VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

SPECIALIZED PROJECT

STUDYING AND DEVELOPING NONBLOCKING DISTRIBUTED MPSC QUEUES

Major: Computer Science

THESIS COMMITTEE: 0
MEMBER SECRETARY:
SUPERVISORS: THOẠI NAM
DIỆP THANH ĐĂNG

—ooo—

STUDENTS: ĐỖ NGUYỄN AN HUY - 2110193

HCMC, 03/2025

Disclaimers

I affirm that this specialized project is the product of my original research and experimentation. Any references, resources, results which this project is based on or a derivative work of have been given due citations and properly listed in the footnotes and the references section. All original contents presented are the culmination of my dedication and perserverance under the close guidance of my supervisors, Mr. Thoại Nam and Mr. Diệp Thanh Đăng, from the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology. I take full responsibility for the accuracy and authenticity of this document. Any misinformation, copyright infrigment or plagiarism shall be faced with serious punishment.

Acknowledgements

This thesis is the culmination of joint efforts coming from not only myself, but also my professors, my family, my friends and other teachers of Ho Chi Minh University of Technology.

I want to first acknowledge my university, Ho Chi Minh University of Tecnology. Throughout my four years of pursuing education here, I have built a strong theoretical foundation and earned various practical experiences. These all lend themselves well to the completion of this thesis. Especially, I want to extend my gratitude towards my supervisors, Mr.Thoại Nam and Mr. Diệp Thanh Đăng, who have acted as constant counselors and advisors right from the project's inception throughout. They have provided guidance on the project's direction and laid the academic basis for this project, upon which my work is essentially built upon. Furthermore, they inspired me to work hard and push through all the technical obstacles. Without them, this project wouldn't have reached this point of creation.

I also want to give my family the sincerest thanks for their emotional and financial support, without which I couldn't have whole-heartedly followed my research till the end.

Last but not least important, I want to thank my closest friends for their informal but ever-constant check-ups to make sure I didn't miss the timeline for this specialized project, which I usually don't have the mental capacity for.

Contents

List of Tables

List of Images

Chapter I Introduction

The demand for computation power has always been increasing relentlessly. Increasingly complex computation problems arise and accordingly more computation power is required to solve them. Much engineering efforts have been put forth towards obtaining more computation power. A popular topic in this regard is distributed computing: The combined power of clusters of commodity hardware can surpass that of a single powerful machine. To fully take advantage of the potential of distributed computing, specialized algorithms and data structures need to be devised.

Noticeably, multi-producer single-consumer (MPSC) is one of those data structures that are utilized heavily in distributed computing, forming the backbone of many applications. Consequently, an MPSC can easily present a performance bottleneck if not designed properly. A desirable distributed MPSC should be able to exploit the highly concurrent nature of distributed computing. Currently, in the literature, most distributed data structures are designed from the ground up, completely disregarding the any existing data structures developed in the shared memory area, e.g. [2]. This is partly due to the historical differences between the programming models utilized in these two areas. However, since the introduction of specialized networking hardware RDMA and the improved support of the remote memory access (RMA) programming model in MPI-3, this gap has been bridged. Thus, it has opened up a lot new research ([3]) on reusing the principles in the shared memory literature to distributed computing. One favorable characteristic of concurrent data structures that has been heavily researched in the shared memory literature, which is also equally important in distributed computing, is the property of non-blocking, or in particular, lock-freedom. Lock-freedom guarantees that if some processes suspend or die, other processes can still complete. This provides both progress guarantee and fault-tolerance, especially in distributed computing where nodes can fail any time. Thus, the rest of this document concerns itself with investigating and devising efficient non-blocking distributed MPSCs. Interestingly, we choose to adapt current MPSC algorithms in the shared-memory literature to distributed context, which enables a wealth of accumulated knowledge in this literature.

1.1 Motivation

Lock-free MPSC and other FIFO variants, such as multi-producer multi-consumer (MPMC), concurrent single-producer single-consumer (SPSC), are heavily studied in the shared memory literature, dating back from the 1980s-1990s [4], [5], [6] and more recently [7], [8]. It comes as no surprise that algorithms in this domain are highly developed and optimized for performance and scalability. However, most research about MPSC or FIFO algorithms in general completely disregard the available state-of-the-art algorithms in the shared memory literature. With the new RDMA networking hardware support and capabilities added to MPI-3 RMA API: lock-free shared-memory algorithms can be straightforwardly ported to distributed context using this programming model. This presents an opportunity to make use of the highly accumulated research in the shared memory literature, which if adapted and mapped properly to the distributed context, may produce comparable results to algorithms exclusively devised within the distributed computing domain. Therefore, we decide to take this novel route to developing new non-blocking MPSC algorithms: Port and adapt potential lock-free shared-memory MSPCs to distributed context using the MPI-3 RMA programming model. If this approach proves to be effective, a huge intellectual reuse of shared-memory MSPC algorithms into the distributed domain is possible. Consequently, there may be no need to develop distributed MPSC algorithms from the ground up.

1.2 Objective

This thesis aims to:

- Investigate state-of-the-art shared-memory MPSCs.
- Select and appropriately modify potential MPSC algorithms so they can be implemented in popular distributed programming environments.
- Port MPSC algorithms using MPI-3 RMA.
- Evaluate various theoretical aspects of ported MPSC algorithms: Correctness, progress guarantee, time complexity analysis.
- Benchmark the ported MPSC algorithms and compare them with current distributed MPSCs in the literature.
- Discover distributed-environment-specific optimization opportunities for ported MPSC algorithms.

1.3 Scope

- For related works on shared-memory MPSCs, we only focus on linearizable MPSCs that support at least lock-free `enqueue` and `dequeue` operations.
- Any implementation details, benchmarking and optimizations assume MPI-3 settings.
- For optimizations, we focus on performance-related metrics, e.g. time-complexity (theoretically), throughput (empirically).

1.4 Structure

The rest of this report is structured as follows:

Chapter II discusses the theoretical foundation this thesis is based on and the technical terminology that's heavily utilized in this domain. As mentioned, this thesis investigates state-of-the-art shared-memory MPSCs. Therefore, we discuss the theory related to the design of concurrent algorithms such as lock-freedom and linearizability, the practical challenges such as the ABA problem and safe memory reclamation problem. We then explore the utilities offered by C++11 to implement concurrent algorithms and MPI-3 to port shared memory algorithms.

Chapter III surveys the shared-memory literature for state-of-the-art queue algorithms, specifically MPSC and SPSC algorithms (as SPSC can be modified to implement

MPSC). We specifically focus on algorithms that have the potential to be ported efficiently to distributed context, such as NUMA-aware or can be made to be NUMA-aware. We then conclude with a comparison of the most potential shared-memory queue algorithms.

Chapter IV documents distributed-versions of potential shared-memory MPSC algorithms surveys in Chapter III. It specifically presents our adaptation efforts of existing algorithms in the shared-memory literature to make their distributed implementations feasible.

