Name: Brandon Callender

Project # 4

1 Project Design

My HashTable design combines elements of Optimistic Locking, Perfect hashing and is partially Lock-Free; it is inspired by Javaś implementation of ConcurrentHashMap. It is configurable by the number of internal Partitions are available (p) – each Partition is an independent HashTable (created at startup). This means that the HashTable could potentially serve up to p writes simultaneously, provided they hash to different partitions. Each Partition is protected by very simple fine grained locking – all threads have R/W access to each bucket (each will use a lazy list-based hash[1] to ensure thread-safety and correctness). If a write operation requires a resize, the table will be locked – all current operations will be allowed to finish (until the reader-writer pool drains), but any R/W operations that come in will block until the resize is complete.

This design did not change substantially between my design specification and the final product. A few more specifics were ironed out – for example, keys are not hashed to determine which partition they go into, but are hashed once at the partition level to determine bucket membership. Instead of working on a per-bucket level, resize-checking is done on a per-Partition level, when the total number of elements reaches over a threshold (numElements * loadFactor * expectedNodesPerBucket). Oddly, tuning this expectedNodesPerBucket ended up being quite difficult. Real HashTable implementations use very small values, 1 or 2, but I saw more performance at around 8 or 16 elements per bucket (the final testing value was 8 elements per bucket with a loadFactor of 1). The lazy list implementation inserts values into the list in order, based on their key, which allows for more efficient searches and easier semantics for optimistic traversal on add and remove – we know as soon as we reach a point in the list that we will need to add or remove the element in question at that place.

2 Analysis

2.1 Concurrent List

contains(): Wait-Free: any thread that continues to execute instructions within this function will complete in a finite number of steps no locks are obtained and there are a finite number of nodes per bucket to search, returning true only if an unmarked (non-deleted) node is found with a matching key.

addConcurrent()/removeConcurrent(): Deadlock/Starvation Free, locks are acquired

[1]Inspired in part by: Heller et al.

only when the two elements of the list to be modified are reached. After locks are acquired (in ascending order always, to prevent deadlock), the elements are validated to ensure that no other thread has changed the next pointer or marked one of the elements as removed in the time between the last check and the lock. As the nodes are logically removed before their pointers are swapped, we can ensure that lock-free contains() traversals over the list during a remove will never return stale nodes (logically but not physically deleted).

addConcurrentNoLock(): Wait-Free/Lock-Free but must be called serially (only during resizes, when one thread is rebuilding a list. No locks are acquired.

2.2 Serial List

add(): Wait-Free/Lock-Free a finite step operation with no concurrency, any single thread will complete.

contains(): a finite step operation with no concurrency, any single thread will complete.

remove(): a finite step operation with no concurrency, any single thread will complete.

2.3 Coarse Hash Table

resize_table(): Blocking – Deadlock Free. There is no circumstance in which multiple threads could be waiting on each other for a chance to resize the table indefinitely as long as the resizing thread makes progress. After each thread escapes the resize lock, it checks again if the table needs resizing to prevent unnecessary resides from threads that noted that the table needed resizing when they attempted to lock it, but in the mean time another thread resized the table.

add_coarse(): Deadlock Free: As long as the unlocked thread makes progress, all threads will be able to proceed. Lock granting has no guarantee of FIFO or any kind of fairness, so threads may starve waiting for access.

remove_coarse(): Deadlock Free: As long as the unlocked thread makes progress, all threads will be able to proceed. Lock granting has no guarantee of FIFO or any kind of fairness, so threads may starve waiting for access

contains_coarse(): Deadlock Free: As long as the unlocked thread makes progress, all threads will be able to proceed. Lock granting has no guarantee of FIFO or any kind of fairness, so threads may starve waiting for access

2.4 Concurrent Hash Table

resize_table(): Calling thread is blocked until reader pool drains and it is granted an exclusive lock on the table. Starvation-Free, as the R/W lock has a notion of FIFO fairness –

no request that comes after the resize request will be granted until the resize is finished.

