VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

THESIS PROPOSAL

STUDYING AND DEVELOPING DISTRIBUTED BARRIER ALGORITHM USING THE HYBRID PROGRAMMING MODEL COMBINING MPI-4.1 AND C++11

Major: Computer Science

SUPERVISOR(S): THOẠI NAM
DIỆP THANH ĐĂNG
STUDENTS: PHẠM VÕ QUANG MINH - 2111762

HCMC, 12/2024

Disclaimers

I hereby declare that this thesis proposal 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 thesis 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 thesis 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 figures

List of tables

1 Introduction

1.1 Motivation

Applications of high-performance computing have gained tremendous popularity due to its applications in various scientific domains, from simulations of particle movement in physics, weather prediction, and it has gained tremendous popularity with the booming of Large Language Models, 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 are widely used by these applications are the barrier algorithms, which aim 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 et al. [1] 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.

Hence, the goal of this thesis is to research and develop distributed barrier synchronization algorithms using the hybrid programming model.

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 thesis are as follows:

- Explore key concepts in high-performance computing (HPC).
- Study MPI-4.1 and its programming model.
- Investigate C++11 threading and concurrency features.
- Research the hybrid programming model combining MPI-4.1 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-4.1, 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.

References

Lists all cited works.

2 Background

2.1 Message Passing Interface (MPI)

2.1.1 MPI

MPI 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, we’ll 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.

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

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 introduced native support for multithreading directly in the standard library. This eliminated the previous reliance on platform-specific threading libraries like pthread, offering a more portable and standardized approach to parallel computing.

2.2.1 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:

```
#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;
}
```

2.2.2 Synchronization Primitives

C++11 introduced robust synchronization mechanisms:

1. Mutex Operations

```
#include <mutex>

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

2. Condition Variables

```
#include <condition_variable>

std::condition_variable cv;  // Allows thread synchronization
std::mutex cv_mutex;
```

2.2.3 Atomic Operations

The `<atomic>` header provides lock-free concurrency primitives:

```
#include <atomic>

std::atomic<int> counter(0);  // Thread-safe integer
counter++;  // Atomic increment
```

2.2.4 Future and Promise

C++11 introduced powerful asynchronous programming constructs:

```cpp
#include <future>

std::future<int> result = std::async(std::launch::async, []() {
    return 42;  // Compute something asynchronously
});
```

2.2.5 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 marked a pivotal moment in C++ development, providing developers with powerful, standardized tools for concurrent and parallel programming without resorting to platform-specific libraries like pthread.

2.3 HPC Context

In the landscape of high-performance computing, C++11 multithreading offers a nuanced approach to parallel processing. Consider a typical high-performance computing cluster: each node typically comprises one or more multi-core processors.

Here’s where the library shines:

- Within a single personal computer (or computation node), C++11 multithreading provides an efficient mechanism for inter-core communication
- For communication between different computers in a cluster, Message Passing Interface (MPI) remains the preferred approach

This makes C++11′s threading library particularly effective for parallelizing computations within a single, multi-core system, complementing MPI’s inter-node communication capabilities.

3 Related Works

3.1 Barrier algorithms

MellorCrummey et al [2] propose 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:
     - 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:
     - 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.

From this, 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.

4 Algorithm

4.1 Brook Algorithm

Brooks [3] bases the n-process barrier on a two-process barrier using two shared variables. Its 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;

We can extend the concept of Brooks' two-process barrier, where two processes share memory through shared variables, to a 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

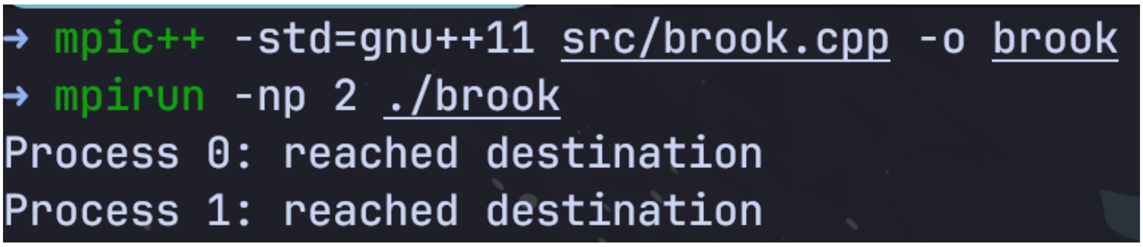
```
#include "mpi.h"
#include "mpi_proto.h"
#include <cstdio>
#include <stdio.h>

