Concurrent Skiplist - Project Report

VU Advanced Multiprocessor Programming

Natalia Tylek*
Vienna University of Technology

Marlene Riegel*
Vienna University of Technology

Maximilian Kleinegger*
Vienna University of Technology

*All authors contributed equally to this project.

Contents

1 Introduction

This report aims to provide insight into the programming project. It includes all the necessary information about the implementation, the experimental setup, and the results obtained. Furthermore, it outlines the challenges that had to be overcome during the development process.

1.1 Problem Specification

"The skiplist is a probabilistic key-value store data structure with expected O(log n) operation times for lists with n elements. The data structure supports the operations insert(), delete() and contains() on allocated nodes containing an integer key field, a payload, as well as pointers and other fields needed to conveniently implement the operations." [1]

2 Skiplist

This section contains all the necessary implementation details of the various skiplist implementations.
For the concurrent implementations, we chose *OpenMP* for thread management.

2.1 Sequential Implementation

To establish a baseline, we started by implementing a straightforward sequential version, which should not be run concurrently. We looked at example implementations on various websites for inspiration on how to implement a skiplist. [2] [3]

2.1.1 Node Structure

In order to implement the skiplist with the features and functions mentioned above, we started by defining the node struct, which we called *"skiplist_node"*. The attributes of the node are the following:

- *key* (long): The key serves as the identifier of the node for finding the right place in the list to add, remove, or search for a node.
- *value* (void pointer): The data held by the node.
- *top_level* (int): This indicates the highest list level in which the node exists.
- *next* (array of size MAX_LEVEL, the highest possible level for the list): Contains pointers to the successor node at each level. This makes it possible to traverse the list.

```
1 typedef struct _node
2 {
3     long key;
4     void *value;
5     int top_level;
6     struct _node *next[MAX_LEVEL];
7 } skiplist_node;
```

Figure 1: Node Structure.

2.1.2 Implementation

Unlike other implementations, we chose not to implement a tail node. This has some possible advantages and disadvantages: On the one hand, not using a tail node saves memory and eliminates interaction with an additional node. On the other hand, the tail node might help avoid NULL checks when traversing the list and introduce a clearer handling of list boundaries.

```
1 void init(skiplist *list);
2 void clean(skiplist *list);
3
4 int add(skiplist *list, long key, void *value);
5 int rem(skiplist *list, long key);
6 int con(skiplist *list, long key);
```

Figure 2: Implemented Functions.

- *init(...)*: The skiplist struct "skiplist" just contains a header node at the beginning, which is initialized in this method to hold the minimum possible key (INT_MIN), span all levels (top_level = MAX_LEVEL-1) and point to NULL on all levels.
- clean(...): Goes through all nodes of the list and frees the memory that was allocated for each of the nodes.
- *add(...)*: Starting at the highest level and going to the bottom level, this method traverses the list to find the correct insertion points for the new node. The insertion points are kept in an array (update[i]) and updated to point to the new node once it is checked that the key is not already in the list. The new node then gets set to point to the successor nodes of the nodes in the update array.
- *rem(...)*: Similarly to the add(...) method, the list is traversed, and an update array is created. After a check if the wanted key is in the list, the method continues by setting the next pointers of all nodes in the update array to the successor nodes of the node that should be removed.
- *con(...)*: This method also traverses the list using the layered structure for a faster search, returning 1 if the node is found and 0 if it is not in the list.

Additionally, we wrote a helper function *randomLevel*, which helps figure out which *top_level* a new node should have. The levels are distributed randomly, favoring lower levels to make the list balanced.

2.2 Global Lock

For the first concurrent version of the skiplist we used a global lock to lock the whole list before any operations are done. This ensures that each thread traverses and operates on the list as if the implementation were sequential.
For the locking, we chose *the omp_lock_t*, which represents a simple lock. The lock has several simple lock routines available, such as *omp_init_lock*, *omp_set_lock* and *omp_unset_lock*. [4]

2.2.1 Node Structure

The node structure for the globally locking skiplist is the same as for the sequential implementation.

2.2.2 Implementation

The implementation stays the same as the sequential implementation other than adding a global lock to the list.

