# Final Project Report

## TOPIC: CONCURRENT TREE

**INTRODUCTION:**

This is an implementation of parallel binary search tree with hand-over-hand locking and read write lock. This tree supports key-value pair insertion, key retrieval and range query search across different keys.

**IMPLEMENTATION OVERVIEW:**

The hand-over-hand locking tree mechanism consists of a bst node given below,

```
typedef struct node
{
    int key;
    int value;
    struct node *right;
    struct node *left;
    pthread_mutex_t lock;
}bst_node;
```

The node consists of a key-value pair, a lock, and pointer to right and left child nodes. The main purpose of having individual lock in every node is to perform multiple operations simultaneously without any write or read conflict. The lock is initialized on creation of every node and it is destroyed on freeing the tree.

Similar to hand-over-hand locking, I have also implemented concurrent tree using read-write lock. This implementation consists of a bst node given below,

```
typedef struct noderw
{
    int key;
    int value;
```

```
    struct noderw *right;

    struct noderw *left;

    pthread_rwlock_t lock;

}bst_noderw;
```

This bst node looks pretty similar to the previous node, expect the change in lock to read-write lock type.

In both the implementation, I have used a global lock shown below,

pthread_mutex_t bst_lock;

pthread_rwlock_t bst_rwlock;

to the lock the root node of the tree. After locking the root node, all other linked nodes can be traversed by acquiring the lock of the next node and then releasing the lock of the current node. This technique ensures the safety of tree nodes from any sort of data race violation.

**SUPPORTED TREE OPERATIONS:**

This parallel tree supports three operations put, get and range query. All of these are discussed below in detail,

- Put Operation
  The API for put operation is shown below,

  ```
  void put_node(bst_node *root, int key, int value, int thread_num)
  ```

  This put_node function takes three arguments – root, key, value, thread number and inserts it based on the key. If the root node is null, in that case a new node is initialized a root. This function also creates a new node using a helper function and initializes the respective mutex or read-write lock. To insert the key-value pair, it traverses the tree recursively by locking the next node and unlocking the current node, until it finds the proper position to link the new node.

- Get Operation
  The API for get operation is shown below,

  ```
  bst_node *get_node(bst_node *root, int key)
  ```

  This get_node function takes two argument – root, key and returns a node upon perfect match with a node key in the tree. The logic behind the key search is to

recursively traverse the tree until the key matches with the node key or the node becomes null.

- Range Query Operation
  The API for range query operation is shown below,

```
void get_nodes_inrange(bst_node *root, int start_key, int end_key, int tid)
```

  This `get_nodes_inrange` function takes four arguments – root, start key, end key, tid and pushes the in range nodes in the global vector. This vector based on the thread id is printed after completion of all the thread operations.

**FILES/FOLDER:**

**/test_files –** This folder contains all the test files

**Makefile –** Contains compilation instruction

**Fine_lock_bst.c –** Contains the definition of all the supported API's for parallel tree based on fine grained locking mechanism

**Fine_lock_bst.h –** Contains the declaration of all the supported API's for parallel tree based on fine grained locking mechanism

**Helper.c –** Contains the functions to parse the command line arguments and populate in the argument handling structure

**Helper.h –** Contains the declaration of helper API's for command line argument parsing.

**Main.c –** This is common interface to execute the application for both mutex lock and reader-writer lock.

**Rw_lock_bst.c -** Contains the definition of all the supported API's for parallel tree based on read-write locking mechanism

**Rw_lock_bst.h -** Contains the declaration of all the supported API's for parallel tree based on read-write locking mechanism

**Unit_test.c –** This files contains standalone test to validate the functionality of all the API's implemented in parallel tree.

**CODE COMPILATION AND EXECUTION INSTRUCTION:**

**Make test –** This build the test code to verify the get, put and range query operation

**Make tree –** This builds the parallel tree that supports the get, put and range query operation.

**Make file –** This builds the file code to create test files that would be the input to the get, put and range query operation.

**Make clean –** This cleans the project directory

**Run test**

To run unit test, do make test, then execute ./test

**Run tree**

To run tree, do make tree, then execute

./tree -i [insert filename] -s [search filename] -r [range query filename] -t [Number of threads] --lock=[lock_name]

These command line arguments are described below,

[Insert filename] - This file consists of all the key-value pair (line separated) that must be inserted in the tree.

[search filename] – This file consists of all the keys (line separated) that must be searched in the tree.

[range query filename] – This file consists of all range query with start key and end key (line separated) that must be performed in the tree.

[Number of threads] – Number of thread that should be used for the operation

[lock_name] – Specify the lock to use. Use “rw_lock” for reader – writer lock and “mutex” for the conventional posix mutex lock.

**Run file**

To run file, do make file and then execute,

./file [insert filename] [Number of inserts] [Maximum limit] [search filename] [Number of search’s] [Range query filename] [number of queries]

This generates all the specified file and these files can be used to execute the parallel tree.

Note: In the test_files folder all the test files are present which can be used for testing. Also, this exe is just to generate test files and it has not been tested extensively. It is recommended to use the existing test files.

