A Faster Parallel Implementation of Lock Free Hash Set

Danny Balter

May 11, 2017

Judging

This is a temporary section to alert finalist judges to the most important points of my paper (Please ignore figure numbers, they were messed up by the addition of newer figures).

1. The summary gives a very good overview of what happened in this project
2. The hardest technical hurdles to overcome were (i) Minimizing impact on performance while maintaining thread safety, (ii) Testing to make sure everything worked in every case, and (iii) Implementing remove within my binary search tree implementation.

   (i) Although locking was somewhat trivial in the coarse grained and mid grained cases, assigning locks in the fine grained case was challenging. I implemented it using hand over hand locking as the program walked down the linked list, while also having a set of bucket locks allowing programs to safely read and modify the head pointers.

   (ii) Testing was a huge challenge. Often times I believed I had some working code, only to run it on a large test case and have it segfault a few minutes in. The testing process for a piece of code was as follows: First, test the code sequentially, printing out the hash set after each operation. Then, using multiple threads, have the program insert numbers, and then after all the inserts, remove the same numbers, printing the hash set at the end of each insert-delete action. This allowed me to check for correctness, as the list should have been empty at the end of that. Finally, I tested for segfaults by running my large test cases from the Section 5, and making sure the programs returned reasonable output.

   (iii) Implementing remove within my binary search tree implementation was a particular challenge. I used the algorithm described here:

   `http://www.algolist.net/Data_structures/Binary_search_tree/Removal.`

   I plan to add diagrams and an in-depth explanation to my paper before Friday, as this was a very significant undertaking. This was particularly challenging because one must mark and atomically modify between 2 and 5 nodes at very different positions in the tree in order to preform a remove.
3. Summary Results are provided in the results section. The high contention case is my "wow-factor" graph, although all of the results are in line with what should be expected.

4. For Friday, you can expect a far more detailed graph breakdown of runtimes and speedups, including performance broken down to the individual functions (insert, search, remove). The graphs for Friday will be more nicely formatted and run extensively to guarantee perfect accuracy. More details about this can be found in my Test cases section (4.4)

Added Sample Result Graphs

These graphs are showing the runtimes of each implementation inserting 600,000 items into 300 buckets varying the number of threads between 1 and 24 (see two pages):

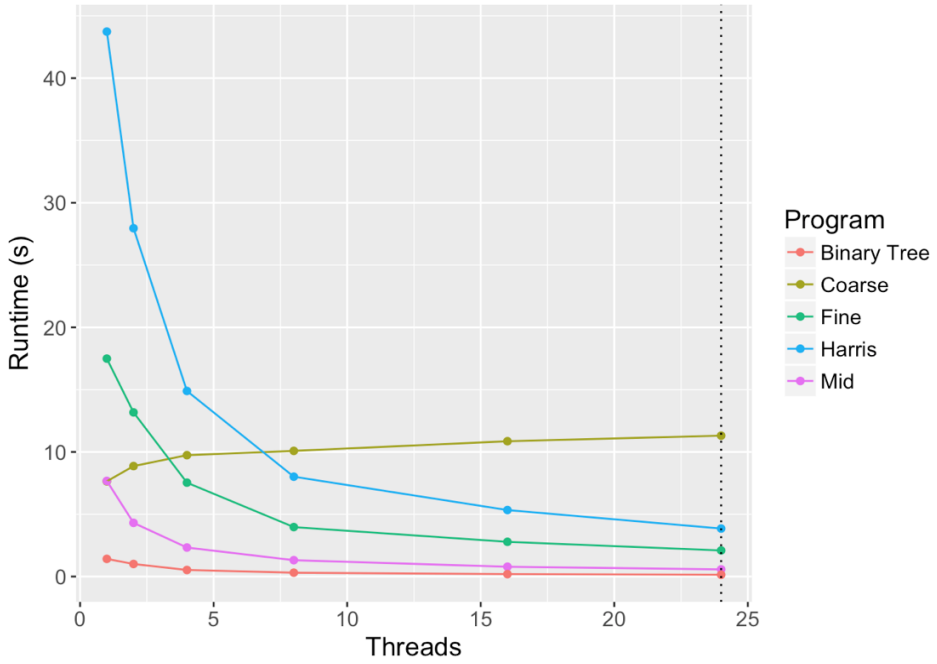

Figure 1: High Contention Insert Runtimes Linear

These graphs are a good example of what will be shown in my presentation on Friday. They allow us to understand where implementations succeed and fail, and which operations of insert, search, and remove are more and less expensive.

It is important to note that although the Harris implementation is the slowest, it has the overall largest speedup with more threads. This leads us to believe that with a machine that can handle more than 24 threads, and a big enough test case, the Harris implementation would surpass the locking implementations. Clearly the Binary Tree implementation blows everything else out of the water. This is unsurprising in a higher contention scenario, as with N nodes per bucket on average, an insert from any of the other four implementations could have to check N elements, whereas the Binary Tree implementation should only take log N comparisons on average.

