A Faster Parallel Implementation of Lock Free Hash Set

Danny Balter

May 12, 2017

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 nock. 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 is exponentially faster than the Harris implementation in high-contention scenarios. All our implementations were benchmarked on the LateDays system (described in the testing section) and support thread safe 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 search) by hashing the values. Its basic structure is shown in Figure 1, below.

It is important to note that the reason a hash set maintains O(1) cost on its primary three operations is that each bucket of the hash set on average has very few elements. However, despite the O(1) function cost, high contention cases can still cause poor performance, as we'll see later.

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. For our hash set implementations, we maintain the invariant that each bucket is sorted in increasing order. This allows us to check fewer nodes when doing searches and inserts, as well as making it easier to check for correctness.

Insert adds an element to the hash set by hashing the key, then inserting a node with that key into the corresponding bucket.

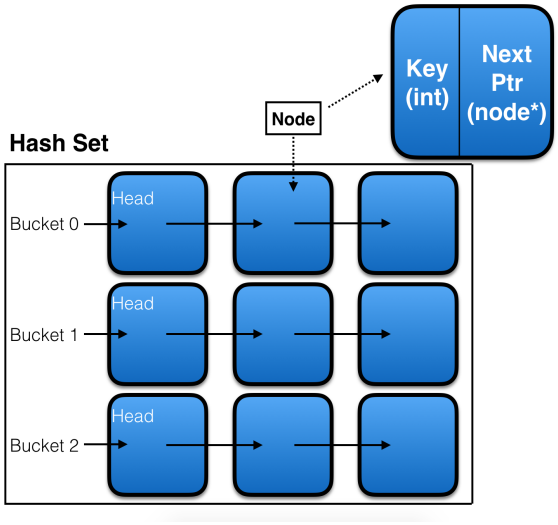

Figure 1: Diagram of a Basic Hash Set

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 true. If the element is not found, it returns false.

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 simultaneously won't corrupt your data.

2.4 The Benefits of Locking

The standard way to create thread-safe, multi-threaded programs is to use locking. Only a single thread is ever allowed to hold a lock at once, and any other threads that attempt to acquire the lock will have to wait until the lock becomes available again. Although locks are great for assuring thread safety, they suffer from two problems: (i) The code within locked blocks must be done sequentially and (ii) if the CPU swaps out a thread holding a lock or preforms a context switch, the lock is still held and no other threads can make progress. The second point can be a huge problem in high contention scenarios, as if a lock at the beginning of a bucket is being held, no other threads can progress into that bucket.

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 "marking" and atomic operations to assure that only one thread is modifying a given node at a time. The key idea behind the marks are that when the threads are swapped out of scope, the marks will also go out of scope and disappear, allowing other threads to make progress again. Additionally, we can use atomic operations in place of taking certain locks. 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 a slight modification of the implementation in Harris [1], 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, recall that the buckets within each hash set are sorted in increasing order to reduce the number of nodes we have to visit for search and remove.

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 2: 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.

Challenges: The biggest challenge with fine grained locking is making sure that when you return, you've already released all of the locks you took control of. If you don't you'll cause another thread to permanently stall, unable to make progress. In order to combat this, I wrote a class to make sure that a lock would be unlocked when I returned. This allowed me to focus on the algorithm instead of constantly checking for locking issues.

3.5 Harris Lock Free

Algorithm: The algorithm for this implementation is based on the lock free hash set implementation of Harris [1]. 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.

