# 15-418 Project Final Report

Aidan Smith

December 15, 2023

## 1 Summary

I implemented a lock-free concurrent hash table using x86-64 compare-and-swap and memory-ordering primitives that achieved a 1.7-2.5x speedup over `boost::concurrent_flat_map`. This is an open-addressing implementation using linear probing and tombstones.

## 2 Background

There are several popular algorithms for implementing hash maps such as open addressing or separate-chaining, each with many different flavors. Many of the popular algorithms have their own trade-offs and implementation challenges, while yielding their own unique benefits. I began my project researching and even began working a couple implementations before I finally found my way to what I have now.

Eventually I landed on an open-addressing hash table with internal keys. I choose open-addressing over separate chaining because of its advantages in terms of simplicity and locality. Open-addressing involves storing all of the elements in a single contiguous array, there are hybrid techniques here, for example in Facebook's F14Map they provides the option to instead store pointers to elements in the array instead of the elements themselves, to reduce memory waste [1]. In my implementation I will be storing keys and values directly in the array, for simplicity and speed. We will handle collisions by probing linearly for the next available slot. This approach can lead to better cache performance as elements are stored contiguously, enhancing locality and reducing cache misses. Additionally, in a concurrent setting, open-addressing can offer better synchronization properties compared to separate chaining, where each linked-list chain can become very complicated to deal with in terms of synchronization and reclamation of nodes.

The key operations for this data structure are inserts and lookups. These are the backbone of a good hash table implementation and are my primary focus in optimizing. Both operations will initially hash the provided key to get a starting index into the data structure, then they probe until they find the matching slot or an empty one. If we are reading and we see an empty slot, we know that our key doesn't exist so we are done, otherwise, we can just do a relaxed load from the data structure to get our result. Inserts are slightly more complicated, when attempting to insert into a new slot, we need to beware that other threads can steal this slot before we do, so there is a two-phase compare-and-swap procedure to ensure safety.

Both these operations become significantly more complicated when a resize is occurring, and this is an area that is probably the biggest parallel hazard for the whole project. A resize will have to allocate a whole new arena of buckets, so we have to go through the process of migrating the whole table, ideally, without blocking threads. The main source for my algorithm was a presentation my Cliff Click about a lock-free concurrent hash table he wrote in Java [3]. He explains a very simple yet fast way of implementing thread-safety for a hash table of this type, and I found his explanations very instructive for when I was trying to reason about the correctness of my code.

## 3 Approach

For this project I used C++ and targeted everyday server or desktop x86_64 machines with (hopefully) at least 16 cores. The problem maps very naturally to multithreaded programs, and we can imagine we have as many reads and writes occurring simultaneously as we have threads executing. Thus, ensuring that reads and writes don't block each other was paramount, and thankfully, fairly straightforward initially with open-addressing. Since the lookup is the same in both cases, we can use the same logic to search through the array in both cases, which made debugging the array walk much easier. I decided to make my array a power-of-2 and have it be circular. This way, we can start at any hash, bitwise-and it with a pre-calculated mask, and get an index into our array that will wrap around. In my implementation, I store hashes instead of keys themselves to save space and ensure we are not errantly performing non-trivial key comparisons. In this hash field there are also several sentinel values that we use, those being `NULL`, `TOMBSTONE`, and the `MOVED` bit.

On lookups, if we find our key, we are done and we can just return a relaxed load of the value. If we see `NULL` we are done, as they key would've been in this spot if it was in this map, and we treat `TOMBSTONE` like any other key, and just continue our search. If we see the `MOVED` bit set, then we know that there is a resize in progress, and we can look in the new table to see the new value. This way readers don't block even if there is a resize in place. Furthermore, if we see a `MOVED` key, we know for certain that someone will have already added the value into the new table.

Inserts are very similar, with the nicety that now `TOMBSTONE`s are also a hit for us, as we can repopulate those erased slots with new key-value pairs. Once we have found a slot, we are not quite out of the woods yet, as we have to compare-and-swap our hash into the slot, and only on success will we have actually reserved it. As I eluded to earlier, another thread can steal our slot before our compare-and-swap completes, which is why we have to check the return value, and if that slot was stolen, we continue searching. Resizes add some complexity to inserts, as there are two distinct cases to consider, first when a resize begins after an insert and persists during, and when an insert occurs triggering or during a resize.

