Exploring Lock-Free Data Structures

Manoj Singireddy

Thursday, April 20

1 Motivation and Description

While learning concurrency, we focus heavily on using mutexes, semaphores, barriers, and other synchronization primitives to ensure program correctness and avoid race conditions. However, this may not always be the best strategy if we have a particularly adversarial thread scheduling algorithm. This situation introduces the motivation for Lock-Free Programming. In this paradigm, the programmer writes concurrent code and protects critical sections without using a blocking primitive such as a mutex or semaphore. There are several approaches to this programming method, but many key themes involve using atomic operations.

For this project, I planned to implement several lock-free algorithms. Algorithms such as these are often implemented by making the relevant data structures lock-free, so I decided to switch my project to implement a few lock-free data structures, including a Stack, HashTable, and a Binary Search Tree (BST). I decided to implement them in order of increasing difficulty and interest from Stack, to HashTable, to BST. For this project, I wanted to explore how this programming paradigm works and gain experience with it. Prior to performing this project, I hypothesized that lock-free data structures would not perform as well as lock-based data structures despite all their fancy implementation details. As will be seen in this report, I was extremely wrong.

2 Infrastructure and Hardware

I built my implementation on my computer, an Apple M1 Macbook Pro running MacOS Ventura 13.2.1. My program was be compiled using the standard g++ compiler, available on my machine as the Clang compiler. For my experiments, I made a Python file that passed various input arguments to my program to express the differences between the different implementations. This Python script also recorded the results of the experimental timings and prepared them for graphical representation.

3 Stack

The stack implementation was by far the easiest part of the project. I started by first implementing a lock-based stack using a mutex to use as a base to implement the lock-free stack and also to have to benchmark against. It works like a traditional stack with a constructor, destructor, push, and pop. However, it also has a mutex, which protects access to stack nodes.

Next, I decided to implement the lock-free push and pop methods. These lock-free implementations rely heavily on the compare and swap (CAS) or compare and exchange methods (called in C++). This is called on an atomic variable and takes in an input of two pointers. The first pointer contains the *expected* value, and the second pointer contains the *new* value. The instruction atomically checks if the value in the atomic variable matches the *expected* value and then replaces it with the *new* value if it does match and does nothing if it does not match. It also returns true if the swap was successful and false if it was unsuccessful. Furthermore, the current value of the atomic variable gets placed in the *expected* variable. A weak and strong variant of the function exists, which perform slightly differently in different use cases. The weak variant fails after trying once, while the strong variant attempts to try in a loop. So depending on if the CAS instruction is already in a loop, you might choose one variant over the other.

For the stack, I chose to use the weak variant. For the pop method, I continually attempted to remove the top of the stack using CAS until success; then, I returned the value. For the push method, I continually attempted to add the new element to the top and set it as the top while marking the next variable as the current node. These strategies are effective at achieving the desired functionality.

Below I've included some graphs that show the stack's performance across a series of read (pop) and write (push) operations. The speedup is graphed by dividing the lock-based stack's performance by the lock-free stack's performance. The first graph scales the number of threads used to conduct a fixed number of parallel operations, while the second scales the number of operations across a fixed number of threads. When not being scaled, each parameter is held at a value of 32. Furthermore, I ran the tests with varying read and write operations proportions. Each thread ran the decided-upon number of operations and used a probability distribution to choose which operation to execute in each iteration.