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

  if (n_ranks != 2) {
    if (rank == 0) {
      fprintf(stderr, "This program requires exactly 2 processes.
\n");
    }
    MPI_Abort(comm, 1);
  }
  bool exposed_buffer{false};
  bool false_value = false;
  bool true_value = true;
  MPI_Win win_buffer_handler;
  MPI_Win_create(&exposed_buffer, sizeof(bool), sizeof(bool),
MPI_INFO_NULL,
                 comm, &win_buffer_handler);
  // barrier
  // step 1: wait for other process to set my exposed buffer to false
  while (exposed_buffer) {
    // busy waiting
  }

  // step 2: set my exposed buffer to true
  // set my exposed buffer to true
  exposed_buffer = true;

  // step 3: wait for the other process to set their exposed buffer
to true
  bool flag_from_other_process = false;
  int target_rank = 1 - rank;
  while (!flag_from_other_process) {
    // get exposed buffer from the other process
    MPI_Win_lock_all(0, win_buffer_handler);
    MPI_Get(&flag_from_other_process, 1, MPI_CXX_BOOL, target_rank,
0, 1,
            MPI_CXX_BOOL, win_buffer_handler);
    MPI_Win_flush_all(win_buffer_handler);
    MPI_Win_unlock_all(win_buffer_handler);
  }

  // step 4: set their exposed buffer to false
  MPI_Win_lock_all(0, win_buffer_handler);
  MPI_Put(&false_value, 1, MPI_CXX_BOOL, target_rank, 0, 1,
MPI_CXX_BOOL,
          win_buffer_handler);
  MPI_Win_flush(target_rank, win_buffer_handler);
  MPI_Win_unlock_all(win_buffer_handler);
  // end barrier
  return MPI_Win_free(&win_buffer_handler);
};

int main(int argc, char **argv) {
  // init the mpi world
  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  int rank;
```

```c
  MPI_Comm_rank(comm, &rank);

  brook_2_proc(comm);

  printf("Process %d: reached destination\n", rank);

  return MPI_Finalize();
}
```

5 Results

I compiled the program using mpich's mpic++ and ran it with mpich's mpirun

The Brook algorithm ran successfully on my Apple M1 chip. The result is as follow:



