PAPER • OPEN ACCESS

Performance analysis of parallel programming models for C++

To cite this article: Guang Zeng 2023 *J. Phys.: Conf. Ser.* **2646** 012027

View the article online for updates and enhancements.

Performance analysis of parallel programming models for C++

Guang Zeng

Nanjing University, Nanjing, China

201250014@smail.nju.edu.cn

Abstract: As multicore processors become more common in today's computing systems and parallel programming models are enriched, programmers must consider how to choose the appropriate and parallel programming model when writing parallel code. The purpose of this paper is to compare and analyze the performance gap between different C++ parallel programming models, such as C++ standard library threads, OpenMP and Pthreads, in terms of matrix operations. The experiments use different libraries to implement matrix multiplication separately and then analyze their performance. The experimental data show that the data size has a significant impact on the performance of the different models. For very small matrices of size magnitude less than or close to the number of threads, the performance of parallel implementations is much lower than that of serial implementations. For small matrices with magnitudes larger than the number of threads, the C++ standard library threads outperforms Pthreads and OpenMP due to its lightweight thread performance on relatively small matrices. pthreads shows the best performance on very large matrices due to its fine-grained control over thread management, communication, and synchronization operations. openMP's is not as stable as the other two libraries, especially for smaller matrices. This paper provides a comparative analysis that can help programmers choose the most appropriate library for their specific computational needs.

Keywords: Parallel programming, Matrix operations, OpenMP.

1. Introduction

With the rapid development of processor technology, multi-core processors have become increasingly prevalent in today's computing systems [1]. However, this shift towards parallelism also brings new challenges for programmers in terms of writing parallel code. To address these challenges, various multi-threaded libraries have been developed to help programmers make the transition from sequential to parallel programming. These libraries provide optimized frameworks and tools for managing threads, and enable programmers to harness the full potential of multi-core processors [2].

Although these libraries have been extensively researched and applied for many years, choosing the appropriate library for different types of computational problems remains a challenge [3]. Different libraries have their own advantages and disadvantages in terms of performance, ease of use, and scalability, making it difficult for programmers to determine which library is best suited for their specific needs [4].

This paper aims to compare and analyze the performance gap of different multi-threaded libraries in the context of matrix operations [5]. By evaluating the performance and code characteristics of several popular multi-threaded libraries, this paper will provide programmers with a comparative analysis that can help them select the most suitable library for their specific computational needs [6].

2. Programming models evaluated

2.1. Basic information

2.1.1. C++ Standard Library thread. C++ Standard Library thread library is available in C++11 and higher versions. It provides classes and functions to create, manage and synchronize threads. It follows the Object-oriented approach, hence it provides many abstractions over the underlying low-level APIs. The library is cross-platform and provides a high-level abstraction for thread management. It also provides mutex, condition variables, and other synchronization tools for thread synchronization [7].

2.1.2. OpenMP. Open Multi-Processing is an API library that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN. OpenMP allows developers to write parallel programs that run on multiple processors in a shared memory system. It provides a set of compiler directives to express parallelism in the code. OpenMP is easy to learn and use, and it can automatically detect the number of available processors and distribute tasks across them [8]. It is suitable for parallelizing loops, sections, and tasks.

2.1.3. Pthreads. Pthreads is a POSIX standard library that allows developers to create and manipulate threads. It is a C library and is commonly used on UNIX or UNIX-like operating systems such as Linux, macOS. Pthreads provides low-level access to the operating system and hardware resources to create and manage threads [9]. It offers thread creation, synchronization, and communication primitives like mutex, semaphore, and conditional variable. It is an efficient and lightweight library but requires more coding effort to use.

2.2. Comparison

In terms of syntax, C++ Standard Library thread is easier to use and provides more high-level abstractions, while Pthreads provides low-level APIs to manage threads [10]. OpenMP provides a way to parallelize blocks of code by using compiler directives.

In terms of platform support, C++ Standard Library thread is available in modern C++ compilers like GCC, Clang, and Visual Studio. Pthreads is available for UNIX and UNIX-like operating systems like Linux and macOS. OpenMP is supported by many compilers on Linux, macOS, and Windows.

