CONCURRENT, LOCK-FREE RED-BLACK TREES

CMU 15-418 PARALLEL COMPUTER ARCHITECTURE AND PROGRAMMING
FINAL PROJECT REPORT

Claire C. Chen
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA
`ccz@cs.cmu.edu`

Joseph McLaughlin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA
`jmmclaug@andrew.cmu.edu`

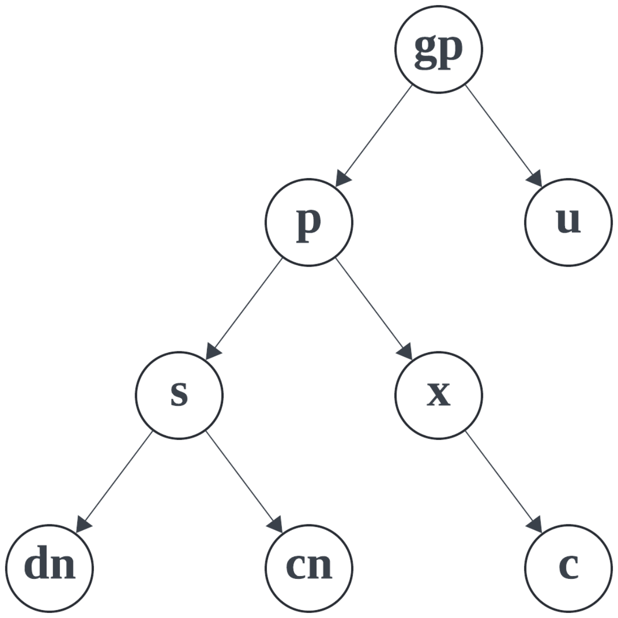
May 5, 2024

ABSTRACT

We present an OpenMP implementation of parallel, lock-free Red-black Trees supporting concurrent insertion operations under the shared-address space model based on previously designed algorithms. We iterated upon a prior implementation and pseudocode described in the literature, identifying sources of correctness issues. Our implementation achieved a sublinear speedup on large input test cases.

1 Notation

Throughout this report we refer to nodes by their relation to the critical nodes in various Red-Black operations such as the nodes being inserted. We use the following naming convention based on the tree below:



Short Name	Full Name
x	Node
p	Parent
c	(Either) Child
s	Sibling
dn	Distant Nephew
cn	Close Nephew
gp	Grandparent
p	Uncle

Table 1: Node Naming Conventions

We additionally use the following terms not shown in the tree:

- An *Outer Child* is a node whose direction from it's parent is the same as the parent's direction from the grandparent. In the above tree, **s** is an outer child.
- An *Inner Child* is a node whose direction from it's parent is opposite of the parent's direction from the grandparent. In the above tree, **x** is an inner child.

2 Background

2.1 Data Structure

Binary Search Trees are useful in a variety of information storage and retrieval algorithms for having potentially sublinear insertion and deletion costs but can't guarantee asymptotically better performance on their own due to imbalancing. Various balancing schemes exist to achieve faster guaranteed bounds on insertion and deletion time at the cost of algorithm intricacy. Red-Black Trees are one commonly used self-balancing binary search tree that achieves logarithmic bounds for common operations such as insertion and deletion.

We also selected Red-Black Trees (RBT) due to their relatively localized effects of RBT operations compared to other self-balancing binary search trees (BST), which enables more opportunities for parallelism, and a low memory footprint typically requiring no more memory overhead than uncolored BST's [1] [4].

Red-Black Trees achieve their time complexity guarantee by maintaining the following invariants beyond the standard BST invariant:

1. Every node is colored either "Red" or "Black"
2. All leaves are considered "Black"
3. No two "Red" nodes are allowed to be adjacent to one another
4. All paths from the root to a leaf node must traverse through the same number of "Black" nodes.

By maintaining these invariants, we can ensure a red-black tree stays roughly balanced since the longest path can only be at most twice as long as the shortest path, as otherwise either the longer path would have more black nodes or two red nodes would appear consecutively in the path.