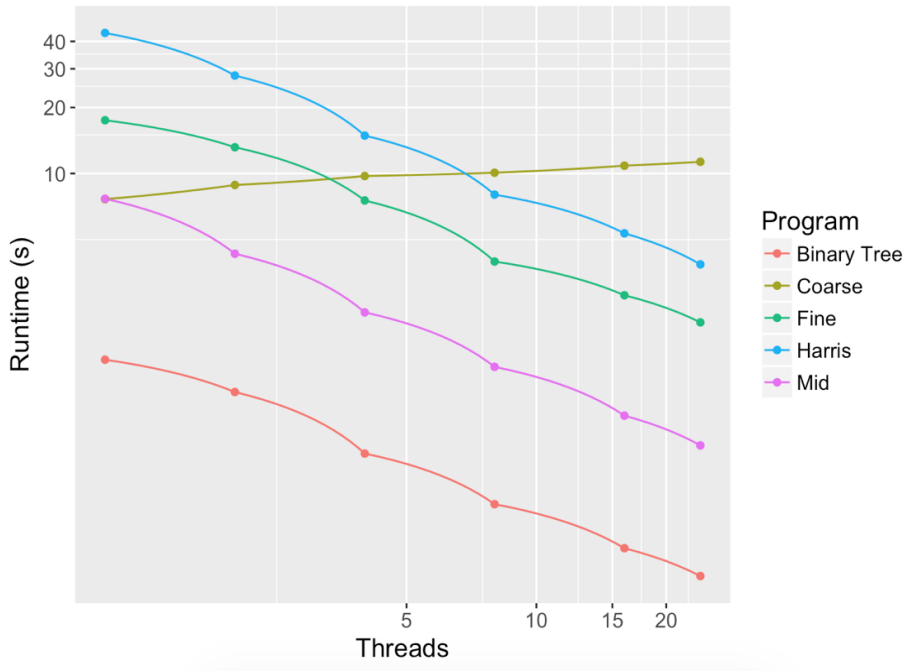

Figure 2: High Contention Insert Runtimes Logarithmic

Program	Threads	Ops	Buckets	NpB	Time
Coarse	1	600,000	300	2,000	7.641879
Coarse	2	600,000	300	2,000	8.863587
Coarse	4	600,000	300	2,000	9.740938
Coarse	8	600,000	300	2,000	10.085331
Coarse	16	600,000	300	2,000	10.859151
Coarse	24	600,000	300	2,000	11.308689
Mid	1	600,000	300	2,000	7.67528
Mid	2	600,000	300	2,000	4.309451
Mid	4	600,000	300	2,000	2.32912
Mid	8	600,000	300	2,000	1.313887
Mid	16	600,000	300	2,000	0.787489
Mid	24	600,000	300	2,000	0.576198
Fine	1	600,000	300	2,000	17.491097
Fine	2	600,000	300	2,000	13.1736
Fine	4	600,000	300	2,000	7.536257
Fine	8	600,000	300	2,000	3.971538
Fine	16	600,000	300	2,000	2.786848
Fine	24	600,000	300	2,000	2.095825
Harris	1	600,000	300	2,000	43.735207
Harris	2	600,000	300	2,000	27.953036
Harris	4	600,000	300	2,000	14.890948
Harris	8	600,000	300	2,000	8.015231
Harris	16	600,000	300	2,000	5.337746
Harris	24	600,000	300	2,000	3.852726
Binary Tree	1	600,000	300	2,000	1.416724
Binary Tree	2	600,000	300	2,000	1.008817
Binary Tree	4	600,000	300	2,000	0.529054
Binary Tree	8	600,000	300	2,000	0.310928
Binary Tree	16	600,000	300	2,000	0.195897
Binary Tree	24	600,000	300	2,000	0.146244

Figure 3: High Contention Insert Runtimes Raw Data

1 Summary

In this paper, we explore 3 parallel lock-based implementations of Hash Set, including a coarse-grained Global Lock, a mid-grained Bucket Lock, and a fine-grained Node Lock. We compare these with our implementation of Harris' 2001 Lock-free algorithm. We find that while the Harris Lock-free algorithm improves upon the lock-based implementations in certain cases, the overall runtime improvement is not substantial. This leads us to create our own Lock-free algorithm, based on Binary Trees, which improves upon the Harris implementation by more than a factor of 10 under high-contention settings. All our implementations were benchmarked on the LateDays system (described in the testing section) and involve insert, remove, and search operations.

2 Background on Hash Set Implementations

2.1 What is a Hash Set?

A hash set is an implementation of set that achieves O(1) cost on its primary functions (insert, remove, and lookup) by hashing the values. It's basic structure is shown in Figure 1, below.