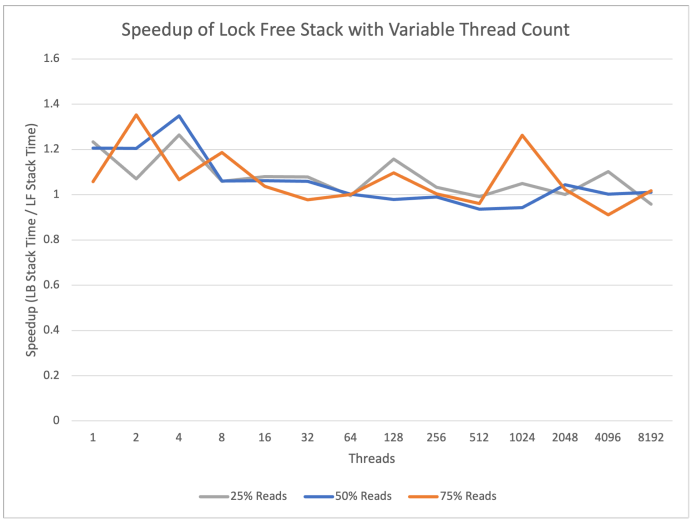

Figure 1: Speedup of Lock-Free Stack with Thread Scaling

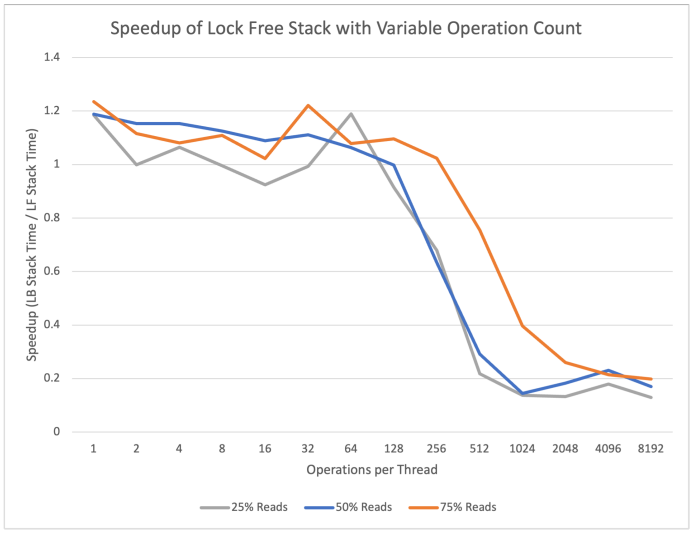

Figure 2: Speedup of Lock-Free Stack with Operation Scaling

Let's first examine *Figure 1*. We can see that initially, at lower thread counts, the speedup is small across the three different ratios. If we continue along the thread scaling, the performance settles at around/slightly above 1. As we scale threads, we do not experience significant differences from the lock-free behavior. However, we do experience benefit at 1024 threads. It's likely this number is friendly to the cache in some manner, which allows for much faster reads in the

lock-free implementation.

Let's now take a look at the operation scaling. We can see from this graph that with a low number of operations, the speedup hovers around one and reaches upward towards 1.2. In particular, it's important to note the peak at 32 threads in the 75% reads, which may be due to cache alignment that is friendly to more reads. After this peak, both 25% and 50% follow very similar trajectories, while the 75% case remains at a greater speedup. All three lines do move downwards towards the greater number of operations, which indicates that the operations per thread scaling is not very favorable for the lock-free stack algorithm. This is likely due to the overhead for acquiring the lock and releasing it being far less across fewer threads. It's important to consider how cache invalidations play a role in this as the mutex is shared across the same number of threads despite the operations continuing to increase. Thus, the performance of the lock-based stack would improve while the performance of the lock-free algorithm would decline due to more frequent contention on the atomic CAS instructions.

Overall, we can see that a lock-free implementation for stacks performs decently well as we scale the number of threads. At the very least, we do not see major performance hits. However, as we increase the number of operations per thread (simulates work per thread), we experience massive declines in performance. This indicates that lock-free stacks might not be a good implementation decision for most use cases unless there is a very specific use case that is particularly friendly to lock-free algorithms due to cache friendliness or other reasons.

4 HashTable

The next data structure I implemented was a hash table. A hash table is a powerful data structure that stores and retrieves values based on unique keys. It uses hashing to generate an index based on the key and map it to a specific location within the table. This process ensures quick access to the stored data, as the search time is generally constant, regardless of the number of elements in the table. When multiple keys produce the same hash index, a collision occurs. Two main strategies are used to resolve collisions: chaining and linear probing.

I started the process by implementing a lock-based hash table to base my lock-free implementation on and to use for benchmarking. It's a very standard chaining-based hash table implementation that uses a mutex to protect the critical section of each method. I then moved on to the lock-free implementation, which once again relied heavily on the atomic CAS instruction. I referenced a paper by Maged M. Michael [2]. I used the implementation description and details given to inspire my implementation of the lock-free hash table. One technique the author used that was particularly useful was chaining. This technique involves storing a linked list at each hash index and storing all hash-collided items within this linked list. This prevents us from having to un-

dergo many of the complications involved with a linear probing approach where collided values are stored at sequential indices within the table.

The approach to the insert method involved first creating a new node. Then, I iterated through the linked list corresponding to the hash I needed to check based on the key. If I found a matching key, I knew the value was already in the hash table, so I return early. I also attempt to add the new node to the top of the linked list, similar to the methodology from the stack push.

The get method does not require a CAS instruction because it iterates through the list corresponding to the hash of the given key and simply returns a value if it finds one. We can implement this strategy because any interleaving of get's and inserts is valid because get should return the correct value based on when it goes through the list. It does need to concern itself with inserts that occur after it's already checked a node in the list.

