Saint Petersburg Electrotechnical University
(ETU)

Education profile	09.04.01 Computer Science and Engineering
Education program	Computer Science and Knowledge Discovery
Faculty	Computer Science and Technology
Department	Information Systems

Accepted for defense

Head of the department V. V. Tcehanovsky

MASTER'S
GRADUATE QUALIFICATION WORK

Topic: ANALYSIS OF THREAD SYNCHRONIZATION EFFICIENCY IN HYBRID MPI+THREADS PARALLEL PROGRAMS

Student		_____ *signature*	Xinyu Qian
Supervisor	(PhD. Associate Professor)	_____ *signature*	A. A. Paznikov
Advisors	(D.E. Prof.)	_____ *signature*	I. V. Medynskaya
		_____ *signature*	O. I. Brikova
	(PhD. Associate Professor)	_____ *signature*	N. A. Nazarenko

Saint Petersburg

2023

TASK FOR THE GRADUATE QUALIFICATION WORK

Approved

Head of the department FKTI

_____ V. V. Tcehanovsky

«___»_____20___ .

Student Xinyu Qian Group 7300

Topic: Analysis of thread synchronization efficiency in hybrid MPI+threads parallel programs

Institution: Saint Petersburg Electrotechnical University (LETI)

Initial data (technical requirements):

Evaluate the thread synchronization efficiency under the hybrid model

Design parallel programs

Contents:

Theoretical basis of parallel computing

Hybrid MPI+Threads Programming model

Implementation and Evaluation of hybrid MPI+ threads programming

List of reporting materials: the text of the GQW, illustrations, other reporting materials

Additional sections: Commercialization of the Research's Results

The task was given

«___»_____20___.

The GQW was submitted for the defense

«___»_____20___ .

Student _____ Xinyu Qian

Supervisor _____ A. A. Paznikov

(PhD. Associate Professor)

CALENDER PLAN FOR THE GRADUATE QUALIFICATION WORK

Approved

Head of the department FKTI

_____ V. V. Tcehanovsky

«___»_____20___ .

Student Xinyu Qian Group 7300

Topic: Analysis of thread synchronization efficiency in hybrid MPI+threads parallel programs

№	Stages	Deadline
1	Bibliography review	15.03 – 01.04
2	Theoretical basis of parallel computing	02.04 – 08.04
3	Hybrid MPI+Threads Programming model	09.04 – 15.04
4	Implementation and Evaluation of hybrid MPI+ threads programming	16.04 – 01.05
5	Commercialization of the Research's Results	02.05 – 05.05
6	Conclusion	06.05 – 08.05

Student _____ Xinyu Qian

Supervisor _____ A. A. Paznikov

(PhD. Associate Professor)

SUMMARY

Explanatory note 83 p., 37 fig., 9 tab., 42 sources.

KEY WORDS: MPI, OPENMP, PTHREAD, HYBRID MODEL, THREAD SYNCHRONIZATION, PARALLELIZATION, MULTITHREADED MPI COMMUNICATION, SHARED MEMORY, DISTRIBUTED COMPUTING

The subject of the research is: Analysis of thread synchronization efficiency in hybrid MPI+threads parallel programs

The target of the GQW – Evaluate the thread synchronization efficiency under the hybrid model, optimize parallel communication and computation processes.

The field of high-performance computing (HPC) is an important tool for contemporary scientific research and engineering computing. As the complexity and scale of computing tasks increase, the need for enhanced computing performance becomes more obvious.

The research object of this thesis is the hybrid MPI+threads parallel programming model in HPC environment. We expound the principle of the model, analyze the performance of processes and threads under different degrees of parallelism, investigate the synchronization efficiency of multithreaded MPI communication, and compare blocking and non-blocking communication methods. And proposed a matrix multiplication solution, which reflects the potential of the hybrid model.

Experiments have proved that Pthreads is superior to MPI processes and OpenMP, especially in terms of fine-grained parallelism. Compared with its single-threaded communication, parallel MPI communication is significantly accelerated. Pthreads also demonstrated excellent thread synchronization efficiency in MPI multithreaded communication. In matrix multiplication, Pthreads effectively improves the degree of communication parallelism and reduces the synchronization overhead and communication calculation ratio in parallel computing programs.

ABSTRACT

This paper provides a comprehensive analysis of the hybrid MPI+Threads parallel programming model, emphasizing its potential in utilizing the multi-level parallel structure of modern computer systems. The paper first explores the current state of parallel computing research, detailing the theoretical foundations of parallel computing, including the architecture of parallel computers and cluster technology, parallel programming models, and performance metrics for parallel programs.

The core of the research is the hybrid MPI+Threads programming model. We implement and evaluate this hybrid programming on a test platform under a single node and cluster environment, assessing the performance of processes versus threads in coarse-grained and fine-grained parallelism, as well as multithreaded ring communication, multithreaded linear broadcasting, blocking versus non-blocking performance, OpenMP versus Pthreads communication performance, and realized matrix multiplication under the hybrid model.

TABLE OF CONTENTS

DEFINITIONS, DESIGNATIONS AND ABBRIVIATIONS

The present explanatory note uses the following abbreviations and designations:

DSM – Distributed Shared Memory

HPC – High Performance Computing

MIMD – Multiple Instruction Multiple Data

MPI – Message Passing Interface

NSU – Novosibirsk State University

Pthreads – POSIX Threads

SIMD – Single Instruction Multiple Data

INTRODUCTION

In contemporary scientific research and engineering calculations, high-performance computing (HPC) [7] has emerged as a potent tool, offering tremendous processing power and computational speed. Many scientific disciplines, including climate modeling, biological sciences, computational fluid dynamics, molecular dynamics, materials science, and artificial intelligence, rely on HPC for simulations, data analysis, and visualization. Within these domains, research often centers around the development and optimization of HPC system applications.

However, with the escalating complexity and scale of computing tasks, the computational prowess of a single processor or node is no longer sufficient. As a result, researchers are increasingly turning to parallel computing methods to enhance computational performance and reduce execution time. In the field of HPC, the Message Passing Interface (MPI) [13] parallel programming model has become one of the most widely used programming methods, particularly for distributed memory systems and large-scale parallel computing.

To fully leverage the multi-level parallel structure of modern computer systems, the hybrid MPI+threads [11, 16] parallel programming method has emerged as a promising solution. By combining MPI and thread parallel programming (such as OpenMP [2, 3] or Pthreads [31]), higher parallel performance can be achieved at different levels. Specifically, MPI can be used to implement distributed processing of tasks between computing nodes, while threaded parallel programming can achieve parallelism between multi-core processors inside nodes. This hybrid programming method aims to maximize the potential performance of modern computer systems, reduce communication overhead, and improve computational efficiency.

The main goal of this thesis is to study the parallel programming method in the hybrid MPI+threads model, realize communication in multithreaded MPI and specific matrix calculation applications, record the runtime of the program, compare

the performance of different optimization methods and thread synchronization strategies, and analyze the experimental results.

1. THE MODERN STATE OF PARALLEL COMPUTING RESEARCH

1.1. Research status

In the realm of serial computing, a unified model for hardware and software design and implementation is already established. This is fundamentally built on the stored program and sequential execution computation model proposed by John Von Neumann, which has laid a solid foundation for the development of serial computing. However, when it comes to parallel computing, a similar unified design and implementation model has not yet emerged. This is primarily because parallel computing involves a variety of different hardware architectures and programming paradigms, each with its own specific advantages and application scenarios.

Scholars have indeed proposed numerous classic parallel computing models, such as PRAM, LOGP, and BSP. These models aid in understanding the design and performance analysis of parallel algorithms.

PRAM [32] is an idealized parallel computing model that integrates multiple processors with shared memory. In the PRAM model, all processors can access shared memory within a constant time, and there is no communication delay between processors. PRAM has multiple variants, including ERW-PRAM (Exclusive Read, Exclusive Write PRAM), CREW-PRAM (Concurrent Read, Exclusive Write PRAM), and CRCW-PRAM (Concurrent Read, Concurrent Write PRAM), with the main difference lying in the restrictions on processor reading and writing shared memory. Although the PRAM model theoretically holds appeal, its practical application is limited as it overlooks many real-world constraints, such as communication delays and memory access conflicts.

The BSP model [9], proposed by Leslie Valiant in 1990, is a parallel computing model. The core idea of the BSP model is to divide the computational and communication process into a series of synchronous super-steps, each comprising three stages: local computation, communication, and global synchronization. The BSP model's strength lies in its provision of a simple and

universal parallel computing abstraction that can be used to design scalable parallel algorithms and evaluate the performance of different hardware platforms. The BSP model separates the computational and communication processes, allowing developers to focus on optimizing each stage without considering the complexity of the entire parallel process. However, the BSP model also has its limitations, such as global synchronization potentially leading to processor idleness and performance loss. Furthermore, the BSP model does not detail the specific implementation of communication and synchronization, making optimization necessary for specific hardware and communication libraries in practical applications.

The LOGP model [8, 30], proposed by Culler et al. in 1993, is a practical parallel computing model used to analyze the performance of parallel algorithms in distributed memory systems. This model considers key factors like latency, bandwidth, and overhead in communication between processors. The LOGP model includes four parameters: L (communication delay between processors), o (communication overhead between processors), g (communication interval), and P (number of processors).

The LOGP model aims to provide a simplified perspective to analyze the performance of parallel algorithms in distributed memory systems. When designing parallel algorithms, the performance of the algorithm can be optimized by adjusting these four parameters, such as improving algorithm efficiency by reducing communication latency, reducing communication overhead, or increasing bandwidth. The LOGP model's beauty lies in its ability to capture key performance characteristics of distributed memory systems while remaining simple enough for performance analysis and optimization. However, the LOGP model also has its limitations, such as overlooking the complexity of network topology and inter-processor communication. Despite this, the LOGP model remains a useful tool for evaluating the performance of parallel algorithms in distributed memory systems.

In conclusion, although a unified design and implementation model has not yet emerged in the field of parallel computing, this doesn't hinder us from utilizing existing parallel computing models to comprehend the design and performance

analysis of parallel algorithms. By understanding and applying these models, we can better leverage the advantages of parallel computing, design more efficient parallel algorithms, and thus solve larger scale and more complex computational problems.

1.2. The modern tasks

For the above three parallel programming models, there are currently widely used tools and parallel programming environments, such as MPI (a tool based on message passing), OpenMP (a parallel programming model based on shared memory), Pthreads (a thread library based on POSIX standards) and OpenACC (parallel programming model based on compiler directives), etc.

MPI is not the realization of a specific function, but is designed and implemented according to the requirements of the application for the message function and the standard requirements of the message interface function jointly formulated by the global industry, application and research departments. This design is to ensure the portability of the program. When formulating such a standard, it is necessary to find a balance between high communication performance, good portability and powerful functions. In view of the later development of MPI, it largely integrates the advantages of multiple parallel environments, bringing together the characteristics of performance, functionality and portability, thus forming a unique competitive advantage. Therefore, within just a few years, MPI has been widely recognized as the standard for message-parallel programming patterns.

The emergence of OpenMP is mainly due to the multi-threaded parallelization of computers serving shared memory. Both it and MPI use the interface pattern, but the difference is that OpenMP is an application programming interface designed for parallelization of multithreaded tasks on computers under shared storage. It includes a set of compiler directives and function libraries.

The POSIX standard is developed by the IEEE. It defines a set of cross-platform operating system interfaces, including file system, process management,

signal processing, etc. Pthreads, as part of the POSIX standard, provides programmers with a unified way to implement multithreaded programming on different operating systems and hardware platforms. This makes it easier for programmers to write portable, multithreaded applications.

1.3. Significance of research

In the MPI model, each process runs independently and communicates and cooperates with each other by sending and receiving messages. Although the pure MPI model performs well in many cases, it also has some disadvantages：

1. Communication overhead: In the pure MPI model, communication between processes needs to be achieved by sending and receiving messages, which may lead to greater communication overhead, especially in large-scale cluster systems.

2. Resource utilization: The pure MPI model does not fully exploit the potential of multi-core processors, which may lead to a waste of processor resources.

3. Memory usage: Each MPI process has independent memory space. When it is necessary to manipulate shared data, it must be copied between processes, which may lead to additional memory usage and data transfer overhead.

Facing some challenges of the MPI model, such as communication overhead, resource utilization, and memory usage, we propose a hybrid MPI+threads model as a solution. In this model, multithreaded MPI is used for communication between nodes, while threads are used for parallel computing within nodes. The main advantages of this model are that it can reduce communication overhead in large-scale cluster systems, more fully utilize the potential of multi-core processors, and reduce memory usage and data transmission overhead.

The main task of this paper is to conduct experiments on a set of computer clusters provided by NSU to verify the performance of this hybrid model. First, we will complete the serial, pure MPI, MPI+OpenMP and MPI+Pthreads algorithms of matrix multiplication on a single node, analyze their operating efficiency, and explore the performance differences between processes and threads. We will then

set up and run multithreaded MPI blocking and non-blocking communication, multithreaded point-to-point and broadcast communication programs on multiple nodes in the cluster, compare the multithreaded communication performance of OpenMP and Pthreads, and test instantiated matrix multiplication algorithms (including multithreaded data communication).

Through these experiments, we hope to have an in-depth understanding and analysis of the performance of the hybrid MPI+threads model, and provide a valuable reference for the optimization of parallel computing.

2. THEORETICAL BASIS OF PARALLEL COMPUTING

2.1. Architecture of parallel computer and cluster technology

The Flynn taxonomy provides a basic framework for the research and design of parallel computer architecture, which divides parallel computers into four categories: SISD (Single Instruction, Single Data), SIMD (Single Instruction, Multiple Data), MISD (Multiple Instruction, Single Data), MIMD (Multiple Instruction, Multiple Data). The SISD architecture is a traditional serial computer architecture, such as an early single-core processor. The MISD system uses multiple processors to execute different instructions on the same data at the same time, this architecture is less common in practical applications because it is usually not as efficient as other architectures. Therefore, this article only introduces SIMD and MIMD [15].

2.1.1 Single Instruction Multiple Data

In the SIMD (Single Instruction, Multiple Data) parallel computing architecture, one or more processors can execute the same instruction on multiple data at the same time. SIMD architecture leverages data-level parallelism (DLP) and is widely used in many fields, such as graphics processing, scientific computing, and digital signal processing.

SIMD instructions are usually used to perform operations such as addition, subtraction, multiplication, and division on vectors or data arrays. By processing multiple data elements at the same time, SIMD instructions can perform calculations more efficiently than scalar instructions that manipulate single data elements in sequence. This can increase throughput and shorten the execution time of tasks suitable for SIMD processing.

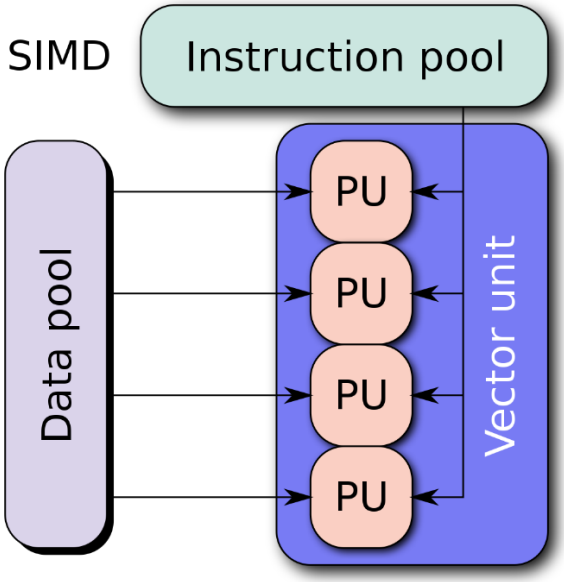

Figure 1 – SIMD Architecture [33]

The following are some common SIMD technologies and hardware:

Vector Processors: It is a representative of early SIMD technology, which can process a set of data at one time to achieve high-performance computing. For example, the Cray-1 supercomputer used vector processors to perform large-scale scientific computing tasks.

Multimedia Extensions: Many modern processors support multimedia extensions, such as Intel's MMX and SSE instruction sets, AMD's 3DNow! and SSE instruction sets, and ARM's NEON instruction set. These instruction sets allow the processor to perform many of the same operations at once, thereby improving the performance of multimedia processing and scientific computing.

GPU (Graphics Processing Unit): GPU is an important application of SIMD architecture, which is dedicated to processing graphics and image-related computing tasks. Modern GPUs (such as NVIDIA's GeForce and Tesla series, and AMD's Radeon series) have a large number of parallel processing cores that can perform complex parallel computing tasks.

AVX (Advanced Vector Extensions): AVX is an advanced SIMD instruction set that supports larger vector widths (such as 256 bits and 512 bits) and provides more functions. The AVX instruction set is supported in modern processors from

Intel and AMD and can be used to accelerate scientific computing, graphics processing, and machine learning.

2.1.2 Multiple Instruction Multiple Data

MIMD (Multiple Instruction, Multiple Data) allows multiple processors to simultaneously execute different instructions and process different data, and a machine using MIMD has many processors that operate asynchronously and independently.