```
→ mpic++ -std=gnu++11 src/brook.cpp -o brook
→ mpirun -np 2 ./brook
Process 0: reached destination
Process 1: reached destination
```

Figure 1: Test on Apple M1

6 Conclusions and Future Works

6.1 Accomplishments This Semester

Within the confines of this semester, I have:

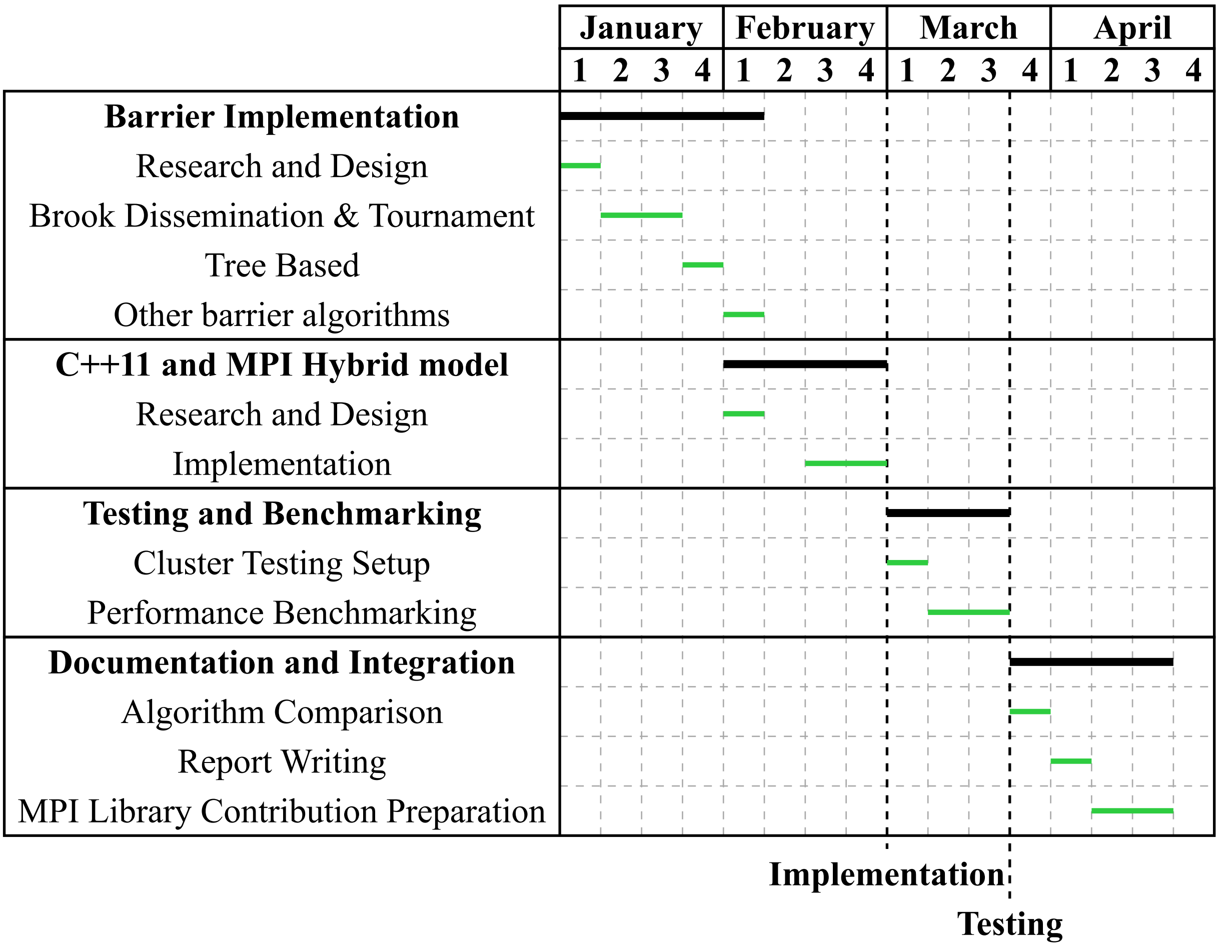
- Studied and familiarized myself with MPI and its One-Sided Communication techniques
- Explored C++11 threading and concurrency features
- Conducted in-depth research on barrier synchronization algorithms for shared memory models
- Successfully implemented the two-process Brooks barrier algorithm utilizing MPI's Remote Memory Access (RMA) Operations

6.2 Planned Research Trajectory

For the upcoming semester, my research will focus on:

- Extending barrier algorithm implementations using MPI's RMA Operations and One-Sided Communication techniques
- Implements barrier algorithm on C++ shared memory model
- Implement C++ and MPI Hybrid model
- Comprehensive testing of proposed algorithms on existing computational clusters
- Rigorous performance benchmarking to identify and select optimal barrier synchronization strategies
- Potential integration and contribution to existing applications

6.3 Gant chart timeline


January
February
March
April
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Barrier Implementation
Research and Design
Brook Dissemination & Tournament
Tree Based
Other barrier algorithms
C++11 and MPI Hybrid model
Research and Design
Implementation
Testing and Benchmarking
Cluster Testing Setup
Performance Benchmarking
Documentation and Integration
Algorithm Comparison
Report Writing
MPI Library Contribution Preparation
Implementation
Testing

References

[1] 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.

[2] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *Association for Computing Machinery*. [Online]. Available: https://doi.org/10.1145/103727.103729

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