Chapter V discusses various interesting theoretical aspects of our distributed MPSC algorithms in Chapter IV, specifically correctness (linearizability), progress guarantee (lock-freedom and wait-freedom), performance model. Our analysis of performance model helps back our empirical findings in Chapter VI, together, they work hand-in-hand to help us discover optimization opportunities.

Chapter VI introduces our benchmarking setup, including metrics, environments, benchmark/microbenchmark suites and conducting methods. We aim to demonstrate some preliminary results on how well ported shared-memory MPSCs can compare to existing distributed MPSCs. Finally, we discuss important factors that affect the runtime properties distributed MPSC algorithm, which have partly been explained by our theoretical analysis in Chapter V.

Chapter VII concludes what we have accomplished in this thesis and considers future possible improvements to our research.

Chapter II Background

2.1 Irregular applications

Irregular applications are a class of programs particularly interesting in distributed computing. They are characterized by:

- Unpredictable memory access: Before the program is actually run, we cannot know which data it will need to access. We can only know that at run time.
- Data-dependent control flow: The decision of what to do next (such as which data tp accessed next) is highly dependent on the values of the data already accessed. Hence the unpredictable memory access property because we cannot statically analyze the program to know which data it will access. The control flow is inherently engraved in the data, which is not known until runtime.

Irregular applications are interesting because they demand special treatments to achieve high performance. One specific challenge is that this type of applications is hard to model in traditional MPI APIs. The introduction of MPI RMA (remote memory access) in MPI-2 and its improvement in MPI-3 has significantly improved MPI's capability to express irregular applications comfortably.

2.2 Multiple-producer, single-consumer (MPSC)

Multiple-producer, single-consumer (MPSC) is a specialized concurrent first-in first-out (FIFO) data structure. A FIFO is a container data structure where items can be inserted into or taken out of, with the constraint that the items that are inserted earlier are taken out of earlier. Hence, it's also known as the queue data structure. The process that performs item insertion into the FIFO is called the producer and the process that performs items deletion (and retrieval) is called the consumer. In concurrent queues, multiple producers and consumers can run in parallel. Concurrent queues have many important applications, namely event handling, scheduling, etc. One class of concurrent FIFOs is MPSC, where one consumer may run in parallel with multiple producers. The reasons we're interested in MPSCs instead of the more general multiple-producer, multiple-consumer data structures (MPMCs) are that (1) high-performance and high-scalability MPSCs are much simpler to design than MPMCs while (2) MPSCs are noticeably as powerful as MPMCs - its consensus number equals the number of producers [9]. Thus, MPSCs can see as many use cases as MPMCs while being easily scalable and performant.

2.3 Progress guarantee

Many concurrent algorithms are based on locks to create mutual exclusion, in which only some processes that have acquired the locks are able to act, while the others have to wait. While lock-based algorithms are simple to read, write and verify, these algorithms are said to be blocking: One slow process may slow down the other faster processes, for example, if the slow process successfully acquires a lock and then the operating system (OS) decides to suspends it to schedule another one, this means until the process is awaken, the other processes that contend for the lock cannot continue. Lock-based algorithms introduces many problems such as:

- Deadlock: There's a circular lock-wait dependencies among the processes, effectively prevent any processes from making progress.
- Convoy effect: One long process holding the lock will block other shorter processes contending for the lock.
- Priority inversion: A higher-priority process effectively has very low priority because it has to wait for another low priority process.

Furthermore, if a process that holds the lock dies, this will halt the whole program. This consideration holds even more weight in distributed computing because of a lot more failure modes, such as network failures, node falures, etc.

Therefore, while lock-based algorithms are easy to write, they do not provide **progress guarantee** because **deadlock** or **livelock** can occur and its use of mutual exclusion is unnecessarily restrictive. These algorithms are said to be **blocking**. An algorithm is said to be **non-blocking** if a failure or slow-down in one process cannot cause the failure or slow-down in another process. Lock-free and wait-free algorithms are to especially interesting subclasses of non-blocking algorithms. Unlike lock-based algorithms, they provide **progress guarantee**.

2.3.1 Lock-free algorithms

Lock-free algorithms provide the following guarantee: Even if some processes are suspended, the remaining processes are ensured to make global progress and complete in bounded time. This property is invaluable in distributed computing, one dead or suspended process will not block the whole program, providing fault-tolerance. Designing lock-free algorithms requires careful use of atomic instructions, such as Fetch-and-add (FAA), Compare-and-swap (CAS), etc.

2.3.2 Wait-free algorithms

Wait-freedom is a stronger progress guarantee than lock-freedom. While lock-freedom ensures that at least one of the alive processes will make progress, wait-freedom guarantees that any alive processes will finish in bounded time. Wait-freedom is useful to have because it prevents starvation. Lock-freedom still allows the possibility of one process having to wait for another indefinitely, as long as some still makes progress.

2.4 Correctness - Linearizability

Correctness of concurrent algorithms is hard to defined, especially when it comes to the semantics of concurrent data structures like MPSC. One effort to formalize the correctness of concurrent data structures is the definition of **linearizability**. A method call on the FIFO can be visualized as an interval spanning two points in time. The starting point is called the **invocation event** and the ending point is called the **response event**. **Linearizability** informally states that each method call should appear to take effect instantaneously at some moment between its invocation event and response event [10].

The moment the method call takes effect is termed the **linearization point**. Specifically, suppose the followings:

- We have n concurrent method calls $m_1, m_2, \ldots, m_n$.
- Each method call m_i starts with the **invocation event** happening at timestamp s_i and ends with the **response event** happening at timestamp e_i. We have $s_i < e_i$ for all $1 \leq i \leq n$.
- Each method call m_i has the **linearization point** happening at timestamp l_i, so that $s_i \leq l_i \leq e_i$.

Then, linerizability means that if we have $l_1 < l_2 < \ldots < l_n$, the effect of these n concurrent method calls $m_1, m_2, \ldots, m_n$ must be equivalent to calling $m_1, m_2, \ldots, m_n$ **sequentially**, one after the other in that order.

