NBLFQ: a lock-free MPMC queue optimized for low contention

Alexandre Denis, Charles Goedefroit

To cite this version:

Alexandre Denis, Charles Goedefroit. NBLFQ: a lock-free MPMC queue optimized for low contention. IPDPS 2025 - 39th International Parallel & Distributed Processing Symposium, IEEE, Jun 2025, Milan, Italy. hal-04851700

HAL Id: hal-04851700
https://inria.hal.science/hal-04851700v1

Submitted on 20 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

NBLFQ: a lock-free MPMC queue optimized for low contention

Alexandre DENIS
Centre Inria de l'Université de Bordeaux, France
Alexandre.Denis@inria.fr

Charles GOEDEFROIT
Eviden, France
Charles.Goedefroit@eviden.com

***Abstract*—The NEWMADELEINE communication library relies extensively on lockless queues for its internal data structures in order to operate well in a multi-threaded application. There is ongoing work to make it use interrupt-based network drivers, and thus its queues have to become truly lock-free and not only lockless.**

In this paper we present NBLFQ, a lock-free MPMC bounded queue optimized for low contention. We show that 97 % of the queue operations in NEWMADELEINE are uncontended, hence the idea to design an algorithm optimized primarily for low contention, but that does not collapse under heavy load.

We present the algorithm that relies on a single CAS for enqueue and for dequeue, and proofs of its properties. We have implemented the algorithm, measured its performance on four different architectures, and compared it against a wide range of other lock-free queue algorithms. We have observed that the best performance at the network communication library level is obtained by our NBLFQ algorithm in almost all cases, with both single thread and massively multi-threaded communications.

I. INTRODUCTION

Queues are the foundation of numerous multi-threaded applications. The use of lock-free algorithms aims at having a smooth execution and good scalability. MPMC (multiple producer, multiple consumers) lock-free queues allow multiple threads to perform enqueue/dequeue operations at the same time.

The NEWMADELEINE [1] communication library is designed for HPC applications and behaves especially well in a multi-threaded context. To do so, it relies on lockless queues for a lot of its internal data structures. However, there is ongoing work to make it use interrupt-based drivers, and thus will need true lock-free queues.

In this paper, we present NBLFQ, a lock-free MPMC bounded queue that exhibits the following properties to make it efficient in NEWMADELEINE:

- it is completely non-blocking without even using dynamic memory allocation, so as to be usable in an interrupt handler;
- it is *optimized for low-contention* usage, since even with a high network load, the pressure imposed on the queues is not so high. This comes in contrast with most lock-free queues algorithm that target scalability.

The rest of this paper is organized as follows. Section II gives some insight on the context and the motivations of this work. Section III presents related work. In Section IV, we present our algorithm. In Section V, we present proofs of several key properties of our algorithm. In Section VI, we evaluate the performance of our algorithm in micro-benchmarks and inside NEWMADELEINE, and compare it with a large panel of competitors. We draw conclusions in Section VII.

II. MOTIVATIONS AND CONTEXT

A. *Queues in* NEWMADELEINE

NEWMADELEINE [1] is a communication library that provides its own native interface in addition to an MPI interface called MadMPI. The originality of NEWMADELEINE compared to other communication libraries and MPI implementations is that it applies a scheduling strategy on the flow of packets, and it decouples the network activity from the calls to the API by the user: all the activity is made of up-calls (event notifiers) triggered from the lowest layer. It behaves especially well with multi-threaded applications using `MPI_THREAD_MULTIPLE` communication mode.

NEWMADELEINE being fully multi-threaded, synchronization is needed when accessing its internal status. It has been shown [2], [3] that a loose synchronization should be preferred; thus NEWMADELEINE relies heavily on lockless queues for requests submission and completion, and for queues of deferred work to exchange tasks between threads or between locked and unlocked sections of the code. That way, it is not needed to acquire and release locks so frequently on the data critical path. Moreover, NEWMADELEINE relies on PIOMAN [4] for communication progression. PIOMAN schedules communications tasks to poll networking boards. In its core, lockless queues are the basis [5] of PIOMAN work queues. All in all, lockless queues are the foundation of NEWMADELEINE work flow.

B. *Lock-free queues in interrupt handlers*

a) Interrupt-safety: To reduce the load caused by active polling and busy waiting, there is ongoing work to use interrupt-based communication, either through Unix signals or user-level interrupts [6]. Therefore some code of NEWMADELEINE core will be called from interrupts/signal handlers. As a consequence, all internal data structures need to be *interrupt-safe* (async-safe) and not only *thread-safe*. We then need our queues to be interrupt-safe. A code that is run in the context of an interrupt handler indeed cannot wait: it is not allowed to call any system call that would cause the process to

become non-runnable. It is not allowed to use mutexes, which could potentially switch the process to a sleeping state. Even spinlocks are not allowed since they could cause a livelock in case the interrupted thread was precisely holding the lock. By transitivity, it is not allowed to call any library function that uses synchronization internally, such as memory allocation.

b) Lockless and lock-free queues: Before NBLFQ proposed in this paper, naive *lockless* queues were used since they behave well on low contention with minimal software overhead. The data structures are a ring buffer and two variables $head$ and $tail$ that keep track of head and tail in the ring buffer. To enqueue a new element, the head is incremented using a *compare-and-swap* (*CAS*) to reserve a cell, then the content is written in the buffer; to dequeue an element, the tail is consumed by incrementing the tail, then the element is read. There is a point where the operation is halfway: the counter has been updated, but not the data in the buffer. To accommodate with this fact, enqueue/dequeue may have to *wait* for the buffer to be updated by another thread so that it may itself complete an operation.

This implies that enqueue/dequeue are *blocking*. They are *lockless* in the sense that they do not involve any lock, but they may block on some occasions and wait for another thread to complete its operation.

In addition, an algorithm may be *non-blocking*, in the sense that *a pending invocation of a total method is never required to wait for another pending invocation to complete* [7]. A lockless algorithm that is *non-blocking* at the same time is said to be *lock-free*. A *lock-free* algorithm that completes in bounded time is called *wait-free*.

A *lockless* algorithm that may block is likely to cause a livelock if done in an interrupt handler. To use queues in an interrupt handler, it is needed that the algorithm is *non-blocking*, and thus we will use *lock-free* queues. We consider that *lock-free* is sufficient since we are in a case with *low contention* (see below), unlikely to cause starvation.

C. Low contention

For our use in a communication library, we have the intuition that the scalability of the queues is not an issue, the granularity of queues and network communications being very different. An operation on a queue is in the order of magnitude of tens of nanoseconds, but latency for high-performance networks such as InfiniBand is typically one microsecond. Thus, even when the network is saturated with a continuous flow of packets, it remains sparse from the point of view of the queues.

To verify this assumption, we have added profiling to our implementation of lockless queues to count the number of failed *CAS*. We have run a network micro-benchmark that performs a ping-pong between a varying number of threads. The results are shown in Figure 1. We observe that the number of *CAS* grows rapidly for a number of threads up to 10, then it remains roughly constant to about $3\,\%$. This test is only representative of a very high network workload; no real life HPC application that we are aware of performs a continuous stream of communications on all cores at the same time. The pressure imposed on the queues by real applications [8] is expected to be much lower.

