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

Chapter I Introduction 8
1.1 Motivation 8
1.2 Objective 9
1.3 Scope 9
1.4 Structure 9
Chapter II Background 11
2.1 Irregular applications 11
2.2 Multiple-producer, single-consumer (MPSC) 11
2.3 Progress guarantee 11
2.3.1 Lock-free algorithms 12
2.3.2 Wait-free algorithms 12
2.4 Correctness - Linearizability 12
2.5 Common issues when designing lock-free algorithms 13
2.5.1 ABA problem 13
2.5.2 Safe memory reclamation problem 14
2.6 C++11 concurrency 15
2.6.1 Motivation 15
2.6.2 C++11 memory model 15
2.6.2.1 Structural aspects 16
2.6.2.2 Concurrency aspects 16
2.6.3 C++11 atomics 16
2.7 MPI-3 18
2.7.1 MPI-3 RMA 18
2.7.1.1 MPI-RMA communication operations 19
2.7.1.2 MPI-RMA synchronization 19
2.7.2 MPI-3 SHM 20
2.8 Porting shared memory algorithms to distributed context approaches 21
2.8.1 Pure MPI 21
2.8.2 MPI+MPI 22
2.8.3 MPI+MPI with C++11 23
Chapter III Related works 24
Chapter IV Distributed queues 25
Chapter V Theoretical aspects 26
Chapter VI Preliminary results 27
Chapter VII Conclusion & Future works 28
References 29

List of Listings

Listing 1 Example memory locations for a user-defined struct 16
Listing 2 An example snippet showcasing our synchronization approach in MPI RMA 22

List of Tables

Table 1 Supported atomic operations on `std::atomic_flag` (C++17) 17
Table 2 Available atomic operations on atomic types (C++17) 17
Table 3 Available `std::memory_order` values (C++17). On the `Load`, `Store` and `RMW` columns, `Y` means that this memory order can be specified on `load`, `store` and RMW operations, - means that we intentionally ignore this entry. 18
Table 4 Specification of `MPI_Win_lock_all` and `MPI_Win_unlock_all` 21

List of Images

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 13
Figure 2 An illustration of passive target communication. Dashed arrows represent synchronization (source: [1]) 20
Figure 3 An illustration of our synchronization approach in MPI RMA 22

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 distributed context, which enables a wealth of accumulated knowledge in this litature.

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 [2], [3], [4] and more recently [5], [6]. 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 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 powerful enough - its consensus number equals the number of producers [7].

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 slow-down 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 [4].

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 [8].

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.

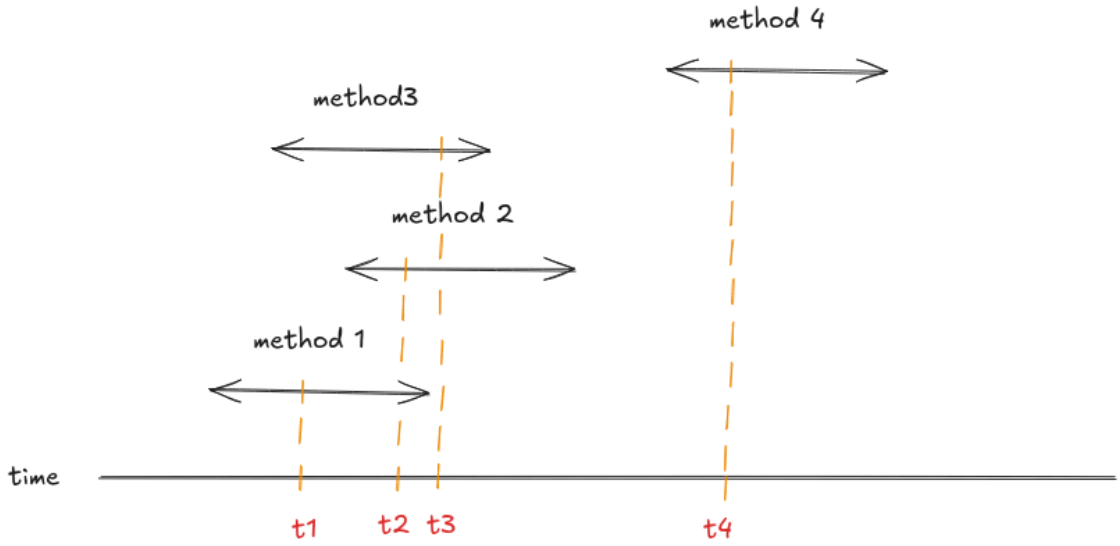

Figure 1: Linerization points of method 1, method 2, method 3, method 4 happens at $t_1 < t_2 < t_3 < t_4$, therefore, their effects will be observed in this order as if we call method 1, method 2, method 3, method 4 sequentially