add()/remove(): Deadlock/Starvation Free with same rationale/guarantee as ConurrentList's add()/remove() methods. A concurrent hashtable has several partitions, each of which is a hash table with concurrent list buckets. An add/remove call selects one of these partitions where the selected key will be stored (a resize will never change which partition a key belongs to, as each partition hashes the key again before adding it to its internal bucket). A shared lock is acquired on the partition and the key is added to its hash bucket using the add/remove methods from concurrent list. A call to these methods might have to wait on a resize operation to finish to be granted a shared lock.

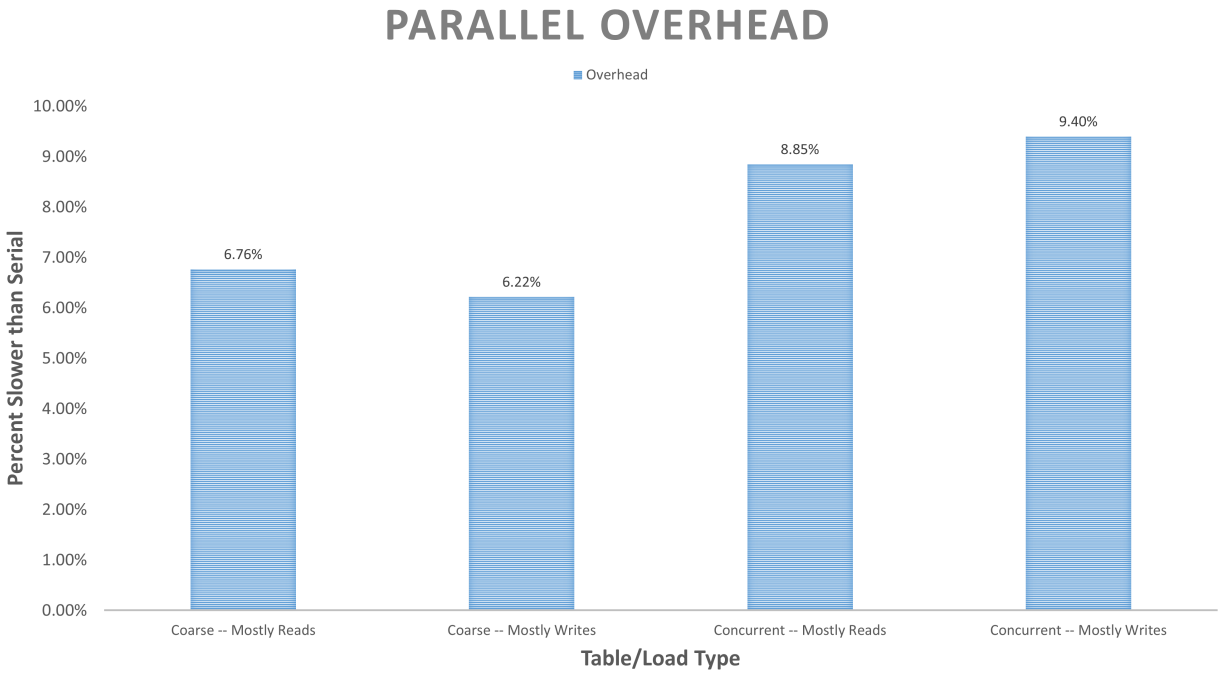
contains(): Wait-Free, as noted above in ConcurrentList. Resize operations do not alter the data already contained in the partition during their execution (they only replace the pointer to the buckets at the end of their execution), so contains calls can proceed concurrently with resizes without acquiring any locks – based on the guarantees discussed above.

