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 DISTRIBUTED BARRIER ALGORITHMS USING THE HYBRID PROGRAMMING MODEL COMBINING MPI-3 AND C++11

Major: Computer Science

THESIS COMMITTEE: 12
MEMBER SECRETARY: NGUYỄN QUANG HÙNG

SUPERVISORS: THOẠI NAM
DIỆP THANH ĐĂNG

—o0o—

STUDENT: PHẠM VÕ QUANG MINH - 2111762

HCMC, 12/2024

Disclaimers

I hereby declare that this specialized project is the result of my own research and experiments. It has not been copied from any other sources. All content presented and implemented within this document reflects my own hard work, dedication, and honesty, conducted under the 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.

All data, references, and sources have been legally cited and are explicitly mentioned in the footnotes and references section.

I accept full responsibility for the accuracy of the claims and content in this specialized project and am willing to face any consequences or penalties should any violations or misconduct be identified.

Acknowledgements

First, I want to express my gratitude to my school and teachers for providing me with a strong foundation in the field of computer science and software engineering. Their guidance has helped me refine my skills and develop my mindset as an engineer and scientist. These experiences have not only supported me in this specialized project but will also guide me throughout my lifelong journey.

I am especially thankful to my mother and brother for their unwavering support during both good times and bad. They have always been there for me, offering emotional and financial support and being the pillars I can rely on when times are tough.

I would also like to extend my deepest gratitude to Mr. Diệp Thanh Đăng for inspiring me and guiding me toward this research topic. His mentorship helped me discover my true passion and led me to choose this meaningful topic for my thesis.

Contents

List of Listings

List of Images

Chapter I: Introduction

Chapter 1 introduces the project's research focus on distributed barrier synchronization algorithms in high-performance computing. Section 1.1. explores the motivation behind the research, highlighting the growing importance of high-performance computing across scientific domains, including large language model training and complex simulations. Particularly, the section emphasizes the critical role of barrier algorithms in synchronizing computational processes across different nodes.

Section 1.2. details the thesis objectives, which encompass a comprehensive exploration of high-performance computing concepts, including MPI-3 and C++11 threading technologies. The research aims to investigate a hybrid programming model that combines the communication strengths of MPI with the synchronization capabilities of C++11. The objectives include surveying existing barrier synchronization algorithms and proposing methods for their deployment on current high-performance computing systems.

The project report structure, outlined in Section 1.3., provides a clear roadmap for the research. The document is organized into six chapters, progressing from foundational concepts to detailed algorithm proposals, experimental results, and future research directions. Each chapter builds upon the previous one, creating a comprehensive narrative of the research journey from introduction to conclusion.

1.1. Motivation

Applications of high-performance computing have gained immense popularity due to its applications in various scientific domains, from simulations of particle movement in physics, weather prediction [1], and it has also gained tremendous popularity with the booming of Large Language Models (LLMs), with the rising demands of distributed machine learning where the models are trained on enormous datasets.

These applications require tremendous effort in communication and synchronization. One of the basic building blocks that is widely used by these applications is the barrier algorithm, which aims to synchronize computation to a point to exchange data between computation nodes.

These barrier algorithms are built upon many programming models. Recently, a new model has gained a lot of attention: a hybrid programming model between MPI and C++11. Specifically, it is a crossover between MPI's power of communicating and synchronizing across different computation nodes, and C++'s power of communicating and synchronizing within one computation node.

A recent research by Quaranta and Maddegedara [2] has shown the potential of this programming model in terms of performance through a newly proposed barrier algorithm. However, they have ignored a large number of existing barrier algorithms that have been designed for similar programming models. [3]

1.2. Objectives

1.2.1. Brief overview

Research and develop distributed barrier synchronization algorithms using the hybrid programming model.

1.2.2. Details

The specific objectives of this report are as follows:
- Explore key concepts in high-performance computing (HPC).
- Study MPI-3 and its programming model.
- Investigate C++11 threading and concurrency features.
- Research the hybrid programming model combining MPI-3 and C++11.
- Survey existing barrier synchronization algorithms in related programming models.
- Analyze and identify promising barrier algorithms for implementation.
- Propose methods to deploy selected barrier algorithms on the current HPC system.

1.3. Thesis structure

Chapter 1: Introduction

Provides an overview of the thesis, including the motivation, objectives, and structure.