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 ∞ - which means it's the most powerful atomic instruction [9] (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 [4]) and **hazard pointer** (introduced in [10]).

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** [10] to announce to other processes that some memory is not to be freed.

2.6 C++11 concurrency

2.6.1 Motivation

C++11 came with a lot of improvements. One such improvement is the native support of multithreading inside the C++ standard library (STL). The main motivation was portability and ergonomics along with two design goals: high-level OOP facilities for working with multithreading in general while still exposing enough low-level details so that performance tuning is possible when one wants to drop down to this level. [11]

Before C++11, to write concurrent code, programmers had to resort to compiler-specific extensions [11]. This worked but was not portable as the additional semantics of concurrency introduced by compiler extensions was not formalized in the C++ standard. Therefore, C++11 had come to define a multithreading-aware memory model, which is used to dictate correct concurrent C++11 programs.

2.6.2 C++11 memory model

The C++11 memory model plays the foundational role in enabling native multithreading support. The C++11 memory model is not a syntatical feature or a library feature, rather it's a model to reason about the semantics of concurrent C++11 programs. In other words, the C++11 multithreading-aware memory model enables the static analysis of concurrent C++11 programs. This, in essence, is beneficial to two parties: the compiler and the programmer.

From the compiler's point of view, it needs to translate the source code into correct machine code. Many modern CPUs are known to utilize out-of-order execution, or instruction reordering to gain better pipeline throughput. This reordering is transparent with respect to a single thread - it still observes the effect of the instructions in the program order. However, this reordering is not transparent in concurrent programs, in which case, synchronizing instructions are necessary, so the compiler has to keep this in mind. With the possibility of concurrency, it needs to conservatively apply optimizations as certain optimizations only work in sequential programs. However, optimization is important to achieve performance, if the compiler just disables the any optimizations altogether in the face of concurrency, the performance gained by using concurrency would be adversely affected. Here, the C++11 memory model comes into play. It allows the compiler to reason which optimization is valid and which is not in the presence of concurrency. Additionally, the compiler can reason about where to place synchronizing instructions to ensure the correctness of concurrent operations. Therefore, the C++11 memory allows the compiler to generate correct and performant machine code.

Similarly, from the programmer's point of view, one can verify that their concurrent program's behavior is well-defined and reason whether their programs unnecessarily disable any optimizations. This, helps the programmer to write correct and performant C++11 concurrent programs.

The C++11 memory consists of two aspects: the **structural** aspects and the **concurrency** aspects [11].

2.6.2.1 Structural aspects

The structural aspects deal with how variables are laid out in memory.

An **object** in C++ is defined as "a region of storage". Concurrent accesses can happen to any "region of storage". These regions of storage can vary in size. One can say that there are always concurrent accesses to RAM. However, do these concurrent accesses always cause race conditions? Intuitively, no. To properly define which concurrent accesses can actually cause race conditions, the C++11 memory model defines the concept of **memory location**. That is, the C++11 memory model views an object as one or more **memory locations**. Only concurrent accesses to the same memory location can possibly cause race conditions. Conflicting concurrent accesses to the same memory location (read/write or write-write) always cause race conditions.

The rule of what comprise a memory location is as follows [11]:

- Any object or sub-object (class instance's field) of a scalar type is a memory location.
- Any sequence of adjacent bit fields is also a memory location.

An example: In the below struct, `a` is a memory location, `b` and `c` is another and `d` is the last.

```
struct S {
  int a;
  int b: 8;
  int c: 8;
       : 0;
  int d: 12;
}
```

Listing 1: Example memory locations for a user-defined struct

2.6.2.2 Concurrency aspects

Generally speaking, concurrent accesses to different memory locations are fine while concurrent accesses to the same memory location cause race conditions. However, race conditions do not necessarily cause undefined behavior. To avoid undefined behavior with concurrent accesses to the same memory location, one must use atomic operations. The semantics of C++11 atomics will be discussed in the next section.