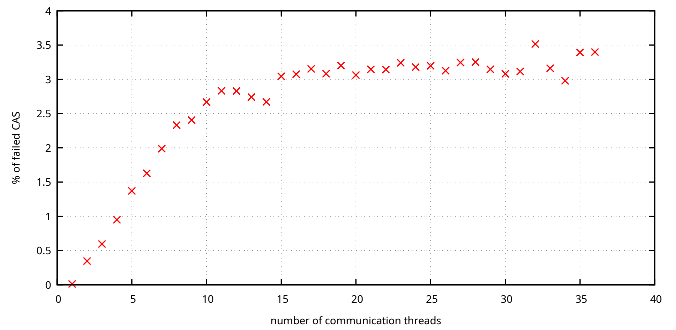

Fig. 1. Number of failed *CAS* with varying number of communication threads on machine `henri` with baseline queues

We conclude that, if even for a workload that is considered to be unrealistically high, the number of *CAS* that need to be restarted stays at $3\,\%$, the level of contention imposed by the communication library on the queues is low. Therefore we will focus on queue algorithms that ensure low latency for low numbers of threads.

III. Related work

In the domain of lock-free MPMC queues, a lot of prior work exists.

a) Lock-free queues: The foundation of several lock-free FIFO queues is Michael and Scott's msqueue [9], that uses a singly linked list. LOO [10], CRDoubleLinkQueue [11] and BitNextQueue [12] are more recent lock-free, unbounded, MPMC lock-free queues based on linked lists. These algorithms use dynamic memory allocation, which may not be interrupt-safe out of the box, but this issue may be addressed with a lock-free memory allocator [13].

There has been some work on array-based queues to increase performance, such as LinearArrayQueue [14] using an msqueue containing arrays, SCQ [15] relying on FAA instead of *CAS* for better performance, the lock-free queue by Tsigas & Zhang [16] that uses a ring buffer, and a lock-free queue by Krizhanovsky [17] that is not linearizable and thus suffers from the same shortcomings as our own naive lockless queue.

It has been proposed [18] to relax the semantics of queues to increase performance; however the relaxed semantics may render these queues unsuitable for use in some contexts.

b) Wait-free queues: Strictly speaking, lock-free queues are enough for our needs. We compare with wait-free queues anyway since he who can do more can do less.

Some well-known wait-free queues using dynamic memory allocation and some kind of memory reclamation (garbage collector, hazard pointers) are Kogan and Petrank queues [19], CRTurn [20], SimQueue [21], YMC wfqueue [22] by Chaoran Yang, and LCRQ [23]. Their use of dynamic memory allocation makes them harder to use in an interrupt handler.

Barrington et al. [24] proposed a wait-free ring buffer that uses dynamic memory allocation though. wCQ [25], derived from SCQ by Ruslan Nikolaev, uses a ring-buffer and only static memory allocation.

c) Alternative design: To share data structures with an interrupt handler, the classical solution is RCU [26] (*read, copy, update*). However, RCU is really relevant in kernel mode and exhibits poor user-space performance. Moreover, it imposes that all data structures access are designed for RCU, but in our case, the use of interruption is only an option.

IV. Proposed algorithm

In this section, we present NBLFQ, the lock-free queue algorithm that we propose. The design aims at low latency in low contention cases, without collapsing under heavier load. The naive *lockless* algorithm was (improperly) called *lfq* internally, thus the new non-blocking algorithm is called NBLFQ (even if it is a little bit redundant).

A. Overview

To deal with the memory constraint, our design is based on a *ring buffer*. It adds the constraint of bounded capacity, which is not an issue in the case of a communication library with flow control. To make the performance as high as possible in the uncontended case, our proposed design is minimalist with a single *CAS* on the critical path.

In our proposed design, we remove $head$ or $tail$ indices found in naive queues: the authoritative source for the state of the queue is the array itself that contains the ring buffer. When needed, head and tail values will be deduced from the state of the ring buffer itself. We keep non-authoritative $head$ and $tail$ variables as an optimization: they are used as starting point when computing the real location of head and tail from the ring buffer. The ring buffer is *always* consistent in memory, so that whenever an interrupt handler is fired, we are guaranteed that data is consistent.

In list-based queues, usually enqueue is done at the tail, and dequeue at the head, as in a waiting line. In circular buffers [27], the convention is often reversed: enqueue at the head and dequeue at the tail, like a snake moving on a circle. For our algorithm, we use the convention from circular buffers: enqueue at the head, and dequeue at the tail.

B. Data structures

Our queue is designed to store non-NIL pointers. With each pointer, we attach a *counter* that the algorithm uses to keep consistency, as commonly used to solve ABA problems. The ring buffer is stored in an array comprised of *pairs* $\langle ptr, counter\rangle$. Conversely, for u an element, $u_{counter}$ is the counter field of the pair and u_{ptr} the pointer field.

Since our data type is a pointer, we use *tagged pointers* to store both pointers and counters in the same 64 bits word. On *x86-64* architecture as well as on *aarch64*, pointers actually use only 48 bits out of the 64 bits, thus leaving 16 free bits (from Intel 64 Manual [28] section 3.3.7.1) usable to store our counter. This allows us to store the $\langle ptr, counter\rangle$ pair in a single 64 bits word.

Algorithm 1 Algorithm for *init* and helper function

```
 1: function INIT
 2:     head ← 0
 3:     tail ← 0
 4:     for i ← 0, S − 1 do
 5:         A[i] ← ⟨NIL, 0⟩
 6:     end for
 7: end function
 8: function PREV(i)    ▷ compute index of previous element
 9:     return (i + S − 1) mod S
10: end function
```

In some cases, it is expected that tagged pointers are not possible: other architectures than *x86-64* and *aarch64*, future architectures where pointers will maybe use all 64 bits, or for data other than pointers using itself all 64 bits. In this case, we use an alternate implementation that stores the pair in two consecutive words. It is possible as long as the architecture provides a double-word *CAS* (DWCAS); it is already available on *x86-64* (`cmpxchg16b`) as well as on modern ARM (double-width LL/SC), but not on RISC-V. In this paper, we mainly describe the version based on tagged pointers. We have implemented both flavors (tagged pointers and DWCAS) and will present results of both.

The data structures are as follows:

- A is the array containing the ring buffer. Each cell in the array contains a $\langle ptr, counter\rangle$ pair. A special value noted NIL is used to mark empty pointers. We call *empty cell* a pair whose pointer is NIL.
- S the size of the array. For simplicity of modulo computation, we assume S to be a power of 2. S must be strictly greater than 1.
- W is the number of iterations before the counter wraps around. In the case of tagged pointers $W = 2^{16}$ (for $\langle 48 + 16\rangle$ tagged pointers). In the case of DWCAS, we would have $W = 2^{64}$, but in practice we consider that it never wraps around (it would take several centuries even with an increment every nanosecond).
- $head$ and $tail$ variable are still used to track the index of head and tail of data in the ring buffer, but they are considered as a cache and are *not authoritative*.

The initial state of these data is given by algorithm 1.

C. Counters

The counters stored alongside data represent the number of times a given cell of the ring buffer has been consumed: counters are initialized to zero; when a value is enqueued, counter is not changed; when the value is dequeued, the counter is incremented by 1. Thus, in the general case, a cell contains the same counter value as its neighbors except for the *tail* and its left neighbor.

