yizhenw, jiaqidon
Yizhen Wu, Jia Qi Dong

15618 Final Project Report

Summary

We implemented a concurrent hash table using linear hashing algorithm and compared performance of coarse-locked, fine-locked, and locked-free versions of the algorithm. We performed experiments on an 8-core Intel Core i7 machine and demonstrated that the lock-free version can achieve 8x speedup compared to lock-based versions in most workloads.

Background

The scenario for our project will be multiple threads trying to access and update the same hash table. This can happen in parallel applications where different threads try to perform read and write to a hash table concurrently. This can also occur in database systems where multiple queries access an index backed by linear hashing concurrently.

Linear hashing is a dynamic data structure which implements a hash table that grows or shrinks as keys are inserted or deleted. Keys are placed into fixed-size buckets and a bucket can be redistributed when overflow occurs. In splitting-round i, the data structure makes use of two hash functions, h_i and h_{i+1}, where $h_k(x) = x \% 2^i$. It also makes use of a split-pointer, p, called next split pointer, that points to the next bucket to be split if overflow occurs. Any keys that hash to buckets before the split pointer use the first hash function to locate the hash bucket, otherwise it uses the second hash function. Compared to extendible hashing, which doubles the number of the buckets in the table whenever there is a split, linear hashing only increases the number of the buckets by one and splits the bucket at split pointer. However, compared to extendible hashing which only preserves one global variable among different threads, linear hashing would need to maintain four different global variables: splitting round, number of elements, number of buckets, and split pointer.

The hash table supports three key operations: insert (where a key-value pair is inserted into the hashtable), get (where the value corresponding to a given key is retrieved), and remove (where a key-value pair is removed from the hash table).

In our workload, multiple threads are concurrently inserting, removing, or retrieving key-value pairs from the hash table. Each of these operations could resize table directory size and increase or decrease the number of buckets. Therefore, this algorithm is not data parallel. We focus our efforts primarily on efficient synchronization of concurrent accesses to the data structure. In particular, multiple threads making updates of a hash table can benefit significantly

from parallelizing accesses and updates from different threads. Our goal is to maximize concurrency of threads adding and reading data from the same hash table without being blocked by other threads. We also need to ensure hash table correctness such that the hash table ends up in a consistent state after a series of modifications.

Approach

Coarse-grained Locking

In coarse-grained locking version of linear hashing, we make use of a mutex in get, insert, and remove operations to ensure that only a single thread can be accessing the entire data structure at one time. This is very inefficient as all other threads need to wait for one thread to complete its operation before they can proceed.

Fine-grained Locking

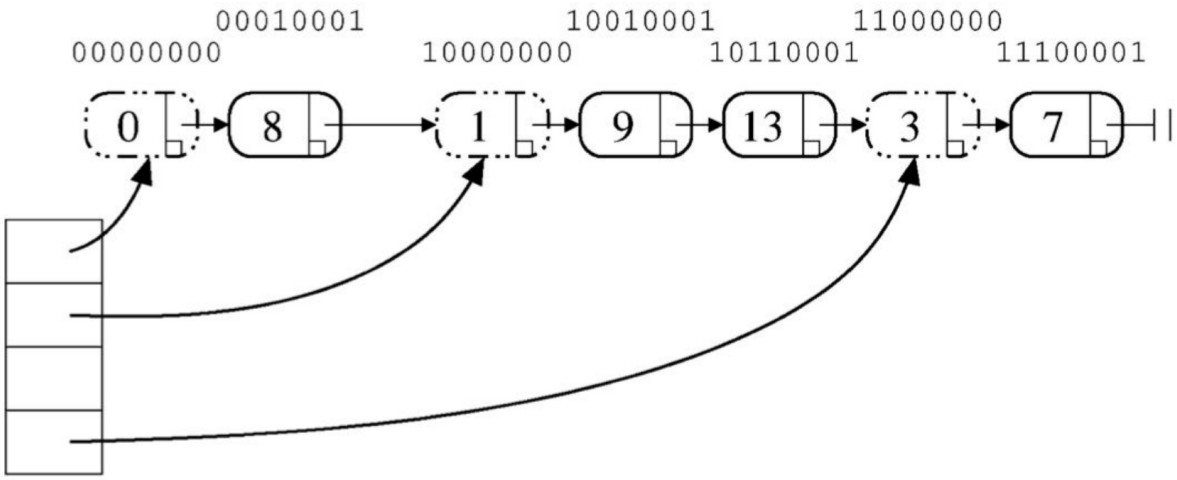
In fine-grained locking of linear hashing, we make use of read-writer lock to enable more parallelism.

The get operation first obtains a read lock on the bucket directory and retrieves the correct bucket. It then obtains a read lock on the bucket and retrieves the value corresponding to the key. Notice that multiple get operations can proceed in parallel because reader-writer lock allows multiple readers to advance simultaneously.

The insert operation first obtains a read lock on the bucket directory and retrieves the correct bucket. It then obtains a write lock on the bucket to gain exclusive access to the bucket and inserts a key value pair into the bucket. Then, it checks whether the bucket overflows and if so, upgrades the read lock on the bucket directory to a write lock. With exclusive access to the bucket directory, the thread updates the split pointer, round counter, and number of elements of the hash table. It also creates a new bucket and obtains a write lock on it, then redistributes the content of the overflowing bucket with this new bucket.