Red-Black trees are particularly interesting for the mechanisms by which they maintain these invariants. Red-Black Trees insert a node by first placing it at a leaf based on its value, thus maintaining the BST invariant of the tree. They attempt to fix their red-black invariants by correcting the color and position of nodes approximately above the node in the tree. A more formal exhibition of these routines is shown below:

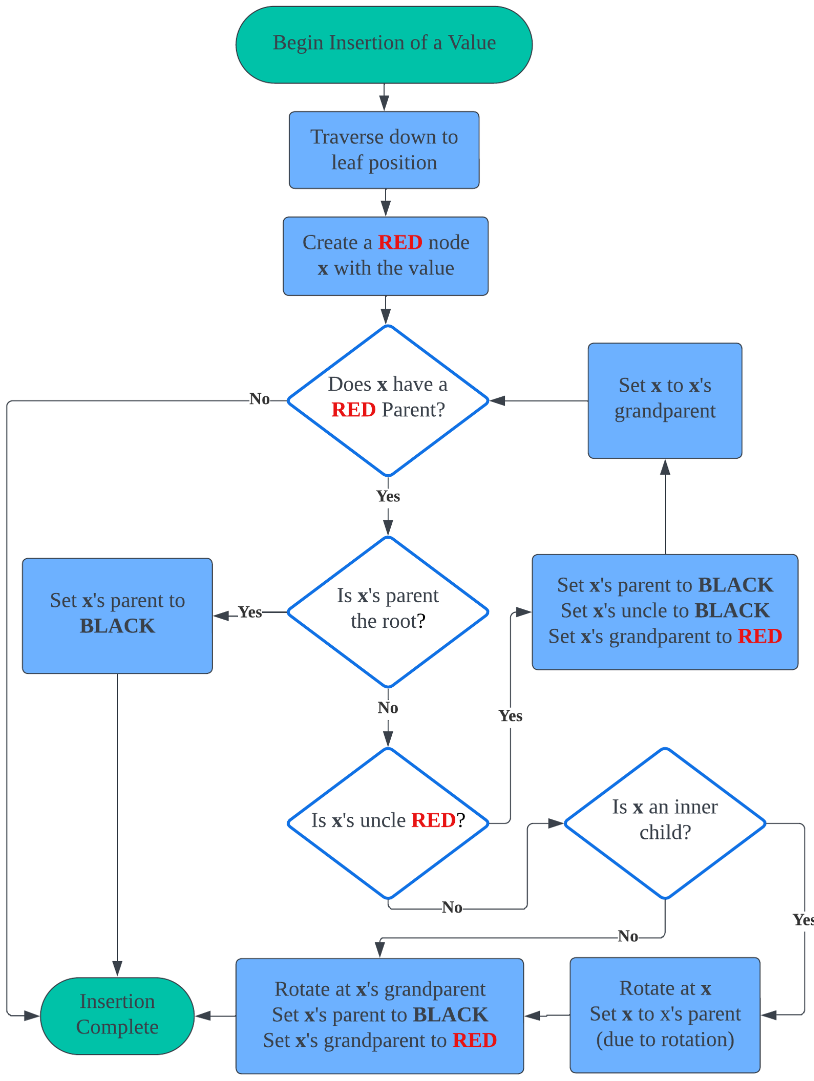


Figure 1: Steps for Sequential Red-Black Tree Insertion

2.2 Opportunities for Parallelism

Opportunities for parallelism come from exploiting a red-black Tree's spatial locality in self-balancing operations when inserting or deleting a node. In particular, an insertion operation requires at most two rotations, while deletion only uses at most three. Additionally, both algorithms only recolor nodes on the general path between the starting leaf and the root node as per the above flowchart, where all reads and writes occur with nodes a constant distance away from the critical node. This leads to the opportunity of two insertion operations occurring simultaneously without interference so long as they are guaranteed to work on distant enough portions of the tree. This provides room for parallelism in bulk insertion operations, particularly for inputs with random values.

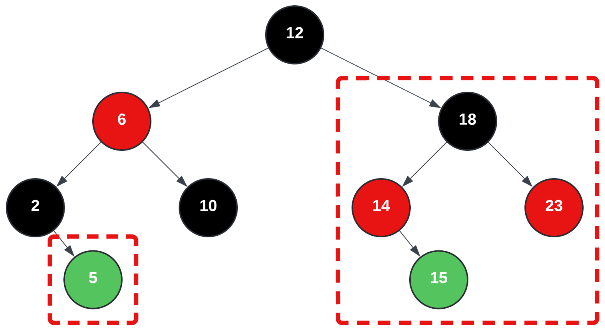


Figure 2: Example of two insertions that don't directly conflict, with affected nodes boxed.