**ANALYSIS:**

To perform the high contention and low contention analysis, I chose a methodology to insert equal number of key-value pairs and then search the same key and execute the same

query for high contention testing. On other hand, for low contention testing search random keys and execute random search queries. Calculate the time taken separately for the high contention execution and low contention execution.

The results of various test has been tabulated below,

For mutex based fine grained locking –

| Number of elements | Lower Bound | Upper Bound | Operation time in high contention case (in seconds) | Operation time in low contention case (in seconds) | Lock |
|---|---|---|---|---|---|
| 10000 | 0 | 10000 | 4.47 | 0.24 | Mutex |
| 10000 | 0 | 10000 | 3.9 | 0.19 | Reader – writer lock |
| 1000 | 0 | 1000 | 0.108 | 0.058 | Mutex |
| 1000 | 0 | 1000 | 0.07 | 0.05 | Reader – writer lock |
| 100 | 0 | 100 | 0.00256 | 0.00032 | Mutex |
| 100 | 0 | 100 | 0.00035 | 0.00017 | Reader – writer lock |

Example instruction to run the high contention and low contention case,

```
./tree -i test_files/insert_10000.txt -s test_files/search_10000_highcontention.txt -r
test_files/range_10000_highcontention.txt -t 6 --lock=rw_lock

./tree -i test_files/insert_10000.txt -s test_files/search_10000_lowcontention.txt -r
test_files/range_10000_lowcontention.txt -t 6 --lock=rw_lock

./tree -i test_files/insert_10000.txt -s test_files/search_10000_highcontention.txt -r
test_files/range_10000_highcontention.txt -t 6 --lock=mutex

./tree -i test_files/insert_10000.txt -s test_files/search_10000_lowcontention.txt -r
test_files/range_10000_lowcontention.txt -t 6 --lock=mutex

./tree -i test_files/insert_1000.txt -s test_files/search_1000_highcontention.txt -r
test_files/range_1000_highcontention.txt -t 6 --lock=rw_lock

./tree -i test_files/insert_1000.txt -s test_files/search_1000_lowcontention.txt -r
test_files/range_1000_lowcontention.txt -t 6 --lock=rw_lock

./tree -i test_files/insert_1000.txt -s test_files/search_1000_highcontention.txt -r
test_files/range_1000_highcontention.txt -t 6 --lock=mutex

./tree -i test_files/insert_1000.txt -s test_files/search_1000_lowcontention.txt -r
test_files/range_1000_lowcontention.txt -t 6 --lock=mutex
```

## Perf Analysis

## For high contention with mutex lock

```
 Performance counter stats for './tree -i test_files/insert_10000.txt -s test_files/search_10000_highcontention.txt -r test_files/range_10000_highcontention.txt -t 6 --lock=mut
ex':

    7,326,321,750      L1-dcache-loads                                       (80.15%)
      324,104,307      L1-dcache-load-misses     #    4.42% of all L1-dcache hits   (79.91%)
       14,299,045      L1-icache-load-misses                                 (79.97%)
    3,984,842,632      branch-instructions                                   (80.00%)
       49,945,894      branch-misses             #    1.25% of all branches         (79.98%)
          506,029      page-faults

      3.500392830 seconds time elapsed
```

## For low contention with mutex lock

```
 Performance counter stats for './tree -i test_files/insert_10000.txt -s test_files/search_10000_lowcontention.txt -r test_files/range_10000_lowcontention.txt -t 6 --lock=mutex
':

       73,355,332      L1-dcache-loads                                       (82.85%)
        5,607,419      L1-dcache-load-misses     #    7.64% of all L1-dcache hits   (73.56%)
        8,894,908      L1-icache-load-misses                                 (75.84%)
       60,533,188      branch-instructions                                   (83.45%)
          874,806      branch-misses             #    1.45% of all branches         (84.30%)
              373      page-faults

      0.182318090 seconds time elapsed
```

## For high contention with read-write lock

```
 Performance counter stats for './tree -i test_files/insert_10000.txt -s test_files/search_10000_highcontention.txt -r test_files/range_10000_highcontention.txt -t 6 --lock=rw_
lock':

    8,039,112,916      L1-dcache-loads                                       (79.96%)
      259,721,138      L1-dcache-load-misses     #    3.23% of all L1-dcache hits   (80.03%)
       14,284,181      L1-icache-load-misses                                 (79.97%)
    4,153,032,449      branch-instructions                                   (79.95%)
       49,292,391      branch-misses             #    1.19% of all branches         (80.09%)
          500,980      page-faults

      5.116267235 seconds time elapsed
```

## For low contention with read-write lock

```
 Performance counter stats for './tree -i test_files/insert_10000.txt -s test_files/search_10000_lowcontention.txt -r test_files/range_10000_lowcontention.txt -t 6 --lock=rw_lo
ck':

       76,365,785      L1-dcache-loads                                       (81.93%)
        6,817,049      L1-dcache-load-misses     #    8.93% of all L1-dcache hits   (81.74%)
       11,127,161      L1-icache-load-misses                                 (78.78%)
       62,343,025      branch-instructions                                   (77.46%)
          939,153      branch-misses             #    1.51% of all branches         (80.10%)
              316      page-faults

      0.180681595 seconds time elapsed
```