In terms of performance, Pthreads and OpenMP are more efficient than the C++ Standard Library thread. This is because Pthreads and OpenMP have less overhead and provide more control over thread management.

Overall, C++ Standard Library thread is the easiest to learn and use, whereas Pthreads provides more fine-grained control over threads, and OpenMP provides a way to parallelize large blocks of code with minimal changes. All three libraries are suited for different use cases and provide different features to manage multi-threaded programming.

3. Test method

In this paper, matrix multiplication is chosen to test and compare the performance of different libraries because it is a common usage scenario for parallel computing and its workload distribution is simple, which makes the comparison of the overhead of each library very clear.

At the beginning of each experiment, two matrices are generated randomly, the size of the matrices is specified as needed, and the elements of the matrices are random integers from 1 to 100.

```
Matrix generate_matrix(int rows, int cols) {
        Matrix m(rows, cols);
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> dis(1, 100);
        for(int i = 0; i < rows; ++i) {
            for(int j = 0; j < cols; ++j) {
                m(i,j) = dis(gen);
            }
        }
        return m;
    }
```

Use serial, C++ standard library thread, OpenMP, Pthreads to implement matrix multiplication, each abstracted as a function. In each function, use std::chrono::high_resolution_clock::now() to record the operation time.

3.1. C++ standard library thread

C++ Standard Library Thread provides a concise API that makes thread creation and task assignment very easy.

```
    void multiply_thread(const Matrix& A, const Matrix& B, Matrix& C) {
        std::vector<std::thread> threads;
        int num_threads = std::thread::hardware_concurrency();
        int chunk_size = A.rows / num_threads;

        auto start_time = std::chrono::high_resolution_clock::now();
        for(int t = 0; t < num_threads; ++t) {
            threads.emplace_back([&A, &B, &C](int begin, int end) {
                for(int i = begin; i < end; ++i) {
                    for(int j = 0; j < B.cols; ++j) {
                        int sum = 0;
                        for(int k = 0; k < A.cols; ++k) {
                            sum += A(i,k) * B(k,j);
                        }
                        C(i,j) = sum;
                    }
                }
            }, t * chunk_size, (t + 1) * chunk_size);
        }
        for(auto& t : threads) {
            t.join();
        }
        auto end_time = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::microseconds>(end_time -
start_time).count();
        std::cout << "multiply_thread time: " << time << " microseconds" << std::endl;
    }
```

3.2. OpenMP

OpenMP is easy to learn and use, and can parallelize serial code without many modifications.

```
    void multiply_thread(const Matrix& A, const Matrix& B, Matrix& C) {
        std::vector<std::thread> threads;
        int num_threads = std::thread::hardware_concurrency();
        int chunk_size = A.rows / num_threads;

        auto start_time = std::chrono::high_resolution_clock::now();

        //assign tasks
        for(int t = 0; t < num_threads; ++t) {
            threads.emplace_back([&A, &B, &C](int begin, int end) {
                for(int i = begin; i < end; ++i) {
                    for(int j = 0; j < B.cols; ++j) {
                        int sum = 0;
                        for(int k = 0; k < A.cols; ++k) {
                            sum += A(i,k) * B(k,j);
                        }
                        C(i,j) = sum;
                    }
                }
            }, t * chunk_size, (t + 1) * chunk_size);
        }

        //wait threads
        for(auto& t : threads) {
            t.join();
        }

        auto end_time = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::microseconds>(end_time -
start_time).count();
        std::cout << "multiply_thread time: " << time << " microseconds" << std::endl;
    }
```