Common usage tells to increment the counter each time a cell is written to memory. We take advantage of the circular

Algorithm 2 Algorithm to compare cell sequence numbers

1: **function** COMP(i, u, j, v)
2: ▷ whether cell u (index i) is logically before v (index j)
3: **if** $u_{counter} = v_{counter}$ **then**
4: **return** $i < j$
5: **else**
6: **return** $((v_{counter} + W - u_{counter}) \bmod W) < \frac{W}{2}$
7: **end if**
8: **end function**

buffer and use the same counter value for the whole round instead of incrementing it for each cell, in order to reduce the chance of counter wraparound given that our counters are encoded on 16 bits; for a queue of size 1024 (typical of uses in NEWMADELEINE), wraparound occurs after 67 millions of enqueues/dequeues, which should be plenty enough, as discussed in section V-A.

Moreover, we increment it only on *dequeue* when writing NIL, not on *enqueue* when writing a value. This simplifies our use of the counter not only to mitigate the ABA problem, but in addition to find *head* and *tail*. This approach does not hinder ABA problem mitigation, since we never overwrite a value with another value without going through NIL in-between.

Counters are used to compute a logical sequence number for cells, which allows to check whether it is logically before or after another in the ring buffer without knowing *head* and *tail*. The algorithm used to compare cells is given as algorithm 2. It uses the content of the cells u and v to access their counters, and their respective index i and j. If the counters are the same, then the cells are part of the same round and thus their order is a simple comparison of their index; if counters are different, then we compare counters, taking into account counter wrapping. This logical sequence number is the same as s_i in the invariant in the proof in section V-A.

D. Enqueue

The algorithm for enqueue is given as algorithm 3. It is comprised of three sections: head chasing, computation of the counter, commit.

The head chasing is performed from line 4 to line 20. The goal of this section is to compute h, the index of the head — where the new element will be stored in the ring buffer — , and p, the content of the predecessor of the head. There are three possible outputs for this section: either we find an empty cell with a non-empty predecessor (head in the general case, line 7, as depicted in figure 2), either the list is empty and thus head and tail are the same (both cells are empty at the point where the counter steps up, line 11), or the list if full and we exit with an error (both cells are full where counter steps up, line 14). These values are purely indicative, since we have no memory synchronization. We use here the content of two unsynchronized memory accesses. We know that at one point this data has existed since each value is read atomically (using compiler intrinsics for atomic read), but we do not know whether it will still be the case later, nor whether

Algorithm 3 Algorithm for *enqueue*

1: **function** NBLFQ_ENQUEUE(e)
2: **loop**
3: $h \leftarrow head$ ▷ take head value from cache
4: **loop** ▷ chase head
5: $u \leftarrow A[h]$
6: $p \leftarrow A[\text{PREV}(h)]$
7: **if** $p_{ptr} \neq NIL \wedge u_{ptr} = NIL$ **then**
8: **break** ▷ empty cell with non-empty pred
9: **end if**
10: **if** $\neg\text{COMP}(\text{PREV}(h), p, h, u)$ **then** ▷ found step
11: **if** $p_{ptr} = NIL \wedge u_{ptr} = NIL$ **then**
12: **break** ▷ empty list
13: **end if**
14: **if** $p_{ptr} \neq NIL \wedge u_{ptr} \neq NIL$ **then**
15: $head \leftarrow h$
16: **return** 1 ▷ list full
17: **end if**
18: **end if**
19: $h \leftarrow (h + 1) \bmod S$
20: **end loop**
21: $c \leftarrow p_{counter}$ ▷ counter for new element
22: **if** $p_{ptr} = NIL$ **then** ▷ empty list
23: $c \leftarrow (p_{counter} + W - 1) \bmod W$
24: **end if**
25: **if** $h = 0$ **then** ▷ wrapping around
26: $c \leftarrow (c + 1) \bmod W$
27: **end if**
28: **if** CAS$(\&A[h], \langle NIL, c\rangle, \langle e, c\rangle)$ **then** ▷ commit
29: $head \leftarrow (h + 1) \bmod S$
30: **return** 0 ▷ success
31: **end if**
32: **end loop** ▷ commit failed; retry
33: **end function**

					h			
index	0	1	2	3	4	5	6	7
counter	43	43	42	42	42	42	42	42
data	∅	∅	•	•	∅	∅	∅	∅
				p	u			

$\varnothing = NIL$, $\bullet =$ non-NIL pointer.

Fig. 2. Expected state after head chasing in enqueue (general case: empty cell after non-empty). Here we have $h = 4$, $p = \langle ptr, 42\rangle$, $u = \langle NIL, 42\rangle$

both values have existed at the same time. Only the success of the *compare-and-swap* will tell whether these values were up-to-date. As an optimization, we initialize h with $head$, the value of the head saved by the previous enqueue operation. In case there was no other competing enqueue, we may hope the value of $head$ to be correct and the loop will exit immediately.

The computation of the expected value for the counter is performed from line 21 to line 27. This section computes the counter that should be stored alongside the data to enqueue. Strictly speaking, the value already there at the head will

not be changed. However, we cannot reliably get *consistent* information about two cells at the same time, so we only keep the content of the predecessor p and then we deduce the counter value of the head from the counter of its predecessor. We keep the value of the predecessor and not the head itself because the value of p is needed to ensure consistency; this property will be used in the proofs in section V. Usually, the counter for the head is the same as in its predecessor except if p is the last cell of the ring buffer and h is the first cell, or in case the queue is empty and thus p is the tail. Please note that it would not be possible to use double-word memory access to read both p and u atomically, since double-word memory access is atomic only if the pointer is aligned, which is true only once in two elements.

The commit phase, which actually performs the *compare-and-swap*, ranges from line 28 to line 31. This is the only place where the ring buffer is actually written. In case of success, we store and update value for the head in the head cache, so that hopefully the next enqueue will not have to loop for head chasing (without guarantee, depending on concurrent enqueues). In case of failure, the *CAS* included a full memory barrier, and thus we retry from scratch with up-to-date data.

E. Dequeue

The algorithm for dequeue is given as algorithm 4. It is comprised of two sections: tail chasing and commit.

The tail chasing is performed from line 3 to line 10. The goal of this section is to compute t, the index of the tail — where the element to be dequeued is stored in the ring buffer —, and u, the content of the tail. The tail is located by the definition of the counters: the tail is where the counter steps down, using function COMP from algorithm 2, on line 6. The resulting state is shown on figure 3. If the list is empty, we exit early (line 11). Like for the head chasing in the enqueue operation, values are purely indicative. We only expect t to be a candidate for the tail. Only the success of the *compare-and-swap* will tell whether it was actually the tail. The same optimization applies: we initialize t with $tail$, the value saved by the previous dequeue operation. In case there were no competing dequeues, $tail$ is hopefully the actual tail and the loop will exit immediately.

The commit phase ranges from line 14 to line 18. It increments the counter and performs the *CAS* that actually writes to the ring buffer. In case of success, we store the new tail in the tail cache for the next dequeue, and we return the dequeued value. In case of failure, the *CAS* included a full memory barrier, and thus we retry from scratch with up-to-date data.

V. PROOFS

In this section, we present proofs about properties of the proposed algorithm: correctness, and non-blocking.

Memory model: As a memory model, we assume *weak consistency*. It means that memory accesses may be reordered by the compiler and by the hardware, except across memory barrier. We assume the hardware is able to perform a *compare-and-swap* (*CAS*) or equivalent (LL/SC), and that the *CAS* has in addition the semantics of a memory barrier.