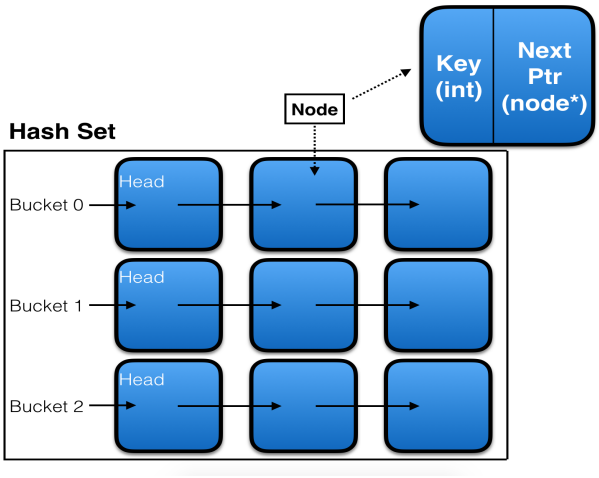

Figure 4: Diagram of a Basic Hash Set

2.2 Supported Functions

A hash set supports three primary functions, insert, remove, and search, all three of which take a key as a parameter.

Insert adds an element to the hash set by hashing the key, then inserting a node with that key into the corresponding bucket, following some inserting invariant (ours was sorted to be increasing).

Remove hashes the key, locates the correct bucket, then goes through the linked list of elements in that bucket until it finds the one with the key it is attempting to remove. That element is then removed without corrupting the linked list.

Search locates the node in question in the same method as remove, but when the element is located it returns 1, and if the element is not found, it returns 0.

2.3 Why Parallel?

The benefits of parallelizing a hash set are undeniable. When attempting to insert, remove, and search for millions of elements, having multiple threads doing work simultaneously provides an obvious speedup. However, with all multi-threaded applications, it is important to make sure your data structure is safe and that different threads attempting to access it at once won't cause race conditions.

2.4 The Benefits of Locking

The standard way to create thread-safe, multi-threaded programs is to use locking. Locks work by only allowing only a single thread to hold a given lock and access the locked code, and blocking all other threads who try to access the locked piece of code until they gain control of the lock. However, locks are not a perfect solution: (i) The code within the locked blocks must be done sequentially, and (ii) if the CPU swaps out a thread holding a lock for another thread, the lock is maintained and no other threads can access the locked code.

2.5 Why is Lock Free Preferable?

Lock free implementations are preferable for thread safe applications because they do not suffer from the second problem listed above. Instead of having threads hold physical locks, lock free implementations use atomic operations to make sure only one thread is performing a given operation at a time. This means that if a thread is swapped out, it will not still be blocking other threads from making progress.

Atomic operations are incredibly useful in creating thread safe applications. By definition, atomic operations can only be done by a single thread at one time, which means that wrapping an unsafe operation in an atomic function can make it thread safe. For example, we used:

```
__sync_bool_compare_and_swap
```

a function from the standard library which allows us to atomically attempt change the value in a given field. If it succeeds it returns 1, and if it doesn't succeed it returns 0.

3 Algorithms and Implementation

In this section we will cover the six implementations of hash set that I wrote for this project. The first implementation (3.1) is fully sequential and not thread safe. The following three implementations (3.2-3.4) are thread safe and use locking of different granularity (all locks are mutexes), and the final two implementations (3.5-3.6) are thread safe and lock free. The implementation in (3.5) is due to Harris et al. *reference*, while the implementation in (3.6) is my own. Additionally, the first five implementations (3.1-3.5) all use the hash set structure defined in figure 1, whereas the final implementation (3.6) uses a different hash set structure in order to increase performance. Finally, throughout every implementation, we hold the invariant that each bucket (linked list) within our hash set is sorted in increasing order. This allows us to search through fewer nodes in high contention situations.

3.1 Sequential Hash Set

Algorithm: Nodes are implemented as a struct that contains a key and a pointer to the next node (Fig. 2). Insert, Remove, and Search were implemented in a very standard way, where the program walks down the appropriate bucket (linked list) until the correct location is found and the operation is performed.

Discussion: This is a serial algorithm, which has the obvious disadvantage of only being able to do one operation at a time, and requiring one operation to finish before another can begin. As we'll see in Section 5, this leads to the sequential hash set being incredibly slow.

```
struct node {
  int key;
  node* next;
};
```

Figure 5: Standard Node Structure

3.2 Coarse Grained Locking

Algorithm: This implementation included a global lock, and whenever a thread attempted to perform an insert, remove, or search, it would take the global lock. When it finished the operation, the lock would be released.

Discussion: This implementation suffers from the same problem as the sequential hash set implementation, in that only one operation can be performed at a given time. This shows that a super coarse grained lock is not useful, as only one thread will be making progress at a time, despite spawning many more.

3.3 Mid Grained Locking

Algorithm: This implementation included an array of locks, with size equal to the number of buckets in our hash set. Then, whenever a thread attempted to perform an operation, it would first have to take the lock of the bucket in which it was trying to insert, remove, or search. As in the coarse case, when the operation finished, the lock would be released.

Discussion: Having a lock for each bucket drastically improves parallelism. Bucket locks allow for operations to be performed across many buckets simultaneously, meaning we could, given enough threads and a powerful enough computer, simultaneously perform an operation on every single bucket at once. This implementation does however suffer in high contention scenarios, where many of the operations are performed on the same bucket.

3.4 Fine Grained Locking

Algorithm: Fine grained locking required a slightly different node set-up. Each node was now allocated with a mutex as one of its fields. Then, whenever an operation encountered a node, it would first take the lock on the node, then modify or read the node, and finally move on (either to another node or return) and unlock the node.