One additional insight is that the exact ordering of bulk insertion operations doesn't affect correctness as defined by the inclusion or exclusion of nodes in a valid self-balancing BST. Even if two operations happen to occur out of order, both elements will still be successfully inserted, which allows us to process insertions without concern for operation ordering. This unfortunately does not apply to mixed test cases with both insertion and deletion, however, as inserting and deleting the same node in opposite orders can cause incorrect results. For example, inserting then deleting an absent value results in no change in tree membership while deleting then inserting a node results in the value being added.

3 Approach

3.1 Theory

The complexity in parallelizing red-black trees occurs with two threads simultaneously working in nearby areas, as the changes caused by one thread can affect the correctness of the other's operations. In particular, insertion operates on four nearby nodes as shown below. To implement this, Ma's lock-free insertion algorithm [2] suggests we mark each operation's area of effect, or *local area*, by setting the `flag` field in each node in the local area. By performing a compare-and-swap operation for acquisition of a node's flag, we can ensure no two operations are applied to the same node at the same time.

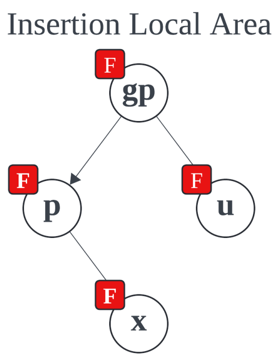


Figure 3: Insertion Local Areas are Flagged

Building on Ma's work involving three types of synchronization primitives, Kim et. al simplified the lock-free insertion algorithm to involve only CAS operations when setting flags during insertion [1], as the operations on nodes in the local area can safely occur when the flags are set. When one thread flags a local area, other threads intending to access nodes in the same local area would fail to set the flags, forcing them to return to the the tree's root and re-attempt the operation.

This leads to a sensible parallel insertion implementation based on the invariant that a thread can only access a node once it acquires a flag for it:

We begin by traversing the tree down to the leaf position that corresponds to where the inserted value would belong. We first flag the root node, then repeatedly move our flag down in a hand-over-hand fashion, ensuring we always have at least one flag as we go down. If we encounter a flagged node on our traversal downwards we are forced to restart our traversal from the root, as otherwise we may prevent the thread with the flag from continuing propagating changes upwards and create a deadlock situation.

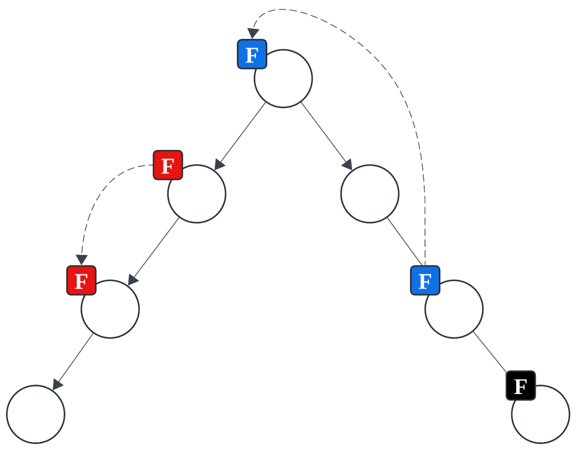


Figure 4: Depiction of flag traversal downwards and restarting when encountering another flag.

Once the leaf node position is found, a new node is created as in the sequential implementation, and the local area for the new node is flagged to prevent other nodes from making unwanted local changes as per the local area tree shown above. We can then go through the same casing as we have for a sequential red black tree implementation, with the following caveats:

- In the fixup case of insertion where we set x to x's grandparent, we need to flag a new local area to do subsequent insertion operations on the new region.
- When rotating, we need to take care to keep track of which vertices a thread has flagged, as they move around during rotation.

Once the insertion routine is complete, we simply clear the flags we instantiated for the local area and return.