Chapter 2: Background

Introduces foundational concepts in HPC, MPI-3, and C++11, focusing on multithreading and synchronization.

Chapter 3: Related Works

Surveys existing research and approaches to barrier synchronization algorithms.

Chapter 4: Algorithm

Details the proposed barrier synchronization methods, including the Brooks barrier algorithm.

Chapter 5: Results

Presents the implementation outcomes, experimental results, and performance analysis.

Chapter 6: Conclusions and Future Works

Summarizes accomplishments, outlines future research directions, and provides a timeline for planned activities.

Chapter II: Background

Chapter 2 provides a comprehensive background on the fundamental technologies underlying the research: Message Passing Interface (MPI) and C++11 Multithreading. The chapter explores these technologies within the context of high-performance computing, laying the groundwork for understanding the hybrid programming model proposed in the thesis.

Section 2.1. delves into MPI, presenting it as a critical message-passing library interface specification. The section goes beyond traditional communication models by highlighting MPI's advanced capabilities, including collective operations, remote-memory access, and parallel I/O. A significant focus is placed on One-Sided Communications, a sophisticated communication mechanism that allows a single MPI process to independently manage communication tasks. The section elaborates on the various Remote Memory Access (RMA) operations and explores two distinct memory models: Separate and Unified Memory Models.

Section 2.2. explores C++11 Multithreading, marking a pivotal advancement in parallel computing. The section demonstrates how C++11 introduced native, platform-independent threading support through its standard library. It comprehensively covers key multithreading features, including thread management, synchronization primitives, atomic operations, and asynchronous programming constructs. The discussion emphasizes the advantages of C++11′s threading model, such as type-safe synchronization and reduced dependency on platform-specific libraries.

The concluding Section 2.3. bridges these technologies within the high-performance computing context. It illustrates how C++11 multithreading and MPI can complement each other: C++11 provides efficient inter-core communication within a single node, while MPI handles communication between different computers in a cluster. This perspective sets the stage for the thesis's exploration of a hybrid programming model that leverages the strengths of both technologies.

By providing this detailed technological background, Chapter 2 establishes the theoretical and practical foundations necessary for understanding the subsequent research on barrier synchronization algorithms in high-performance computing.

2.1. Message Passing Interface (MPI)

In this section, we'll explore Message Passing Interface, its various paradigms for message passing. And then we'll deep dive into its One-Sided Communication mechanics as the focus of this thesis.

2.1.1. Introduction

MPI [4] is a message-passing library interface specification - essentially a set of standard guidelines for both implementors and users of the parallel programming model. In this programming paradigm, data traverses between the address spaces of different processes through cooperative operations - what we might call "hand-shakes" or "classical" message-passing techniques.

Beyond the classical model, MPI extends its capabilities by offering:
- Collective operations
- Remote-memory access operations
- Dynamic process creation
- Parallel I/O

Within this thesis, I will specifically dive deep into remote-memory access operations in MPI, which form the critical building block for MPI's One-Sided Communication, a concept first introduced in MPI-2 [5], and later refined in MPI-3 [6].

2.1.2. One-Sided Communications

Unlike traditional "classical" Point-to-Point communication - where communication is a two-way collaborative effort between processes, with one sending and another receiving - Remote Memory Access introduces a more flexible communication mechanism called One-Sided communication.

In this approach, a single MPI process can now independently orchestrate the entire communication task, specifying communication parameters for both the sending and receiving sides.

Regular send/receive communication requires a matching operations by sender and receiver. And to issue the matching operations, the application needs to distribute the transfer parameters. This may require all MPI processes to participate in a global computation, or to explicitly poll for potential communication requests to response periodically.

The use of RMA communication operations avoids the need for global computation or explicit polling.