If a resize is occurring while we are walking the table trying to insert, we may be alerted to it, if we come across a `MOVED` hash. If it isn't our hash, we just continue, but if it is, similar to the lookup case, we will have to go to the new table and insert it, with the same semantics that we would use normally. On the other hand, if when we enter the insert, the number of slots before going over load capacity is 0, then we will either trigger or join a resize if one is already in progress. There are two stages to a resize, first, there is some contention to create the new allocation, then all the inserters can work together to copy elements. We use a double-checked CAS to ensure that only one person will do the large amount of work to allocate a new table. This looks like first checking the value, then trying to CAS it to a placeholder, then actually storing the allocated address. If any steps fail, we know that someone else has done it for us, and we can just move on to the next step. Copying elements over is done one at a time, as if we were inserting each old element into the new array, and getting rid of the tombstones that have built-up. As inserters join the resize, they can take 'work', in the form of indices to copy, off a counter, and then migrate the cell at that index. Migrating a cell may move it to a new index, so it is important that we recalculate that from the hash that is store, and then the actual swap is another two-stage process. To avoid losing any writes that are happening to the old table, we first write the old value to the new table, and try to compare and swap in the old value while setting the `MOVED` bit. If this fails, then someone has written to the value, so we repeat the process, writing the new value and trying to CAS again. This process is repeated until the whole table has migrated.

This allows for readers to carry on during resizes, while all the inserters help to do the copying, as they are the ones that need the extra space. This is why readers should never block, but writes sometimes will. We still provide lock-free semantics, as we always guarantee system-wide progress is being made. There are a lot of different approaches with how you perform resizes, how you gather workers to help, and how much work each worker does. This approach seemed like a good balance of being speedy and being very good for the readers, which has been one of my main goals with the project.

## 4 Results

Now looking at some benchmarking results. All benchmarking was done on a 2.3 GHz 8-Core Intel Core i9-9980HK processor with 32GB of RAM. I am comparing my hash table primarily against Boost's `boost::concurrent_flat_map`, as it is one of the few production grade open-addressing concurrent hash maps I could find. I wanted to compare against another implementation that used the same approach to make a more inline comparison. The biggest advantage that this map has over mine is more general typing support, for both keys and values. My map only allows for trivially-copyable values to be stored, and also doesn't store strings. The specificity of my implementation has allowed me to slim down the insert and find processes making them very efficient in benchmarking, in almost all scenarios.