Lock-Free Algorithm

In order to make a linear hash table lock free, we take the idea from Ori Shalev and Nir Shavit. Their paper proposes the idea of a split ordered list extensible hash table. In traditional extendible hashing algorithm, each time a new element is added, it is added to a bucket, and when a split occurs, the bucket number will be doubled, and all buckets will need to redistribute the items inside of them. However, Ori and Nir reversed this step. Instead of assigning key value pairs to buckets, all key value pairs are stored in a singly linked list. Each time a bucket is split, a bucket node will be inserted in its corresponding location in the linked list. Redistribution of bucket contents now is as simple as inserting a node in the correct location to the linked list. The binary reversal of keys are stored as the actual search keys of each node in the linked list. This is referred to as a split order of the keys. Using split order ensures that inserting a bucket starting point to the linked list will just be a sequential search of its position.



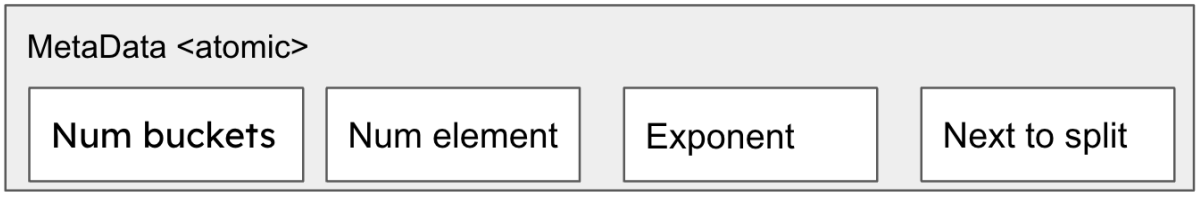
In the figure given above, the nodes with dashed lines are bucket pointers, whereas nodes with solid lines are keys stored in the hash table. The linked list maintains order based on binary reversal of each key – for example, 8's binary representation is 1000, which is stored as 0001 in the linked list. Notice that the least significant bit of this reversal key is 1. This is to differentiate between dummy nodes (bucket pointers) whose LSB is 0, and key nodes whose LSB is 1.

Our approach is based on this method, but we made several important modifications in order to adapt this data structure for a linear hashing algorithm. The next section explains in detail the split-ordered based linear hashing algorithm.

Initialize

We use a split-order based linked list to store all data inserted into the hash table and also a bucket directory that stores a list of bucket pointers. These pointers point to a dummy node in the linked list that represents the starting point of a bucket. Initially Bucket 0 is initialized when the hash table is first created. To distinguish between bucket nodes and data nodes, we mark the last bit of the bucket's hash value to be 0, and all other last bits of data nodes' hash value as 1.

In order to be able to atomically update four variables which will be used during linear hashing, our solution is to put them all in one variable. We use a $uint64$ integer, and divide it into four parts: first 16 bits for number of buckets, second 16 bits for number of elements, third 16 bits for exponents which decide the version of hash function currently used by the hash table, and last 16 bits for the next bucket to split. The bits can be altered according to different use cases. For example, if the default bucket size is very big, then we can distribute more bits to the number of elements, and less bits to the number of buckets and the next bucket to split.



Insert

When inserting a new element to the hash table, we follow the algorithm of linear hashing:

1. Find the bucket:
   $bucket = hash_i(x);\ if\ bucket < next\ bucket\ to\ split,\ bucket = hash_{i+1}(x)$
   $bucket\ address = bucket\ list[bucket]$
2. Find the hash value of the current insert:
   $hashval = reverse\ binary(x);$
3. Starting from $bucket\ address$, insert the $hashval$ in ascending order.
4. If $current\ size * load\ factor > bucket\ num * bucket\ size$, split $next\ bucket\ to\ split$; if $next\ bucket\ to\ split = 2^i - 1$, reinitialize $next\ bucket\ to\ split = 0,\ i = i + 1$
5. When splitting, $next\ bucket\ to\ split$ is splitted into $next\ bucket\ to\ split$ and $new\ bucket$, which equals to $num\ of\ buckets$ (in our context, both $next\ bucket\ to\ split$ and $num\ of\ buckets$ all refers to the bucket index which starts at 0); and by the linear hashing algorithm, during each split,
   $next\ bucket\ to\ split\ \%\ 2^i == new\ bucket\ \%\ 2^i,$
   $next\ bucket\ to\ split\ \%\ 2^{i+1} + 2 == new\ bucket\ \%\ 2^{i+1},$
   This means that for all the values now hashed to the $next\ bucket\ to\ split$ but not $new\ bucket$, their binary reversal will be less than the values that previously hashed to the $next\ bucket\ to\ split$ and now can also be hashed to $new\ bucket$. And thus inserting the $new\ bucket$ dummy node will be as simple as inserting a new element in order starting from $next\ bucket\ to\ split$.
6. After splitting, one thread will use CAS to update the atomic variables, and add the $new\ bucket$ to $bucket\ list$.

Find

We follow linear hashing algorithm to find an element in the hash table:

1. Use a copy of current metadata to find the element's current bucket
2. Starting from the location of the bucket, sequentially compare the value of the elements in the bucket

Remove

Since using a lock-free linked list, remove is simpler than original algorithm of linear hashing:

We just simply find which bucket the element is in, and ask the lock free linked list to remove the element starting with the bucket dummy node.
Remove is not done at once. In order to avoid ABA problem, we followed the method proposed first by Harris: first mark the element to be removed in the list as removed, then later when another element will be inserted in, the linked list will notice the logically removed element and remove it. In this way no element will be inserted after the removed element.

Language

We used C++ to implement all the hash tables. C++ suits our need for using CAS operations, and is also able to atomically mark pointer addresses in remove.

