Concurrent, Lock-free Red-black Trees

Claire Chen, Joseph McLaughlin

Carnegie Mellon University

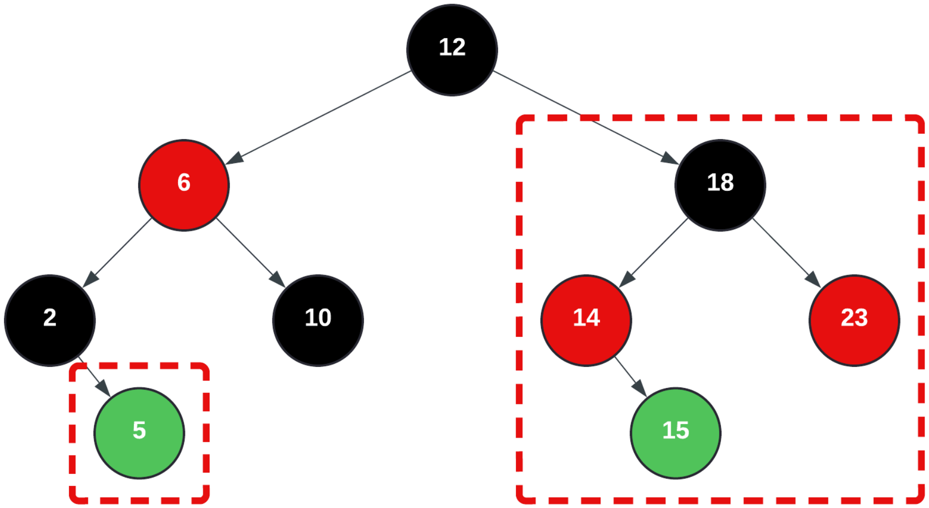
We present an OpenMP implementation of parallel, lock-free Red-black Trees supporting concurrent insertion operations under the shared-address space model based on previously designed algorithms. We iterated upon a prior implementation and pseudocode described in the literature, identifying sources of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

Why Red-Black Trees?

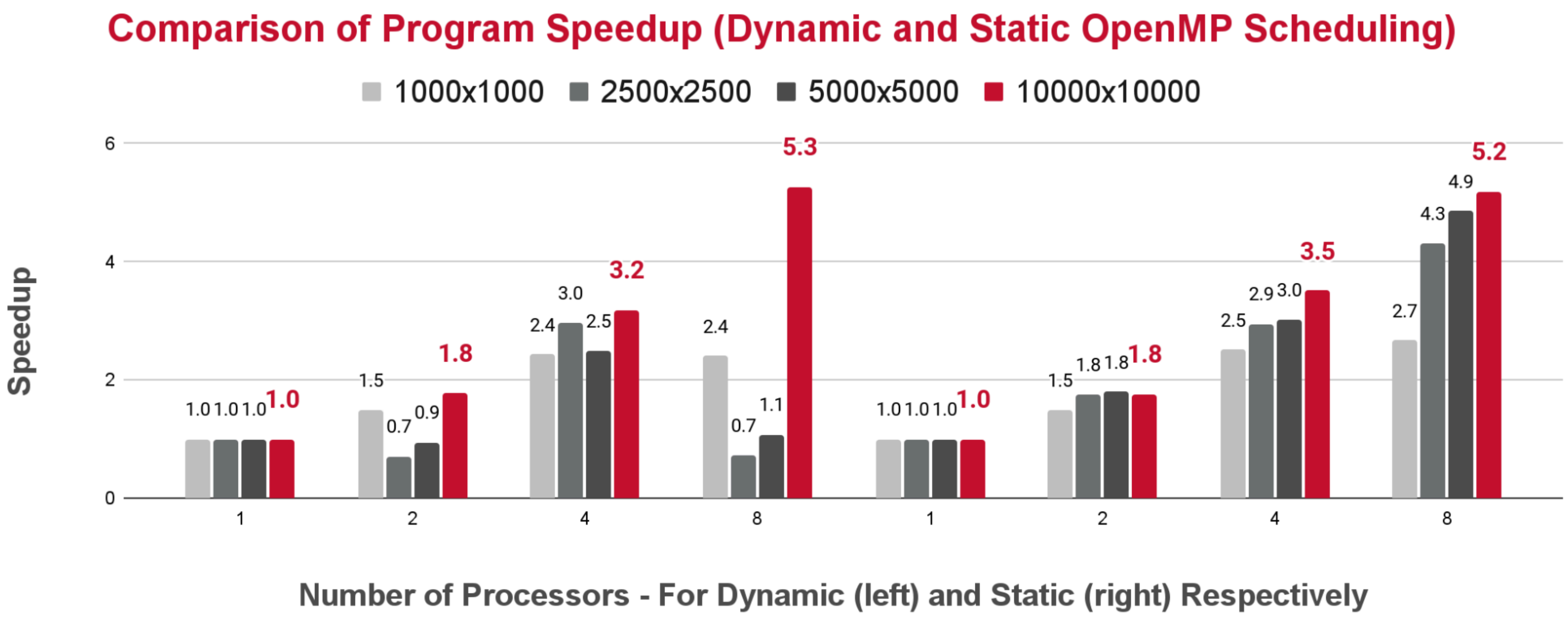
1. The effects of RBT operations are **relatively localized** compared to other BST's → more opportunities for parallelism
2. Asymptotic runtime efficiency: Logarithmic guarantee.
3. Memory efficiency: RBT's only require an additional bool (1 bit) to store color at each node.
4. Arbitrary ordering of concurrent insertions **doesn't affect the correctness**