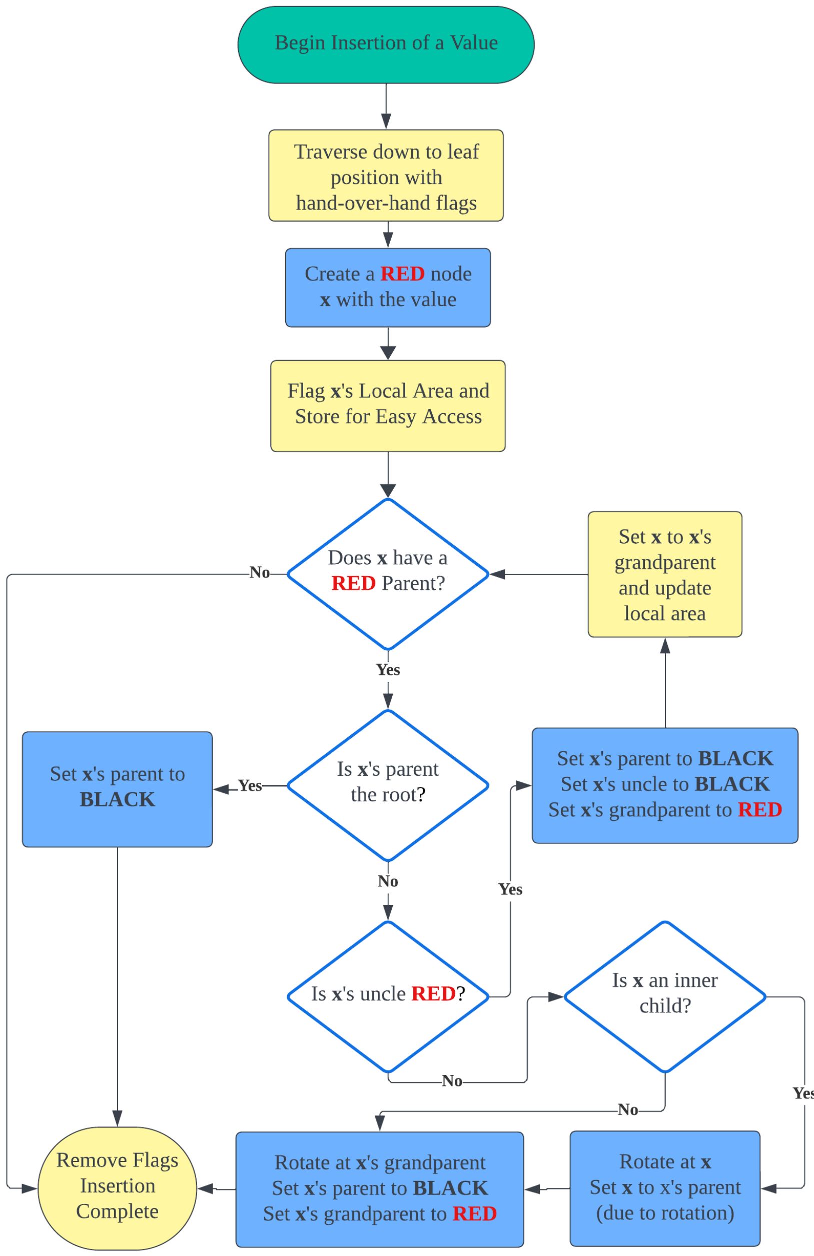
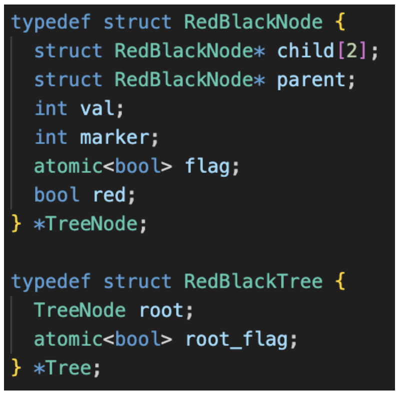

Begin Insertion of a Value
Traverse down to leaf position with hand-over-hand flags
Create a RED node x with the value
Flag x's Local Area and Store for Easy Access
Does x have a RED Parent?
No
Yes
Set x to x's grandparent and update local area
Is x's parent the root?
Yes
No
Set x's parent to BLACK
Set x's parent to BLACK Set x's uncle to BLACK Set x's grandparent to RED
Yes
Is x's uncle RED?
No
Is x an inner child?
Yes
No
Remove Flags Insertion Complete
Rotate at x's grandparent Set x's parent to BLACK Set x's grandparent to RED
Rotate at x Set x to x's parent (due to rotation)

Figure 5: Steps for Parallel Red-Black Tree Insertion (Changes from Sequential Highlighted in Yellow)

3.2 Implementation

We used OpenMP for our implementation since it is the most popular and widely used shared-memory parallel programming API. Since OpenMP is available in C, C++, and Fortran, we selected C++ for its support for convenient pointer manipulation and automatic type inference, as Red-Black Trees are a complex, multi-link data structure. We first created a sequential implementation of Red-Black trees adapted from the Wikipedia page on Red-Black Trees [4].