Algorithm 4 Algorithm for *dequeue*

```
 1: function NBLFQ_DEQUEUE
 2:     loop
 3:         t ← tail                      ▷ take tail value from cache
 4:         p ← A[PREV(t)]
 5:         u ← A[t]
 6:         while COMP(PREV(t), p, t, u) do        ▷ chase tail
 7:             t ← (t + 1) mod S
 8:             p ← u
 9:             u ← A[t]
10:         end while
11:         if p_ptr = NIL ∧ u_ptr = NIL then
12:             return NIL                          ▷ empty queue
13:         end if
14:         c ← (u_counter + 1) mod W               ▷ new counter
15:         if CAS(&A[t], u, ⟨NIL, c⟩) then
16:             tail = (t + 1) mod S
17:             return u_ptr
18:         end if
19:     end loop                                    ▷ retry
20: end function
```

			t					
index	0	1	2	3	4	5	6	7
counter	43	43	42	42	42	42	42	42
data	∅	∅	•	•	∅	∅	∅	∅
		p	u					

Fig. 3. Expected state after head chasing in dequeue (step down in the counter). Here we have $t = 2$, $p = \langle NIL, 43\rangle$, $u = \langle ptr, 42\rangle$

To program these operations, we use `gcc` built-in functions. For *CAS*, we use `__sync_bool_compare_and_swap()`; the documentation explicitly states that it is a full memory barrier at the same time.

A. Correctness

a) Invariant: Our data structures exhibit the following invariant properties:

1) we define the logical sequence number (modulo W) of an element in the cell i in the ring buffer:
$s_i = i + A[i]_{counter} \times S$
This sequence number is the same that is the basis of the comparison function defined in algorithm 2.
Counters satisfy the following property:
$\exists t \in [0; S-1] \mid \forall k \in [0; S-1] s_{(k+S-t) \bmod S} \equiv k + s_t \pmod W$
Intuitively, it means that there exists an index t starting from which the logical counter is the index of the cell offset by the sequence number of cell t, with wrapping of counters at W and wrapping of indexes at S. In practice, t is the tail.

2) all the non-empty cells of the ring buffer A are contiguous, starting at cell t with the t defined above, in the sense of a ring buffer: first and last cell are considered to be neighbors.

We consider our data to be in a *consistent* state if and only if it verifies both properties of this invariant.

b) ABA problem and counter wrapping: Counters ensure that in case the *CAS* succeeds, we are guaranteed that no enqueue/dequeue occured in between and would have written the same value again (ABA problem). There is however the case where a series of W full rewrites of the queue ($W \times S$ enqueues and dequeues) would have been performed by other threads between the beginning of an enqueue and its *CAS*, which would make the counter to wrap around. However, we consider it implausible that several millions of such enqueues/dequeues would occur while a single thread runs only a few instructions. It may be interrupted by an interrupt handler that should be short, or may be unscheduled by the system scheduler and wait for another time slice of some tens of milliseconds. In contrast, it would need at least a full second to perform several millions of enqueue/dequeue; a pause of more than one second between two schedules of a given thread is very implausible. If the user believes the workload is likely to cause thread interruptions of several seconds, then he should use the DWCAS variant which is immune to this issue since its counter would take centuries to wrap around.

c) Consistency of enqueue: For correctness, the head chasing (from line 4 to line 20 of algorithm 3) plays no role. It only gives h the index of a good candidate for the *head*, and p the content of the predecessor of h; we could use a random value for h as well, it would not affect data consistency, only performance.

When we reach the *CAS* on line 28, we have two cases: either p is not empty, or it is empty.

If p is not empty, then we try to insert the new element after p, at index h. If the *CAS* succeeds, by definition of the *CAS* we know that the cell at index h was actually $\langle NIL, s\rangle$, thus we know it was empty, its counter was left unchanged, and the counter is consistent as the right neighbor of p. The consistency of the counters implies that the element at index h has not been overwritten by enqueues and dequeues that would have left it empty, since any dequeue operation would have incremented the counter which would have broken the counter consistency and would have made the *CAS* fail. With this information, we can conclude that either p is still in place, and thus the new element at index h is contiguous to existing values, at their right, preserving property 2 of the invariant, or data has been dequeued up to p, but not more, thus we are enqueuing in an empty queue; since the last dequeued element was p, we know that $t = h$, thus storing the new element at index h preserves property 2 of the invariant too.

If p is empty, then we try to insert an element in a seemingly empty queue. To preserve consistency, we need to insert at the tail, the index t as defined in property 1 of the invariant. The way c the value for the counter is computed on line 23 in algorithm 3 reflects this case. If the *CAS* succeeds, then we have the information that: the cell was actually empty, the counter at index h is consistent with what we computed thus no enqueue/dequeue happened in-between (applying the same discussion about ABA and counter wrapping as above), and thus counter of p is still valid, else it would have needed series of enqueue/dequeue, which would have changed counter at h and would have been detected. Since counters at index h and in p are actually real values, then we know that $t = h$ thus writing the new element at index h preserves property 2 of the invariant.

About property 1 of the invariant, the *CAS* expects its value to be c and writes c again, thus in all cases, counters are not changed by the enqueue operation. We conclude that if the property is true before the enqueue, it is still true after.

We have shown here that if both properties of the invariant hold before the enqueue, they still hold after the enqueue, and thus the enqueue operation maintains data consistency.

d) Consistency of dequeue: For *dequeue*, the outcomes of the *tail chasing* (from line 3 to 10) are the index t, u a value read at index t, and p a value read for the predecessor of t at index $(t-1) \bmod S$. We have no clue whether values of p and u are actual nor whether they have existed at the same time in memory.

We first show that when the *CAS* succeeds, t is the actual value for the tail, as defined in the invariant. It materializes as a *down step* in the counters, modulo wrapping at W.

If *CAS* succeeds, we deduce that value of u is the actual value of cell at index t in memory at this precise time, by definition of the *CAS*. Then we may investigate the various cases for local variable p consistency with in-memory value of $A[(t-1) \bmod S]$:

- p cannot be *more recent* than u, since u is the *actual* value in the ring buffer at the time of *CAS*.
- p may be the actual value of data in memory. Since the *tail chasing* loop exit condition is precisely the detection of the down step of the tail, if p and u are consistent with memory, then t, the index of u is the actual value for the tail.
- local variable p may be outdated compared to memory. Since counters are incremented and never decremented (mod W), then we have:
  $p_{counter} \leq A[prev(t)]_{counter}$
  By the exit condition of the loop, we have:
  $u_{counter} < p_{counter}$
  By definition of the success of *CAS*, we have:
  $u = A[t]_{counter}$
  Then we deduce:
  $A[t]_{counter} < A[prev(t)]_{counter}$
  which, again, is the down step of the tail.

In all cases, t is an index where the in-memory value of the counters steps down. By property 1 of the invariant, the only place where it occurs is the tail, then the t found here in the algorithm is the same as the t of the invariant.

As a consequence, the *CAS* actually access the tail of the ring buffer. The outcomes are twofold:

- the algorithm increments the counter at index t by 1. In the new state, property 1 still holds, with an incremented t.
- the algorithm replaced the value at index t with NIL. Since it was the tail, non-empty cells are still contiguous and starting at the updated t, thus property 2 still holds.