Opportunity for Parallelism

We exploit the localized effects of self-balancing operations in RBTs. Insertion requires at most two rotations from the starting node. All reads and writes occur to nodes with a constant distance away from the critical node. Hence, operations can be concurrent insofar as they are guaranteed to occur at sufficiently safe distances.



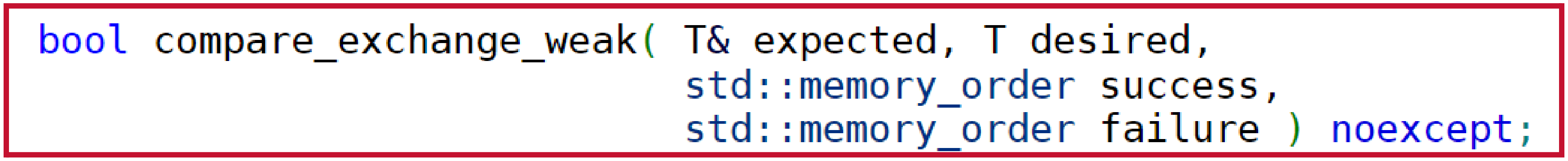
Two insertions with non-overlapping effects can safely occur in parallel.



Compared to dynamic scheduling, static scaled better and had consistently higher speedup. The overhead of dynamic likely eclipses any potential benefits for balancing workloads across concurrent insertion operations. There are few stalls due to similar data access times (since pointer operations on RBTs resemble random memory accesses) and the absence of locking.

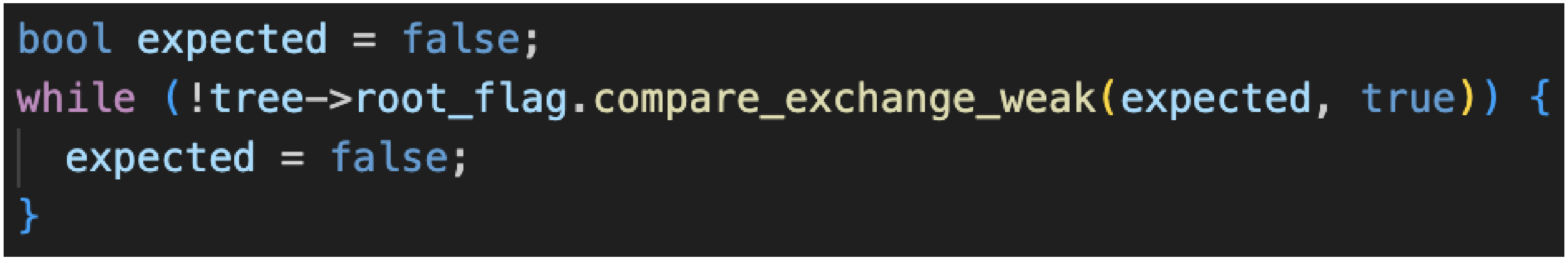
Mechanism for Safe Concurrency:

