Advanced Multiprocessor Programming
FIFO Queues

Konstantin Röhrl
Daniel Schloms

June 2021

1 Introduction

For our project we chose project 23 with the aim to implement and evaluate a lock-free FIFO queue. The background for our work was a paper by Ruslan Nikolaev [Nik19], where the author explains the theory and pseudocode for different implementations, such as a naive version as well or the scalable version on which we focussed, can be found. We chose C++ as our implementation language for multiple reasons. First, we simply prefer the language over Java and we both wanted to improve our C++. Furthermore, our goal was to evaluate the queue based on performance or throughput, which favours C++. Additionally, the author of the paper also had a C implementation of the queue in a GitHub repository [Nik20], where we could see how he implemented the entry object and learned how he changed that object efficiently using bit manipulation. How we tested and evaluated the queue will be explained below, but in short, first we wanted to run tests to ensure that the queue works as expected, and then we wanted to benchmark the queue and compare it to a baseline. For this baseline we implemented 2 versions of a lock-based queue as seen in the lecture slides.

2 Overview of the different Queue Types

2.1 Locking queue with one lock

The first queue type which we would like to present is the locking queue with one lock. This queue type wraps locks on the operations `enq()` and `deq()`. As the name suggests, it is providing one lock for either the `enq()` or `deq()` operation as seen in figure [1] and [2], which means that only one thread can successfully execute one of these operations at a time. We have chosen to use `std::mutex` for the lock in our implementation.

```
1  bool LockQueue::enq(int x){
2      lock->lock();
3      if (tail - head == this->size){
4          // Queue full, unlock and return false
5          lock->unlock();
6          return false;
7      }
8      items[tail % this->size] = x;
9      this->tail ++;
10     // Enqueue successful, unlock and return true
11     lock->unlock();
12     return true;
13 }
```

Figure 1: The enqueue operation

```
int LockQueue::deq(int * error_code){
   lock->lock();
   if (tail == head){
        // Queue empty: unlock, set error code to -1 and return 0
        lock->unlock();
        *error_code = -1;
        return 0;
   }
   int x = items[head % this->size];
   head ++;
   // Dequeue successful: unlock, set error code to 0 and return the value
   lock->unlock();
   *error_code = 0;
   return x;
}
```

Figure 2: The dequeue operation

2.2 Locking queue with two locks

The locking queue with two locks basically works exactly the same way as the version with one lock. The only difference is the fact that it uses two different locks for the `enq()` and `deq()` operations. With this approach, different threads can concurrently execute these operations. Due to the fact that a lock acts as a bottleneck, we expect a slight improvement in the performance in comparison to the version with one lock.

2.3 Lock-Free Queue Data Structure

Both the Naive Circular 2.4 and the Scalable Circular Queue, which are presented in chapter [2.4] and [2.5], are the "backend" for a simple type of interface so to say. This is to say that neither queue actually stores data entries themselves, but only records indices to the actual piece of data (indirection). The data is managed in the aforementioned interface, along with two queues. One queue records the available/free indices, the other the occupied slots (`fq` and `aq` respectively).
To enqueue a data element, an available index has to be dequeued from `fq`. If the queue is full, the first dequeue operation will return -1, after which the function returns unsuccessful. Otherwise the data is stored at the dequeued index position in the data array. Then this index is enqueued in `aq`.
When dequeuing a data element, an index is dequeued from `aq`. If the queue is empty this operation will return -1 and the function returns unsuccessfully. Otherwise the data element at this index is returned and that index is enqueued into `fq` again.

2.4 Naive Circular Queue

The Naive Circular Queue implements a non-optimal version of the circular queue. To enqueue an index, the tail value is loaded and its cycle is checked against the cycle of the entry at position `k = tail % n`, with n being the size of the data array (and also the queue). The cycle is just how many times the head/tail have wrapped around the ring buffer already. If the cycles matches, then the tail is already behind and must be incremented. If the tail is exactly 1 cycle ahead of the entry, the entry is replaced by using a CAS loop.

Dequeuing is similar, the cycle of the head is checked to see if it matches the required entry, otherwise the queue is either empty or the head value is stale. At the end the head is incremented by using a CAS loop, after which the index is returned.

While the NCQ is lock-free, as proven in the paper [Nik19], there are some flaws. First, threads could get stuck in the CAS loops at the end of the enqueue and dequeue operations if another thread manages to change the entry or head just before the loop is reached. Furthermore, CAS is not as performant as other atomic instructions that are used in the SCQ.