Machine Model:

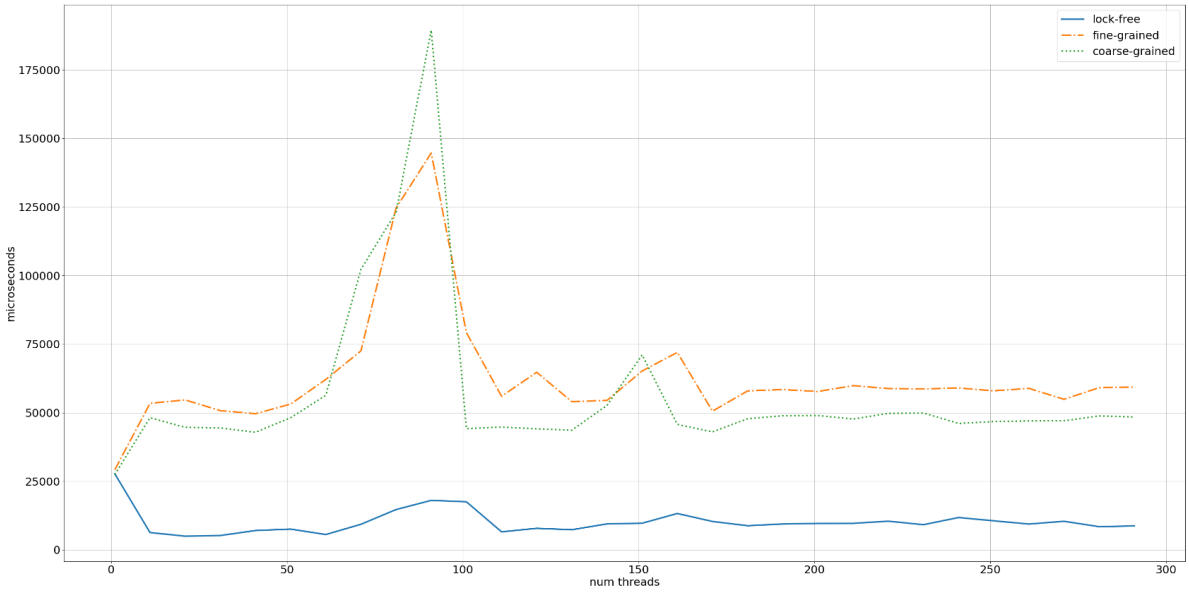
Intel(R) Core(TM) i7-9700 CPU

Our data structure is designed to be used in a multi-thread situation, which can be tested by calling std::thread library from C++ and performing concurrent operations.

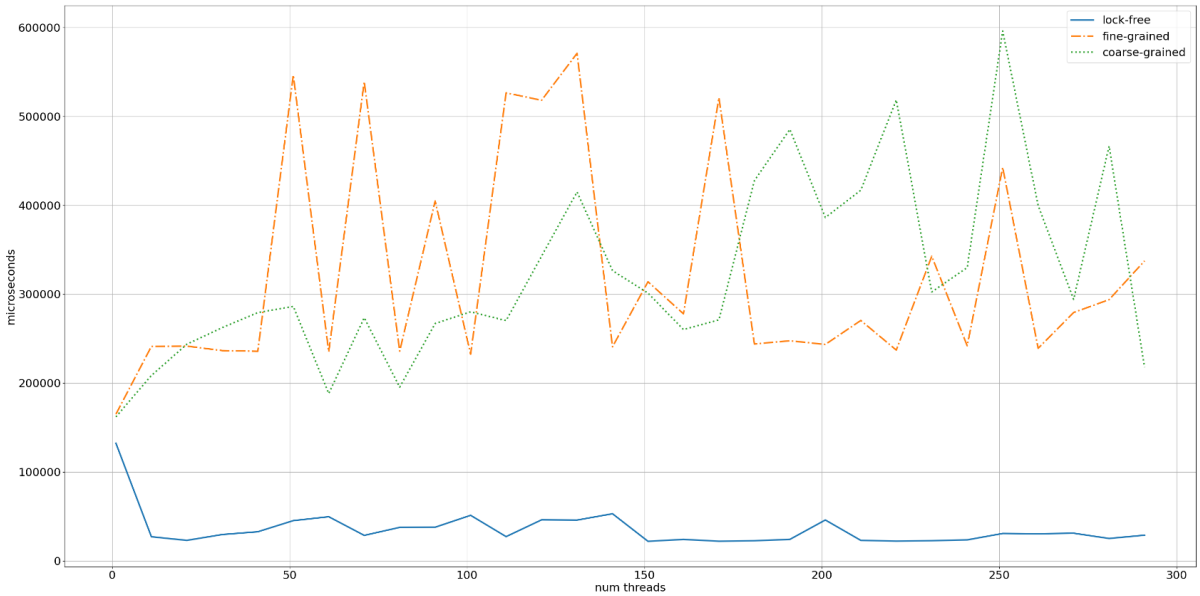
Result

We first measured the performance of insert-heavy workloads, with bucket sizes of 100 and 1000.