- *init(...)*: Initializes the global lock with

```
1 omp_init_lock(&list->lock);
```

- clean(...): Destroys the lock with

```
1 omp_idestroy_lock(&list->lock);
```

- *add(...), rem(...), con(...)*: At the start of each of these methods, the global lock is set and the list is locked. After all operations of the method are finished, the lock is released again.

```
1 omp_set_lock(&list->lock);
2 // add
3 // remove
4 // or contain
5 // methods as before
6 omp_unset_lock(&list->lock);
```

2.3 Lazy skiplist with Fine-Grained Locking

For the fine-grained locking implementation of the skiplist, we added node locks, locking single nodes while traversing and operating on the list.

2.3.1 Node Structure

The node structure for the fine-grained locking skiplist is more complex compared the sequential implementation. We used the same attributes as before: *key (long)*, *value (void *)*, *top_level* (int) and *next* (array [MAX_LEVEL]. Then, we also added the *node lock* and two integers, *marked* and *fullyLinked*.

```
1 typedef struct _node
2 {
3     long key;
4     void *value;
5     int top_level;
6     struct _node *next[MAX_LEVEL];
7
8     omp_lock_t lock;
9     volatile int marked;
10    volatile int fullyLinked;
11 } skiplist_node;
```

2.3.2 Implementation

The implementation for the lazy skiplist with fine-grained locking differs from the previous approaches. We added a *find(...)* method for looking for a key in the list.
Our implementation is based heavily on the implementation introduced in the book "The Art of

Multiprocessor Programming". We also tried to find different approaches, but did not succeed in implementing the task in a different way and ultimately decided to go with this approach, also based on already existing correctness proofs. [6]

- *init(...)*: Initializes the lock of the header node and sets header→marked to 0 and header→fullyLinked to 1.
- *clean(...)*: Destroys the lock for all nodes in the list before freeing them.
- *find(...)*: Traverses the list to find predecessors and successors of a node with a certain key and returns lfound, the highest level where the node has been found in the list. It may also return -1, if the key is not found in the list.
- *add(...)*: Utilizing find(...) to locate the insertion points of the new node, this method returns unsuccessful, if the key already exists and the node is fullyLinked. If the node exists, but is marked, the method continues as if the key was not in the list. To find the right nodes to be locked, we look in the predecessor array (filled by the find(...) method) for unique nodes that have to be updated when inserting the new node.

```
1 skiplist_node *nodesToLock[MAX_LEVEL];
2 int numNodesToLock = 0;
3 nodesToLock[numNodesToLock++] = preds[0];
4 for (int level = 1; level <= topLevel; level++) {
5     if (preds[level]->key != preds[level - 1]->key)
6 {
7 nodesToLock[numNodesToLock++] = preds[level];
```

Figure 3: Unique Node Locking

Afterwards, the found nodes are locked. The locked nodes are validated by checking whether they are reachable and correctly point to the successor nodes. After validation, the new node is initialized, its forward pointers are set, and it is linked into the list by changing the pointers of the predecessors. Finally, the node is marked as fullyLinked and the acquired locks are released.

- *rem(...)*: Again using *find(...)* to look for the node in the list and its predecessors and successors. Then, it is checked, whether the node is ready for deletion. In the first iteration, fullyLinked, marked and top_level of the victim node have to be correct. After the first iteration, this has already been checked and isMarked is set to 1 to indicate that the node is in the process of being deleted and the node is logically deleted by setting victim→marked = 1.
  Afterwards, we continue by locking the correct unique predecessor nodes and validate them again after locking to check whether they are still reachable and have victim as their successor.
  If the check is successful, the victim is physically removed.
- *con(...)*: Traverses the list using the layered structure.

2.3.3 Linearization Points

The linearization points for successful and unsuccessful method calls differ for the add(...) and the remove(...) method.

- *add(...)*: For a successful call, the linearization point is line 162, setting the new nodes fullyLinked flag to 1, making the new node accessible to other threads.

```
for (int level = 0; level <= topLevel; level++) {
    new_node->next[level] = succs[level];
    preds[level]->next[level] = new_node;
}
new_node->fullyLinked = 1; // Linearization point
```

Figure 4: Linearization Point of successful add(...)

For an unsuccessful call, it is at line 100:

```
if (!nodeFound->marked){
    while (!nodeFound->fullyLinked)
    {
    }
    return 0; // Linearization point
}
```

Figure 5: Linearization Point of unsuccessful add(...)