2.6.3 C++11 atomics

An atomic operation is an indivisible operation, that is, it either hasn't started executing or has finished executing [11].

Atomic operations can only be performed on atomic types: C++11 introduces the `std::atomic<T>` template type, wrapping around a non-atomic type to allow atomic operations on objects of that type. Additionally, C++11 also introduces the

`std::atomic_flag` type that acts like an atomic flag. One special property of `std::atomic_flag` is that any operations on it is guaranteed to be lock-free, while the others depend on the platform and size.

By C++17, `std::atomic_flag` only supports two operations:

Operation	Usage
`clear`	Atomically sets the flag to `false`
`test_and_set`	Atomically sets the flag to `true` and returns its previous value

Table 1: Supported atomic operations on `std::atomic_flag` (C++17)

Because of its simplicity, `std::atomic_flag` operations are guaranteed to be lock-free.

Some available operations on other atomic types are summarized in the following table [11]:

Operation	`atomic<bool>`	`atomic<T*>`	`atomic <integral-type>`	`atomic <other-type>`
`load`	Y	Y	Y	Y
`store`	Y	Y	Y	Y
`exchange`	Y	Y	Y	Y
`compare_exchange_weak`, `compare_exchange_strong`	Y	Y	Y	Y
`fetch_add`, `+=`		Y	Y	
`fetch_sub`, `-=`		Y	Y	
`fetch_or`, `|=`			Y	
`fetch_and`, `&=`			Y	
`fetch_xor`, `^=`			Y	
`++`, `--`		Y	Y	

Table 2: Available atomic operations on atomic types (C++17)

Each atomic operation can generally accept an argument of type `std::memory_order`, which is used to specify how memory accesses are to be ordered around an atomic operation.

Any atomic operations beside `load` and `store` is called read-modified-write (RMW) operations.

The following is the table of possible `std::memory_order` values:

Name	Usage	Load	Store	RMW
`memory_order_relaxed`	No synchronization imposed on other reads or writes	Y	Y	Y
`memory_order_acquire`	No reads or writes after this operation in the current thread can be reordered before this operation	Y		Y
`memory_order_release`	No reads or writes before this operation in the current thread can be reordered after this operation		Y	Y
`memory_order_acq_rel`	No reads or writes before this operation in the current thread can be reordered after this operation. No reads or writes after this operation can be reordered before this operation			Y
`memory_order_seq_cst`	A global total order exists on all modifications of atomic variables	Y	Y	Y
`memory_order_consume`	Not recommended	-	-	-

Table 3: Available `std::memory_order` values (C++17). On the `Load`, `Store` and `RMW` columns, `Y` means that this memory order can be specified on `load`, `store` and RMW operations, `-` means that we intentionally ignore this entry.

In conclusion, atomic operations avoid undefined behavior on concurrent accesses to the same memory location while memory orders help us enforce ordering of operations accross threads, which can be used to reason about the program.

2.7 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].

2.7.1 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.

2.7.1.1 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.

2.7.1.2 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 [12].

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`.