Challenges: The biggest challenge with Harris was implementing the lock-free algorithm. This was my first time writing lock-free thread safe code, so getting used to the marking and learning when a compare and swap could be used was somewhat difficult. Another big challenge was dealing with the A-B-A problem, which many lock-free programs suffer from. Thankfully, Harris did describe the algorithm for that in his paper, so I had some guidance.

```
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 3: 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 increased contention. 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 to write and my implementation was done following the standard removal algorithm for binary search trees.

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: The biggest challenge in this was the marking, especially in the remove case. Whereas in the Harris implementation, we never had more than three nodes locked concurrently, in the binary tree implementation, we can have up to five marked at a time. Additionally, I had to implement two separate instances of hand over hand marking, one while searching for the node to remove, and another to find the node to swap with it.

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 Correctness Test Cases

The first testing I did after confirming the implementations worked on small test cases was to test for correctness on a large scale. This was done 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 corruption and memory problems, the sets were run more than half an hour each, repeatedly inserting, searching for, and removing random numbers, making sure that the application didn't crash due to error at any point. It is important to note that I attempted to have an even number of removes and inserts here, in order to not overfill the hash set and cause unnecessary slowdowns.

4.3 Contention-like Metric

Before discussing our performance testing, we must first define a contention-like metric:

$$r = \text{MAX} \ (\ \#\text{Elements} \ / \ \text{Bucket}\ )$$

r represents the number of elements in the highest contention bucket. In a low contention scenario, r could be as low as 1, where as in a high contention scenario, r could be as high as 100,000. It is important to note that no matter what value r holds, the tests will still be set up to observe the hash set invariant that the average number of nodes per bucket is very small.

4.4 Performance Test Cases (Workload)

There are two important scenarios that we use to test our hash set implementations:

1. Low Contention: Section 5.1
2. High Contention: Section 5.2

Low contention is the ideal case for a hash set. This is set up by inserting 6 million random keys into a hash set with 30 million buckets. We then perform 6 million searches followed by 6 million removes. It is important to note that on average there are only .2 nodes per bucket in this case, and the probability of having a bucket with more than a handful of nodes is incredibly low. Thus we expect very little contention for locks or marks.

The high contention scenario is when we start seeing interesting behavior. In order to test this we set up an experiment where we insert nodes in approximately equal quantity to buckets. However, we have 99-percent of the inserts use keys that hash to the same bucket. Obviously, this creates a bucket with an incredibly high number of nodes and absolutely ruins our expected O(1) runtime operations.