The remove method works similarly to the pop method for the stack by finding the correct list and then taking off the node corresponding to the correct key.

Below I've included some graphs that show the HashTable's performance across a series of read (get) and write (insert) operations. These graphs were generated using the same methodology as those previously shown. It's also important to note that random key values were inserted into the graphs to ensure that hash collisions would occur occasionally.

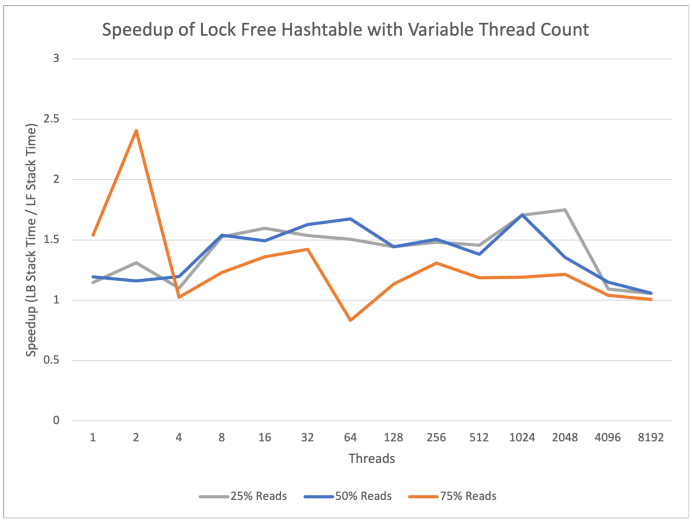

Figure 3: Speedup of Lock-Free Hash Table with Thread Scaling

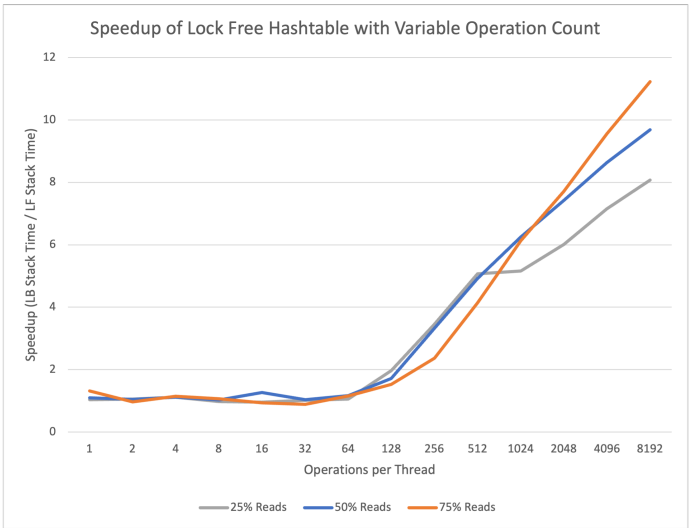

Figure 4: Speedup of Lock-Free Hash Table with Operation Scaling

Let's take a look at the first graph. As we can see we mostly hover at a speedup that is greater than one. This is a good sign for the performance for the lock free stack. in the 0.75% case we have peak at 2 threads. This is likely due to the fact that with a lock-free data structure we probably lined up reads and writes really well to promote optimal speed. The remainder of the graph is fairly uniform. It looks like for a greater number of threads if we have more writes then we experience greater benefit in speedup.

The second graph is far more interesting. As stated before, we hold the number of threads constant at 32, while we increase the number of operations per thread. In this model, we can see that speedup increases greatly as we increase the number of operations per thread. This is likely due to the lack of contention between multiple threads. Also, we only need our CAS structure on colliding hashes. Non-colliding hashes will not interfere with each other at all with parallel reads and write. Thus, with a large number of operations, we see massive benefits in speedup. Furthermore, due to this parallelism, we also experience very similar trajectories for various read/write ratios. It seems due to the parallelism it does not matter much, which instructions are happening in parallel.

Overall, we can see that a lock-free approach to a hash table benefits performance. Though we do not experience massive benefits from thread scaling, we do see that with thread scaling, we still have a speedup that is greater than one, which indicates that the lock-free method is faster. However, we see the true benefit of an increase in the number of operations. In cases where the hash table needs to be used repeatedly in a parallel fashion, it could be very beneficial to use a lock-free hash table such as the one I implemented.

5 Binary Search Tree

The final data structure I decided to implement was a BST. The implementation of the insert, remove and get methods are trivial for a lock-based approach. For the insert method, we lock the entire method while we find the appropriate location for the key in the tree and insert it. For the get method, we simply traverse the tree for our target node. And finally, for the remove method, we remove the node and replace it with its children appropriately based on the children's keys.