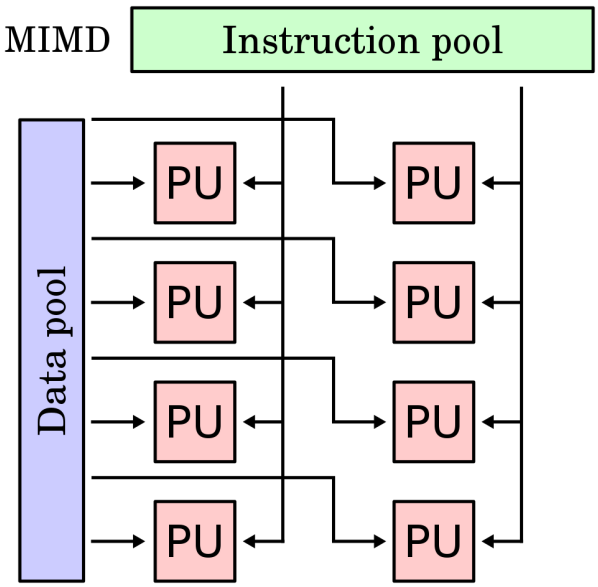

Figure 2 – MIMD Architecture [34]

MIMD architectures can be classified according to how memory is organized and how processors communicate: Shared Memory, Distributed Memory, Distributed Shared Memory.

In a shared memory MIMD system, all processors share a unified memory space, and they communicate by reading and writing to this memory space. Shared memory MIMD computers can be further subdivided into:

Uniform Memory Access (UMA): The time overhead for all processors to access memory is the same. This type of system is usually called a Symmetric Multiprocessor (SMP) system.

Non-Uniform Memory Access (NUMA): The time overhead for the processor to access memory depends on the memory location where the data is located, and the memory access overhead for a longer distance is higher.

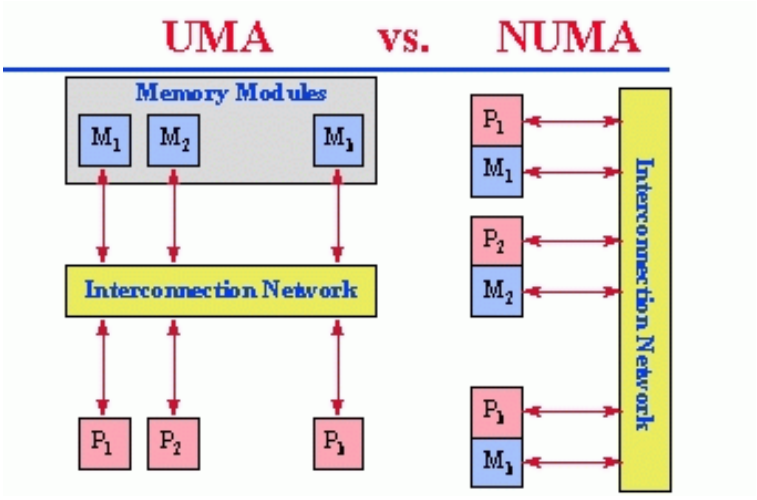

Figure 3 – UMA and NUMA [35]

In a distributed memory MIMD system, each processor has its own private memory space and communicates with other processors through message passing. These systems, often referred to as multicomputer systems, include clusters and massively parallel processors (MPP). The processors or nodes in this architecture are connected through a network, such as Ethernet or InfiniBand. Each processor has its own local memory that can only be directly accessed by that specific processor. To exchange data between processors, explicit communication is necessary, where messages are sent over the network to transfer data from one processor's local memory to another. This distributed memory approach enables data sharing and parallel processing across multiple processors, facilitating the execution of complex and computationally demanding tasks.

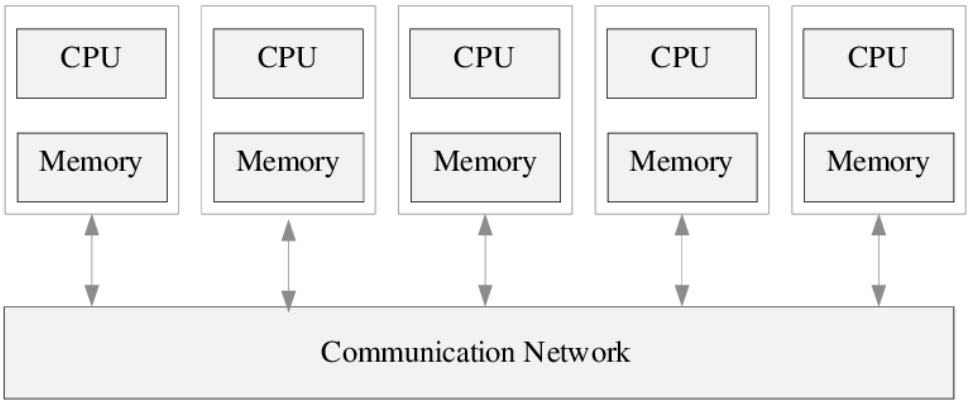

Figure 4 – Distributed Memory Architecture

The Distributed Shared Memory (DSM) system provides a programming abstraction similar to a shared memory system for a distributed memory system. In the DSM model, although physical memory is distributed on different computing nodes, logically all nodes see a unified memory address space. This allows programmers to operate distributed memory systems like shared memory when writing parallel programs, simplifying programming complexity. Although messaging libraries (such as MPI) are mainly used for communication in distributed memory systems, they can also be used to implement the DSM model. In this case, the programmer needs to manage the global memory address space by himself and use the messaging interface to achieve access to remote memory.

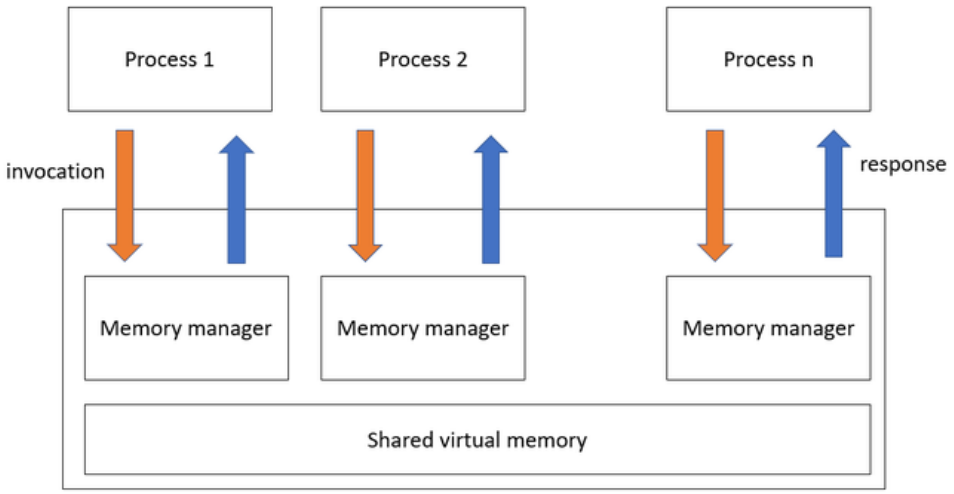

Figure 5 – DSM Architecture

2.1.3 SMP, COW, PVP and MPP

SMP (Symmetric Multi-Processing, symmetric multi-processing):

All processors in the SMP architecture share the same memory space and have the same function and status. Processors in an SMP system can perform arbitrary tasks independently, without the constraints of other processors. This architecture utilizes the task scheduling mechanism of the operating system to assign tasks to individual processors. The advantage of the SMP system is that it simplifies the communication and data sharing between processors, but memory access competition and synchronization problems may occur when the number of processors is large.

COW (Cluster of Workstations, workstation cluster):

The COW architecture is a parallel computing system based on loose coupling, which connects multiple general-purpose workstations through a high-speed network to form a computing cluster. COW systems usually adopt distributed memory MIMD architecture and use message passing technology (such as MPI) for inter-processor communication. The advantage of COW is that it can utilize existing hardware resources and infrastructure, and has lower construction and maintenance costs. However, COW systems may require more complex communication and data distribution mechanisms compared to SMP and MPP architectures.

PVP (Parallel Vector Processor, parallel vector processor):

PVP is a parallel computer architecture for vector processing. In a PVP system, processors can perform operations on vector data concurrently. This architecture is suitable for handling massive data-intensive computing tasks, such as scientific computing and graphics processing. The advantage of the PVP system is that it can achieve efficient data processing performance, but it may not be suitable for non-vector processing tasks.

MPP (Massively Multi-Processing, large-scale multi-processing):

The MPP architecture is mainly used in massively parallel computer systems, such as supercomputers. MPP systems usually adopt MIMD architecture and

organize multiple processors into distributed memory or shared memory. The MPP architecture can scale to thousands or even millions of processors, so it is highly scalable. In order to achieve efficient parallel computing, MPP systems need to use high-speed interconnection networks, highly optimized software and algorithms, and operating systems and programming models specially designed for parallel computing.

Table 2.1 – Comparison of the four models

Attributes	SMP	COW	PVP	MPP
Isomorphism	MIMD	MIMD	MIMD	MIMD
Synchronization	Asynchronous or weak synchronization	Asynchronous or weak synchronization	Asynchronous or weak synchronization	Asynchronous or weak synchronization
Communication mechanism	Shared memory	Messaging	Shared memory	Messaging
Address space	Single Space	Multiple Space	Single Space	Multiple Space
Access memory model	UMA	NORMA	UMA	NORMA

2.1.4 Supercomputer

Supercomputer (giant computer) are computers with higher computing speed, larger storage capacity, and more complete functions than mainframe computers. It usually refers to an electronic computer that can perform trillions (or even higher) of floating-point operations per second on average. It plays an important role in intensive computing, massive data processing and other fields.

As an important embodiment of a country's information technology, supercomputers will first play an important role in national defense science and technology, industrialization, aerospace satellites and other fields, and secondly, it will show its advantages in fields such as meteorology, physics, and exploration.

Supercomputers have a strong ability to calculate and process data. Their main characteristics are high speed and large capacity. They are equipped with a variety of external and peripheral equipment and rich, high-function software systems. The supercomputer adopts a turbine design, and each blade is a server, which can work together and can be increased or decreased at any time according to the needs of the application.

Taking China's domestically produced “Sunway TaihuLight” supercomputer as an example, its continuous performance is 930 million per second, and its peak performance can reach 1.25 billion per second. The " Sunway TaihuLight" supercomputer consists of 40 computing cabinets and 8 network cabinets. Each computing cabinet is slightly larger than a household two-door refrigerator. When the cabinet door is opened, 4 super nodes composed of 32 computing plug-ins are distributed among them. Each plug-in consists of 4 computing node boards, and one computing node board contains 2 “SW26010” high-performance processors. There are 1,024 processors in one cabinet, and the entire “Sunway TaihuLight” has a total of 40,960 processors.

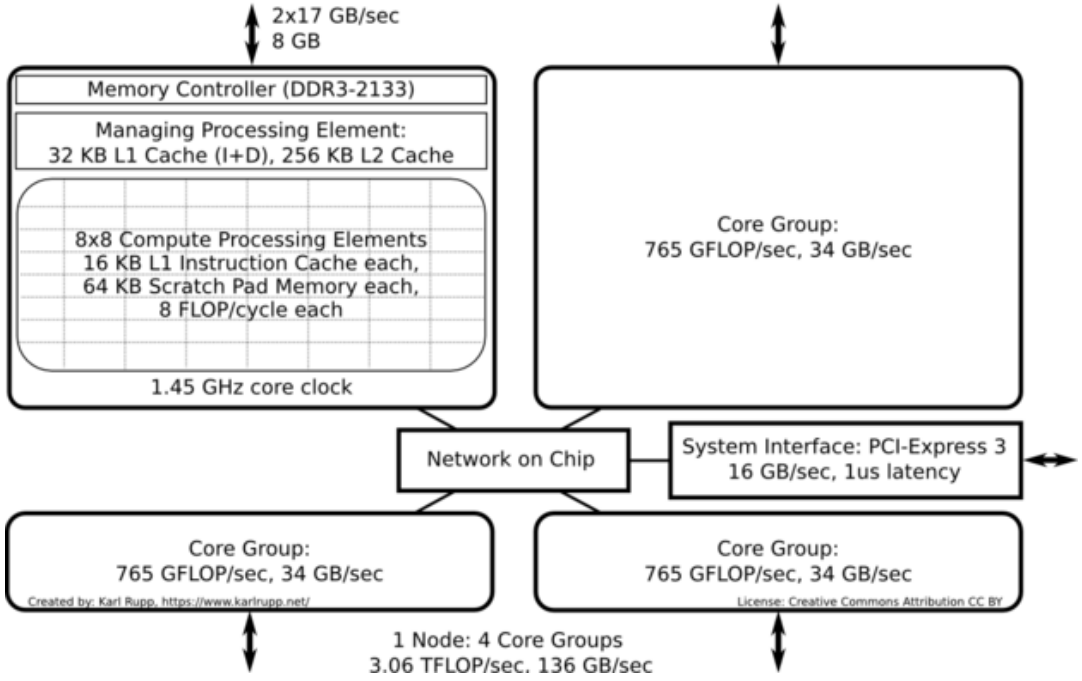

Figure 6 – Node Schematic of “Sunway TaihuLight” [36]

2.2. Parallel Programming Models

2.2.1 Data Parallelism Model

The data parallelism model enables parallel computing by breaking large amounts of data into smaller parts and assigning these parts to multiple processors or computing nodes. In the data parallelism model, the processor performs the same operation, but on different subsets of data. Data parallelism models are usually suitable for problems with regular data access patterns and computational loads, such as vector and matrix operations, image processing, etc.

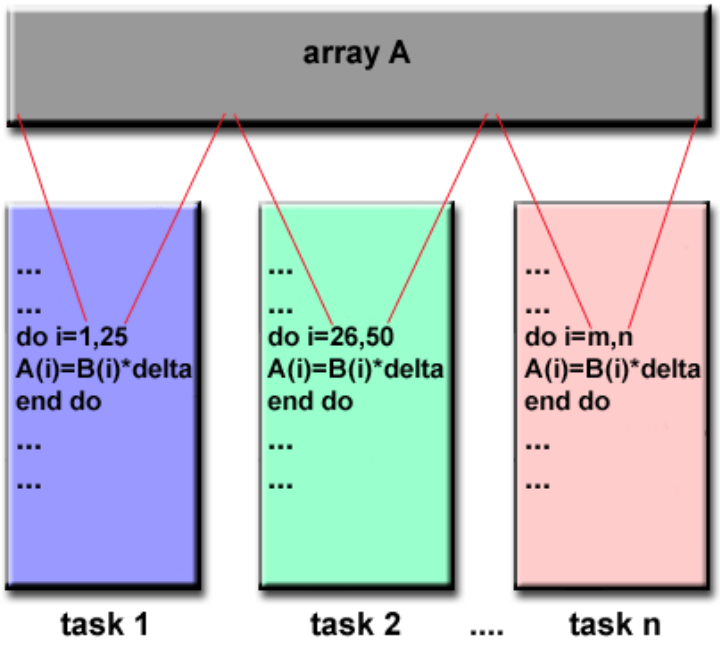

Figure 7 – Data Parallelism Model

The data parallelism model has the following characteristics：

1. Aggregated data structure: In a program under the data parallelism model, the same statement can call different data set elements, and it can also call elements on other data structures.

2. Single control flow: In a program under the data parallelism model, single-threaded control can complete a single control flow task, which can make data parallelism execute like a sequential program.

3. Global namespaces: The global namespaces allow variables to live here, so that when the rules of variable scope are met, you can use access statements to call any variable here.

4. Loose synchronization: In a program under the data parallelism model, there is an implicit synchronization after each statement. This method is called loose synchronization.

5. Implicit interaction: In a program under the data parallelism model, there is an implicit synchronization after each statement, which creates a roadblock to implicit synchronization, so there is no need to display synchronization.

2.2.2 Thread-level parallelism

Thread-level parallelism (TLP) is a parallel computing strategy that involves creating and executing multiple threads within a single program. The purpose of TLP is to enable simultaneous execution of these threads on a multi-core processor or multi-processor system. By leveraging TLP, tasks can be divided into smaller units and executed independently or semi-independently by different threads.

The main objective of TLP is to achieve task-level parallelism, where multiple threads can execute their respective tasks concurrently. Each thread typically operates on different data or performs a distinct computation, allowing for parallelization of the workload. TLP is especially beneficial in scenarios where tasks can be divided into smaller, independent units that can be executed simultaneously.

Modern programming languages and operating systems offer extensive support for thread-level parallelism. They provide libraries, APIs, and tools to create and manage threads efficiently. Multi-core processors, which have become commonplace in today's computing systems, greatly benefit from TLP as it enables better utilization of the available cores.

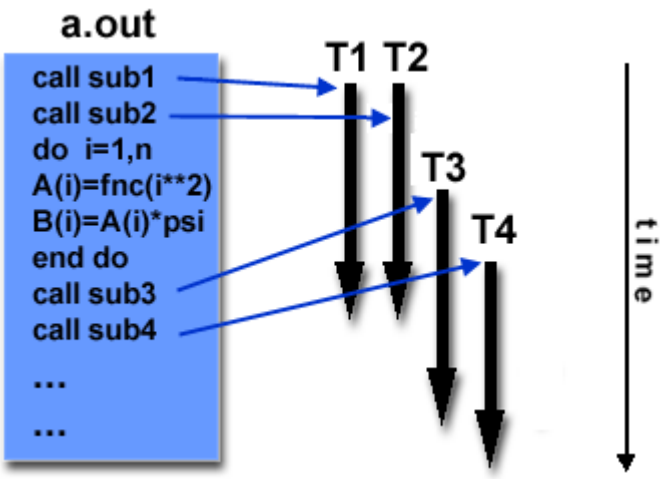

Figure 8 – Thread Parallelism

Characteristics of thread parallelism:

1. Make full use of multi-core processors: Thread-level parallelism can effectively use the computing power of multi-core processors and multi-processor systems. By executing multiple threads on different processor cores, the execution speed of the program can be improved.