2.5 Scalable circular queue

The SCQ makes some major improvements over the NCQ. First of all, there are no CAS loops any more, so single threads can't get stuck in them. Additionally, many operations leverage better performing atomic instructions. In both the enqueue and dequeue operation the tail and head values are incremented by using FAA. Moreover, when consuming entries while dequeuing, an atomic or instruction can be used to change the entry instead of CAS.

SCQ size: The size of an SCQ is doubled from n to $2n$. Along with using CAS to enqueue entries this makes it possible to bound the maximum distance between the last dequeuer and the last enqueuer, which is not the case with the infinite array queue (not NCQ!) presented in the paper that introduces the threshold idea first. This is needed because as one can see in the infinite array queue, the SWAP operation that is used to enqueue indices just swaps the entry without caring if the slot is occupied by a previous cycle. To clarify, the actual data array still only has n values and only indices from 0 to $n-1$ can be dequeued from `aq` and `fq`.

Livelocks: Protection against livelocks is given by introducing the aforementioned threshold variable. According to the paper, a livelock happens if dequeuers constantly invalidate slots that enqueuers want to use. This is remedied by setting this threshold value (to $3n-1$ in the case of SCQ) when successfully enqueuing an index. Each time a dequeuer fails, the threshold is decremented, and if it falls below 0, no dequeuer will even attempt to dequeue an element until an enqueue operation has reset the threshold. This way, there is also a fast path for the dequeuer in case of an enqueue on a full queue or a dequeue on an empty queue.

Initialization: SCQs can be initialized in two ways: either the queue is filled at the beginning or it is empty. Logically, `aq` is empty in the beginning and `fq` should contain all available indices. One can see in initialization function 3 that, depending on the `full` argument, the head, tail, threshold, and entries are initialized differently. In a full queue, the tail value is ahead of the head value by the size of the data array, the threshold is set so that a dequeuer has the maximum amount of tries, and the first half of entries contain all the data indices. In an empty queue the head and tail values are the same, the threshold is set to -1 as no dequeuer should be able to dequeue, and all entries are initialized with a special index ($2n-1$, sometimes called `F_INDEX` in our code) that signifies that an enqueuer can write an index into that entry.

```
1  SCQ::SCQ(int capacity, bool full){
2    this->threshold = (std::atomic<int>*)malloc(sizeof(std::atomic<int>));
3    this->head = (std::atomic<size_t>*)malloc(sizeof(std::atomic<size_t>));
4    this->tail = (std::atomic<size_t>*)malloc(sizeof(std::atomic<size_t>));
5    size = capacity;
6    if (full){
7      this->threshold->store(3*capacity-1);
8      this->head->store(0);
9      this->tail->store(capacity);
10   }
11   else{
12     this->threshold->store(-1);
13     this->head->store(2*capacity);
14     this->tail->store(2*capacity);
15   }
16   for (size_t i = 0; i < 2*capacity; i++){
17     Entry e;
18     e.entr = new std::atomic<uint64_t>(i);
19     if (!full){
20         e.entr->fetch_or(2*size - 1);
21     }
22     else{
23       if (i < capacity){
24
25       }
26       else{
27         e.entr->fetch_or(2*size-1);
28       }
29     }
30     entries_lli.push_back(e);
31   }
32 }
```

Figure 3: Initialization of the SCQ

Implementation: Both the SCQ enqueue and dequeue functions in this report have been inspired by the GitHub repository linked in the paper [Nik20]. By this we refer to the usage of an unsigned 64 bit integer as an entry and using bit manipulation to change the entry. The first iteration did not make use of those more efficient operations, there, an entry struct with 3 fields + padding was used. When testing against the bit-manipulation based implementation it was quite a bit worse, especially since the atomic or operation in the dequeue function can only be used with so called "integral types" (which a struct is not).