In the low contention case, we will compare runtime against number of threads, along with speedup against number of threads. In doing this, we will be able to see which implementations are the fastest at low contention, as well as which implementations take the most advantage of parallelism.

In the high contention case, we will compare runtime against our contention-like metric r, as well as runtime against number of threads. This will allow us to see which implementation scales the best with high contention, as well as which implementation is the fastest.

5 Results and Analysis

5.1 Low Contention

Figure 4 shows a plot of runtime against the number of threads for the low contention setup described in section 4.4. We can see that for the most part, the two locking implementations are faster than the two lock free implementations. This makes sense, as the real problem with locks comes at higher contention when threads spend long periods of time waiting to acquire locks on which progress isn't being made. Additionally, when contention isn't causing slowdowns to locks, the atomic operations are more expensive than the lock acquisitions, and take longer.

One interesting thing to note is that the binary tree implementation, which one might expect to be the fastest, is actually the slowest here. This is because in very low contention situations, where there are very few nodes per bucket, there is no advantage to using a binary tree over a linked list, as you check approximately the same number of nodes either way. Additionally, the tree structure was slightly more complicated to set up, which causes the results we see.

Finally, note the vertical line at 24 threads. The latedays machines are only capable of doing work on 24 threads at one time. This means that beyond that threshold, threads are being spawned that must be swapped in and out in order to complete their work. Unsurprisingly, using further threads over 24 leads to a slowdown in every implementation. However, based on our earlier analysis, one would expect the lock free implementations to slow down more

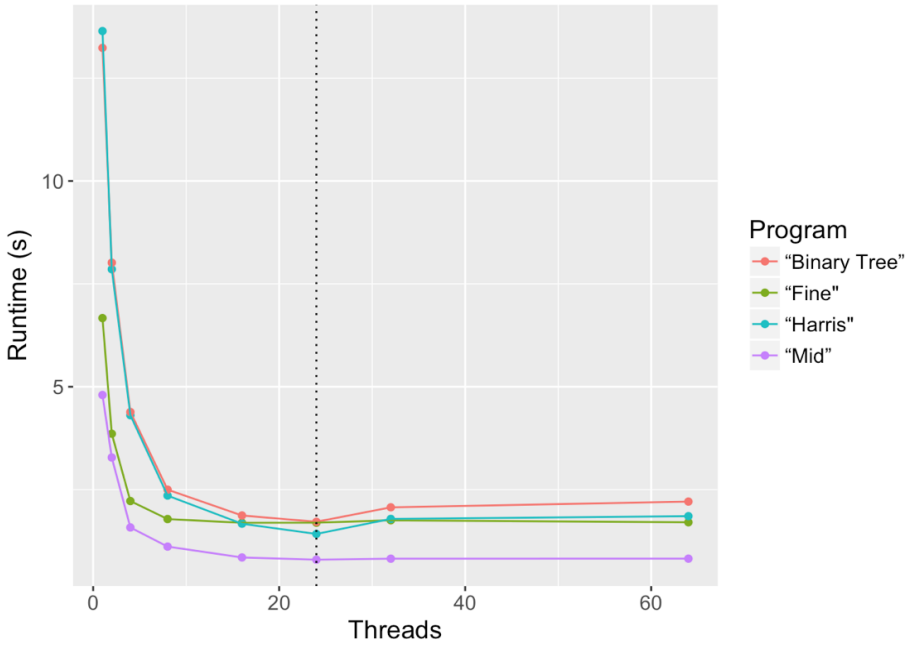

Figure 4: Low Contention Runtime Vs Threads (Linear Scale)

from the excess threads. However, because this is a low contention case, it is unlikely that a thread that was swapped out will be holding a lock that another thread needs. Thus there are no large slowdowns caused by the CPU thread swapping, leading to relatively even slowdowns across all implementations.

Figure 5 shows the speedup each implementation experienced over their own single threaded performance plotted against the number of threads. One interesting thing to notice is that although the lock-free implementations were slower than the locking implementations in Figure 5, we see that the lock-free implementations were able to take better advantage of more threads. The reason for this is that as more threads work simultaneously, whatever little contention exists will be increased, and in that situation the mark and compare and swap system will perform better than the locks. This leads me to believe that even in a very low contention situation, if we had a machine with an unlimited number of execution contexts (unlimited threads), the lock-free algorithms would eventually be faster than the locking algorithms.