Since lock-free data structures are only possible in a shared memory system, we targeted machines with multi-core CPU's, such as the GHC machines and PSC machines. The Gates cluster machines each have 8 cores, with hyperthreading disabled, so only 8 threads are available.



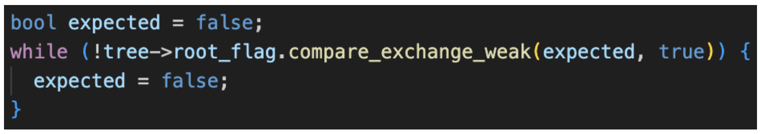
```
typedef struct RedBlackNode {
  struct RedBlackNode* child[2];
  struct RedBlackNode* parent;
  int val;
  int marker;
  atomic<bool> flag;
  bool red;
} *TreeNode;

typedef struct RedBlackTree {
  TreeNode root;
  atomic<bool> root_flag;
} *Tree;
```

Figure 6: Tree Node and Tree structs

We began by defining our Red-Black Tree as a struct with access to the root node. Nodes are defined as structs that include, in addition to the val, parent, children, and color required to implement a sequential red-black tree, an atomic boolean to act as the flag a thread sets to claim access to a node. One note about our implementation is that while Ma's algorithm only explicitly calls for a flag at each node, we found our implementation with a separate Tree struct necessitated an additional "root_flag" to grant access to updating the root of the red-black tree. We found that without this flag, we would run into cases where an insertion call would attempt to start propagating down by finding the root node, but a rotation at the root would cause the root node to change and make any subsequent traversal by the insertion invalid.

We discovered what we believe to be a critical correctness issue with a prior 618 final project implementation on lock-free red-black trees [5], which we referred to while implementing our own. In particular, we also implemented concurrency-safe flag acquisition using the atomic library's `compare_exchange_weak` function (as our CAS primitive), either looped on it or re-called the insert function upon failure. During our extensive testing stage, we discovered a correctness issue with Zhang et. al's prior implementation, specifically relating to their use of `compare_exchange_weak`. As shown in the code snippet below, when `compare_exchange_weak` is called on `expected=false`, if it fails, the value of `expected` is reset to the desired value to change to (`true`). This almost always causes `compare_exchange_weak` to evaluate to `true` in the second iteration of the while loop, since the operation that set the flag to true likely hasn't finished in the time an empty while loop completes an iteration. Hence, the given thread's operation would proceed without properly securing exclusive access to the flag, since the flagged local area is still in use by another thread. This correctness issue went unheeded, possibly due to the prior implementation's lack of a correctness checker. To mitigate this correctness issue, we reset the expected `argument` to CAS in the loop body, since the implementation overrides the "expected" parameter with the true value if it fails.



```
bool expected = false;
while (!tree->root_flag.compare_exchange_weak(expected, true)) {
  expected = false;
}
```

Figure 7: Example Compare Exchange Weak Loop

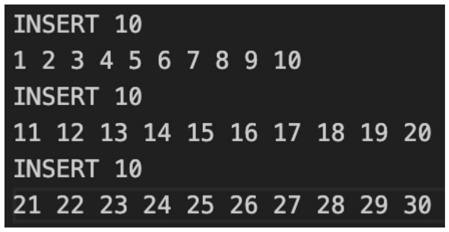
Once a leaf node was reached, we implemented a helper routine to acquire the flags for the node's local area. We additionally store these nodes in a vector associated with the thread in order to keep track of them easily for deletion and rotation. We then used the same red-black algorithm as the sequential case to perform the red-black operations once the local area has been set. Finally, we clear the local area by simply iterating over the vector of stored flagged nodes and unflagging each one.