SCQ Enqueue: The enqueue operation 4 begins with a while loop in which the tail is fetched and incremented and from that value the corresponding data array index `j` is calculated. This entry is loaded and its cycle and the tail cycle are calculated. In the tail cycle calculation the tail is shifted left by one to compensate for the ORing with $2n-1$ instead of $n-1$. This is also done in the dequeue function when calculating the head cycle and makes it easy to compare it to the entry cycle (by also ORing $2n-1$) or use the cycle to make a new entry. Then it is checked whether the entry is eligible to take a new index. This condition looks complicated but can be broken down according to the pseudocode in the paper: the entry cycle must be smaller than the tail cycle AND the entry contains the special index AND (the entry must be marked safe OR all dequeuers are behind). The author of the paper doesn't add an extra `is_safe` bit in the entry, he reuses the $ld(2*n)$th bit for that purpose. This means that one does not need to check for the special index and the safety bit by themselves. Instead, if the special index is detected (`ent == ent_cycle`) then the entry is already safe. If the index is $n-1$ and the safety bit is not set, then the dequeuers must be behind so that the entry can be enqueued (conditions after the OR operator). A CAS then tries to update the entry, if it fails then the process is repeated from line 11.

```
1  bool SCQ::enq(uint64_t index){
2    index ^= (size - 1);
3    while (true){
4      size_t t = tail->fetch_add(1);
5      // In the pseudocode, cache_remap is used to reduce false sharing
6      // j = cache_remap(T % (2*n))
7      size_t j = t % (2*size);
8
9      entry_load_enq:
10     // Pseudocode in the paper uses goto, so wewe do as well
11     uint64_t ent = entries_lli[j].entr->load();
12
13     uint64_t ent_cycle = ent | (2*size - 1);
14     uint64_t t_cycle = (t << 1) | (2*size - 1);
15
16     if (ent_cycle < t_cycle &&
17         (ent == ent_cycle ||
18         ((ent == ent_cycle ^ size) && (head->load() <= t)))){
19
20       uint64_t new_entry = t_cycle ^ index;
21       if (!entries_lli[j].entr->compare_exchange_weak(ent, new_entry)){
22         goto entry_load_enq;
23       }
24
25       if (threshold->load() != (3*size) - 1){
26         threshold->store((3*size) - 1);
27       }
28
29       return true;
30     }
31   }
32 }
```

Figure 4: SCQ enqueue operation

SCQ Dequeue: The dequeue operation in figure [6] begins with an if statement that is the fast path for empty queues, if the threshold is less than 0, the dequeuer doesn't even have to bother trying. Otherwise the head is fetched and incremented and the entry index is calculated along with the entry- and head cycles. If the cycles match then the index contained in the entry is returned, otherwise the entry is marked unsafe and the cycle is updated if the entry is empty and the head cycle is ahead. If the CAS that is responsible for that update fails, then the process is repeated from line 11. If the CAS is successful, then the function returns unsuccessfully if the tail is behind the head + 1 or if the threshold falls below 0, otherwise the dequeuer repeats the while loop. Additionally, if the head is ahead of the tail, a catchup function is called that updates the tail to the head + 1 in a CAS loop (see figure [5]).

```
1  void SCQ::catchup(size_t t, size_t h){
2    while (!tail->compare_exchange_weak(t, h)){
3      h = head->load();
4      t = tail->load();
5      if (t >= h){
6        break;
7      }
8    }
9  }
```

Figure 5: SCQ catchup function

```
int SCQ::deq(){
  // Checks if queue is empty
  if (threshold->load() < 0){
      is_empty = true;
      return -1;
  }
  while (true){
    size_t h = head->fetch_add(1);
    size_t j = h % (2*size);
    entry_load_deq:
    uint64_t ent = entries_lli[j].entr->load();
    uint64_t ent_cycle = ent | (2*size - 1);
    uint64_t h_cycle = (h << 1) | (2*size - 1);
    if (ent_cycle == h_cycle){
      entries_lli[j].entr->fetch_or(size - 1);
      return ent & (size - 1);
    }
    uint64_t new_ent = ent & (~size);
    if ((ent | size) == ent_cycle){
      new_ent = h_cycle | (ent & size);
    }
    if (ent_cycle < h_cycle){
      if (!entries_lli[j].entr->compare_exchange_weak(ent, new_ent)){
        goto entry_load_deq;
      }
    }

    size_t t = tail->load();
    if (t <= h + 1){
      catchup(t, h+1);
      threshold->fetch_sub(1);
      return -1;
    }
    if (threshold->fetch_sub(1) <= 0){
      is_empty = true;
      return -1;
    }
  }
}
```

Figure 6: SCQ dequeue operation

Correctness: As mentioned in the lecture slides, a full correctness proof is not required, but a intuitive argument about the correctness can still be made. To enqueue a value, an index must be dequeued from `fq`. If one tries to enqueue more than the allowed number of elements, the entry and head cycles will mismatch and any dequeuers will fail/take the fast path after spinning for a finite amount of times. When dequeuing, the same argument can be made for the `aq`. This means in any case the relevant enqueue operations can only start when there is a slot reserved for them which they will always find, so the unbounded while loop in the enqueue function is not dangerous.