3 Experimental Results

3.1 Dispatcher Throughput

The maximum throughput recorded for the dispatcher at n=16 was 2,270 packets per millisecond. Since the last time we tested throughput (HW2), Ive made some adjustments to the particulars of the dispatch algorithm, but not the core structure. The changes have helped to increase the rate of the dispatcher at high thread counts in HW2 the dispatcher relayed packets at a rate of 336 packets per millisecond at n = 16 (though at substantially higher rates, up to 7,000,000 packets per second at n = 1). I did not bump up against this upper barrier on throughput at any time during the next experiments.

3.2 Parallel Overhead



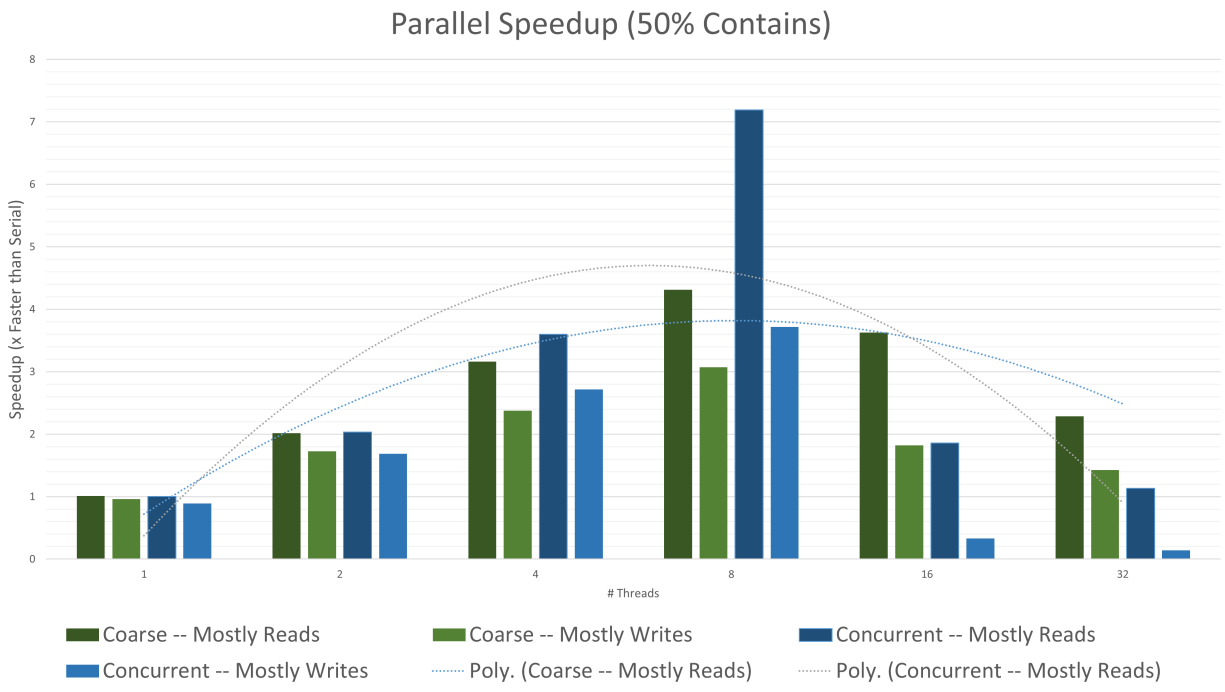
3.2.1 Mostly Reads

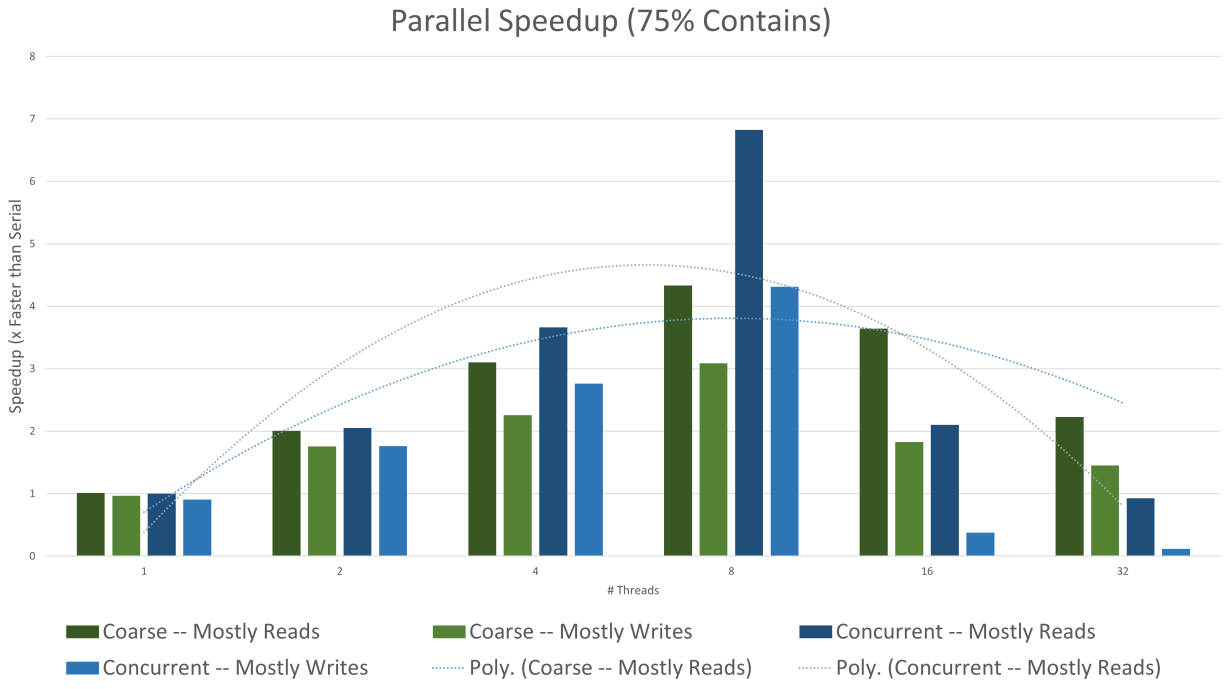
For some reason, performance in both cases was better than expected. Concurrent performance was a bit worse than expected, mainly because I did not take into account just how many locks I would need to acquire for each operation, even if they were more parallelizable.