We have shown here that if both properties of the invariant hold before the dequeue, they still hold after the dequeue, and thus the dequeue operation maintains consistency.

e) Data consistency: We have shown that both enqueue and dequeue maintain data consistency. Initial state, as initialized by algorithm 1, respects trivially the invariant too, then data consistency is maintained at all times.

Moreover, we can show that no data is lost: enqueue only writes cells that were previously NIL, as a property of the *CAS*; dequeue returns to the user all values of cells that it resets to NIL.

Additionally, we have already shown that non-empty cells are contiguous at all times, that enqueue writes data at the right of the contiguous data, and that dequeue consumes data at the left of the contiguous data, thus FIFO is preserved.

As a conclusion, the proposed algorithm respects the semantics of a queue, that maintains data consistency, and preserves data and FIFO order.

f) Linearizability: Informally, linearizability is the principle that *each method call should appear to take effect instantaneously at some moment between its invocation and response* [7].

We have shown that the value of variable $head$ and $tail$ plays no role in correctness, and that data stored in the ring buffer A is touched only once, by the *CAS*.

Therefore, the linearization point of methods *enqueue* and *dequeue* is the point where their *CAS* succeeds, which is the only point where global data is touched, and thus these methods are linearizable.

B. Non-blocking

Both *enqueue* and *dequeue* algorithms contain two loops: an outer loop to retry when the *CAS* fails, and an inner loop to chase head/tail. The exit condition of these loops depends on data in memory, and thus on the activity of other threads. Therefore, the algorithms are not *wait-free*: in case new threads appear and perform operations on the queue, *starvation* of individual threads is possible.

We will show here that the algorithms are *non-blocking*: system-wide, progress is guaranteed.

a) Dequeue tail chasing: For the tail chasing loop of the dequeue algorithm, the exit condition is true when a *down step* is found in the sequence number s_i, with s_i as defined in the invariant.

If the data read from memory is consistent, it is straightforward that, in an endless ring, the strictly increasing sequence cannot grow indefinitely and at the same time have its end join its beginning. There is a seam where the sequence number goes down.

In case of inconsistency caused by the weak memory model, even assuming the data read from memory is random, there is no combination where $s_i = i + A[i]_{counter} \times S$ could be always increasing when considered in an endless ring. The only solution for a non-decreasing sequence to still be non-decreasing when it wraps around is to be constant. However, for $S > 1$ there is no way for s_i to be constant.

In case data changes while we iterate in the loop because of competing memory writes, N competing operations will at most make the sequence advance of at most N, and thus the sequence will neither increase indefinitely.

b) Enqueue head chasing: For the head chasing loop of the enqueue algorithm, there are multiple exit conditions: either we find a non-empty cell whose right neighbor is empty, or we find a down step in the sequence number and at this place, we have two empty or two non-empty cells.

First, we are guaranteed that a down step in the sequence number will be found, in the same fashion as in the dequeue algorithm above. Once it is found, there are four cases:

- p and u are empty, then the list is empty, and we exit the loop;
- p and u are non-empty, then the list is full, and we exit the loop;
- p is non-empty and u is empty. This is an inconsistent state, since such combination is not expected at the tail, but if ever it occurs because of weak memory consistency, it is an exit condition of the loop nonetheless;
- p is empty and u is non-empty, then we are at the beginning of the contiguous block of data at the tail. We then iterate up to the end of the block, until we find an empty cell.

  If ever concurrent writes make the queue full in-between so that we never find an empty cell, we will at least find the tail again after S more iterations, then exit upon the p and u non-empty condition.

  Symmetrically, if concurrent dequeues make the queue empty in-between, we will exit the loop anyway since we keep the previous value of u as p for the next iteration, and thus the first non-empty element will appear as empty u, non-empty p, even if the in-memory value of p has actually become empty.

Without locks, some values may be outdated, some p, u couples may be inconsistent, but counters are always consistent with data of the same cell since memory access is atomic for aligned single-word. Thus when testing whether list is empty or full, we know that data at the tail is consistent with the counters that indicate that it is actually the tail.

c) Enqueue and dequeue CAS *retry:* Both enqueue and dequeue use a similar pattern to retry failed *CAS* with a loop. We try to prove here that with a given set of N threads performing enqueue/dequeue operations concurrently, progress is guaranteed to be made in at least one thread.

We take the point of view of one thread after a *CAS* fail. If the *CAS* fails, we retry the full operation from scratch with

up-to-date data since the *CAS* includes a full memory barrier. After this retry, there are two cases:

- we successfully find head/tail and the *CAS* succeeds. Then we have progress.
- the *CAS* fails again. If our *CAS* fails after a memory barrier, the only possibility is that head/tail chasing was again inconsistent because another thread changed data in memory in-between. However, we observe that there is no write to the ring buffer A except through *CAS*. We conclude that another thread had a *CAS* success, and thus in the global set of N threads, at least one thread did progress.

d) Conclusion: We have shown that, with a given set of threads performing an enqueue/dequeue concurrently, there exists at least one thread with all of its loops iterating a finite number of times, thus there is always system-wide progress. As a conclusion, the algorithm is non-blocking.

VI. Evaluation

In this section, we evaluate the performance of our algorithm. First we evaluate the performance using a producer/consumer micro-benchmark, then we measure its performance in a real world context in the NewMadeleine communication library.

a) Experimental testbed: Our experimental platform is comprised of the following machines:

- `henri`: the nodes are dual Intel Xeon 6140 (2×18 cores) equipped with a Mellanox ConnectX-4 InfiniBand EDR network board (MT27700).
- `william`: the nodes are dual Intel Xeon E5-2650 (2×8 cores) equipped with Mellanox ConnectX-3 InfiniBand FDR boards (MT27500).
- `adastra`: the nodes are dual AMD Epyc 9654 Genoa (2×96 cores) equipped with Cray Slingshot-11 200 Gb/s network boards.
- `pyxis`: the nodes are dual Cavium ThunderX2 99xx (2×32 cores) equipped with Mellanox ConnectX-6 InfiniBand EDR boards (MT28908).

Machine `henri` is typical of Intel-based servers. Machine `william` is representative of ancient hardware. Machine `adastra` is representative of AMD-based servers, and of servers with a high number of cores. Machine `pyxis` is typical of ARM-based servers.

b) Software: Our queues are implemented in Puk, a sub-module of NewMadeleine [29]. Our new algorithm is Nblfq in Puk, the DWCAS-variant is Nblfq 2, and the *baseline* is the lockless flavor of Puk base queues (called `atomic` internally in the code).

We have integrated other queue algorithms in Puk too so as to be able to perform direct comparisons. We use the reference implementation by the original authors for LOO [30], YMC wfqueue [31], wCQ and SCQ (through its SCQ2 variant that stores pointers using double-width *CAS* and SCQD that stores pointers with an indirection) [32] as well as NCQ, the naive queues used as baseline for SCQ [15], SimQueue through libconcurrent [33], and we use the implementation from ConcurrencyFreaks [34] for CRTurnQueue, KoganPetrank, LCRQ, MS Queue, CRDoubleLink, BitNextQueue, and LinearArrayQueue.

The LCRQ implementation we use does not support ARM processors, thus we have no result for it on machine `pyxis`. We use default configuration for all external code, except the following. For YMC wfqueue, we had to patch the original implementation to make it build on ARM with a recent compiler. We had to patch ConcurrencyFreaks code that had a maximum of 128 threads hardwired in the code; we increased it to 256 so as to make it run on machine `adastra` with 192 cores. For wCQ, we found a bug in the implementation and signaled it to the author who fixed it.

