Lock-free Skip List

15418/15618 - Parallel Computer Architecture and Programming Project

Chen He, Yida Wu

Summary

We implemented three versions of the skip list including a coarse-grained lock-based version using a global mutex, a fine-grained lock-based version based on multi-level locking and a lock-free version using CAS. Given the experimental results on a 8-core shared memory multiprocessor, we demonstrate the performance of the fine-grained lock-based version and lock-free version and their benefits and drawbacks.

Project URL: https://supertaunt.github.io/CMU_15618_project.github.io/

Background

Our goal is to explore parallelism and concurrency in skip list data structure.

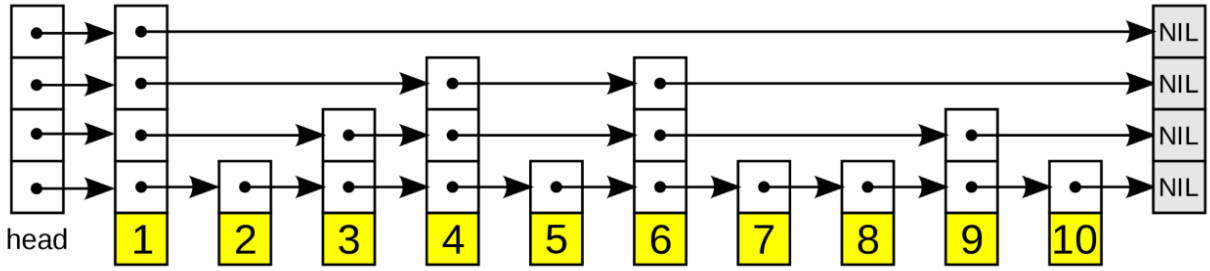
Skip list is a relatively novel data structure and it has O (log n) search, insert and delete cost while keeping the elements in order. The skip list is also a potential replacement of red black tree in high concurrency scenarios. This is because that red black needs to ensure the ownership of key data in the root node for every write (modification) operation, otherwise, the red black tree cannot keep its defined data structure. On the contrary, the skip list does not have a global share key data node, thus skip list can do multiple write operations at the same time without locking a share variable. Only the neighboring nodes matter to each other. Therefore, skip lists are quite competitive with respect to the red black tree in high concurrency scenarios, especially in large concurrent write cases. This is the main parallelism aspect we want to explore on the skip list.

The dependencies in our design is that each node in the skip list is dependent on previous linked nodes. Since other non-linked nodes would not affect the structure of the current node, there is no dependency between them. This is the challenging part of the problem, we overcome this problem by designing various features in different skip list implementations. Another dependency is that read operation depends on atomicity of several key variables like lock in coarse grained cases and atomic variable in fine-grained and lock free cases.

The theoretical maximum parallelism is half the size of the highest level linked node for both our designed fine-grained and lock-free skip list. Since we need to keep data consistent in both current and previous same level nodes. And the highest level has the minimum nodes, and thus the parallelism is limited by the highest level.

The skip list is inherently with poor spatial locality as linked lists, but could benefit from temporary locality, especially on the high level nodes that all searching may possibly access. One way our implementation benefits from spatial locality is that we store pointers to different next level nodes in a heap allocated array, we hence have locality in heap. In addition, we need to have access to every different previous level node, which means the higher the height of the skip list we have, we need to access more sparse data, it is also another locality issue.

The below figure is a typical skip list. Each list is a sublist of the list at levels below it. Upper level list serves as the index for the lower level list so that the average searching time could be O(logn).



Our interface and operations with inputs and outputs are listed below. The type of input is integer for testing purposes. It could be any type of data structure.

```
class Skiplist {
    public:
        /**
        * Insert the element into the skip list.
        * @param e element to be inserted
        * @return true if successfully added, false if already exists.
        */
        virtual bool insert(int e) = 0;

        /**
        remove the element from the skip list.
        * @param e element to be removed
        * @return true if successfully removed, false if element does not exist.
        */
        virtual bool remove(int e) = 0;

        /**
        * check if the element is inside the skip list.
        * @param e element to be checked
        * @return true if inside skip list and false otherwise
        */
```

```
    virtual bool contain(int e) = 0;
};
```

Approach

Coarse-grained Lock-based Skip List Implementation

For the coarse-grained lock-based skip list, we used a single mutex lock to protect the whole skip list. For all the operations (e.g. insert, remove and contain), they will firstly lock the mutex before searching skip list and unlock before they return. This means that all of the operations will run sequentially without any parallelism. This is our base case.

Fine-grained Lock-based Skip List Implementation

The fine-grained lock-based skip list is designed in a very small granularity and there is also no read blocking in such small granularity which means we have no read lock. The approach to overcome read locks will be explained later. In addition, our fine-grained lock-based skip list implementation, unlike the lock-free version, keeps all link connections fully linked all the time, which will always speed up the lockups.

Challenges:

1. Locks Management
   Since the fine-grained skip list is designed in a very small granularity: we need to lock previous nodes in all layer connections for each modification operation. There are extremely large amounts of locks to manage at the same time. Our basic approach is to acquire locks from bottom level to the top level and release them from the top level to the bottom level. But even if we manage locks in such a way, there are still many cases where we do not need to acquire the lock for some specific levels and do not release locks for these specific levels. That is when a level connects to the same node with a different level, if the different level has acquired the lock, then the level will not acquire the lock, otherwise, it would be a deadlock. And these states is keeped to help the releasing locks stage not to release node's lock twice.

2. Non-blocking Read
   Our design for the read is non-blocking and each read sees itself as in a single-thread view, and thus we do not implement any locks for reading operations. Since there is no lock for reading, we must have some other mechanism to overcome this problem. Here we introduce two more variables called fullylinked and marked whose type are std::atomic_bool and keeps pointers as atomic in the implementation to keep the state of a node in the skip list. Thus for each lockup, even if there are other concurrent write operations the operation goes as normal single thread operation – since we keep the skip list fully linked all the time, there will not be any unexpected problems. Whenever it

finds a node, it just checks whether it is fully linked: fully added to the skip list; and marked: marked as delete but not yet finally deleted.

Optimization Iterations:
There are several iterations for the fine-grained lock-based skip list. The reason is that the complex lock management introduces extra difficulty in concurrency scenarios.

1. Our first design that is write locks are acquired in both predecessors and successors. But as we progressed the design, we found out that the successors do not need to acquire locks to ensure the concurrency correctness. Instead, it introduces extra difficulties in locks management. Thus, we move forward to a better approach to reduce program complexity and better performance with no lock acquiring in successors.

2. Second iteration is our final implementation of the fine-grained lock-based skip list. In this iteration, we implemented node-based lock acquiring and non-block read and several other features like std::atomic_bool fullylinked variable.

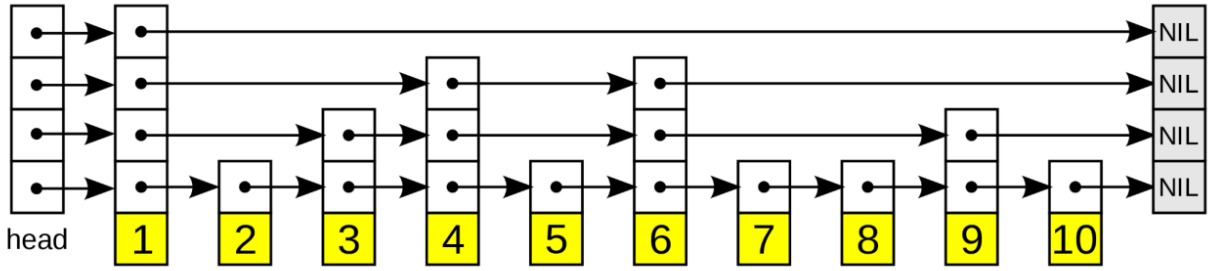
Final Design Key Concepts:

1. Fully Linked List
   Since we designed a non-block fine-grained lock-based skip list, we need to ensure the correctness of the skip list in concurrency scenarios.Therefore, we must ensure that the skip list is fully linked so that each lookup can act as like in a single thread view. To keep the skip list fully linked, we included the fullylinked variable whose type is std::atomic bool. If the not is just added but fully added to the skip list, its fullylinked variable is false, in which case, loopup operation will not return the value if it is not yet fully linked.

2. Logical Deletion
   Deleting node would not delete directly, instead it would first mark it as deleted node in the variable marked whose type is std::atomic_bool. This is because the delete node involves multiple CPU instructions, if we do not first mark it as deleted, other lookup operations may not be aware this node is deleting, it may return partially delete value. The other case is for insert operations, if we do not mark the node as deleted, an insert operation may acquire lock ahead and make links from this deleted node to itself. This is undesired behavior. Thus we mark the node deleted first to inform other threads that this node is deleting so that other threads can do responses accordingly.

3. Locks and Assignment Based Pointers
   Due to the fact that we designed a non-blocking read operation, we need to ensure that it will not access any half modified pointers that may lead to unpredictable behavior. Thus, we created atomic pointers for each node to point to the next different level of nodes. Assume there are ongoing write operations, it will change pointers value atomically. In addition, since each write would acquire locks for according nodes so that

there will not be any other write on the same node could happen simultaneously. Thus, lookup operations could operate in the correct way.



Fully linked fine-grained lock-based skip list looks the same as generally skip list.

Operations Explanation :

Insert

The insert operation will first call an internal function find_node() to determine whether the target element already existed in the skip list, and then find its predecessors and successors in all levels. After finding all these nodes, the operation will lock all predecessors from low level to high level order. If all locks are acquired, then it will create the new node with all pointers pointing to the potential successors and update all pointers in the predecessors to this node. Now this new node now physically exists in the skip list and the fully linked variable is updated to true. If it fails to acquire all locks, the current operation must be aborted and start over again because there may be modification on nodes that it is trying to acquire locks upon.

Remove

The remove operation will first call an internal function find_node() to determine whether an unmarked node with a target element is present in the level list. If the corresponding node is found, it will acquire the lock of these nodes and set the marked variable. Then it will try to acquire all predecessors' locks, and check eligibility of predecessors (e.g. whether predecessors are under deletion or creation). If the eligibility check is failed, it will roll back to state before the delete operation. After all locks have been acquired, it will then first relink all its predecessors to its successors. And after this is done, it will release all acquired locks and the deletion operation is done.

Contain

The contain operation is basically the same as other versions of the skip list, which search from top level to the bottom level to find the target element. There are only two major differences. The first one is that it will ignore the node without setting the variable fullylinked to true. The second is that it will check whether the variable marked is false.

Lock-free Skip List Implementation

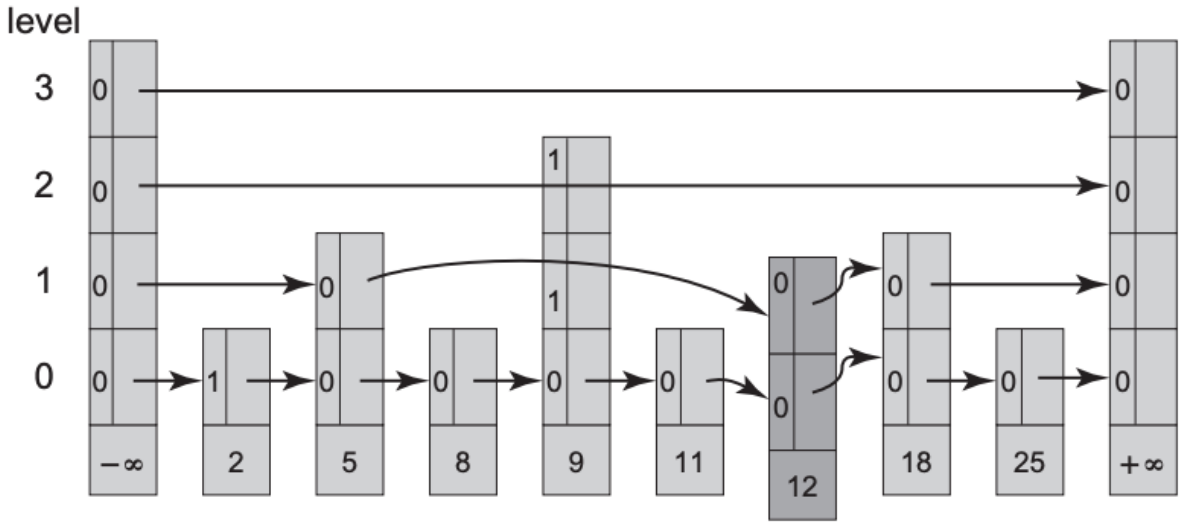
The main challenge for the lock-free skip list is that we can not use lock to ensure that the modification on the multiple levels of the node happens atomically. While other threads are searching across the skip list, some of the levels have already been modified (e.g. inserted or deleted) while some of the other levels still remain original. Thus, we can not maintain the skip list's property that each list is a sublist of the list at levels below it. In this approach, we will treat each level of the skip list as a separate lock-free linked list but they together produce the same result as the skip list.