2. Improve concurrency: Thread-level parallelism can improve the concurrency of programs. Multiple threads can execute at the same time, which allows the program to complete more tasks in a shorter period of time.

3. Improve responsiveness: Thread-level parallelism can improve the responsiveness of the program. By assigning time-consuming tasks to background threads, the program can continue to respond to user input and process other tasks.

4. Shared memory: Threads can communicate and exchange data directly through shared memory, which makes collaboration easier and more efficient. However, shared memory may also cause race conditions and data inconsistencies, so synchronization primitives (such as mutexes, semaphores, etc.) need to be used to ensure correct collaboration between threads.

5. Flexibility: Thread-level parallelism can adapt to different types of applications and tasks. Programmers can flexibly create and manage threads according to specific needs.

Therefore, when implementing thread-level parallel programs, issues such as thread management, synchronization, and communication need to be considered to ensure that concurrent programs are implemented correctly and efficiently.

2.2.3 Shared Memory Model

The shared memory model allows multiple processors or threads to access the same memory space. This means that all processors can access and modify data in memory at the same time. The shared memory model simplifies the process of data communication and collaborative work, because there is no need to explicitly send and receive data between processors.

Shared memory architecture can be divided into two types: one is the global address space shared memory architecture (SMP), which is a computer system with multiple CPUs (central processing units), all processors share the same piece of memory; the other is the distributed shared memory architecture (NUMA), which is a computer system with multiple CPUs, but each CPU only shares the accompanying local memory.

Using the shared memory model in parallel programming, multiple threads or processes can be used to share data, thereby improving the execution efficiency of the program. However, since multiple threads or processes access shared data at the same time, data competition and deadlocks may occur, so a synchronization mechanism needs to be used to ensure the correctness of the data and the execution order of the program.

Some popular programming models and APIs (application programming interfaces), such as OpenMP and Pthread, can be used to write parallel programs in shared memory. At the same time, the architecture and performance of modern processors pose challenges to the shared memory model, and it is necessary to use cache coherency mechanism and NUMA-aware technology to ensure the efficiency and correctness of memory access.

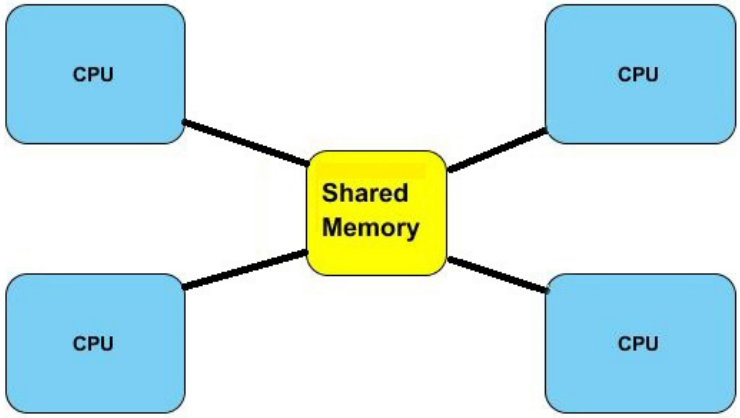

Figure 9 – Shared Memory

2.2.4 Message Passing Model

In the messaging model of parallel programming, processes running on different nodes communicate with each other by exchanging messages over a network. Messages can contain instructions, data, synchronization signals, or interrupt signals. This model allows for communication and coordination between processes located on separate nodes in a distributed computing system.

In a messaging parallel program, the programmer is responsible for explicitly assigning data and computational tasks to individual processes. This makes the messaging model more suitable for developing coarse-grained parallelism, where larger tasks are divided among processes. Each process can be multithreaded and asynchronous, meaning they can execute tasks independently and concurrently. However, explicit synchronous operations, such as roadblocks or barriers, are necessary to ensure the correct execution order and synchronization between processes.

One key feature of the messaging model is that each process has its own separate address space. This means that each process has its own private memory space that is not directly accessible by other processes. Communication between processes occurs by sending messages over the network, and the receiving process then processes the message and acts accordingly.

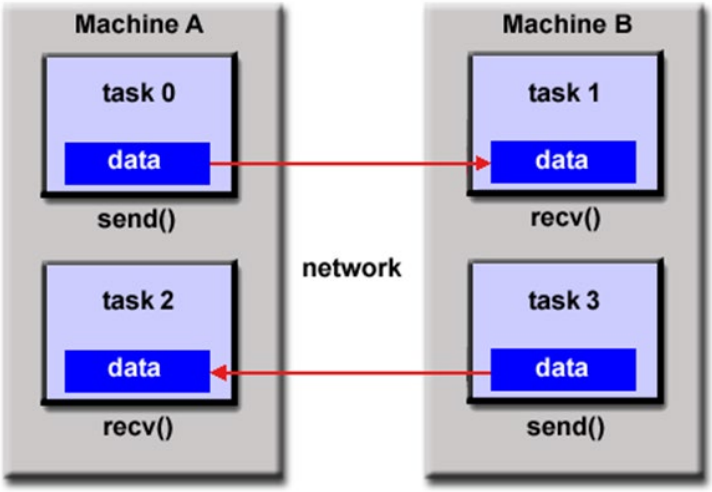

Figure 10 – Message Passing

In addition, it also has the following advantages:

1. Extensibility: The messaging model is very suitable for large-scale parallel systems, such as computing clusters and supercomputers. Since each processor has its own independent memory space, the system can be easily expanded to more processors and nodes.

2. Good locality: In the messaging model, each process manipulates data in its own address space, so it has good locality. This helps to improve cache hit rate and memory access performance.

3. Explicit communication: The messaging model requires programmers to explicitly handle data communication and synchronization. Although this increases the difficulty of programming, it also enables programmers to better control the communication overhead and the execution of parallel programs.

4. Error tolerance: Since each processor has independent memory space, the messaging model has good error tolerance. If a processor fails, you can keep the system running normally by rescheduling tasks and resending messages.

2.3. Performance Metrics for Parallel Programs

2.3.1 Amdahl's Law

Amdahl's law is one of the important quantitative principles of computer system design. It was first proposed by Amdahl, the main designer of the IBM360 series of machines, in 1967.

This law refers to the degree of system performance improvement that can be obtained by adopting a faster execution method for a certain component in the system, depending on the frequency of this execution method being used, or the proportion of total execution time.

Amdahl's law actually defines the performance improvement or acceleration ratio of execution time that can be obtained after taking measures to enhance (accelerate) the processing of a certain part of the function.

Amdahl's law can be expressed in the following definitions:

$$S = \frac{1}{1 - a + \frac{a}{n}} \tag{2.1}$$

S is speedup, which represents the global acceleration speed (the original total time/the total time after acceleration).

a is the proportion of parallel computing (the amount of code can be calculated in parallel/the total amount of code).

n is the number of parallel nodes processed, which can be understood as the number of CPU cores.

It can be seen from the formula that increasing the number of processors does not necessarily effectively increase the acceleration ratio. If the parallelization program of the system is not high, that is, the value of a is close to 100%, even if n is infinite, the acceleration ratio is close to 1, and it will not play any role in the performance optimization of the system, but the cost has increased infinitely.

Therefore, the overall acceleration or performance improvement that can be achieved is limited by the execution time fraction represented by the improved component.

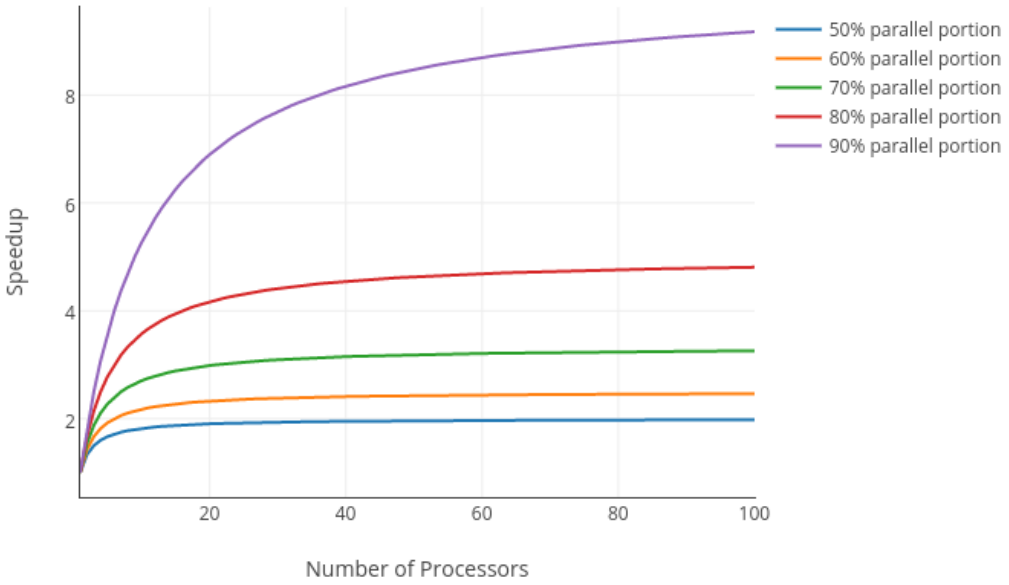

Figure 11 – Amdahl's law [37]

This law reminds us that when designing and optimizing computing systems, we should think about how to optimize resource allocation and task division, and reduce the proportion of serial computing to avoid performance bottlenecks. In practical applications, we can use technical means such as task parallelization, data parallelization, and pipelining to reduce the proportion of serial calculations and improve the parallel computing power of the system, so as to obtain better performance.

2.3.2 Gustafson's Law

Gustafson's law is a theorem proposed by another computer scientists John Gustafson and Edwin Barsis in 1988. Similar to Amdahl's law, Gustafson's law assumes that the scale of the problem can be expanded, and believes that as the available computing resources increase, the proportion of parallelizable parts can increase, so parallel computing can be extended to a larger problem scale, and the acceleration ratio will not be limited by the serial part, thereby improving the performance of the entire program.

Gustafson's law revisits the calculation of the acceleration ratio. In Gustafson's law, the acceleration ratio can be calculated by the following formula:

$$\text{Acceleration ratio (S)} = P + (1\text{-}P) \times N \tag{2.2}$$

P represents the proportion of the parallel part of the program to the total execution time.

N represents the number of processors.

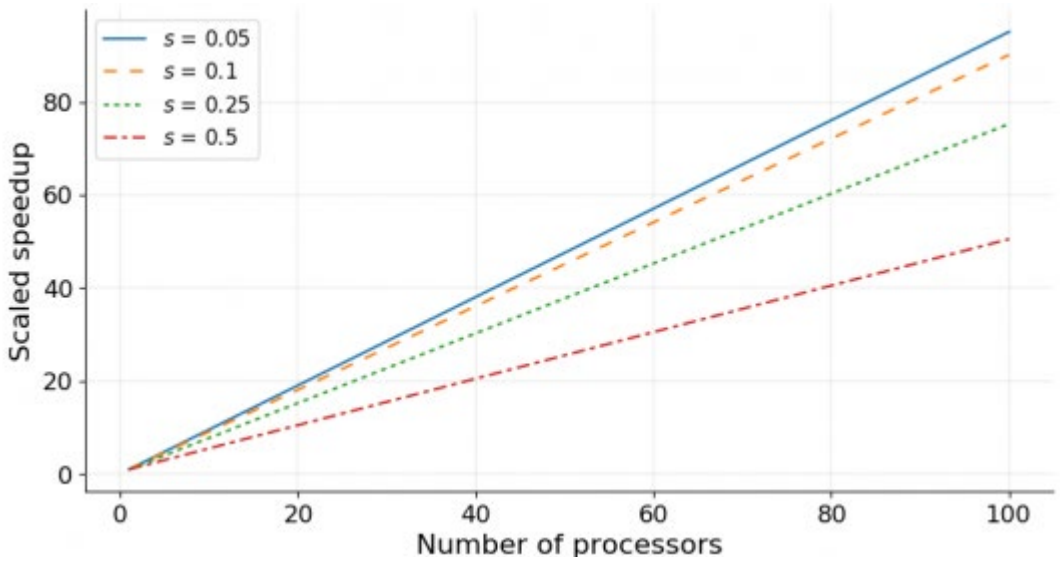

Figure 12 – Gustafson's law [38]

This formula shows that as the scale of the problem increases, a scalable parallel computing system can achieve a higher acceleration ratio.

Gustafson's law challenges the hypothesis of fixed problem scale in Amdahl's law, prompting researchers and engineers to think about the performance and extensibility of parallel computing from different angles. This helps to more comprehensively evaluate the performance of parallel algorithms and systems, and provides a new perspective for the design and optimization of parallel computing systems. However, the actual parallel computing system may be affected by a variety of factors, such as communication delays, load imbalances, etc. Therefore, when

evaluating and optimizing the actual parallel computing performance, it may be necessary to combine Gustafson's law and other theoretical knowledge.

2.3.2 Communication-to-Computation Ratio

The Communication-to-Computation Ratio is an important indicator used to evaluate the performance of parallel programs. It represents the ratio between the communication time and the calculation time in a parallel program. This ratio helps to understand the relative overhead of communication and computing in parallel programs, so that a suitable balance can be found when designing and optimizing parallel programs.

The calculation formula of Communication-to-Calculation ratio is as follows:

$$\mathrm{CCR} = \frac{communication\ time}{calculation\ time} \tag{2.3}$$

In parallel computing, communication time refers to the time required to transmit data between processors, including data transmission, synchronization, and communication overhead. Calculation time refers to the time it takes for the processor to perform a calculation task. Ideally, people want the communication time of parallel programs to be as small as possible so that the processor can focus more on computing tasks, thereby improving overall performance.

Generally, the following measures can help reduce the ratio of communication and computation, thereby improving the efficiency of parallel computing:

First, communication overhead can be reduced by consolidating communication operations and reducing overall communication times.

Secondly, choosing an appropriate communication mode (such as point-to-point communication, collective communication, etc.) and algorithms can improve communication efficiency.

In addition, through reasonable data distribution and task division, the demand for data transmission between processors can be reduced, thereby reducing communication overhead.

Finally, by making full use of the idle time between processors, the impact of communication on computation can be further mitigated by overlapping computation and communication. These four measures can optimize the efficiency of parallel computing to a certain extent and reduce the ratio of communication and computing.

3. HYBRID MPI+THREADS PROGRAMMING MODEL

3.1 Overview of MPI, OpenMP and Pthreads

3.1.1 MPI

In the hybrid model, the MPI interface is used for process communication between nodes in the cluster.

One of the most commonly used communication modes in MPI is the point-to-point communication mode, which can realize communication between any two processes.

Another commonly used communication mode is the collective communication mode, through which communication between all processes can be realized, such as broadcasting and reduction operations.

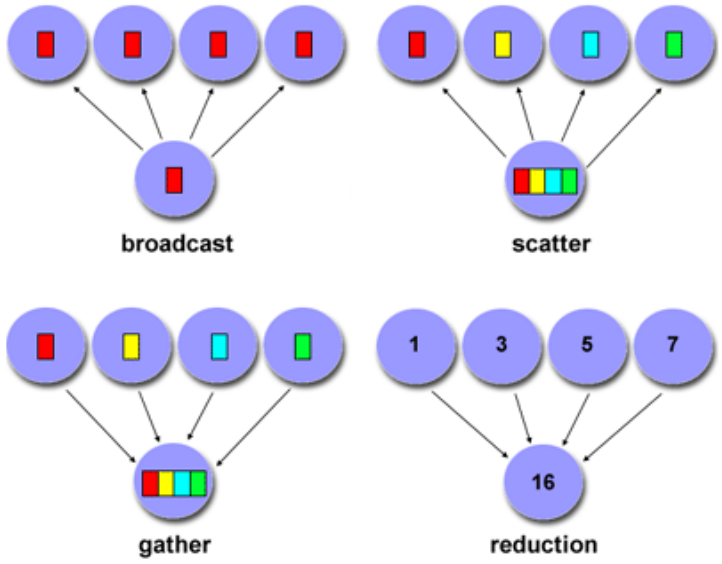

Figure 13 – MPI collective Communication [39]

In MPI, communication operations can be divided into two types: Blocking and non-blocking.

Blocking communication refers to the fact that during the MPI function call, the process must wait for the communication operation to complete before proceeding. In other words, when a blocking MPI send or receive operation is called, the calling process will be suspended until the operation is completed. The main advantage of blocking communication operations is that they are simple and easy to use, but the disadvantage is that they may cause the process to wait for too long, thereby affecting the execution efficiency of the program.

