Concurrent-BST:
A key Value Store

Name – Pramod Kulkarni

Email – prku8035@colorado.edu

Algorithm Description:

Implemented a concurrent binary search tree that supports inserts, search, range queries and delete operations. Threads call the operations concurrently and synchronize between accesses using a mutex lock or a shared lock(cpp - 17). Based on the command line arguments a lock type is selected and operations are performed accordingly on the tree. Hand-over hand locking is used to avoid data – races.

Treenode class

Each node in the tree has a key and a value, pointers to left, right and parent nodes. Depending upon the type of argument supplied, lock type can either be a mutex lock or shared lock.

Concurrent_bst class

All the operations have been defined in this class. There are a couple of atomic member variables which are used to keep count of the successful operations during the run. Finally, these variables are used to verify if the algorithm indeed works. A global mutex lock is used to lock the tree initially and is released once lock on root node is acquired.

Insert

Insert takes in actual root, parent, key, value, and thread id as arguments. When insert is called from the main function the parent is set to null.

First condition that is checked in insert is whether parent is NULL. If that is the case then thread tries to acquire the tree lock, checks if actual root is NULL if that's true then a new node is created and assigned to actual root and tree lock is released and function returns. If actual root is not null then thread tries to acquire lock on root, after it successfully acquires the lock, tree lock is released.

If the key to be inserted is less than root's key, then it checks if root's left child is NULL and if true then thread creates a new node and sets it as left child of the root. Otherwise, attempt is made to acquire lock on left child. Root is unlocked once left child's lock is acquired. Insert is recursively called with left child now as the root.

If the key to be inserted is greater than root's key then we check if root's right child is NULL and if that's the case then thread creates a new node and sets it as right child of the root. Otherwise, attempt is made to acquire lock on right child. Root is unlocked once right child's lock is acquired. Insert is recursively called with right child now as the root.

If the key already exists, then the value is updated.

If a new node is inserted, then an atomic variable is incremented in the function.

Search

Search takes pointer to root node and parent, key and value, thread id. As insert, search acquires tree lock if the parent is NULL. It then checks if actual root is NULL then tree lock is released and returns false as tree is empty. If actual root exists, then lock to the root node is acquired and tree lock is released.

If key to be searched is less than root key, then root's left child is checked. If it does not exist, then search returns false and unlocks root. If left child exists, then thread tries to acquire lock on the left child. Once it gets the lock, root node is unlocked, and search is called recursively with left child now as the root.

If the key to be searched is greater than root key, then root's right child is checked. If it is NULL, then search returns false and unlocks root. If right child exists, then thread tries to acquire lock on the right child. Once it gets the lock then root node is unlocked and search is called recursively with right child now as the root.

If the key is found, then its value is checked with the one that is supplied. If both the values match, then an atomic variable is incremented to flag a successful search.

Range queries

Range query's initial part is same as that of search. It implements breadth first search. After root node's lock is acquired it is pushed into the queue. After popping node from the queue, thread tries to acquire left and right child node locks if their keys fall within the range and pushes them onto the queue. Key value in the popped node is matched with the supplied key range. If it falls within the query range, then it is added to a vector with its value. After queue is fully processed the vector is sorted and values are printed.

Delete

Delete operation is the most complicated one. We start with checking if the node we want to delete is the root. If root has no children, we delete root and set actual root to NULL and unlock the tree. If key does not match root node key, then we search for the node to be deleted in the tree and use hand over hand locking as we traverse the subtrees. If the node exists in the tree, then search returns pointer to the node and we can be sure that we have locked the node to be deleted and its parent node. There can be a case where the node we are trying to delete is the actual root because some other thread deleted the rest of the nodes in the tree. If node to be deleted is a leaf node then parent pointers are updated accordingly, and parent node is unlocked. If none of the above cases hold then thread is deleting either root with subtree or an internal node. Algorithm follows a copy mechanism and moves the node to be deleted. It then finds inorder successor and inorder predecessor and the tree is adjusted accordingly.

NOTE: For reader writer lock type multiple threads can hold shared lock during searching but insert and delete need unique lock.

Experimental results:

The code is tested using the parameters supplied via command line.

