Homework4 report

Kangdong Yuan

Introduction

In this part, the goal of this homework is to use openMP in C++. The original program of two parts has a lot of math calculations. Both parts of the homework involve a lot of mathematical operations, including addition, multiplication and comparison. These multiplications and additions are not related to each other before and after, which allows me to make this series of parallel operations to decrease the running speed of the program. The system I run those programs is red-hat 7, and CPU is Intel E5 2680 with 20 cores, and the random-access memory has 256G size.

Part1

1.1 Solution

To improve the performance, we need to use the feature of openmp. First, there are two loops calculating the mean and the standard deviation of the array, we can use *"#pragma omp parallel for"* to parallel them to decrease the computing time. But there is a need for us to do reduction for each loop, because the parallel computing of summation could cause data race. Then there is a loop to find the max and min value of the array, and each iteration contains two comparisons. I also use *"#pragma omp parallel for"* to improve this loop, but I also use two reduction clauses *"reduction(max:mymax) reduction(min:mymin)"* to prevent data racing. Moreover, I assign the array A as shared because it has been defined in outer scope.

For the function to calculate the threshold result, I use the same method to improve the performance of loops. But I use *#pragma omp critical* in the second loop, because this part of code must be executed by a single thread at a time.

1.2 Running time analysis

My estimate of the running time is that as the number of threads increases, the running time will decrease proportionally. And as the amount of calculation decreases exponentially, the running time will decrease exponentially.

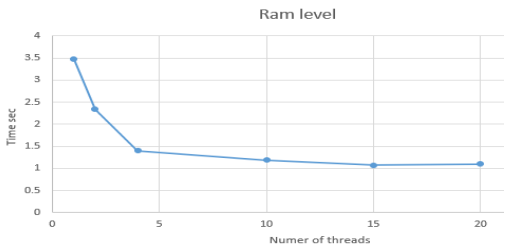

*Figure 1 N=1*10^9*

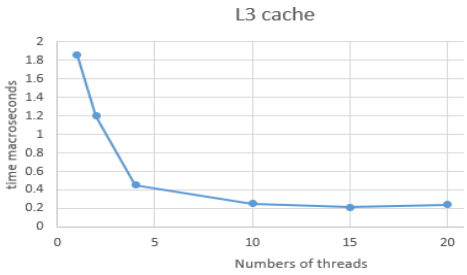

*Figure 2 N=1*10^5*

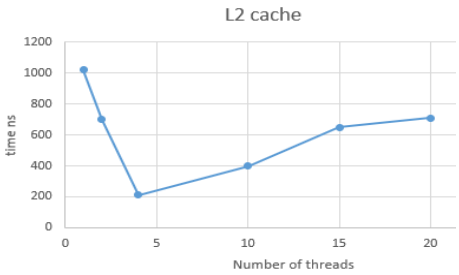

*Figure 3 N=1*10^3*

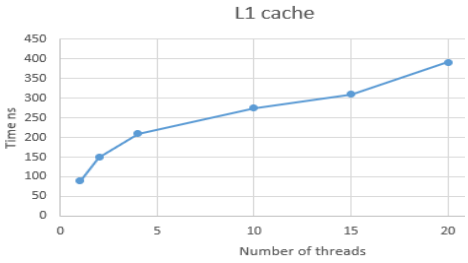

Figure 4 N=100

I have tested the running time of this program in different memory levels and threads numbers. For large arrays (fit in ram and L3 cache size), the running time decreases as the number of threads increases, although the improvement rate decreases. But for small size arrays that fit in L2 and L2 cache, the running time increases as the number of threads increases, which is a wired result that is inconsistent to our expectation. I search the textbook to find the solution, and I find that the start and collect of each thread need extra times that often cost almost 100 clocks. So, as the number of threads increases, the running time for start and collect threads. For very small size arrays (fit in L1, L2), the proportion of computing is small, the majority of time consumed on threads process.

But, when it compares to the std::thread method, I find the openMP has better efficiency. Especially in small array size, the cost of starting and collecting threads of openmp is much faster than std::thread package.