Atomic CAS Primitive - `compare_exchange_weak()`



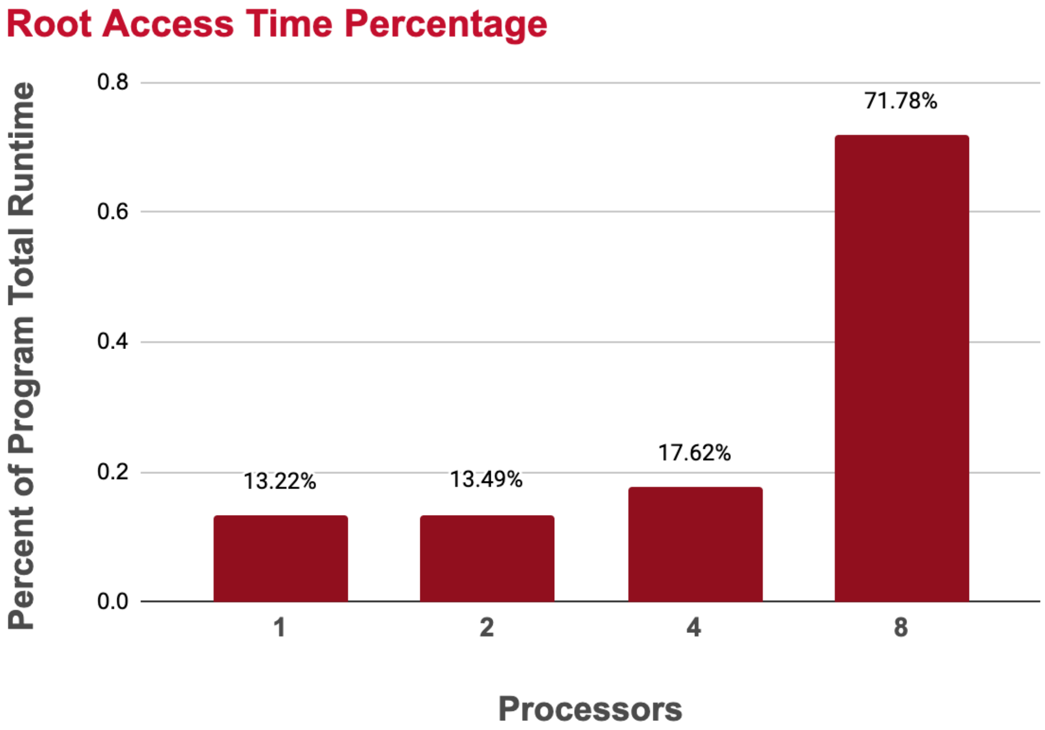
```
bool compare_exchange_weak( T& expected, T desired,
                            std::memory_order success,
                            std::memory_order failure ) noexcept;
```

which loads the actual value stored in **`*this`** into **`expected`** if the operation fails, returning **`false`**.