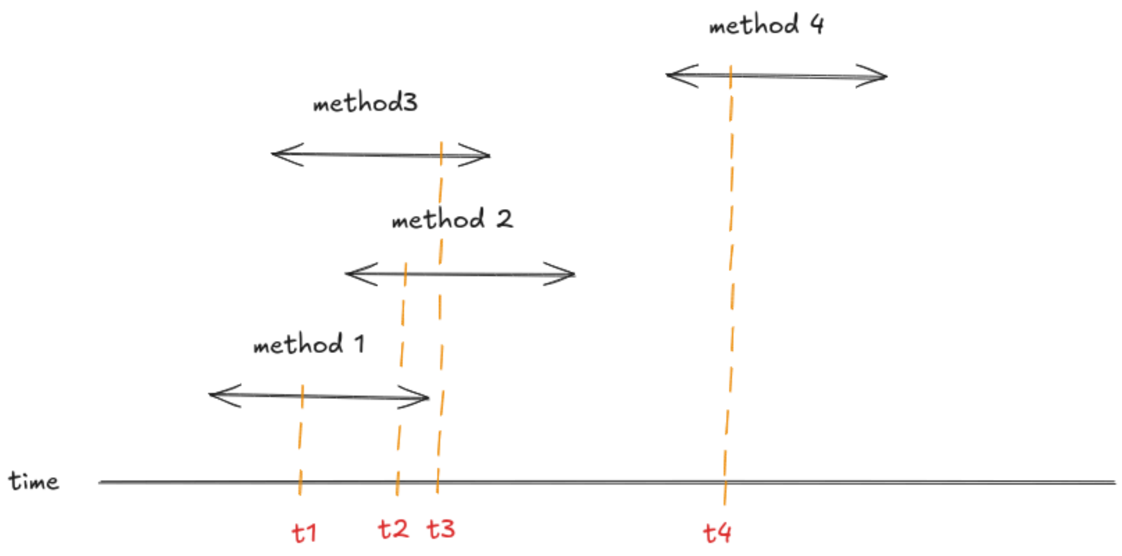

Figure 1: Linerization points of method 1, method 2, method 3, method 4 happens at $t_1 < t_2 < t_3 < t_4$, therefore, their effects will be observed in this order as if we call method 1, method 2, method 3, method 4 sequentially

2.5 Common issues when designing lock-free algorithms

2.5.1 ABA problem

In implementing concurrent lock-free algorithms, hardware atomic instructions are utilized to achieve linearizability. The most popular atomic operation instruction is compare-and-swap (CAS). The reason for its popularity is (1) CAS is a **universal atomic instruction** - it has the **concensus number** of ∞ - which means it's the most powerful atomic instruction [11] (2) CAS is implemented in most hardware (3) some concurrent lock-free data structures such as MPSC can only be implemented using powerful atomic instructions such as CAS. The semantic of CAS is as follows. Given the instruction `CAS(memory location, old value, new value)`, atomically compares the value at `memory location` to see if it equals `old value`; if so, sets the value at `memory location` to `new value` and returns true; otherwise, leaves the value at `memory location` unchanged and returns false. Concurrent algorithms often utilize CAS as follows:

1. Read the current value `old value = read(memory location)`.
2. Compute `new value` from `old value` by manipulating some resources associated with `old value` and allocating new resources for `new value`.
3. Call `CAS(memory location, old value, new value)`. If that succeeds, the new resources for `new value` remain valid because it was computed using valid resources associated with `old value`, which has not been modified since the last read. Otherwise, free up `new value` because `old value` is no longer there, so its associated resources are not valid.

This scheme is susceptible to the notorious ABA problem:

1. Process 1 reads the current value of `memory location` and reads out `A`.
2. Process 1 manipulates resources associated with `A`, and allocates resources based on these resources.
3. Process 1 suspends.
4. Process 2 reads the current value of `memory location` and reads out `A`.
5. Process 2 `CAS(memory location, A, B)` so that resources associated with `A` are no longer valid.
6. Process 3 `CAS(memory location, B, A)` and allocates new resources associated with `A`.
7. Process 1 continues and `CAS(memory location, A, new value)` relying on the fact that the old resources associated with `A` are still valid while in fact they aren't.

To safe-guard against ABA problem, one must ensure that between the time a process reads out a value from a shared memory location and the time it calls `CAS` on that location, there's no possibility another process has `CAS` the memory location to the same value. Some notable schemes are **monotonic version tag** ([6]) and **hazard pointer** ([12]).

2.5.2 Safe memory reclamation problem

The problem of safe memory reclamation often arises in concurrent algorithms that dynamically allocate memory. In such algorithms, dynamically-allocated memory must be freed at some point. However, there's a good chance that while a process is freeing memory, other processes contending for the same memory are keeping a reference to that memory. Therefore, deallocated memory can potentially be accessed, which is erroneous. Solutions ensure that memory is only freed when no other processes are holding references to it. In garbage-collected programming environments, this problem can be conveniently push to the garbage collector. In non-garbage-collected programming environments, however, custom schemes must be utilized. Examples include using a reference counter to count the number of processes holding a reference to some memory and **hazard pointer** [12] to announce to other processes that some memory is not to be freed.

2.6 MPI-3

MPI stands for message passing interface, which is a **message-passing library interface specification**. Design goals of MPI includes high availability across platforms, efficient communication, thread-safety, reliable and convenient communication interface while still allowing hardware-specific accelerated mechanisms to be exploited [1].

2.6.1 MPI-3 RMA

RMA in MPI RMA stands for remote memory access. As introduced in the first section of Section Chapter II, RMA APIs is introduced in MPI-2 and its capabilities are further extended in MPI-3 to conveniently express irregular applications. In general, RMA is intended to support applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing [1]. In such applications, one process, based on the data it needs, knowing the data distribution, can compute the nodes where the data is stored. However, because data acess pattern is not known, each process cannot know whether any other processes will access its data.

Using the traditional `Send`/`Receive` interface, both sides need to issue matching operations by distributing appropriate transfer parameters. This is not suitable, as previously explain, only the side that needs to access the data knows all the transfer parameters while the side that stores the data cannot anticipate this.

2.6.2 MPI-RMA communication operations

RMA only requires one side to specify all the transfer parameters and thus only that side to participate in data communication.

To utilize MPI RMA, each process needs to open a memory window to expose a segment of its memory to RMA communication operations such as remote writes (`MPI_PUT`), remote reads (`MPI_GET`) or remote accumulates (`MPI_ACCUMULATE`, `MPI_GET_ACCUMULATE`, `MPI_FETCH_AND_OP`, `MPI_COMPARE_AND_SWAP`) [1]. These remote communication operations only requires one side to specify.

2.6.3 MPI-RMA synchronization

Besides communication of data from the sender to the receiver, one also needs to synchronize the sender with the receiver. That is, there must be a mechanism to ensure the completion of RMA communication calls or that any remote operations have taken effect. For this purpose, MPI RMA provides **active target synchronization** and **passive target synchronization**. In this document, we're particularly interested in **passive target synchronization** as this mode of synchronization does not require the target process of an RMA operation to explicitly issue a matching synchronization call with the origin process, easing the expression of irregular applications [13].

In **passive target synchronization**, any RMA communication calls must be within a pair of `MPI_Win_lock`/`MPI_Win_unlock` or `MPI_Win_lock_all`/`MPI_Win_unlock_all`. After the unlock call, those RMA communication calls are guaranteed to have taken effect.

One can also force the completion of those RMA communication calls without the need for the call to unlock using flush calls such as `MPI_Win_flush` or `MPI_Win_flush_local`.