- *rem(...)*: For a successful call, the linearization point is line 218 . For an unsuccessful call, it can be either line 202, if the key was not found, or line 215, if the node was already marked by another thread.

```
if (isMarked == 0)
{
    topLevel = victim->top_level;
    omp_set_lock(&victim->lock);
    if (victim->marked == 1)
    {
        omp_unset_lock(&victim->lock);
        return 0; // Linearization point of unsuccessful call
    }
    victim->marked = 1;
    isMarked = 1; // Linearization point of successful call
}
```

Figure 6: Possible Linearization Points

```
int lFound = find(list, key, preds, succs);
if (lFound == -1)
{
    return 0; // Linearization point
}
```

Figure 7: Possible Linearization Point of unsuccessful rem(...)

- *con(...)*: The linearization point occurs when the method returns either 0 or 1 based on whether it found a marked node or an unmarked, fully linked node with the correct key.

2.4 Lock-free skiplist

For the implementation of the lock-free skiplist, we again looked closely at the implementation suggested in "The Art of Multiprocessor Programming", using a similar way to structure the

code and implement method calls. [6]
We also partly based the implementation on the lock-free linked list implementation found on the github repository by parlab-tuwien, where we found examples for managing atomicity, marks, and pointers. Additionally, the retry structure was also inspired by this implementation. [5]

2.4.1 Node Structure

The node structure for the lock-free implementation is similar to the structure for the sequential implementation and the global lock. The next array uses the *_Atomic* keyword to indicate atomic types.

```
1 typedef struct _node
2 {
3     long key;
4     void *value;
5     int top_level;
6     _Atomic(struct _node *) next[MAX_LEVEL];
7 } skiplist_node;
```

Figure 8: Node Structure of Lock-free Implementation

2.4.2 Implementation

For this implementation, the atomic Compare and Swap "atomic_compare_exchange_strong_explicit" was used to ensure waitfreeness.

- *init(...)*: Initializes the header node of the new list.
- *clean(...)*: Frees all nodes.
- *find(...)*: This method starts traversing the list at the top_level and proceeds down the list, recording predecessors and successors at each level. If it encounters a marked node on its way, it physically removes it from the list. In order to do this, a CompareAndSet operation is used to set the pred→next pointer to the successor of the node that should be removed. The method returns true if the node has been found in the list.

```
1 while (ismarked(LOAD(&curr->next[level])))
2 {
3     if (!CAS(&pred->next[level], &curr, succ)) {
4         goto retry;
5     }
6     ...
7 }
```

Figure 9: CAS to remove marked nodes

- *add(...)*: The method calls find(...) to get the correct insertion points for the new node. Afterwards, the next pointers of the new node are set first. Then, the method tries to insert the new node on level 0 by using CAS to set the next pointer of the predecessor node while checking that the predecessor still correctly pointed to the successor node. If that is not the case, the method tries again. If the CAS was successful, the method continues for all levels up until the top_level.

- *rem(...)*: Again, find(...) is called to locate the victim node in the list. If the node is found, the method retrieves the successor node, starting at the highest level and continuing down to level 1, and uses CAS to mark the next pointer of the victim node. Once level 0 is reached, after a successful CAS, find(...) is called again to physically remove the node from the list.

- *con(...)*: The method traverses the list similar to the find(...) method, but does not remove marked nodes, instead it does not alter the list, which makes the method read-only and reduces interference. If a marked node is encountered, the method just skips over it, simply moving on to the next unmarked node.

2.4.3 Linearization Points

- *find(...)*: The linearization point of successful and unsuccessful calls is line 57 or 70, when the curr pointer at level 0 is set in the last iteration, since here the information whether the node is in the list or not is gathered.

```
1 curr = getpointer(LOAD(&pred->next[level])); // Linearization point
```

Figure 10: Linearization Point of find(...)

- *add(...)*: The linearization point of the unsuccessful call is the same as the linearization point of the find(...) method, line 132 in our code. For the successful call, the linearization point is the CAS operation to add the node on level 0 (line 157), since from that point on, the node is visible to other threads.

```
1 if (!CAS(&preds[0]->next[0], &succs[0], newNode)) // Linearization point
2 {
3     free(newNode);
4     continue;
5 }
```

Figure 11: Linearization Point of the successfull add(...)

- *rem(...)*: For the unsuccessful call, the linearization point is the same as the linearization point of the find(...) method, line 169 in our code. For the successful call, the linearization point is the CAS operation to mark the node on level 0 (line 195). From that point on, the node is logically removed and will be physically removed by the next find(...) call.

```
1 int success =
2     CAS(&nodeToRemove->next[bottomLevel], &bottomNext, setmark(bottomNext));
3 bottomNext = getpointer(LOAD(&nodeToRemove->next[bottomLevel]));
```

Figure 12: Linearization Point of the successfull rem(...)