A. Micro-benchmark: producer/consumer

First we run a micro-benchmark of the queue itself. The benchmark follows a producer/consumer scheme, with N producer threads and N consumer threads, with a varying number of threads. When running on all cores, N is half the total number of cores of the machine: half of the cores are producers, and the other half are consumers. Each thread produces or consumes 64k entries.

The results for this producer/consumer micro-benchmarks on all four machines are represented in figures 4, 5, 6, and 7. Results are shown in millions of operations per second, thus the higher the better, and represent the average over the 64k entries. It must be noted that the graph scale is logarithmic.

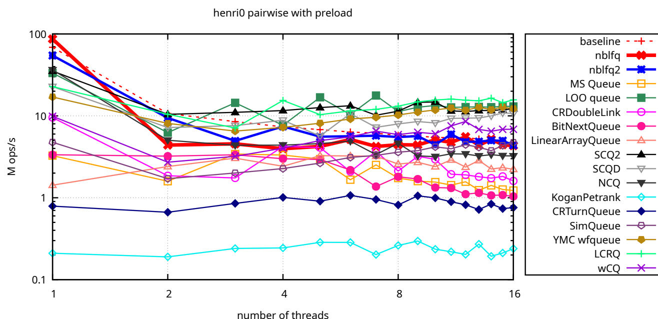

Fig. 4. Producer-consumer micro-benchmark on machine `henri`

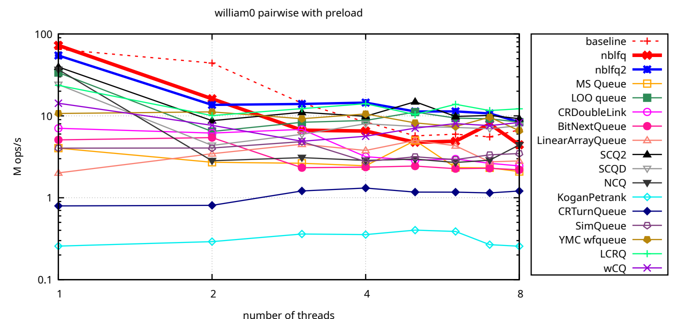

Fig. 5. Producer-consumer micro-benchmark on machine `william`

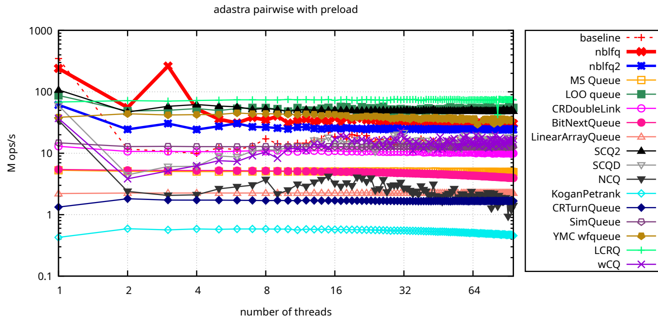

Fig. 6. Producer-consumer micro-benchmark on machine `adastra`

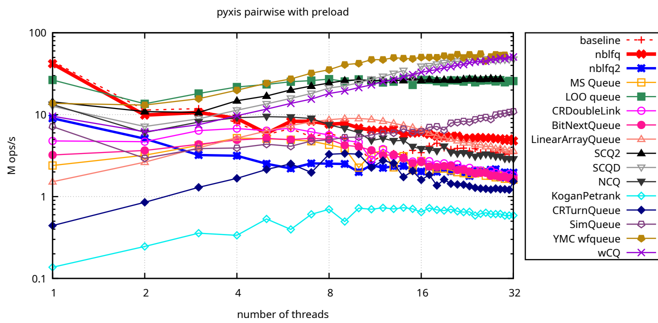

Fig. 7. Producer-consumer micro-benchmark on machine `pyxis`

We observe a large diversity of behaviors between queues, but with roughly the same hierarchy across machines. SCQ2, LCRQ, LOO and wfqueue are globally the best performing at scale. On ARM, wCQ is much faster than on x86. Regarding our NBLFQ, it is in the pack.

However, as stated in Section II-C, for our application we are in a case of *low contention*. Thus, the most interesting point on each graph is the result for a single producer and single consumer thread ($N = 1$). On all four machines, the best performing queue in this case is our NBLFQ, roughly *twice as fast* as the second best performing algorithm.

We note NBLFQ 2 is slightly slower than NBLFQ, except on machine `pyxis` (ARM) where the difference is higher.

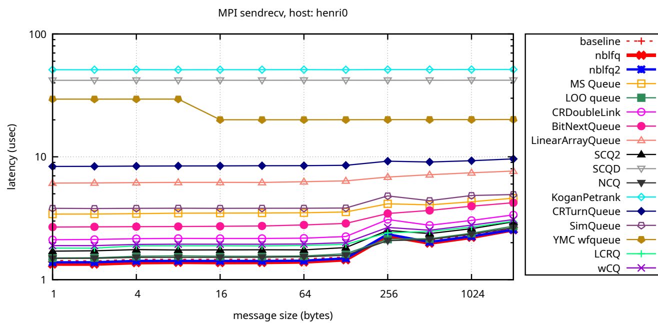

Fig. 8. Send/receive communication benchmark on machine `henri`

B. NEWMADELEINE *latency*

Then we run a network latency benchmark of NEWMADELEINE. Through PUK, the communication library NEWMADELEINE is able to use all these queue algorithms for its internal queues (submission queue, completion queue), as presented in section II. We have run a *single thread* latency benchmark. The goal of this benchmark is to measure the overhead introduced by each type of queue in the context of network communications.

The results for this latency benchmark on all four machines are represented in figures 8, 9, 10, and 11. Latency is measured in microseconds, thus the lower the better. We use the median value of 10000 roundtrips. Again, scale is logarithmic to make all results fit the same graph; some results are actually multiple orders of magnitude from each other.