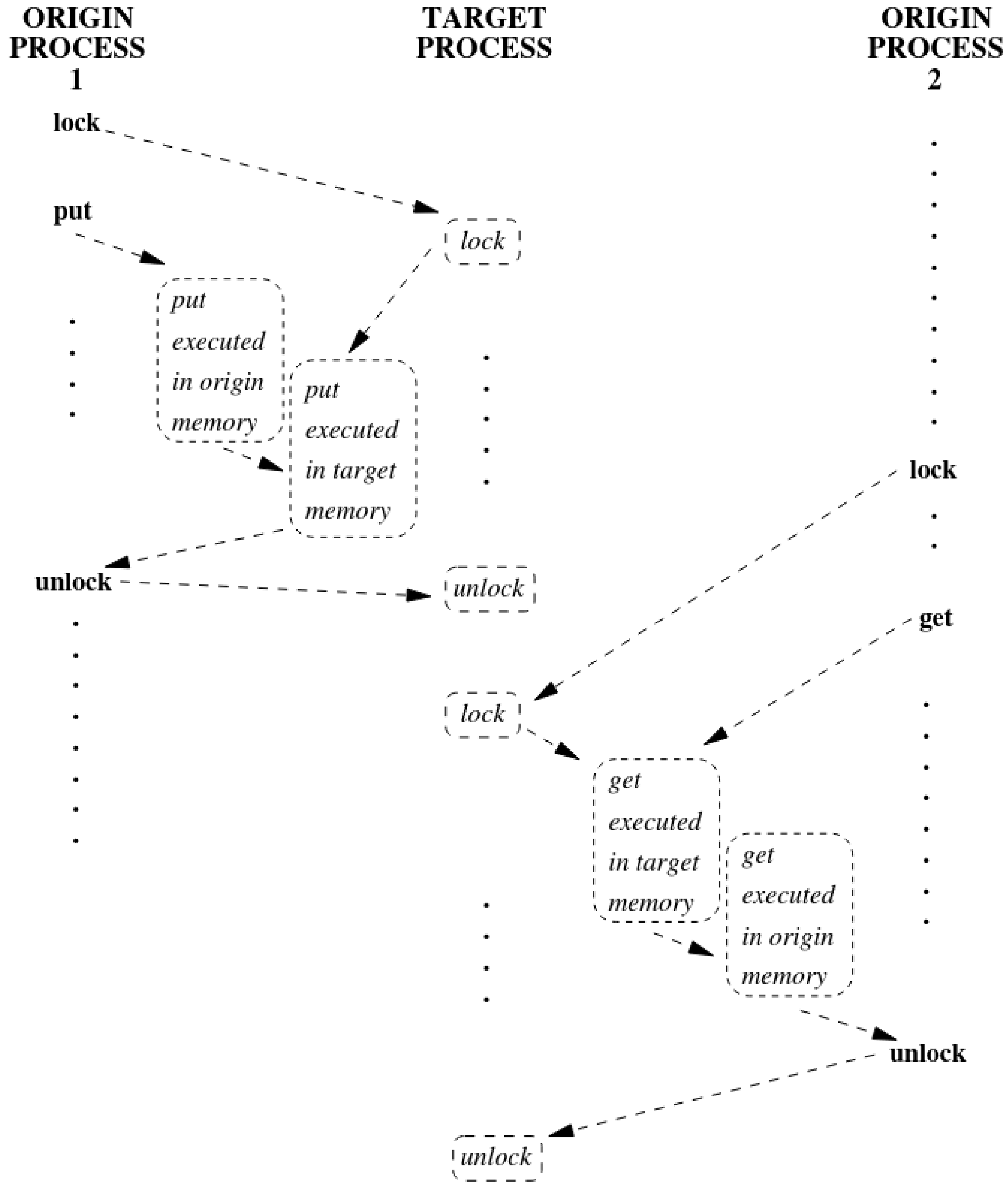

Figure 2: An illustration of passive target communication. Dashed arrows represent synchronization (source: [1])

2.7 Pure MPI approach of porting shared memory algorithms

In pure MPI, we use MPI exclusively for communication and synchronization. With MPI RMA, the communication calls that we utilize are:

- Remote read: `MPI_Get`
- Remote write: `MPI_Put`

- Remote accumulation: `MPI_Accumulate`, `MPI_Get_accumulate`, `MPI_Fetch_and_op` and `MPI_Compare_and_swap`.

For lock-free synchronization, we choose to use **passive target synchronization** with `MPI_Win_lock_all`/`MPI_Win_unlock_all`.

In the MPI-3 specification [1], these functions are specified as follows:

Operation	Usage
`MPI_Win_lock_all`	Starts and RMA access epoch to all processes in a memory window, with a lock type of `MPI_LOCK_SHARED`. The calling process can access the window memory on all processes in the memory window using RMA operations. This routine is not collective.
`MPI_Win_unlock_all`	Matches with an `MPI_Win_lock_all` to unlock a window previously locked by that `MPI_Win_lock_all`.

Table 1: Specification of `MPI_Win_lock_all` and `MPI_Win_unlock_all`

The reason we choose this is 3-fold:

- Unlike **active target synchronization**, **passive target synchronization** does not require the process whose memory is being accessed by an MPI RMA communication call to participate in. This is in line with our intention to use MPI RMA to easily model irregular applications like MPSCs.
- Unlike **active target synchronization**, `MPI_Win_lock_all` and `MPI_Win_unlock_all` do not need to wait for a matching synchronization call in the target process, and thus, is not delayed by the target process.
- Unlike **passive target synchronization** with `MPI_Win_lock`/`MPI_Win_unlock`, multiple calls of `MPI_Win_lock_all` can succeed concurrently, so one process needing to issue MPI RMA communication calls do not block others.

An example of our pure MPI approach with `MPI_Win_lock_all`/`MPI_Win_unlock_all`, inspired by [13], is illustrated in the following:

```
MPI_Win_lock_all(0, win);

MPI_Get(...); // Remote get
MPI_Put(...); // Remote put
MPI_Accumulate(..., MPI_REPLACE, ...); // Atomic put
MPI_Get_accumulate(..., MPI_NO_OP, ...); // Atomic get
MPI_Fetch_and_op(...); // Remote fetch-and-op
MPI_Compare_and_swap(...); // Remote compare and swap
...

MPI_Win_flush(...); // Make previous RMA operations take effects
MPI_Win_flush_local(...); // Make previous RMA operations take
effects locally
...

MPI_Win_unlock_all(win);
```