1.3 Conclusion

The openMP is a good library to do parallel computing, but we need to take care to avoid data racing problems and arrange threads in suitable order by using critical, master or other methods. And the number of threads should fit into the current situation, because the cost to start and collect thread are expensive.

Part 2

2.1 solution

This part is work to improve the performance of matrix multiplication, because its running time is very long in serial computation (usually more than one minute). The key part of improving efficiency is data access optimization and parallel computing.

I implement two techniques first is the using the transpose matrix to decrease compute time. I define a transpose function and run it before matrix multiplication. In the transpose function, I use openmp clause *"#pragma omp parallel for"* to parallel the function.

Moreover, in the main function, there are three loops which are very high load for the processor. So, I use the *"#pragma omp parallel for"* to parallel the computations in main function.

2.2 Running time analysis

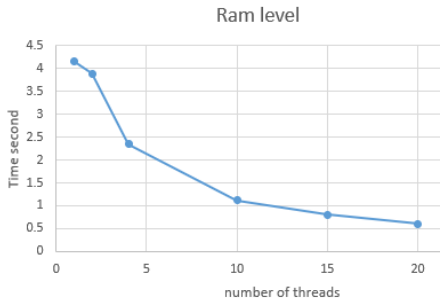

Figure 1 N=2000

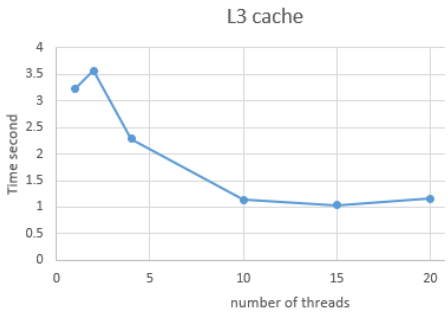

Figure 1 N=500

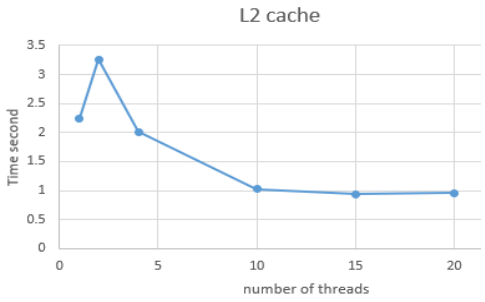

Figure 1 N=50

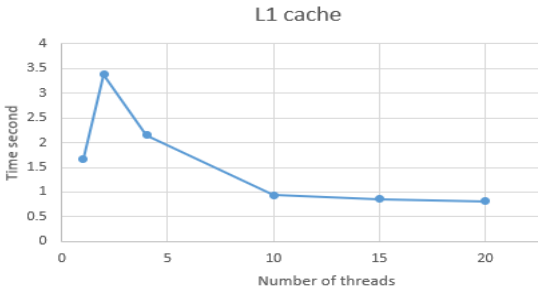

Figure 1 N=10

My estimate of the running time is that as the number of threads increases, the running time will decrease proportionally. And as the dimensions of the matrix decrease, the running time will decrease exponentially.

The data table above gives me a clear regular pattern. The running times for the same memory level (different threads numbers) are different, and the running time decreases as the number of threads increases. But the running time for the same thread number (different memory level) is almost equal. This running time seems very weird and consistent to our expectation, because the number of computations decrease rapidly as memory level changes. I searched for a book, and I assumed this running time pattern because of two reasons. The first reason is the data access speed, this program reads and writes many times. If we use multi threads, it enables us to use multi cache in different cores. So, the multi-threading enables the program to run faster. Second, the start and collect of the threads need to take many times, so the major proportion of running time is multithreading setup, and the computation is small proportion. So, the running time for different memory levels (same threads number) are similar.

When comparing the running time to std::thread, the running time of the openmp version is much faster because I use the matrix transpose in this function to accelerate the computation. Moreover, the openmp is more efficient than std::thread in multithreading.

2.3 conclusion

In matrix multiplication program data access speed is an important factor that affects running time. And the number of threads should be assigned to a suitable value, because the cost of each thread is expensive. Moreover, in small dimension matrix multiplication, serial computation is better than parallel computation.