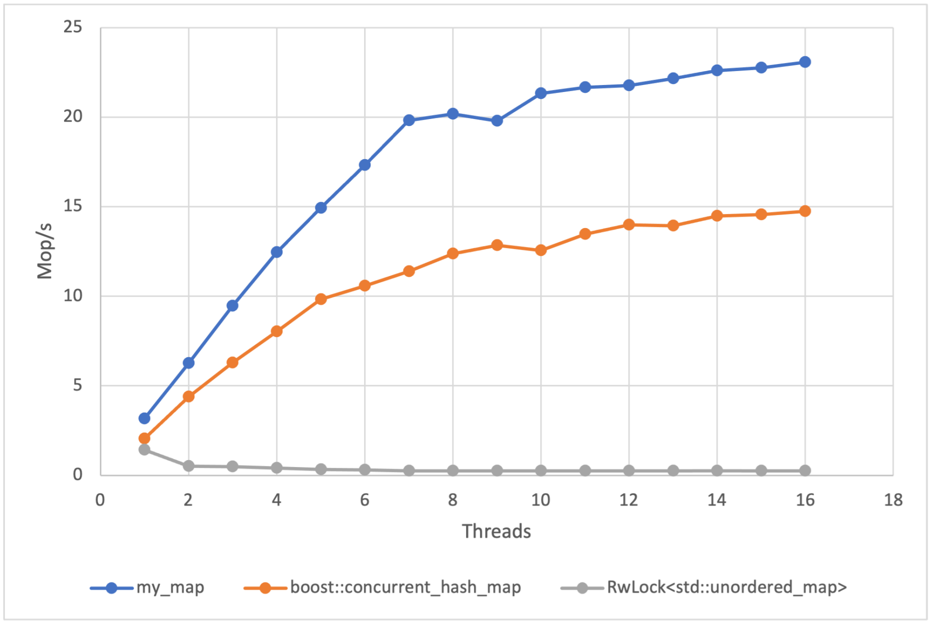

Figure 1: Read-biased workload

The first two benchmarks I have are from a 90:10 read-bias workload (Figure 1) and a 90:10 write-bias workload (Figure 2). Right off the bat, my map performs 1.5-2.5x faster which is quite a solid improvement. Perhaps even more surprising, the Boost map is about 150% faster in the read-biased test than the write-biased test, whereas, my map is about 15% faster in the write-biased test over the read-biased test. I think this is likely due to the fact that the more write-biased a workload is, the more it is accessing the same slots over and over, which is what my implementation shines at. When there is more reading going on, specifically misses, where the result is not in the table, our linear probing will cause us to walk for longer.

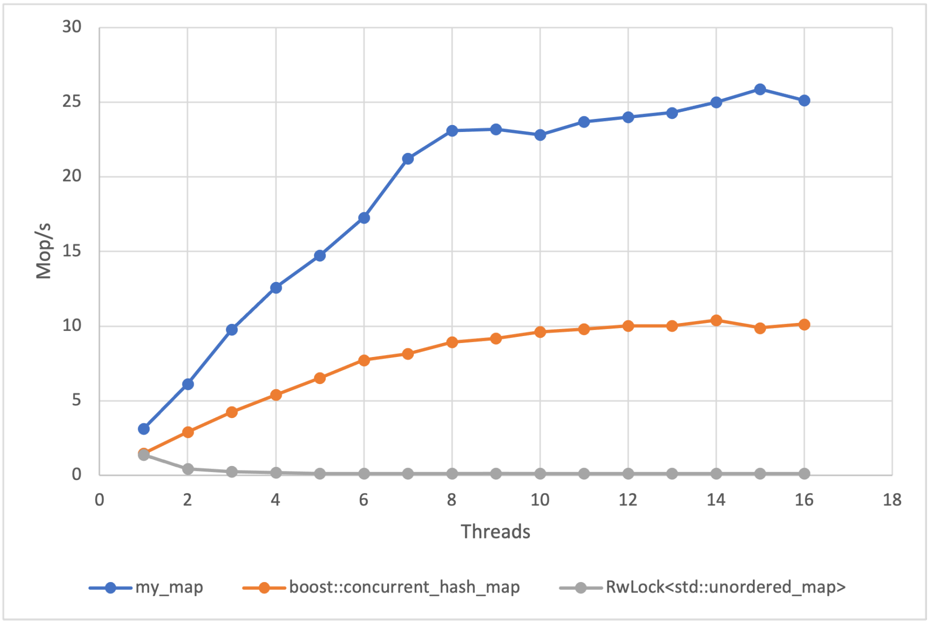

Figure 2: Write-biased workload

That leads us to the even workload, which is by far the worst performing benchmark for my implementation. Building on what I mentioned before, searching for keys that don't exist in a populated table is the worst operation for this table. Thus, this workload is in many ways the worst of both worlds. In the read-heavy, there are fewer inserts, so we have to walk less distance for each access. In the write-heavy workload there are many entries, but we almost always are searching for a hit, so we know we don't have to walk too far. In this case, we insert quite a bit, and we walk quite a bit, so we end up taking those longer walks more often, leading to a drop in performance.

