Concurrent Extensible Hash Table

15418/15618 - Parallel Computer Architecture Project

Danié Alvarado, Xinzhu Cai

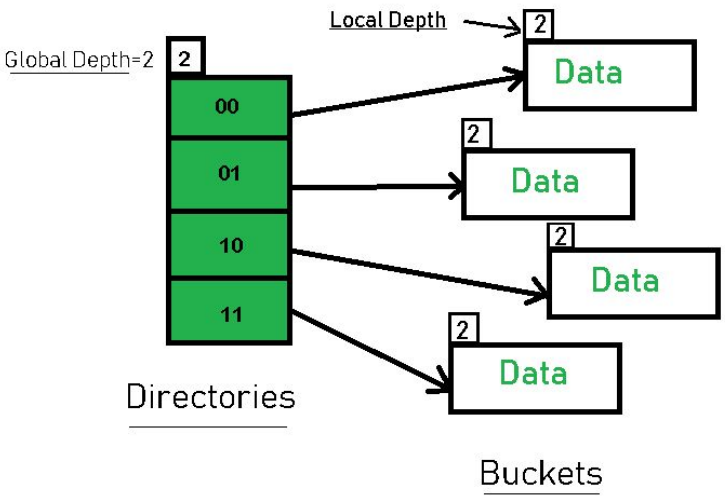
Summary

We implemented three versions of the concurrent extensible hash table including a lock-free extensible hash table using CAS [1], which is based on a split-order list, a coarse-grained lock-based version, and a fine-grained lock-based version based on a two-level locking mechanism. Given the experimental results on a 12-core shared memory multiprocessor, we demonstrate that the lock-free version is more efficient and scalable in most cases and is significantly better than lock-based ones under heavy or skewed workloads.

Project URL: https://xinzhu-cai.github.io/418project.github.io/

Background

Extensible hash tables serve as a key building block of many high-performance concurrent systems. A typical extensible hash table contains directories that store addresses of the buckets in pointers, and each bucket holds an expected constant number of elements. The global depth denotes the number of bits used by the hash function to categorize the keys into directories. The number of directories is $2^{global\ depth}$. The local depth is associated with a bucket. If the local depth of a bucket is equal to the global depth of the hash table, there is only one pointer to the bucket. Otherwise, there exists more than one pointer from the directory to the bucket, and the bucket can be split. When a bucket is full with a local depth equal to the global depth, inserting a new element leads to a bucket overflow and further a directory expansion.



Extendible Hashing

Structure of an extensible hash table
(https://www.geeksforgeeks.org/extendible-hashing-dynamic-approach-to-dbms/)

The concurrent extensible hash tables we implemented is created by passing an int value to specify the load factor and it supports the following three key operations:

```
/**
* Create a new extensible hash table with a specified load_factor.
Initially, there is only one bucket. More buckets are created as
the number of elements exceed load_factor value.
* @param bucket_size max number of elements stored in a bucket
*/
ConcurrentExtensibleHashTable(int load_factor)

/**
* Find a key in the extensible hash table.
* @param key: used to find the corresponding value
* @param value: used to store the found value
* returns true and stores result in value if the key exists in the
hash table; returns false otherwise.
*/
bool find(hash_key_t key, hash_value_t *value)

/**
* Insert a key-value pair into the extensible hash table.
* Update the corresponding value if the key already exists in the
hash table.
```

```
* @param key: key to be inserted
* @param value: value to be inserted
* Return true if the key already exists in the hash table
*/
bool put(hash_key_t key, hash_value_t value)

/**
* Remove a key-value pair based on the key from the extensible hash
table.
* @param key: used to identify the key-value pair to be removed
* Return true if the key exists in the hash table and has been
removed
**/
Bool remove(hash_key_t key)
```

Note that both `hash_key_t` and `hash_value_t` are `uint64_t` in our implementation. They can be generalized to other types.

The sequential extensible hash table only allows one operation at a time. Intuitively, multiple find operations can run in parallel as long as the hash table is not being updated during this process. Moreover, multiple remove and insert operations can run in parallel on different buckets as long as the directories structure does not change.

In an extensible hash table, each operation should first access directories, get a bucket address based on the key and directories, and then access the corresponding bucket. Remove() and Put() might change the content of directories, so this program is not data parallel. It is not amenable to SIMD execution either.

Therefore, we focus on implementing concurrent extensible hash tables on multiprocessor machines, where efficient synchronization of concurrent access to data structures is essential.

Approach

Targeting modern multiprocessor machines, we implement a C++ library that contains a concurrent extensible hash table interface and three implementations: a coarse-grained lock-based version, a fine-grained lock-based version, and a lock-free version. There can be multiple threads executing operations on the same concurrent extensible hash table at the same time.

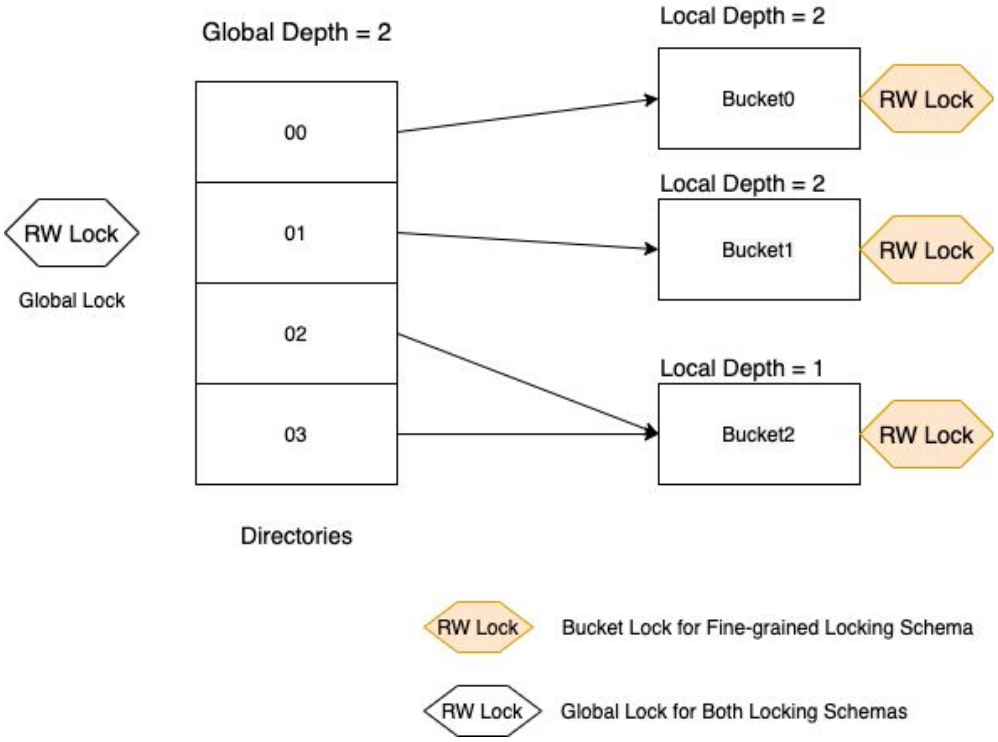
Coarse-grained Lock-based Hash Table Implementation

For the coarse-grained hash table, we used a single reader-writer lock to protect the whole hash table. That means to protect the logic inside of find() with the read lock and protect the logic

inside of both put() and remove() with the write lock. This actually leads to sequential put() and remove() execution, and only find() operations can be run in parallel. Thus, there exists significant overheads.

Fine-grained Lock-based Hash Table Implementation

To obtain a higher concurrency degree, in the fine-grained version, we designed and implemented a two-level locking mechanism. There is a reader-writer lock for every bucket for insert and delete operations besides the global lock for directories.



In find(), we grab the global read lock and then the read lock of the corresponding bucket. Note that, to avoid deadlocks, the global lock is always taken before bucket locks and released after bucket locks.

In put(), we first grab the global read lock and the write lock of the corresponding bucket that contains the specified element. It directly updates the bucket if the bucket is not full yet or the bucket already contains this key. Otherwise, it shall restart by taking the global write lock and the write lock of the corresponding bucket because the bucket will split and result in changes in directories. The size of directories may be doubled if the local depth of this bucket is the same as the global depth. In that case, the bucket pointers contained in directories will need to be updated accordingly.

Similar to put(), in remove(), we first grab the global read lock and the write lock of the corresponding bucket that contains the specified element. It directly returns false if there is no

such key in the bucket. Otherwise, we remove the key-value pair and grab the global write lock. Then we merge the bucket with its pair if it becomes empty and half the size of directories if half of them point to empty buckets.

Even with the fine-grained version, the resize operation remains a global process of redistributing the elements in all the hash table's buckets among newly added buckets. A global write lock is unavoidable on resizing. Resizing is necessary as more buckets are created. Lock-free algorithms are proposed to overcome this drawback.

