Multiprocessor Programming Final Project Report

Ryan Frost

Aims & Objectives

The objective of this project is to implement a bucketized chained hash table. The problem with current hash tables is that they are typically not thread safe, and a global lock on the table is wildly inefficient. To demonstrate the efficiency differences of a lock-based strategy versus a lock-free strategy, two thread-safe implementations are created. The lock-based strategy uses striped locking, which spreads a fixed number of locks across all hash indexes such that multiple indexes share a single lock. The lock-free strategy uses OpenMP's atomic compare capture to swap a new node onto the head node of each bucket.

Related Work

As mentioned before, creating a course, global lock across the whole table is the naive and inefficient way to solve the hash table problem. Another naive solution would be to have a lock for each possible index. This fine-grained approach significantly increases the memory footprint of the hash table. Striped locking reduces memory by sharing locks between hash indices. A bucket approach typically uses an array at each index to store multiple values. This format does not lend itself well to compare and swap, which will be used for the lock-free implementation. The implementations rather use linked list and rely on the Michael-Scott queue mechanics for lock-free updates to specific buckets.

Other hash types were also considered, such as Cuckoo hashing. Cuckoo hashing gives every unique key two possible buckets for placement. If both buckets are full, a random entry is "kicked" out of its spot and replaced with the new value. The thread then finds a new home for the kicked value in one of its possible two buckets. The cycle of kicking continues until every entry has a spot or the table resizes. The two possible buckets were problematic for a lock-free implementation since they typically require a single value to compare and swap.

Solution Approach

The solutions are implemented in C using OpenMP. The system uses an input file of commands. One thread reads large sections of the file and creates tasks to process them. Each thread takes in the file buffer and processes each operation one by one. There are two possible operations implemented: insert and lookup. Parallelized resizing is also supported, although each resize requires a sync up of all threads before and after, effectively acting as a global lock on the table during resizing. Future implementations will add a remove/delete function to the hash table.

Lock-based implementation:

The striped lock implementation maps N buckets to M locks using the modulo operator (N % M). If a thread needs to lock a specific bucket, it locks all buckets that are mapped to that lock. A

lock to bucket ratio of 8 was used, although this could easily be adjusted. For sufficiently large tables, even with 16 threads, the chances that another thread needs that specific lock are very small.

Lock-free implementation:
The lock-free implementation gets rid of the locks and instead relies on compare and swapping the head pointer at each bucket.

```
#pragma omp atomic compare capture
{
  old_head = chained->buckets[bucket].head;
  if (chained->buckets[bucket].head == expected) {
    chained->buckets[bucket].head = add_item;
  }
}
```

This effectively pushes to the front of the list, and if another thread happens to push before another thread on the same bucket, one of the compare and sets will fail. The failed thread will then retry. This is even lower contention for a specific bucket as the odds two threads try to access the same bucket is very low.

Solution Properties & Correctness
The main challenge of implementing a concurrent chained hash table is dealing with race conditions on the head pointer of each bucket. The lock-based version simply locks the bucket, preventing the race condition and guaranteeing correctness by mutual exclusion. The lock-free version maintains correctness by using the compare and swap loop.

Compare and swap loop:

1. The thread reads the current head of the target bucket
2. The thread creates a new node and sets the new node's next value to the current head
3. The thread tries to swap the current head with the new node using #pragma omp atomic compare capture
    a. The thread only succeeds if the head has remained the same
4. The thread continues the loop until it has won

When the compare and set succeeds, the insertion into the hash table is linearized at that point.

When updating existing values, `#pragma omp atomic write` is used to guarantee that no other thread sees a torn or corrupted value. When performing lookups, `#pragma omp atomic read` is used to ensure memory consistency.

The ABA problem is a known issue with lock-free linked list. Currently, only inserting and updating operations are supported. Since nodes are not deleted and freed, the ABA problem is avoided. Deleting nodes would require hazard pointers to ensure memory is not freed then reallocated while another thread is operating with it.

Experimental Results

The differences in performance are best shown under high contention. In larger hash tables, contention is low and the lower overhead of striped locking is generally more efficient than compare and set. In memory-restricted applications, compare and set outperforms striped locking.

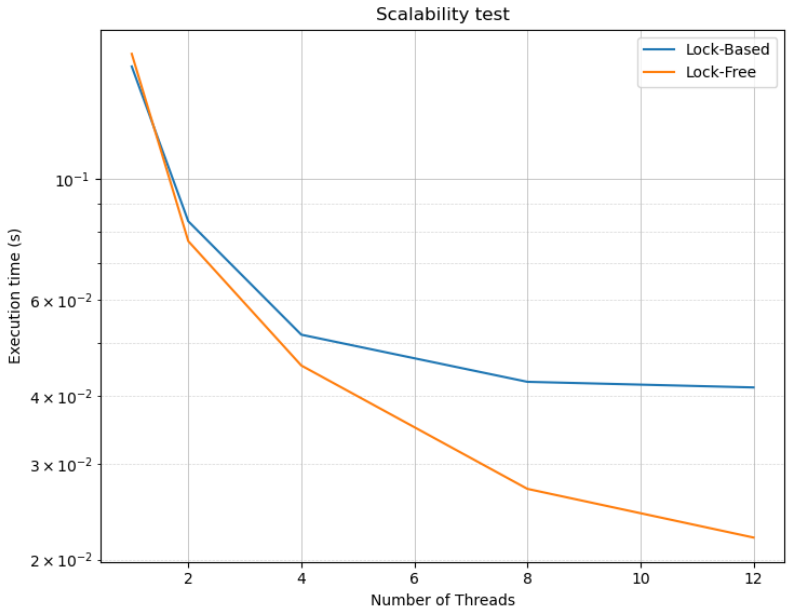

Table 1. Lock-Based vs Lock-Free Performance Speed
(Balanced data set, 100,000 entries, 64 buckets, no resizing)

The lock-based implementation plateaus at 8 thread due to the high contention on the 8 locks for the 64 buckets. The lock-free implementation continues to see performance benefits and performs faster than lock-based.