- Insert_only_100



- Insert_only_1000



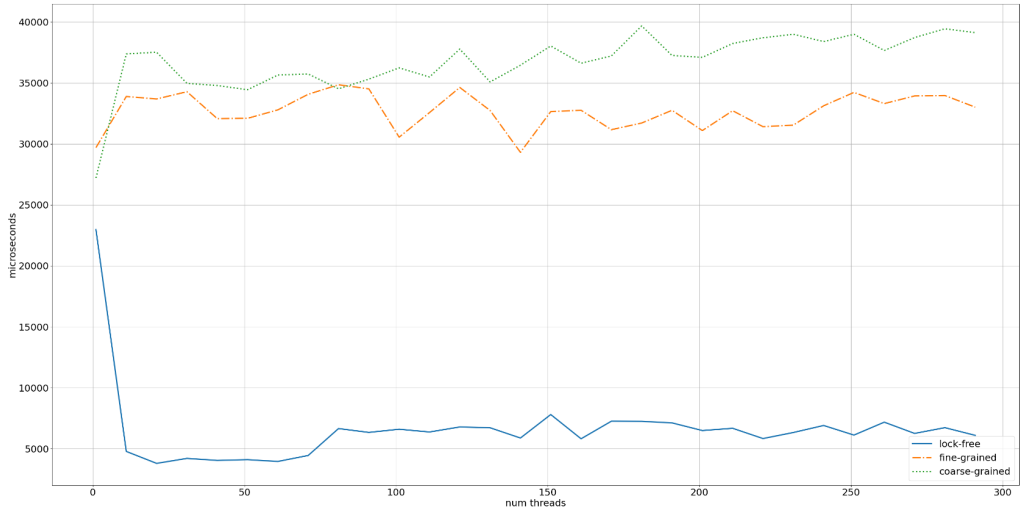
From these graphs we can see that lock free hash table perform better than both fine grained hash table and coarse grained hash table among all tasks. For coarse grained and fine grained locked hash table, the increase in thread number does not necessarily reduce the overall time they spent on the same tasks due to multiple threads trying to grab the same locks. However,

for lock free hash table, the overall time is dramatically reduced when threads number increases from 1 to 11, and the time spent on the tasks remains faster than when only one thread is performing the task.

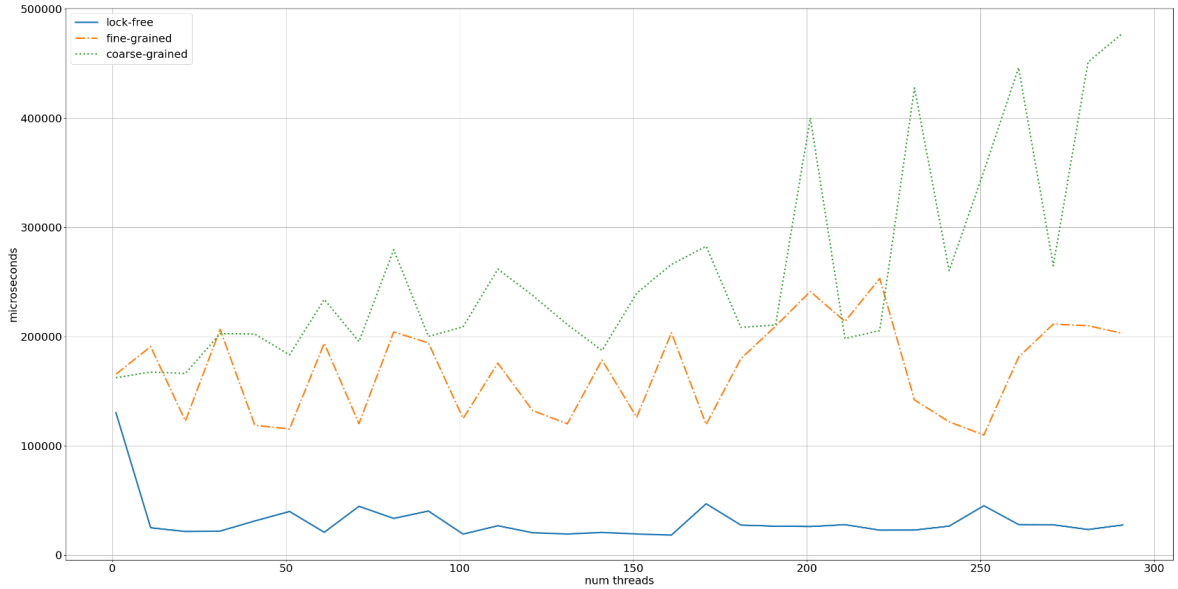
There are several interesting observations to be made from these two graphs. First, the lock-free version of the algorithm is significantly faster than lock-based versions, and its performance continuously improves as the number of threads increases. Second, the runtime for bucket size of 1000 is in general much higher than runtime for bucket size of 100. This is because each insert operation performs a linear search first, and a bigger bucket size results in a much costlier search. Further, a bigger bucket size means that more keys can be grouped under one bucket, resulting in higher contention. Third, performance is heavily influenced by redistribution of buckets for lock-based versions. Redistributing buckets with 1000 elements is much slower than redistributing buckets with 100 elements.

We also compared the performance of lock-free hash table vs fine-grained hash table and coarse grained hash table over two additional tasks: insert and get with bucket size of 100 and 1000, and insert and remove with bucket size of 100 and 1000.