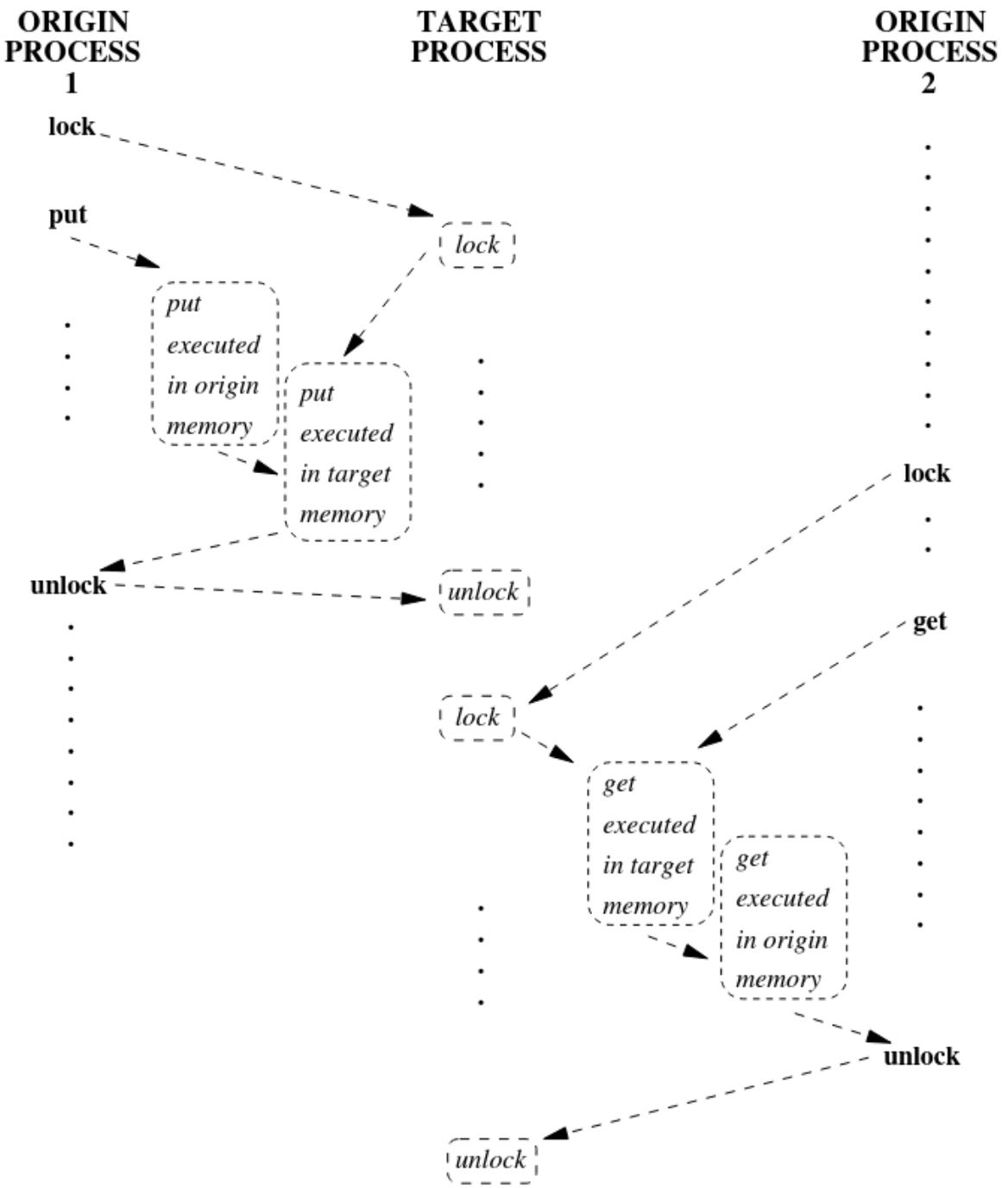

Figure 2: An illustration of passive target communication. Dashed arrows represent synchronization (source: [1])

2.7.2 MPI-3 SHM

Historically, MPI as a message passing framework is often used in combination with other shared-memory frameworks such as OpenMP or pthreads to optimize communication within processes in a node. MPI-3 SHM (shared memory) is a capability introduced in MPI-3 to optimize intra-node communication within MPI RMA windows. This leads to the rise of MPI+MPI approach in distributed programming [13]. In MPI-3, **shared-memory windows** can be created via `MPI_Win_allocate_shared`. Shared memory windows can be used for both one-sided communication and shared memory access. Besides using MPI-RMA facilities for communication and synchronization in these

shared-memory windows, other communication and synchronization mechanisms provided by other shared-memory frameworks such as C++11 atomics can also be used. Typically, C++11 atomics allows for much more efficient communication and synchronization compared to MPI-RMA. Therefore, MPI-3 SHM can be used as an optimization for intra-node communication within MPI RMA programs. A general approach in using shared memory windows with tradition MPI RMA is discussed further in [13].