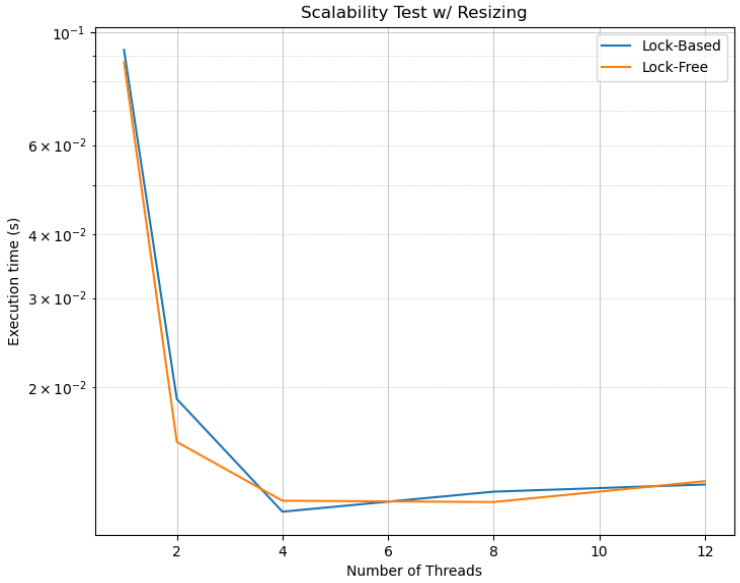

Table 2. Lock-Based vs Lock-Free Performance w/ Resizing
(Balanced data set, 100,000 entries, 64 buckets, yes resizing)

Resizing is shown to cover up any performance gains between the two implementations. Each thread must sync up before and after resizing, and those barriers effectively mask the marginal performance gains. The resize work is split in a #pragma omp for to effectively use the threads, but there is still no performance benefit to using multiple threads when resizing.

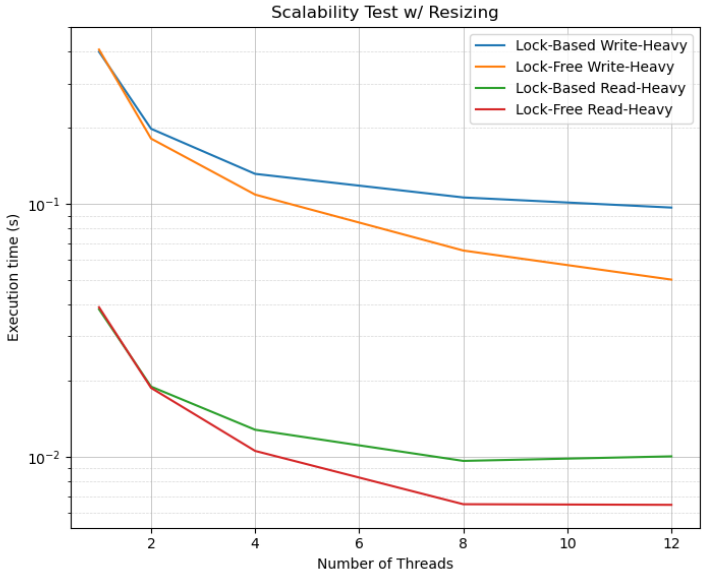

Table 3. Lock-Based vs Lock-Free Performance w/Heavy Writing and Heavy Reading
(Heavy write or heavy read dataset, 100,000 entries, 64 buckets, no resizing)

Since the lookup function relies on `#pragma omp atomic read`, there is less total contention between the threads. A write-heavy workload causes more lock contention and more compare and set fails.

Challenges

I initially pursued a Cuckoo hashing scheme because I saw a recent research paper on it and thought it would be interesting to implement. I realized that the implementation was very advanced and decided to shift to a linked list scheme.

Having one thread generate the tasks for many other threads initially became a bottleneck. I tried reading in line by line and creating a task per line, but the task workload itself was far less than the processing of the file was. To solve this, I created a buffer to read a large part of the file into and then pass that so each worker thread would process their lines and be occupied for much longer.

One small improvement I made was to add a PaddedLock struct so that the locks would be on different cache lines. This was discussed in lecture and I thought it would be interesting to implement.

Parallelizing the resize function gave me good practice defining barriers and syncing threads together. I discovered that adding a barrier even after an implied barrier was crucial to making sure all threads saw the updated table.

Future Work

Adding a delete function is a significant challenge due to the ABA problem. There are methods to solve this, but I have not implemented them.

Creating a more parallelizable resize function instead of putting barriers around it would be interesting.

References

1. Li, W., Cheng, Z., Chen, Y., Li, A., Deng, L. (2023). Lock-Free Bucketized Cuckoo Hashing. In: Cano, J., Dikaiakos, M.D., Papadopoulos, G.A., Pericàs, M., Sakellariou, R. (eds) Euro-Par 2023: Parallel Processing. Euro-Par 2023. Lecture Notes in Computer Science, vol 14100. Springer, Cham. https://doi.org/10.1007/978-3-031-39698-4_19
2. https://en.wikipedia.org/wiki/Cuckoo_hashing
3. Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20). USENIX Association, USA, Article 55, 799–812.