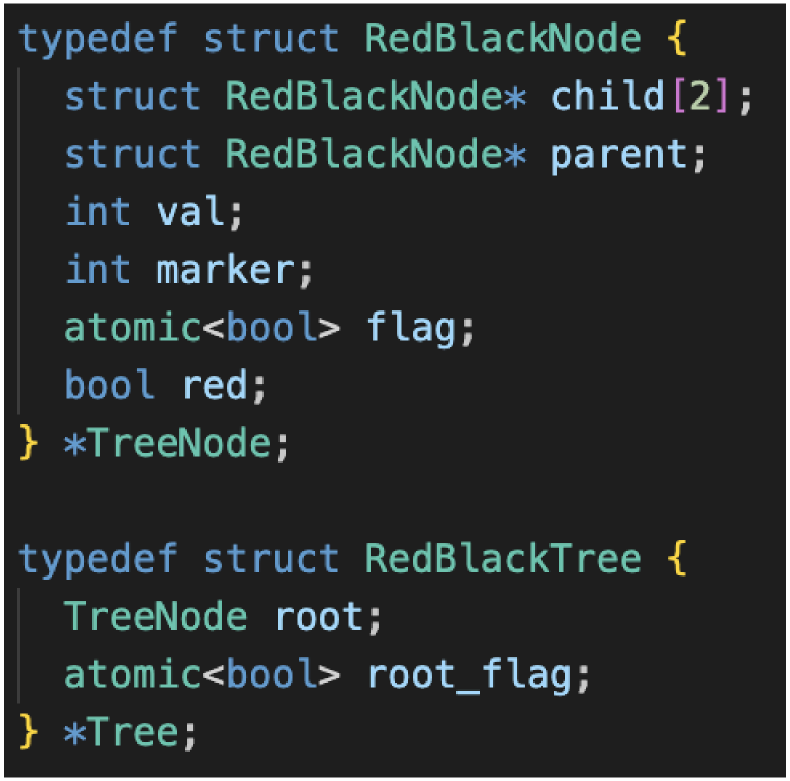


```
bool expected = false;
while (!tree->root_flag.compare_exchange_weak(expected, true)) {
  expected = false;
}
```

Hence, we identified a correctness issue with a prior parallel RBT implementation which failed to reset the value of **`expected`** to false when flag acquisition is not possible. Without **`expected=false`**, if the first call fails, the second call will always succeed, often incorrectly permitting the given thread's operation to proceed without properly securing exclusive access to the flag.

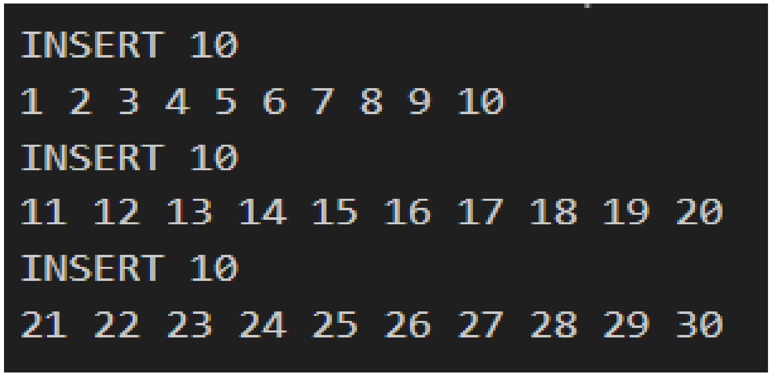


The limited speedup likely stems from the threads' contention for the root node flag. All insertion operations must begin by taking control of the root node's flag, causing the flag to be under high contention, especially on smaller inputs.