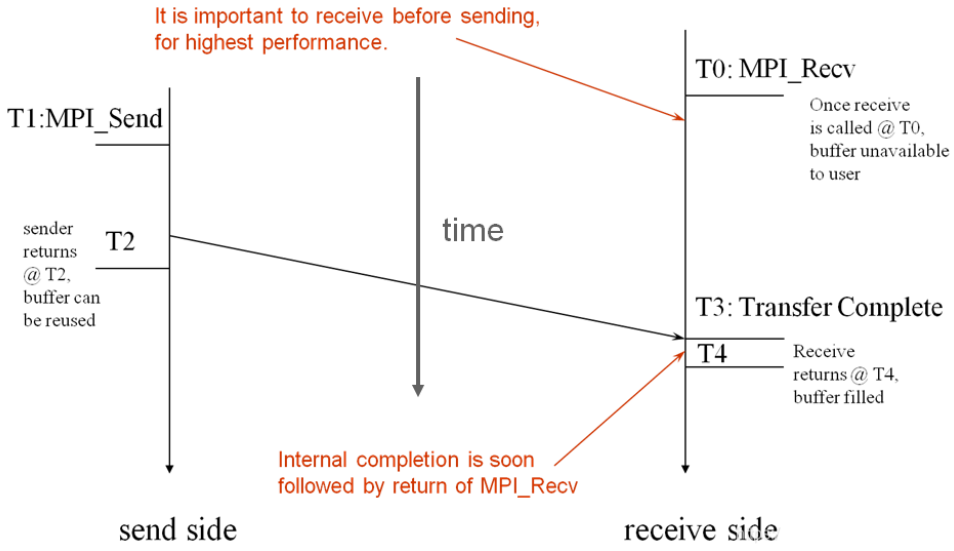

Figure 14 – MPI blocking send and receive

Non-blocking communication means that the MPI function call will return immediately, and there is no need to wait for the communication operation to complete. In non-blocking communication, a process can perform other tasks while the communication operation is in progress. This method can improve the execution efficiency of the process and reduce waiting time, but programming is more difficult and synchronization and data consistency issues need to be dealt with. For non-blocking transmission, we must use the MPI_Wait function to wait for the asynchronous operation to complete. The MPI_Wait function waits until the asynchronous operation is completed and returns after the asynchronous operation is completed.

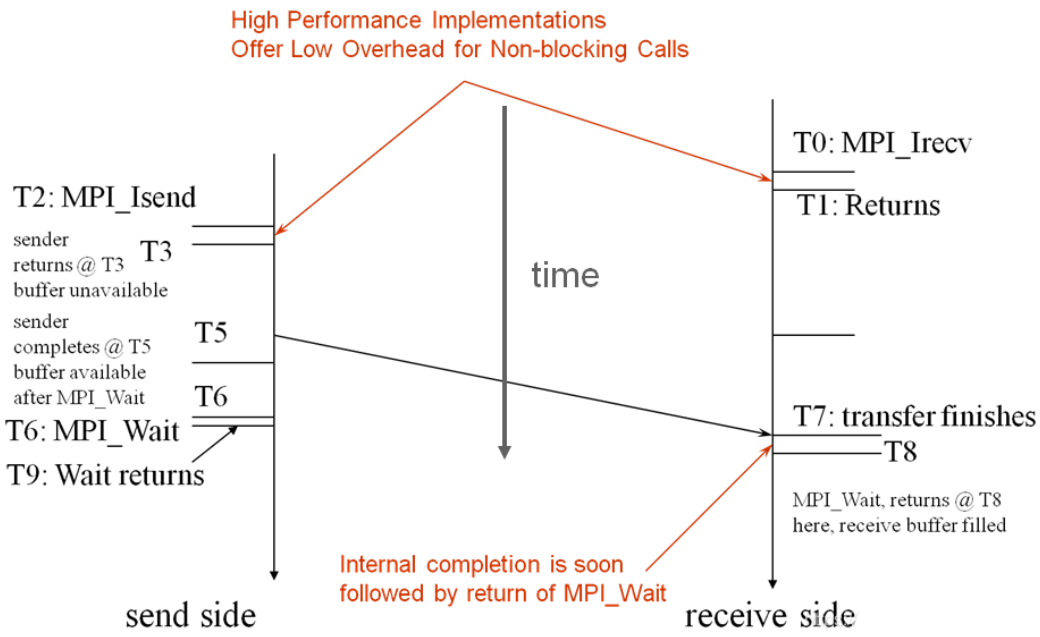

Figure 15 – MPI non-blocking send and receive

In MPI parallel programs, deadlocks are a common problem. When a deadlock occurs, multiple processes wait for each other to complete their operations, causing the program to fall into a state where it cannot continue to execute. The following are some situations that may cause MPI deadlocks:

1. Mismatched send and receive operations: If there is a problem with the matching between the send and receive operations, such as the 'tag' or communicator of the 'sender' and 'receiver' do not match, the process may keep waiting for the operation that cannot be completed.

2. Different sending and receiving order: If the process sends and receives messages in a different order, it may cause a deadlock. For example, process A first sends a message to process B, and then waits for a message from process B, and process B also sends a message to process A first, and then waits for a message from process A. In this case, both processes are waiting for the other to complete the operation, resulting in a deadlock.

3. Blocking communication and resource competition: deadlocks may occur when multiple processes use blocking communication functions. Because blocking

communication requires waiting for the operation to complete, if multiple processes compete for limited communication resources, it may lead to a deadlock.

4. Mixing collective communication with point-to-point communication: Collective communication operations involve the participation of all processes. If some processes perform point-to-point communication operations between collective communication operations, it may cause a deadlock.

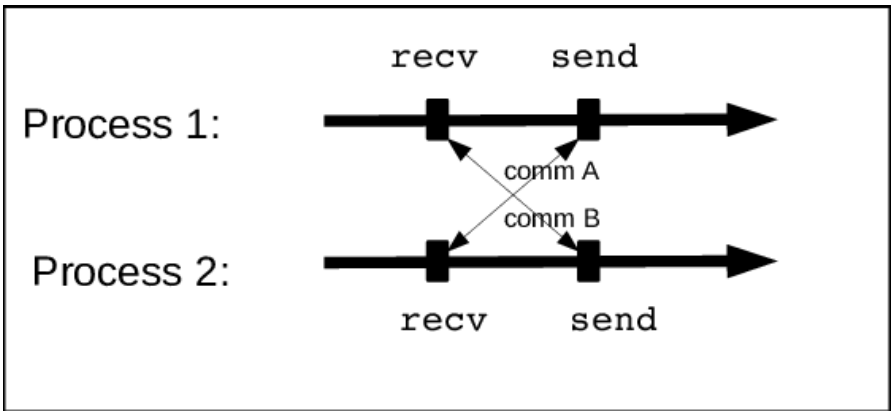

Figure 16 – A possible situation of deadlock

In order to deal with the problem of deadlocks, the following strategies can be adopted: ensure that the sending and receiving operations match, use non-blocking communication, avoid mixing collective communication and point-to-point communication, and adopt synchronization and mutual exclusion mechanisms. These strategies help ensure the stability and efficiency of parallel programs.

3.1.2 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) used to write shared memory parallel programs, supporting C, C++ and Fortran languages. OpenMP uses a combination of compiler instructions, library functions, and environment variables to allow programmers to easily create parallel programs.

OpenMP provides an instruction-based parallel programming model. Programmers can insert special compiler instructions (using the '#pragma' keyword) into the code to instruct the compiler to introduce parallelism when generating executable code. This method allows developers to gradually introduce parallelism without changing the original code structure.

An OpenMP process consists of multiple threads and uses a fork-join parallel execution model. The OpenMP program starts with a separate main thread (Master Thread), the main thread executes serially, and encounters a Parallel Region (Parallel Region) to start parallel execution.

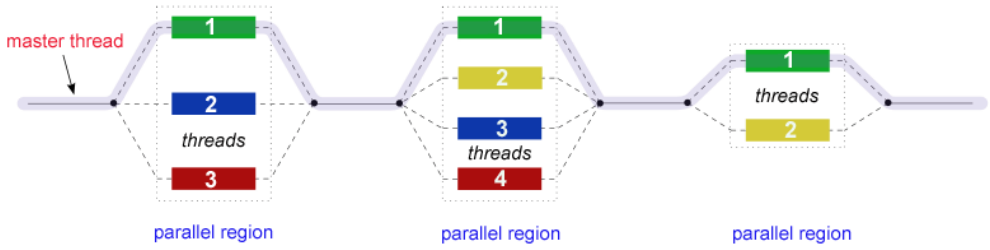

Figure 17 – Fork - Join Model [40]

In the OpenMP parallel programming environment, thread safety is of the utmost importance. Methods to ensure thread safety include:

Using private variables to ensure that each thread has independent access to variables to avoid competition conditions.

Using critical zones to protect shared resources to ensure that only one thread accesses at the same time.

Using atomic operations to reduce synchronization overhead and achieve efficient access to shared resources.

Using the lock mechanism to synchronize threads' access to shared resources to avoid data competition.

Using barrier to synchronize threads to ensure the orderly execution of key parts of the code.

Using the reduction clause to ensure thread safety while simplifying access to shared resources. Reduction operation of shared variables.

3.1.3 Pthreads

Pthreads (Portable Operating System Interface for UNIX Threads) is a standard C library used to implement multithreaded parallel computing, which is widely supported on a variety of UNIX-like operating systems.

Pthreads provides a series of thread management functions, including thread creation, destruction, synchronization, and mutex. The following are some commonly used Pthreads functions:

The ‘pthread_create’ function is used to create a new thread and let it execute the specified function.

The ‘pthread_join’ function is used to wait for the execution of the specified thread to complete in order to collect the return value of the thread or ensure that resources are released.

Functions such as ‘pthread_mutex_init’, ‘pthread_mutex_lock’, ‘pthread_mutex_unlock’, and ‘pthread_mutex_destroy’ are used to implement thread mutexes to protect shared resources from problems caused by concurrent access.

Functions such as ‘pthread_cond_init’, ‘pthread_cond_wait’, ‘pthread_cond_signal’, ‘pthread_cond_broadcast’, and ‘pthread_cond_destroy’ are used to implement condition variables in order to achieve more complex synchronization between threads.

Functions such as ‘pthread_attr_init’, ‘pthread_attr_setdetachstate’ are used to set the properties of the thread, such as whether to separate the thread, etc.

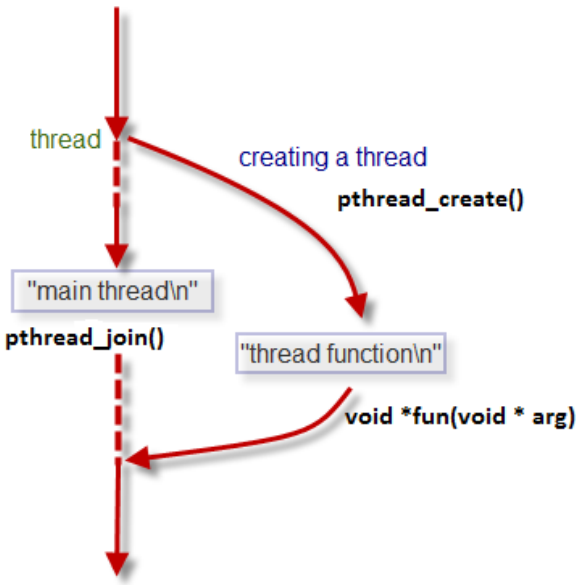

Figure 18 – The working process of pthreads [41]

Although the API of Pthreads is relatively low-level, it provides developers with powerful thread management functions, allowing efficient parallel programs to be implemented on multi-core processors and multi-processor systems.

3.2 Multithreaded MPI communication

3.2.1 Thread support in MPI

MPI supports multiple thread safety levels, as described in the figure 19 below:

Support Levels	Description
MPI_THREAD_SINGLE	Only one thread will execute.
MPI_THREAD_FUNNELED	Process may be multi-threaded, but only main thread will make MPI calls (calls are ”funneled" to main thread). “Default”
MPI_THREAD_SERIALIZE	Process may be multi-threaded, any thread can make MPI calls, but threads cannot execute MPI calls concurrently (MPI calls are ”serialized").
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, no restrictions.

Figure 19 – Thread Safety in MPI

In order to implement multi-threaded MPI communication, it is necessary to specify the required thread safety level when initializing MPI. As shown in the following code:

```
int required = MPI_THREAD_MULTIPLE;
int provided;
MPI_Init_thread(&argc, &argv, required, &provided);
```

In this paper, the thread level used by MPI in all hybrid programming code examples is 'MPI_THREAD_MULTIPLE'. This means that any thread in each process can call MPI functions at the same time without restrictions.

3.2.2 The meaning of merging threads with MPI

The hybrid parallel programming model adopts a combination of threading and MPI.

On the one hand, by using multithreading, a group of threads can perform computing tasks, while a group of threads can handle communication tasks. This can realize the overlap between communication and computing, thereby improving the performance of the application.

On the other hand, multithreading can be used to enhance the efficiency and concurrency of MPI communication, to achieve finer-grained parallelization and more efficient communication.

At the same time, the hybrid parallel programming model combines the advantages of shared memory and distributed memory models, and has the following meanings:

Possibility for better scaling of communication costs.

Simpler or faster code does not need to distribute as much data as possible, because all threads in the process can share it already.

Higher performance from using memory caches better.

Reduced need to dedicate a rank solely to communication coordination in code using a manager-worker paradigm.

3.3 Overview of the hybrid model

As mentioned in the previous section, the hybrid model system not only has local shared memory, but also communicates between processors through messaging, so to some extent it can be said that this hybrid model has some characteristics of the DSM model.

As shown in the figure 20, in the hybrid model, MPI message passing communication is used between nodes, and OpenMP or pthreads shared memory is used in nodes for parallelism.

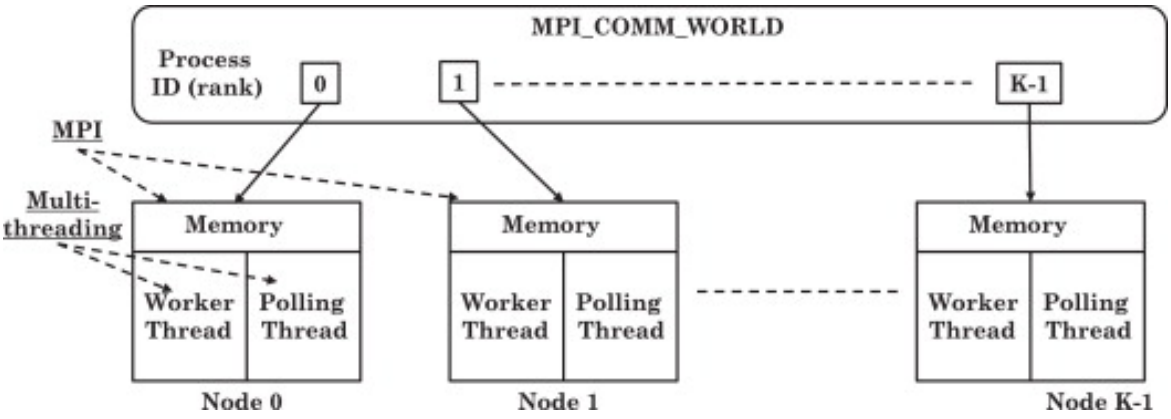

Figure 20 – Hybrid MPI model

This hybrid programming model has the highest degree of matching with the architecture of the SMP cluster, and can give full play to the potential of the hardware architecture.

The following figure 21 shows how threads work in the hybrid model (communication part):

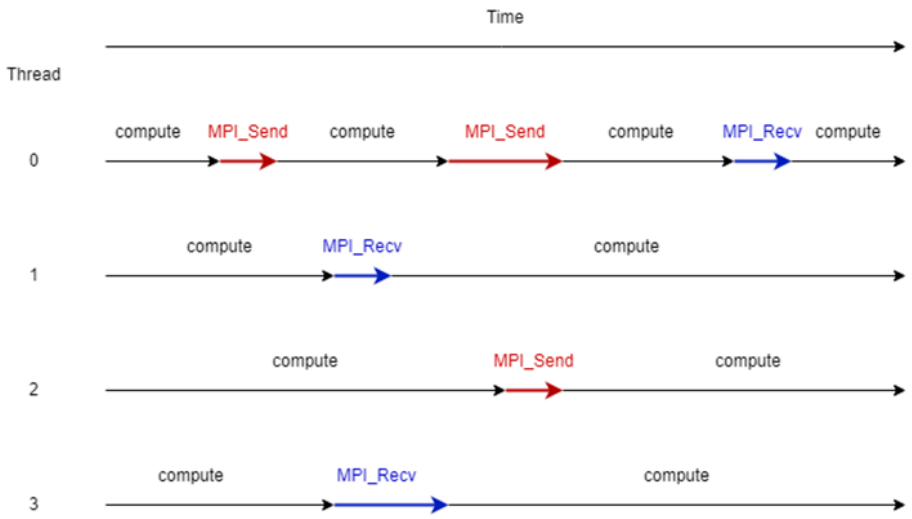

Figure 21 – MPI communication under multithreading [42]

Therefore, this model realizes that multiple threads in a single process within a node can receive or send messages without blocking at the same time, and any thread in any process can freely communicate with each other (see Figure 22), which can theoretically improve the communication efficiency of MPI.