There are several key concepts listed below.

1. Bottom Level List
   Since we can not maintain the skip list property that each list is a sublist of the list at levels below it, the higher level list will likely contain elements that haven't been fully added to the skip list or elements that have been partially deleted from the skip list. Thus, the higher level list only serves as the index instead of the real list. Only the elements in the bottom level list matters. If an element exists in the bottom level list and is not marked, which means logically deleted, then this element is stored in the skip list.

2. Logical Deletion
   When we want to physically delete a node from the skip list, we must modify the pointers inside the previous node in all levels. This will involve concurrency issues because the previous node may also be modified (e.g. deleted or additional node inserted) before the modification in all levels finishes. We will lose the current modification if that situation happens. Thus, instead of changing the pointers in the previous node, we mark the pointers inside the current node to indicate logical deletion. Other threads can safely treat this node as physically deleted when they encounter references with marks at each layer. We can later physically change the pointers in the previous node.

3. Markable Reference
   We need to put a mark on the pointers to indicate logical deletion. This mark must also be able to use CAS to set with the pointer itself atomically. However, this feature is not natively supported by C++ so we have to implement it ourselves. We designed an additional structure to store the pointer and mark so that we can modify the pointer to this structure using CAS in one operation. However, this increases the number of memory accesses to get the next node from one to two. This brings additional overhead to search through the skip list and degrades our lock-free skip list performance.



Example of lock-free skip list during insert(12) and remove(9). Both element 5 and element 9 points to element 12 on level 1. While element 9 still exists in the skip list, it is already partially logically deleted for some levels. It is possible that pointers may skip some nodes on the non-bottom level. The order sequence on each level is still maintained.

Below is the further explanation for each operation.

1. Insert
   The insert operation will first call an internal function find_node() to determine whether the target element already existed in the skip list, and then find its predecessors and successors in all levels. Then it will create the new node with all pointers pointing to the potential successors. It will first try to add the new node using CAS to the bottom level. If successful, this new node now physically exists in the skip list. If it fails, the current operation must be aborted and start over again because either the node is already inserted or the neighbor in the bottom level is modified. At last, it will modify the pointers in all the previous nodes to point to the current node. The predecessors in the upper level in this step are possibly outdated. If any modifications happen on the current node or direct neighbors, the insert operation has to call find_node() to find the update-to-date predecessors and successors to retry.

2. Remove
   The remove operation will first call an internal function find_node() to determine whether an unmarked node with a target element is present in the bottom level list. If the current operation is able to find the node, it will start to mark from the top level until one level above the bottom level. At this point, the node is only deleted from the upper level but still exists in the skip list. After all upper level references are marked, we will mark the bottom level reference to indicate that this node has been logically deleted from the skip list. If any steps fail, we will call the find_node() function again to find the up-to-date predecessors and successors to continue the logical deletion. Multiple threads can logically delete the same node at the same time without concurrency problems. When

logical deletion of the target node finishes, we can physically modify the pointers from the previous node until successful.

3. Contain
   The contain operation is basically the same as other versions of the skip list, which search from top level to the bottom level to find the target element. There are only two major differences. The first one is that it will ignore the node with a reference mark. The second point is that it must reach the bottom level to check if the element is inside the bottom level. It can not stop and return if it finds the element on the upper level.

Experiments

Experimental Setup

We ran a series of tests to evaluate the performance of our three implementations under different conditions. The main metric we used for performance comparison is the overall time to finish a series of operations distributed to threads in milliseconds. The OpenMP framework is used to run operations with multiple threads concurrently. Each test describes its detailed settings in the description section later. We ran 100,000 operations for each test by default because we believe this is a reasonable number to allow multiple threads to speed up the executions. We set skip list max height to be 15 by default because log(100,000) is around 16. We believe any max height values around 16 should be reasonable to allow the skip list to be high enough to search a specific node in log(n) time on average. We run our experiments on the GHC machine with 8 cores with no hyperthreading.

Before running each test, we called an internal warmup function to make sure enough memory is available on the heap. This warmup function will sequentially insert lots of elements, which number is much more than any tests we run, and then free the space. This can prevent spending significant time on extending the heap.

Due to the random height generated for each node, the test performance on all versions of the skip list varies a lot. This feature is the skip list's property and can not be avoided. Thus, we ran each test at least three times to take the average. We also removed the outrange results (more than 30% away from average) and then ran additional tests to make sure our results are meaningful.

Test Type

In order to evaluate the efficiency and execution behaviour of our implementation, we ran the following tests.

1. Basic Operations

We run each operation (insert, contain, remove) individually with both sequential and random inputs.
2. Varying Workload Size
   We run the combination of the three operations together with various workload sizes.
3. Operations with Different Distribution
   We run the combination of the three operations with different disbritutions between them.
4. Effect of Max Height
   We run the combination of the three operations with different max height limitation.
5. Workload with High Collision Rate
   We run the combination of the three operations in a high collision input within a small range.

For test cases with the combination of the three operations, we run the three operations in the mixed order by using round-robin fashion. We first run a portion of insert, then contain, and remove operations on the input. Then continue to run the next iteration until the input array is exhausted. The portion of each operation in each iteration is determined by the number of threads and is always smaller than 1% to achieve a good operation mixture.

Test #1 - Basic Operations

Description

For this test we observe the performance of individual operations (insert, contain and remove). We run our test with 100,000 operations each and a max height of 15. For each version of skip list, we run insert, contain and remove in sequential order and record the time individually. This test can help us determine the behavior pattern and the proper input array order.

Observations & Analysis

For the insertion operation, all versions have a non-decreasing total execution time for the sequential input elements as the threads number increases. The possible reason is that all threads will attempt to add elements at the end of their element ranges, bringing a consistently increased searching time. This is caused by the data shape of the sequential elements. The fine-grained and lock-free version have similar performance compared to the coarse-grained version, which means that their performance is similar to a non-parallel version. Under this circumstance, adding additional threads seems to increase the total execution time. It is more reasonable to use a non-parallel version to insert sequential elements to achieve better performance. In addition, since every node depends directly on the previous node, it will always wait for previous node to finish its insertion, this is another reason for slow sequential insertion,

Among the six tests we listed, lock-free versions mostly have a much lower performance when there's only one thread and then have similar good performance with fine-grained versions when the thread number increases. The reason is that the lock-free version has higher

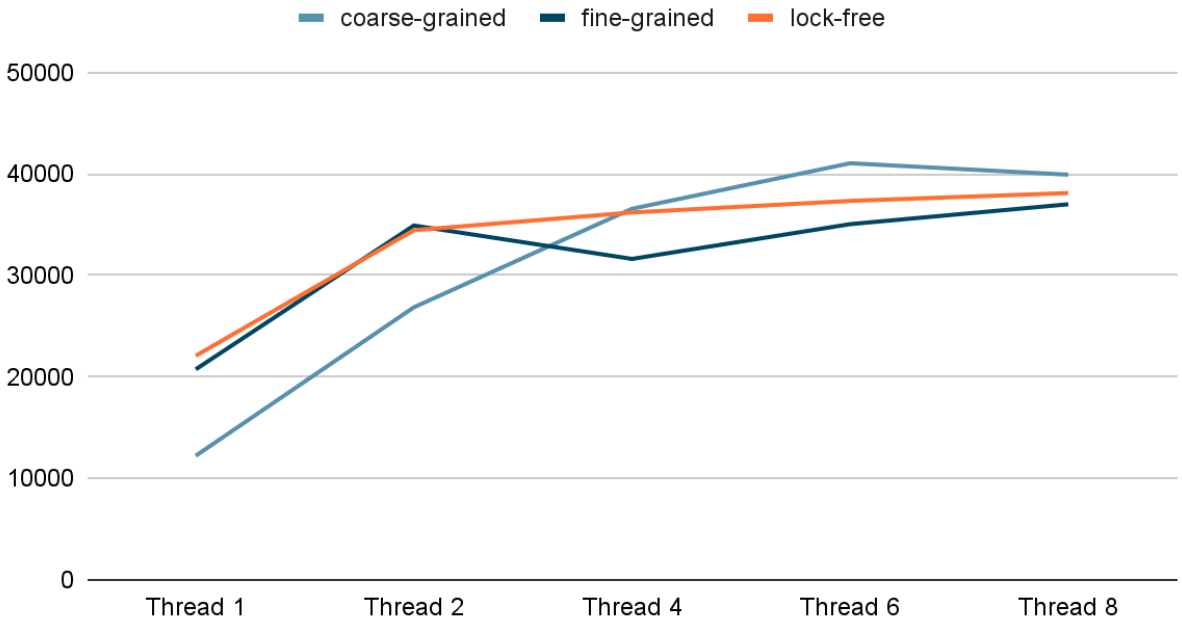
overhead to search through the list. The detailed reason will be explained in the deeper analysis section. As the number of threads increases, this drawback is compensated by the speedup of multi-threading. Lock-free versions have lower overhead in adding and removing the elements with low collision because it does not need to lock. The high overhead in the lock-free version will not be mentioned in the future tests because it all behaves similarly.

We observed that when the thread number increases from one to two, the performance will sometimes drop, or have no ideal speedup for two threads. The likely reason is caused by the overhead for multi-thread scheduling. When there's only one thread to execute the task, OpenMP may not need to spawn additional threads so the overhead is low. Thus, there may be a sudden increase in the time when there are more than one threads. The additional overhead is caused by spawning new threads and waiting for threads to finish.

