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 Listings

# 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. Therefore, an MPSC can easily present a performance bottleneck if not designed properly, resulting in loss of computation power. A desirable distributed MPSC should be able to exploit the highly concurrent nature of distributed computing. One favorable characteristic of distributed data structures is non-blocking or more specifically, 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 port into distributed context using the approach introduced in Chapter III.

## 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 [1], [2], [3] and more recently [4], [5]. 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. This is largely because the programming model used for distributed computing differs from that of shared memory. However, the gap between the two domains has been bridged with the new 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 potential MPSC algorithms to be ported to distributed MPSC algorithms using MPI-3 RMA.
- Adapt/Optimize the ported algorithms to fit the constraints of distributed computing.
- Benchmark the ported algorithms.

## 1.3 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 discusses the general idea we use to port shared-memory algorithms while keeping their lock-freedom characteristic using MPI-3 RMA. We further discuss the possibilities of further optimization using MPI-3 SHM and C++11 to optimize intra-node communication. This presents a potential performance boost for NUMA-aware shared-memory algorithms

Chapter IV 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 V selects some of the algorithms we have surveyed and introduce modification to fit the distributed context. We further introduce optimization based on our domain knowledge, which the shared-memory algorithms, in their inception, are oblivious to.

Chapter VI introduces our setup and benchmarking processes. We also analyze the result to assess the various factors that affect the performance of an algorithm and its implementation.

Chapter VII and Chapter VIII 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 powerful enough - its consensus number equals the number of producers [6].

## 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 OS decides to suspends it to schedule another one, this means until the process is awken again, 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 corrupt the whole program, and this possibility can happen more easily in distributed computing, due to 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 unnecessarily restrictive regarding its use of mutual exclusion. These algorithms are said to be **blocking**. An algorithm is said to be **non-blocking** if a failure or slowdown in one process cannot cause the failure or slowdown 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. One well-known technique in achieving lock-freedom is the help mechanism, made popular by [3].

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

- We have $n$ concurrent method calls $m_1, m_2, ..., m_n$.
- 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 $l_1 < l_2 < ... < l_n$, the effect of these $n$ concurrent method calls $m_1, m_2, ..., m_n$ must be equivalent to calling $m_1, m_2, ..., m_n$ **sequentially**, one after the other in that order.

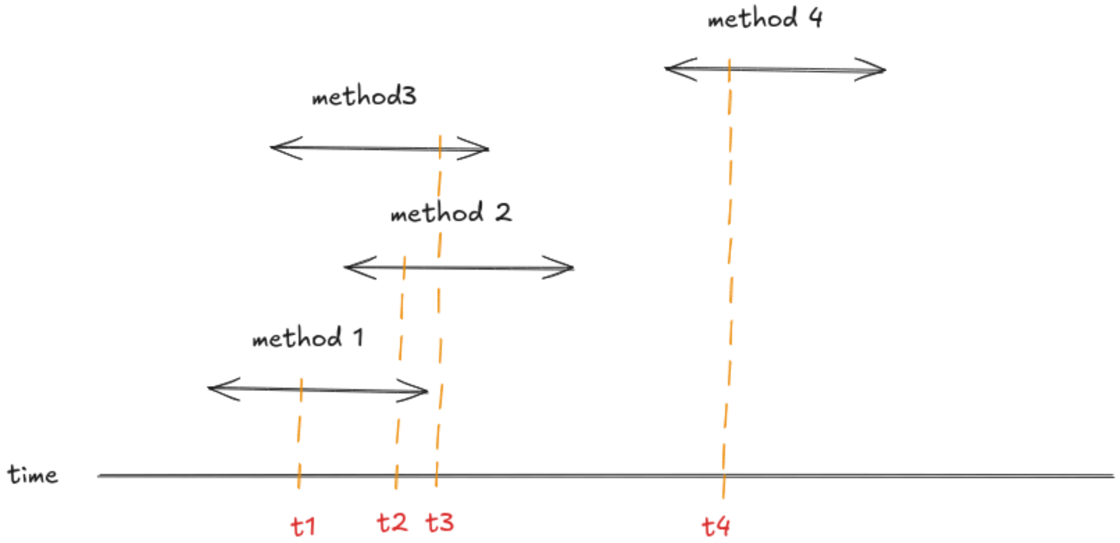

Figure 1: Linerization points of method 1, method 2, method 3, method 4 happens at $t_1 < t_2 < t_3 < t_4$, therefore, their effects will be observed in this order as if we call method 1, method 2, method 3, method 4 sequentially

## 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 $\infty$ - which means it's the most powerful atomic instruction [8] (2) CAS is implemented in most hardware (3) some concurrent lock-free data structures such as MPSC can only be implemented using powerful atomic instruction 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** (used in [3]) and **hazard pointer** (introduced in [9]).

### 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 erroneneous. 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** [9] to announce to other processes that some memory is not to be freed.

## 2.6 C++11 concurrency

### 2.6.1 Motivation

### 2.6.2 C++11 memory model

### 2.6.3 C++11 atomics

## 2.7 MPI-3

### 2.7.1 MPI-3 RMA

### 2.7.2 MPI-3 SHM

# Chapter III Approach

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 IV Literature review

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 V Porting

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 VI Evaluation

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

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 VIII 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 tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

# References

[1] John D. Valois, “Implementing Lock-Free Queues,” 1994.

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

[3] 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.

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

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

[6] 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.

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

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

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