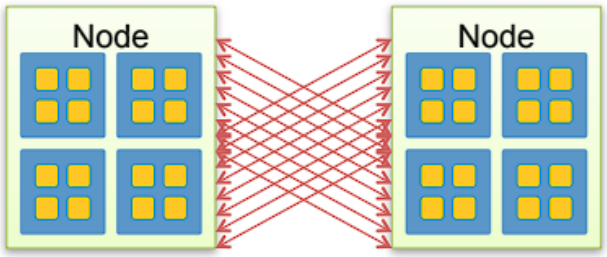

Figure 22 – Rank-thread ID to any Rank-thread ID

4. IMPLEMENTATION AND EVALUATION OF HYBRID MPI+ THREADS PROGRAMMING

4.1. Introduction to the test platform

The experiments in this paper are implemented in the cluster environment provided by NSU.

There are a total of 48 double-density servers (two computers in one chassis) HP BL2x220c G7 (2011) in the cluster, each of which contains two motherboards, each of which is equipped with:

Table 4.1 – Cluster hardware

CPU	Two 6-core Intel Xeon X5670 processors (clocked at 2932 MHz)
Memory	DDR3 24 GB
Communication network	InfiniBand 4x DDR, QDR and DDR
Transmission network	Gigabit Ethernet
Network storage system	Panasonic ActiveStor 18 (312 TB)
Peak performance	21.2 Tflops

The HP BL2x220c G7 server has a dual-node dual-channel architecture, each node has its own memory and processor. This architecture is closer to a distributed memory cluster (also known as an MPP cluster) because each node has its own memory space.

However, inside each node, multiple processors (two 6-core Intel Xeon X5670 processors) share memory, which forms a small SMP (symmetrical multiprocessor) system inside the node.

4.2. Testing under a single node

In this chapter, we create a single node with 12 processor cores and 8gb of running memory on the cluster as a test environment. In addition, in order to initially explore the synchronization performance of processes and threads in the hybrid model, matrix multiplication will be used as a benchmark. The size of the matrix sample is 2880*2880.

4.2.1 Coarse-grained parallelism and fine-grained parallelism

Coarse-grained parallelism and fine-grained parallelism are two different parallel methods in the field of parallel computing. They describe how to divide computing tasks into subtasks that can be executed simultaneously on multiple processors. The main difference between these two types of parallelism lies in the communication and synchronization overhead between subtasks.

Coarse-grained parallelism refers to the division of computing tasks into relatively large subtasks, which are executed in parallel on multiple processors. Due to the large subtasks, the processor can work independently for a long period of time, so the communication and synchronization overhead is relatively small. Coarse-grained parallelism is usually suitable for situations where communication costs between processors are high, such as in distributed memory systems (such as MPI).

The advantage of coarse-grained parallelism is that the communication and synchronization costs are relatively low, and good extensibility can be achieved. However, its limitation is that it needs to find independent subtasks large enough to keep the processor busy, otherwise it may lead to uneven load and performance degradation.

Fine-grained parallelism refers to the division of computing tasks into relatively small subtasks that are executed in parallel on multiple processors. Due to the small subtasks, the processor needs to communicate and synchronize frequently. Fine-grained parallelism is usually suitable for situations where communication

costs between processors are low, such as in shared memory systems (such as OpenMP).

The advantage of fine-grained parallelism is that it can make better use of processor resources and achieve higher parallelism. However, its limitation is that communication and synchronization costs may be higher, resulting in performance degradation.

In matrix multiplication, coarse-grained parallelism usually involves dividing the input matrices A and B into larger sub-matrices, and calculating the product of these sub-matrices in parallel on different computing nodes.

If in the hybrid model, in each MPI process, we can use OpenMP or pthread to achieve fine-grained parallelism to further accelerate matrix multiplication calculations. This involves dividing the submatrix multiplication task in each process into smaller subtasks and executing them in parallel on multiple threads within the process.

4.2.2 Process performance vs Thread performance

In order to explore the performance differences of matrix multiplication in process parallelism and thread parallelism under a single node, pure MPI, OpenMP, and Pthreads libraries will be used to parallelize matrix operations separately.

Under pure MPI (process parallelism), the rows of matrix A are divided into different MPI processes, and matrix B is broadcast to all MPI processes, so that each process is responsible for calculating a portion of the rows of the resulting matrix C. Finally, use the MPI_Gather function in the root process to collect and synchronize the results of other processes.

The OpenMP and Pthreads libraries are used to implement thread parallelism, which is reflected in the synchronous work between threads. Each thread is responsible for calculating a part of the matrix multiplication result. The task allocation between each thread is based on the number of rows in the matrix to achieve uniform load distribution.

Record and compare the running time of each program at different granularity, the results are shown in the following figure 23:

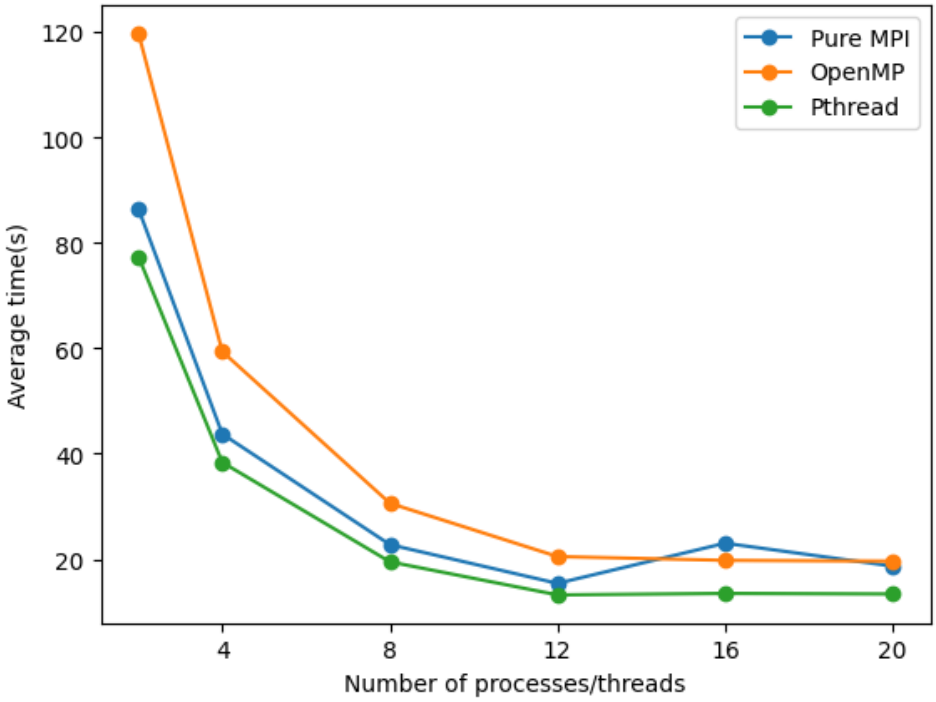

Figure 23 – Performance of processes and threads in parallel at different granularity

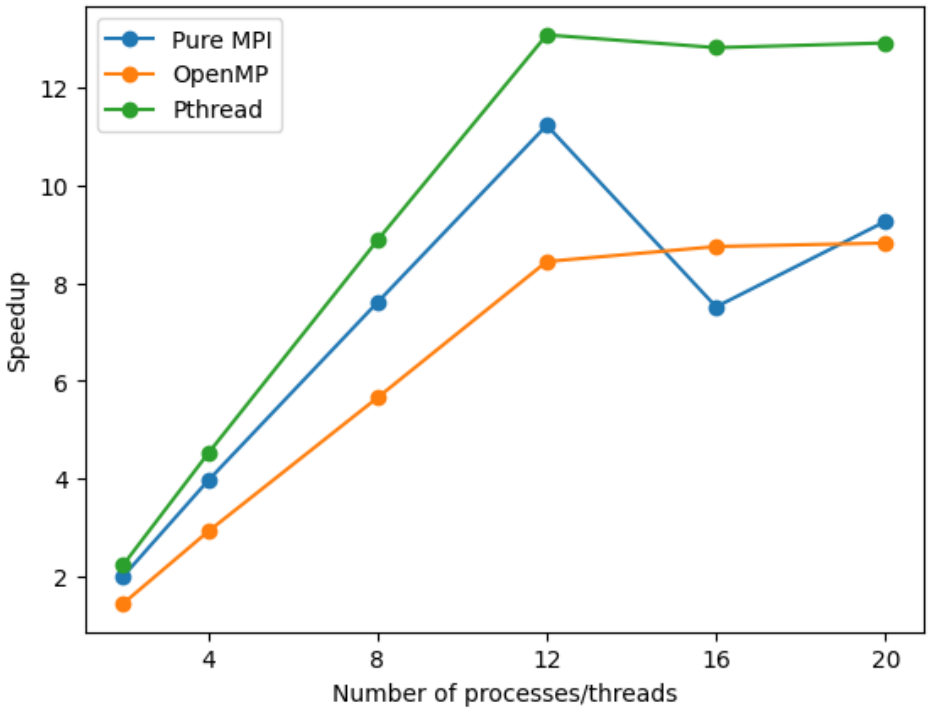

Figure 24 – Speedup ratio of process and thread

From the above results, we can intuitively see that Pthreads, as a lightweight and high-performance library, has parallel performance that exceeds MPI process and OpenMP.

The MPI process also has good performance at coarse-grained levels, even better than OpenMP.

However, at fine-grained levels, the performance of thread synchronization is better.

4.3. Hybrid model testing under cluster

In this chapter, we will analyze the performance of multithreaded MPI communication in a clustered environment and realize matrix multiplication under a hybrid MPI model. Due to the resource constraints of the cluster, the number of processor cores called by each user on the node at the same time cannot exceed 72. Therefore, in the following test, we will adjust the number of processes and threads of each node appropriately according to the task.

4.3.1 Multithreaded ring communication

MPI ring communication refers to the transfer of information through MPI interprocess communication in a ring process structure. In this communication mode, each process is both a sender and a receiver. Each process sends a message to the process on its left and receives a message from the process on its right.

In ring communication, the MPI_Send and MPI_Recv functions are usually used for point-to-point communication. The sending process uses MPI_Send to send messages to the process on its left, and the receiving process uses MPI_Recv to receive messages from the process on its right (see Figure 25).

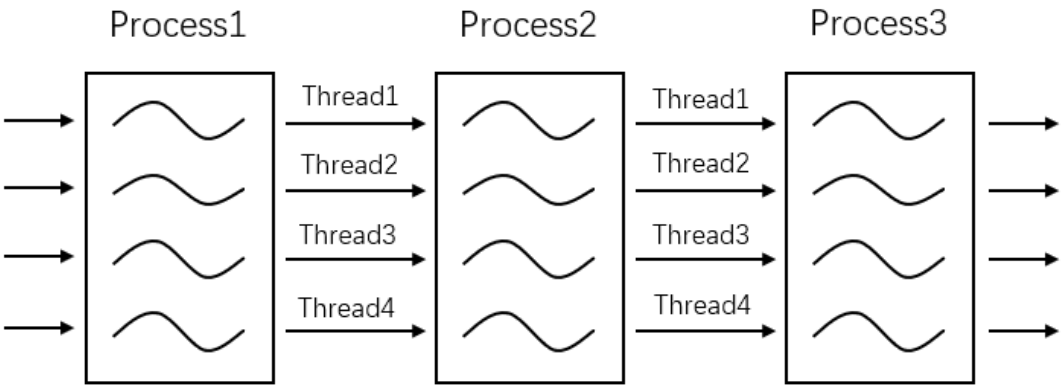

Figure 25 – Multithreaded ring communication

We create four threads in each process, send messages to the corresponding threads in the next process, and receive messages from the corresponding threads in the previous process. Use the MPI_Wtime function to record the time consumed by the thread communication in each process, and use the MPI_Reduce protocol to calculate the sum of the time in the root process. Get the result shown in the figure 26 below:

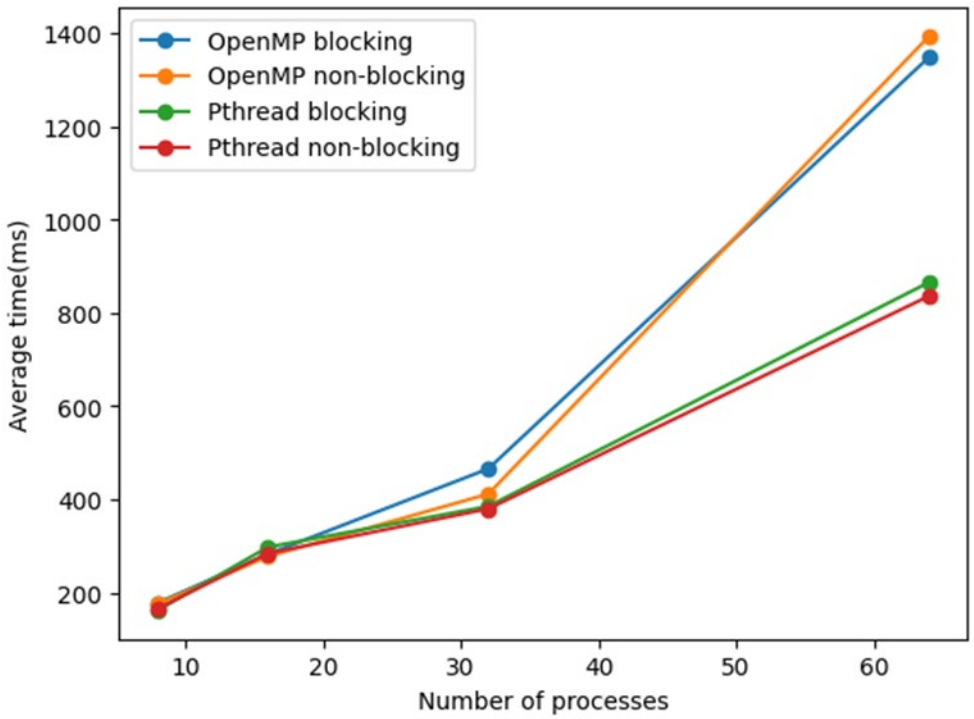

Figure 26 – Performance of Thread Parallelism in ring communication

From the results on the figure 26, it can be seen that with the growth of the number of processes, that is, the increase of the number of thread communications, Pthreads has more and more obvious synchronization performance advantages compared to OpenMP.

However, under this ring communication model, the performance of MPI blocking and non-blocking is almost indistinguishable.

4.3.2 Multithreaded linear broadcasting

Linear broadcast is a basic communication mode in MPI parallel computing. It involves the root process sending the same message directly to all other processes. Linear broadcast is straightforward to implement and intuitive, one notable characteristic is the number of communications involved. The root process needs to perform P-1 independent sending operations, where P represents the total number of processes. Consequently, the number of communications for linear broadcast grows linearly with the number of processes.

In massively parallel systems with a large number of processes, this linear growth in communications can result in performance bottlenecks. The root process may become a limiting factor in performance due to the significant number of independent sends it has to execute. Despite this drawback, linear broadcasting offers high stability and reliability, because it relies on basic point-to-point communication and lacks complex communication patterns.

Although linear broadcasting may not be the most efficient broadcasting strategy in certain cases, its simplicity and reliability make it a suitable choice for many application scenarios.

In situations where performance constraints are not critical and a straightforward communication pattern is desirable, linear broadcast can be an appropriate option.