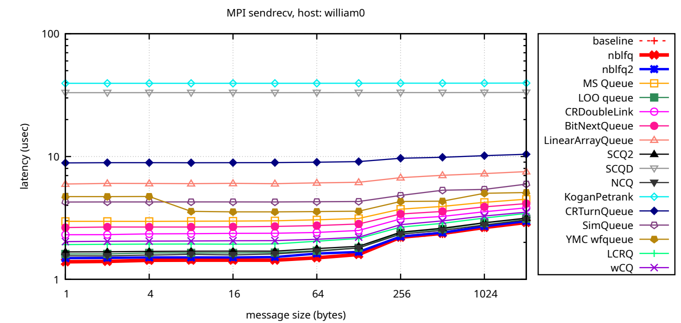

Fig. 9. Send/receive communication benchmark on machine `william`

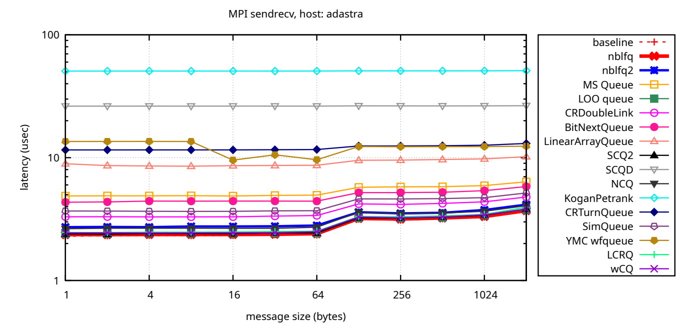

Fig. 10. Send/receive communication benchmark on machine `adastra`

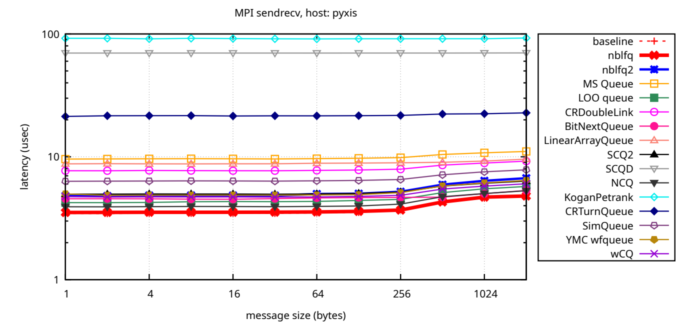

Fig. 11. Send/receive communication benchmark on machine `pyxis`

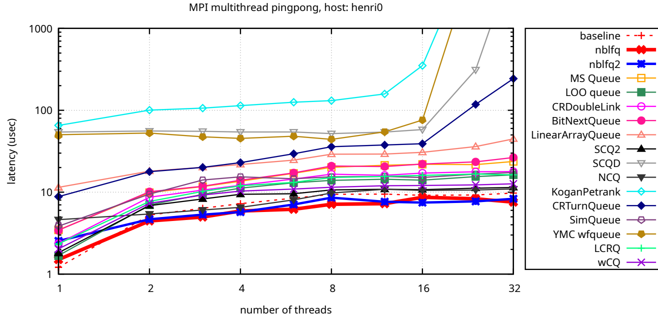

Fig. 12. Multi-threaded communication benchmark on machine `henri`

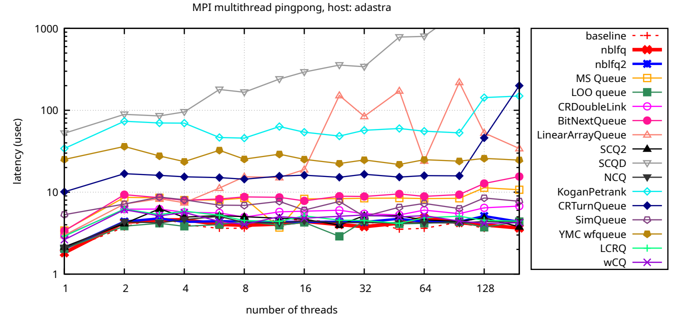

Fig. 14. Multi-threaded communication benchmark on machine `adastra`

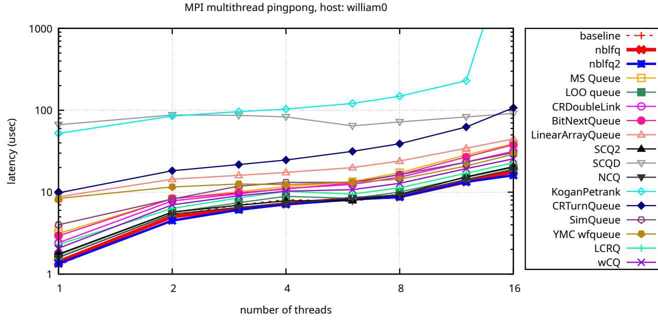

Fig. 13. Multi-threaded communication benchmark on machine `william`

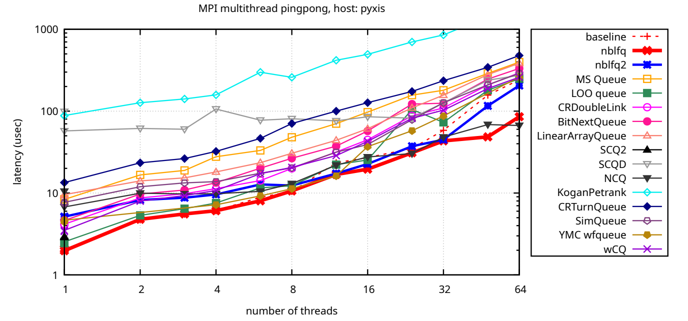

Fig. 15. Multi-threaded communication benchmark on machine `pyxis`

Again we observe a large diversity of behaviors between all algorithms. The base network latency with these kinds of high-speed networks is in the $1 - 3\,\mu s$ range. For some algorithms, the overhead caused by the queue multiplies the latency by a factor 2, 10, or even 25. Using these queues in a communication library would render useless the low latency network hardware. The fastest algorithms are roughly the same as for producer/consumer benchmark: NBLFQ, NBLFQ 2, LOO, SCQ2, LCRQ, with the addition of NCQ.

Some results are surprising. YMC wfqueue and SCQD are much slower than expected after the producer/consumer benchmark. NCQ is faster. These results are reproducible and consistent across machines. The performance of a lot of queues are almost identical on `adastra` (AMD) and on the contrary, NBLFQ wins by a wide margin on `pyxis` (ARM) and `william` (old Intel). As expected from the microbenchmark, NBLFQ 2 is slower on ARM than on *x86-64*.

Overall, the best performing algorithm in this single thread communication benchmark is our NBLFQ, which gets the best latency on all four machines. Its performance is very close to the performance of the baseline lockless algorithm, which was our goal.

The second best performing algorithm causes a latency penalty ranging from $5\,\%$ on `adastra` to $16\,\%$ on `william`, which is still a large impact for the algorithm used for queues of requests in a communication library.

C. NEWMADELEINE *with threads*

The last benchmark is a multi-threaded ping-pong through the network with NEWMADELEINE. For this benchmark, N threads on one node are doing a ping-pong to N threads on another node through the network. The payload is only one byte so as to maximize the number of exchanged packets. When using one thread per core, this is the most network intensive workload. This is the same benchmark as used in section II-C. The parallel workload imposed by this benchmark on the network library is higher than any real world application that we run usually [8] on MPI libraries.

The results for this multi-threaded network benchmark on all four machines are represented in Figures 12, 13, 14, and 15. The measure is a latency in microseconds, thus the lower the better. The value represented on the graph is the median of 1000 roundtrips. It should be noted that the 1-thread latency in the multi-threaded benchmark here may be different than the single thread latency in the previous benchmark, since we use a single thread build in one case and a multi-threaded build in the other case, so the code path is different. It must be noted that on machine `pyxis` (ARM), the build with algorithm SCQ2 did not run with more than one thread, thus the graph is missing, in addition to LCRQ which does not build on ARM.