Testing for correctness: The correctness for the serial case was easy to evaluate: Dequeues on empty queues were correctly identified by writing -1 to the error code, and enqueues on full queues returned false. Concurrent correctness is a bit harder to evaluate. For a simple case we tried concurrently enqueueing 32 values into a queue of size 16, then concurrently dequeued 32 values. Afterwards we compared the successful enqueues with the successful dequeues where we saw that they matched. When multiple enqueuers and dequeuers are constantly enqueueing and dequeueing, checking for 100% correctness would be out of the scope of this assignment, however, there is an easy way to check for sanity. We introduced an array of atomic integers with 32 elements, 1 for each thread (i.e. we tested with 32 threads). As each thread just enqueues its thread ID, when enqueuing, a thread simply increments its own element, and when dequeuing, the element at the returned index is decremented. At the end the dequeuers are allowed to dequeue all elements before terminating, which means that all elements in that array should be 0. At this time there

is a bug present where at most 1 element will be enqueued one time too many. We suspect that it possible for an index that is equal to the size - 1 to be enqueued into `fq` the wrong way such that a dequeuer will not recognize it as a dequeueable element. However, this bug proved to be extremely hard to track, as not only does the bug not manifest every time, but when it does it occurs at a random time. That being said, as this bug only affects the last index, the impact on the benchmarking should be negible, because when it occurs, the penalty is that the last entry of the data array is now blocked. With a queue size of 2^{16} elements this should not have an impact on any measurements.

Correctness update from a later date: We removed the correctness evaluation from the demonstration program, and now we cannot recreate the bug anymore. This means that the successful enqueues match the successful dequeues + any left over elements in the queue every time, but there is no check whether every enqueue has a corresponding dequeue. However, as all but one index already worked every time with the correctness evaluation in place, we suspect that there was some very weird interference from the correctness evaluation, but there are obviously no guarantees.

False sharing: In the paper the author mentions and uses a special cache remap function that ensures that different entries are stored on different cache lines. In our implementation, we wrap all entries in a struct which contains a padding array that fills out a 64 byte cache line. For the throughput test in the demonstration program (not the benchmark, which doesn't use thread local counters anyway) we use a cache offset in our counter arrays.

3 Benchmarks

3.1 Overview

In order to compare our results in the best manner, we adapted our benchmarks to the ones presented in the paper [Nik19]. Each data point has been measured 30 times and the average value as well as the confidence interval is presented as the result. In order to not exhaust the compute systems, we reduced the total number of operations for each measurement to 1000000. The capacity for the queues are set to 2^{16}. We use millions of operations per second as a measurement for performance.

All benchmarks are executed on the Nebula System of the TU Vienna which provides up to 64 threads for our measurements. Further information for this computing cluster can be read in the documentation [Doc].

3.2 Dequeue operation on the empty queue

Our experiments start with measuring the performance of a single `deq()` operation on the empty queue. The results are shown in figure [7]. The fact, that both lock based queues perform the same is quite obvious, because only the `deq()` operation is executed and therefore the double lock queue can not utilize both locks. Obviously, the naive circular queue performs the worst. The scalable circular queue has by far the best performance and the fast path in the `deq()` might be one reason for it.
In contrast to the benchmarks in the paper [Nik19], the throughput starts to decrease with a larger number of threads. This happens in other benchmarks in the paper as well and the author states that the computer architecture can be a factor in when the performance starts to decrease with a larger number of threads. Another curiosity was that local tests without extensive multithreading often massively outperformed the Nebula system, managing up to 5 times the throughput of Nebula in the non-benchmark demo throughput test.