- *con(...)*: The linearization point occurs when the method returns either 0 or 1 based on whether it found an unmarked node with the correct key.

3 Experimental Setup

Our benchmark consists of two main components: a benchmarking library written in C and a CPython integration that calls this C library. The Python script is responsible for taking in the parameters specified in Section 3.1, storing the results, and averaging them. The core functionality, however, is implemented in the C library, which essentially performs all the tasks outlined in the exercise description.

Using the *OpenMP* framework for thread management, only a single thread is used to perform a basic correctness test, as shown below using the

```
1 #pragma omp single
2 {
3     basic_correctness_test(...)
4 }
```

Figure 13: Single thread for basic correctness test.

In addition to the correctness test, the C library calculates various metrics, including: *Average time per experiment, Total number of operations, Number of operations per thread, Number of successful insertions, Number to total insertions, Number of successful deletions, Number of total deletions, Number of successful contains, Number of total contains, Number of total operations, Number of total operations per thread.* These metrics are returned to the Python script for further processing.

To execute the actual benchmark, we set the number of threads beforehand and start the parallel execution using the following directive:

```
1 omp_set_num_threads(num_of_threads);
2 ...
3 #pragma omp parallel
4 {
5     #benchmark
6     ...
7 }
```

Figure 14: Parallel execution of the benchmark.

The benchmark begins with key selection, where the respective selection strategy is used to obtain a key. Next, a random number determines which operation, insert, delete, or contains, is executed. The selected operation is then performed, and the results are recorded.

After completing the benchmark operations, all threads synchronize using the barrier directive *#pragma omp barrier*. This ensures that all threads have completed their tasks before the statistics are aggregated.

Once the benchmark is complete, the previously mentioned metrics are collected and returned. The skiplist is then freed to release memory. The stopping criterion is solely defined by the runtime, which can be specified, and the number of operations per thread is also counted and included in the final results.

3.1 Configurations

We continue by presenting the configurations which were evaluated through the benchmark.

Implementation	Ex.	Threads	Key Range	Range Type	Op. Mixes	Time
Sequential Skip List	E2	1	[0 .. 100,000]	N/A	(10%,10%,80%) (40%,40%,20%)	1 s 5 s
Global Lock	E3	1,2,4,8, 10,20,40,64	[0 .. 100,000]	Shared Disjoint	(10%,10%,80%) (40%,40%,20%)	1 s 5 s
Fine-Grained Lock	E4	1,2,4,8, 10,20,40,64	[0 .. 100,000]	Shared Disjoint	(10%,10%,80%) (40%,40%,20%)	1 s 5 s
Lock-free	E5	1,2,4,8, 10,20,40,64	[0 .. 100,000]	Shared Disjoint	(10%,10%,80%) (40%,40%,20%)	1 s 5 s

Table 1: Summary of Skiplist Benchmark Configurations

Remarks:

- Shared vs. Disjoint: Shared means that all threads select keys from the same range, potentially causing high contention. Disjoint means that each thread is assigned its own non-overlapping subrange of keys, lowering contention.
- We successfully ran a **full benchmark** for the *sequential*, *global-locking*, and *fine-grained-locking* libraries with 3 repetitions, taking average results of attempted and successful operations.
- The full benchmark of the *lock-free* library was run with only one repetition per configuration, because longer runs were canceled due to a time limit (the execution did not progress quickly enough, possibly caused by starvation issues).
- Example job scheduling for a full benchmark can be seen below for fine-grained locking.

```
bench-custom: all
    python ./benchmark.py --library library_finelocking.so \
        --repetitions-per-point 3 \
        --num-of-threads 1 2 4 8 10 20 40 64 \
        --base-range 0 100000 \
        --runtime-in-sec 1 5 \
        --operations-mix 40 40 20 \
        --selection-strategy 0 \
        --basic-testing \
        --seed 42 \
        --basedir . \
        --name fine_lock
```

3.2 Computational Environment

As a computational environment the Nebula cluster provided by the lecture team was used. To save some space, we recommend visiting the Homepage[1] for more information concerning the cluster specification.