The real challenge comes with implementing the lock-free algorithm. Several algorithms are out there for implementing this data structure, and I chose to take inspiration from a paper by Nathan G. Bronson and others [1]. For this implementation, the inspiration was heavy, and I outlined my solution as described in the paper. To avoid copying and pasting the paper (as I believe they did a better job describing the algorithm than I could), I will briefly describe the algorithm and focus my discussion on the performance. The implementation uses various helper methods to attempt to get, remove, and insert, which can be retried frequently. The get, remove, and insert methods bottle up their actions into a data structure that indicates what they want to achieve. Then they each iterate while attempting to complete their action via a CAS. The code for this section got tricky, and some of the challenges will be explained further in the challenges section.

Below I've included some graphs that show the BST's performance across a series of read (get) and write (insert) operations. These graphs were generated using the same methodology as those previously shown.

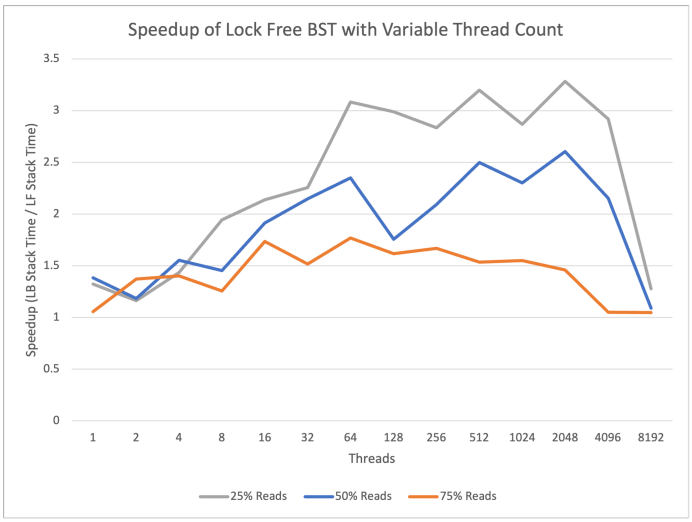

Figure 5: Speedup of Lock-Free BST with Thread Scaling

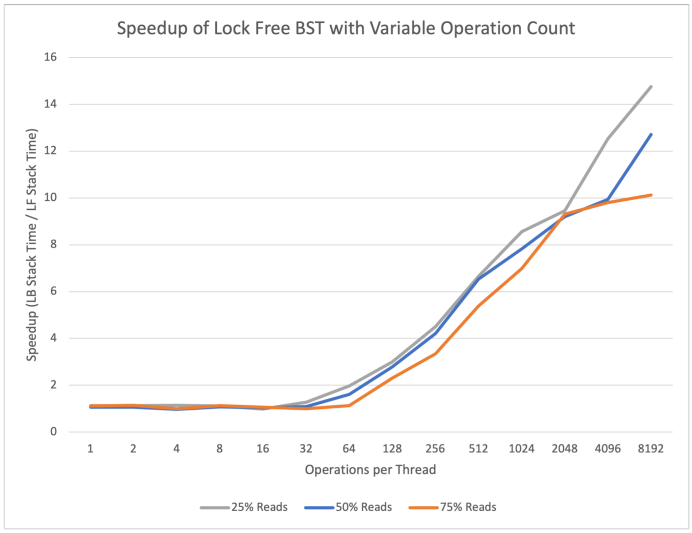

Figure 6: Speedup of Lock-Free BST with Operation Scaling

Let's first take a look at *Figure 5*. Out of all three data structures, there is the most difference between this graph's read/write ratios. It also has a very interesting trajectory across the thread scaling. The speedup is above one across all the thread scaling. We see increasing speedup across fewer threads until we level out at 64-2048 threads. It's important to note the difference in the read/write ratios here. This is likely because we experience much better speedup with the

lock-free writes than with the lock-free reads. Finally, we experience a decline at 2048 threads because the lock-free contention on CAS instructions is probably so high that we start to lose performance benefits.

Then, if we look at *Figure 6*, we can see much better scaling. All ratios follow a similar increasing trajectory, indicating that as operations scale, we experience massive performance benefits with the lock-free algorithm.

Overall, we can see that a lock-free BST is a very attractive option, especially if we have a use case that requires a lot of writes and fewer reads. And as we have seen with the hash table, as the concurrent work per thread increases, we experience more and more benefits from the lock-free algorithm.