- Insert_and_get_100

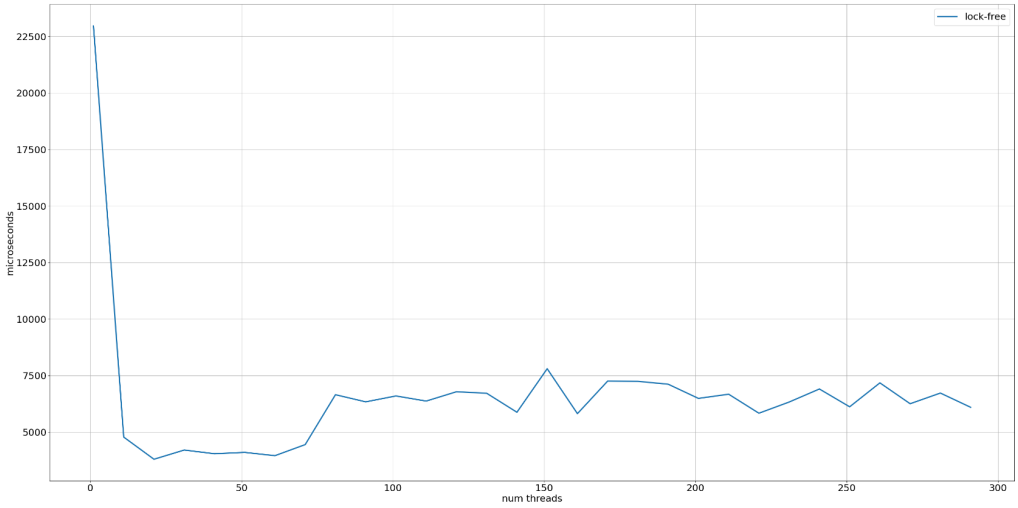


- Insert_and_get_1000



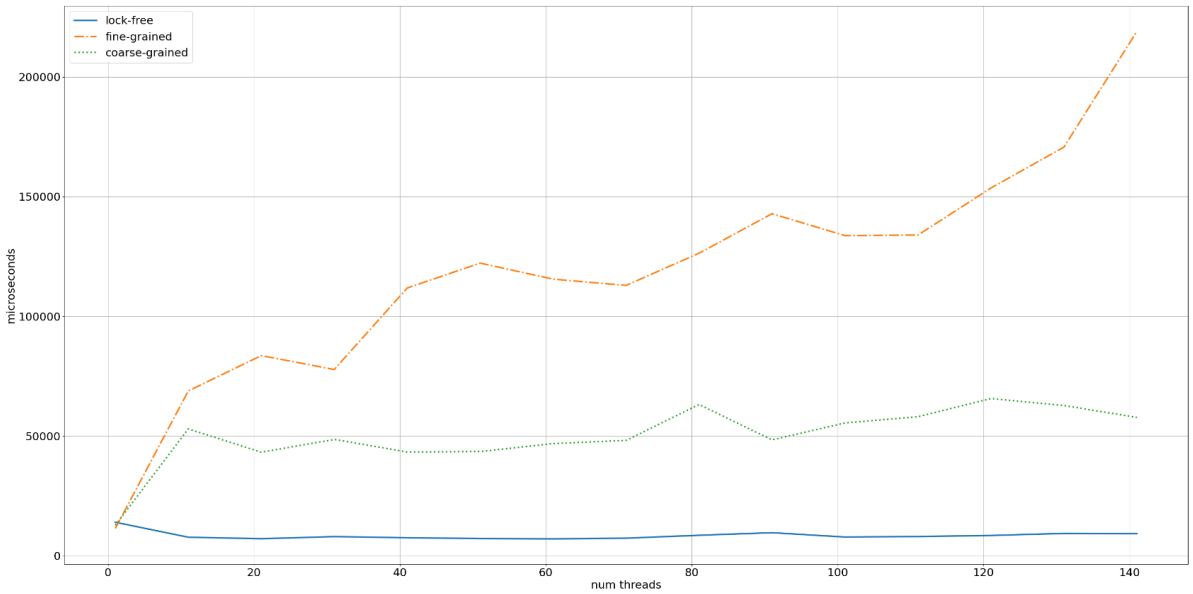
We can see that for lock-based versions, fine-grained locking is consistently more efficient than coarse-grained locking. This is because fine-grained locking allows concurrent execution of get operations and more parallelism in insert operations. Lock-free version is still much more efficient than lock-based versions.

- Insert_and_get_100 lock free

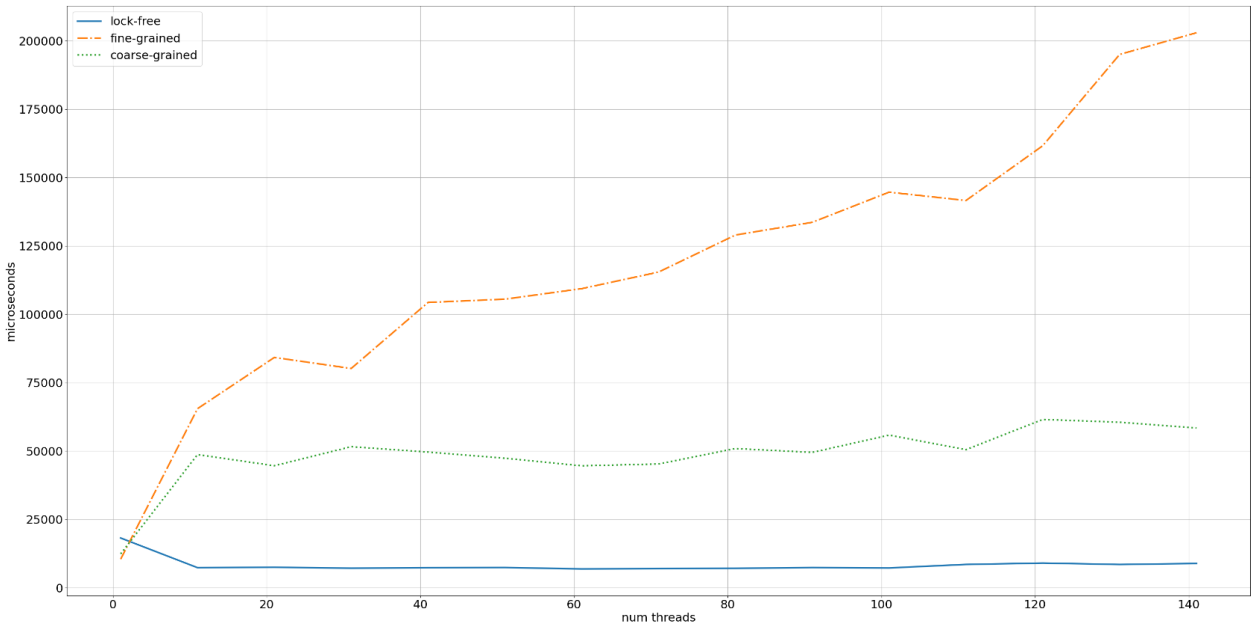


Looking at the lock free version alone, we see that performance drastically improves for lower number of threads (10-60) threads. When more threads are used, there is greater contention of CAS operations among the threads and therefore performance suffers as a result.