3.3. Pthread

Pthreads provides low-level abstraction, is efficient and flexible, and can meet more complex requirements. However, it is difficult to learn and more difficult to use.

```
    void* multiply_thread_func(void* args) {
        auto& params = *(std::tuple<int, int, int, const Matrix*, const Matrix*, Matrix*>*)args;
        int begin = std::get<0>(params);
        int end = std::get<1>(params);
        int cols = std::get<2>(params);
        const Matrix& A = *std::get<3>(params);
        const Matrix& B = *std::get<4>(params);
        Matrix& C = *std::get<5>(params);

        for(int i = begin; i < end; ++i) {
            for(int j = 0; j < cols; ++j) {
                int sum = 0;
                for(int k = 0; k < A.cols; ++k) {
                    sum += A(i,k) * B(k,j);
                }
                C(i,j) = sum;
            }
        }

        pthread_exit(nullptr);
    }

    void multiply_pthread(const Matrix& A, const Matrix& B, Matrix& C) {
        std::vector<pthread_t> threads;
        int num_threads = sysconf(_SC_NPROCESSORS_ONLN);
        int chunk_size = A.rows / num_threads;

        auto start_time = std::chrono::high_resolution_clock::now();

        std::vector<std::tuple<int, int, int, const Matrix*, const Matrix*, Matrix*>>
params(num_threads);
        int begin = 0;
        for(int t = 0; t < num_threads; ++t) {
            int end = (t == num_threads - 1) ? A.rows : (begin + chunk_size);
            std::get<0>(params[t]) = begin;
            std::get<1>(params[t]) = end;
            std::get<2>(params[t]) = B.cols;
            std::get<3>(params[t]) = &A;
            std::get<4>(params[t]) = &B;
            std::get<5>(params[t]) = &C;
            pthread_t thread;
            pthread_create(&thread, nullptr, multiply_thread_func, &params[t]);
            threads.push_back(thread);
            begin = end;
        }
        for(auto& thread : threads) {
            pthread_join(thread, nullptr);
        }
        auto end_time = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::microseconds>(end_time -
start_time).count();
        std::cout << "multiply_pthread time: " << time << " microseconds" << std::endl;
    }
```

4. Performance comparison

For C++ standard library thread, Pthreads, OpenMP, different codes are used to implement matrix multiplication. This experiment tests the performance of these different parallel library implementations, as well as serial implementations, in order to compare and analyze each other. Testing was performed on Ubuntu 20.04 virtual machine based on VMware. The processor of the physical machine is 10th Generation Intel® Core™ i7, 8 cores, 16 threads, 2.3 GHz ~5.1 GHz. The virtual machine has 8 cores, 8 threads and frequency is 2.3 GHz. In parallel implementation of matrix multiplication, 8 threads are used.

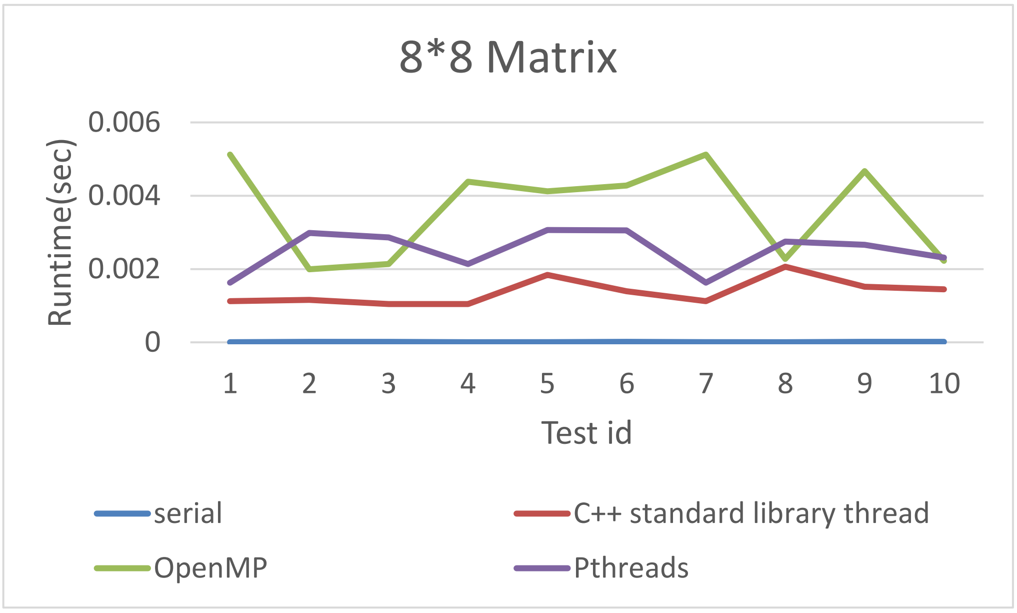

Figure 1. 8*8 Matrix.