Discussion: Although this implementation creates far more locks than the previous two, it has a distinct benefit over them. Much like in mid grained locking, operations can be done on multiple buckets simultaneously, but fine grained locking also allows us to do multiple operations within a single bucket at a given time. Because we only need to hold locks on 3 total nodes to insert or remove, and 2 nodes to search, multiple threads can be operating within the same bucket simultaneously, giving us additional opportunities for parallelism. It is worth noting that in low contention situations, this is a very minor improvement over mid grained locking, but in high contention situations (long bucket lists), fine grained locking allows for substantially more parallelism.

3.5 Harris Lock Free

Algorithm: The algorithm for this implementation is based on the lock free hash set implementation of Harris et al. [ref]. Harris reasoned that locking nodes can potentially lead to large slowdowns if a thread is paused by the CPU. Instead, Harris proposed "marking" the nodes that we intend to change (with a market bit), signaling to other threads that nobody else should attempt to take access, and then attempting to perform an atomic operation to modify the node in question. By doing this, we avoid the slowdown

of locks, while still guaranteeing that only one thread is modifying a given node at a time. Additionally, by creating an efficient marking and compare and swapping class (Fig. 3), I was able to substantially simplify the algorithm, while maintaining the same, or better performance as the originally proposed algorithm from Harris.

Discussion: Although this method doesn't reduce contention, it does reduce the degradation caused by contention. Therefore, in high contention scenarios, we expect this lock free algorithm to be superior to the locking algorithms described above.

```
class MarkedNode {

  public:
    bool AcquireMark(node** nd) {
      node* curr = *nd;
      if(curr == NULL) return false;
      *temp = *curr;
      bool is_marked = temp->marked;
      temp->marked = true;
      while(is_marked ||
        !__sync_bool_compare_and_swap(
        nd,curr,temp.get())) {
        curr = *nd;
        *temp = *curr;
        is_marked = temp->marked;
        temp->marked = true;
      }
      Acquire(temp.release());
      temp.reset(curr);
      return true;
    }
    node* Transfer() {
      node* temp = marked;
      marked = NULL;
      return temp;
    }
    void Acquire(node* nd) {
      if(marked != NULL) {
        marked->marked = false;
      }
      marked = nd;
    }
    MarkedNode():temp(new node()) {}
    ~MarkedNode() {
      Acquire(NULL);
    }

  private:
    std::unique_ptr<node> temp;
    node* marked = NULL;
};
```

Figure 6: Marking and CAS Class

3.6 Binary Search Tree Lock Free

Motivation: The motivation to create this implementation came from the slightly lackluster results that the Harris Lock Free implementation delivered. Despite my best efforts to optimize the Harris algorithm, it was only performing marginally better than the

fine grained locking implementation in most cases. I realized that one of the major problems was the structure of the bucket being a linked list. Linked lists are inherently hard to parallelize because all the threads must walk in the same direction across the same nodes, leading to a lot of contention and thusly a slowdown. In order to combat this, I decided to implement a hash set using a different structure for my buckets that would better lend itself to parallelism.

Algorithm: The main insight here was the idea to make each of my buckets a binary search tree. Binary search trees lend themselves to parallelism quite well, as the further you go down the tree, the fewer threads you're competing with, thus lessening contention. My binary search trees were implemented in the simplest way possible, containing only a key field, a marker field, and two "next" pointers, called left and right. The tree maintained the ordering invariant of binary search trees, but notably, was not balanced. Another important implementation detail was the removal of nodes. This function was by far the most complicated and my implementation was done following the algorithm on: *reference*

Discussion: Although using binary search trees instead of linked lists makes each node slightly larger due to the addition of an extra pointer (now have left and right instead of next), the benefits are undeniable. Similarly to randomized quicksort, in expectation, our tree will be reasonably balanced, and even in the worst case, where keys are inserted in increasing or decreasing order, the hash set will still have identical performance to the linked list implementations from above. However, the place where the binary search tree really shines is in high contention scenarios. With many items in a bucket, the lookup time for a linked list becomes O(n) in the length of the bucket. However, a binary search tree will always have O(log n) lookup time, as long as it stays reasonably balanced, which ours will due to the remove function.

Challenges: *To Be Filled In*

4 Experimental Setup

4.1 Architecture

All testing and benchmarking was done using the latedays servers provided to us as 15-418 students. These machines each have two 6-core hyper threaded CPUs, giving each machine 24 execution contexts, i.e. the ability to simultaneously do work on 24 threads. The cores are each clocked at 2.4GHz.

4.2 Workload

4.3 Correctness Test Cases

The hash set implementations were tested for correctness in two ways. First, a large set of numbers was inserted in parallel, and then removed. At the end, the hash set was printed to confirm no elements remained in it.

Secondly, in order to check for other problems, the sets were run for more than an hour each, repeatedly inserting, searching for, and removing random numbers, in order to confirm that the tables did not have memory leaks, race conditions, or storage problems.