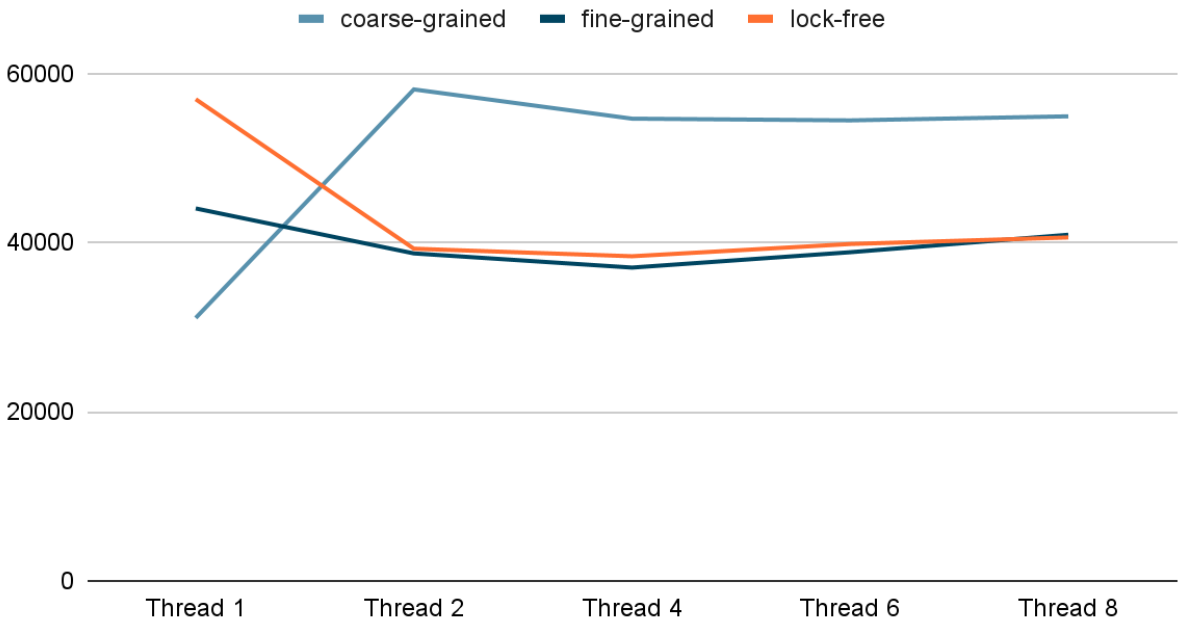
Within the tests except sequential insertion, fine-grained and lock-free versions have an increased speedup but decreased speedup speed when the thread number increases. At the beginning the speedup is benefited by having multiple threads to run the task. The decrease in the speedup speed is caused by the multithreading scheduling overhead and competition between threads. Another reason is that, as thread number increases, the change to cache locality is decreased. This is because each node needs to acquire data of all previous linked nodes; if there are 15 levels, it needs to access 15 different addresses. Since the number of threads grows, the number of different addresses increases as well. Some addresses may require read exclusive, which would make the same data on other threads invalidated. As the number of threads increases, the chances for these scenarios increase.

Since randomized input elements have much more obvious effects as thread number increases and are more realistic, we decide to use randomized input elements to conduct the remaining tests.

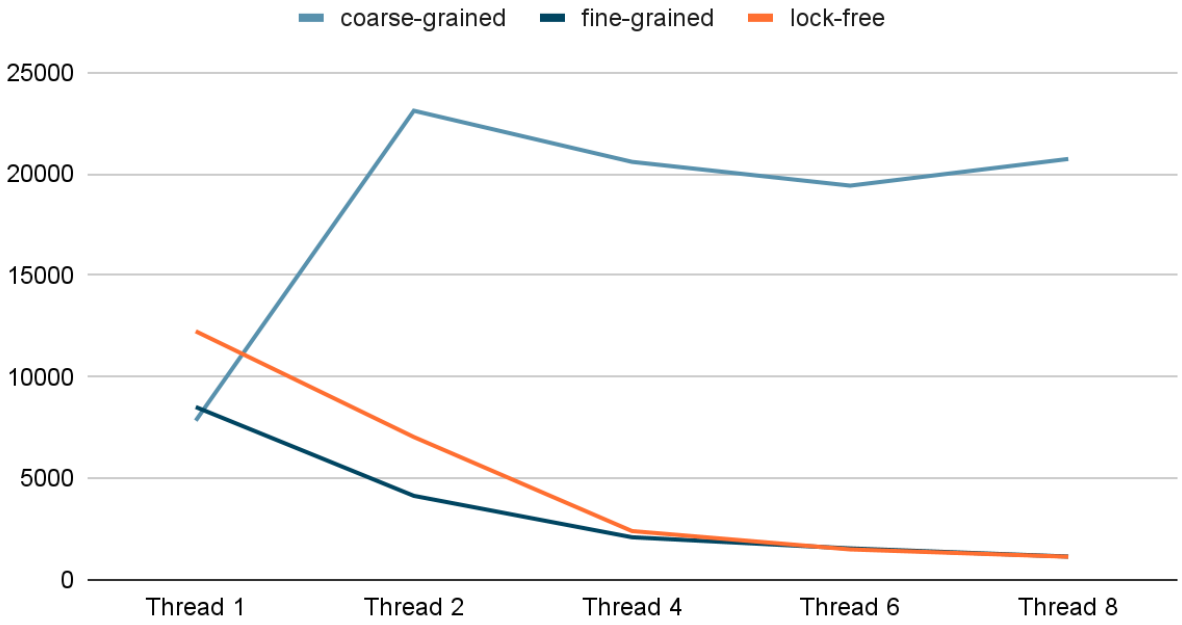
Total Time for Insert (Sequential)



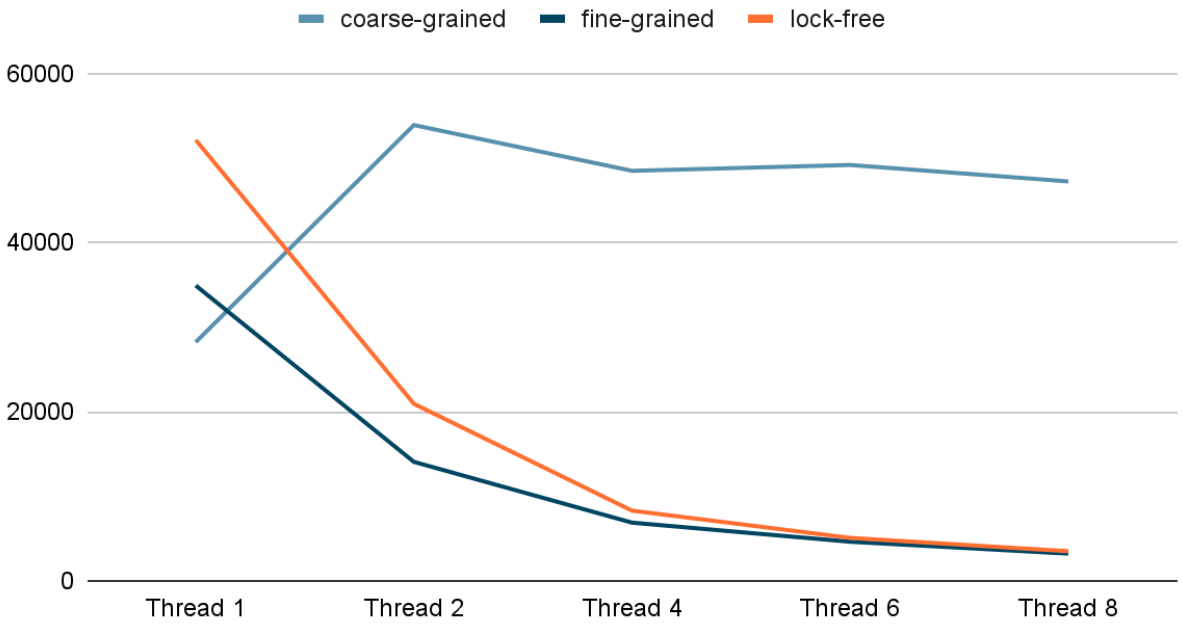
Total Time for Insert (Random)

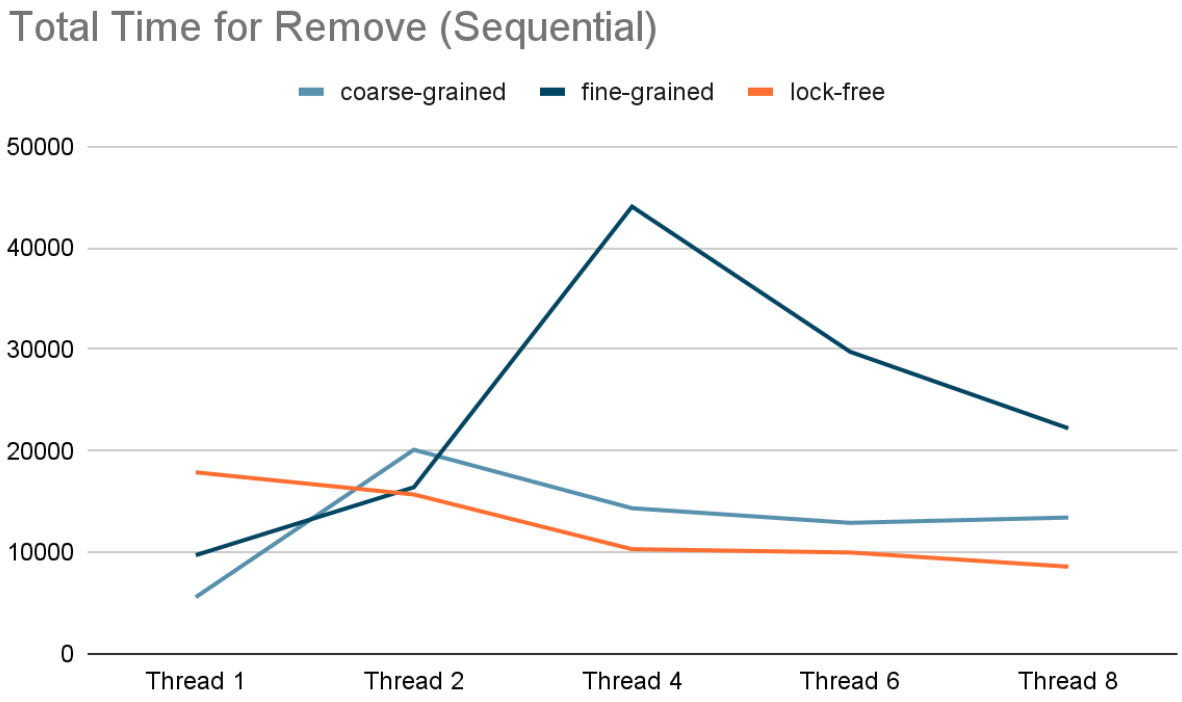


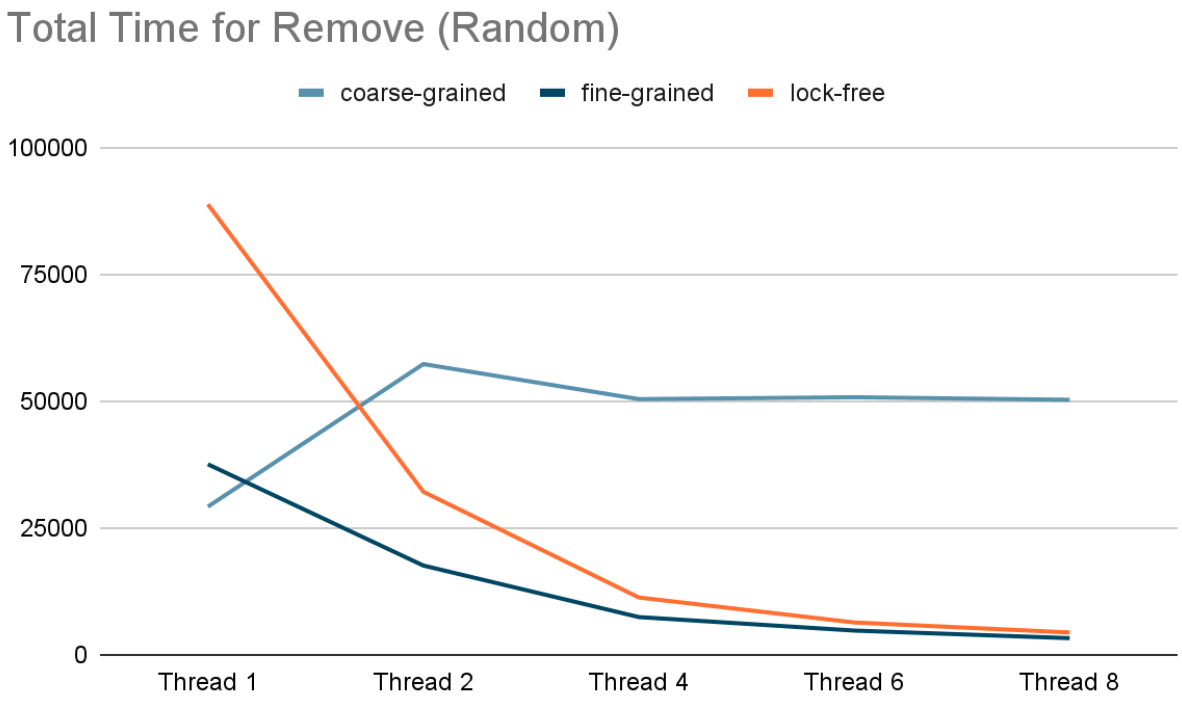
Total Time for Contain (Sequential)