For very small matrices(Fig1), the serial implementation can be completed in about 15 ms, while the parallel implementation brings no performance optimization and, on the contrary, consumes much more time than the serial implementation. Comparing between parallel implementations, it can be found that C++ standard library has the best performance and the most stable performance among the three libraries. OpenMP has the worst performance and is very unstable. Pthread's performance is in between the above two.

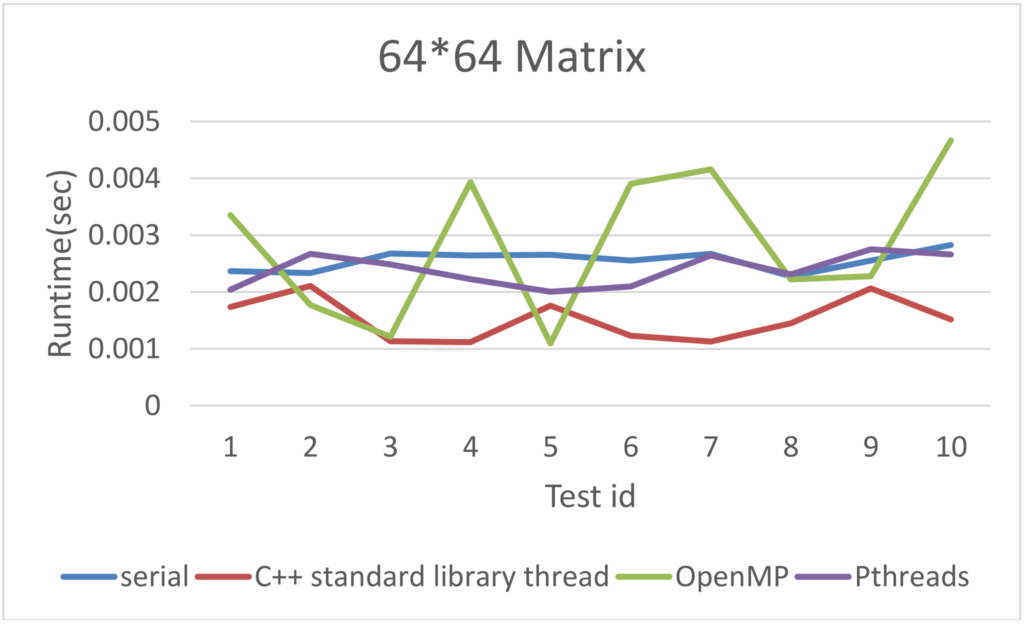

Figure 2. 64*64 Matrix.

For slightly larger matrices(Fig2), the serial implementation consumes similar time as the parallel implementation. And comparing the performance of the three parallel libraries reflects the same characteristics as when performing 8*8 size matrix multiplication