4.4 Performance Test Cases

There are two major situations we'd like to test. In an ideal world, the hash set would have significantly more buckets than hashed numbers, giving us the O(1) operations we desire. This will be our low contention state, and will be represented by inserting 6 million random elements into 30 million buckets, searching for those same 6 million elements, and then removing those same 6 million elements, varying the number of threads in between runs. This gives us an expectation of .2 elements per bucket. We will also test a high contention scenario, in which we will generate 600 thousand random elements, and perform the same operations as above, using 300 buckets, leaving us with 2000 elements per bucket (Note: high contention is slower so we had to decrease the number of elements). In these test cases we will compare the overall runtime of each implementation, as well as the individual speeds of Insert, Remove, and Search. Finally, we will compare the time per Insert, Search, or Remove as a function of average bucket length for each of our implementations.

5 Results and Analysis

Please note these results are only preliminary and will not be identical to the results presented on Friday. They will be reformatted for clarity (more axis ticks, better labelling), tests repeated more for further accuracy, and not all of them will be in the final paper. They are, however, an accurate representation of the current performance of my implementations.

I also did not include the sequential or coarse grained locking implementation, as they are incredibly slow and not particularly relevant to the results I'm showing here. The baseline for a given implementation will be it run with only 1 thread.

5.1 Low Contention

Six million inserts, lookups, and removes on a thirty million bucket set. Please note the dotted line represents the test on 24 threads, the maximum amount the machine can execute simultaneously.

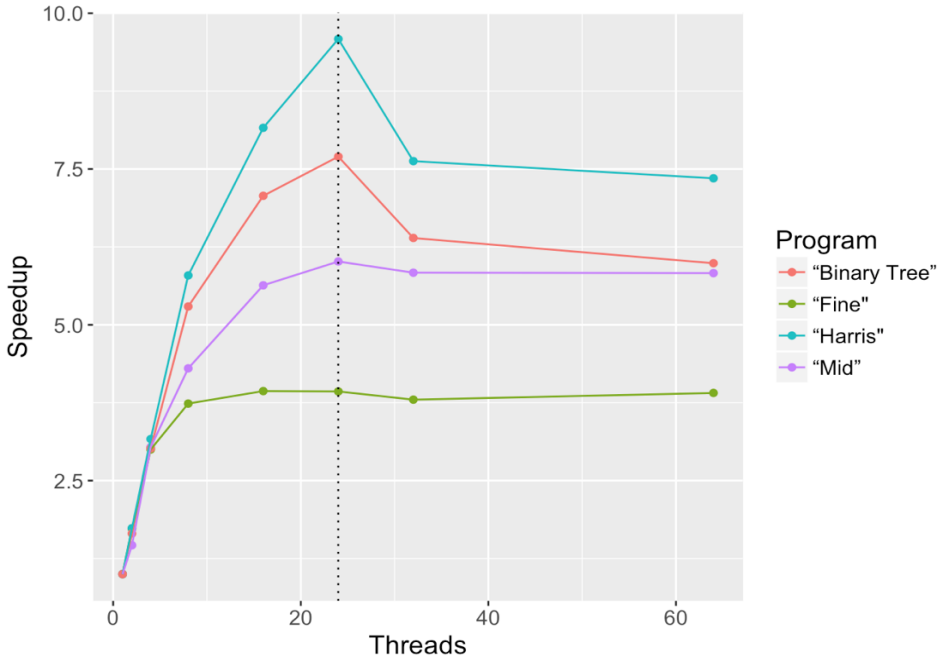

Figure 7: Low Contention Speedup Vs Threads (Linear Scale)

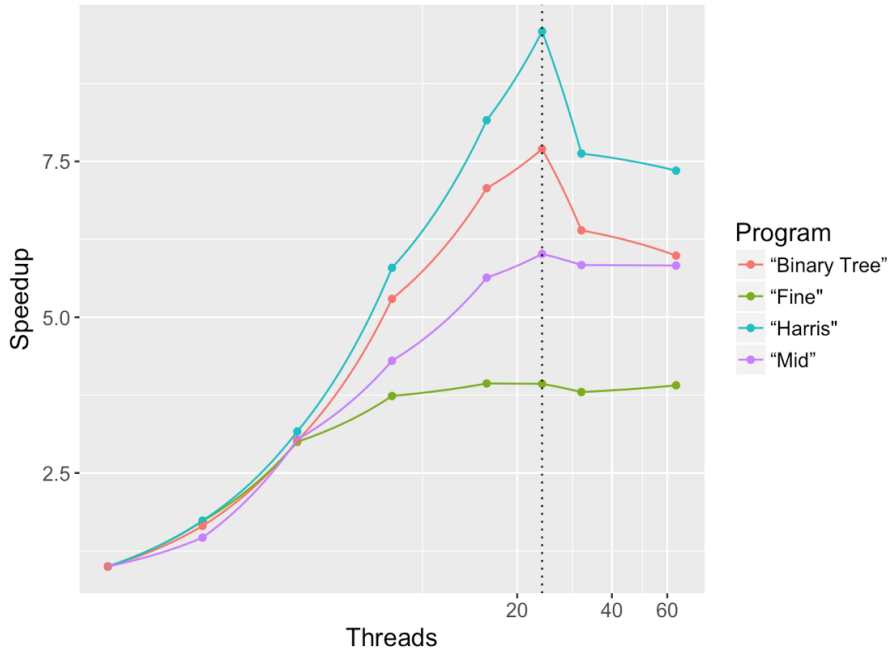

Figure 8: Low Contention Speedup Vs Threads (Log Scale)