Lock-free Hash Table Implementation

Lock-free algorithms are an appealing alternative to lock-based ones as they utilize strong primitives such as CAS to achieve fine-grained synchronization. The main difficulty of lock-free extendible hashing is that when resizing the table, several items from old buckets must be moved to new buckets, which cannot be achieved via a single CAS instruction. The paper [1] we are implementing proposes to utilize a recursive split-order list structure to eliminate the requirement of double-compare-and-swap instructions on the underlying hardware.

The authors proposed to move buckets among items instead of moving the items among the buckets. Unlike moving an item, the operation of directing a bucket pointer can be done in a single CAS operation, and since items are not moved, they are never lost.

There are two key data structures: a bucket list and an ordered lock-free linked list. The basic idea is to keep all the items in one ordered lock-free linked list, and gradually assign the bucket pointers to the places in the list where a sublist of correct items can be found. Every bucket in the bucket list stores a corresponding dummy head node in this linked list. This algorithm requires items in the list to be sorted on binary reversal representation of keys such that items in a given bucket are adjacent in the list throughout the repeated splitting process as more elements are inserted. Any bucket's sublist can be split by directing a new bucket pointer within it and this operation is recursively repeatable with the help of storing binary reversal representations of keys.

As shown in the following figure, there are 3 dummy nodes stored in the bucket list with the original key as 0, 1 and 3. The split-order of 1 is 10000000 and the split-order of 3 is 1100000. The least-significant-bits of non-dummy nodes' split orders are set to 1. Node 8 belongs to the first bucket because its split-order is less than 10000000 but greater than 0000000.

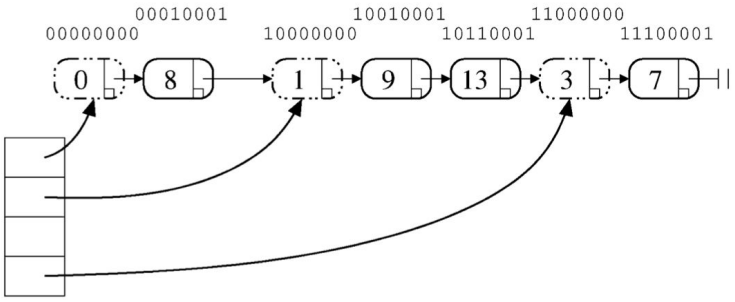


Figure 2: Structure of the lock-free extensible hash table

In find(), we find the dummy node pointer of the bucket the input key belongs to. Note that whenever we find a bucket to be accessed has not been initialized yet, a dummy node is created and inserted into the linked list first. All updates including inserting nodes, deleting nodes, and resizing bucket list are done atomically with the help of CAS.

In put(), we create a node containing the input key-value pair and insert into the linked list following the corresponding bucket dummy node while maintaining the sort order. The bucket list is resized when the average number of elements in buckets exceed load_factor.

Similarly, In remove(), Then we find the key-value node by following the dummy node pointer of the corresponding bucket in the ordered linked list and delete it.

We also explored several optimization techniques:

- **Lazy deletion** A node is marked as delete inside of remove(). It is actually deleted during list traversal. This is used to improve the response time of remove() operations.

- **Dynamic-sized array optimization** technique proposed in paper [1]. It introduces an additional level of indirection for accessing buckets: a “main” array points to segments of buckets, each of which is a bucket array. This is to avoid the unnecessary overhead of having all other processes waiting while one process is trying to reallocate the bucket table. Here, a segment is allocated only on the first access to some bucket within it. This segment table is much smaller than the original bucket table.

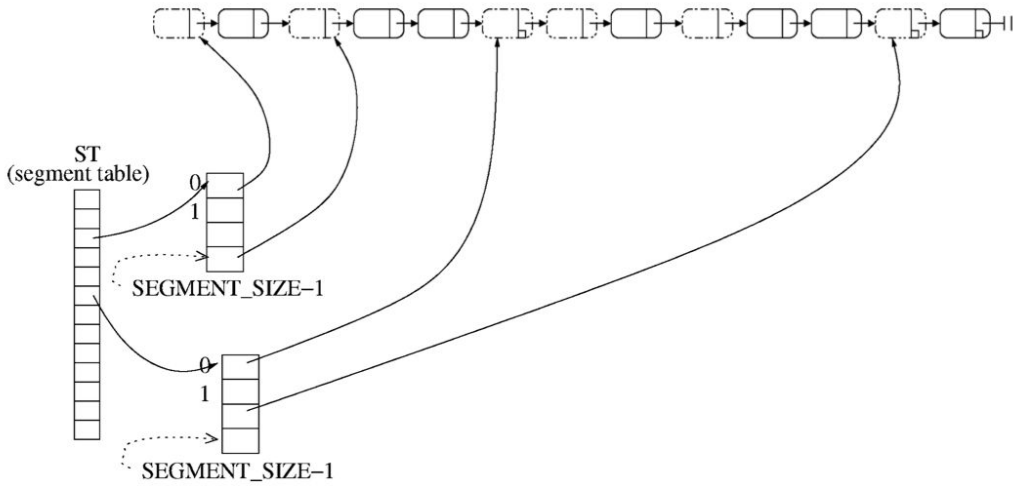


Figure 3: lock-free extensible hash table with dynamic sized array optimization optimization

Experiments

We ran a series of tests to evaluate the performance of our 3 implementations: Coarse-grained Lock-based, Fine-grained Lock-based, and Lock-free Hash Table.

Experimental Setup

Varying different variables, we came up with a series of tests to observe the performance of the algorithm and how it is affected by each of these variables. The main metric we used for performance comparison is the operation throughput that is defined as the number of operations per microsecond (ops/ms). Speedup is calculated by using a single-threaded CPU code as baseline. The OpenMP framework is used to run operations with multiple threads concurrently. In each experiment, threads repeatedly do the find, put, and remove operations. The problem size of our project is the number of threads rather than the number of elements being processed since we want to measure the contention levels of every concurrent extensible hash table implementation. We run experiments on a Mac machine with 12 cores.

Tests

To evaluate the efficiency, scalability and execution behavior of the three implementations, we conducted extensive tests with:

- various workload scales that have 1E4, 1E5, and 1E6 operations (Test #1)
- various operation distributions including a typical one (Test #1) with 10% put operations, 88% find operations, and 2% remove operation, a regularly distributed one (Test #2) with 30% puts, 40% finds, 30% removes, and a insertion heavy distribution (Test #3) with 10% finds and 90% puts.

- various load factors (Test #4) from 2 to 12 to measure the effect of load factor on execution behavior.
- various workload types including a randomized workload (Test #1), a workload with high collision rate (Test #5), a skewed workload (Test #6), sorted workloads (Test #7, Test #8).

Our tests measure the total wall-clock time it takes to complete 100000 operations on each different hash table implementation along with a serial implementation with no measures for concurrency. We tested with up to 12 threads, a number chosen because attempts to run on machines with more cores yielded unsuccessful. All tests were given randomized inputs except the ones where it is otherwise noted, set to the same seed prior to every test.

Worth mentioning is that we distributed these operations to be mixed in order, i.e. not all inserts first, then all finds, then all deletes, but instead weaved with each other. The interleaving is front-loaded with each thread beginning to perform the types of operations in a round-robin fashion, then as it has performed all the operations of one type specified, it simply continues with the remaining types.

Test #1 - Varying Workload Sizes

Description

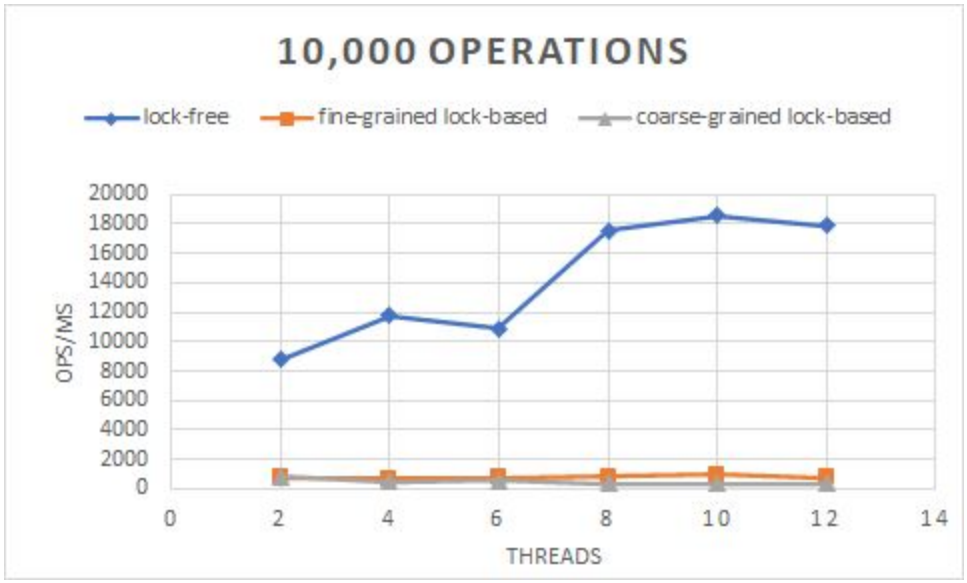
For this test we observe the performance of operating on the hash table with a typical operation distribution for hash tables: 10% put operations, 88% find operations, and 2% remove operations and a load factor of 3. We varied the workload size - that is, the number of operations - and ran this performance test with 1E4, 1E5, and 1E6 operations. Additionally, we showed the speedup graph with the 1E5 workload size.