Total Time for Contain (Random)




Total Time for Remove (Sequential)
coarse-grained
fine-grained
lock-free
50000
40000
30000
20000
10000
0
Thread 1
Thread 2
Thread 4
Thread 6
Thread 8


Total Time for Remove (Random)
coarse-grained
fine-grained
lock-free
100000
75000
50000
25000
0
Thread 1
Thread 2
Thread 4
Thread 6
Thread 8

Test #2 - Varying Workload Sizes

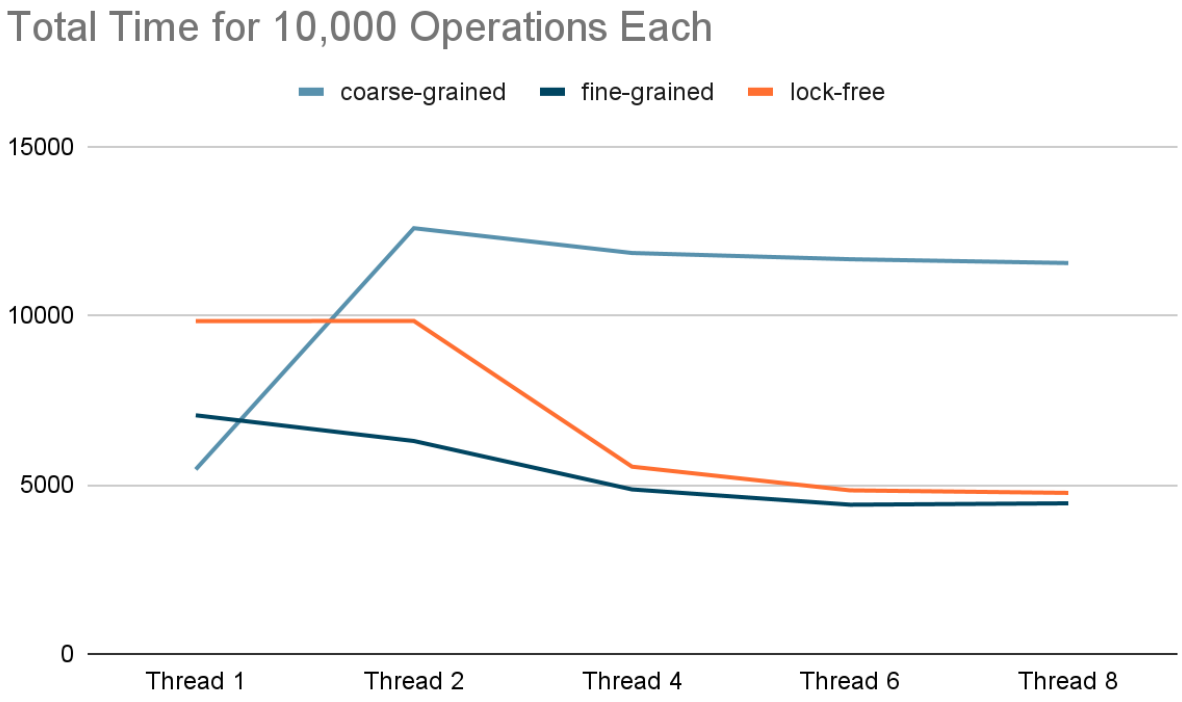
Description

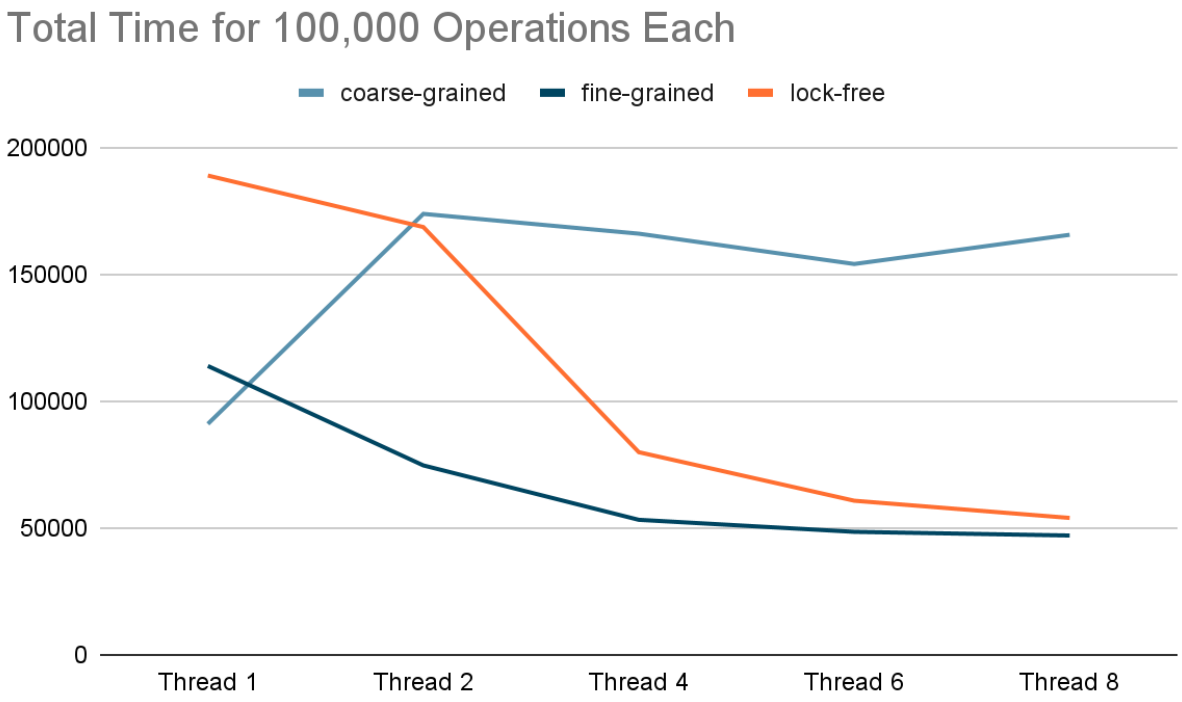
For this test we observe the performance of the combination of the operations under different sizes of workload. We run our test with 10,000, 100,000 and 1,000,000 operations each, a max height of 15 and randomized input elements. For each version of skip list, we run insert, contain and remove a mixed order. We didn't test on larger or smaller sizes because size too small may not benefit from multi-threading and size too large requires increase in the max height to make the search time O(logn).

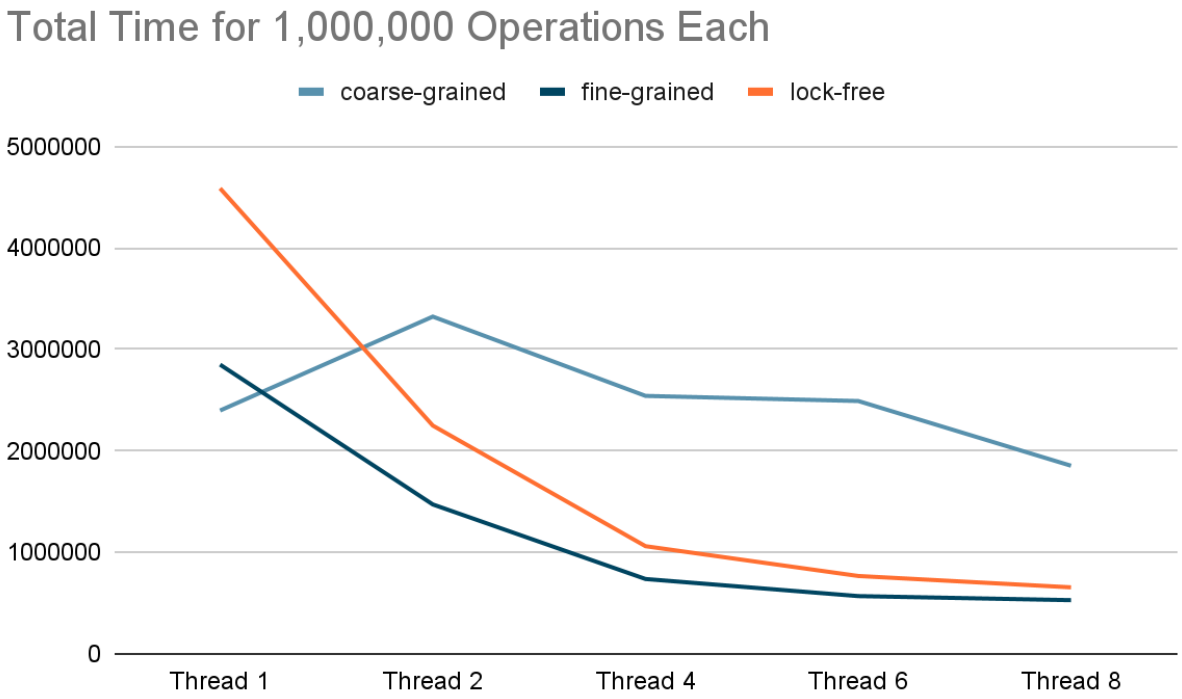
Observations & Analysis

When the number of operations increases, both fine-grained and lock-free versions have smoother performance lines as the thread number increases. Both the versions have better performance but reduced speedup increases speed as thread number increases. Both versions can still have increasing performance when threads number increase over four threads but become less obvious. We propose four possible reasons for this. The first is that the competition between threads increases as the thread number increases.This is unlikely to be true given the large number of randomized elements. The second reason is due to the multithreading overhead. The more threads to have, the more time spent on spawning and blocking for results. This is more likely to be the reason for fewer operations because the execution time is relatively small. The third reason is that the max height of the node is limited. Thus, this is the fastest speed threads could achieve. If we allow higher node height or more well distributed node height shape, we could achieve better speedup in high threads. The fourth possible reason is due to the cache invalidity. Though linked lists do not benefit too much from cache due to poor spatial locality, they could still benefit from temporary locality, especially on the node with high level. All threads need to search from high level to low level. If any modification happens on the node with high level, it will invalidate that node's cache in all threads. As the number of threads increases, this likelihood is also increased. Further experiments are needed to determine which suggestion is the dominant reason.

It is also obvious for lock-free and coarse-grained versions that there's high overhead when the thread increases from one to two. We believe that this is mainly due to the multithreading scheduling overhead. When the number of operations increases, this overhead is less obvious because threads spend a higher percentage of time on executing the task. Multithreading overhead is more related to the number of threads than operation size. Thus, this test proves that running with more operations can benefit from multiple threads more because the portion of overhead for multithreading is reduced.