4 Results

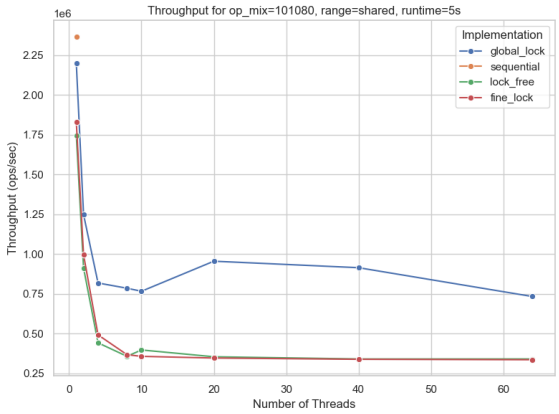
This section presents all the results gathered from the experiments. Since it is not strictly a result but rather a basic requirement for the system, we checked the basic test for all runs, which always returned *true*, thereby ensuring that the correctness of the given skiplist implementation is maintained in each run.

4.1 Throughput

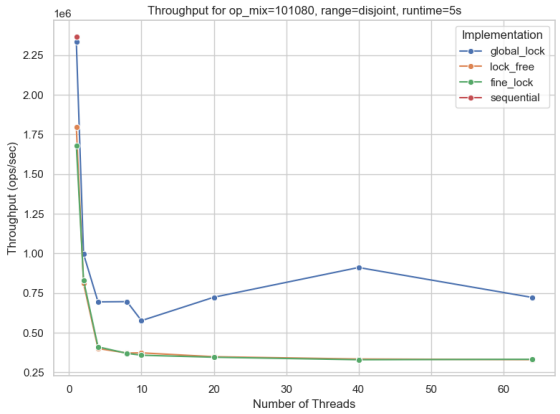
First, we looked at the throughput vs. number of threads for disjoint and shared ranges and for operation mixes 10-10-80 and 40-40-20. We plotted the results from the runs that took 5 seconds. This is due to the greater runtime and no notable difference compared to the 1 seconds runs. It can be seen in Figure 15 that for both operation mixes and ranges, the global

[1] `https://doku.par.tuwien.ac.at/docs/machines/nebula/`

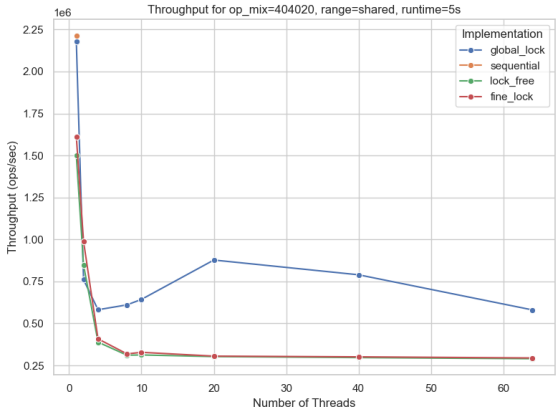
lock implementation produces the highest throughput. This might be rooted in the minimal locking overhead introduced by the global lock compared to the other implementations. However, this comes at the cost of reduced parallelism as only one thread can operate on the skiplist at any time. In the plot it seems that around 20-40 threads might be a sweet spot for this implementation, as for more than 20-40 threads the throughput seems to decrease.
The fine-grained locking approach as well as the lock-free variation show lower throughput regardless of the number of threads. Interestingly, different from the global lock, the throughput does not decrease any more after increasing the number of threads to more than 10. This might indicate better scalability than the global lock.



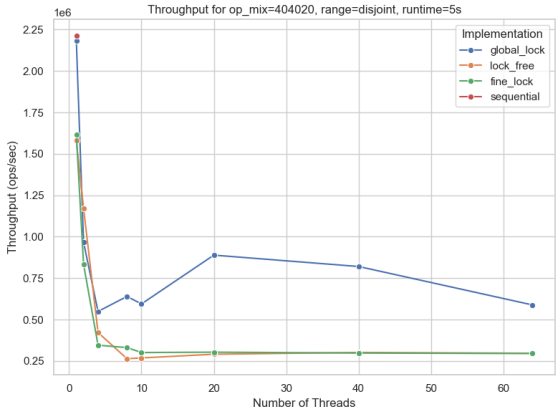
(a) Throughput for Operation Mix 10-10-80, Shared Range, 5s Runtime.



(b) Throughput for Operation Mix 10-10-80, Disjoint Range, 5s Runtime.



(c) Throughput for Operation Mix 40-40-20, Shared Range, 5s Runtime.



(d) Throughput for Operation Mix 40-40-20, Disjoint Range, 5s Runtime.

Figure 15: Comparison of Throughput for multiple operation mixes and key-ranges (5s runtime).