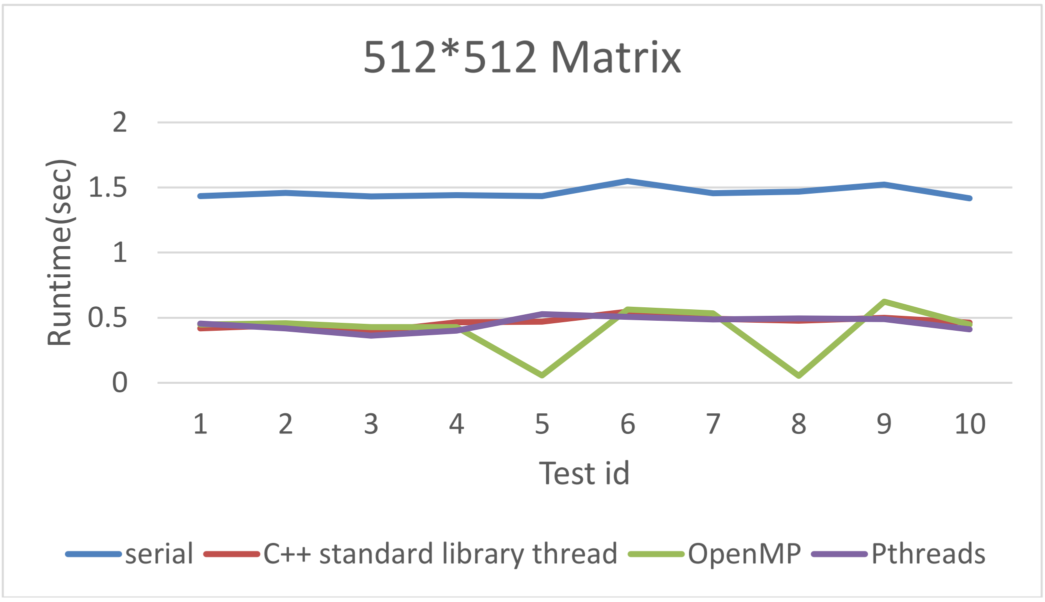

Figure 3. 512*512 Matrix.

For a matrix of size 512*512(Fig3), the parallel implementation shows a significant advantage over the serial implementation, which consumes about three times as much time as the parallel implementation. The three parallel libraries do not show a large gap between them, the OpenMP library is still more unstable than the other two, but its stability has improved compared to its performance when computing matrices of size 64*64.

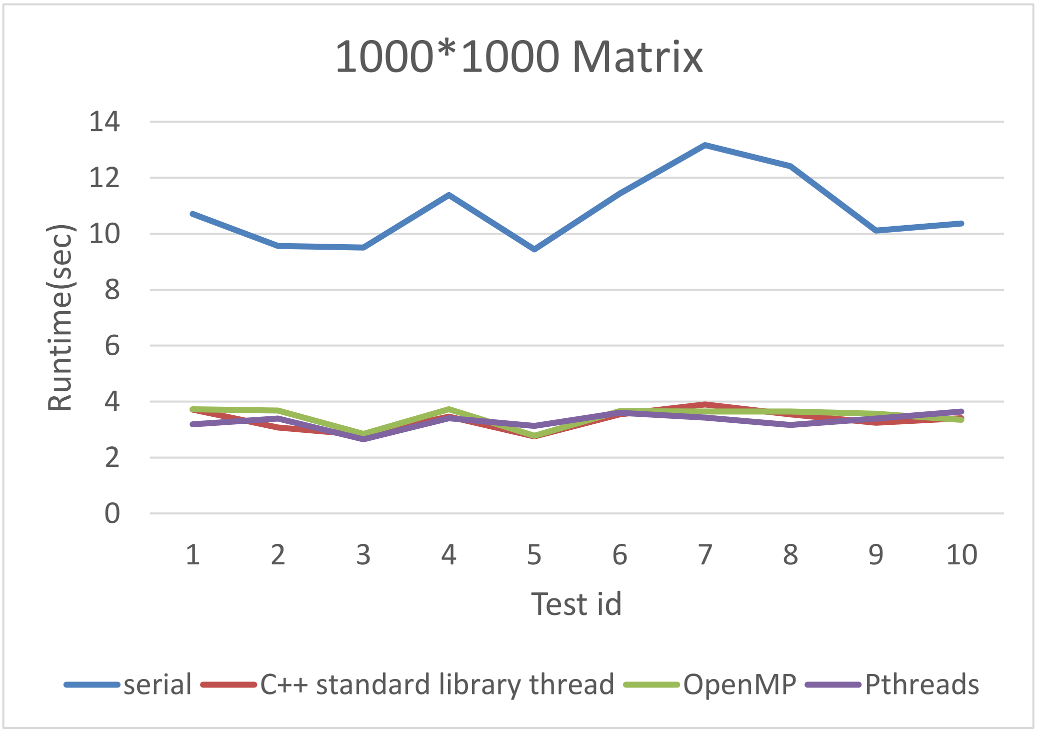

Figure 4. 1000*1000 Matrix.

For a matrix of size 1000*1000(Fig4), the parallel implementation is still three times faster than the serial implementation, with no significant expansion of the advantage.