The speedups we see in figure 5 do support the "catching up" we saw from the lock-free algorithms in figure 4. As more threads were added, the difference in runtime between the implementations shrunk, and that is supported by the fact that the slower implementations made better use of the extra threads than the faster ones.

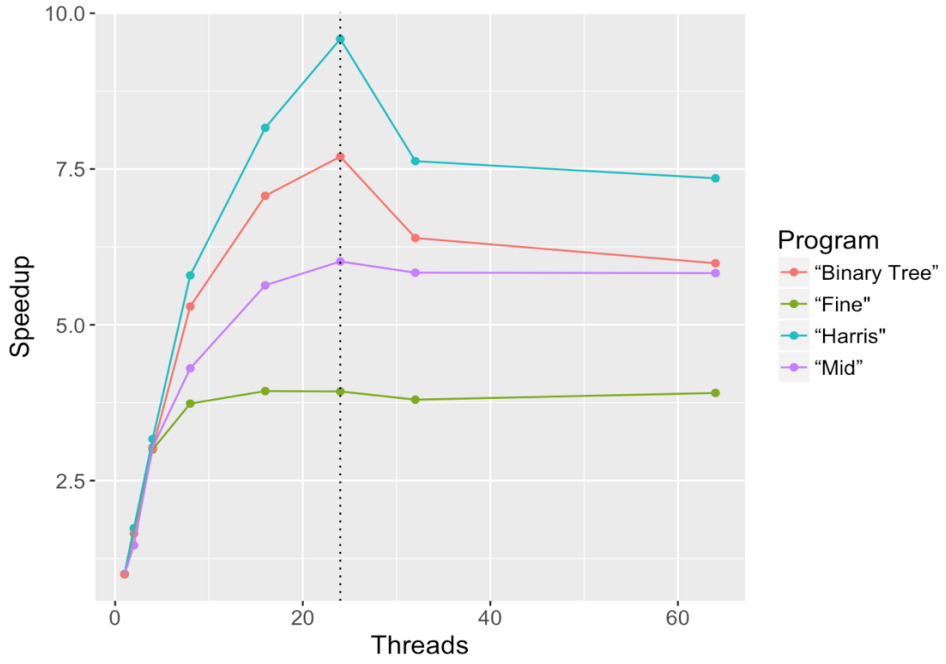

Figure 5: Low Contention Speedup Vs Threads (Linear Scale)

5.2 High Contention

The motivation for the high contention cases will come in the form of an example. One common use of a hash set is web servers to hash data they collect (in the interest of performance). It is very possible that due to a major news event, one individual link on a webpage would be clicked many times more than any other link, resulting in many hashes to the same bucket. This could easily create the high contention cases presented in figures 6 and 7.

Recall that the high contention scenario was created by inserting 99-percent of elements into the same bucket. It is important to know that due to high runtimes, the next two implementations are only inserts. I found through testing that remove and insert took similar amounts of time, with search taking less. Given how long some of the runs were taking, they were done with only inserts. The first case we see in Figure 6 has an r-value of 20,000:

The first thing that should jump out from this figure is that the Binary Tree performance went from being one of the worst performers to far and away the best. In fact, even when it looks like the mid-grained locking implementation is running almost as fast with 24 threads, it's still 4x slower than the Binary Tree. With 20,000 nodes in a single bucket, the performance gain of only searching through an average of log n elements per operation is absolutely massive. This performance advantage will become even more apparent when we look at super high r-value cases in figure 7.

Another surprising note here is that the Harris implementation is still the slowest. This is

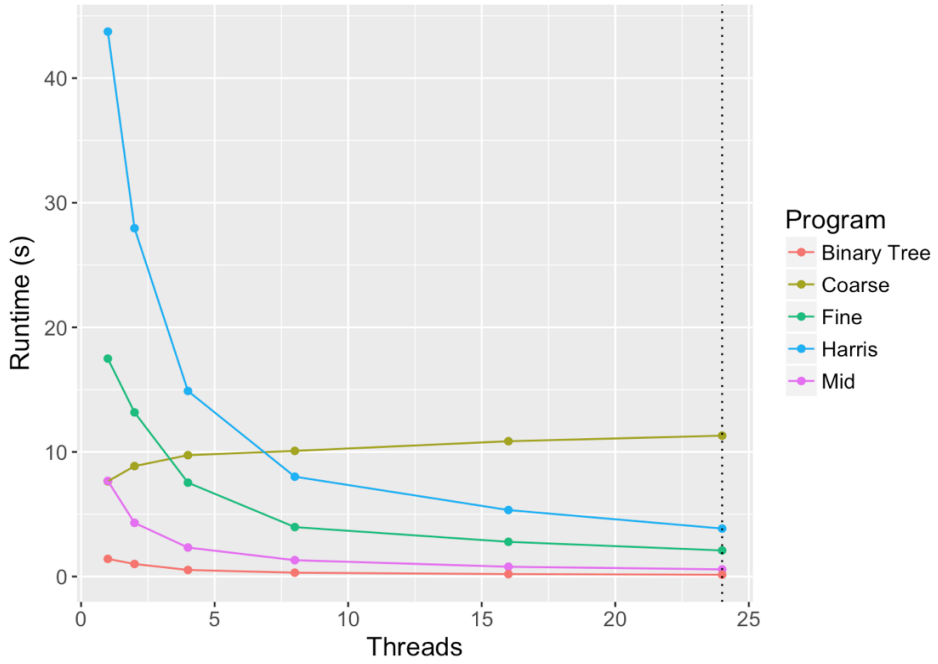

Figure 6: High Contention Runtimes vs Threads

simply due to the amount of contention not being high enough. As with the lower contention cases, we can see the Harris implementation catching up to the locking implementations as we increase the number of threads. If we increase the r-value or the number of threads, we will see the Harris implementation surpass the locking implementations.

Additionally, the fine grained locking implementation is slower than the mid grained locking implementation, which is the opposite of what we'd expect. The reason for this, is that although we have created contention, it is not enough to overcome the extra cost of taking a lock on every node as the implementation traverses the bucket list. Even though we can do multiple insertions into the same bucket simultaneously, the benefit is more than countered by the work of the extra locks. Similarly to the Harris implementation, we would expect our performance to approach the theory as contention or number of threads increased.

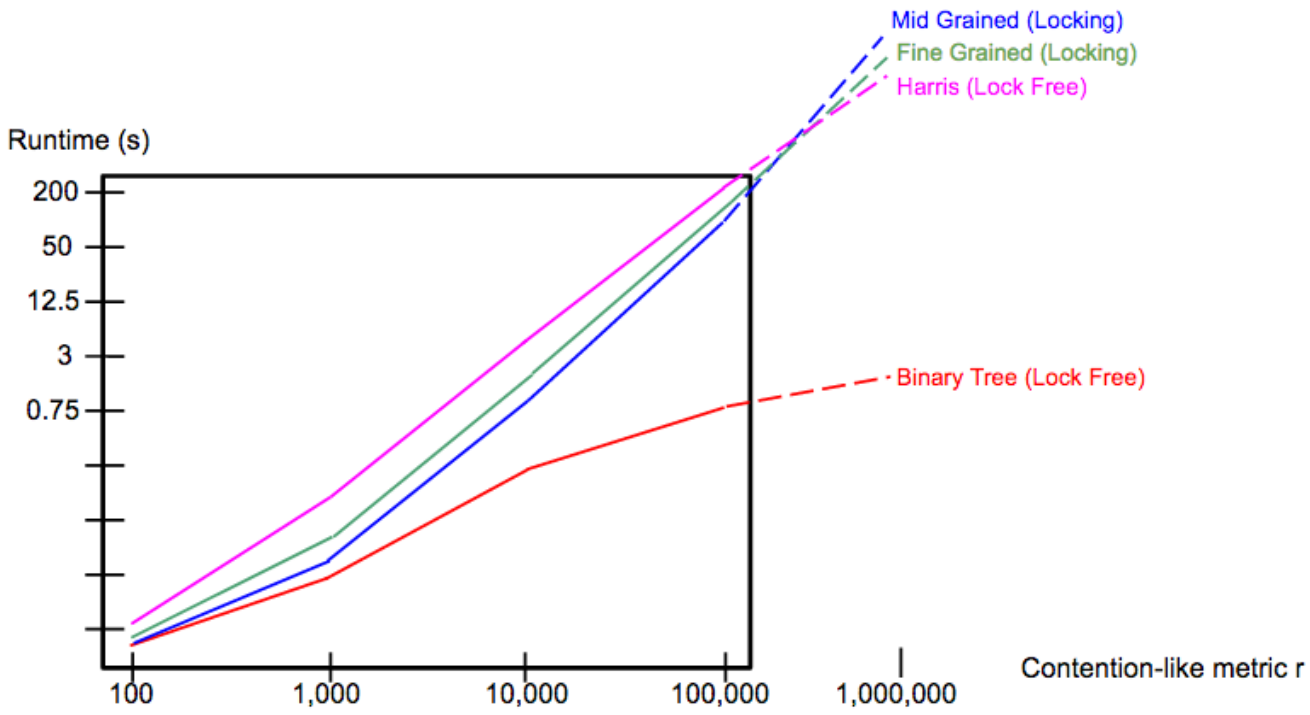

Figure 7: Runtime as a Function of Increasing Contention

Figure 7 shows runtime (in seconds), on a logarithmic scale, plotted against our contention-like metric r, also on a log scale. The high contention situation was created in the same way as it was for the figure 6 tests. There are a few important takeaways from this graph:

1. The binary tree implementation seems to be growing at a logarithmic pace as opposed to a linear pace like the rest of the implementations.