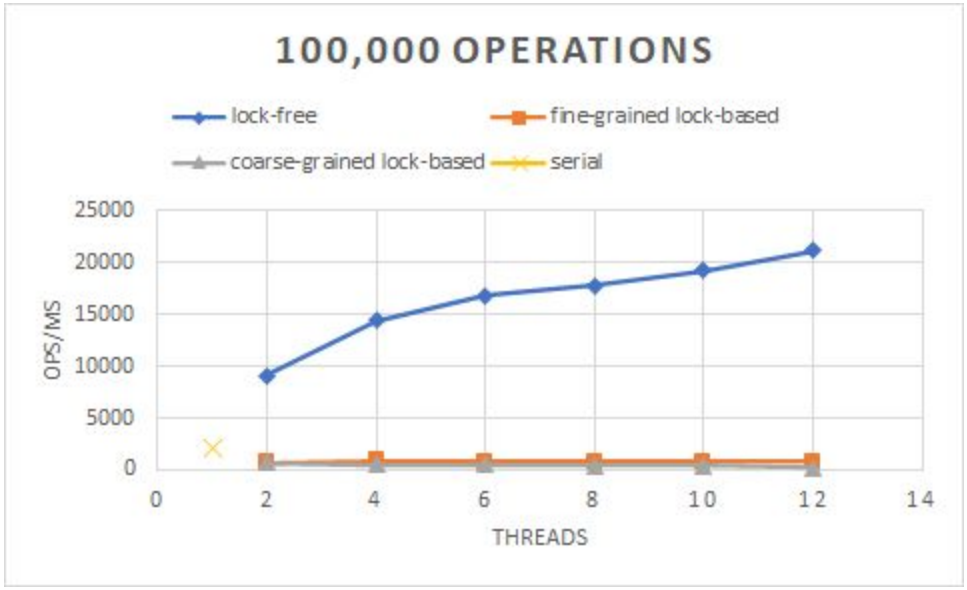
Observations & Analysis

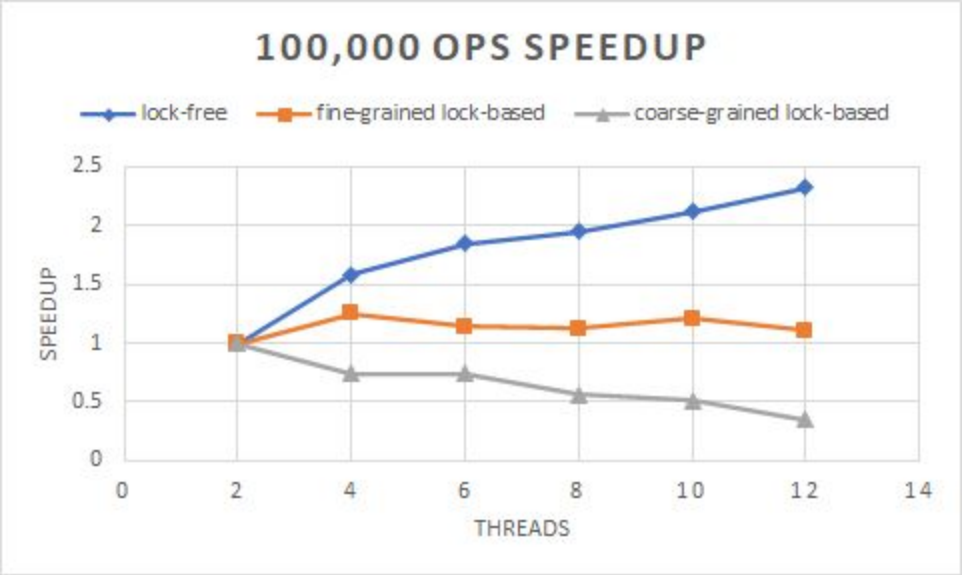
Under a typical operation distribution, the lock-free implementation achieves a 10x throughput compared with the lock-based ones. In addition, its throughput keeps increasing as the number of threads increases. That means the lock-free one is more efficient and scalable. From the speedup figure, we know that the lock-free implementation is able to significantly improve the performance. In comparison, the fine-grained lock-based implementation could achieve some speedup but does not scale as the number of threads increases. We will explain why this happened in the Deeper Analysis section. The coarse-grained one becomes even worse as the number of threads increases.

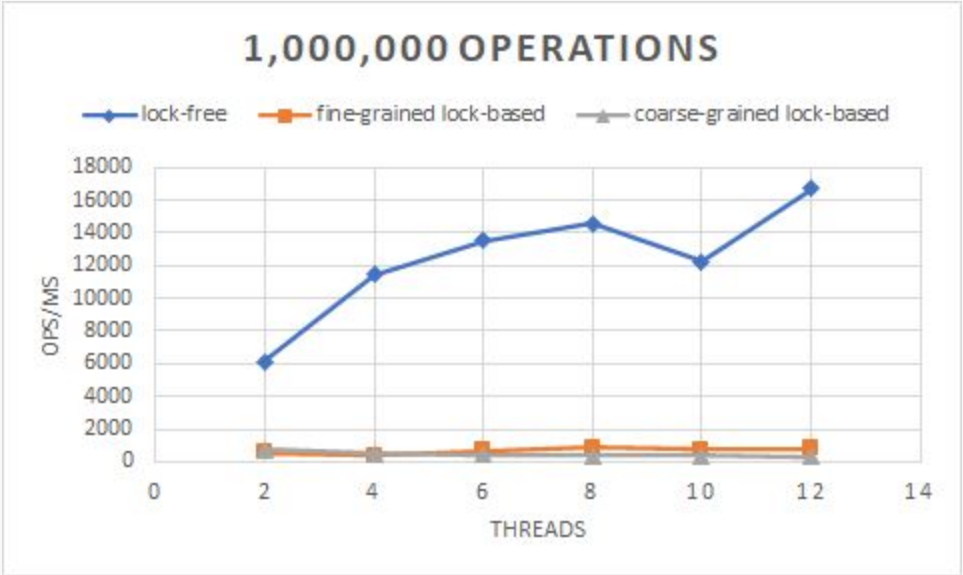
Another goal behind this test was to see whether the execution behavior of different implementations change as workload size changes. The workload size is considered as the number of operations to be executed. We set it in three scales so that it is not too small that other factors besides the ones we are controlling could affect the results, and not too large that it would take an unnecessary amount of time to run the tests. At all three scales, our lock-free

implementation achieved a similar overall increasing tendency. Since we only record results of one run, some decreasing points could be considered as noise.









Test #2 - Regularly Distributed Operations

Description

For this test we perform the same procedure followed in Test #1, for 100,000 elements, with a more even distribution of operations: 30% put, 40% find, 30% remove. This allows us to see the performance under an update intensive operation distribution.

Observations & Analysis

The lock-free implementation has good throughput results, however the lock-based ones decline as the number of threads is increased. On one hand, performance of the lock-free

implementation is not affected by having more updates. On the other hand, performance of lock-based ones hurts a lot compared with results shown in Test#1 as when there are more update operations including put() and remove() than find() operations. This makes sense because lock-based ones rely on a global write lock to protect the directories when they need to be updated and we can expect the directories to be updated more frequently under a update-heavy operation distribution.