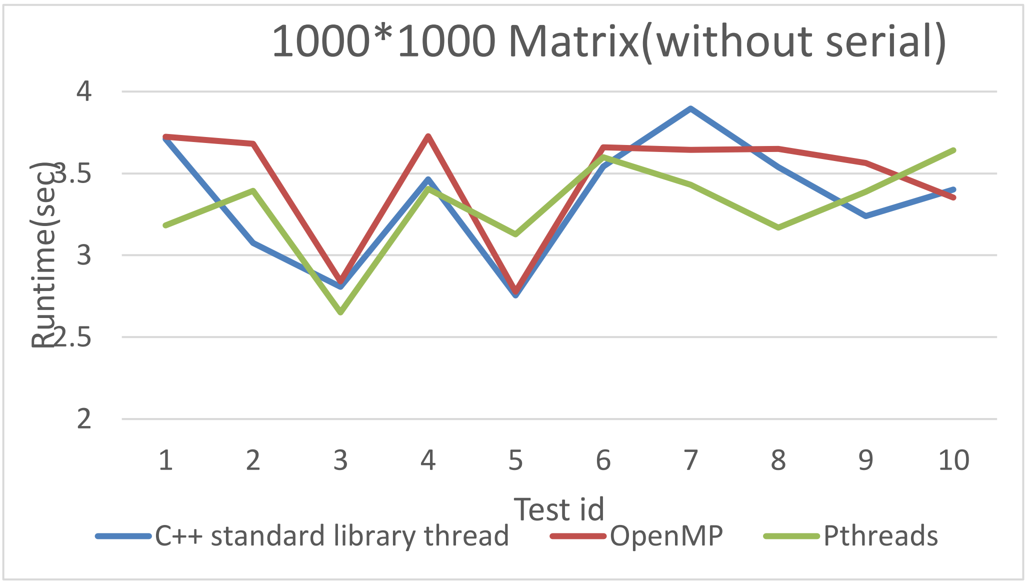

Figure 5. 1000*1000 Matrix(without serial).

Zooming in on the 1000*1000 matrix multiplication data, the figure shows that the performance gap between the three parallel libraries is still not reflected. In order to compare the runtime gap between the three libraries, this experiment also tested the matrix multiplication of 10000*10000 size.

Table 1. 10000*10000 Matrix.

10000*10000	C++ standard library thread	OpenMP	Pthreads
Runtime	5808.558858 s	4961.146563 s	4607.18698 s
	96.8 min	82.6 min	76.8 min

For the very large matrix(Table 6), the execution time becomes significantly longer, and the performance of the three libraries is sorted by C++ standard library thread < OpenMP < Pthreads. Combining the above experimental data, the following phenomena can be summarized. For small matrix(8*8,64*64) multiplication, parallelism does not improve the speed very well. If the matrix is too small, using multiple threads to perform the matrix multiplication will instead greatly increase the execution time Most of the time there is not much difference in the runtime of different libraries. When the matrix is small(8*8,64*64), the C++ standard library thread performs best, but for larger matrices(512*512,1000*1000), the runtime of the other two libraries becomes close to C++ standard library thread. For very large matrices (10000*10000), the performance ranking is Pthreads > OpenMP > C++ standard library thread. OpenMP is less stable compared to the other two libraries.

5. Analysis

The above experimental data reflect several phenomena, according to which the following analysis can be made: Parallel computation requires additional overhead to manage thread creation and destruction, as well as to coordinate communication and synchronization between threads. Although multi-threaded matrix multiplication does not require peer-to-peer and synchronization between threads, for smaller matrices, the overhead for thread creation and destruction can be more costly than the computation itself. On the other hand, for larger matrices, the time cost of the computation itself is much higher than the overhead of coordinating and managing threads, so parallel computation can more efficiently harness the power of multiple processor cores, resulting in faster computation.

For smaller matrices (8*8, 64*64) C++ standard library threads perform better, probably because its implementation uses lightweight threads that can be created and destroyed faster, avoiding excessive thread management overhead. For example, the join() method of std::thread uses a concurrent thread to implement the thread waiting feature. Before the join() method is called, the current thread suspend and waits for the target thread to finish. When the target thread finishes, the current thread resumes execution. This concurrent implementation avoids the overhead of using system calls and context switches, improving the performance and efficiency of the threads. The pthread and openmp implementations, on the other hand, require more management overhead, resulting in inferior performance to the C++ standard library threads.