Message-passing in general achieves two effects: communication of data from sender to receiver and synchronization of sender with receiver. RMA design separates these two functions. Within MPI standards, the primitive communication operations are as follows:

- Remote write operations: MPI_PUT, MPI_RPUT
- Remote read operations: MPI_GET, MPI_RGET
- Remote update operations: MPI_ACCUMULATE, MPI_RACCUMULATE
- Combined read and update operations: MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP
- Remote atomic swap: MPI_COMPARE_AND_SWAP

2.1.3. Memory Model

MPI supports two distinct memory models:

1. Separate Memory Model:
   - Highly portable
   - No inherent memory consistency assumptions
   - Similar to weakly coherent memory systems
   - Requires explicit synchronization for correct memory access ordering
2. Unified Memory Model:
   - Exploits cache-coherent hardware
   - Supports hardware-accelerated one-sided operations
   - Typically found in high-performance computing environments

The RMA design's flexibility allows implementors to leverage platform-specific communication mechanisms, including:
- Coherent and non-coherent shared memory
- Direct Memory Access (DMA) engines
- Hardware-supported put/get operations
- Communication coprocessors

While most RMA communication mechanisms can be constructed atop message-passing infrastructure, certain advanced RMA functions might necessitate support from asynchronous communication agents like software handlers or threads in distributed memory environments.

Terminology clarification:
- Origin (or origin process): The MPI process initiating the RMA procedure
- Target (or target process): The MPI process whose memory is being accessed

In a put operation: source = origin, destination = target

In a get operation: source = target, destination = origin

2.2. C++11 Multithreading

C++11 (ISO/IEC 14882:2011) introduced native support for multithreading directly in the Standard Template Library (STL) [7]. This eliminated the previous reliance on platform-specific threading libraries like POSIX thread (pthread) and the Microsoft

Windows API, offering a more portable and standardized approach to parallel computing. The introduction of these features marked a significant advancement in C++ development, providing developers with powerful, standardized tools for concurrent and parallel programming.

2.2.1. Core Components

The STL threading components include:

- Thread management (`<thead>`)
- Mutual exclusion (`<mutex>`)
- Condition variables (`<condition_variable>`)
- Atomic operations (`<atomic>`)
- Futures and promises (`<future>`)

2.2.2. Thread Management

The `<thread>` header has now become the pivot of C++11′s multithreading support. It provides a lightweight, platform-independent mechanism for creating and managing threads

2.2.2.1. Creating Threads

```cpp
#include <thread>
#include <iostream>

void worker_function() {
    std::cout << "Thread is running" << std::endl;
}

int main() {
    std::thread t(worker_function);  // Create a thread
    t.join();  // Wait for the thread to complete
    return 0;
}
```

Listing 1: Basic thread creation in C++11

Important characteristics of `std::thread`:

- Non-copyable but movable
- Must be either joined or detached before destruction, or else `std::terminate()` will be called, resulting in immediate termination of the entire application. This is a safety feature to prevent accidental thread abandonment.
- Supports various callable objects (functions, lambdas, function objects)

2.2.2.2. Safe Thread Joining

When exceptions occur in the parent thread, proper thread management becomes crucial. To prevent unexpected program termination, we can use RAII patterns or C++20's `std::jthread`.

```cpp
#include <thread>
#include <stdexcept>

class thread_guard {
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_) : t(t_) {}
    ~thread_guard() {
        if(t.joinable()) {
            t.join();
        }
    }
    thread_guard(const thread_guard&) = delete;
    thread_guard& operator=(const thread_guard&) = delete;
};

void may_throw() {
    throw std::runtime_error("Exception occurred");
}

void worker_function() {
    // Thread work here
}

int main() {
    std::thread t(worker_function);
    thread_guard g(t);  // RAII wrapper ensures join
    try {
        may_throw();
    }
    catch (...) {
        // Thread will be automatically joined
        throw;
    }
    return 0;
}
```

Listing 2: RAII-based safe thread joining

2.2.3. Race Condition Prevention

C++11 provides several RAII-compliant mechanisms to handle mutual exclusion and prevent race conditions

2.2.3.1. Basic Mutex Operations

Mutexes (mutual exclusion objects) provide the most fundamental way to protect shared data. While direct lock/unlock operations are available, they should be used with caution as they don't provide automatic cleanup in case of exceptions.

```
std::mutex mtx;
mtx.lock();    // Acquire lock
// Critical section
mtx.unlock();  // Release lock
```

Listing 3: Basic Mutex in C++11

2.2.3.2. RAII Lock Guards

RAII (Resource Acquisition Is Initialization) lock guards provide a safer alternative to manual mutex locking and unlocking. They automatically release the mutex when going out of scope, preventing resource leaks and deadlocks.

```
#include <mutex>

class shared_resource {
    std::mutex mtx;
    int data;
public:
    void safe_operation() {
        std::lock_guard<std::mutex> lock(mtx);
        // Protected operations here
    } // Automatically unlocked
};
```