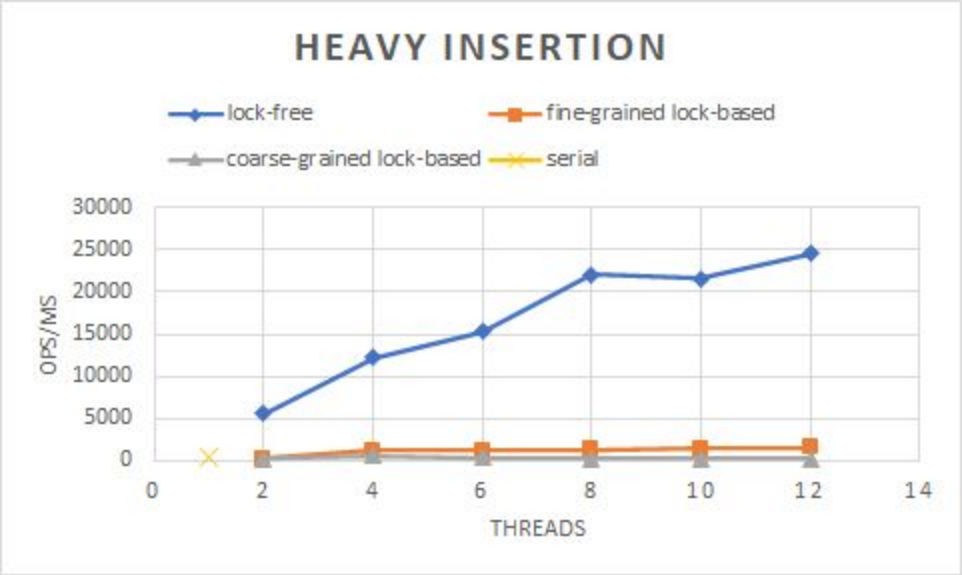
Test #3 - Heavy Insertion

Description

For this test we perform the same procedure followed in Test #1, for 100,000 elements, with a distribution of operations heavily focused on insertions: 10% finds and 90% puts. This workload demonstrates performance for the most complicated operation of the three.

Observations & Analysis

From the following figure, we know that the lock-free implementation follows the expected speedup trend. The throughput of the fine-grained lock-based implementation increases from 2 to 4 threads but tapers down afterwards. The coarse-grained implementation seems to suffer more heavily in performance as threads are increased and does not see any throughput improvement. Similar to Test #2, the lock-based implementations suffer on this update-heavy operation distribution.



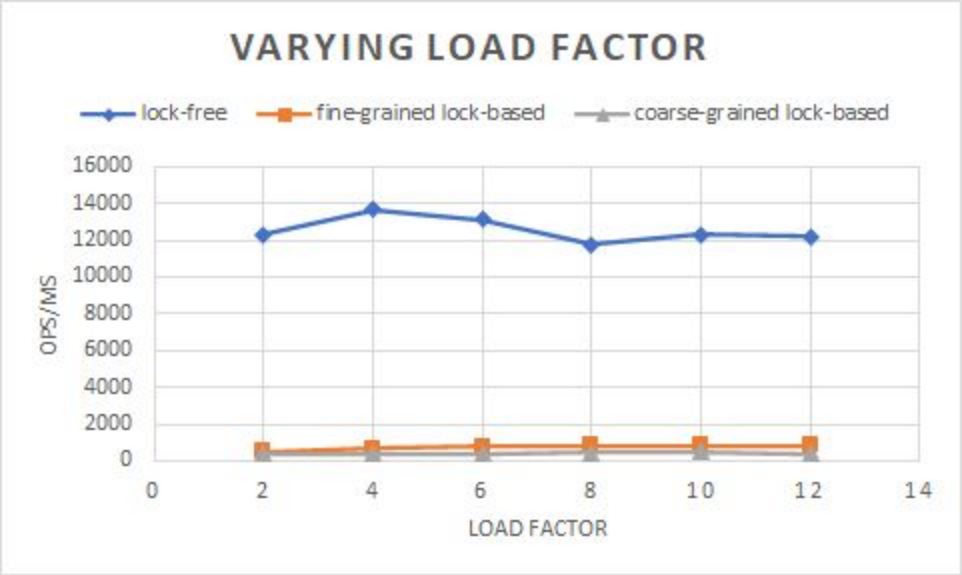
Test #4 - Effect of Load Factor

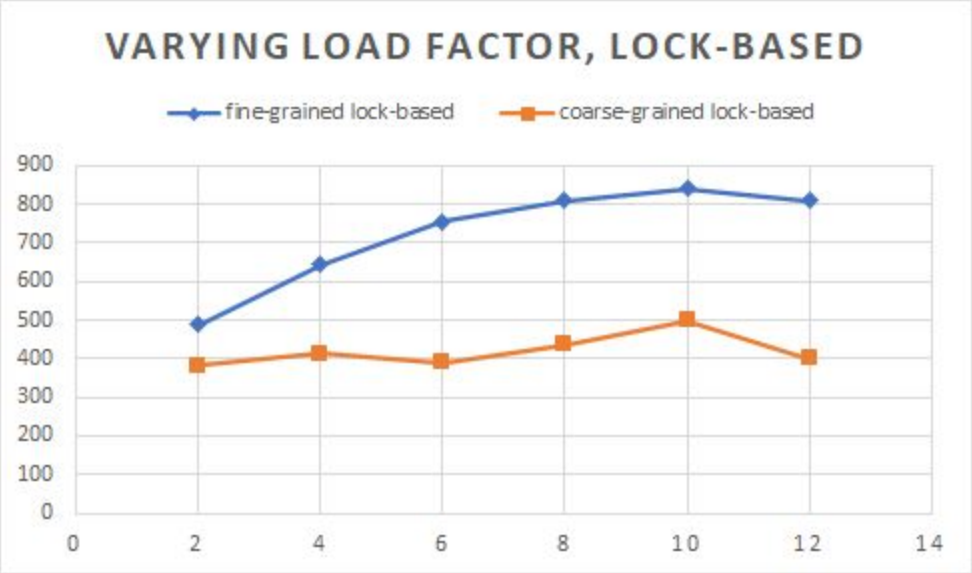
Description

With the same operation distribution as Test #1 - 10% puts, 88% finds, and 2% removes - and 100000 operations, we vary the load factor (3 for all other tests) and run the workload with 8 threads.

Observations & Analysis

The peak performance for the lock-free implementation is at a smaller load factor, with the highest ops/ms at a load factor of 4. This aligns with what we learned from our research [1], which recommended a load factor of 3 for testing a lock-free extensible hash table. The lock-based implementations did not see much of a variation in performance throughout, and the lowest performance among the different load factors for these two implementations was at out minimum load factor of 2. For the fine-grained implementation in particular, a load factor of greater than 8 was advantageous to smaller ones. This is because there is one lock per bucket. As lock factor increases, the number of locks created in the fine-grained implementation decreases, leading to less locking overhead.





Test #5 - Workload with High Collision Rate

Description

Using the same even operation distribution as in Test #3, 30% inserts, 40% finds, and 30% deletes. However, this test inserts random keys only from range [0,9], resulting in a high amount of collisions. With a high amount of collisions, we can observe how the algorithms perform with a high level of contention for specific nodes.

Observations & Analysis

The lock-free implementation sees the expected increasing speedup that was observed in Test #2, which has the same operation distribution, but the lock-based implementations more heavily suffer from the collisions: they drastically slow down as threads increase. To note about the first

data point in the lock-free implementation, 2 threads, however, is that is is greater than on Test #2 which utilizes random keys. This can be speculated to be that because of the smaller amount of keys, finds can occur faster.

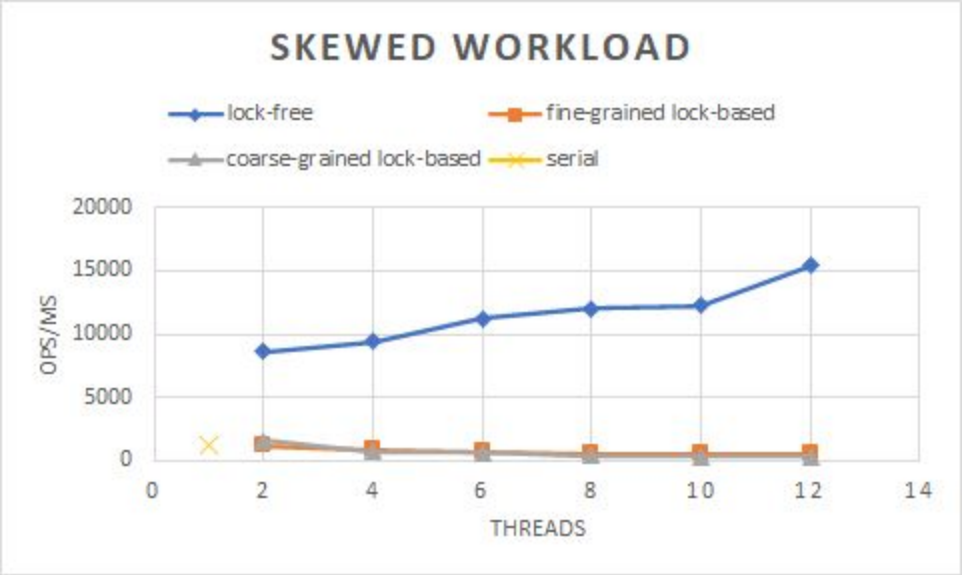
Test #6 - Skewed Workload

Description

Using the same even operation distribution as in Test #3, 30% puts, 40% finds, and 30% removes. However, the workload is skewed in the range of keys inserted into the hash table. The first 10% inserts fall in the range [0,99], the next 10% in [1000,4999], and the last 10% in [5000,8999].