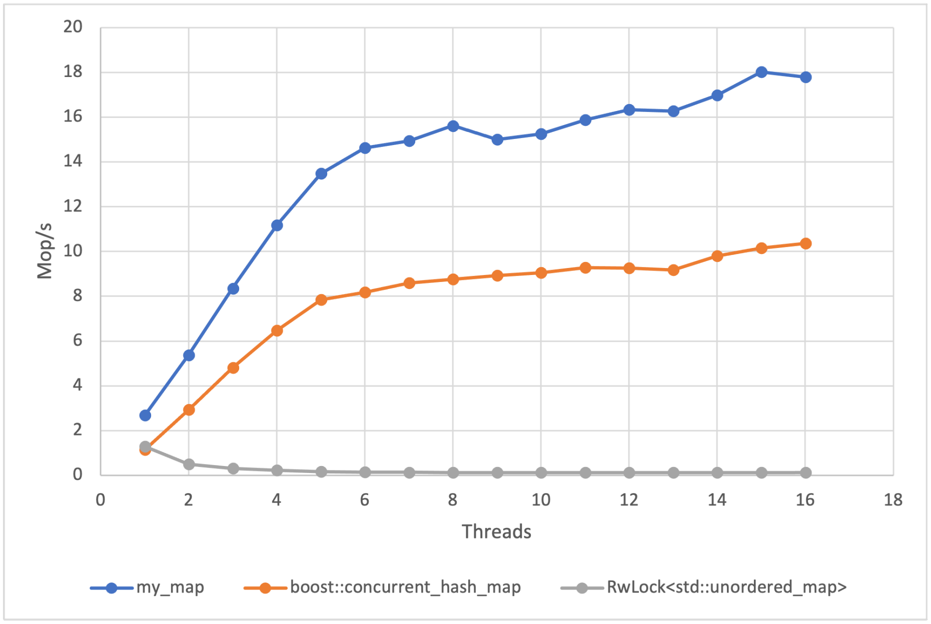

Figure 3: Even workload

Then the last two benchmarks I ran my table configured with a high (Figure 4) and low (Figure 5) maximum load capacity both running on a 75:25 reader:writer split. This provides some insight of how much the load effects my table, and helps users make an informed decision about what load factor they can tolerate.

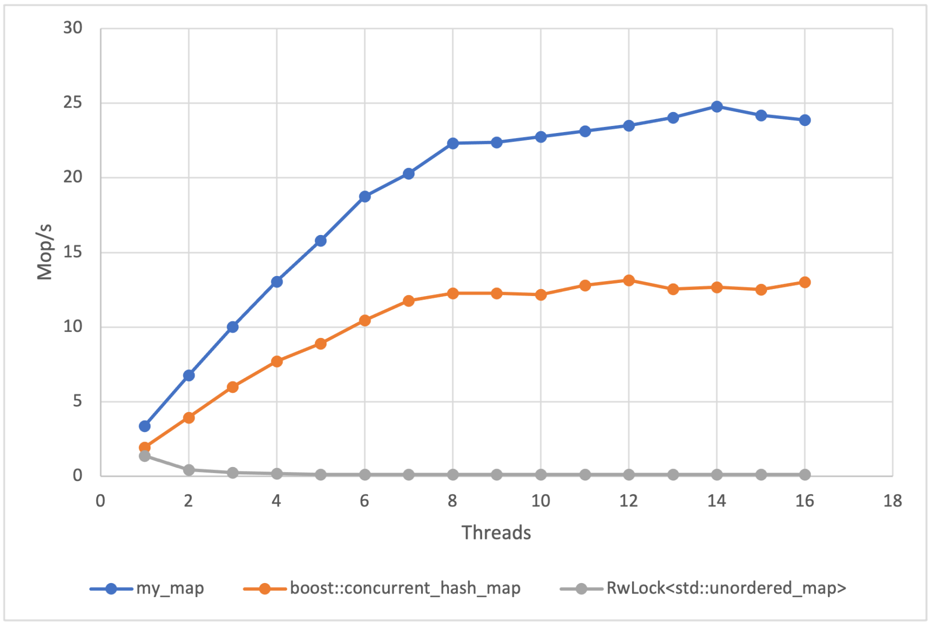

Figure 4: High ($> 0.8$) load factor

We can see a bit a of drop in performance as expected, however, only to the tune of about $\sim 10\%$. This is fairly good, considering the table is about 4 times smaller in the high load factor example. I would expect this behavior to degenerate quite a bit though, and in the future, I would like to try more granular numbers, and some very extreme examples e.g. $> 0.95$, $> 0.99$, etc.