Listing 4: RAII-based mutex locking

2.2.3.3. Deadlock Prevention

Deadlocks occur when multiple threads are waiting for each other to release resources. C++11 provides `std::lock` to acquire multiple mutexes simultaneously using a deadlock avoidance algorithm. Later, C++17 introduced `std::scoped_lock` which combines this functionality with RAII principles, making it safer and more convenient to acquire multiple mutexes simultaneously while preventing deadlocks.

```cpp
#include <mutex>

class complex_resource {
    std::mutex mtx1, mtx2;
public:
    void safe_operation() {
        std::scoped_lock lock(mtx1, mtx2); // Deadlock-free
        // Protected operations here
    }
};
```

Listing 5: Deadlock prevention using scoped_lock

2.2.3.4. Condition Variables

Condition variables enable threads to coordinate based on actual values or conditions rather than simply taking turns.

```cpp
#include <condition_variable>
#include <mutex>
#include <queue>

std::queue<int> data_queue;
std::mutex mtx;
std::condition_variable cv;

void producer() {
    std::lock_guard<std::mutex> lock(mtx);
    data_queue.push(42);
    cv.notify_one();
}

void consumer() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, []{ return !data_queue.empty(); });
    int data = data_queue.front();
    data_queue.pop();
}
```

Listing 6: Producer-consumer pattern using condition variables

2.2.4. Atomic Operations

Atomic operations provide lock-free programming capabilities, ensuring that certain operations are performed as a single, uninterruptible unit.

```cpp
#include <atomic>

struct atomic_counter {
    std::atomic<int> value{0};

    void increment() {
        value.fetch_add(1, std::memory_order_relaxed);
    }

    bool compare_exchange(int expected, int desired) {
        return value.compare_exchange_strong(expected, desired,
                                             std::memory_order_acq_rel);
    }

    int load() const {
        return value.load(std::memory_order_acquire);
    }
};
```

Listing 7: Atomic operations with memory ordering

2.2.5. Future and Promise

C++11 provides powerful mechanisms for asynchronous programming:

```cpp
#include <future>
#include <stdexcept>

std::promise<int> compute_promise;

void compute() {
    try {
        int result = /* complex computation */42;
        compute_promise.set_value(result);
    } catch (...) {
        compute_promise.set_exception(std::current_exception());
    }
}

int main() {
    std::future<int> result = compute_promise.get_future();
    std::thread t(compute);

    try {
        int value = result.get(); // Will throw if computation failed
        std::cout << "Result: " << value << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Computation failed: " << e.what() << std::endl;
    }

    t.join();
    return 0;
}
```

Listing 8: Asynchronous computation with future and promise

2.2.6. Key Advantages

- Platform-independent threading
- Standard library support
- Type-safe synchronization
- Reduced dependency on platform-specific libraries
- Simplified concurrent programming model

The introduction of these features provided developers with powerful, standardized API for concurrent and parallel programming without resorting to platform-specific libraries like pthread.

2.3. HPC Context

In the landscape of high-performance computing (HPC), C++11 multithreading provides a standardized approach to shared-memory parallel processing. When considering

hybrid parallelization strategies in modern HPC environments, C++11′s threading capabilities play a crucial role alongside distributed computing protocols.

To illustrate the complementary relationship between C++11 multithreading and Message Passing Interface (MPI), consider a typical HPC cluster architecture:

- Within a single computation node: **C++11 multithreading provides efficient shared-memory parallelism across CPU cores** Thread communication occurs through direct memory access **Synchronization is handled through native C++11 primitives (mutex, atomic operations)** Zero-copy data sharing is possible between threads
- Between cluster nodes: **MPI handles inter-node communication through message passing** Network infrastructure facilitates data transfer **Explicit data serialization and communication required** Each node runs independent processes

This hierarchical approach to parallelism, often called hybrid parallelization, leverages the strengths of both paradigms:
1. C++11 threads efficiently utilize shared memory within a node
2. MPI manages distributed memory communication across the cluster
3. The combination can potentially reduce the overall MPI process count
4. Memory footprint can be optimized by sharing data between threads

This makes C++11′s threading library particularly effective in modern HPC applications, where it serves as the intra-node parallelization layer in hybrid MPI+threads programs.

Chapter III: Related Works

This section examines two key areas relevant to my implementation: barrier synchronization algorithms and hybrid parallelization approaches.