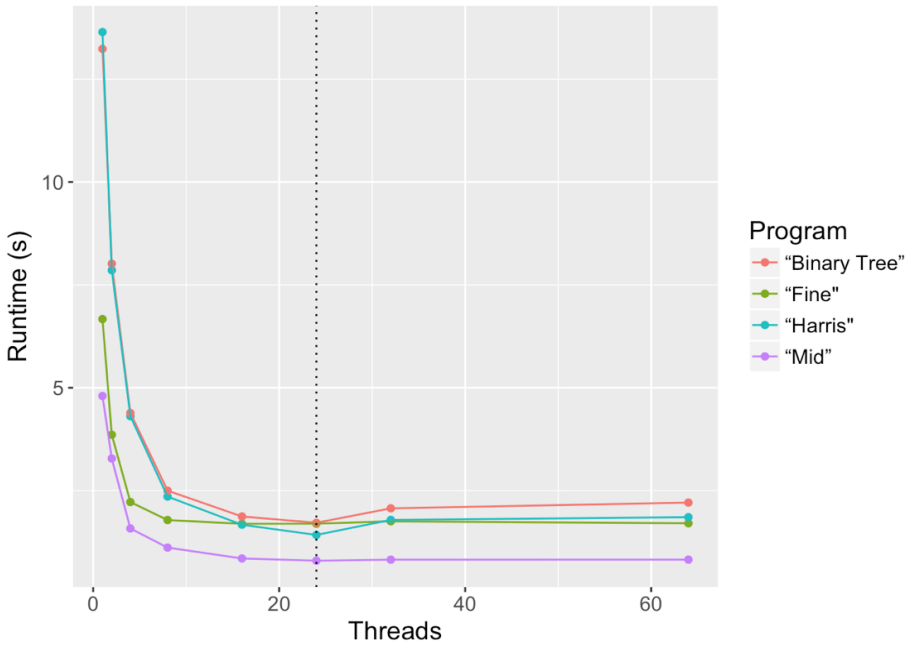

Figure 9: Low Contention Runtime Vs Threads (Linear Scale)

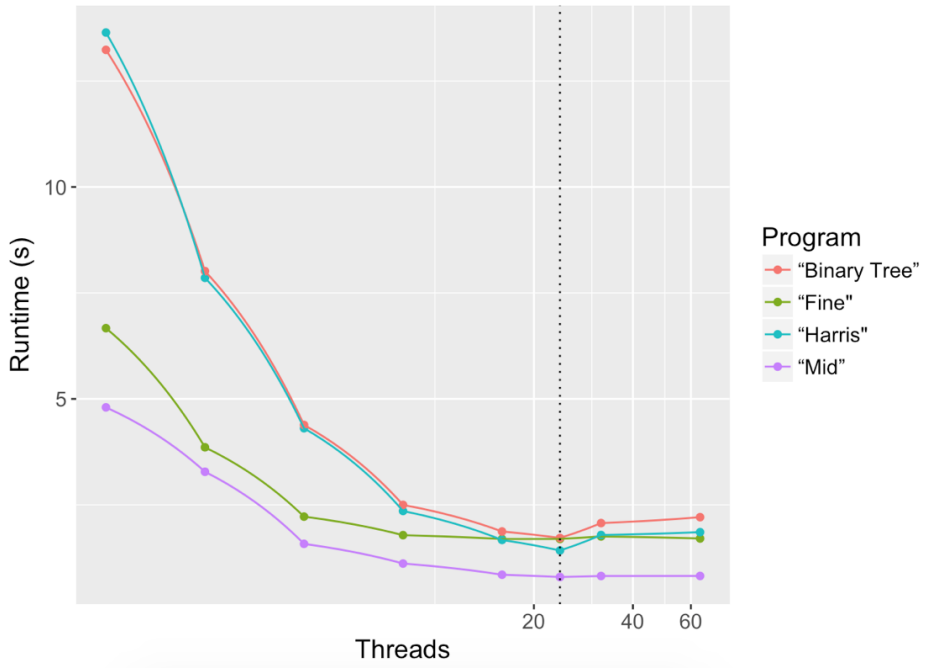

Figure 10: Low Contention Runtime Vs Threads (Log Scale)

There are a few important things to note about these graphs. In low contention situations, such as this one, most thread safe algorithms will preform similarly. This is because there is very little competition for locks or markers, so most threads can execute their work without waiting or being stalled by another thread in any way. This explains the relatively similar runtimes we see throughout this test case.