- Remove_insert_100



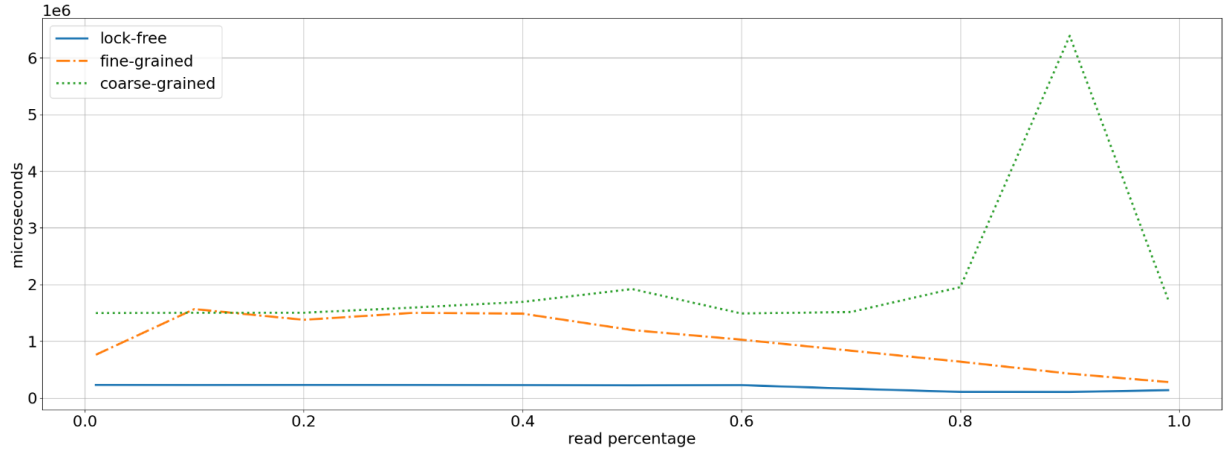
- Remove_insert_1000



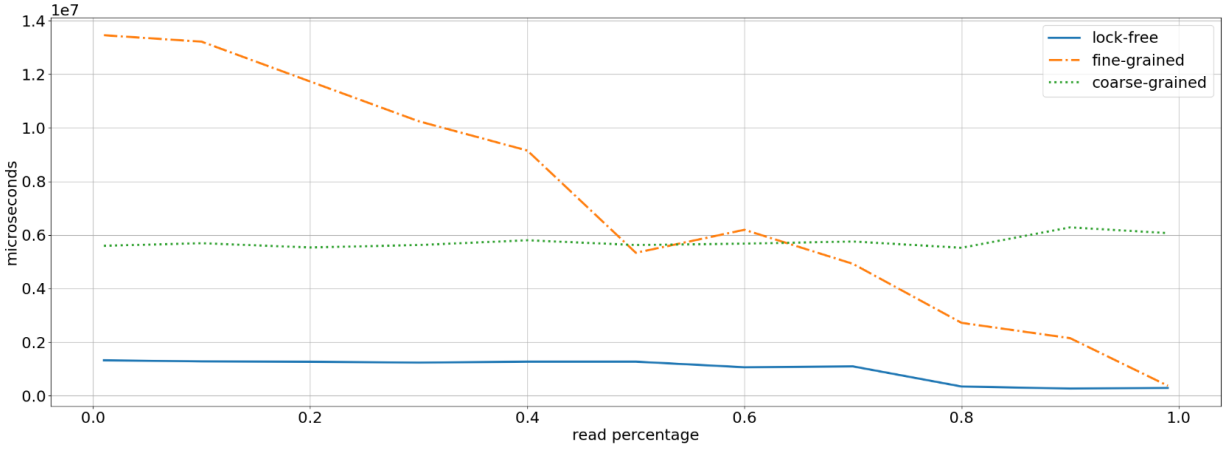
For removal, lock-free hash table can also maintain the benefit brought by using multiple threads to perform a task, while for coarse grained and fine grained locked table, the time spent on waiting for turns to remove elements did not decrease with the increase of thread number. This is because we did not implement shrinking of buckets. Coarse grained locked table tends to spend similar time on remove after thread number increase over 11, and fine grained locked table tends to spend more time with the increase of thread number.

We also measure the performance of our hash tables under different workloads by varying read-write ratio. With a low read-write ratio, we simulate a write-heavy workload, whereas a high read-write ratio simulates a read-heavy workload. In the following graphs, given a fixed number of operations, x axis represents the percentage of read operation in these graphs and the rest of the operations are writes (increase and remove).

- Reader_writer_100



- Reader-writer_1000



For reader-writer with bucket size of 1000, we see a decreasing trend for fine-grained version as read ratio increases. For a write heavy workload (read percentage <= 0.4), most writes happen in the same bucket since bucket size is large. This causes multiple write threads to contend for the same write lock on the same bucket, resulting in greater overhead as additional locks are acquired to redistribute buckets. As read percentage increases and we transition into a read heavy workload, the performance benefit of read lock is demonstrated, since multiple readers

can proceed in parallel, thus reducing the overall runtime to be less than coarse-grained locked version. We also see that the execution time for coarse-grained lock is independent from read-write ratio. This is because a single mutex is obtained for each get, insert, and write operation, therefore only one operation can proceed at one time. For lock free version, we see that better performance is observed with a high read ratio. This is also expected since write operations are based on CAS operations on linked list nodes and these CAS is more likely to fail when there are more operations that modify the hashtable.

Perf Stats

To better understand performance bottlenecks of each version of the algorithm, we run perf command on a 50% read, 50% write workload.