Total Time for 10,000 Operations Each
coarse-grained
fine-grained
lock-free
15000
10000
5000
0
Thread 1
Thread 2
Thread 4
Thread 6
Thread 8


Total Time for 100,000 Operations Each
coarse-grained
fine-grained
lock-free
200000
150000
100000
50000
0
Thread 1
Thread 2
Thread 4
Thread 6
Thread 8



Test #3 - Operations with Different Distribution

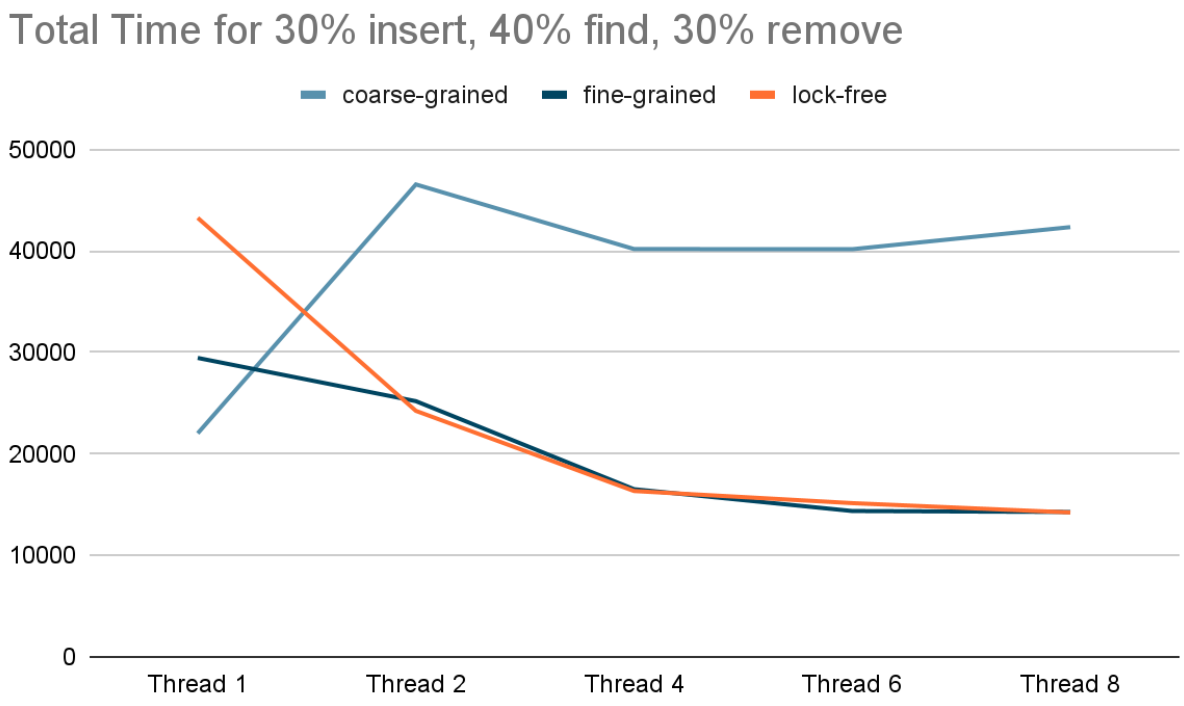
Description

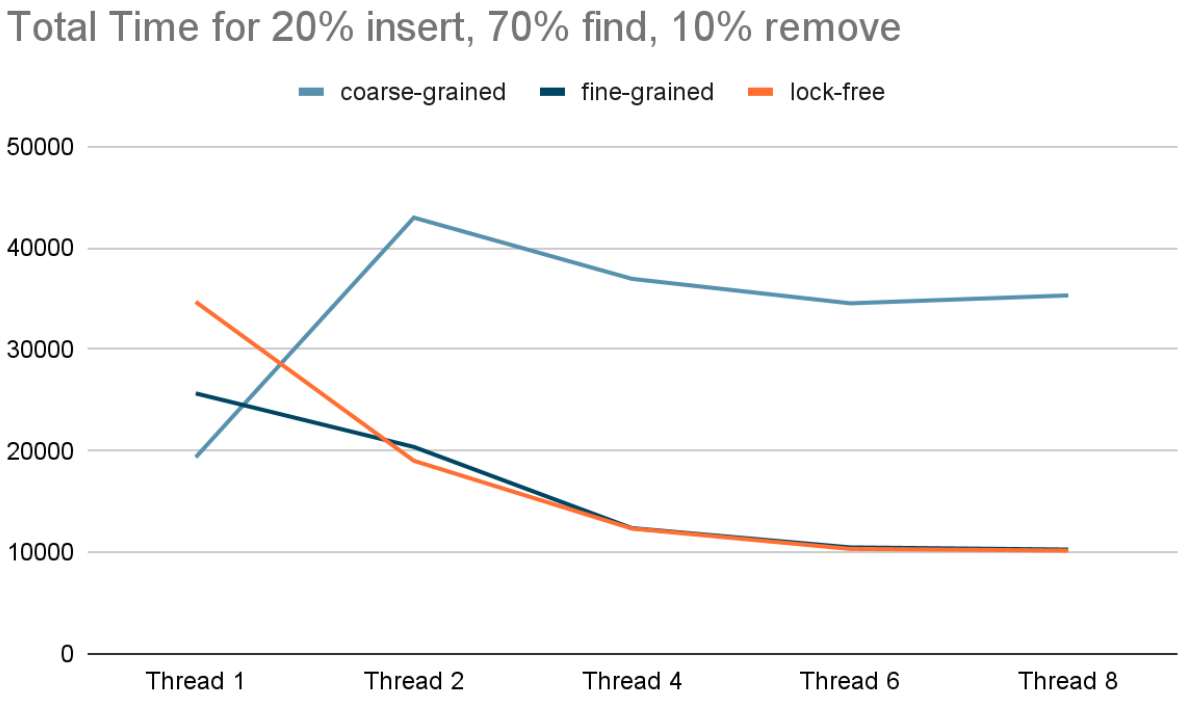
For this test we observe the performance of the combination of the operations under different operation distributions. We run our test with 100,000 operations, a max height of 15 and randomized input elements. For each version of skip list, we run insert, contain and remove a mixed order. This test is used to stimulate usage in real circumstances, including equal distribution (30% insert, 40% find, 30% remove), heavy searching (20% insert, 70% find, 10% remove) and heavy insertion (80% insert, 20% find, 0% remove)

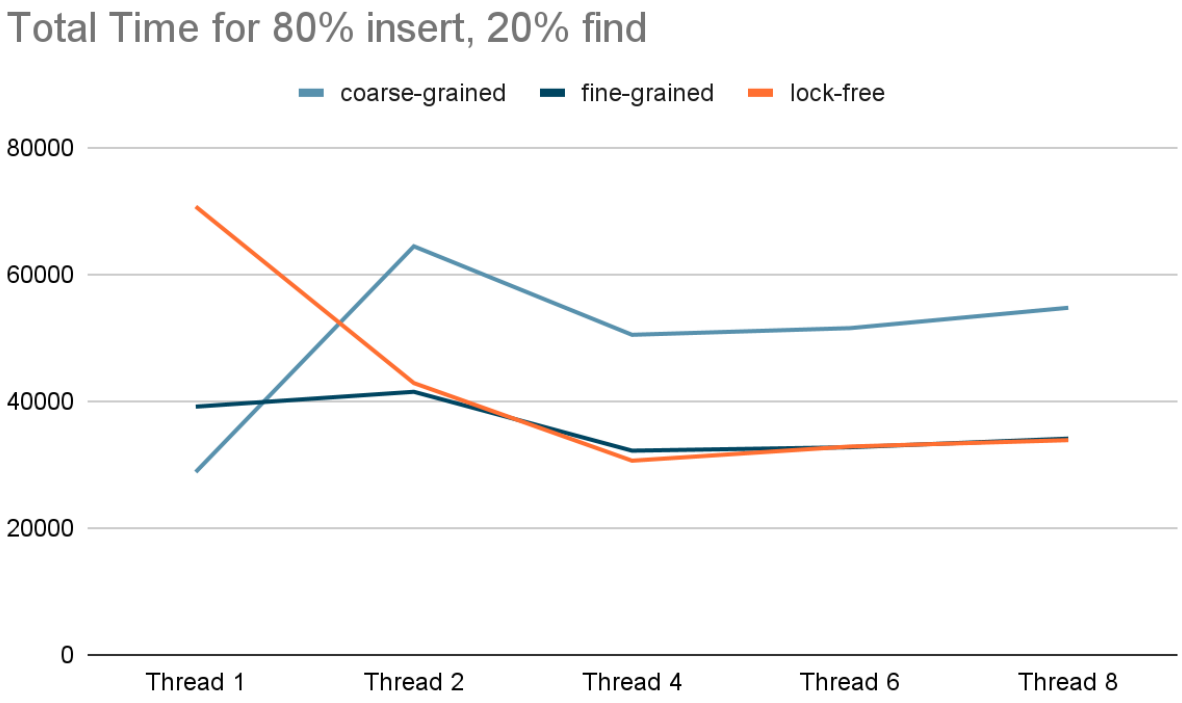
Observations & Analysis

For the equal distribution and heavy searching tests, both fine-grain and lock-free have similar outcomes compared to the previous tests. They performed well under multithreading and improvement decrease and become less obvious as thread number increases. In the heavy insertion test, however, both fine-grain and lock-free do not benefit too much from multithreading. Though they still have better performance than the coarse-grained version, they are less than two times speedup compared to the coarse-grained version. The possible reason is similar to the reasons mentioned above. It could be the competition between threads because insert needs to modify the neighboring nodes while remove only uses logical deletion. Cache Invalidity could be another major reason. Inserting new nodes with high levels is likely to invalidate the cache line in other threads, which lower the performance. The more threads to have, the more frequent cache invalidation happens on the high level nodes.

In conclusion, both fine-grained and lock-free versions are much better than coarse-grained versions for multithreading conditions. It is more obvious under normal distribution and heavy finding distribution and less obvious under heavy insertion distribution.




Total Time for 20% insert, 70% find, 10% remove
coarse-grained
fine-grained
lock-free
50000
40000
30000
20000
10000
0
Thread 1
Thread 2
Thread 4
Thread 6
Thread 8


Total Time for 80% insert, 20% find
coarse-grained
fine-grained
lock-free
80000
60000
40000
20000
0
Thread 1
Thread 2
Thread 4
Thread 6
Thread 8

Test #4 - Effect of Max Height

Description

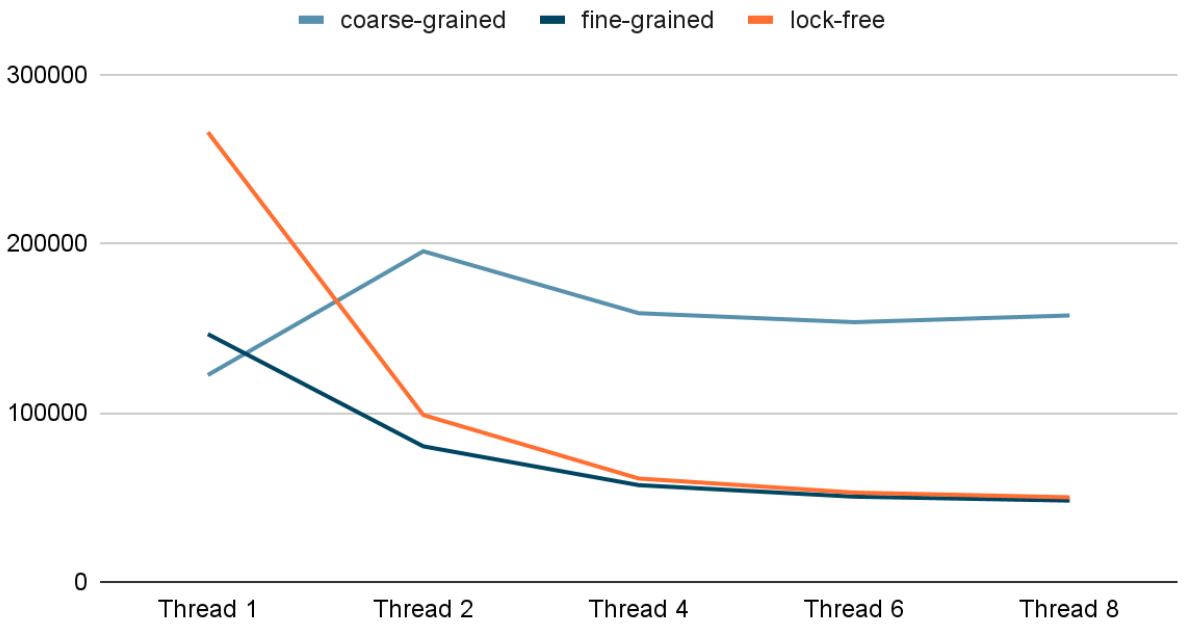
For this test we observe the performance of the combination of the operations under different max height. We run our test with 100,000 operations, a max height of 10, 20, 100 and randomized input elements. For each version of skip list, we run insert, contain and remove a mixed order with equal distribution between three operations.