Observations & Analysis

The lock-free implementation sees the expected increasing speedup that was observed in Test #2, which has the same operation distribution. The lock-free implementation did not see much change from a regularly distributed workload as in Test #2, except for the first data point being noticeably higher than on Test #2, matching results with Test #5 which also shares the characteristic of having a smaller set of keys assigned. The lock-base implementations, however, did see a sharper decrease in ops/ms as the number of threads was increased than on Test #2, due to the extra contention introduced by this workload, as we saw how drastically this can affect the performance from Test #5.



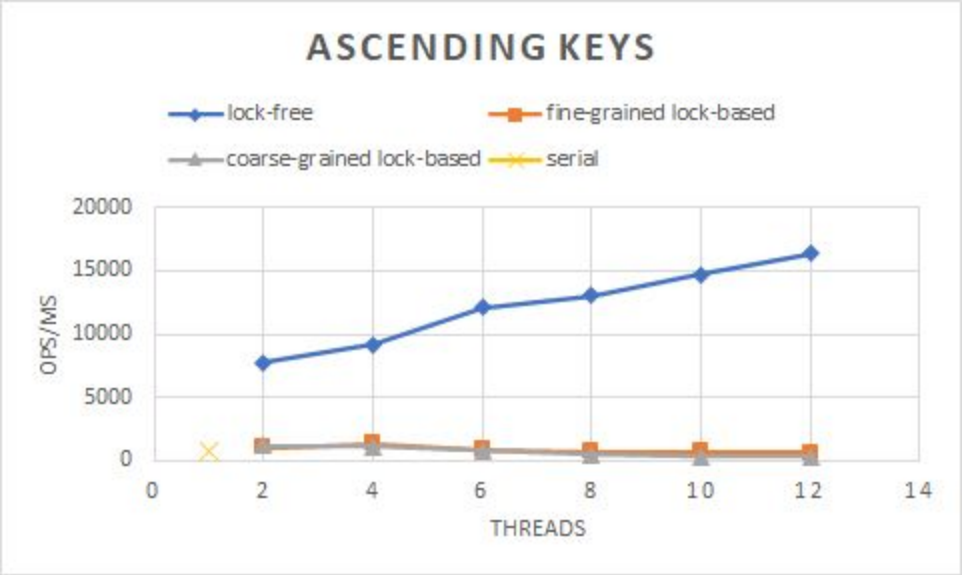
Test #7 - Ascending Keys

Description

Using the same even operation distribution as in Test #3, 30% puts, 40% finds, and 30% removes. Keys are inserted into the hash table in ascending order. This lets us observe how the location where insertion into the linked list that represents the hashtable is significant.

Observations & Analysis

The lock-free implementation sees the expected increasing speedup that was observed in Test #2, which has the same operation distribution, but similar to Test #6 and Test #5, the first data point for the lock-free implementation was noticeably higher than on a randomized workload. The reason for this is be different than the inferred reason for this occurring on Tests #5 and #6, it could be that because there are no repeated keys (strictly ascending), this results in no collisions and therefore higher performance from the start.



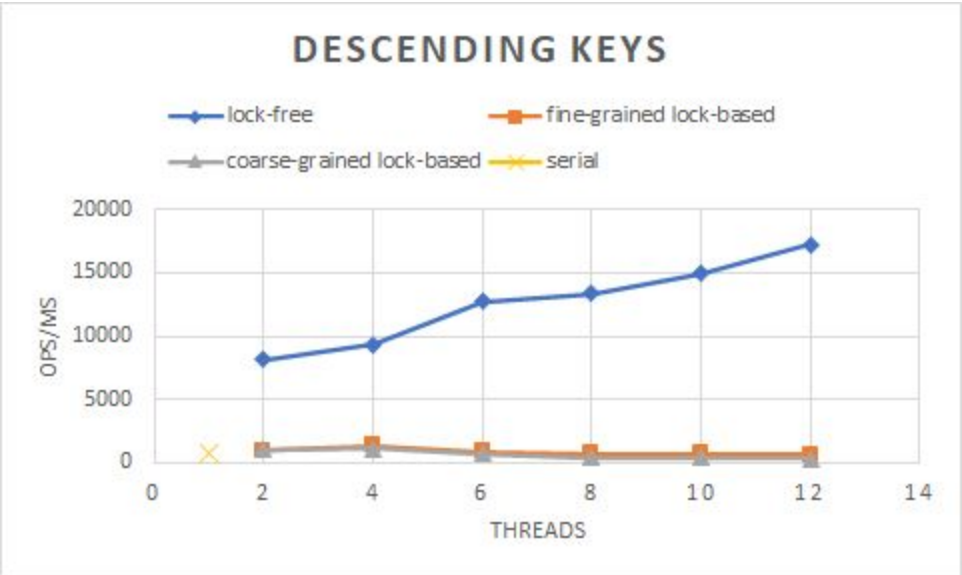
Test #8 - Descending Keys

Description

Using the same even operation distribution as in Test #3, 30% inserts, 40% finds, and 30% deletes. Keys are inserted into the hash table in descending order. This lets us observe how the location where insertion into the linked list that represents the hashtable is significant.

Observations

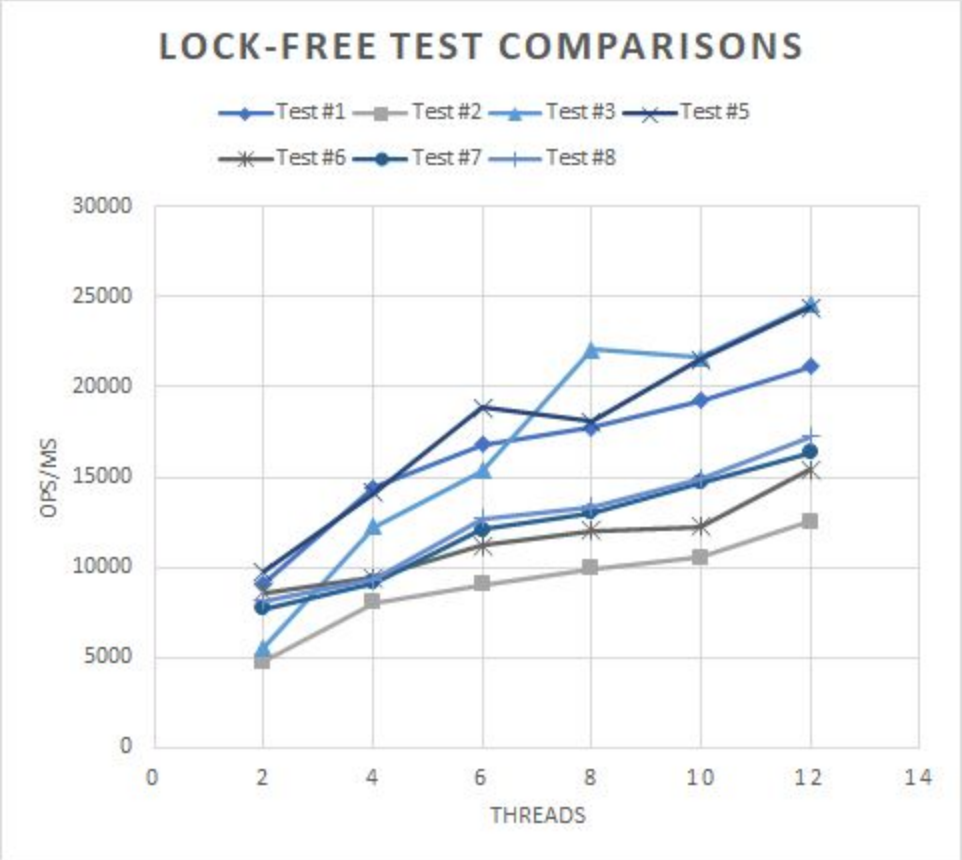
Extremely similar results to Test #7, indicating that the ops/ms are definitely affected by either ascending or descending order in a very similar fashion.



