other nodes of original tree, removing the need of copying the whole data structure each time.
The proposed algorithm can be extended to n-ary tree and the performance can be evaluated against the sequential traversal. Also, the proposed algorithm works best for balanced binary tree. The distribution of number of threads to be assigned to a subtree can be optimized further to optimize the performance of un-balanced tree.

♦ References

[1] Bronson, Nathan G., et al. "A practical concurrent binary search tree." *ACM Sigplan Notices.* Vol. 45. No. 5. ACM, 2010.
[2] Kalra, N. C., and P. C. P. Bhatt. "Parallel algorithms for tree traversals."*Parallel Computing* 2.2 (1985): 163-171.
[3] http://pages.cs.wisc.edu/~skrentny/cs367-common/readings/Binary-Search-Trees/index.html