4 Results

To benchmark our implementation, we ran our code on the 8-core GHC cluster machines with randomized inputs of various sizes. We created a testing script (`test-gen.py`) to allow us to generate test cases with scenarios simulating any number of concurrent RBT operations at each timestep, as well as the number of timesteps. We utilized trace-based simulation in order to mirror how red-black trees are used to store, query, and modify data, as real-world systems are designed to handle concurrent requests.

One note about our traces is that because lock-free RBT algorithms are still blocking (as mentioned in prior work by Natarajan et. al on RBT's [3]), sorted or adversarially ordered inputs could limit the degree of parallelism as threads accessing similar regions of the RBT repeatedly re-attempt, reducing the concurrent implementation to a sequential one. We therefore designed our testing suite to support the generation of various sizes of *random* inputs.

We determined the algorithm's computation time for each test case by summing the time spent on calls to bulk insertion, using the C++ `chrono` library. Since we wanted to test the efficiency of our red-black tree algorithm rather than the supporting routines of our testing code, this allowed us to isolate the time spent on bulk insertion. We computed speedup by finding the ratio between time spent with 1 processor and multiple processors.



```
INSERT 10
1 2 3 4 5 6 7 8 9 10
INSERT 10
11 12 13 14 15 16 17 18 19 20
INSERT 10
21 22 23 24 25 26 27 28 29 30
```

Figure 8: A (toy) example of a test file we used for benchmarking. Each pair of rows represents one bulk insert operation. In our actual test cases the values inserted were unsorted, random, unique integers. The trace pictured would simulate 10 concurrent insertion operations for values 1-10 in timestep 0, 10 for values 11-20 in timestep 1, and 10 for values 21-30 in timestep 2.

4.1 Insertion Speedup

In general, we found speedup to be substantial, especially for large test cases, but still sublinear. The speedup graph below corroborates this, showing larger test cases reaching as high as 4.85x speedup on larger test cases while the smallest test case exhibited worse performance with more processors.

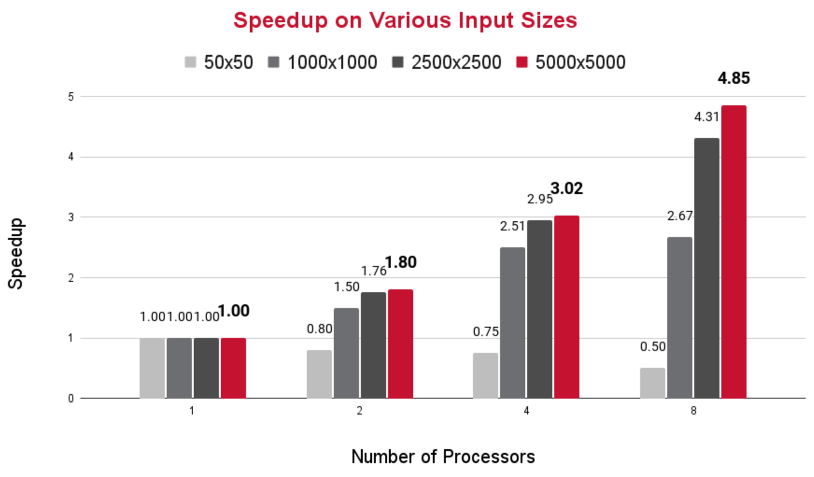


Figure 9: Insertion speedup on various input sizes, where nxn represents a file with n bulk insertions of n values to be inserted in each timestep.

This inefficiency seems to be largely due to contention for the root node flag in our implementation. The standard red-black tree implementation requires insertion to begin by searching for the location the node should be inserted into, which is done by searching down from the singular root node. Since our implementation uses a hand-over-hand method to traverse down the tree while controlling flags along the way which is necessary to ensure the insertion isn't interfered with by other operations, all insertion operations must begin by taking control of the root node's flag. This causes the flag to be under high contention, especially on smaller inputs where the rest of the insertion algorithm happens faster and the root node is also more likely to be flagged by insertion fixup operations propagating to the root node.

We confirmed this hypothesis by measuring the total time each thread spent while waiting for access to the root node during insertion. We wrapped timing code around the code that allows a thread to first access the root node, and observed the total time spent in the timed code section. We found that this waiting took as much as half of the total time spent for each processor with 8 processors, and still took noticeably longer with 4 processors than with fewer.