Test Comparisons

Lock-Free Performance

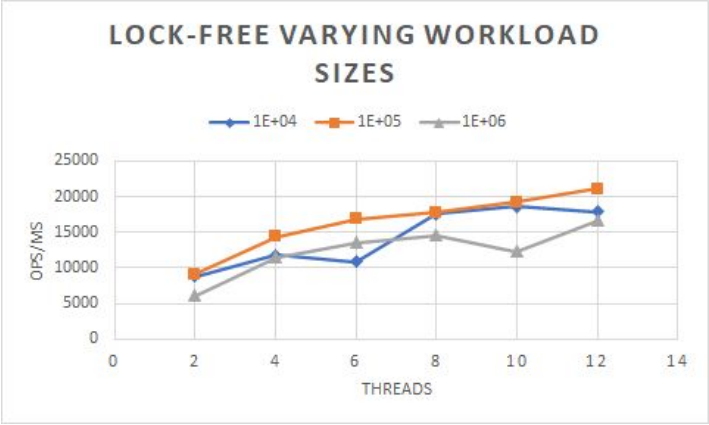
Following all the tests, we compared the performance of the lock-free implementations across all tests that varied in thread count:

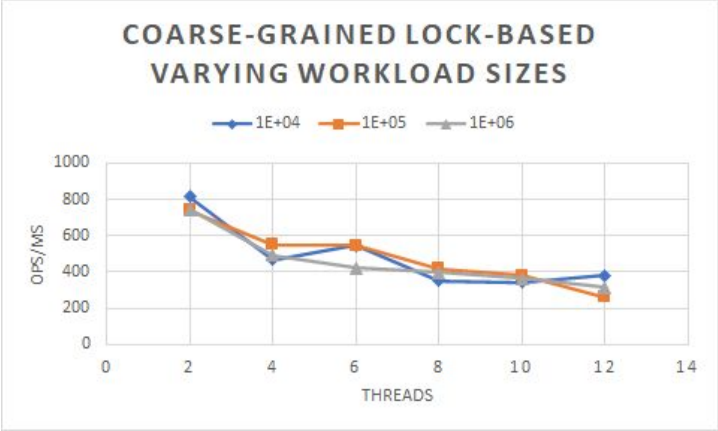


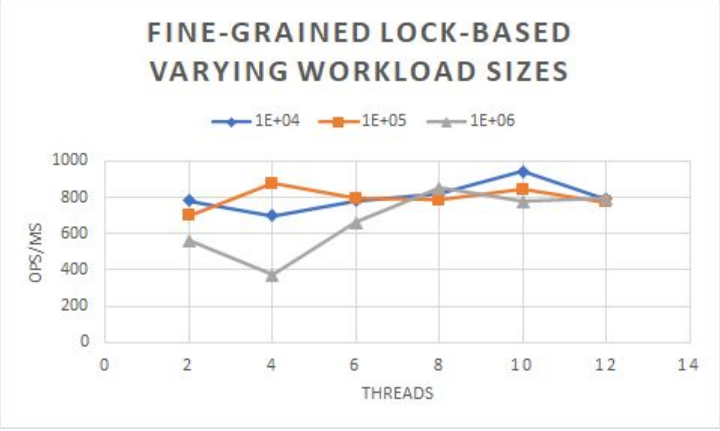
The highest performance tests were Test #5 and #3. This could indicate that insert operations are the fastest as Test #3 is 90% insertion operations. Test #5, on the other hand, while only having 30% insertion operations, had a small range of keys to insert, which is most likely the cause of it performing so well. After that is Test #1, a 10:88:2 INS:FIND:DEL operation distribution with random keys, which was our base benchmark. The other tests perform more poorly and in general indicate that on non-random workloads, skewed in some way or another, the hash table does not perform as well. Following we explore the different groups of tests in more detail.

Varying Workload Sizes

Below this are the graphs for the tests that varied in workload size. For the lock-free implementation, 1E+05 operations proved the most consistent. For the coarse-grained lock-based implementation, all three workload sizes performed rather poorly, with negative speedup, along the same trendline. Finally, for the fine-grained lock-based implementation, the trend was no speedup, but 1E+04 and 1E+05 workload sizes were most consistent to this. In this implementation as well as the other two, although most visibly the lock-free implementation, 1E+06 operations showed the most variance, possibly indicating that we've hit a memory threshold of structure size that impacts performance negatively.

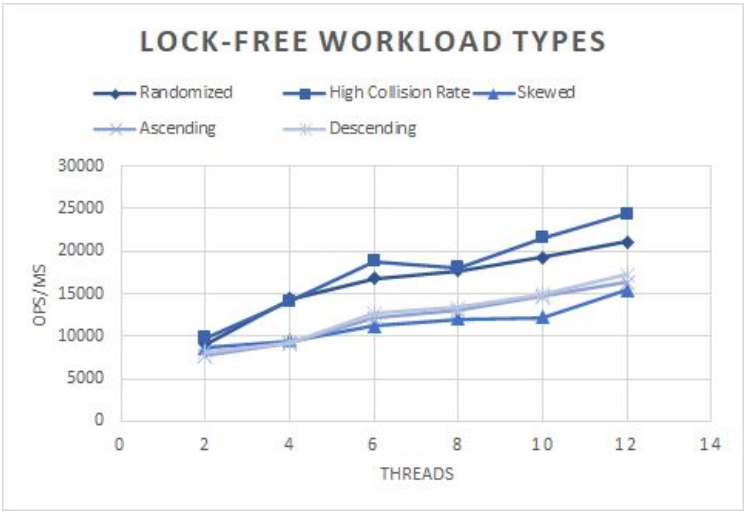


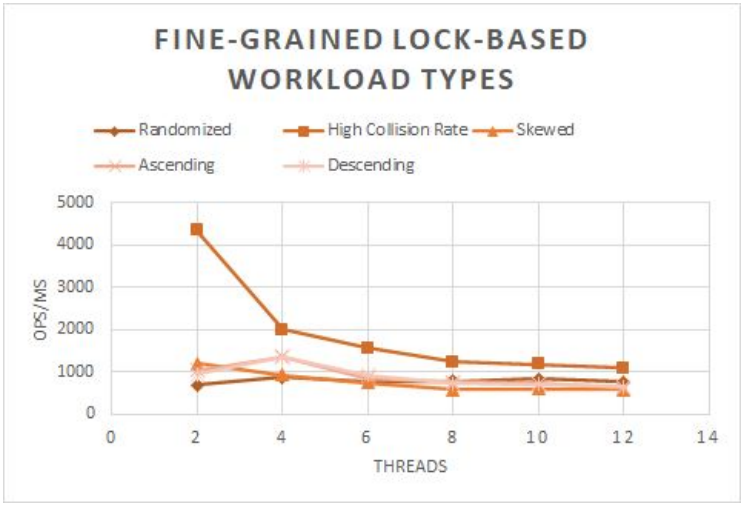


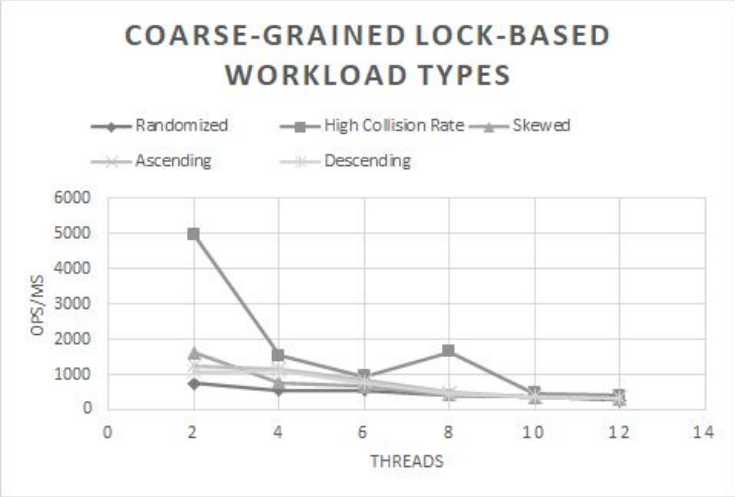


Varying Workload Types

In these comparisons, we can see two trends: The high collision rate tests perform best for all implementations and that a randomized workload performs better than the rest (besides high collision rate) only in the lock-free implementation. The very likely cause of the fast performance of the high collision rate workload is that the penalty in performance incurred by the high collision rate is offset by the performance gain in a much smaller range of keys being inserted. The randomized workload performing better on the lock-free implementation indicates that this implementation is more sensitive to the values of the keys it is given than the others, and seems to be optimized to work under an even distribution such as at random.