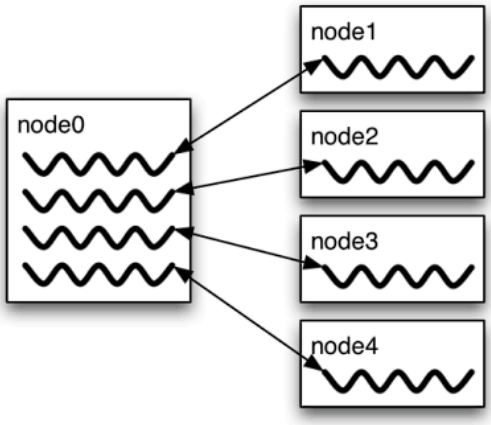

Figure 27 – Multithreaded linear broadcasting

We create 4 threads in the root process, and send an array of size 3000*3000 to other processes in parallel. The threads are all blocking communication. Record the time when the data sending in the main process is completed, and the result shown in the following figure 28 is obtained:

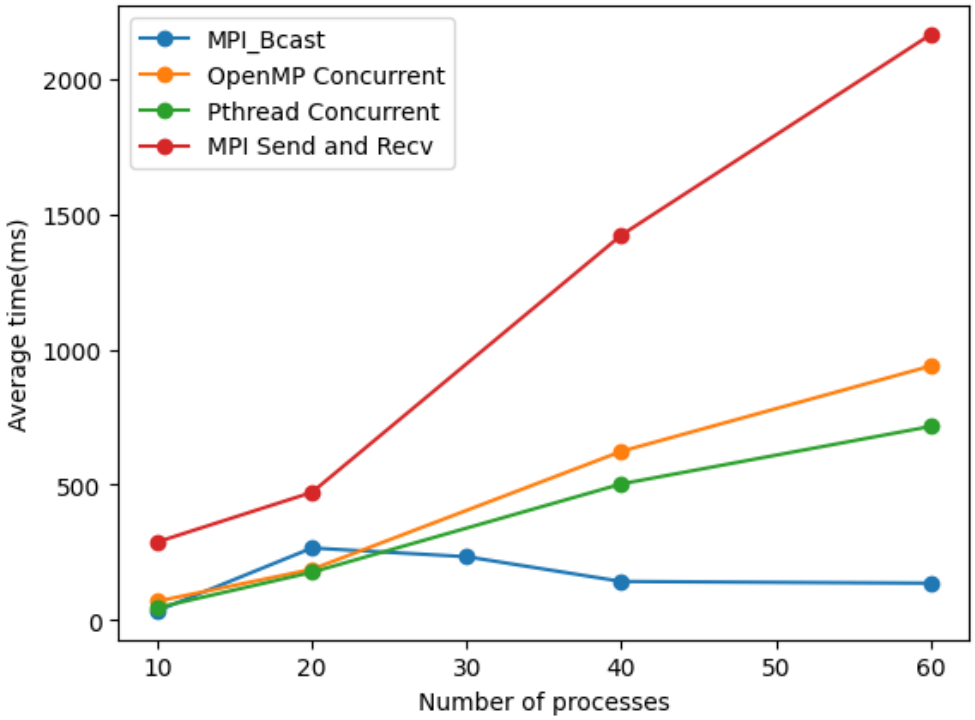

Figure 28 – Performance of MPI linear blocking broadcasting under multithreaded synchronization

Then calculate and get the Speedup ratio of OpenMP and Pthreads:

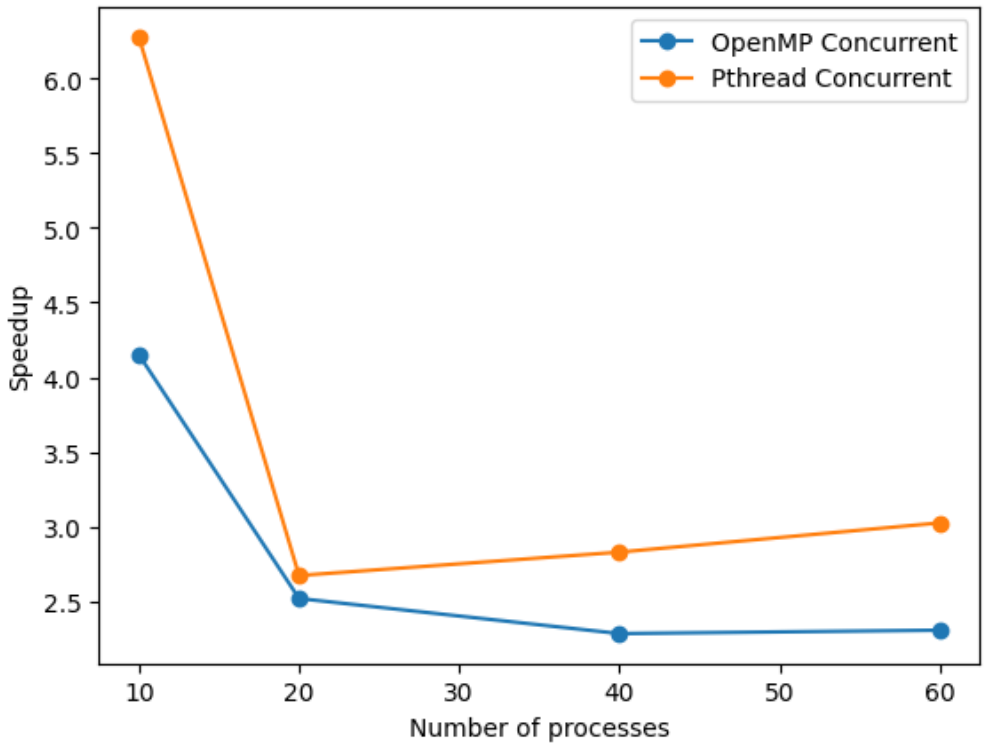

Figure 29 – The Speedup ratio of MPI linear blocking broadcasting under OpenMP and Pthread synchronization

The results in the figure 28 show that compared with single-threaded broadcasting, the performance of multithreading is significantly improved.

However, the MPI_Bcast function that comes with MPI can broadcast faster as the number of processes increases. This means that MPI_Bcast may use binary trees or other more complex algorithms for broadcasting to improve efficiency, rather than simply point-to-point data replication. The time complexity of this method is O (log n).

From Figure 29, we can see that the Speedup ratio of OpenMP and Pthreads was the largest at the beginning. However, Pthreads reaches its minimum when the number of processes is 20. After the number of processes is greater than 20, the Speedup ratio of OpenMP gradually tends to 2.

According to Amdahl's Law analysis, the Speedup ratio of Pthreads should eventually be closer to 2, because in this test we allocated two CPU cores to the root process.

4.3.3 Blocking performance vs non-blocking performance

In Figure 26 of Chapter 4.3.1, we compared the performance of blocking and non-blocking under multithreaded MPI ring communication, but the results were not ideal. Therefore, we continue to compare under the model of linear broadcasting.

Similarly, we create four threads for the root process, each thread uses the MPI_Isend function to send data to other processes non-blocking, and finally uses the MPI_Waitall command to send the completed operation synchronously.

Record and get the results of the figure 30 below：

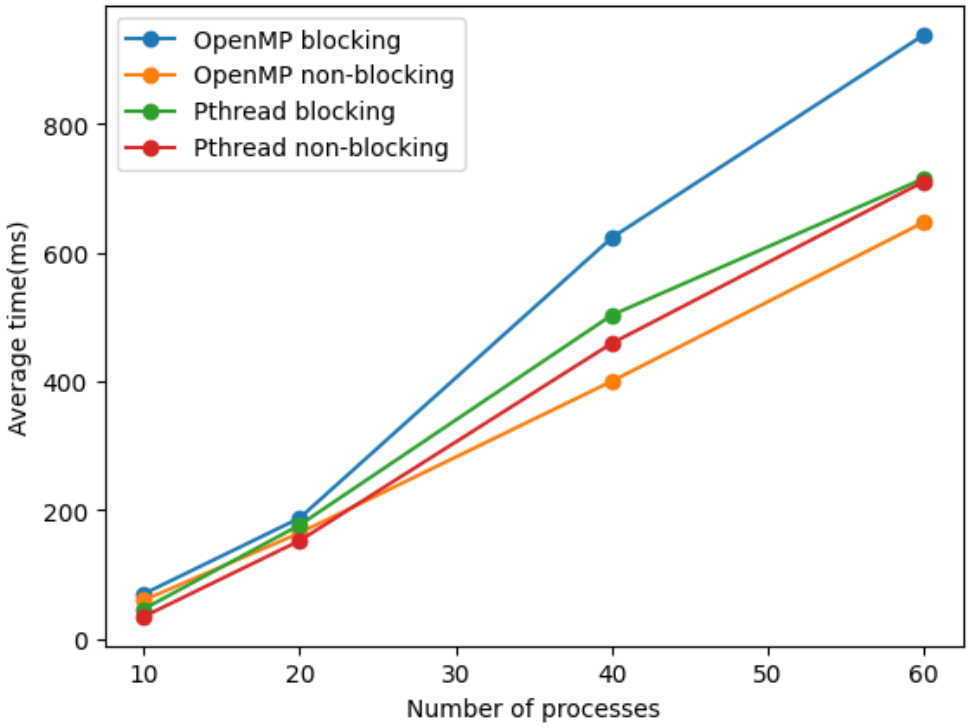

Figure 30 – Blocking and non-blocking under multithreaded linear broadcasting

From this Figure 30, we see that under the linear broadcast model, the performance of multithreaded non-blocking sending and receiving is obviously better than that of multithreaded blocking sending and receiving.

4.3.4 OpenMP performance vs Pthreads performance

Generally, Pthreads provides finer parallel control, allowing programmers to directly manage the creation, synchronization, and termination of threads, enabling them to handle more complex parallel tasks. In addition, as part of the POSIX standard, Pthreads can be used on all operating systems that support POSIX. However, this kind of fine-grained control comes at the expense of increasing programming complexity, while lacking advanced parallel features such as parallel loops and task scheduling.

In contrast, OpenMP greatly simplifies the complexity of parallel programming, parallelizes by compiling instructions, and automatically manages the life cycle of threads. Although OpenMP performs well in scenarios such as loop parallelization and task parallelization, the parallelism granularity of OpenMP is relatively coarse, and its performance may not be as good as Pthreads for tasks that require fine-grained control of thread behavior.

Analyze Figure 23, Figure 26, Figure 28 and Figure 30 in the previous chapters, it is not difficult for us to find that performance of Pthreads is better than OpenMP under the fine-grained parallelism of computing tasks, as well as under the blocking and non-blocking communication of MPI, which is consistent with the theory.

4.3.5 Matrix multiplication with hybrid model

In matrix multiplication, each part of the matrix A can be distributed to other processes through the MPI_Scatter function.

The MPI_Scatter function is a commonly used decentralized operation that distributes data from one process (usually the root process) to all other processes. The traditional implementation of MPI_Scatter is linear, that is, the root process will send data to all other processes in turn. However, this approach may become a performance bottleneck when the number of processes is large.

The MPI_Gather function collects data from multiple processes into one process. In the traditional reality, the communication mode of MPI_Gather is also linear, that is, each process sends data to the root process, and the root process receives data from its own other processes in turn.

To solve this problem, we aim to parallelize all MPI communication parts in traditional MPI matrix multiplication. We have designed two instance models to evaluate the performance of the hybrid MPI+threads approach. Both models use non-blocking communication, with one model using only OpenMP and the other combining both OpenMP and Pthreads.

By parallelizing the communication part using a combination of MPI and threads, our goal is to improve the overall performance of matrix multiplication. The hybrid model allows effective distribution and collection of data, reducing potential bottlenecks caused by sequential communication in traditional methods. The use of non-blocking communication further improves performance by enabling overlapping communication and computing tasks.

In the first hybrid MPI+OpenMP model (see Figure 31), the root process is not responsible for the calculation of the matrix, but only focuses on multithreaded receiving and sending data, collecting the calculation results of other processes and finally summarizing. First, in the parallel domain of the root process, the part of matrix A and the entire matrix B are distributed to other processes at the same time, and then use '#pragma omp barrier' to wait for all broadcast communications to complete, and finally use the MPI_Irecv function in parallel to receive the results, and write the variable results to array C.

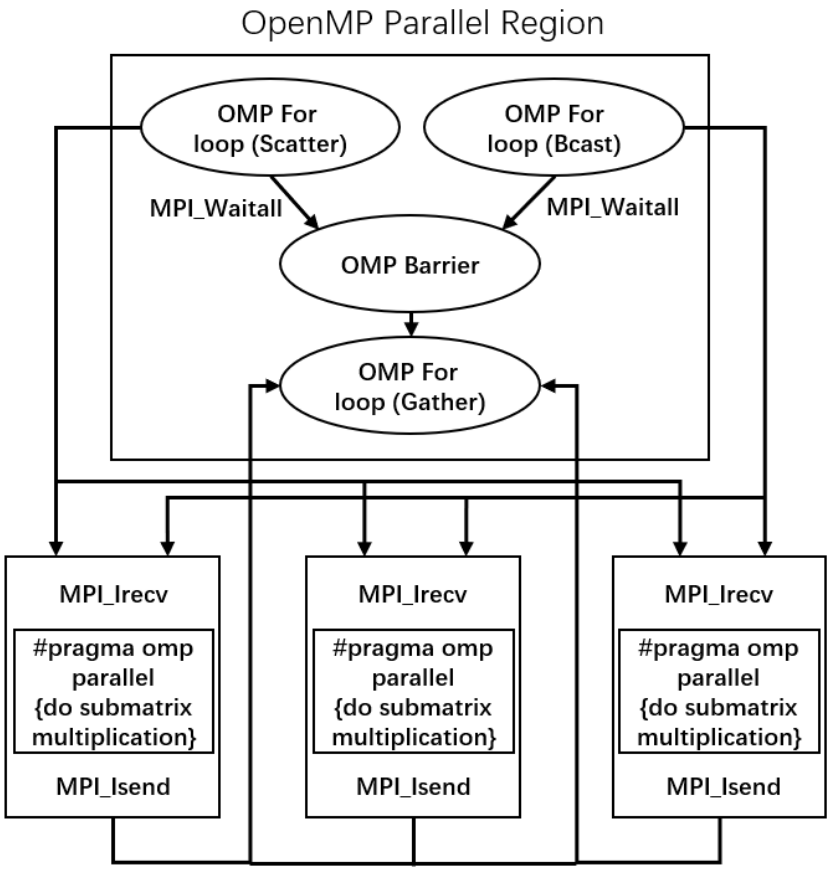

Figure 31 – A model for simultaneously parallelizing matrix multiplication and MPI communication using OpenMP in multiple nodes

In the second hybrid MPI+OpenMP+pthreads model (see Figure 32), the root process still participates in the operation of the first small piece of the matrix, and the parallelization of calculations within each process is still implemented by OpenMP. However, before the parallel domain of OpenMP, three threads are created with 'pthread_create'. The first thread is used to linearly distribute matrix A, the second thread calls MPI_Ibcast to broadcast matrix B, and the third thread waits after it is created until the end of the OpenMP working domain before starting to send local results to the root process. The working domain of OpenMP only starts after the first two threads have ended. In this way, each process can calculate while waiting for communication.

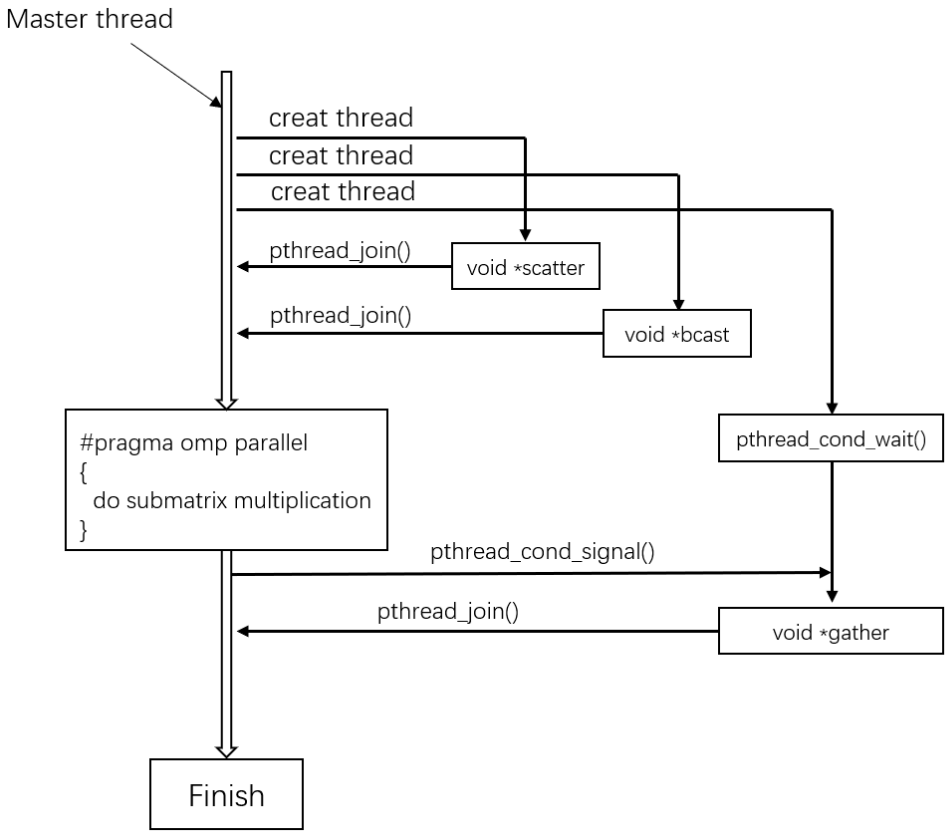

Figure 32 – A model of parallelizing matrix multiplication with OpenMP and parallelizing MPI communication with Pthreads in multiple nodes

In the above model, ‘pthread_cond_wait’ and ‘pthread_cond_signal’ are two functions used for inter-thread synchronization. These two functions are usually used together with ‘pthread_mutex’ to implement a more complex thread synchronization mechanism.

The third thread responsible for collecting the results will obtain the mutex before modifying the shared data in the critical area, and then wait for the condition variable. After the main thread has modified the shared data, it will send a signal of the condition variable, wake up the third thread, and finally release the mutex.