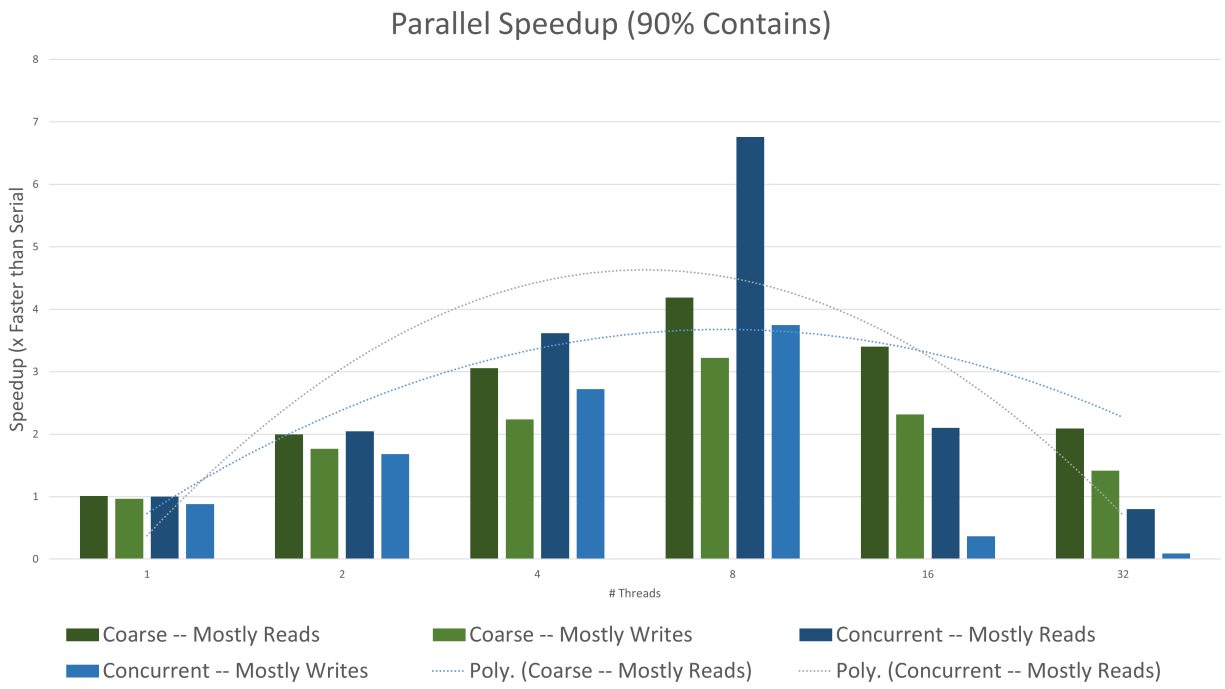
3.2.2 Mostly Writes

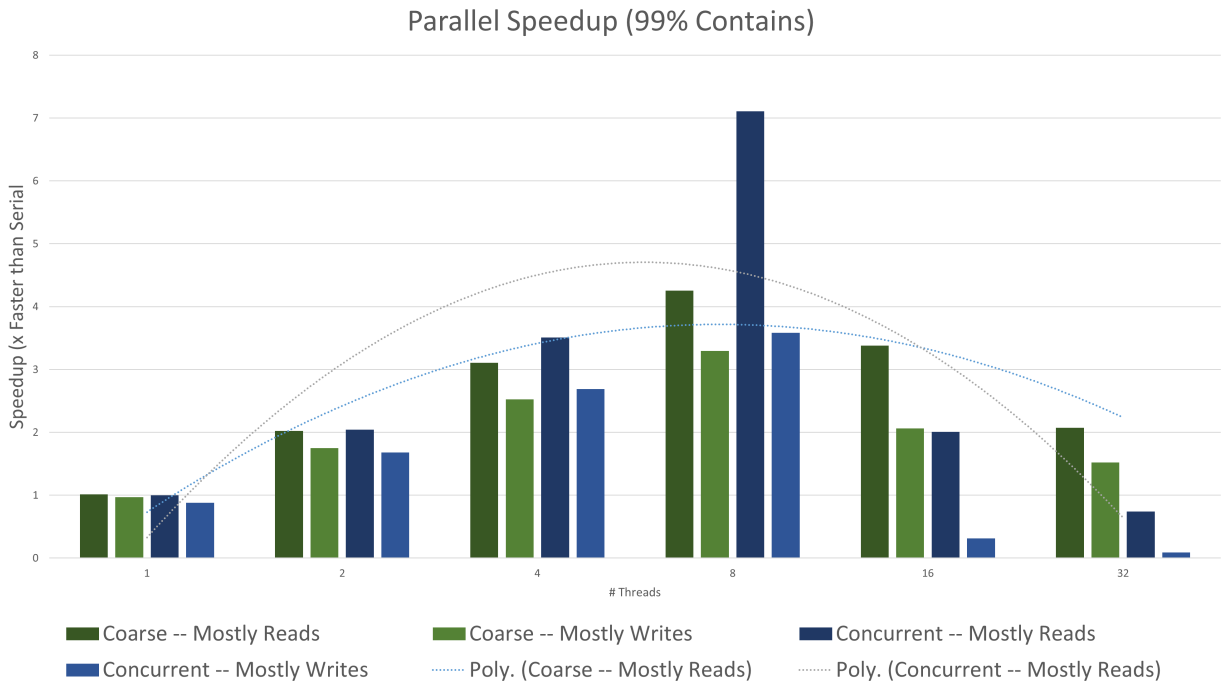
I overestimated the extent to which less frequent resizes would impact performance of the Concurrent Hash Table. I expected it to outperform coarse, but the indirection of having to interact with the external table, then a partition-level hash must have overcome whatever advantage I might have obtained by lessening resize pressure. Moreover, in a single threaded environment, the conurrentList implementation is certainly slower than the serialList, as it requires fine grain locking (4 pthread_mutex syscalls) per add/remove operation. Moreover the acquisition of the pthread_rw lock to access the partition might have contributed to the increased overhead of concurrent with mostly writes.