Section 3.1. review MellorCrummey's foundational work on barrier algorithms, which provides crucial insights into synchronization strategies for different multiprocessor architectures.

Section 3.2. explore recent advances in hybrid parallelization, particularly Quaranta and Maddegedara's novel approach combining MPI-3 shared memory windows with the C++11 memory model.

Based on these works, Section 3.3. formulate the implementation strategy that leverages both efficient barrier synchronization and modern memory model capabilities.

3.1. Barrier Synchronization Algorithm Selection

MellorCrummey et al [8] proposed nuanced recommendations for barrier synchronization:

1. Broadcast-Based Cache-Coherent Multiprocessors:
   - For modest processor counts: Utilize a centralized counter-based barrier
   - For larger scales: Implement a 4-ary arrival tree with a central sense-reversing wakeup flag
2. Multiprocessors Without Coherent Caches or With Directory-Based Coherency:
   - Dissemination Barrier [9]:
     - Distributed data structures respecting locality
     - Critical path approximately one-third shorter than tree-based barriers
     - Total interconnect traffic complexity of O(P logP)
   - Tree-Based Barrier [8]:
     - Alternative approach with different performance characteristics
     - Total interconnect traffic complexity of O(P)

The dissemination barrier demonstrates superior performance on architectures like the Butterfly, which can execute parallel non-interfering network transactions across multiple processors.

3.2. Hybrid Parallelization Approaches

Quaranta and Maddegedara [2] proposed a novel hybrid approach combining MPI-3 shared memory windows with C11/C++11 memory model. Their work demonstrates several key advantages:

- Efficient intranode communication using MPI shared memory windows
- Fine-grained synchronization using C++11 atomic operations
- Reduced variance in execution times
- Enhanced synchronization between processes, especially in multi-node environments
- Significant performance improvements in ghost updates compared to flat MPI
- More efficient synchronization of shared data compared to RMA-based methods

3.3. Design Decisions

From these works, my proposal for a barrier algorithm within a computation node like C++ will use the broadcast-based cache-coherent multiprocessors method. For inter-core communication, I will implement the second approach. This choice aligns with the expectation that modern multi-core processors typically maintain cache coherence. The implementation will also consider the synchronization techniques demonstrated by Quaranta and Maddegedara, particularly their use of C++11 atomic operations for fine-grained control.

Chapter IV: Algorithm

This chapter presents Brook's barrier algorithm within the context of the shared memory model (Section 4.1.).

Section 4.2. provides a straightforward implementation of Brook's two-process barrier algorithm using MPI's Remote Memory Access (RMA) operations.

Finally, Section 4.3. offers an in-depth analysis of my implementation, highlighting its key components and comparing it to the shared memory model.

4.1. Brook Algorithm

Brook [10] bases the n-process barrier on a two-process barrier using two shared variables. The algorithm is as follows:

Step	Process 1	Process 2
1	while SetByProcess1 do wait;	while SetByProcess2 do wait;
2	SetByProcess1 := true;	SetByProcess2 := true;
3	while not SetByProcess2 do wait;	while not SetByProcess1 do wait;
4	SetByProcess2 := false;	SetByProcess1 := false;

4.2. My proposed implementation of Brook's algorithm

We can extend the concept of Brooks' two-process barrier, where two processes share memory through shared variables, to the one-sided communication model.

In this model, one process can directly access the variables of another process. Instead of storing the shared variables in a common memory segment, each process maintains its own copy, allowing the other process to read and modify it using one-sided communication primitives.