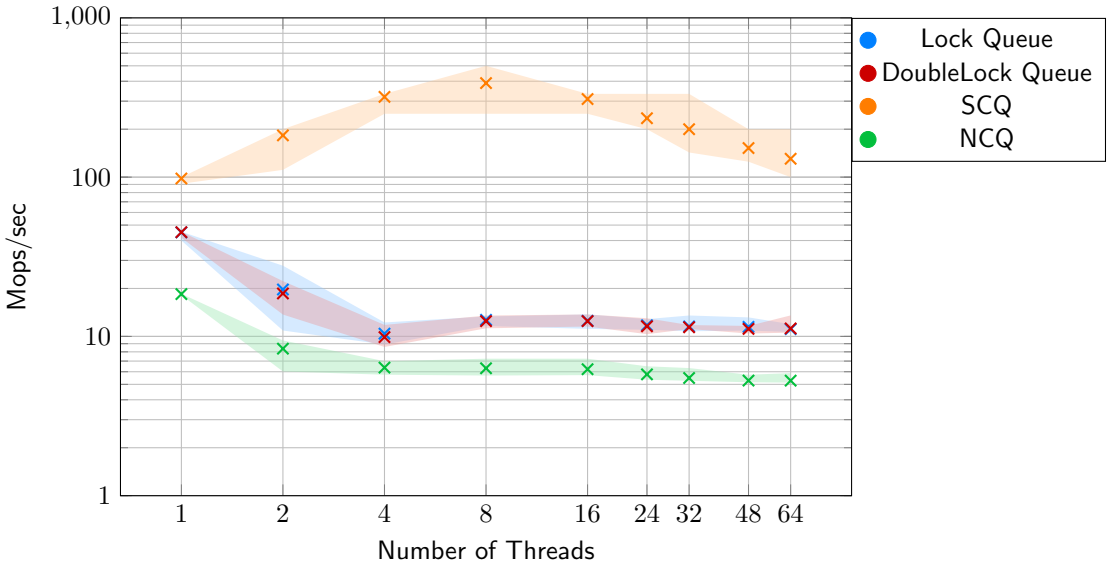

Figure 7: Measuring the performance with dequeue operations on empty queue.

3.3 Executing pairs of `enq()` and `deq()`

The goal of this measurement is to evaluate the performance of executing `enq()` with a subsequent `deq()` operation. As suggested in the paper [Nik19], this does not necessarily mean that this results in alternating calls of `enq()` and `deq()` on the queue due to the fact, that other threads can execute one of these two operations in between. Therefore, the order of operations is not predefined. The results are displayed in figure [8].
The naive circular performs worse in comparison to the other implementations. The results presented in the paper [Nik19] have already indicated a bad performance for the naive circular queue. For one thread, the lock based queues are slighlty faster than the scalable circular queue but as the number of threads increases, the scalable circular queue performs better. Another interesting aspect is the fact that the lock queue performs a little bit better than the double lock queue with large number of threads.

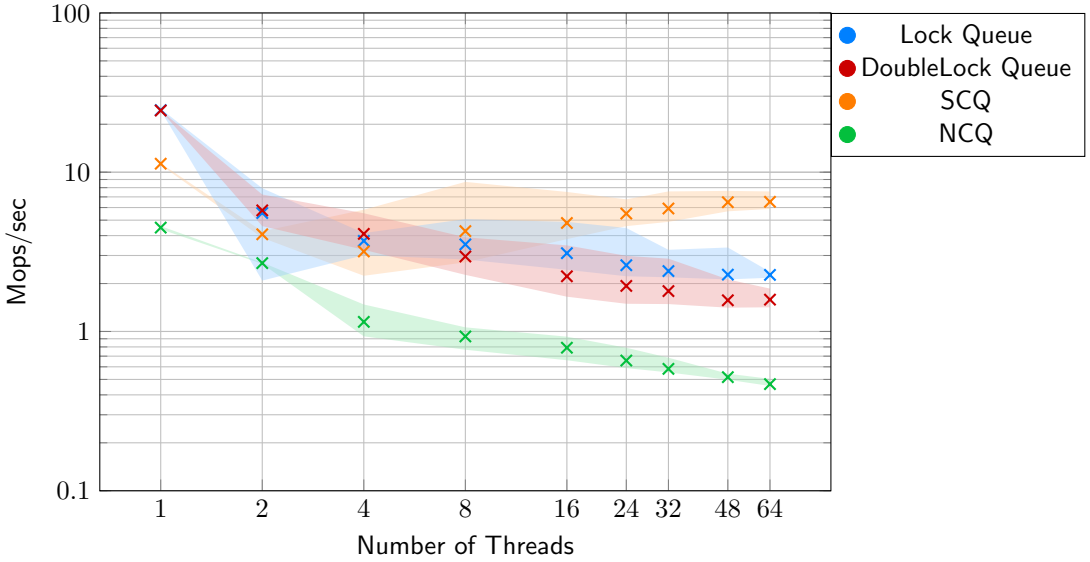

Figure 8: Measuring the performance of pairs of `enq()` and `deq()` operations

3.4 Random `enq()` and `deq()` operations

The final measurement to evaluate the implemented queues is based on a random execution of `enq()` and `deq()` operations. The selection is based on a 50% chance, therefore roughly the same amount of both operations are executed. As displayed in figure [9], the lock based queues perform better than the lock free approaches. The naive circular queue is still the loser of all implemented queues, but the performance of scalable and naive circular queue is quite similar. Overall, the double lock queue is the winner for this scenario, even if it is only a small advantage over the other queues.