And for very large matrices, pthread performs best, probably because the way pthread implements threads is more underlying and closer to the kernel, it has finer control over multi-thread management, communication and synchronization operations, while getting better support in terms of hardware and OS optimizations. For example, pthread creates threads by calling the system call pthread_create(), which interacts directly with the OS kernel to allocate and manage thread resources, and is therefore closer to the bottom level; in addition, pthread provides very fine-grained thread scheduling and control, and can control the priority of foreground and background threads, cleanup, cancellation and resource recycling through the API. It supports real-time thread scheduling mechanism, which can be used in application scenarios that require real-time response.

Compared with the other two libraries, OpenMP is a high-level parallel programming model designed to simplify the complexity of parallel programming. Although OpenMP provides some instruction sets for controlling threads, such as omp_set_num_threads() and omp_get_num_threads(), these instructions cannot finely control the threads. OpenMP encapsulates the creation, synchronization, and memory management of threads operations in the underlying library, making it impossible for programmers to have low-level control over threads, such as thread scheduling, inter-thread synchronization, and thread private variables. Since the implementation of matrix multiplication with OpenMP does not allocate tasks finely, when calculating smaller matrices, the thread load is not evenly distributed; some threads have more workload, while others are idle. This results in unstable performance of OpenMP.

6. Conclusion

This study tested the performance of three commonly used C++ parallel programming models for simple parallel operations, and analyzed their respective features. The experiment selected the common application scenario of parallel computing, which is matrix multiplication as the test program. The experimental data showed that data size has a significant impact on the performance of different models. First, when the processed matrix is very small, the performance of parallel implementation is much lower than that of serial implementation. Next, when the matrices are relatively small, C++ standard library threads have a better performance than Pthreads and OpenMP due to their lightweight thread performance. As the matrix size expands, the performance ranking of C++ standard library threads becomes the lowest due to its large overhead, while Pthreads performs the best. In addition, the performance of OpenMP is unstable due to uneven workload distribution for smaller matrices, but improves and tends to be stable as the matrix size increases.

References

[1] Akhter S, Roberts J. Multi-core programming[M]. Hillsboro, Oregon: Intel press, 2006.
[2] Ravi V T, Ma W, Chiu D, et al. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations[C]//Proceedings of the 24th ACM international conference on supercomputing. 2010: 137-146.
[3] Treangen T J, Salzberg S L. Repetitive DNA and next-generation sequencing: computational challenges and solutions[J]. Nature Reviews Genetics, 2012, 13(1): 36-46.
[4] Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming[J]. IEEE computational science and engineering, 1998, 5(1): 46-55.

[5] Akimova E N, Gareev R A. Algorithm of automatic parallelization of generalized matrix multiplication[C]//CEUR Workshop Proceedings. CEUR-WS, 2017, 2005: 1-10.
[6] Laaziri M, Benmoussa K, Khoulji S, et al. A comparative study of laravel and symfony PHP frameworks[J]. International Journal of Electrical and Computer Engineering, 2019, 9(1): 704.
[7] Barney B. POSIX threads programming[J]. National Laboratory. Disponível em:< https://computing. llnl. gov/tutorials/pthreads/> Acesso em, 2009, 5: 46.
[8] Chu W W, Holloway L J, Lan M T, et al. Task Allocation in Distributed Data Processing[J]. Computer, 1980, 13(11): 57-69.
[9] Wang W, Dey T, Mars J, et al. Performance analysis of thread mappings with a holistic view of the hardware resources[C]//2012 IEEE International Symposium on Performance Analysis of Systems & Software. IEEE, 2012: 156-167.
[10] Ismail A, Shannon L. FUSE: Front-end user framework for O/S abstraction of hardware accelerators[C]//2011 IEEE 19th annual international symposium on field-programmable custom computing machines. IEEE, 2011: 170-177.