It is interesting to note that the two lock-free implementations had longer runtimes across the board, and especially in single threaded cases. This is due to the fact that marking and compare and swapping is a more expensive operation than taking and releasing a lock (when there isn't contention). In a state with contention, the marking and compare and swapping preforms better (as we'll see in the high contention test). In a scenario with low overall contention, there is little competition for locks, so there is a very minor amount of waiting for locks to make progress. As the number of threads increases, contention increases due to more operations attempting to be done simultaneously, and that's why we see the lock-free implementations catching up in higher thread count situations.

The tests cases were run up to 64 threads in order to test the performance of the algorithms when threads were being swapped in and out by the CPU. The results are not particularly interesting in this case, but in the high contention case, we see an interesting, albiet expected, pattern.

Another interesting note is that the two lock-free implementations saw a much greater increase in speedup from additional threads. This, again is because the marking and compare and swapping operation is much more efficient than trying to gain access to a lock in higher

contention situations (more threads creates higher contention). This leads me to believe that on a machine with a higher number of execution contexts, we would eventually see the lock-free implementations become faster than the locking implementations, as we increased the number of threads.

5.2 High Contention

Six hundred thousand inserts, lookups, and removes on a three hundred bucket set. Please note the dotted line represents the test on 24 threads, the maximum amount the machine can execute simultaneously.

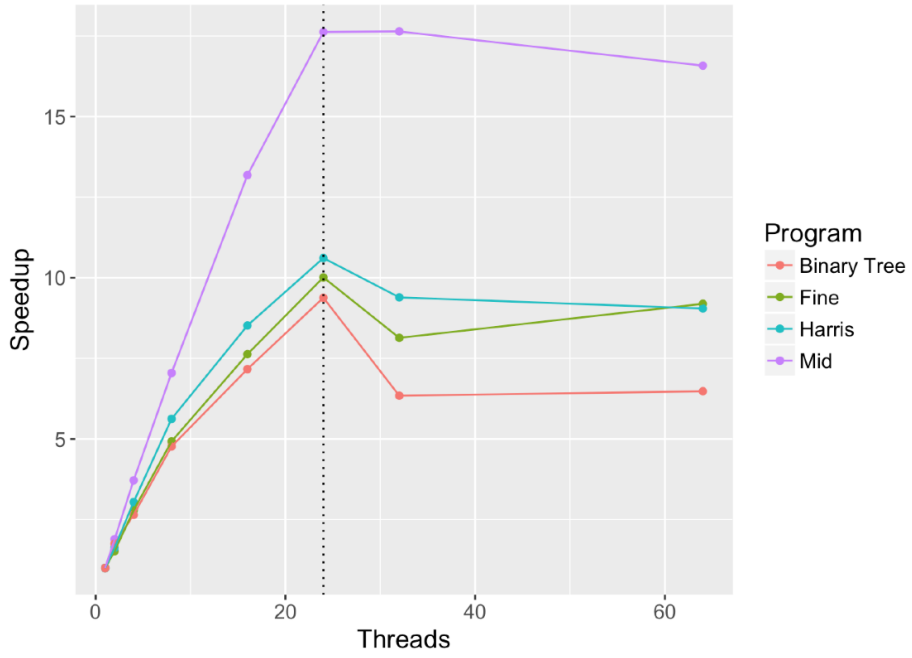

Figure 11: High Contention Speedup Vs Threads (Linear Scale)

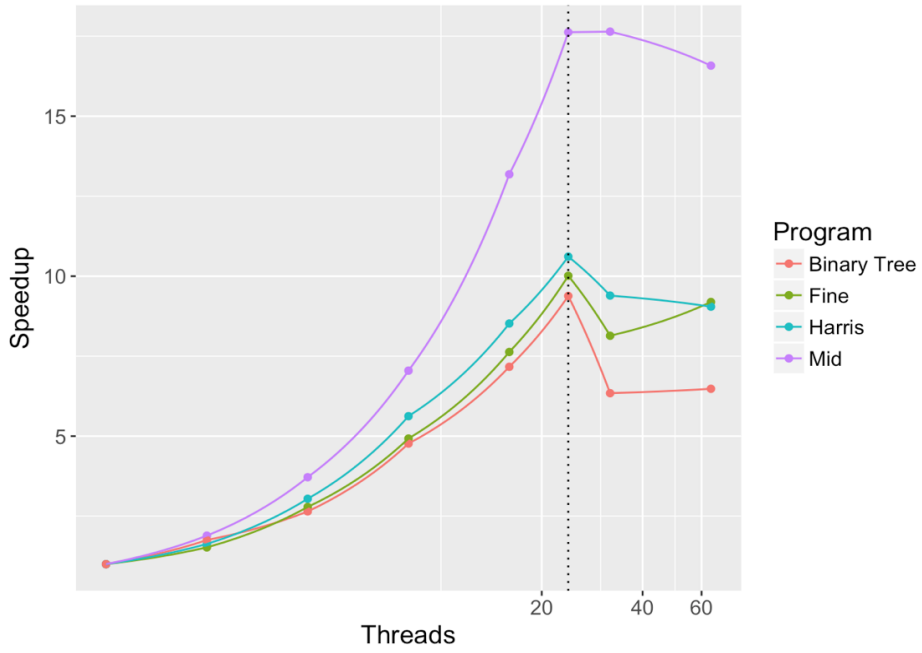

Figure 12: High Contention Speedup Vs Threads (Log Scale)