4.2 Total and Successful Operations

Afterwards, we also looked at the number of total vs successful operations, here for our plots we chose to display inserts as an example, since contains and remove operations had a similar level of success rate. It can be seen in 16, 17, 18 and 19 that for different operation mixes and ranges, the success rate is pretty similar and always lies around 50-60%. It is important to note that successful inserts indicate method calls where the wanted node has successfully been added to the list and unsuccessful can either indicate that the node was already in the list or a failed memory allocation.
The ratio of total and successful operations indicates that for an operation mix of 40-40-20, the global lock has a slightly lower success rate for inserts than the other two implementations, but a slightly higher one for deletions. For an operation mix of 10-10-80, the same pattern occurs.
The reason for this might the higher throughput of the global lock, leading to more keys already existing in the list and therefore leading to an unsuccessful insert.
Furthermore, it is notable that with the *lock-free* implementation, we encountered issues during testing related to starvation. This is entirely explainable, as lock-free does not imply starvation-free; however, we observed this behavior even when using only four concurrent threads. This was quite frustrating and led to a problem investigation that could not be resolved, as the algorithm itself is not modifiable.
All in all, the consistent success rate of the operations across different ranges and mixes suggests robust implementations that handle concurrency and maintain correctness of the list.
We did not include any plots related to remove or contains operations, as they exhibited similar patterns and lacked any notable features. To save space, we omitted them from the report; however, they can be discussed during the presentation or reviewed in our hand-in, where you can easily generate the corresponding plots.

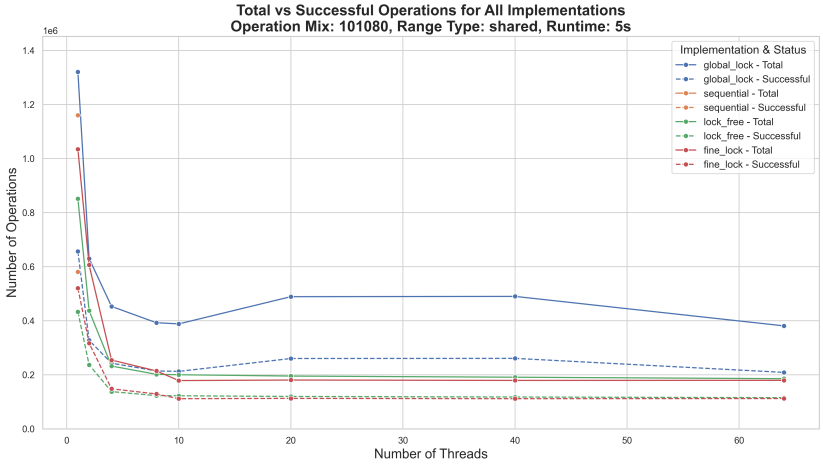


Figure 16: Total and Successful Inserts for Operation Mix 10-10-80, Shared Range, 5s Runtime.

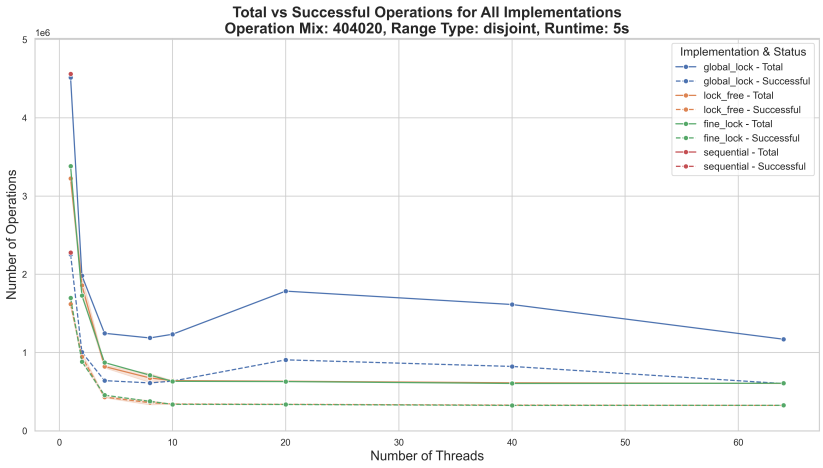


Figure 17: Total and Successful Inserts for Operation Mix 40-40-20, Disjoint Range, 5s Runtime.