- NUM_ITERATIONs
- NUM_THREADS
- CONTENTION_TYPE
- LOCK_TYPE

The iterations range from 10 – 10,000 per thread. Number of threads range from 12 – 36 threads. Based on the thread id the first num_threads/4 insert into the tree. Next $1/4^{th}$ of the threads perform search, next $1/4^{th}$ delete, and final set executes parallel range queries. For every iteration, the thread sets seed and picks a random key to insert, delete or search. If the contention type is high, then thread with id 1 creates a skewed tree. Key and values range from 0 to 100000. All the operations try to access subset of keys which lead to scenario of high contention. Following tables showcase the runtimes for mutex and shared lock with high and low contention:

Mutex lock

Number of threads	Number of iterations	Low Contention(s)	High Contention(s)
12	100	0.004268	0.014805
12	1000	0.095395	0.295099
12	5000	0.481089	1.980946
12	10000	0.935547	19.299225
24	10000	40.089843	247.082904
36	10000	187.509448	727.812382

Reader writer lock

Number of threads	Number of iterations	Low Contention(s)	High Contention(s)
12	100	0.017426	0.016047
12	1000	0.159609	0.270248
12	5000	7.626684	9.756805
12	10000	7.842879	69.563456
24	10000	50.778322	517.907089
36	10000	423.478184	1533.988

From the above tables we observe that in both mutex lock and reader writer lock the low contention runs are significantly faster than high contention runs. Mutex lock performs better on both high and low contention runs than the reader writer locks. As search and range queries are happening in parallel, inserts and deletes must wait for exclusive till all reader threads exit making way for writer thread to acquire exclusive lock. That increases runtime significantly due longer wait times. In case of mutex lock such scenario does not arise and node locks are always exclusively held by threads trying to acquire it.

Analysis of results using perf:

Mutex lock page faults

Number of threads	Number of iterations	Low Contention	High Contention
12	100	170	176
12	1000	261	552
12	5000	1,384	2,480
12	10000	1,494	15,234
24	10000	2,736	13,545
36	10000	8,979	25,299

Reader writer page faults

Number of threads	Number of iterations	Low Contention	High Contention
12	100	170	178
12	1000	266	577
12	5000	716	2,278
12	10000	1,287	4,202
24	10000	3,604	16,214
36	10000	8,030	25,674

Page faults tell a similar tale. Page faults are more during high contention runs than low ones. Between lock types reader writer performs well in some cases on low contention and page faults are almost comparable. We notice bit of a difference in high contention runs between the two locks because modifications must wait for all the reader threads to exit. Reader threads run parallelly and hard page faults due to change in tree structure due to insert and deletes are less frequent than they were in mutex lock.

code organization:

1. Main function extracts the parameters supplied as command line arguments. *Lock type*, *num iterations*, *num threads* and *contention type* are populated.
2. Threads are created and worker function is called. If contention type is high, a skewed tree is constructed. Based on lock type either operations in concurrent_bst are called or operations in concurrent_bst_rw.
3. Worker threads are divided into four groups. Each group executes the assigned operations- inserts, search, delete and range query.
4. At every iteration in every operation atomic variables are incremented, both in main and respective functions.
5. At the end of the run inorder traversal is executed to verify correctness. Number of nodes in the tree after run should be equal to number of successful inserts – number of successful deletes.
6. Start time is recorded in worker and end time after joining all the threads. Finally, time taken for execution is printed.
7. When contention type is high, the initial insertion time is not accounted for in the final calculation.

A description of every file submitted:

1. *Concurrent-bst-main.cpp* houses the code for creating threads, calling tree functions based on command line arguments. Checks the correctness of the code after the execution completes and all worker threads join back.
2. *Concurrent-bst.cpp* implements concurrent operations like insert, delete, search and range queries performed on binary search tree. Nodes of the bst have a mutex lock and for convenience tree has a global lock.
3. *Concurrent-bst.h* declares the concurrent-bst and treenode classes.
4. *Concurrent-bst-rw.cpp* implements bst operations using a reader writer lock which is a shared lock in cpp 17. Same as concurrent-bst the operations use hand over hand locking. Every treenode has a shared lock and for convenience there is a global tree lock.
5. *Concurrent-bst.h* declares the concurrent-bst-rw and treenode-rw classes.
6. *Sense.cpp* cpp 17 does not have a barrier. Cpp - 20 does have a barrier but cpp – 20 isn't supported on jupyter. So, I decided to use barrier we created in lab-2. In high contention type, threads need to wait till thread id 1 creates a skewed tree, hence the need for a barrier.
7. *Sense.h* declares the sense barrier class and member functions
8. *Makefile* takes all the above-mentioned files and compiles them giving concurrent-bst executable application.
9. *Commons.h* has all the included needed for the project.
10. *Test.sh* is a shell script to run couple of examples.

Compilation instructions:

Cd into the project directory and just type – *make clean; make*

Execution instructions:

./concurrent-bst --name will print name and exit.

./concurrent-bst [--lock=<fg, rw>] [--contention=<low , high>] [-t NUM THREADS] [-i=NUM ITERATIONS]

Note: You need not execute above command. Running ***test.sh*** will run couple of test examples that execute in no time.

Extant Bugs:

None so far and hopefully it remains that way.