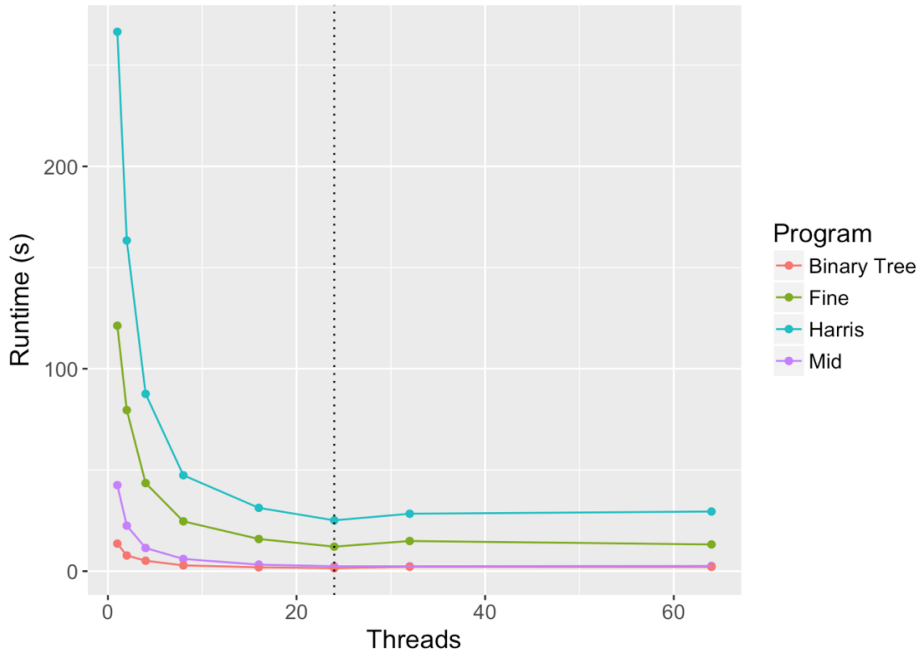

Figure 13: High Contention Runtime Vs Threads (Linear Scale)

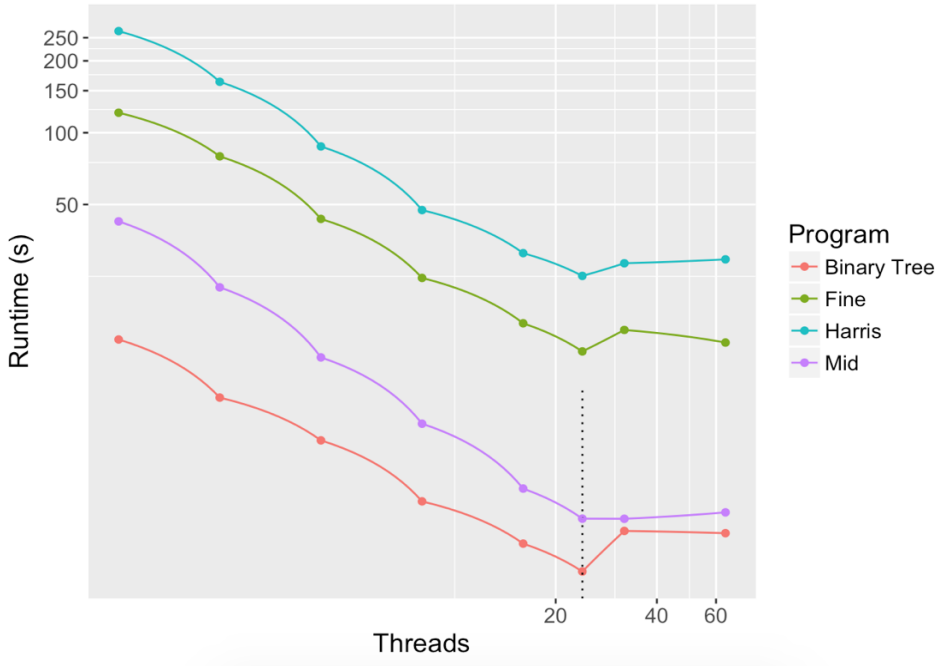

Figure 14: High Contention Runtime Vs Threads (Log Scale)

Clearly we can see that the binary tree is far better than any other implementation. It is substantially faster on every number of threads. One interesting note is that the Harris implementation was the slowest. This has a lot to do with the low number of total operations. Due to time constraints, I was only able to make graphs for 600k operations. However, running the same experiment on 6 million operations in the same 300 buckets leads to the same runtime for fine grained locking and the Harris lock-free implementation, showing the Harris implementation catching up as contention further increases. As contention increases further, the Binary Search Tree implementation also improves, surging even further ahead of the rest of the implementations in runtime. This is in line with what we'd expect as the Binary Search Tree implementation only has to search through O(log n) terms for a given operation, as opposed to O(n) terms for a linked list implementation, where n is the number of terms in the bucket.

6 Limitations and Future Improvements