On machine `henri`, our NBLFQ is a clear winner with the best latency for any number of threads. On machine `william` and `pyxis`, others are closer and may take a very slight advantage at some specific points. On machine `adastra`, graphs for NBLFQ, NBLFQ 2, LOO, SCQ2, NCQ and LCRQ

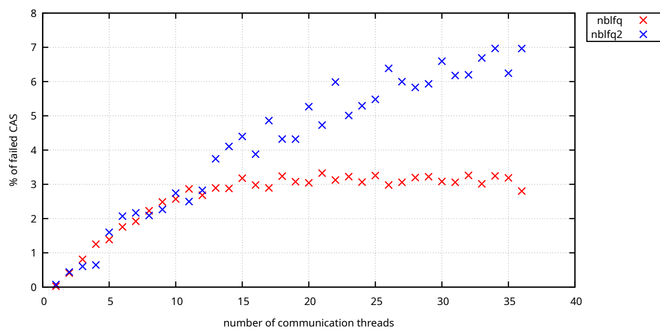

Fig. 16. Number of failed *CAS* with varying number of communication threads on machine `henri` with NBLFQ

are intertwined and thus their results are very similar. Again, NBLFQ 2 gets similar performance as NBLFQ, except on machine `pyxis` (ARM) where it is slightly slower.

Overall on all four machines, the best latencies are obtained with our NBLFQ.

D. Low-contention hypothesis

The basic hypothesis that our algorithm relies on is that in our communication library we use lock-free queues in a *low contention* context. We checked this hypothesis for the baseline queues in Section II-C. Here we check that this assumption still holds for the final NBLFQ algorithm.

We ran the same benchmark and measured *CAS* failures in the case of NBLFQ. The results are shown in Figure 16. For NBLFQ, the graph exhibits the same shape as in Figure 1, with all points below 3.5 % of *CAS* that actually need retry. For NBLFQ 2, the rate is higher, up to 7 %, which is still low.

This confirms the hypothesis that even when the communication library is at full throttle, the pressure imposed on the queue is still low and that what counts in this real world usage is the performance in the context of low contention.

VII. Conclusion

In the NEWMADELEINE communication library, lockless queues are widely used as submission list, deferred tasks list, completion list, to support multi-threaded communications. There is ongoing work to implement interrupt-based communication, that needs plain lock-free queues.

In this paper, we have presented NBLFQ, a lock-free queue that exhibits very high performance in uncontended cases. We have shown that the uncontended case is which occurs in 97 % of queue operations in NEWMADELEINE. We have described the algorithm, proven its correctness, and evaluated its performance in micro-benchmarks as well as in the communication library, and compared the performance of 17 queue algorithms. All in all, our proposed algorithm exhibits the best performance when used in the NEWMADELEINE communication library, even under heavy multi-threaded load.

As future works, we will work on using user-space interrupts for network communications.

Acknowledgment

This work was granted access to the HPC resources of CINES under the allocation 2024-A0160601567 attributed by GENCI (Grand Equipement National de Calcul Intensif).

Some experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Authors thanks Ruslan Nikolaev for his help to use wCQ and his quick bug fix.

References

[1] O. Aumage, E. Brunet, N. Furmento, and R. Namyst, "NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks," in *Workshop on Communication Architecture for Clusters (CAC 2007)*, Long Beach, California, United States, Mar. 2007. [Online]. Available: https://hal.inria.fr/inria-00127356

[2] F. Trahay, E. Brunet, and A. Denis, "An analysis of the impact of multi-threading on communication performance," in *Communication Architecture for Clusters*, Rome, Italy, May 2009. [Online]. Available: https://inria.hal.science/inria-00381670

[3] A. Denis, "Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests," in *CCGrid 2019 - 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing*, Larnaca, Cyprus, May 2019. [Online]. Available: https://hal.inria.fr/hal-02103700

[4] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *IEEE International Conference on Cluster Computing*. New Orleans, LA, United States: IEEE Computer Society Press, Aug. 2009. [Online]. Available: https://inria.hal.science/inria-00408521

[5] A. Denis, "pioman: a pthread-based Multithreaded Communication Engine," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, Mar. 2015. [Online]. Available: https://hal.inria.fr/hal-01087775

[6] S. Mehta, "User interrupts – a faster way to signal," in *Linux Plumbers Conference*, Sep. 2021.

[7] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufman, 2012.

[8] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 427–436.

[9] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731598914460

[10] O. Giersch and J. Nolte, "Fast and portable concurrent fifo queues with deterministic memory reclamation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 604–616, 2022.

[11] P. Ramalhete and A. Correia, "Doublelink - a low-overhead lock-free queue," http://concurrencyfreaks.blogspot.com/2017/01/doublelink-low-overhead-lock-free-queue.html, 2016.

[12] ——, "Bitnext - a lock-free queue," Feb. 2017. [Online]. Available: http://concurrencyfreaks.blogspot.com/2017/02/

[13] M. M. Michael, "Scalable lock-free dynamic memory allocation," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 35–46. [Online]. Available: https://doi.org/10.1145/996841.996848

[14] P. Ramalhete and A. Correia, "Lineararrayqueue - MPMC lock-free queue," 2016. [Online]. Available: http://concurrencyfreaks.blogspot.com/2016/11/lineararrayqueue-mpmc-lock-free-queue.html

[15] R. Nikolaev, "A scalable, portable, and memory-efficient lock-free fifo queue," in *International Symposium on Distributed Computing*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:199552164

[16] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 134–143. [Online]. Available: https://doi.org/10.1145/378580.378611
[17] A. Krizhanovsky, "Lock-free multi-producer multi-consumer queue on ring buffer," *Linux J.*, vol. 2013, no. 228, apr 2013.
[18] G. Kappes and S. V. Anastasiadis, "A family of relaxed concurrent queues for low-latency operations and item transfers," *ACM Trans. Parallel Comput.*, vol. 9, no. 4, Dec. 2022. [Online]. Available: https://doi.org/10.1145/3565514
[19] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 223–234. [Online]. Available: https://doi.org/10.1145/1941553.1941585
[20] P. Ramalhete and A. Correia, "Poster: A wait-free queue with wait-free memory reclamation," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 453–454. [Online]. Available: https://doi.org/10.1145/3018743.3019022
[21] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 325–334. [Online]. Available: https://doi.org/10.1145/1989493.1989549
[22] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851168
[23] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 103–112. [Online]. Available: https://doi.org/10.1145/2442516.2442527
[24] A. Barrington, S. Feldman, and D. Dechev, "A scalable multi-producer multi-consumer wait-free ring buffer," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1321–1328. [Online]. Available: https://doi.org/10.1145/2695664.2695924
[25] R. Nikolaev and B. Ravindran, "Wcq: A fast wait-free queue with bounded memory usage," in *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 307–319. [Online]. Available: https://doi.org/10.1145/3490148.3538572
[26] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," in *AUUG Conference Proceedings*, vol. 175. AUUG, Inc., 2001.
[27] D. Howells and P. E. McKenney, "Circular buffers." [Online]. Available: https://www.kernel.org/doc/Documentation/circular-buffers.txt
[28] I. Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Sep. 2016.
[29] The PM2 team, "Pm2: Parallel multithread machine." [Online]. Available: https://gitlab.inria.fr/pm2/pm2
[30] O. Giersch, "looqueue." [Online]. Available: https://github.com/oliver-giersch/looqueue
[31] C. Yang, "Fast wait free queue." [Online]. Available: https://github.com/chaoran/fast-wait-free-queue
[32] R. Nikolaev, "wCQ: A fast wait-free queue with bounded memory usage," https://github.com/rusnikola/wfqueue.
[33] N. Kallimanis, "The sync framework." [Online]. Available: https://github.com/nkallima/sim-universal-construction
[34] P. Ramalhete and A. Correia, "Concurrencyfreaks." [Online]. Available: https://github.com/pramalhe/ConcurrencyFreaks/tree/master/CPP/queues