Listing 1: An example snippet showcasing our synchronization approach in MPI RMA

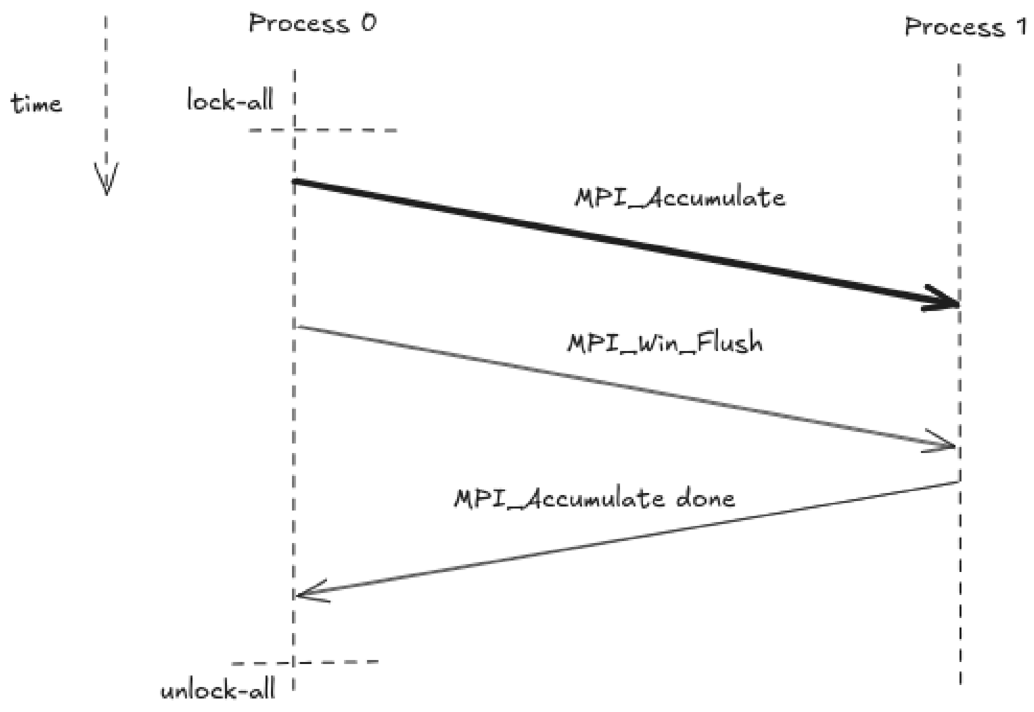

Figure 3: An illustration of our synchronization approach in MPI RMA

Chapter III Related works

There exists numerous research into the design of lock-free shared memory MPMCs and SPSCs. Interestingly, research into lock-free MPSCs are noticeably scarce. Although in principle, MPMCs and SPSCs can both be adapted for MPSCs use cases, specialized MPSCs can usually yield much more performance. In reality, we have only found 4 papers that are concerned with direct support of lock-free MPSCs: LTQueue [7], DQueue [9], WRLQueue [14] and Jiffy [8]. Table 2 summarizes the charateristics of these algorithms.

MPSCs	LTQueue	DQueue	WRLQueue	Jiffy
ABA solution	Load-link/Store-conditional	Incorrect custom scheme (*)	Custom scheme	Custom scheme
Memory reclamation	Custom scheme	Incorrect custom scheme (*)	Custom scheme	Custom scheme
Progress guarantee of dequeue	Wait-free	Wait-free	Blocking (*)	Wait-free
Progress guarantee of enqueue	Wait-free	Wait-free	Wait-free	Wait-free
Number of elements	Unbounded	Unbounded	Unbounded	Unbounded

Table 2: Characteristic summary of existing shared memory MPSCs. The cell marked with (*) indicates that our evaluation contradicts with the author's claims

LTQueue [7] is the earliest wait-free shared memory MPSC to our knowledge. This algorithm is wait-free with $O(\log n)$ time complexity for both enqueues and dequeues, with n being the number of enqueuers. Their main idea is to split the MPSC among the enqueuers so that each enqueuer maintains a local SPSC data structure, which is only shared with the dequeuer. This improves the MPSC's scalability as multiple enqueues can complete the same time. The enqueuers shared a distributed counter and use it to label each item in their local SPSC with a specific timestamp. The timestamps are organized into nodes of a min-heap-like tree so that the dequeuer can look at the root of tree to determine which local SPSC to dequeue next. The min-heap property of the tree is preserved by a novel wait-free timestamp-refreshing operation. Memory reclamation becomes trivial as each MPSC entry is only shared by one enqueuer and one dequeuer in the local SPSC. The algorithm avoids ABA problem by utilizing load-link/store-conditional (LL/SC). This, on the other hand, presents a challenge in directly porting LTQueue as LL/SC is not widely available as the more popular CAS instruction.

DQueue [9] focuses on optimizing performance. It aims to be cache-friendly by having each enqueuer batches their updates in a local buffer to decrease cache misses. It also try to replace expensive atomic instructions such as CAS as many as possible. The MPSC is represented as a linked list of segments (which is an array). To enqueue, the enqueuer reserves a slot in the segment list and enqueues the value into the local buffer. If the local buffer is full, the enqueuer flushes the buffer and writes it onto every reserved slot in the segment list. The producer dequeues the values in the segment list in order, upon encountering a reserved but empty slot, it helps all enqueuers flush their local buffers. For memory reclamation, DQueue utilized a dedicated garbage collection thread that reclaims all fully dequeued segments. However, their algorithm is flawed and a segment maybe freed while some process is holding a reference to it.

WRLQueue [14] is a lock-free MPSC for embedded real-time system. Its main purpose is to avoid excessive modification of storage space. WRLQueue is simplfy a pair of buffer, one is worked on by multiple enqueuers and the other is work on by the dequeuer. The enqueuers batch their enqueues and write multiple elements onto the buffer once at a time. The dequeuer upon invocation will swap its buffer with the enqueuer’s buffers to dequeue from it. However, this requires the dequeuer to wait for all enqueue operations to complete in their buffer. If an enqueue suspends or dies, the dequeuer will have to wait forever, this clearly violates the property of non-blocking.

Jiffy [8] is a fast and memory-efficient wait-free MPSC by avoiding excessive allocation of memory. Like DQueue, Jiffy represents the queue as a linked list of segments. Each enqueue reserves a slot in the segment, extends the linked-list as appropriately, writes the value into the slot and sets a per-slot flag to indicate that the slot is ready to be dequeued. To dequeue, the dequeuer repeatedly scan all the slots to find the first-ready-to-be-dequeue slot. Jiffy shows significant good memory usage and throughput compared to other previous state-of-the-art MPMCs.

Chapter IV Distributed MPSCs

Based on the MPSC algorithms we have surveyed in Chapter III, we propose two wait-free distributed MPSC algorithms:

- One is a direct modification of LTQueue [7] without any usage of LL/SC.
- One is inspired by the timestamp-refreshing idea of LTQueue [7] and repeated-rescan of Jiffy [8]. Although it still bears some resemblance to LTQueue, we believe it to be more optimized for distributed context.