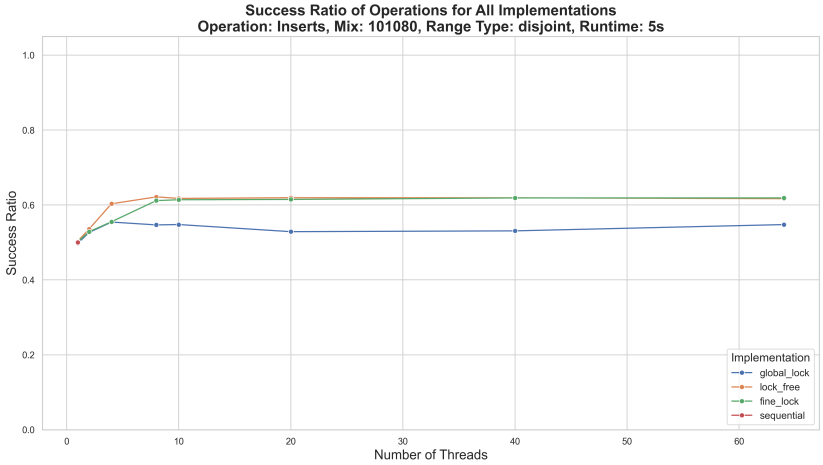


Figure 18: Ratio of Total and Successful Inserts for Operation Mix 10-10-80, Disjoint Range, 5s Runtime.

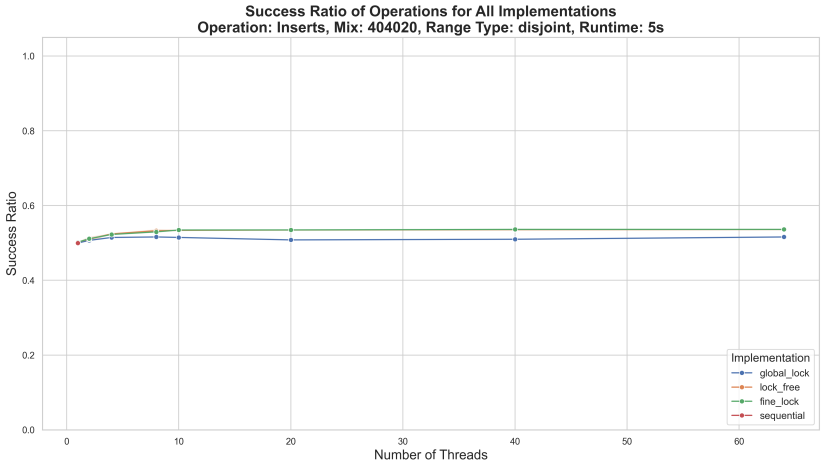


Figure 19: Ratio of Total and Successful Inserts for Operation Mix 40-40-20, Disjoint Range, 5s Runtime.

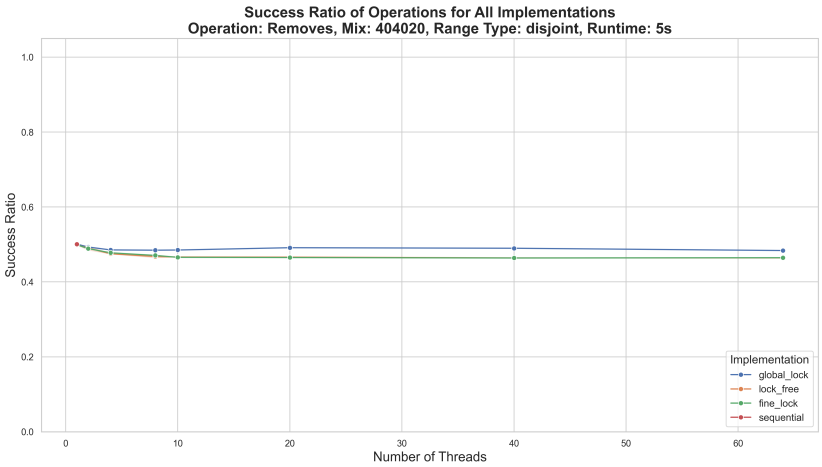
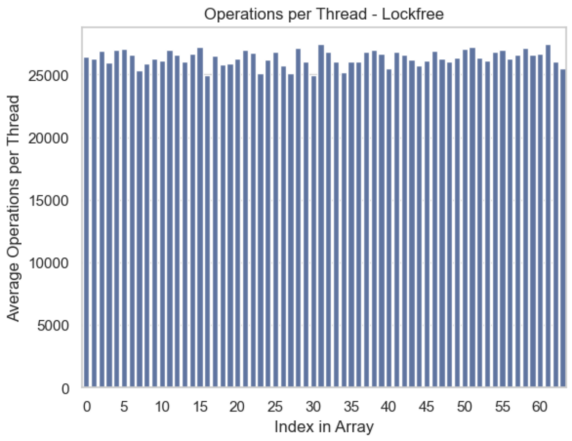