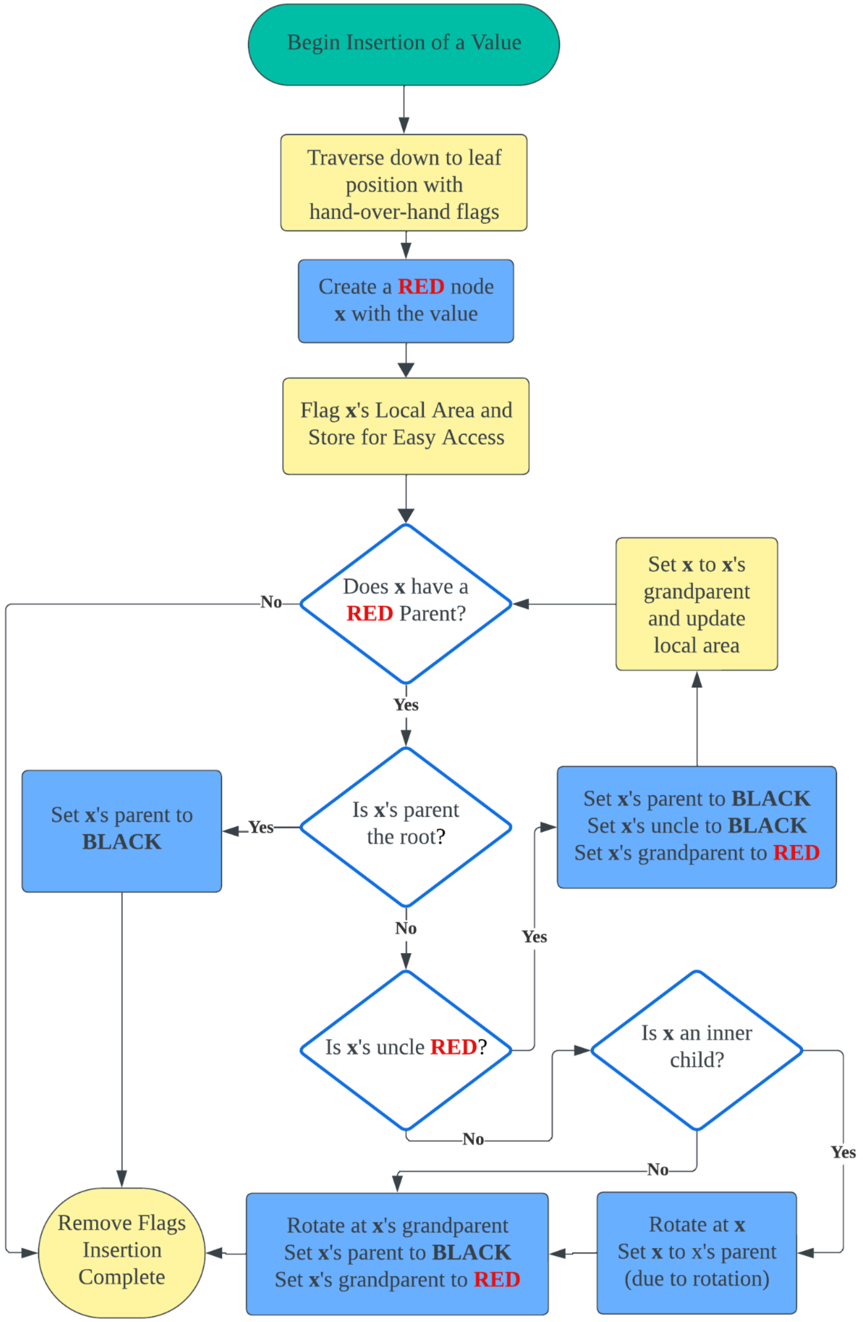
```
typedef struct RedBlackNode {
  struct RedBlackNode* child[2];
  struct RedBlackNode* parent;
  int val;
  int marker;
  atomic<bool> flag;
  bool red;
} *TreeNode;

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

Approaches described in Ma's pseudocode only suggest adding a flag to each node. However, for our implementation, the separate `Tree struct` required an additional `root_flag` to grant access to updating the root of the red-black tree. Without this flag, rotations impacting the root could occur as an inserting thread traverses down the tree, thereby rendering subsequent reads/writes invalid or causing invariant violations.



```
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
```

We adopted a trace-based simulation and generated test cases with scenarios simulating any number of randomized, concurrent RBT operations at each timestep, as well as the number of timesteps.


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
Set x's parent to BLACK
No
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)

Parallel Insertion Algorithm

1 Find a leaf to insert while setting flags hand-over-hand

2 Create a new node and flag the local area

3 Proceed with Red-Black fixup operations, moving up the local area if necessary

4 Clear flags in local area