MPSC	**LTQueue without LL/SC**	**Optimized distributed LTQueue**
Correctness	Linearizable	Linearizable
Progress guarantee of dequeue	Wait-free	Wait-free
Progress guarantee of enqueue	Wait-free	Wait-free
Time complexity of dequeue	$O(\log n)$ R + $O(\log n)$ A	constant R + $O(n)$ A
Time complexity of enqueue	$O(\log n)$ R + $O(\log n)$ A	constant R + constant A
ABA solution	Unique timestamp	No harmful ABA problem
Memory reclamation	Custom scheme	Custom scheme
Number of elements	Unbounded	Unbounded

Table 3: Characteristic summary of our proposed distributed MPSCs. n is the number of enqueuers, R stands for **remote operation** and A stands for **atomic operation**

In this section, we present our proposed distributed MPSCs in detail. Any other discussions about theoretical aspects of these algorithms such as linearizability, progress guarantee, time complexity are deferred to Chapter V.

In our description, we assume that each process in our program is assigned a unique number as an identifier, which is termed as its **rank**. The numbers are taken from the range of `[0, size - 1]`, with `size` being the number of processes in our program.

4.1 Distributed primitives in pseudocode

Although we use MPI-3 RMA to implement these algorithms, the algorithm specifications themselves are not inherently tied to MPI-3 RMA interfaces. For clarity and convenience in specification, we define the following distributed primitives used in our pseudocode.

`remote<T>`: A distributed shared variable of type T. The process that physically stores the variable in its local memory is referred to as the **host**. This represents data that can be accessed or modified remotely by other processes.

`void aread_sync(remote<T> src, T* dest)`: Issue a synchronous read of the distributed variable `src` and stores its value into the local memory location pointed to by `dest`. The read is guaranteed to be completed when the function returns.

`void aread_sync(remote<T*> src, int index, T* dest)`: Issue a synchronous read of the element at position `index` within the distributed array `src` (where `src` is a pointer to a remotely hosted array of type `T`) and stores the value into the local memory location pointed to by `dest`. The read is guaranteed to be completed when the function returns.

`void awrite_sync(remote<T> dest, T* src)`: Issue a synchronous write of the value at the local memory location pointed to by `src` into the distributed variable `dest`. The write is guaranteed to be completed when the function returns.

`void awrite_sync(remote<T*> dest, int index, T* src)`: Issue a synchronous write of the value at the local memory location pointed to by `src` into the element at position `index` within the distributed array `dest` (where `dest` is a pointer to a remotely hosted array of type `T`). The write is guaranteed to be completed when the function returns.

`void aread_async(remote<T> src, T* dest)`: Issue an asynchronous read of the distributed variable `src` and initiate the transfer of its value into the local memory location pointed to by `dest`. The operation may not be completed when the function returns.

`void aread_async(remote<T*> src, int index, T* dest)`: Issue an asynchronous read of the element at position `index` within the distributed array `src` (where `src` is a pointer to a remotely hosted array of type `T`) and initiate the transfer of its value into the local memory location pointed to by `dest`. The operation may not be completed when the function returns.

`void awrite_async(remote<T> dest, T* src)`: Issue an asynchronous write of the value at the local memory location pointed to by `src` into the distributed variable `dest`. The operation may not be completed when the function returns.

`void awrite_async(remote<T*> dest, int index, T* src)`: Issue an asynchronous write of the value at the local memory location pointed to by `src` into the element at position `index` within the distributed array `dest` (where `dest` is a pointer to a remotely hosted array of type `T`). The operation may not be completed when the function returns.

`void flush(remote<T> src)`: Ensure that all read and write operations on the distributed variable `src` (or its associated array) issued before this function call are fully completed by the time the function returns.

4.2 A basis distributed SPSC

The two algorithms we propose here both utilize a distributed SPSC data structure, which we will present first. For implementation simplicity, we present a bounded SPSC, effectively make our proposed algorithms support only a bounded number of elements. However, one can trivially substitute another distributed unbounded SPSC to make our proposed algorithms support an unbounded number of elements, as long as this SPSC supports the same interface as ours.

Placement-wise, all shared data in this SPSC is hosted on the enqueuer.

Types

data_t = The type of data stored

Shared variables

First: remote<uint64_t>

The index of the last undequeued entry. Hosted at the enqueuer.

Last: remote<uint64_t>

The index of the last unenqueued entry. Hosted at the enqueuer.

Data: remote<data_t*>

An array of data_t of some known capacity. Hosted at the enqueuer.

Enqueuer-local variables

Capacity: A read-only value indicating the capacity of the SPSC

First_buf: The cached value of First

Last_buf: The cached value of Last

Dequeuer-local variables

Capacity: A read-only value indicating the capacity of the SPSC

First_buf: The cached value of First

Last_buf: The cached value of Last

Enqueuer initialization

Initialize First and Last to 0

Initialize Capacity

Allocate array in Data

First_buf = Last_buf = 0

Dequeuer initialization

Initialize Capacity

First_buf = Last_buf = 0

The procedures of the enqueuer are given as follows.

Procedure 2: bool spsc_enqueue(data_t v)

```
1 new_last = Last_buf + 1
2 if (new_last - First_buf > Capacity)
3     aread_sync(First, &First_buf)
4     if (new_last - First_buf > Capacity)
5         return false
6 awrite_sync(Data, Last_buf % Capacity, &v)
7 awrite_sync(Last, &new_last)
8 Last_buf = new_last
9 return true
```

spsc_enqueue first computes the new Last value (line 1). If the queue is full as indicating by the difference the new Last value and First-buf (line 2), there can still be the possibility that some elements have been dequeued but `First-buf` hasn't been synced with `First` yet, therefore, we first refresh the value of `First-buf` by fetching from `First` (line 3). If the queue is still full (line 4), we signal failure (line 5). Otherwise, we proceed to write the enqueued value to the entry at `Last_buf % Capacity` (line 6), increment `Last` (line 7), update the value of `Last_buf` (line 8) and signal success (line 9).

Procedure 3: `bool spsc_readFront`$_e$`(data_t* output)`

```
10 if (First_buf >= Last_buf)
11     return false
12 aread_sync(First, &First_buf)
13 if (First_buf >= Last_buf)
14     return false
15 aread_sync(Data, First_buf % Capacity, output)
16 return true
```

`spsc_readFront`$_e$ first checks if the SPSC is empty based on the difference between `First_buf` and `Last_buf` (line 10). Note that if this check fails, we signal failure immediately (line 11) without refetching either `First` or `Last`. This suffices because `Last` cannot be out-of-sync with `Last_buf` as we're the enqueuer and `First` can only increase since the last refresh of `First_buf`, therefore, if we refresh `First` and `Last`, the condition on line 10 would return `false` anyways. If the SPSC is not empty, we refresh `First` and re-perform the empty check (line 12-14). If the SPSC is again not empty, we read the queue entry at `First_buf % Capacity` into `output` (line 15) and signal success (line 16).

The procedures of the dequeuer are given as follows.