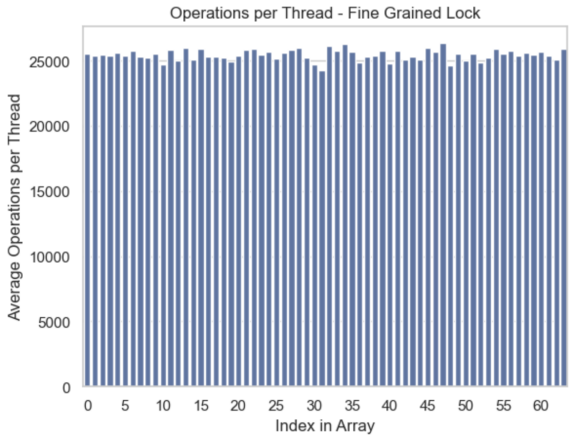
Figure 20: Ratio of Total and Successful Inserts for Operation Mix 40-40-20, Disjoint Range, 5s Runtime.

4.3 Operations per Thread

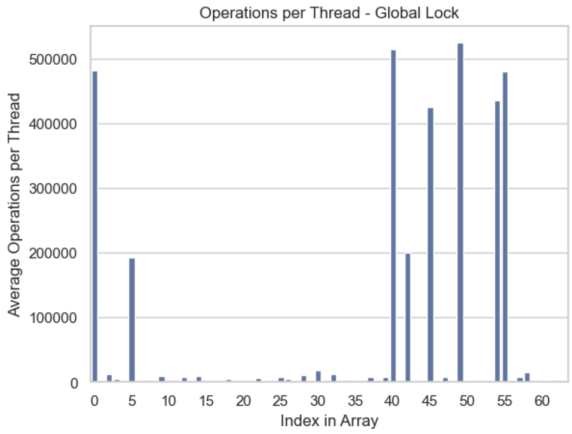
Finally, we wanted to examine the distribution of workload between the different implementations. This can be seen in Figure 21. It can be seen that for the global lock, few threads execute a disproportionally large number of operations, while other threads do barely any operations. For fine-grained locking and the lock-free implementation, the work is much more evenly shared between the threads. The reason for this might be the bottleneck caused by the global lock, where more threads contend for the lock which may lead to scenarios where some threads acquire the lock much more often than other threads, resulting in limited parallelism. In contrast, fine-grained locking allows multiple threads to work on different sections of the list concurrently, only locking the nodes that are currently modified.
Similarly, the lock-free implementation also allows multiple threads to manipulate different parts of the list at the same time. This leads to better utilization of all threads.



(a) Operations per Thread - Lock-Free.



(b) Operations per Thread - Fine-Grained Lock.



(c) Operations per Thread - Global Lock.

Figure 21: Comparison of Operations per Thread for Different Locking Mechanisms (Operation Mix 10-10-80, Shared Range, 5s Runtime).

5 Conclusion

This project explored three concurrent skiplist implementations in addition to the sequential one.
The global lock showed the highest throughput due to its simplicity and minimal locking overhead. However, it suffered from an uneven workload distribution and limited scalability, which could be rooted in higher contention compared to the other implementations.
In contrast, the fine-grained locking approach showed better workload distribution and therefore better potential scalability, but had an overall lower throughput for the configurations we observed. This might be caused by the higher locking overhead in case of the fine-grained locking algorithm.
The lock-free variation faced challenges due to starvation and longer execution times. It also offered a decreased throughput compared to the global lock, but was able to distribute the workload evenly among threads, similar to the fine-grained variation.

To further improve the concurrency and performance of the skiplist implementation, it would be recommended to try and reduce the starvation issues faced by the lock-free variation. Additionally, one could explore hybrid approaches to combine the strengths of the different algorithms.

References

[1] Prof. Dr. Jesper Larsson Träff, *Advanced Multiprocessor Programming (AMP) WS 2024/25 Programming Project*, 2024.

[2] https://www.geeksforgeeks.org/skip-list/

[3] https://www.baeldung.com/java-skiplist

[4] https://www.openmp.org/spec-html/5.0/openmpse31.html

[5] https://github.com/parlab-tuwien/lockfree-linked-list/tree/main

[6] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear, *The Art of Multiprocessor Programming*, Version 2, 2021.