This is my simple implementation of Brook's two-process barrier technique

```
#include "mpi.h"

int brook_2_proc(const MPI_Comm &comm) {
  // Get number of ranks and current rank
  int n_ranks, rank;
  MPI_Comm_size(comm, &n_ranks);
  MPI_Comm_rank(comm, &rank);

  // Ensure exactly 2 processes are used
  if (n_ranks != 2) {
    if (rank == 0) {
      fprintf(stderr, "This program requires exactly 2 processes.\n");
    }
    MPI_Abort(comm, 1);
  }

  // Initialize shared window
  bool exposed_buffer{false};
  MPI_Win win_buffer_handler;
  MPI_Win_create(&exposed_buffer, sizeof(bool), sizeof(bool),
                 MPI_INFO_NULL, comm, &win_buffer_handler);
```

```
  // Barrier synchronization
  // Step 1: Wait for my exposed buffer to be reset
  while (exposed_buffer) {
    // Busy-waiting loop
  }

  // Step 2: Set my exposed buffer to true
  exposed_buffer = true;

  // Step 3: Wait for the other process's exposed buffer to be true
  bool flag_from_other_process{false};
  int target_rank = 1 - rank;
  while (!flag_from_other_process) {
    MPI_Win_lock_all(0, win_buffer_handler);
    MPI_Get_accumulate(&flag_from_other_process, 0, MPI_CXX_BOOL,
                       &flag_from_other_process, 1, MPI_CXX_BOOL,
                       target_rank, 0, 1, MPI_CXX_BOOL,
                       MPI_NO_OP, win_buffer_handler);
    MPI_Win_flush_all(win_buffer_handler);
    MPI_Win_unlock_all(win_buffer_handler);
  }

  // Step 4: Reset the other process's exposed buffer
  bool false_value{false};
  MPI_Win_lock_all(0, win_buffer_handler);
  MPI_Accumulate(&false_value, 1, MPI_CXX_BOOL,
                 target_rank, 0, 1, MPI_CXX_BOOL,
                 MPI_REPLACE, win_buffer_handler);
  MPI_Win_flush(target_rank, win_buffer_handler);
  MPI_Win_unlock_all(win_buffer_handler);

  // Cleanup
  return MPI_Win_free(&win_buffer_handler);
}
```

Listing 9: My implementation of Brook's two-process barrier using MPI primitives

4.3. Commentary on Implementation

We analyze the implementation step by step, comparing it with Brook's shared memory model:

4.3.1. Memory Setup

In Brook's shared memory model:
- Two variables are shared between processes
- Both processes can directly access these variables

- No explicit initialization needed beyond variable declaration

In my MPI implementation:
- Each process owns a local buffer (`exposed_buffer`)
- MPI Window creation establishes remote memory access capability
- Explicit initialization required through `MPI_Win_create`

4.3.2. Step 1: Wait for Reset

Brook's model:

```
while SetByProcess1 do wait;  // Direct memory access
```

My implementation:

```
while (exposed_buffer) {
    // Busy-waiting loop
}
```

The implementation directly checks the local buffer since each process owns its buffer. This is simpler than Brook's model as we're checking local memory.

4.3.3. Step 2: Set Flag

Brook's model:

```
SetByProcess1 := true;  // Direct shared memory write
```

My implementation:

```
exposed_buffer = true;  // Local memory write
```

Similar to Step 1, we're working with local memory, making this operation straightforward.

4.3.4. Step 3: Wait for Other Process

Brook's model:

```
while not SetByProcess2 do wait;  // Direct memory read
```

My implementation:

```
while (!flag_from_other_process) {
    MPI_Win_lock_all(0, win_buffer_handler);
    MPI_Get_accumulate(&flag_from_other_process, 0, MPI_CXX_BOOL,
                       &flag_from_other_process, 1, MPI_CXX_BOOL,
                       target_rank, 0, 1, MPI_CXX_BOOL,
                       MPI_NO_OP, win_buffer_handler);
    MPI_Win_flush_all(win_buffer_handler);
    MPI_Win_unlock_all(win_buffer_handler);
}
```

Key differences:
- Uses `MPI_Get_accumulate` instead of simple `MPI_Get` to ensure atomic operations
- The operation is atomic, preventing potential race conditions
- Requires explicit synchronization through window locks and flushes
- More complex due to RMA operation setup and synchronization

4.3.5. Step 4: Reset Other's Flag

Brook's model:

```
SetByProcess2 := false;  // Direct shared memory write
```

My implementation:

```
bool false_value{false};
MPI_Win_lock_all(0, win_buffer_handler);
MPI_Accumulate(&false_value, 1, MPI_CXX_BOOL,
               target_rank, 0, 1, MPI_CXX_BOOL,
               MPI_REPLACE, win_buffer_handler);
MPI_Win_flush(target_rank, win_buffer_handler);
MPI_Win_unlock_all(win_buffer_handler);
```

Key differences:
- Uses `MPI_Accumulate` instead of `MPI_Put` for atomic operation
- Requires explicit synchronization similar to Step 3
- More complex due to RMA operation requirements

4.3.6. Use of Atomic Operations

The implementation uses `MPI_Get_accumulate` and `MPI_Accumulate` instead of `MPI_Get` and `MPI_Put` for two critical reasons:

1. Atomicity Guarantee:
   - These operations are guaranteed to be atomic by the MPI specification
   - Prevents race conditions when multiple processes access the same memory location

- Ensures consistency of memory operations

2. Memory Ordering:
   - Atomic operations provide stronger memory ordering guarantees
   - Help maintain the happens-before relationship between operations
   - Essential for correct barrier synchronization behavior

Chapter V: Preliminary Results

This chapter showcases the preliminary test result of the Brook's two-process barrier algorithm using MPI's RMA operations.