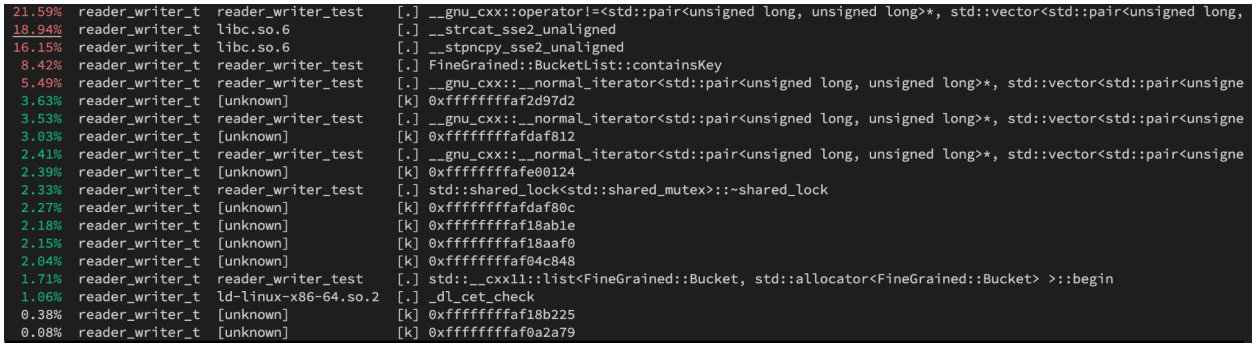
Lock free

```
21.77%  reader_writer_t  reader_writer_test  [.] LFLinkedList::find_internal_helper
21.73%  reader_writer_t  reader_writer_test  [.] MarkPtrType::MarkPtrType
10.28%  reader_writer_t  reader_writer_test  [.] MarkPtrType::equal
10.15%  reader_writer_t  reader_writer_test  [.] MarkPtrType::mark_delete
 6.42%  reader_writer_t  reader_writer_test  [.] LFHashTable::reverse_bits
 4.21%  reader_writer_t  reader_writer_test  [.] std::bitset<16ul>::bitset
 3.93%  reader_writer_t  reader_writer_test  [.] LFLinkedList::find_internal
 3.33%  reader_writer_t  reader_writer_test  [.] std::_Base_bitset<1ul>::_Base_bitset
 3.21%  reader_writer_t  [unknown]           [k] 0xffffffffaf2c0d41
 2.63%  reader_writer_t  [unknown]           [k] 0xffffffffafd9d81e
 2.45%  reader_writer_t  reader_writer_test  [.] MarkPtrType::get_node
 2.41%  reader_writer_t  reader_writer_test  [.] HashTable::hash
 2.14%  reader_writer_t  [unknown]           [k] 0xffffffffaf2c28fe
 1.95%  reader_writer_t  reader_writer_test  [.] MetaData::atomic_set_data
 1.73%  reader_writer_t  [unknown]           [k] 0xffffffffaf6a1dd7
 1.16%  reader_writer_t  [unknown]           [k] 0xffffffffaf39da86
 0.36%  reader_writer_t  reader_writer_test  [.] MarkPtrType::is_deleted
 0.05%  reader_writer_t  [unknown]           [k] 0xffffffffaf0a2a79
 0.02%  reader_writer_t  [unknown]           [k] 0xffffffffaf0a2a7c
```

Find_internal_helper takes the most amount of time. This function attempts to locate the correct node within the linked list. It is the bottleneck of lock free version because under high contention, using CAS to change pointers to linked list is very likely going to fail, triggering a repeated find operation, which significantly increase execution overhead. Starvation can also occur if one thread has to repeat find operation multiple times due to other threads always completing insert or delete before it can.

An interesting observation is that a significant amount of computations (6.42%) is spent on reverse_bits function which simply reverses the bits of the key. We can potentially optimize this time away by creating a large hash table that contains the bit reverse of all possible keys in the key space.