Procedure 4: `bool spsc_dequeue(data_t* output)`

```
15 new_first = First_buf + 1
16 if (new_first > Last_buf)
17     aread_sync(Last, &Last_buf)
18     if (new_first > Last_buf)
19         return false
20 aread_sync(Data, First_buf % Capacity, output)
21 awrite_sync(First, &new_first)
22 First_buf = new_first
23 return true
```

`spsc_dequeue` first computes the new `First` value (line 15). If the queue is empty as indicating by the difference the new `First` value and `Last-buf` (line 16), there can still be the possibility that some elements have been enqueued but `Last-buf` hasn't been synced with `Last` yet, therefore, we first refresh the value of `Last-buf` by fetching from `Last` (line 17). If the queue is still empty (line 18), we signal failure (line 19). Otherwise, we proceed to read the top value at `First_buf % Capacity` (line 20) into `output`, increment `First` (line 21) - effectively dequeue the element, update the value of `First_buf` (line 22) and signal success (line 23).

Procedure 5: `bool spsc_readFront`$_d$`(data_t* output)`

```
24 if (First_buf >= Last_buf)
25     aread_sync(Last, &Last_buf)
26     if (First_buf >= Last_buf)
27         return false
28 aread_sync(Data, First_buf % Capacity, output)
29 return true
```

`spsc_readFront`$_d$ first checks if the SPSC is empty based on the difference between `First_buf` and `Last_buf` (line 24). If this check fails, we refresh `Last_buf` (line 25) and recheck (line 26). If the recheck fails, signal failure (line 27). If the SPSC is not empty, we read the queue entry at `First_buf % Capacity` into `output` (line 28) and signal success (line 29).

4.3 Modified LTQueue without LL/SC

The structure of our modified LTQueue is shown as in Image 1.

We differentiate between 2 types of nodes: **enqueuer nodes** (represented as the rectangular boxes at the bottom of Image 1) and normal **tree nodes** (represented as the circular boxes in Image 1).

Each enqueuer node corresponds to an enqueuer. Each time the local SPSC is enqueued with a value, the enqueuer timestamps the value using a distributed counter shared by all enqueuers. An enqueuer node stores the SPSC local to the corresponding enqueuer and a `min_timestamp` value which is the minimum timestamp inside the local SPSC.

Each tree node stores the rank of an enqueuer process. This rank corresponds to the enqueuer node with the minimum timestamp among the node's children's ranks. The tree node that's attached to an enqueuer node is called a **leaf node**, otherwise, it's called an **internal node**.

Note that if a local SPSC is empty, the `min_timestamp` variable of the corresponding enqueuer node is set to `MAX` and the corresponding leaf node's rank is set to a `DUMMY` rank.

Placement-wise:

- The **enqueuer nodes** are hosted at the corresponding **enqueuer**.
- All the **tree nodes** are hosted at the **dequeuer**.

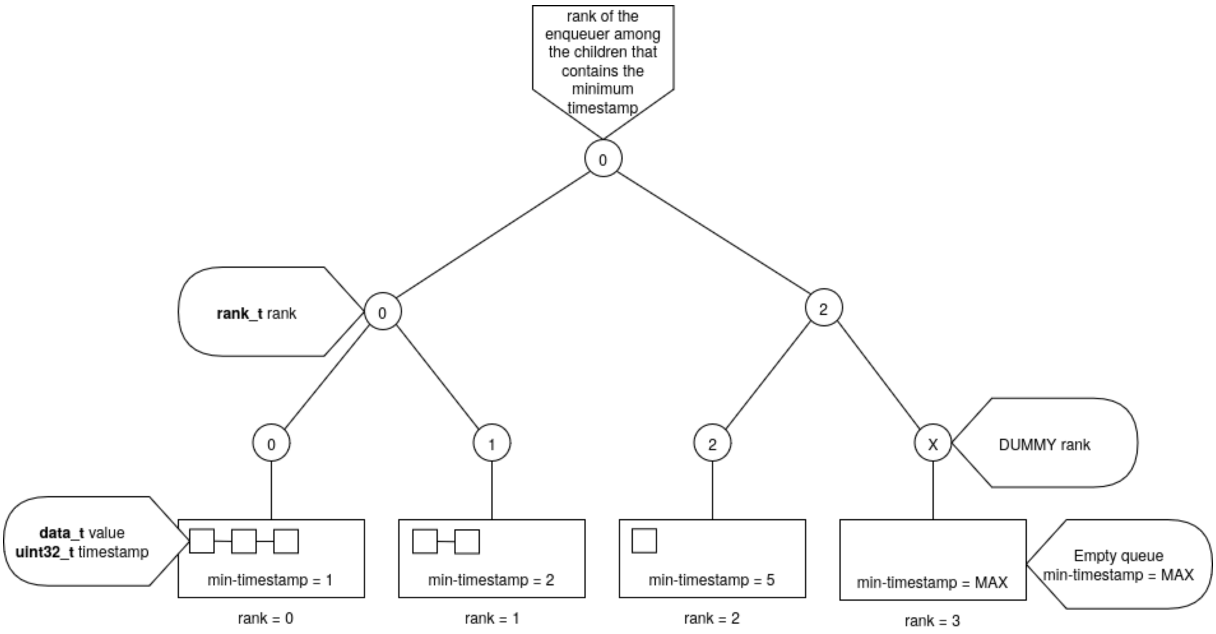

Image 1: Modified LTQueue's structure

- The distributed counter, which the enqueuers use to timestamp their enqueued value, is hosted at the **dequeuer**.

Below is the types utilized in our version of LTQueue.

Types

`data_t` = The type of the data to be stored

`spsc_t` = The type of the SPSC

`rank_t` = The type of the rank of an enqueuer process tagged with a unique timestamp (version) to avoid ABA problem

> **struct**
> > `value`: `uint32_t`
> >
> > `version`: `uint32_t`
>
> **end**

`timestamp_t` = The type of the timestamp tagged with a unique timestamp (version) to avoid ABA problem

> **struct**
> > `value`: `uint32_t`
> >
> > `version`: `uint32_t`
>
> **end**

Types (cont)

`enqueuer_t` = The type of an enqueuer node

> **struct**
> > `spsc`: `spsc_t`
> >
> > `min_timestamp`: `timestamp_t`
>
> **end**

`node_t` = The type of a tree node

> **struct**
> > `rank`: `rank_t`
>
> **end**

The shared variables in our LTQueue version are as followed.

Note that we have described a very specific and simple way to organize the tree nodes in LTQueue in a min-heap-like array structure hosted on the sole dequeuer. We will resume our description of the related tree-structure procedures `parent()` (Procedure 7), `children()` (Procedure 8), `leaf_node_index()` (Procedure 9) with this representation in mind. However, our algorithm doesn't strictly require this representation and can be subtituted with other more-optimized representations & distributed placements, as long as the similar tree-structure procedures are supported.