Observations & Analysis

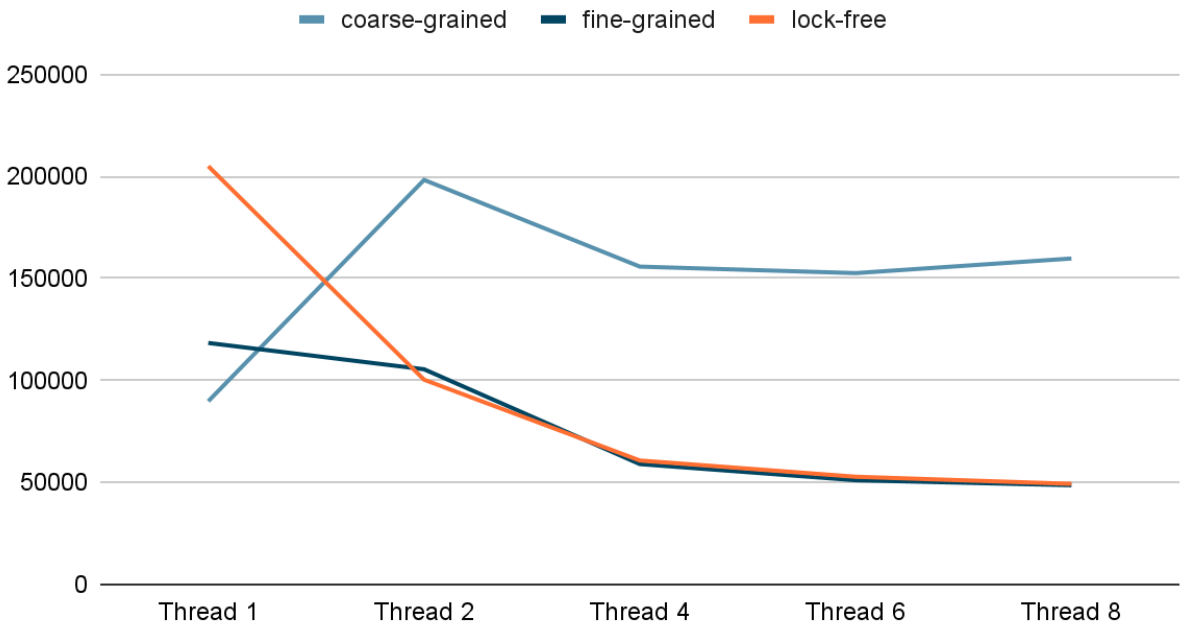
The performance line all the three figures have similar shapes. The performance with higher and lower than suitable height seems to have little decrease in performance but not obvious enough. We tried to increase the height to 100, which is a super high height compared to the amount of operations. The decrease in the performance has increased a little but still not obvious enough. We also tried to increase the height to unlimited or even lower but the running time becomes too long to collect the data. The possible reason is due to the random generation of the node height. When the maximum height allowed increases, it is more likely that nodes in the skip list have inappropriate height shape, which leads to worse performance in searching. When the max height is very low, nodes can not generate a good height shape to allow O(logn) searching time because it is not high enough. When the max height allowed does not vary from the suitable height too much, the possibility for having nodes height in good shape is high, which brings a relatively similar searching time.

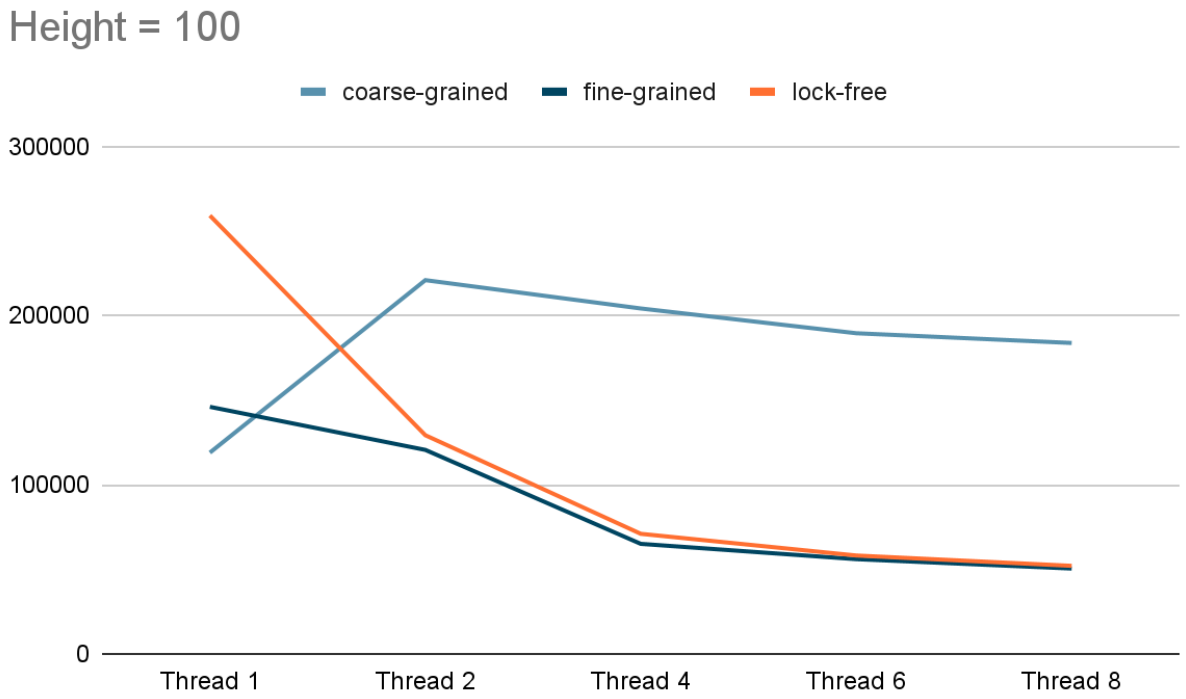
Thus, When the height is close to the suitable height, it does not affect the performance too much. When the height is far away from the suitable height (e.g. super low or unlimited), it has a higher possibility to decrease the performance a lot. The affection of max height on the performance depends more on the probability to generate node in good height shape instead of the height itself.

Height = 10



Height = 20





Test #5 - Workload with High Collision Rate

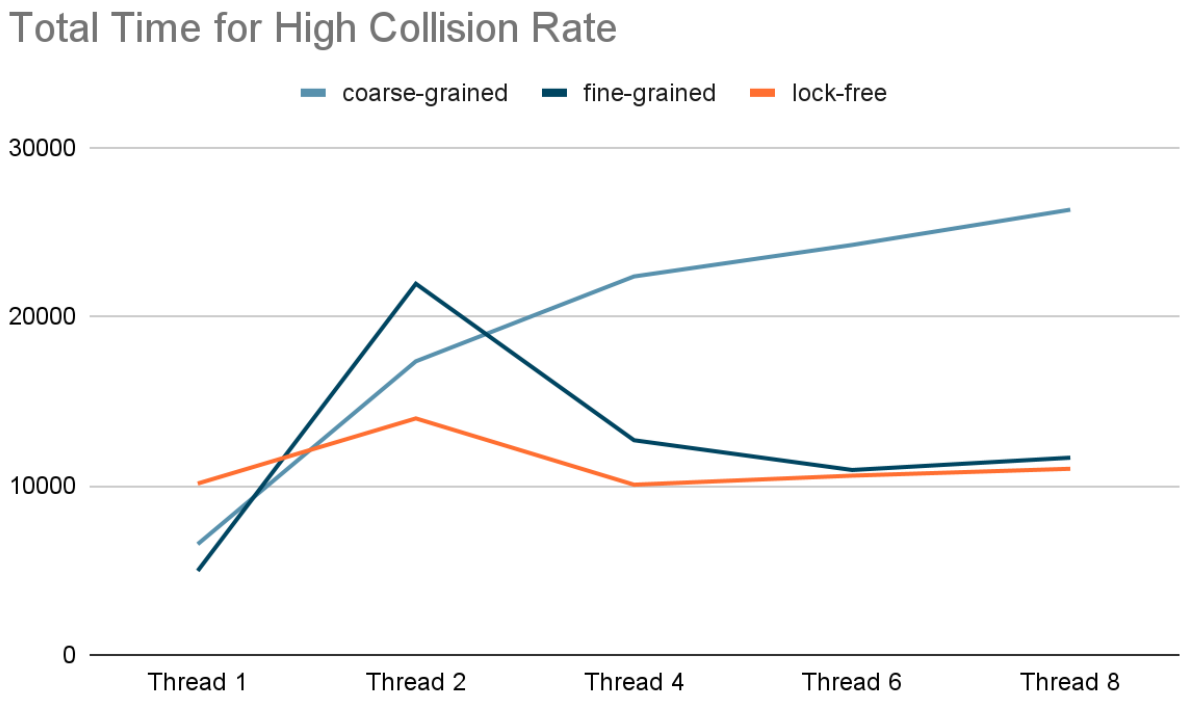
Description

For this test we observe the performance of the combination of the operations under high collision rate. We run our test with 100,000 operations and a max height of 15. For each version of skip list, we run insert, contain and remove a mixed order with equal distribution between three operations. The elements in the input array are randomized and fall within a small range. Each thread may have multiple same or neighboring elements to perform the operations. This will generate a lot of collisions between threads.

Observations & Analysis

When the collision rate becomes high, all the three versions do not benefit from the multithreading. The main reason could be the competition between the neighbouring nodes since all the elements fall within a small range. It is also possible that the cache is updated and invalidated frequently. Though linked lists do not benefit too much from cache due to poor spatial locality, they could still benefit from temporary locality, especially on the node with high level. Since the range of elements is not too large, some of the nodes could be stored in the cache line and shared across the threads. However, since all threads are updating within the small range, most of the nodes will be invalided from cache line before reuse again. This effect is likely to happen more frequently on the nodes with high levels. All the threads will try to search from high level to low level so any modifications on the high level nodes in such a small

range will possibly invalidate all caches in other threads.



Deeper Analysis

Profiling Fine-grained skip list

We use perf to profile the time spent in each function. The input test is 100,000 operations, a max height of 15, randomized input elements and 4 threads running. We can find from the profiling results that FineGrainedSkipList::insert takes most of the time within the execution, which is around 18.09% of total execution time. FineGrainedSkipList::remove takes 15.65% of total execution time and FineGrainedSkipList::contains takes 15.86%. Also, this percentage looks higher than following lock free analysis because the coarse grained skip list test is done with lock free and here we only use coarse grain skip list for warm up purposes.

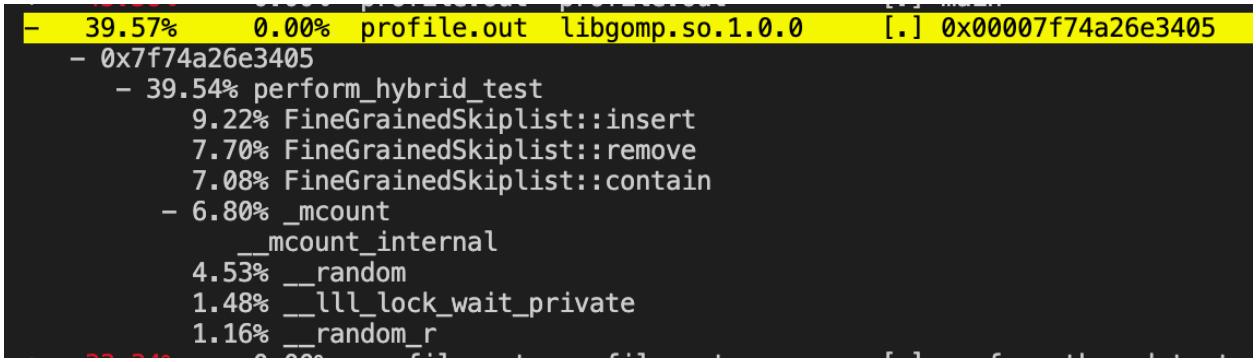
From the profiling data we can see that FineGrainedSkipList::insert takes most of the time. It is reasonable because insert needs to create a new node and modify the predecessors' pointers to point to the current node. All these operations require locks and atomic assignment to atomic variables and retry the whole functions when acquiring lock fails. FineGrainedSkipList::remove takes less time than insert because remove does not need to create a new node and the amount of links modified is less than insert. On the other hand, FineGrainedSkipList::contains is only a search through the list to find a match so it is much quicker than removing. We can see that the remove function takes almost the same time as searching through the list from the profile data.