3.3 Parallel Speedup









3.3.1 Mostly Reads

My hypothesis that this is where the Concurrent implementation would shine indeed rings true – partially. Concurrent matches Coarse at 2 threads, slightly outperforms it at 4 threads, and nearly reaches the maximum 8x speedup (hitting 7.1x at c = 0.99) at 8 threads. This is due to its non-blocking architecture on reads, which can always be served regardless of the

state of a partition and whatever add or remove calls are interleaved with them. Differing percentages of contains calls didn't significantly change the results of the experiments (with the standard deviation of speedup values of the same mostly_reads load configuration and thread count results [with different contains values] 0.055851408 and 0.183103864 for Coarse and Concurrent respectively). I expected this to affect Coarse more, as the serial list implementation does not maintain an ordering, so a contains() call would have to search through the entire list before replying in the negative that a value is not present in the list.

Performance beyond the maximum core count in SLURM, however, is another beast altogether. Performance falls off substantially, falling far behind coarse and at 32 threads dipping as low as 0.08x speedup. I have a few theories for why this is, but none of them are definitive. I tried profiling my implementation and found that while Coarse spent the majority of its time calculating fingerprints and retriving packets, Concurrent spent most of its time waiting on an empty queue – which is odd, because it wasn't pressing up against the total dispatch rate (especially odd at the very high thread counts where performance dropped significantly). Perhaps the profiler didn't track time spent in locks (as these would be presumably syscalls and not tracked by gprof – in which case, the performance drops may have come from a high number of threads happening to hit a specific partition all at once as a resize was happening and had to wait for the r/w lock to become available, contentiously spinning on the same value in memory. In relatively short testing times (5s) this could cause a few bottlenecks. Moreover, the concurrentList implementation, while very fast in theory, might have strained under the stress of so many threads attempting to add values at once (though if so, that stress didn't show while profiling.)

3.3.2 Mostly Writes

I hypothesized that ConcurrentHashTable would falter a bit under a high write load, but would outperform the coarse-grained implementation. At $4 \geq n \leq 8$, it outperformed slightly the coarse-grained locking, but at low and high extremes it was trounced. The low thread counts are easy to explain, the extra parallelizability didn't make up for the increased overhead that came with managing the partitions and their concurrency guarantees (grabbing multiple locks in add/remove). At high thread counts, I can only assume that resize bottlenecks and contention for adding/removing elements become more and more prevalent, and the implementation as a whole is unable to handle higer throughput. Because the Concurrent implementation performs so much worse under high write pressure than read pressure (at n = 16 and n = 32), I am lead to believe that the high rate of lock acquisition and retrying required to maintain consistency in the concurrentList implementation dogged the performance numbers.

4 Conclusion

I suppose my largest point of confusion came from the fact that I implicitly thought that more concurrency (even if it came with higher overhead) would result in higher overall performance, which I have now seen is surprisingly false. Moreover, I was surprised at how

much of a performance gain we saw out of the Coarse-Grained solution, it held its own against a far more complex and theoretically concurrent implementation at 4 and 8 threads, managing a maximum 4x speedup over a serial implementation.

In the future, I will not hesitate to implement the simplest solution first, it might be sufficient and the most efficient in terms of programmer time (which is typically at a higher premium than processing time).