2.8 Porting shared memory algorithms to distributed context approaches

2.8.1 Pure MPI

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 4: 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 [12], 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 2: An example snippet showcasing our synchronization approach in MPI RMA

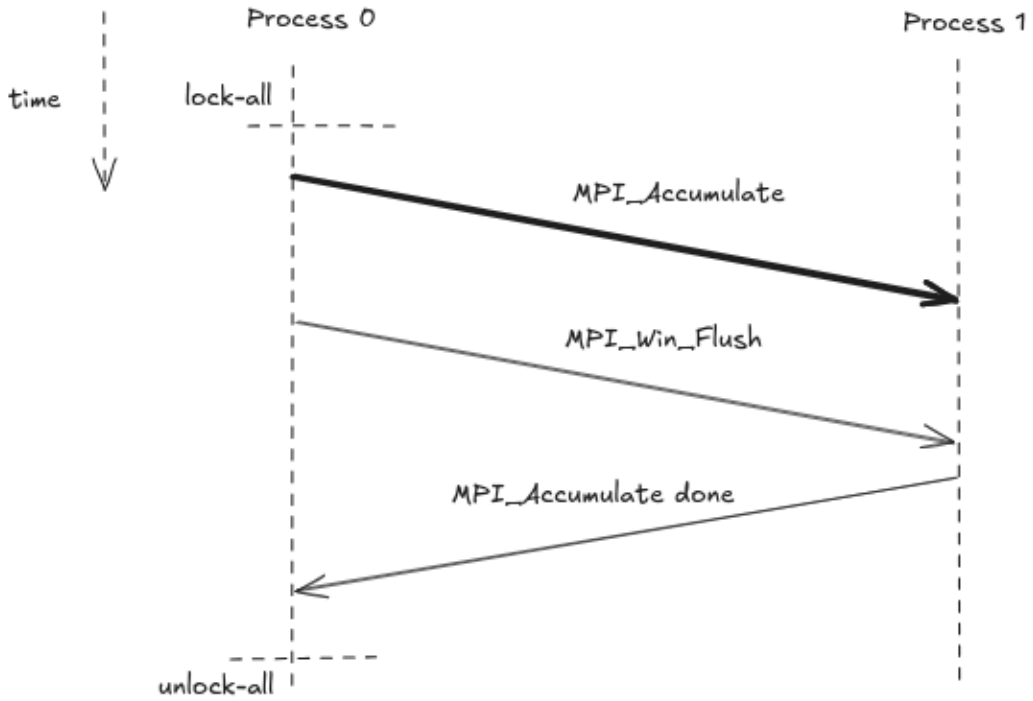

Figure 3: An illustration of our synchronization approach in MPI RMA

2.8.2 MPI+MPI

MPI is highly optimized for inter-node communication, and in recent years, there is also a trend to use MPI both for intra-node communication [13], [14]. MPI-3 has introduced many improvements to MPI RMA to make this scheme feasible. Compared to pure MPI,

MPI+MPI can be more efficient because the fact that some processes locating on the same node is exploited to improve communication.

The general approach is as follows:

1. `MPI_Comm_split_type` is used with `MPI_COMM_TYPE_SHARED` to split the communicator to shared-memory communicator.
2. `MPI_Win_allocate_shared` is called on each shared-memory communicator to obtain a shared-memory window.
3. Inside these shared-memory window, we can use other communication and synchronization primitives that are optimized for shared-memory context.

2.8.3 MPI+MPI with C++11

As discussed in the previous section, we can use C++11 atomics and synchronization facilities inside shared-memory windows. [14] has shown this approach has the potential to obtain significant speedups compared to pure MPI.

Chapter III Related 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.

Chapter IV Distributed queues

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

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

[4] 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.

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

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

[7] 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.

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

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

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

[11] A. Williams, *C++ Concurrency in Action*. Manning, 2019.

[12] 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.

[13] H. Zhou, J. Gracia, and R. Schneider, “MPI Collectives for Multi-core Clusters: Optimized Performance of the Hybrid MPI+MPI Parallel Codes,” 2019, *Association for Computing Machinery*. doi: 10.1145/3339186.3339199.

[14] L. Quaranta and L. Maddegedara, “A novel MPI+MPI hybrid approach combining MPI-3 shared memory windows and C11/C++11 memory model,” 2021, *Academic Press, Inc.* doi: 10.1016/j.jpdc.2021.06.008.