From the profiling data, the thread lock and unlock does not take much time. The lock takes 2.27% of total execution time and unlock takes 2.02 of execution time. The lock takes more time than the unlock is probably because there are threads waiting overhead for the lock. Although this waiting for lock is not very long, we can still think about the optimization to reduce this waiting lock overhead part. Overall, the lock does not consume too much time in profiling, which means that the major bottleneck resides on our execution logic other than lock contentions. This is probably because non-blocking reads introduce quite complicated logic.

We also check the profiling data for OpenMP. We found that three is 4.53% time spent on generating random node height. From our understanding, such time is not a small amount of time. Maybe we need to find a better algorithm to determine the level for each created node.

Below are profiling data screenshots:

```
Samples: 5K of event 'cycles:uppp', Event count (approx.): 4066559014
  Children      Self  Command      Shared Object         Symbol
+   70.39%     0.17%  profile.out  profile.out           [.] perform_hybrid_test
+   47.43%     0.00%  profile.out  [unknown]             [.] 0xec83485354415541
+   45.38%     0.00%  profile.out  libc-2.17.so          [.] __libc_start_main
+   45.38%     0.00%  profile.out  profile.out           [.] main
+   39.57%     0.00%  profile.out  libgomp.so.1.0.0      [.] 0x00007f74a26e3405
+   33.34%     0.00%  profile.out  profile.out           [.] perform_thread_test
+   32.00%     1.07%  profile.out  profile.out           [.] test_hybrid
+   18.09%    18.01%  profile.out  profile.out           [.] FineGrainedSkiplist::insert
+   15.86%    15.86%  profile.out  profile.out           [.] FineGrainedSkiplist::contain
+   15.65%    15.65%  profile.out  profile.out           [.] FineGrainedSkiplist::remove
+   11.97%     0.25%  profile.out  profile.out           [.] warmup
+   10.31%     0.63%  profile.out  libc-2.17.so          [.] _mcount
+    9.77%     9.74%  profile.out  libc-2.17.so          [.] __mcount_internal
+    7.78%     7.78%  profile.out  libgomp.so.1.0.0      [.] 0x0000000000018b1f
+    7.78%     0.00%  profile.out  libgomp.so.1.0.0      [.] 0x00007f74a26e5b21
+    7.20%     7.06%  profile.out  libc-2.17.so          [.] __random
+    3.15%     3.12%  profile.out  profile.out           [.] CoarseGrainedSkiplist::remove
+    2.88%     2.83%  profile.out  profile.out           [.] CoarseGrainedSkiplist::insert
+    2.77%     0.00%  profile.out  [unknown]             [.] 0000000000000000
+    2.75%     2.49%  profile.out  libc-2.17.so          [.] _int_malloc
+    2.27%     2.16%  profile.out  libc-2.17.so          [.] __random_r
+    2.27%     2.12%  profile.out  libpthread-2.17.so    [.] pthread_mutex_unlock
+    2.02%     1.94%  profile.out  libpthread-2.17.so    [.] pthread_mutex_lock
+    1.89%     1.89%  profile.out  libc-2.17.so          [.] malloc_consolidate
+    1.80%     1.51%  profile.out  libc-2.17.so          [.] __lll_lock_wait_private
+    1.65%     1.47%  profile.out  libc-2.17.so          [.] malloc
     1.09%     1.09%  profile.out  libc-2.17.so          [.] _int_free
+    0.76%     0.00%  profile.out  [unknown]             [.] 0x0000000000000021
     0.58%     0.30%  profile.out  libc-2.17.so          [.] rand
     0.57%     0.57%  profile.out  [unknown]             [k] 0xffffffff82d96098
```



Profiling Lock-free Skip list

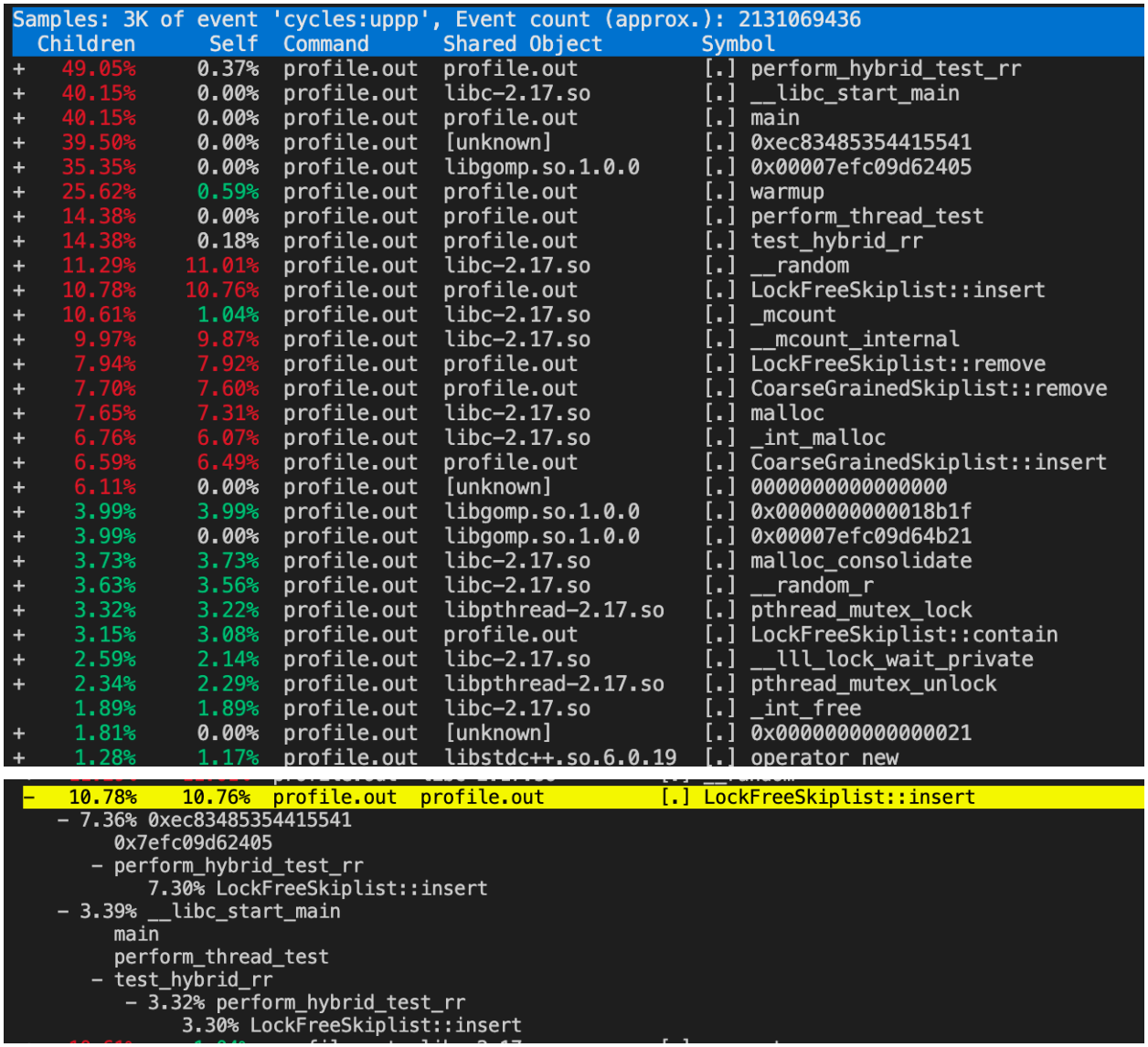
We use perf to profile the time spent in each function. The input test is 100,000 operations, a max height of 15, randomized input elements and 4 threads running. We can find from the profiling results that LockFreeSkipList::insert takes most of the time within the execution, which is around 10.78% of total execution time. LockFreeSkipList::remove takes 7.94% of total execution time and LockFreeSkipList::contains takes 3.15%. We tried to dive deeper to each individual function but perf seems to not provide any further information. We tried gprof as well and received similar outcomes.

From the profiling data we can see that LockFreeSkipList::insert takes most of the time. It is reasonable because insert needs to create a new node and modify the predecessors' pointers to point to the current node. All these operations require CAS and retry the whole functions in some cases. LockFreeSkipList::remove takes much less time than insert because remove only modifies the current node to logical delete. On the other hand, LockFreeSkipList::contains only a search through the list to find a match so it is much quicker than removing. We can see that the logical remove takes almost the same time as searching through the list from the profile data.

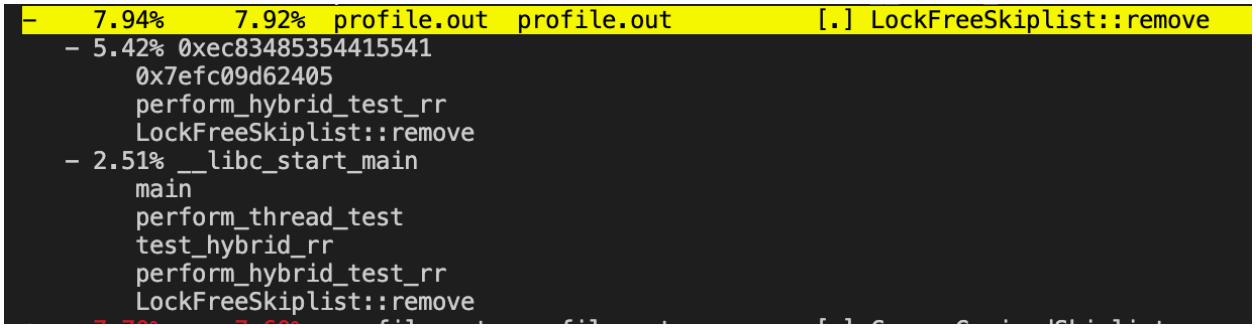
We also check the profiling data for OpenMP. We found that random takes 8% of the total time. Random is used to generate the height of the current node. This could be a possible reason for the slow speed for insert. We may need to find a faster random algorithm in the future to generate the random node height. We are not sure what does _mcount means in the OpenMP section and why it takes 7%, but we can find that lock_wait_private also takes an additional 2% of the total time. The scheduling and synchronization overhead could be another bottleneck of our lock-free implementation.