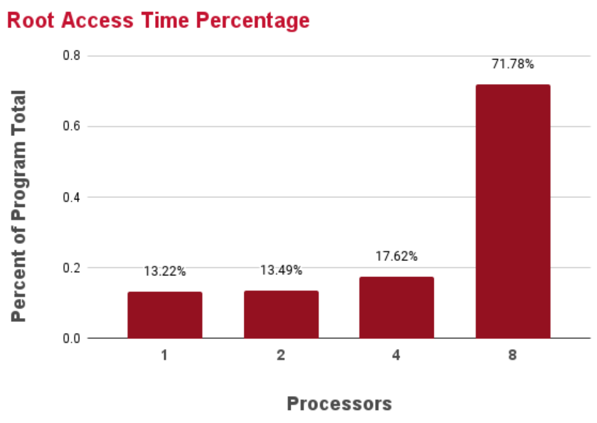
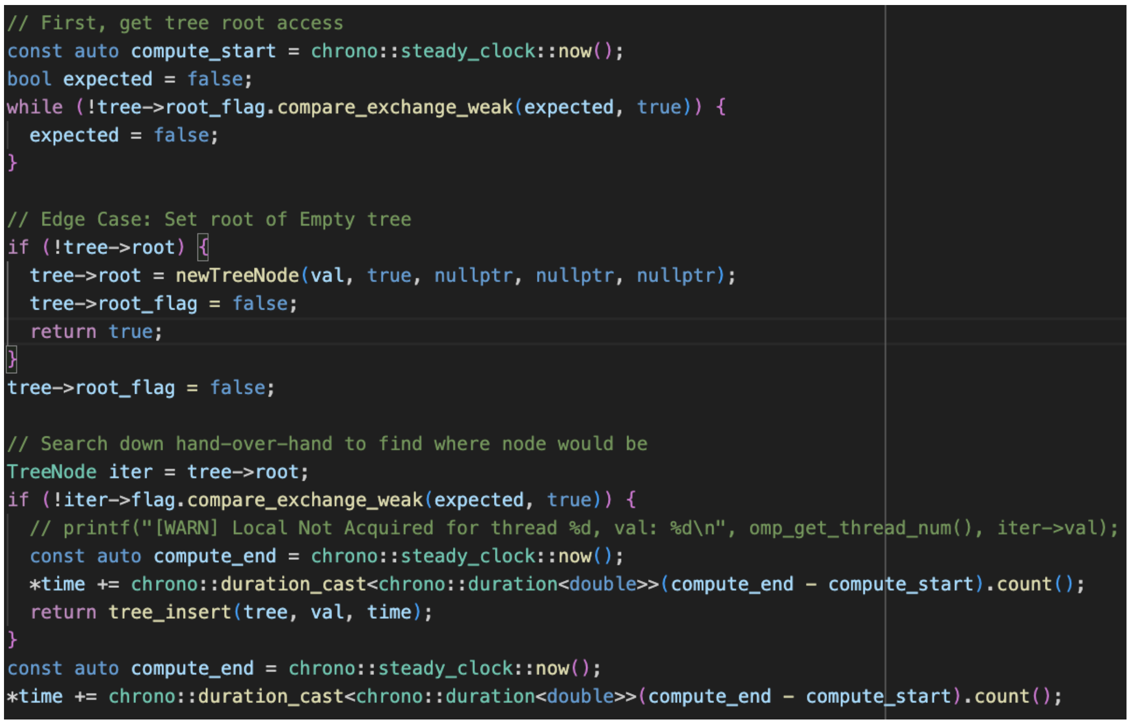


Figure 10: Percent of time spent on acquiring root access on a 1x100000 test case to demonstrate the time spent on a single bulk insert call.



```
// First, get tree root access
const auto compute_start = chrono::steady_clock::now();
bool expected = false;
while (!tree->root_flag.compare_exchange_weak(expected, true)) {
  expected = false;
}

// Edge Case: Set root of Empty tree
if (!tree->root) {
  tree->root = newTreeNode(val, true, nullptr, nullptr, nullptr);
  tree->root_flag = false;
  return true;
}
tree->root_flag = false;

// Search down hand-over-hand to find where node would be
TreeNode iter = tree->root;
if (!iter->flag.compare_exchange_weak(expected, true)) {
  // printf("[WARN] Local Not Acquired for thread %d, val: %d\n", omp_get_thread_num(), iter->val);
  const auto compute_end = chrono::steady_clock::now();
  *time += chrono::duration_cast<chrono::duration<double>>(compute_end - compute_start).count();
  return tree_insert(tree, val, time);
}
const auto compute_end = chrono::steady_clock::now();
*time += chrono::duration_cast<chrono::duration<double>>(compute_end - compute_start).count();
```