Fine Grained

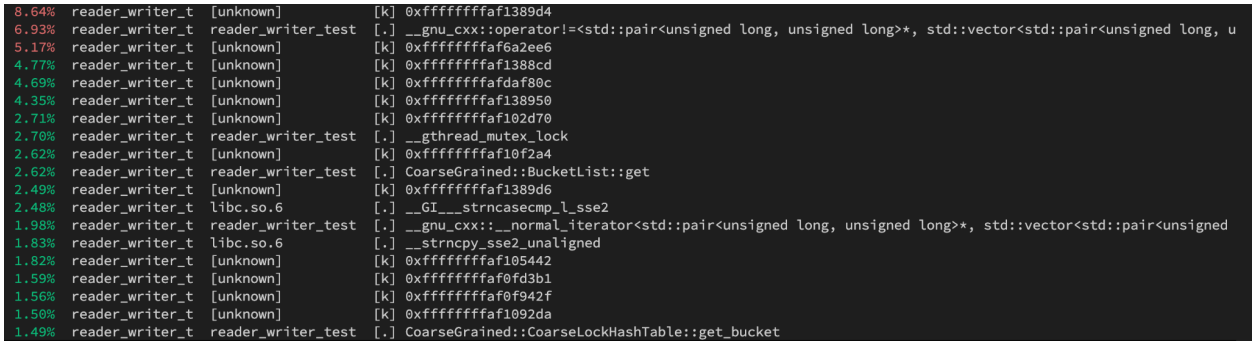


```
21.59%  reader_writer_t  reader_writer_test    [.] __gnu_cxx::operator!=<std::pair<unsigned long, unsigned long>*, std::vector<std::pair<unsigned long,
18.94%  reader_writer_t  libc.so.6             [.] __strcat_sse2_unaligned
16.15%  reader_writer_t  libc.so.6             [.] __stpncpy_sse2_unaligned
 8.42%  reader_writer_t  reader_writer_test    [.] FineGrained::BucketList::containsKey
 5.49%  reader_writer_t  reader_writer_test    [.] __gnu_cxx::__normal_iterator<std::pair<unsigned long, unsigned long>*, std::vector<std::pair<unsigne
 3.63%  reader_writer_t  [unknown]             [k] 0xffffffffaf2d97d2
 3.53%  reader_writer_t  reader_writer_test    [.] __gnu_cxx::__normal_iterator<std::pair<unsigned long, unsigned long>*, std::vector<std::pair<unsigne
 3.03%  reader_writer_t  [unknown]             [k] 0xffffffffafdaf812
 2.41%  reader_writer_t  reader_writer_test    [.] __gnu_cxx::__normal_iterator<std::pair<unsigned long, unsigned long>*, std::vector<std::pair<unsigne
 2.39%  reader_writer_t  [unknown]             [k] 0xffffffffafe00124
 2.33%  reader_writer_t  reader_writer_test    [.] std::shared_lock<std::shared_mutex>::~shared_lock
 2.27%  reader_writer_t  [unknown]             [k] 0xffffffffafdaf80c
 2.18%  reader_writer_t  [unknown]             [k] 0xffffffffaf18ab1e
 2.15%  reader_writer_t  [unknown]             [k] 0xffffffffaf18aaf0
 2.04%  reader_writer_t  [unknown]             [k] 0xffffffffaf04c848
 1.71%  reader_writer_t  reader_writer_test    [.] std::__cxx11::list<FineGrained::Bucket, std::allocator<FineGrained::Bucket> >::begin
 1.06%  reader_writer_t  ld-linux-x86-64.so.2  [.] _dl_cet_check
 0.38%  reader_writer_t  [unknown]             [k] 0xffffffffaf18b225
 0.08%  reader_writer_t  [unknown]             [k] 0xffffffffaf0a2a79
```

Comparison operation of key value pairs account for the most computations. This is because get, insert, and remove operations all require linear traversal of buckets to find the correct element, which results in high search time. The search algorithm is linear search for each bucket, and this results in high search cost for buckets with larger sizes. Notice that search time is an indirect representation of synchronization stall. When a thread walks through a bucket list in an attempt to find elements, it prevents other threads from performing the same operation.

To optimize the search cost of buckets, a bloom filter can be used. When a key is to be searched within a particular bucket, it first consults the bloom filter to determine whether the key exists in the bucket before actually traversing through it. This can significantly reduce the search time.

Coarse grained



```
8.64%  reader_writer_t  [unknown]           [k] 0xffffffffaf1389d4
6.93%  reader_writer_t  reader_writer_test  [.] __gnu_cxx::operator!=<std::pair<unsigned long, unsigned long>*, std::vector<std::pair<unsigned long, u
5.17%  reader_writer_t  [unknown]           [k] 0xffffffffaf6a2ee6
4.77%  reader_writer_t  [unknown]           [k] 0xffffffffaf1388cd
4.69%  reader_writer_t  [unknown]           [k] 0xffffffffafdaf80c
4.35%  reader_writer_t  [unknown]           [k] 0xffffffffaf138950
2.71%  reader_writer_t  [unknown]           [k] 0xffffffffaf102d70
2.70%  reader_writer_t  reader_writer_test  [.] __gthread_mutex_lock
2.62%  reader_writer_t  [unknown]           [k] 0xffffffffaf10f2a4
2.62%  reader_writer_t  reader_writer_test  [.] CoarseGrained::BucketList::get
2.49%  reader_writer_t  [unknown]           [k] 0xffffffffaf1389d6
2.48%  reader_writer_t  libc.so.6           [.] __GI___strncasecmp_l_sse2
1.98%  reader_writer_t  reader_writer_test  [.] __gnu_cxx::__normal_iterator<std::pair<unsigned long, unsigned long>*, std::vector<std::pair<unsigned
1.83%  reader_writer_t  libc.so.6           [.] __strncpy_sse2_unaligned
1.82%  reader_writer_t  [unknown]           [k] 0xffffffffaf105442
1.59%  reader_writer_t  [unknown]           [k] 0xffffffffaf0fd3b1
1.56%  reader_writer_t  [unknown]           [k] 0xffffffffaf0f942f
1.50%  reader_writer_t  [unknown]           [k] 0xffffffffaf1092da
1.49%  reader_writer_t  reader_writer_test  [.] CoarseGrained::CoarseLockHashTable::get_bucket
```

For coarse grained locked table, it is interesting to notice that mutex_lock accounts for a significant portion of runtime (2.7%). This shows the inefficiency of mutex in high contention scenarios.

Reference

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In Proceedings of the 15th International Conference on Distributed Computing (DISC '01). Springer-Verlag, Berlin, Heidelberg, 300–314.

Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. J. ACM 53, 3 (May 2006), 379–405. https://doi.org/10.1145/1147954.1147958.

List of Work

Jia Qi Dong

- Implement fine-grained locking algorithm
- Design C++ CAS primitives
- Implement lock-free linked list and write unit test, stress tests, and integration tests
- Implement lock-free linear hashing and write unit test, stress tests, and integration tests
- Write report
- Draw graphs for final analysis

Yizhen Wu

- Implement coarse-grained locking algorithm
- Write unit tests, correctness tests, and benchmark tests for lock-based algorithms
- Develop benchmark tests and extensive experiment tests for lock-free
- Implement lock-free linear hashing remove functionality
- Draw graphs for final analysis
- Write report

Distribution of total credit:
50% (Jia Qi Dong); 50%(Yizhen Wu)