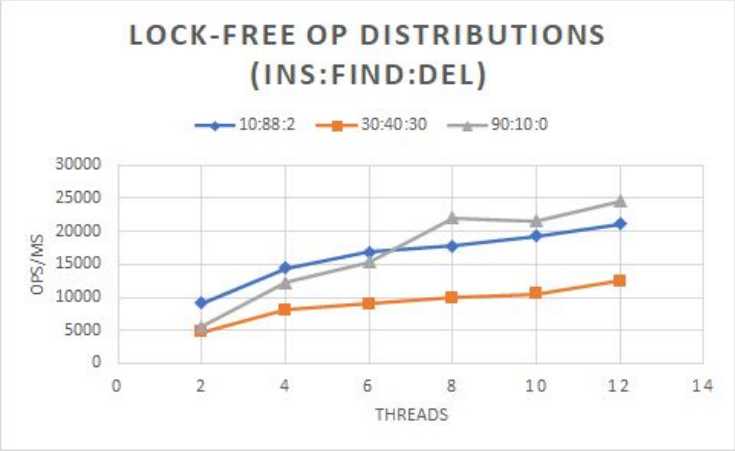


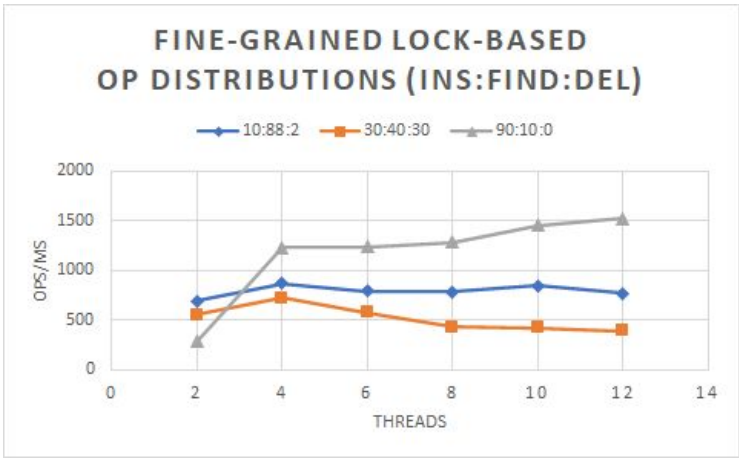


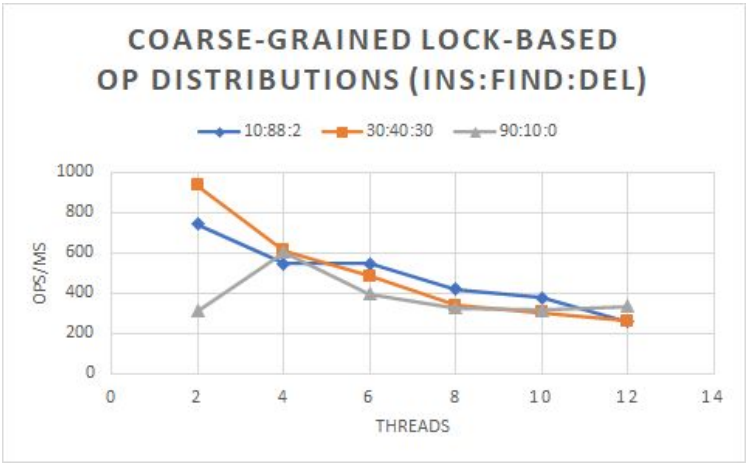


Varying Operation Distributions

By comparing each algorithm individually we can spot clear trends in this comparison. In both the lock-free implementation and the fine-grained lock-based implementation, a 10:88:2 INS:FIND:DEL operation distribution outperforms the 90:10:0 distribution at lower levels of parallelism but gets overtaken at higher levels. At the same time it always outperforms a 30:40:30 distribution. For the course-grained implementation, however, all of the distributions seem to follow the same trend besides a stark difference in speed at low parallelism, 2 threads. In this case, the 90:10:0 is still below the other distributions as it is in the other implementations. All the inputs for these tests are randomized, and thus we can conclude that as the general rule for our implementations, that inserts and finds tend to be less intensive operations than deletes for concurrent hash table implementations.







Further Experiments

Due to time constraints, we were not able to perform all the experiments we would have like to better understand the performance of our concurrent hash table implementations, but the ones we had in mind were:

More Parallelism on Dedicated Machines

Due to time constraints, we were unable to test the hash table implementations on machines with more than 12 processors. At higher parallelism, there can be more contention and those results would have provided further insight.

Synchronization Measurements

Under different workloads, we could have measured the time each implementation spent while waiting for another thread as opposed to how much time was spent executing implementation code.

Testing Against an Optimized Version of the Lock-Free Implementation

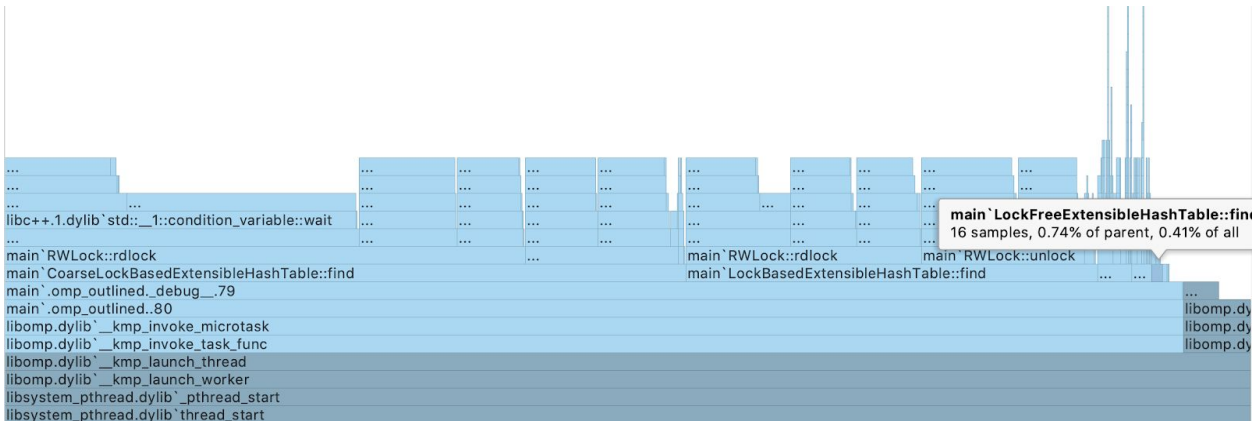
The paper by Shalev, Ori, and Nir Shavit [1] described possible optimizations to their lock-free implementation, which we were unable to completely produce due to time limitations.

Deeper Analysis

Profiling Lock-based Implementations

The experimental results consistently show that both the fine-grained lock-based and the coarse-grained lock based implementations achieved a poor throughput and do not scale well as the number of cores increase. We profiled the program to explore the bottleneck.

We obtained the following frame graph after running the three implementations with 12 threads under Test #1 setting. 64% of execution time is spent on CoarseLockBasedExtensibleHashTable::find(); 2.44% is spent on CoarseLockBasedExtensibleHashTable::put(); 29% is spent on FineLockBasedExtensibleHashTable::find(); 1.42% is spent on FineLockBasedExtensibleHashTable::put(); 0.74% is spent on LockFreeExtensibleHashTable::find().



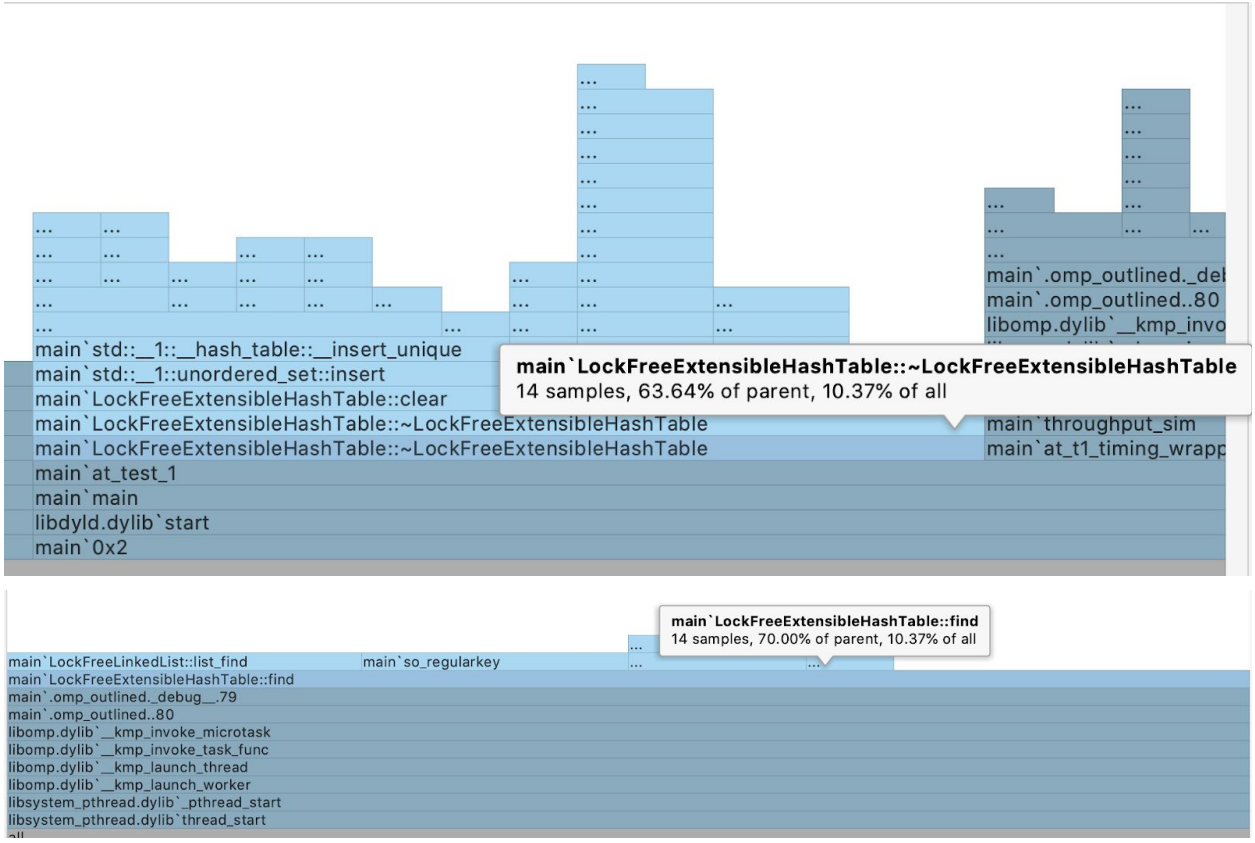
About 10% time is used to create and destroy unique_lock. This indicates that the lock implementation is inefficient. Our current read-write lock implementation prefers writers. Implementing a more efficient read-write lock in necessary for performance.