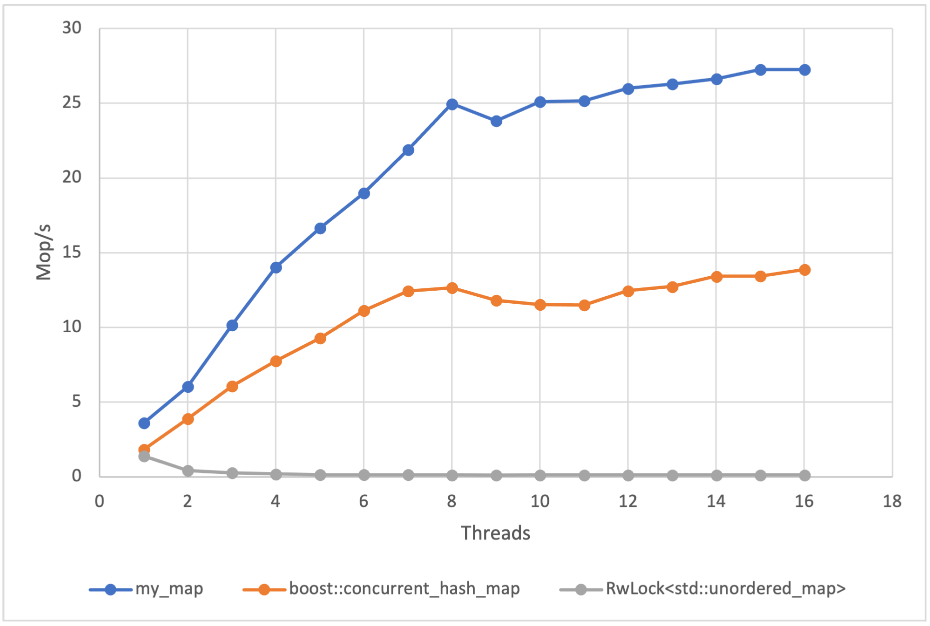

Figure 5: Low ($< 0.2$) load factor

The main limitations in speedup come from searching in the array, especially if it is filled with lots of values and tombstones. Thus, the two main speedups for this implementation I would look at next are vectorizing the lookup, and doing clean-up as we go. Maps like Facebook's F14Map use SIMD instructions to search chunks of buckets at a time instead of probing one by one [1]. I think this would provide a big boost on the search time, especially for high load factors, which make provide a nice memory use reduction. Another direction is augmenting the linear probing scheme I use to something that is more intelligent about where it jumps, or doing some housekeeping as we search, moving entries into more advantageous slots.

With more time I would try to implement something like hopscotch hashing which aims to move buckets into more convenient locations [4].

On all of the graphs, we can see a performance impact as we get past 8 threads, I believe this is likely due to those last 8 threads being hyperthreads. I would've liked to do some more testing on a bigger machine, however, the `boost::concurrent_flat_map` was only added in Boost 1.83.0, which was not the version on any of the clusters I tested, and I ran into issues trying to build from source.

## 5 References

[1] 2019. Open-sourcing F14 for faster, more memory-efficient hash tables. Engineering at Meta. Retrieved December 15, 2023 from `https://engineering.fb.com/2019/04/25/developer-tools/f14/`

[2] Joaquín M. López Muñoz. 2023. Inside boost::concurrent_flat_map. Bannalia. Retrieved from `https://bannalia.blogspot.com/2023/07/inside-boostconcurrentflat` `html`

[3] A Fast Wait-Free Hash Table. Retrieved from `https://www.youtube.com/watch?v=WYXgtXWejRM`

[4] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In Distributed Computing, Gadi Taubenfeld (ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–364. `https://doi.org/10.1007/978-3-540-87779-0_24`