The matrix sample size of the experiment is 3072*3072. We keep the number of processor cores allocated to each process at 2, and the number of threads in the OpenMP parallel domain in each process at 4, the results of the following figure 33 are measured:

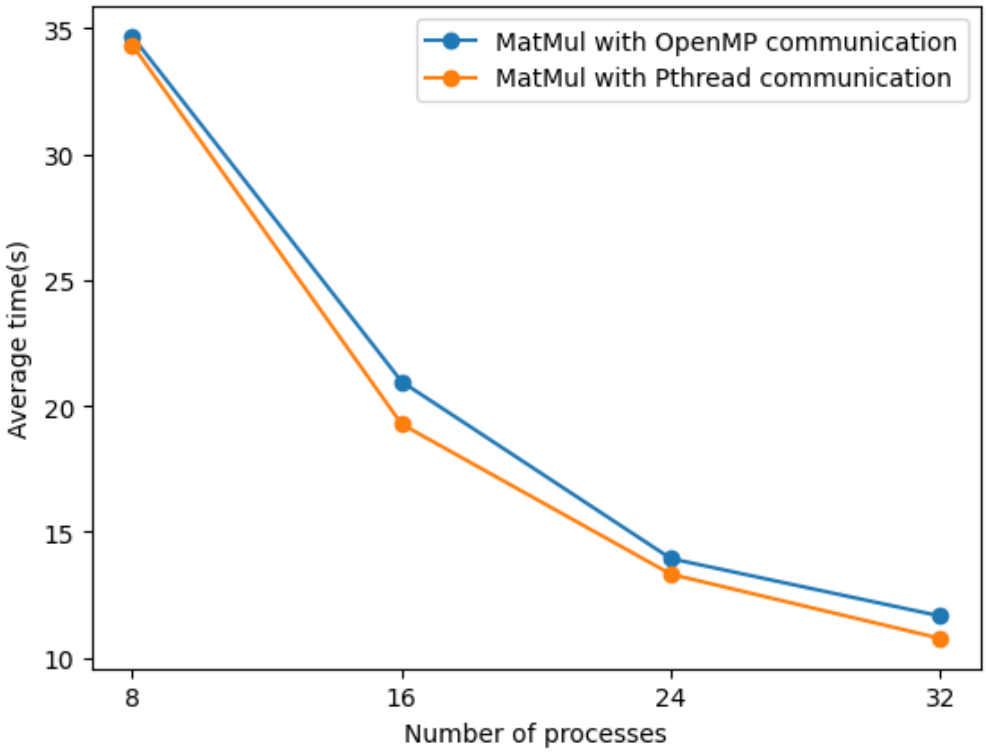

Figure 33 – Performance comparison of the two models

The final result is shown in the figure 33. Although most of the time of the program belongs to the calculation time, MPI communication time accounts for only a small part, but Pthreads still improves the degree of parallelism for communication compared to OpenMP, and further reduces the overhead of communication synchronization.

Therefore, Pthreads can better reduce the communication-to-calculation ratio of the entire parallel computing program.

COMMERCIALIZATION OF THE RESEARCH'S RESULTS

1. Executive Summary

My research project is hybrid MPI+threads parallel programming, which is mainly used in the field of High Performance Computing (HPC). HPC is a very important technical field, and it is widely used in various scientific research and business environments. In terms of hardware, HPC usually uses supercomputers, cluster systems, or cloud computing platforms.

Our venture, High Performance Computing Solutions (HPCS), aims to introduce advanced high-performance computing (HPC) and parallel computing solutions to the Russian market. With the growing demand for data processing and complex computations in various sectors such as scientific research, finance, media, healthcare, and energy, we recognize a significant opportunity for our specialized services and products.

Our primary offerings include customizable HPC hardware, software platforms, and technical support services which are designed to handle extensive data processing and complex computational tasks efficiently. Our products and services will support businesses and research institutions in accelerating their decision-making processes, enhancing productivity, and ultimately driving their innovation forward.

The HPC market in Russia exhibits a robust growth trajectory, driven by increasing demand from research institutions and large enterprises, coupled with government initiatives to foster HPC development. However, there is currently a limited supply of efficient and reliable HPC solutions in the market, creating an ideal environment for HPCS to thrive.

With our cutting-edge technology, experienced team, and customer-oriented approach, we plan to establish a strong foothold in the Russian HPC market and become a leading provider of HPC and parallel computing solutions. Our initial funding will be directed towards hardware acquisition, software development, and

marketing initiatives, with the aim to achieve break-even in the first year and steady revenue growth in the following years.

Our mission is to empower Russian businesses and research institutions with the computational power they need to excel in their respective fields, making a significant contribution to the technological advancement in Russia.

2. Product description

2.1 Purpose of the Product

High Performance Computing Solutions (HPCS) provides a suite of high-performance computing (HPC) and parallel computing solutions designed to handle extensive data processing and complex computational tasks. These solutions serve to accelerate decision-making processes, enhance productivity, and fuel innovation in various sectors such as scientific research, finance, media, healthcare, and energy.

2.2 Main Product Features

HPC Hardware Systems: Our HPC systems feature cutting-edge processors, high-speed memory, and large storage capacities. They are customizable and scalable to meet diverse computational needs.

Software Platforms: Our proprietary software platforms optimize the performance of HPC systems, facilitate efficient utilization of computational resources, easy job scheduling, and seamless data management.

Technical Support Services: We offer comprehensive support services that include installation and setup of HPC systems, training of personnel, maintenance and troubleshooting, and consulting services.

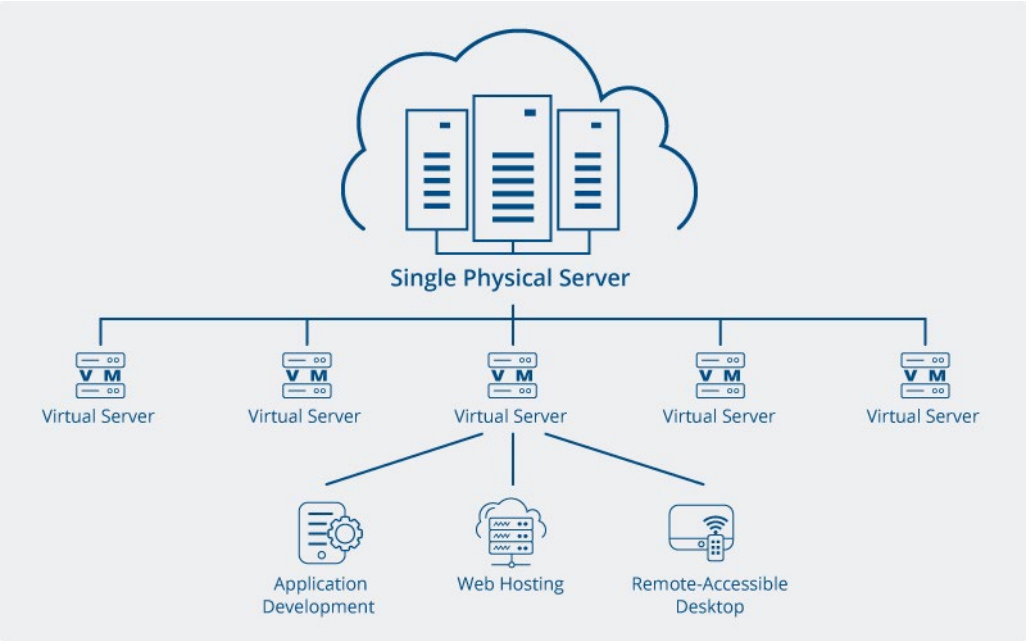

Figure 34 – HPC system

2.3 Consumer Properties of the Product

Our HPC solutions are user-friendly, reliable, and efficient. They are designed to be scalable and customizable, allowing our clients to adjust their computational capacity based on their changing needs.

2.4 Product Competitive Advantages

Our HPC solutions offer several competitive advantages:

1. Cutting-edge technology: Our products feature the latest advancements in HPC technology.

2. Customizability: We provide solutions that can be tailored to the specific needs of our clients.

3. Comprehensive support: We provide end-to-end support, ensuring our clients can focus on their core work while we handle the technical aspects.

4. Experienced team: Our team comprises industry experts with deep knowledge and experience in HPC.

2.5 The Main Consumers and Uses of Products

Our main consumers are research institutions and large enterprises in Russia that require high computational power for tasks such as scientific modeling, data analysis, and machine learning.

2.6 Assortment and Structure of Production

Our product assortment includes a range of HPC hardware systems, software platforms, and technical support services. We have partnerships with leading hardware manufacturers and our software solutions are developed in-house.

2.7 Legal Protection of Products

Our software platforms are proprietary and protected under intellectual property laws. We ensure compliance with all relevant regulations in our operations.

2.8 Service

We provide comprehensive technical support services including installation and setup of HPC systems, training of personnel, regular maintenance, troubleshooting, and consulting services. Our service team is dedicated to ensuring the smooth operation of our HPC solutions at client sites.

3. Marketing plan

3.1 Market Analysis

The Russian high-performance computing (HPC) market is poised for significant growth, largely driven by the increasing demand from research institutions, large enterprises, and governmental initiatives to foster technological advancements. Despite the growing demand, there is a gap in the supply of efficient, reliable, and user-friendly HPC solutions, which presents a substantial opportunity for our venture, High Performance Computing Solutions (HPCS).

Key sectors with high computational needs such as scientific research, finance, healthcare, and energy will be our primary target. We will also pay attention to emerging sectors where the application of HPC could provide significant benefits.

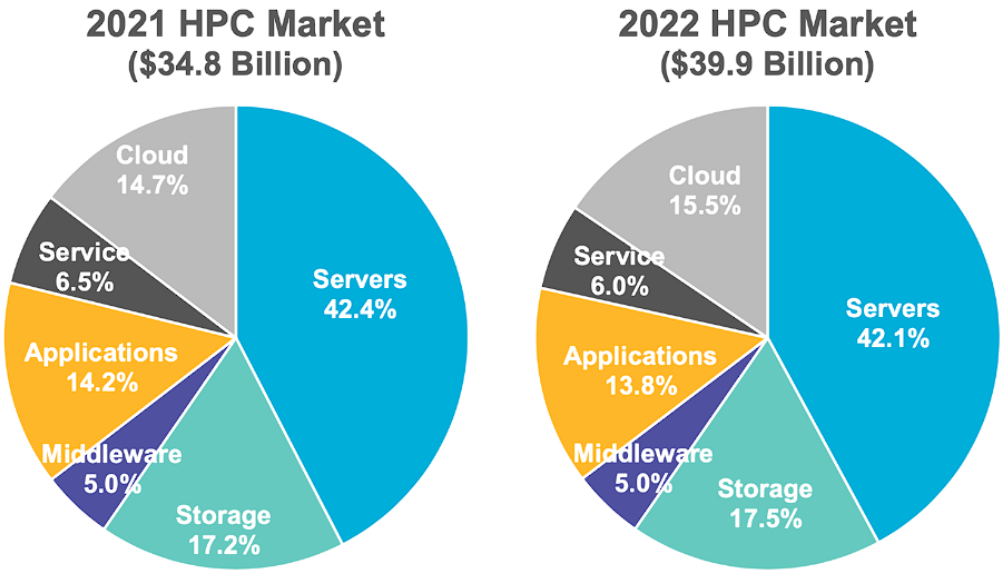

Figure 35 – HPC Market

3.2 Trade Policy

Our trade policy will be centered on developing strong relationships with clients and partners. We will focus on providing high-quality products and excellent customer service to establish a reputable brand. In terms of international trade, we will ensure compliance with all relevant laws and regulations. We also plan to collaborate with local institutions and distributors to broaden our reach.

In addition to our core products, we offer comprehensive technical support services, including installation, training, maintenance, and troubleshooting. We also provide consulting services to help our customers choose the right HPC solutions for their specific needs. Our aim is to provide end-to-end solutions that enable our customers to fully leverage the benefits of HPC technology.

Our brand represents quality, innovation, and customer-centricity. We are committed to providing high-quality products and exceptional customer service, and we continuously strive to innovate and stay at the forefront of HPC technology. Our goal is to build a strong brand reputation in the Russian HPC market and be recognized as a trusted and reliable provider of HPC solutions.

3.3 Price policy

Our pricing strategy will be value-based, reflecting the superior quality, reliability, and efficiency of our HPC solutions. We will conduct thorough market research to ensure our prices are competitive and provide good value for our clients. Furthermore, we will offer scalable solutions to cater to a variety of budgetary needs. For larger contracts or long-term partnerships, we may consider providing flexible payment terms or volume discounts.

Table 1 – Product price

Computing servers	1.04 RUB per CPU core hour 1.04 RUB per 6 GB of memory/hour 43.37 RUB per GPU hour
Storage space	6938 RUB per 1 TB per year 13877 RUB per 1 TB per year (2x replicated data) 2602 RUB per 1 TB per year (A copy stored on tape) 16479 RUB per 1 TB per year (Replicated+tape stored data)

Virtual server hosting	0.26019 RUB per core hour 0.11275 RUB per RAM (GB) hour 11968.54 RUB per SSD storage (TB) hour

3.4 Promotion plan and budget

Our promotional activities will include online marketing (SEO, social media, content marketing), offline marketing (networking at industry events, direct outreach), and advertising (online platforms, industry magazines, trade shows).

Given the importance of promotion in achieving our business objectives, we will allocate a significant portion of our initial budget to promotional activities. We aim to use our promotion budget efficiently by focusing on marketing channels that offer the best return on investment. We will continuously monitor and adjust our promotion budget based on the effectiveness of our promotional activities and changes in the market conditions.

Table 2 – Promotion budget

Promotion tools	The cost of the first year (RUB)
Search engine	400000
Social media	200000
Online advertising	400000
Industry events	200000
Industry magazine	200000
Trade show	100000
Total	1500000

4. Production plan

4.1 Data Center Costs

The data center is a crucial component of our high-performance computing solutions. We will need to secure a data center facility that offers sufficient space, power, cooling, and security to accommodate our HPC systems. The cost of the data center will include rental fees, utility expenses, maintenance costs, and any necessary upgrades or improvements to the facility.

Table 3 – Data Center Costs

Expense item	Cost per year (RUB)
Computer room rent	1000000
Electricity bill	300000
Hardware maintenance	500000
Software maintenance	600000
Network communication costs	400000
Backup power	20000
Total	2820000

4.2 The Need for Production Equipment

We expect to purchase more than 500 computers to form a computer cluster.

To deliver efficient and reliable HPC solutions, we will require state-of-the-art production equipment. This includes high-performance servers, networking equipment, cooling systems, and backup power supplies. The cost of acquiring, installing, and maintaining this equipment will be factored into our production budget.

Table 4 – Equipment cost

Purchase items	Purchase cost (RUB)
Cluster hardware	20000000

Network equipment	1000000
Power facilities	600000
Software license	400000
Total	22000000

4.3 Labor Resources

Our production team will be responsible for assembling, testing, and maintaining our HPC systems. We will need skilled technicians, engineers, and other production staff to manage these tasks. The cost of recruiting, training, and retaining these labor resources will be included in our production expenses.

Table 5 – Labor Resources

Position	Numbers	Salary per year (RUB)
Managers	5	5*2000000
Software engineer	40	40*1500000
Hardware engineer	20	20*1200000
Marketing	10	10*800000
Finance and Human Resources	5	5*800000
Total	80	106000000

4.4 Production Cost

The production cost will comprise the following components:

1. Data center costs: Including rent, utilities, maintenance, and upgrades.

2. Equipment costs: Including the acquisition, installation, and maintenance of servers, networking equipment, cooling systems, and backup power supplies.

3. Labor costs: Including salaries, benefits, and training expenses for the production team.

4. Raw material costs: Including the cost of components and materials required for the assembly of our HPC systems.

5. Quality control and testing: Including the cost of implementing quality control measures and testing equipment to ensure the reliability and performance of our HPC solutions.

Our production plan aims to optimize the use of resources, minimize production costs, and maintain a high level of quality and reliability in our HPC solutions. We will regularly review and adjust our production plan based on market conditions, customer needs, and technological advancements in the industry.

5. Organizational plan

5.1 Organizational structure

High Performance Computing Solutions (HPCS) will adopt a functional organizational structure to promote efficiency and specialization. The main divisions will include:

Management: Oversees the entire operation and guides the strategic direction of HPCS. This includes the CEO, CFO, and other top executives.

Product Development: Responsible for developing and improving our HPC hardware systems and software platforms. This division will consist of hardware engineers, software developers, and product managers.

Sales and Marketing: Tasked with promoting our products and services, attracting new clients, and maintaining relationships with existing ones. This team will include sales representatives, marketing specialists, and customer service representatives.

Technical Support: Provides installation, training, maintenance, and troubleshooting services to our clients. This team will include technical support engineers and trainers.

Operations and Logistics: Handles procurement, inventory management, and delivery of our products. This division will consist of operations managers and logistics coordinators.

Human Resources: Responsible for hiring, training, and managing the welfare of our employees.

Finance and Accounting: Manages financial planning, budgeting, accounting, and ensuring financial compliance.

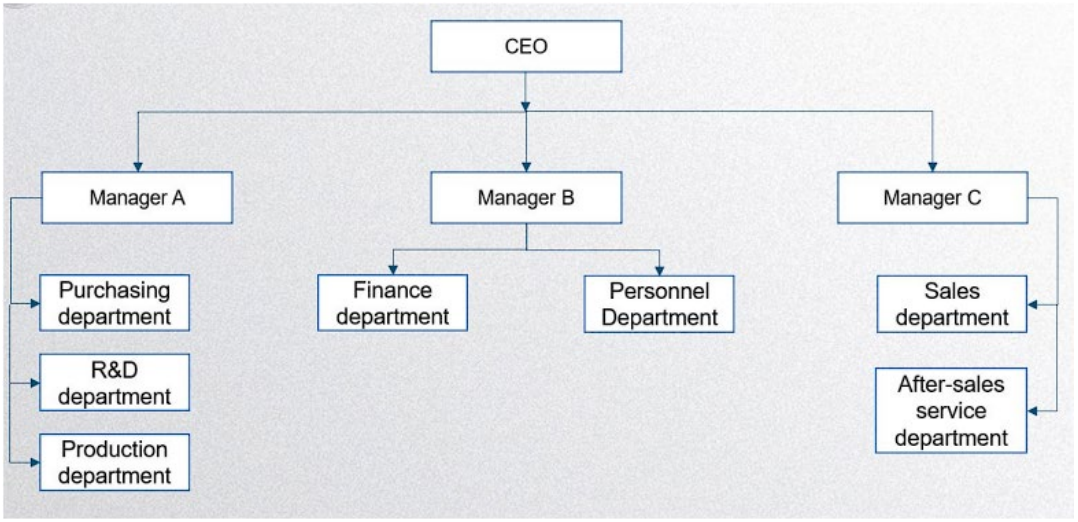

Figure 36 – Organizational structure

5.2 Staffing Plan

We aim to hire a diverse and talented team that shares our vision and commitment to providing high-quality HPC solutions. We will invest in the training and development of our employees to ensure they have the necessary skills and knowledge to perform their roles effectively.