5.1. Test Environment

- Hardware: Apple M1 chip
- Compiler: MPICH's mpic++ compiler
- MPI Implementation: MPICH
- Compilation flags: -std=c++11
- Operating System: macOS

5.2. Test Implementation

The test program was designed to verify the basic functionality of the barrier synchronization:

```
int main(int argc, char **argv) {
  // init the mpi world
  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  int rank;
  MPI_Comm_rank(comm, &rank);
  brook_2_proc(comm);
  printf("Process %d: reached destination\n", rank);
  return MPI_Finalize();
}
```

Listing 10: Main function to test brook_2_proc function

5.3. Compilation and Execution

The program was compiled using the following command:

```
mpic++ -std=c++11 src/brook.cpp -o brook
```

Execution was performed using MPICH's mpirun with two processes:

```
mpirun -np 2 ./brook
```

5.4. Results

The test results demonstrated successful barrier synchronization between two processes:

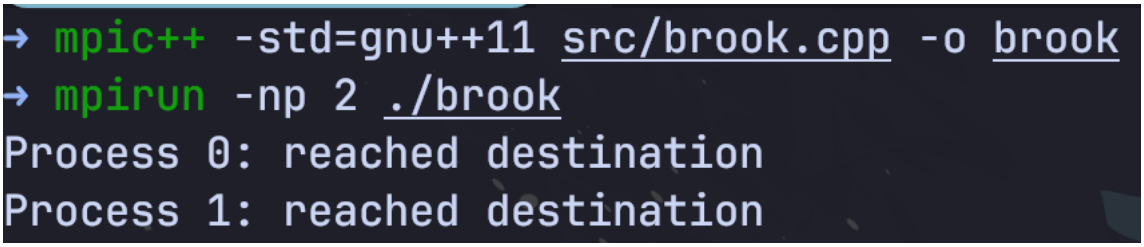

Figure 1: Test results on Apple M1

Chapter VI: Conclusions and Future Works

This chapter presents the accomplishments and future direction of the research.

Section 6.1. highlights the progress made this semester, including the implementation of Brook's two-process barrier algorithm using MPI's RMA operations.

Section 6.2. outlines the planned research trajectory, focusing on further algorithm development, benchmarking, and integration into larger systems. This chapter concludes with a Gantt chart timeline illustrating the key milestones and activities for the upcoming semester.

6.1. Accomplishments This Semester

Within the confines of this semester, I have:

- Studied and familiarized myself with MPI and its One-Sided Communication techniques
- Explored C++11 threading and concurrency features
- Successfully implemented the two-process Brooks barrier algorithm utilizing MPI's Remote Memory Access (RMA) Operations

6.2. Planned Research Trajectory

For the upcoming semester, my research will focus on:

- Understanding of C++ and MPI Memory model to implement C++ and MPI Hybrid model
- Extending barrier algorithm implementations using MPI's RMA Operations and One-Sided Communication techniques
- Implement barrier algorithm on C++ shared memory model
- Testing of proposed algorithms on existing computational clusters
- Performance benchmarking to identify and select optimal barrier synchronization strategies
- Integration and contribution to existing applications