Figure 11: Timing code used to wrap around root access and usage portions of the insert function. Note the two compute_end lines at the end are to calculate the time regardless of whether each individual compare_exchange succeeds.

Since this root access is part of the base red-black tree algorithm and must be flagged to prevent a situation in which one insertion call attempts to locate the root node while the root node is being changed, we feel this time spent acquiring the root node is largely unavoidable.

4.2 Dynamic vs. Static Scheduling

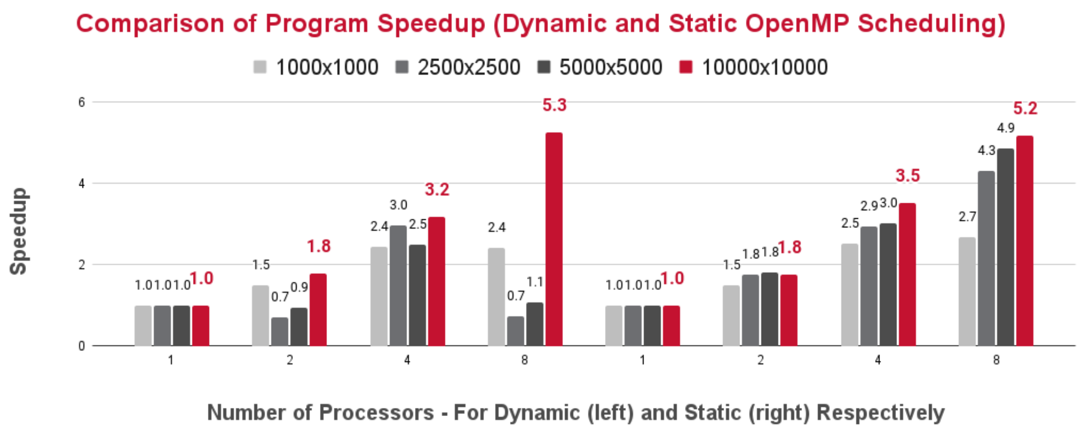


Figure 12: Speedup for Dynamic and Static Scheduling on various nxn graphs

As part of our optimization stage, we experimented with using either a dynamic or static scheduling policy for threads to parallelize over concurrent insertion operations in the same timestep (with the associated graph on the next page). We discovered that static scheduling ended up scaling and led to consistently higher speedup as the number of processors increased, especially in the 8-processor case. This observation holds true across test inputs of varying sizes. For the red-black tree data structure, the overhead of dynamic scheduling eclipses any potential benefits for balancing workloads across concurrent insertion operations. We hypothesize that this is possibly due to the relatively localized nature of red-black tree operations, so the threads don't have to significantly busy wait. Furthermore, there are few stalls due to data access (since pointer operations on RBT's resemble random memory accesses) or locking (our implementation is lock-free).

5 Work Distribution

Overall, the work distribution was **50% - 50%**, with both team members contributing an equal amount. The two team members completed the sequential and parallel implementation synchronously either in-person or via Zoom and VSCode Liveshare. In particular, Joe led the effort with the sequential implementation, while Claire led the effort on our testing schema. Our utility functions and parallel implementation were done synchronously and thus reflect an equal effort from both of us.

References

[1] Jong Ho Kim, Helen Cameron, and Peter C. J. Graham. Lock-free red-black trees using cas. 2011.

[2] Jianwen Ma. Concurrent, lock-free insertions in red-black trees, 2003.

[3] Aravind Natarajan, Lee H. Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 45–60, Cham, 2013. Springer International Publishing.

[4] Wikipedia. Red–black tree — Wikipedia, the free encyclopedia, 2024.

[5] Shun Zhang and Guancheng Li. Zhangshun97/lock-free-red-black-tree: Implementation of lock-free red-black tree using cas, 2019.