5.3 Location

Our headquarters will be located in St. Petersburg to ensure access to top talent and potential clients. We will also establish local offices in key markets across Russia to better serve our clients.

6. Financial Plan

6.1 sales plan

Our sales plan aims to use our high-performance computing (HPC) solutions to meet the needs of the Russian market. Our goal is to acquire at least 1,500 customers in the first year and double this number in the next two years. We will promote our products through network marketing, partnerships and participation in industry events.

6.2 Financial breakdown

Our financial rules will list our income and expenses in detail, including our sales revenue, R&D costs, production costs, marketing expenses, administrative expenses and employee salaries.

Table 6 – Financial breakdown

Profit	Year1	Year2	Year3	Year4
Sales of HPC solutions	60 000 000	72 000 000	86 400 000	103 680 000
Hardware resources	15 000 000	18 000 000	20 000 000	20 000 000

Technical service revenue	30 000 000	36 000 000	43 200 000	44 000 000
Software revenue	28 000 000	40 000 000	40 000 000	40 000 000
Investment income	0	0	0	0
Other income	0	0	0	0
Cash inflows	133 000 000	166 000 000	189 600 000	207 680 000
Expenditure	Year1	Year2	Year3	Year4
Employee salary	106 000 000	106 000 000	106 000 000	106 000 000
Purchase cost	22 000 000	1 000 000	500 000	500 000
Data center operating costs	2 820 000	2 820 000	2 820 000	2 820 000
Promotion cost	1 500 000	500 000	200 000	200 000
Investments	0	0	0	0
VAT	400 000	800 000	1 000 000	1 000 000
Cash outflows	132 720 000	111 120 000	110 520 000	110 520 000
Net cash flow	280 000	54 880 000	79 080 000	97 160 000

We expect that cash flow will remain positive in the first year, mainly due to our stable sales revenue and reasonable cost and expense control. We will conduct regular cash flow forecasts to ensure that we have enough cash to support our operations and growth.

Our break-even analysis will determine the sales volume we need to achieve in order to cover our fixed and variable costs and start to realize profits. We expect to reach the break-even point at the end of the first year, and we expect that as sales grow, our profit margins will also increase year by year.

The financial plan is designed to support our business plan and ensure that our company remains financially sound while achieving growth. We will regularly review and update our financial plan based on our business performance and market changes.

7. Economic Results

Net Present Value (NPV) is a method used to evaluate the financial benefits of an investment project or business plan. NPV considers the factor of the depreciation of money over time, which is the so-called time value. In simple terms, NPV calculates the net present value of future cash inflows and outflows. If NPV is positive, it means that the expected cash inflow exceeds the cash outflow, and the investment project or business plan can bring profits. Conversely, if NPV is negative, it means that the expected cash outflow exceeds the cash inflow, and the investment project or business plan may lose money.

The calculation formula for NPV is:

$$NPV = \sum_{t=1}^{T} \frac{CF_t}{(1+r)^t} - I \tag{1}$$

CFt is the cash flow in period t (cash inflows are positive, cash outflows are negative);

r is the discount rate, which is the interest rate used to discount future cash flows to the present;

t is the number of periods (usually in years);

I is the initial investment or project cost.

Our investment cost in the first year is about 14,000,000 rubles. Assuming a discount rate of 5%, then: **NPV = 58 290 494.19**.

IRR is the abbreviation of Internal Rate of Return and is a financial indicator used to measure the return on investment projects. It is a discount rate that makes the net present value of an investment project equal to zero. The higher the IRR, the higher the expected return on the investment project, and therefore more attractive. Under normal circumstances, if the IRR is higher than the investor's opportunity cost or capital cost, the investment project is attractive. Similar to NPV (net present value), IRR also considers the concept of time value.

Formula for IRR:

$$NPV = 0 = \sum_{t=1}^{n}\left[\frac{CF_n}{(1+IRR)^t}\right] + CF_0 = \frac{CF_1}{(1+IRR)^1} + \frac{CF_2}{(1+IRR)^2} + \cdots \frac{CF_n}{(1+IRR)^t} + CF_0 \quad (2)$$

IRR result of our project:

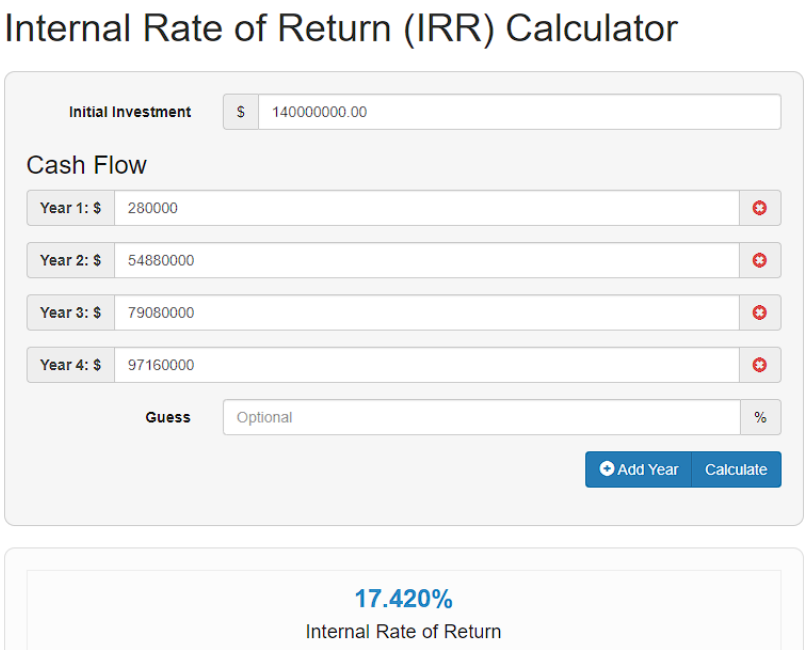

Figure 37 – IRR

8. Risk assessment

The most important risks of the project are shown in the table:

Table 7 – Risk

Risk	Impact	How to reduce the outcome
Technical risk	High-performance computing is a rapidly developing field, and the emergence of new technologies and algorithms may make existing products and services obsolete.	In order to remain competitive, continuous investment in research and development is required to keep up with the pace of technology.
Market risk	Although the Russian high-performance computing market is currently growing rapidly, changes in market demand are still a major risk. If demand drops suddenly, it may have a negative impact on the business.	Conduct regular market research to understand and predict changes in customer needs. At the same time, it is possible to consider diversifying products and services to reduce dependence on the single market.
Competitive risk	More companies may enter this field, intensifying competition.	Build brand awareness, provide excellent customer service, and

		continue to develop leading products and services to stand out from the competition. Partnerships and strategic alliances may also help improve competitiveness.
Regulatory risk	Changes in policies and regulations may affect the provision of high-performance computing services. For example, changes in data protection and privacy regulations may increase operating costs.	Maintain an understanding of relevant regulations and consult legal experts to ensure compliance. At the same time, plans can be preset to respond to possible regulatory changes.
Financial risk	The equipment and R&D costs of high-performance computing are very high. If insufficient funds are available, it may affect the company's operations and expansion plans.	Develop a detailed financial plan, including budget and cash flow forecasts. Seek the support of investors, or consider bank loans to obtain the necessary funds.

9. Conclusion

The business plan established a new company called "High performance Computing Solutions (HPCS)", described in detail the company's goals and its advantages over other companies, and defined target customers. The product

description section reveals the company's product type, its functions, consumer attributes, advantages, and the services provided with the product. The production plan shows the resources required for the implementation of the project, and summarizes the investment cost and current production cost. The financial plan shows the calculation results of the company's income, expenses, and net cash flow.

Our team has a wealth of experience and expertise. We believe that through our efforts, we will be able to achieve significant success in the Russian HPC market, provide customers with excellent products and services, and promote their innovation process.

CONCLUSION

This paper expounds the basic principles of the hybrid MPI+threads model, compares the performance of processes and threads in coarse-grained parallelism, explores the synchronization efficiency of multithreaded MPI communication under the hybrid model, and completes the comparison between blocking and non-blocking communication, and finally proposes two methods to solve matrix multiplication using the hybrid model.

Experiments conducted on a single cluster node emphasized the superior performance of Pthreads. As a lightweight, high-performance library, Pthreads consistently outperformed MPI processes and OpenMP, especially in the context of fine-grained parallelism. Although MPI processes exhibited commendable performance at the coarse-grained level, occasionally surpassing OpenMP, their performance in fine-grained tasks was not as impressive.

In multi-node cluster experiments focusing on multithreaded MPI communication, we found that parallelized MPI communication can yield substantial acceleration compared to its single-threaded counterpart. While non-blocking mode didn't show significant benefits in ring communication, its time-saving advantage was clearly evident in linear broadcasting mode. Furthermore, Pthreads proved to have superior thread synchronization efficiency under MPI multithreaded communication compared to OpenMP.

Our parallel computing experiments in matrix multiplication revealed that, despite the majority of the program execution time being dominated by computation, with MPI communication only making up a small part, Pthreads still managed to enhance the degree of parallelism for communication relative to OpenMP. Consequently, this led to a decrease in communication synchronization overhead, indicating that Pthreads can more effectively diminish the communication-to-calculation ratio in the entire parallel computing program.

Overall, this study provides an in-depth comparative analysis, and comprehensively evaluates the parallel performance of MPI, Pthreads, and OpenMP

in the hybrid model. The results highlighted their respective comparative advantages and disadvantages. In our test environment, Pthreads demonstrated excellent overall performance. The results of this research not only enrich our understanding of these parallel programming models, but also provide a reference for future parallel computing optimization research.

BIBLIOGRAPHY

1. Song T. et al. Hybrid Message Passing with Performance-Driven Structures for Facial Action Unit Detection // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2021.

2. He Y., Ding C.H.Q. MPI and OpenMP paradigms on cluster of SMP architectures: The vacancy tracking algorithm for multi-dimensional array transposition // Proceedings of the International Conference on Supercomputing. 2002. Vol. 2002-November.

3. Terboven C. et al. OpenMP: Heterogenous Execution and Data Movements // Lecture Notes in Computer Science. 2015. Vol. 9342.

4. Pacheco P. An Introduction to Parallel Programming // An Introduction to Parallel Programming. 2011.

5. Zhou H. et al. Collectives in hybrid MPI+MPI code: Design, practice and performance // Parallel Comput. 2020. Vol. 99.

6. Tabakov A. V., Paznikov A.A. Algorithms for optimization of relaxed concurrent priority queues in multicore systems // Proceedings of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2019.

7. Netto M.A.S. et al. HPC Cloud for Scientific and Business Applications // ACM Comput Surv. 2019. Vol. 51, № 1.

8. Culler D. et al. LogP: Towards a Realistic Model of Parallel Computation // ACM SIGPLAN Notices. 1993. Vol. 28, № 7.

9. Goudreau M.W. et al. Portable and efficient parallel computing using the BSP model // IEEE Transactions on Computers. 1999. Vol. 48, № 7.

10. Yakubov S. et al. Hybrid MPI/OpenMP parallelization of an euler-lagrange approach to cavitation modelling // Comput Fluids. 2013. Vol. 80, № 1.

11. Zambre R. et al. Logically Parallel Communication for Fast MPI+Threads Applications // IEEE Transactions on Parallel and Distributed Systems. 2021. Vol. 32, № 12.

12. Ibanez D., Dunn I., Shephard M.S. Hybrid MPI-thread parallelization of adaptive mesh operations // Parallel Comput. 2016.

13. Gropp W., Thakur R. Thread-safety in an MPI implementation: Requirements and analysis // Parallel Comput. 2007. Vol. 33, № 9.

14. Saillard E., Carribault P., Barthou D. MPI thread-level checking for MPI+OpenMP applications // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2015.

15. Gerzhoy D. et al. Nested MIMD-SIMD parallelization for heterogeneous microprocessors // ACM Transactions on Architecture and Code Optimization. 2019. Vol. 16, № 4.

16. Zambre R., Chandramowlishwaran A. Lessons Learned on MPI+Threads Communication // International Conference for High Performance Computing, Networking, Storage and Analysis, SC. 2022.

17. Poolla C., Saxena R. On extending Amdahl's law to learn computer performance // Microprocess Microsyst. 2023. Vol. 96.

18. Volosatova T., Kiselev I., Knyazeva S. MULTITHREADING IN POSIX STANDARD // East European Scientific Journal. 2022.

19. Amer A. et al. Lock contention management in multithreaded MPI // ACM Transactions on Parallel Computing. 2019. Vol. 5, № 3.

20. Thakur R., Gropp W. Test suite for evaluating performance of multithreaded MPI communication // Parallel Comput. 2009. Vol. 35, № 12.

21. Grant R.E. et al. Finepoints: Partitioned multithreaded MPI communication // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2019. Vol. 11501 LNCS.

22. Paznikov A.A., Kupriyanov M.S. Adaptive MPI collective operations based on evaluations in LogP model // Procedia Computer Science. 2021. Vol. 186.

23. Lilja D.J. A multiprocessor architecture combining fine-grained and coarse-grained parallelism strategies // Parallel Comput. 1994. Vol. 20, № 5.

24. Dang H.V. et al. Advanced Thread Synchronization for Multithreaded MPI Implementations // Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017.

25. Amer A. et al. Locking Aspects in Multithreaded MPI Implementations // Implementations ACM Trans. Parallel Comput. 2016.

26. Luo M. et al. Initial study of multi-endpoint runtime for MPI+OpenMP hybrid programming model on multi-core systems // ACM SIGPLAN Notices. 2014. Vol. 49, № 8.

27. Sala K. et al. Integrating blocking and non-blocking MPI primitives with task-based programming models // Parallel Comput. 2019. Vol. 85.

28. Nazaruk V., Rusakov P. Blocking and Non-Blocking Process Synchronization: Analysis of Implementation // Scientific Journal of Riga Technical University. Computer Sciences. 2012. Vol. 44, № 1.

29. Al-Tawil K., Moritz C.A. Performance Modeling and Evaluation of MPI // J Parallel Distrib Comput. 2001. Vol. 61, № 2.

30. Zharikov V. V. et al. Adaptive Barrier Algorithm in MPI Based on Analytical Evaluations for Communication Time in the LogP Model of Parallel Computation // 2018 International Multi-Conference on Industrial Engineering and Modern Technologies, FarEastCon. 2018.

31. Pacheco P.S., Malensek M. Shared-memory programming with Pthreads // An Introduction to Parallel Programming. 2022.

32. Löwe W., Zimmermann W. Upper time bounds for executing PRAM-programs on the LogP-machine // Proceedings of the International Conference on Supercomputing. 1995. Vol. Part F129361.

33. Single instruction, multiple data. Wikipedia - [Electronic resource]. URL: https://en.wikipedia.org/wiki/Single_instruction,_multiple_data (date of access: 08.04.2023).

34. Multiple instruction, multiple data. Wikipedia - [Electronic resource]. URL: https://en.wikipedia.org/wiki/Multiple_instruction,_multiple_data (date of access: 08.04.2023).

35. Parallel Computing and Crystallography - [Electronic resource]. URL: https://cse.buffalo.edu/faculty/miller/Talks_HTML/Erice-Parallel/sld015.htm (date of access: 11.04.2023).

36. China Trumps Top500 with Sunway TaihuLight - [Electronic resource]. URL: https://www.karlrupp.net/2016/06/china-trumps-top500-with-sunway-taihulight (date of access: 13.04.2023).

37. Digital Transformation with Amdahl's & Gunther's Law - [Electronic resource]. URL: https://blog.knoldus.com/digital-transformation-with-amdahls-gunthers-law (date of access: 14.04.2023).

38. Gustafson's law. Wikipedia - [Electronic resource]. URL: https://en.wikipedia.org/wiki/Gustafson%27s_law (date of access: 14.04.2023)

39. MPI Communications. Distributed Computing Fundamentals - [Electronic resource]. URL: http://selkie.macalester.edu/csinparallel/modules/DistributedMemoryProgramming/build/html/MPICommunication/MPICommunication.html (date of access: 16.04.2023).

40. OpenMP Programming Model | LLNL HPC Tutorials - [Electronic resource]. URL: https://hpc-tutorials.llnl.gov/openmp/programming_model (date of access: 18.04.2023).

41. POSIX Thread APIs – Tech Access Info - [Electronic resource]. URL: https://techaccess.in/2021/05/14/posix-thread-apis (date of access: 18.04.2023)

42. INTERMEDIATE MPI - [Electronic resource]. URL: https://enccs.github.io/intermediate-mpi/ (date of access: 23.04.2023).