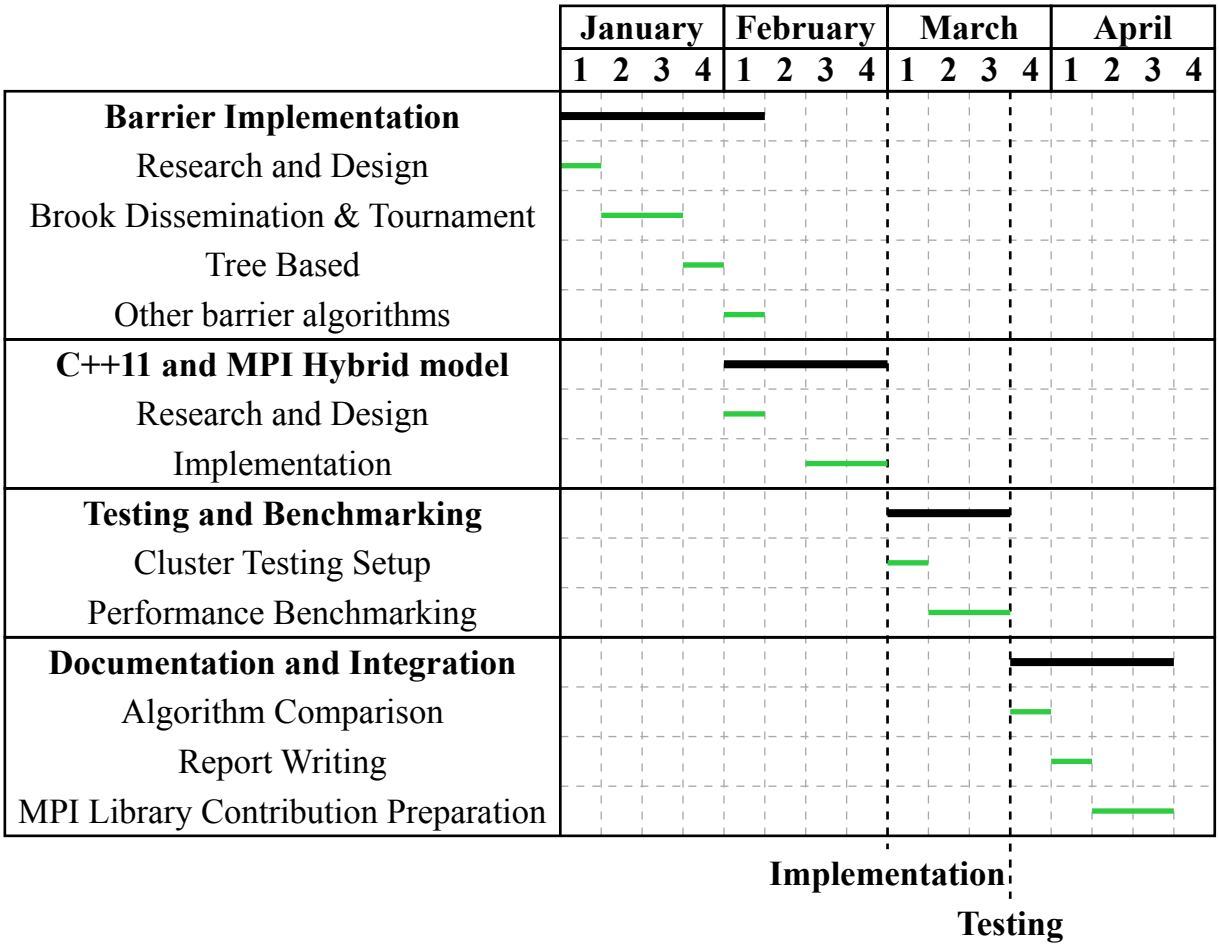

Figure 2: Gantt chart timeline.

References

[1] "High Performance Computing (HPC) Applications and Examples," Nov. 16, 2023. Accessed: Nov. 26, 2024. [Online]. Available: https://phoenixnap.com/kb/hpc-applications

[2] L. Quaranta and L. Maddegedara, "A Novel MPI+MPI Hybrid Approach Combining MPI-3 Shared Memory Windows and C11/C++11 Memory Model," *Journal of Parallel and Distributed Computing*, vol. 157, pp. 125–144, Nov. 2021, doi: 10.1016/j.jpdc.2021.06.008.

[3] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming*. Morgan Kaufman.

[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1*.

[5] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1*. 2003. Accessed: Nov. 26, 2024. [Online]. Available: https://www.mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf

[6] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1*. 2012. Accessed: Nov. 26, 2024. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

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

[8] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *Association for Computing Machinery*. doi: 10.1145/103727.103729.

[9] "An Optimal Scheme for Disseminating Information," University of Kentucky, Department of Computer Science.

[10] E. D. Brooks, "The butterfly barrier," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 295–307, Aug. 1986, doi: 10.1007/BF01407877.