The speedup of the fine-grained lock-based implementation is also limited by the number of locks we use to protect data structures. Our implementation assigns one lock per bucket. Compared with the highly optimized lock-based implementation in the Java Concurrency Package[2], there are several ways we could improve: 1. Use a more sophisticated locking scheme that involves a small number of high level locks rather than a lock per bucket; 2. Allow concurrent searches while resizing the table, but not concurrent inserts or deletes.

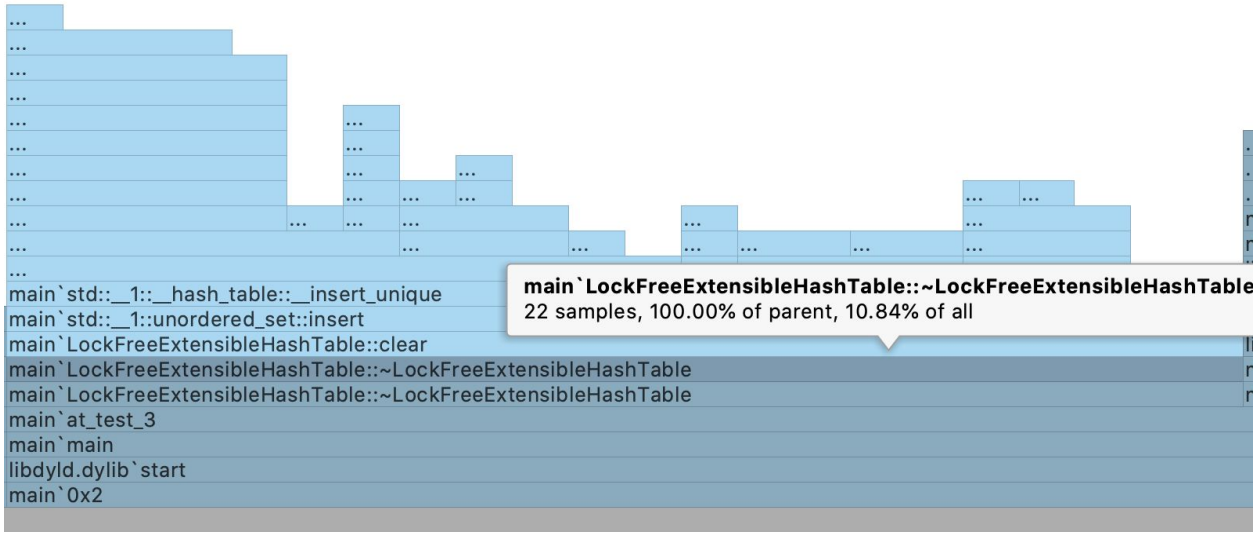
Profiling Lock-free Implementation

Our optimization flag was -O0. Once we compiled our library with an -O3 flag, we observed a throughput improvement by about 5%.

During the development process, we profiled our lock-free implementation with 6 threads under test#1 setting. About 60% of the execution time is spent on omp::wait(); 10% on LockFreeExtensibleHashTable::find(); 10% on LockFreeExtensibleHashTable::~LockFreeExtensibleHashTable(). We found that the deconstructor is taking a long time because we used an unorder_set to ensure memory safety. However, this was not necessary. Therefore, we conducted memory management more carefully in other functions and erased the use of unorder_set without hurting correctness.



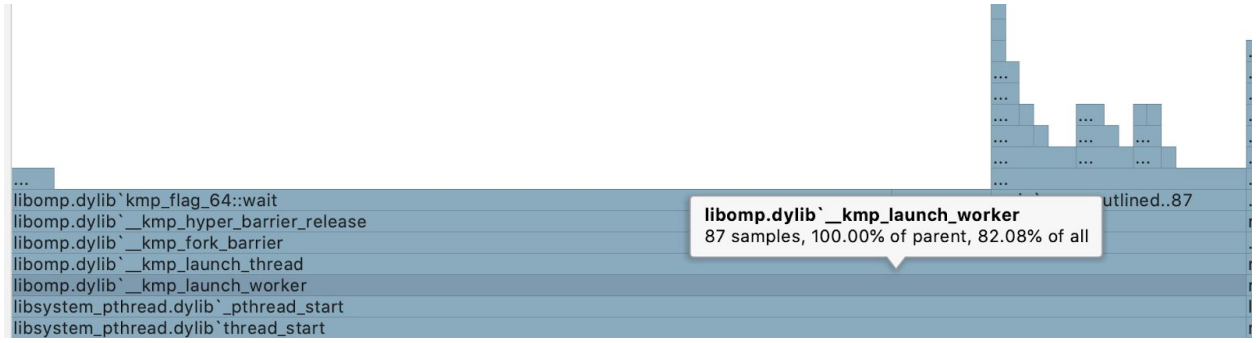
After this optimization, we profiled again and found that only 2% time is spent on the deconstructor.



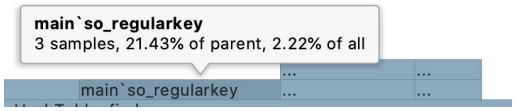
We show the detailed time distribution among the three key operations here
Put(4.72%) : so_regular_key 1.89% list_insert 0.94%, initialize_bucket 0.94%, Node constructor 0.94%
Remove(2.83%): list_delete 1.89%
Find (2.83%): list_find 2%

We also profiled the lock-free implementation on a workload containing 30% puts 30% removes 40% finds with 6 threads. The time taken by three key operations were: Put (6.90%), Find (3.3%), Remove (4.06%). Compared with test #1, Put() and Remove() took a longer time on test #2 because their percentage increased.

We found that in both test#1 and test#2, the kmp_fork_barrier in the omp library took about 80% of the execution time. That means scheduling and synchronization overhead is the bottleneck of our lock-free implementation.



Besides this, we found that the following function took nearly 2% of the total time. so_regular_key is used to reverse a 64-bit int. Therefore, improving the performance of it is a potential direction to improve the overall throughput.



Conclusion

In this project, we implemented both lock-free and lock-based extensible hash tables and found that the lock-free extensible hash table is consistently more efficient, scalable compared with the lock-based ones.

Our lock-free implementation achieved stable performance on various workload scales. It achieved better performance on typical workloads than skewed and sorted ones.

Based on profiling results, efficiency of the lock implementation and the one lock per bucket schema are the bottleneck of our fine-grained lock-based implementation

Scheduling and synchronization are the bottleneck of our lock-free implementation. Further, we can improve the efficiency of calculating split-order keys in the lock-free implementation to improve the overall performance.

List of Work by Students

List of Xinzhu's work:

- Implemented coarse-grained lock-based extensible hash table
- Implemented fine-grained lock-based extensible hash table
- Implemented lock-free extensible hash table
- Unit test for correctness
- Designed performance tests
- Optimization & Analysis
- Final poster and report (Cooperate)

List of Danié's work:

- Studying options of machines for running tests on
- Creating performance tests
- Studying and verification of implementations
- Creating graphs and Final Report Experiment Section
- Final poster and report (Cooperate)

Distribution of total credit:
60% (Xinzhu) - 40%(Danié)

Reference

[1] Shalev, Ori, and Nir Shavit. "Split-ordered lists: Lock-free extensible hash tables." *Journal of the ACM (JACM)* 53.3 (2006): 379-405.
[2] Lea, Doug. "Hash table util. concurrent. ConcurrentHashMap, revision 1.3." *JSR-166, the proposed Java Concurrency Package* (2003).