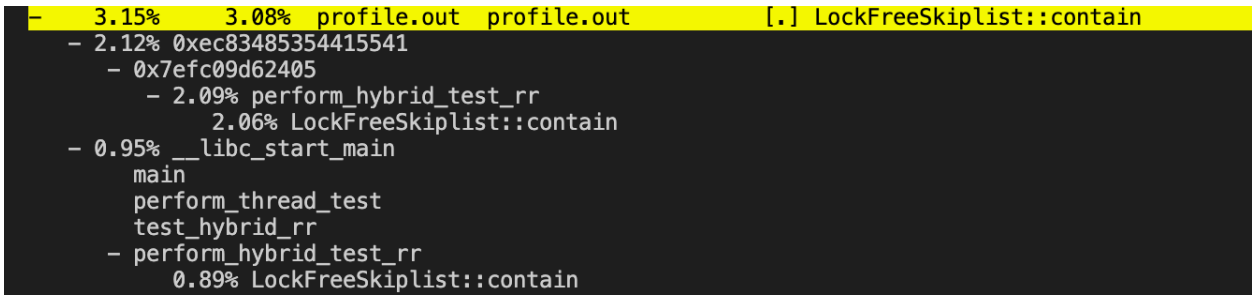
After analyzing the performance from the previous graph, we find the performance for lock-free version is low under one thread condition. It means that there's high additional overhead. According to our implementation details, it is mainly caused by our solution for the markable reference. In other versions of skip list, it takes one memory access to dereference the pointer and get the next node address. However, since markable reference is not natively supported by C++, we have to implement our own data structure to stimulate this feature. This will require one

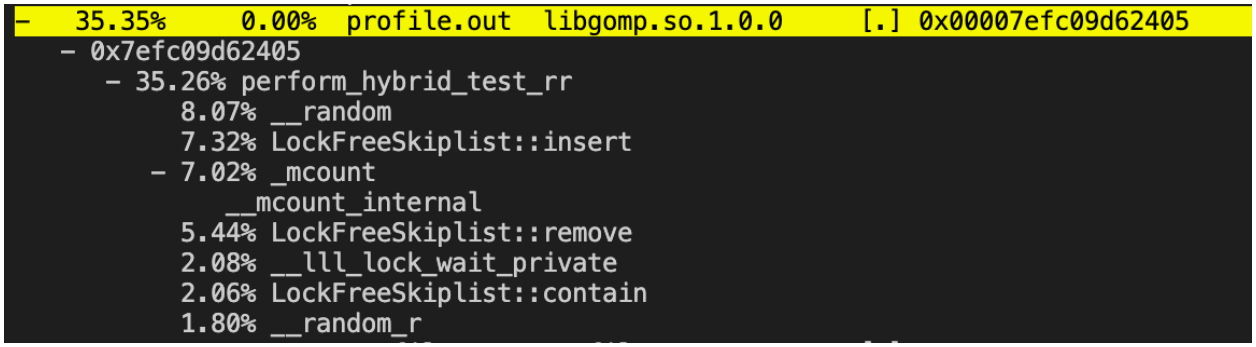
more memory access to get the next node address. This brings high overhead to search through the skip list and is the main reason for the high overhead in low thread numbers. When the thread number is high, a lock-free version can benefit from multi-threading a lot since it does not need the acquire lock. Thus, it has similar additional overhead and much closer performance compared to the fine-grained version when the thread number is high. If we could find other solutions for markable reference, we could lower the overhead in searching through the list and have a better performance in low thread number.



```
Samples: 3K of event 'cycles:uppp', Event count (approx.): 2131069436
  Children      Self  Command      Shared Object        Symbol
+   49.05%     0.37%  profile.out  profile.out          [.] perform_hybrid_test_rr
+   40.15%     0.00%  profile.out  libc-2.17.so         [.] __libc_start_main
+   40.15%     0.00%  profile.out  profile.out          [.] main
+   39.50%     0.00%  profile.out  [unknown]            [.] 0xec83485354415541
+   35.35%     0.00%  profile.out  libgomp.so.1.0.0     [.] 0x00007efc09d62405
+   25.62%     0.59%  profile.out  profile.out          [.] warmup
+   14.38%     0.00%  profile.out  profile.out          [.] perform_thread_test
+   14.38%     0.18%  profile.out  profile.out          [.] test_hybrid_rr
+   11.29%    11.01%  profile.out  libc-2.17.so         [.] __random
+   10.78%    10.76%  profile.out  profile.out          [.] LockFreeSkiplist::insert
+   10.61%     1.04%  profile.out  libc-2.17.so         [.] _mcount
+    9.97%     9.87%  profile.out  libc-2.17.so         [.] __mcount_internal
+    7.94%     7.92%  profile.out  profile.out          [.] LockFreeSkiplist::remove
+    7.70%     7.60%  profile.out  profile.out          [.] CoarseGrainedSkiplist::remove
+    7.65%     7.31%  profile.out  libc-2.17.so         [.] malloc
+    6.76%     6.07%  profile.out  libc-2.17.so         [.] _int_malloc
+    6.59%     6.49%  profile.out  profile.out          [.] CoarseGrainedSkiplist::insert
+    6.11%     0.00%  profile.out  [unknown]            [.] 0000000000000000
+    3.99%     3.99%  profile.out  libgomp.so.1.0.0     [.] 0x0000000000018b1f
+    3.99%     0.00%  profile.out  libgomp.so.1.0.0     [.] 0x00007efc09d64b21
+    3.73%     3.73%  profile.out  libc-2.17.so         [.] malloc_consolidate
+    3.63%     3.56%  profile.out  libc-2.17.so         [.] __random_r
+    3.32%     3.22%  profile.out  libpthread-2.17.so   [.] pthread_mutex_lock
+    3.15%     3.08%  profile.out  profile.out          [.] LockFreeSkiplist::contain
+    2.59%     2.14%  profile.out  libc-2.17.so         [.] __lll_lock_wait_private
+    2.34%     2.29%  profile.out  libpthread-2.17.so   [.] pthread_mutex_unlock
     1.89%     1.89%  profile.out  libc-2.17.so         [.] _int_free
+    1.81%     0.00%  profile.out  [unknown]            [.] 0x0000000000000021
+    1.28%     1.17%  profile.out  libstdc++.so.6.0.19  [.] operator new
-   10.78%    10.76%  profile.out  profile.out          [.] LockFreeSkiplist::insert
   - 7.36% 0xec83485354415541
        0x7efc09d62405
      - perform_hybrid_test_rr
           7.30% LockFreeSkiplist::insert
   - 3.39% __libc_start_main
        main
        perform_thread_test
      - test_hybrid_rr
         - 3.32% perform_hybrid_test_rr
              3.30% LockFreeSkiplist::insert
```







A Special Speculation on Implementing Fine-grained Skip List and Thinking

When we implement the fine-grained skip list, we encounter a special case. That is if we instead use an array of ordinary pointers rather than an array of atomic pointers and use ordinary bool instead of atomic bool, the test still generates the correct result in concurrency scenarios.

We think the first reason is that we only do assignments on these variables, and the underlying architecture and the compiler automatically assign these variables on fine granularity and align very well.

The other reason is that these variables only affect program execution on read operations (because write operation requires locks, two writes cannot occur on the same node at the same time). And the order is not that important in our non-blocking read design since reordering these non-atomic operations may change the result of the read functions, but it will finally reach the intended state, which means, a later read will only read out the correct result. And only a

non-blocking read on ongoing write functions may have either true or false returns. But the final correctness is maintained.

Although this is interesting, we still submit an atomic version of the fine-grained skip list. Just for true correctness.

Further Improvements and Experiments

During the project development, we found we had many interesting ideas. We didn't have enough time to explore them due to time constraints.

Batch Insertion

Skip list is popular in the database area. It could potentially replace Red-black trees in the circumstance of sequential inserting (e.g. insert from 1 to 100) and range predicates (e.g. find elements bigger than 10). These operations will result in the Red-black tree to take additional time to rotate and recurse. We originally planned to implement an improved version of fine-grained lock-based skip list as a nice-to-have feature. We planned to have a batch insert operation so that we can better utilize the lock and improve performance in batch insert cases. We investigated this feature but didn't implement it due to time limitations.

Execution Time for Individual Thread

We found the max execution time for a single thread is also interesting. It seems that the Coarse-grained version has a higher chance to have one longer execution thread while time in each thread in lock-free version is more on average. We didn't have enough time to include individual thread time in our report. We believe it could be another interesting point to analyze.

Test on Machine with More Cores

We conduct our test on the GHC machines, which only have 8 cores. We check that the GHC machine does not enable hyperthreading. We run tests with more than 8 threads and results in at almost the same time as 8 threads. Thus, we didn't include the data from more than 8 threads because we believe it is not meaningful. We could test on machines with higher cores to observe the time and speedup for each version in the future.

Relationship Between Height and Total Element

We find the test results on the same number of operations with different heights are not obvious. Theoretically, increasing the height could allow the skip list to hold more elements because it can increase the height of the index list. However, nodes with higher levels could also potentially bring lower performance in fine-grained and lock-free versions because they have to lock/CAS on more levels. There's also some interesting papers to suggest dynamic height based on total elements or unlimited height. This area is worth further researching.

Retry Checkpoint

One of the possible reasons for the low performance of lock-free insert is the penalty of retry. When CAS fails in some cases, the insert operations have to search from the beginning of the skip list and start over again. Instead of searching from the beginning, we could have some retry checkpoint on the search path. Only the current nodes and neighboring nodes matter in the skip list, so any modification on the way to the current node can be ignored. This could save a lot of time when retry happens, but bring additional overhead for successful operations and it is complex to implement.

Cache Utilization

The cache utilization of the skip list and linked list is inherently low. We also found some research papers in this area to improve it. We found an interesting idea that we could store multiple elements in the same node in the upper index list so that we can read and compare multiple elements at the same time. This can better utilize the cache line and possibly use SIMD to read multiple data. However, it will introduce complexity in the upper index list. This upper index list needs to hold multiple elements in each node and involves split and merge operations. This is more similar to a B tree and is beyond this report's topic.

Transaction

Transaction is also another popular way to do concurrency. We also studied papers on this area and investigated the Software Transactional Memory (STM) library but didn't have enough time to follow it to implement another version.

Coarse-grained Lock-based Skip List with Read-write Lock

We could use the read-write lock to replace the mutex so that we can have good parallelism in the contain operations. However, after the experiments we found that the read-write lock will bring an unstable performance. The read-write lock we used favors either reader or writer, which causes that all read operations happen before write or all write operations happen before read. This forced the read operations to perform on either a list with lots of elements or with limited elements, which makes the experiment result not stable and obvious. We could explore this approach further. For example, we could try a read-write lock with fairness to balance the read and write operation workload.

Conclusion

In this project, we implemented coarse-grained lock-based skip list, fine-grained lock-based skip list and lock-free skip list. Coarse-grained versions have better performance under single thread condition because it has the least overhead. Both fine-grained and lock-free versions have good performance under multithreading conditions. For now, our fine-grained version is a little better than lock-free version under low threading condition because it has smaller overhead. Under high threading conditions, they have similar performance and the speedup does not increase

too much. Further study is needed to seek improvement on the lock-free version to reduce the overhead.

List of Work by Students

List of Chen He's work

Investigate on the research paper and discuss the idea together.
Implement coarse-grained skip list.
Implement lock-free skip list.
Design tests for correctness check.
Design tests for performance evaluation.
Conduct tests on lock-free skip list.
Write report for lock-free skip list
Write the remaining part of the report and poster together.

List of Yida Wu's work

Investigate on the research paper and discuss the idea together.
Implement fine-grained skip list.
Design tests for correctness check.
Modified tests for performance evaluation.
Conduct tests on a fine-grained skip list.
Write report for fine-grained skip list
Write the remaining part of the report and poster together.

Distribution of Total Credit

Chen He - 50% (for Lock-free)
Yida Wu - 50% (for Fine-grained)

References

[1] Herlihy, Maurice & Lev, Yossi & Luchangco, Victor & Shavit, Nir. (2010). A Provably Correct Scalable Concurrent Skip List.
[2] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi and W. -F. Wong, "Parallelizing Skip Lists for In-Memory Multi-Core Database Systems," 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 119-122, doi: 10.1109/ICDE.2017.54.
[3] Kobus, Tadeusz & Kokociński, Maciej & Wojciechowski, Paweł. (2021). Jiffy: A Lock-free Skip List with Batch Updates and Snapshots.

[4] Herlihy, Maurice & Lev, Yossi & Luchangco, Victor & Shavit, Nir. (2007). A Simple Optimistic Skiplist Algorithm. 124-138. 10.1007/978-3-540-72951-8_11.
[5] Maurice Herlihy and Nir Shavit. 2008. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[6] Fomitchev, M., & Ruppert, E. (2004). Lock-free linked lists and skip lists. PODC '04.
[7] Zhang, Jingtian & Wu, Sai & Tan, Zeyuan & Chen, Gang & Cheng, Zhushi & Cao, Wei & Gao, Yusong & Feng, Xiaojie. (2019). S3: a scalable in-memory skip-list index for key-value store. Proceedings of the VLDB Endowment. 12. 2183-2194. 10.14778/3352063.3352134.