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

- Study high performance computing
- Study programming model MPI-4.1
- Study C++11
- Study the model MPI-4.1 and C++11
- Survey barrier algorithms on similar programming models.
- Analyze and choose potentially good barrier algorithm
- Propose methods to deploy the barrier algorithms on the current HPC system we have.

1.3 Thesis structure

Chapter 1

Chapter 2 Chapter 3 Chapter 4 Chapter 5

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.

2.2.6 Computational 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 designed for shared memory system

Since

4 Algorithm

4.1 Brook Algorithm

Chuong 4 Noi ve giai thuat

- Hien thuc dung MPI

5 Results

Chuong 5 Ket qua so bo

- Show ket qua

6 Conclusions and Future Works

- Tom tat lai da lam duoc nhung gi
- timeline tiep theo (gantt chart,…)

Bibliography

[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.