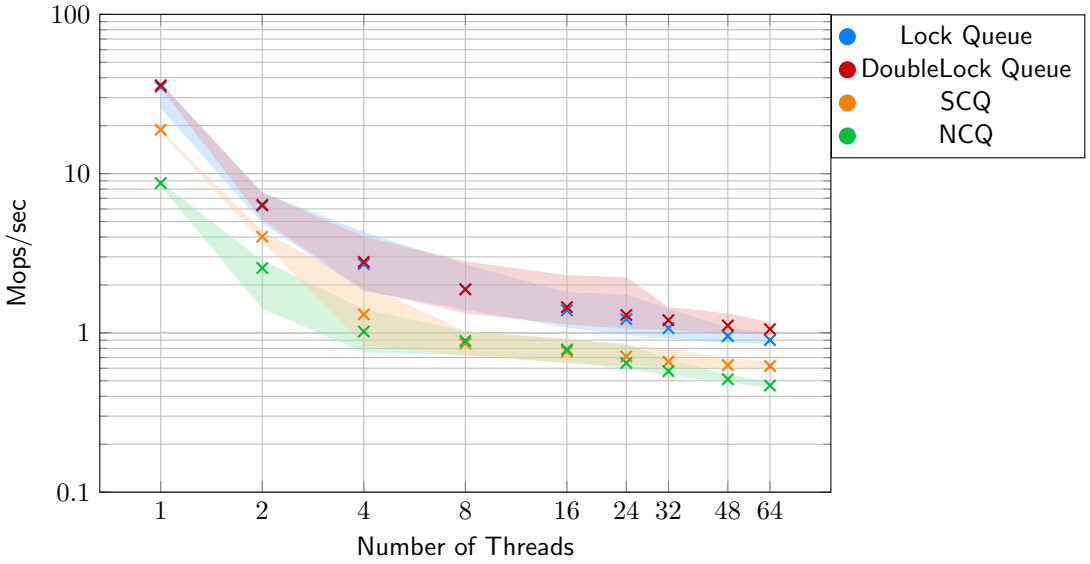

Figure 9: Measuring the performance of pairs of `enq()` and `deq()` operations

4 Summary

We implemented four different queues that follow two different main approaches. While the lock based queues follow a quite simple approach of simply using locks for their `enq()` and `deq()` operations, the lock free queues need a more advanced way of enabling concurrent execution of these operations.
On the one hand, this is the underlying data structure, which is used by both naive and scalable circular queue. On the other hand, both lock free queues can spin in order to complete their operations successfully (or fall below the threshold in case of the SCQ dequeue).
The scalable circular queue of course is a further development of the naive circular queue. Therefore it is not surprising, that the scalable circular queue always out performs the naive circular queue, at least in our benchmarks.
However, a winner between lock based and lock free queues can not be established easily, because both types have specific advantages. The fast path of the scalable circular queue clearly outperforms the other queues for `deq()` operations on empty queues. But the lock based approach performs better when `enq()` and `deq()` operations are executed in a random manner. Additionally, the paper mentions that there are environments where locking is prohibited anyway, in which case the SCQ offers a performant alternative regardless of what queue is the fastest. Lastly, performance also depends heavily on the amount of concurrent threads as well as the underlying computer architecture.
Therefore, the common case, environmental restrictions, and the level of concurrency must be kept in mind when deciding on which FIFO queue is the best fit for any particular use case.

References

[Doc] Documentation of Software and Hardware. https://doku.par.tuwien.ac.at/docs/machines/nebula/. Accessed on 19.06.2021.

[Nik19] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. aug 2019.

[Nik20] Ruslan Nikolaev. Github repository: lfqueue. https://github.com/rusnikola/lfqueue/, dec 2020. Commit 0bfbc34.