   This is not surprising. As was mentioned earlier, the binary tree implementation only has to check log n nodes on average per operation, and thusly, as r increases, the runtime of the binary tree implementation will increase logarithmically.

   This pattern of increasing logarithmically is why the binary tree implementation is exponentially faster than the other implementations we've discussed in this report.

2. As r increases to very high contention scenarios, Harris becomes the second best, as we would expect. Similarly, fine grained locking passes mid grained locking.

   This is in line with our expectations. As contention increases, so does the slowdown from locking. Eventually, Harris, a lock free implementation, will surpass any linked-list locking implementation.

   Similarly, when the length of a bucket gets long enough, the benefit of doing parallel inserts into a single bucket will surpass the excess cost of the locks. It is interesting to note, that when almost every single node is inserted into the same bucket, as is the case in very high contention cases, the mid grained locking implementation functions almost sequentially, as it can only insert one element at a time into the bucket where the majority of elements need to go.

One important note about this graph is that only the values within the black border were actually computed experimentally. Based on the trends from the data I had, I extrapolated how long each implementation would take for an r-value of 1,000,000. This reason for this was due to the increase in runtime, for the locking implementations, the next datapoint would have taken more than 5 hours to compute. The only implementation fast enough to experimentally do r=1,000,000 was the binary tree.

5.3 Take-Aways

There are three major take aways from this analysis of hash set performance:

1. In low contention cases, it makes very little difference which implementation you use. Locking is generally faster than lock-free, due to the higher cost of implementing a lock-free algorithm, but the performance gain is not particularly significant.

2. By contrast, at high, and especially very high, contention, using a lock-free implementation provides unparalleled gains in speed.

3. Regardless of whether one chooses a lock-free or locking implementation, choosing an intelligent data structure, such as a binary tree, can vastly improve performance and can even change the algorithmic complexity of operations.

6 Limitations and Future Improvements

There are a few limitations that this project suffered from. Firstly, the cluster our experiments were run on could only support 24 threads at once. Running our simulations on a cluster that could support a greater number of threads simultaneously would have made the benefits of a lock-free implementation even more clear.

Additionally, I would like to have built a locking binary tree implementation, to compare against the lock-free implementation I have in very high contention cases. The lock-free implementation should eventually be faster, but it would be interesting to see how much contention must be created for that to occur.

Given more time, I would have liked to run larger experiments, such as the r=1,000,000 case that was predicted in figure 7, to confirm that my predictions were in fact accurate.

References

[1] Timothy Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing.*, 2001.