Shared variables

> `Counter`: `remote<uint64_t>`
>
> > A distributed counter shared by the enqueuers. Hosted at the dequeuer.
>
> `Tree_size`: `uint64_t`
>
> > A read-only variable storing the number of tree nodes present in the LTQueue.
>
> `Nodes`: `remote<node_t>`
>
> > An array with `Tree_size` entries storing all the tree nodes present in the LTQueue shared by all processes.
> >
> > Hosted at the dequeuer.
> >
> > This array is organized in a similar manner as a min-heap: At index `0` is the root node. For every index $i > 0$, $\left\lfloor \frac{i-1}{2} \right\rfloor$ is the index of the parent of node i. For every index $i > 0$, $2i + 1$ and $2i + 2$ are the indices of the children of node i.
>
> `Dequeuer_rank`: `uint32_t`
>
> > The rank of the dequeuer process.

Similar to the fact that each process in our program is assigned a rank, each enqueuer process in our program is assigned an **order**. The following procedure computes an enqueuer's order based on its rank:

Procedure 6: `uint32_t enqueuer_order(uint32_t enqueuer_rank)`

```
return enqueuer_rank > Dequeuer_rank ? enqueuer_rank - 1 : enqueuer_rank
```

This procedure is rather straightforward: Each enqueuer is assigned an order in the range `[0, size - 2]`, with `size` being the number of processes and the total ordering among the enqueuers based on their ranks is the same as the total ordering among the enqueuers based on their orders.

Enqueuer-local variables

> `Enqueuer_count`: `uint64_t`
>
> > The number of enqueuers.
>
> `Self_rank`: `uint32_t`
>
> > The rank of the current enqueuer process.
>
> `Self_node`: `remote<enqueuer_t>`
>
> > Shared by the dequeuer and this enqueuer.

Dequeuer-local variables

> `Enqueuer_count`: `uint64_t`
>
> > The number of enqueuers.
>
> `Enqueuers`: **array** `[0..size - 2]` of `remote<enqueuer_t>`, with `size` being the number of processes
>
> > An entry at index i corresponds to the `Self_node` value at the enqueuer with an order of i.

Enqueuer initialization

> Initialize `Enqueuer_count`, `Self_rank` and `Dequeuer_rank`.
>
> Initialize `Self_node.spsc` to the initial state.
>
> Initialize `Self_node.min_timestamp` to `timestamp_t {MAX_TIMESTAMP, 0}`.

Dequeuer initialization

> Initialize `Enqueuer_count`, `Self_rank` and `Dequeuer_rank`.
>
> Initialize `Counter` to `0`.
>
> Initialize `Tree_size` to `Enqueuer_count * 2`.
>
> Initialize `Nodes` to an array with `Tree_size` entries. Each entry is initialized to `node_t {DUMMY_RANK}`.
>
> Initialize `Enqueuers`, synchronizing each entry with the corresponding enqueuer.

We first present the tree-structure utility procedures that are shared by both the enqueuer and the dequeuer:

Procedure 7: `uint32_t parent(uint32_t index)`

```
2 return (index - 1) / 2
```

`parent` returns the index of the parent tree node given the node with index `index`. These indices are based on the shared `Nodes` array. Based on how we organize the `Nodes` array, the index of the parent tree node of `index` is `(index - 1) / 2`.

Similarly, `children` returns all indices of the child tree nodes given the node with index `index`. These indices are based on the shared `Nodes` array. Based on how we organize the `Nodes` array, these indices can be either `index * 2 + 1` or `index * 2 + 2`.

Procedure 8: `vector<uint32_t> children(uint32_t index)`

```
left_child = index * 2 + 1
right_child = left_child + 1
res = vector<uint32_t>()
if (left_child >= Tree_size)
    return res
res.push(left_child)
if (right_child >= Tree_size)
    return res
res.push(right_child)
return res
```

Procedure 9: `uint32_t leaf_node_index(uint32_t enqueuer_rank)`

```
return Tree_size + enqueuer_order(enqueuer_rank)
```

`leaf_node_index` returns the index of the leaf node that's logically attached to the enqueuer node with rank `enqueuer_rank` as in Image 1.

The followings are the enqueuer procedures:

Procedure 10: `bool enqueue(data_t value)`

```
count = FAA(Counter)
timestamp = timestamp_t {count, Self_rank}
spsc_enqueue(Self_node.spsc, (value, timestamp))
propagate_e(Self_rank)
```

Procedure 11: `void propagate_e()`

Procedure 12: `bool refresh_e(node_t* currentNode)`

Procedure 13: `bool refreshTimestamp_e(uint32_t rank)`

Procedure 14: `bool refreshLeaf`$_e$`(uint32_t rank)`

The followings are the dequeuer procedures:

Procedure 15: `bool dequeue(data_t output)`

Procedure 16: `void propagate`$_d$`(uint32_t rank)`

Procedure 17: `bool refresh`$_d$`(node_t* currentNode)`

Procedure 18: `bool refreshTimestamp`$_d$`(uint32_t rank)`

Procedure 19: `bool refreshLeaf`$_d$`(uint32_t rank)`

4.4 Optimized LTQueue for distributed context

Chapter V Theoretical aspects

Chapter VI Preliminary results

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

Chapter VII Conclusion & Future works

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

References

[1] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 3.1*. 2015. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[2] B. Brock, A. Buluç, and K. Yelick, "BCL: A Cross-Platform Distributed Data Structures Library," 2019, *Association for Computing Machinery*. doi: 10.1145/3337821.3337912.

[3] Thanh-Dang Diep, Phuong Hoai Ha, and Karl Fürlinger, "A general approach for supporting nonblocking data structures on distributed-memory systems," 2023. doi: https://doi.org/10.1016/j.jpdc.2022.11.006.

[4] John D. Valois, "Implementing Lock-Free Queues," 1994.

[5] L. Lamport, "Specifying Concurrent Program Modules," 1983, *Association for Computing Machinery*. doi: 10.1145/69624.357207.

[6] Mage M.Michael and Michael L.Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," 1996, *Association for Computing Machinery*. doi: 10.1145/248052.248106.

[7] P. Jayanti and S. Petrovic, "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks," 2005, *Springer-Verlag*. doi: 10.1007/11590156_33.

[8] D. Adas and R. Friedman, "A Fast Wait-Free Multi-Producers Single-Consumer Queue," 2022, *Association for Computing Machinery*. doi: 10.1145/3491003.3491004.

[9] J. Wang, Q. Jin, X. Fu, Y. Li, and P. Shi, "Accelerating Wait-Free Algorithms: Pragmatic Solutions on Cache-Coherent Multicore Architectures," 2019. doi: 10.1109/ACCESS.2019.2920781.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[11] M. Herlihy, "Wait-free synchronization," 1991, *Association for Computing Machinery*. doi: 10.1145/114005.102808.

[12] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," 2004, *IEEE Press*. doi: 10.1109/TPDS.2004.8.

[13] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the MPI 3.0 one-sided communication interface," 2016, *John Wiley and Sons Ltd*. doi: 10.1002/cpe.3758.

[14] Q. Yang, L. Tang, Y. Guo, N. Kuang, S. Zhong, and H. Luo, "WRLqueue: A Lock-Free Queue For Embedded Real-Time System," 2022. doi: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00197.