6 Challenges

While implementing this many data structures, I ran into many challenges. Due to the nature of the project, many of the issues I faced were highly algorithmic. This meant it was very difficult to find issues with correctness by relying on segmentation faults or specific output. To convince myself of correctness, I thought I had to write scripts to compare the output to a sequential output or to a lock-based output. However, due to the nature of the threads and the non-determinism in which ones could get scheduled in which order getting deterministic output to judge correctness was very difficult. Instead, I had to convince myself of correctness by closely reading and studying my code for potential issues. This process was way more involved than expected because I had to study exactly how the CAS instructions work and think about all possible interleavings. It was a rigorous process, but ultimately I learned a lot about proofs of algorithmic correctness.

I experienced some challenges while implementing the stack. It was a good introduction to building a lock-free data structure with the CAS instruction. I also learned a lot about the ABA problem and how it can be addressed. Specifically, it describes how one thread might attempt to execute atomic operations assuming a data value is A when it actually might be B, or it might be switching back to A. Thus, it can be difficult to handle this dangerous interleaving. I did some research and came across a couple of strategies to handle it and decided to pursue deferred reclamation using hazard pointers. It was definitely challenging to implement but I learned a lot about how to build correct lock-free data structures.

My next challenge was with implementing a hash table. My initial approach was to use a linear probing model because that was what a quick Google search suggested. However, after investigating (writing and debugging code that just didn't work), I decided to switch my approach. I then found the paper by Michael [2], which provided good insights into how to build an efficient hash ta-

ble. This was my first time reading, studying, and implementing what I learned from a research paper. It was an exciting process and I learned a lot about how to translate algorithmic language into actual code I could run. It was also very helpful to read proofs of correctness, so I knew the algorithm was correct.

The final challenges I ran into involved implementing the lock-free bst. This was by far the most challenging data strucure to implement. It turns out there are several algorithms out there, each more confusing than the last. The process is extremely involved and complicated, so I was initially very daunted by the task. The get method is trivial but to implement insert, and especially remove a lot of work is required. I initially attempted to try to translate my lock-based implementation, but I quickly ran into issues (I had no idea what I was doing). So then, I turned to the paper by Bronson and others [1]. This paper took me quite a while to understand and really disect. However, once I had that change I started programming with an algorithmic intent in mind. The code was defintely more challenging to write and I ran into issues several times with my code infinite looping. However, eventually I got it to work. This experience taught me a lot about the difficulties of concurrent algorithmic work and how involved of a process it can be.

7 Limitations and Future Work

There were a couple of key limitations of this project. First of all, the hardware was a limitation. The machine only had 8 cores, so there's only so much parallelism I can achieve with that. Furthermore, this project does not seek to implement the most complicated, most efficient lock-free algorithm. Instead, I intended to become more familiar with this programming paradigm and learn how useful these techniques can be. It's also important to note that the project results do not show workloads involving algorithmic work other than accessing the data structures. Thus, in practice, there could be a lot less parallelism involved in the use of concurrent data structures for concurrent algorithms. So, much of the results we presented are likely far more optimistic than actual performance in practice in a use case.

To build upon these limitations, future work should include testing these data structures on more robust machines that can handle much higher degrees of parallelism. Testing the algorithms in other programming languages outside of C++ is also important to see how the runtimes can be improved. Furthermore, there are several more efficient variants of the algorithms I expressed in this paper, which deserve further study and reflection. It will also be important to see how these data structures perform in practice with workloads that cause a buffer between data structure operations. Finally, this project explores just three data structures. There are many more data structures available with exciting lock-free implementations and algorithms that should be explored and verified.

8 Conclusion

I hoped to use this project to understand why certain lock-free algorithms work or do not. My original intention/hypothesis was to show that lock-free algorithms are far worse than lock-based algorithms in most cases. However, my results proved me wrong, especially in the case of the HashTable and BST. It's very exciting to see how powerful lock-free algorithms are. I was surprised to see how well lock-free algorithms scaled to a number of operations, but it makes a lot of sense. There is a lot of overhead required for each thread to acquire the mutex every time it does an operation, so if there are a few threads doing a lot of operations it makes sense to use a lock-free implementation. This programming paradigm is extremely powerful but equally tricky to implement. It takes time and care to get correct, so it probably is not worth investing time to implement in most cases unless the efficiency improvements make a big difference to the use case. Overall, I learned a lot about these kinds of algorithms, and I will definitely incorporate this knowledge into projects I do in the future.

I would also like to note I spent approximately 30 hours on this project. A lot of the time was spent researching rather than coding. Though, there is a significant amount of coding time that went into this project as well.

References

[1] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10). Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/1693453.1693488

[2] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '02). Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881