HW 3 report

Kangdong Yuan

Introduction

In this part, the goal of this homework is to use std::thread library 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 and multiplication. These multiplications and additions are not related to each other before and after, which allows me to make this series of parallel operations to increase 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

Problem statement

This program has two functions, the first one is calculating standard-deviation, mean, min, max of an array. The second one is calculating the number and index of elements that meet specific requirements in the array. But the calculations of two functions are all disjoint, which means the calculation can be divided into multi pieces and done separately, this is a sign to do parallel computing in this program. Moreover, the calculation of finding max and min can also be paralleled by finding the max and min of the sub-array, then we find the max and min of the whole array by just comparing the returning result of each sub-array.

Solutions

The solution to part one is very straightforward: dividing the for loop into many small parts, and each part starts a thread to calculate the result. The length of the array is N, and the number of threads we need to create is P. So, it means we need to divide the for loop into P parts and start P threads to compute those sub-result. Then use a for loop to join all the threads, which tell the program to wait after the execution of all threads. After all threads are completed, we combine all the returning values together to the final result. Moreover, I define a new struct called MYPARAM to store all the values and parameters in different threads.

Analysis

When I run the default case in a test program, the paralleled version of the program shows significantly improvement in time. For standard-deviation function, it improves from 4 seconds to 0.57 second. For Threshold function, it improves from 2.5 seconds to 0.468 second. However, some statistical results of standard-deviation function do not pass, although it returns the same value. And, I try to reduce N to 10000, 1000, and 100, which is the size fit to L1, L2, L3 Cache. I find that the standard-deviation function can pass sometimes, but sometimes standard-deviation function cannot be passed. I searched google for the wired error, it said that it is a float point error that is hard to avoid. And there is a new finding that the multi-threading is not faster than the serial method for N=1000 or 100. I think the reason for this phenomenon is that the start and join of thread take longer time than serial computation. So, if the cost of thread setup is more than serial computation, the multithreading will impair the efficiency.

Conclusion

Multithreading is useful to help save time on repetitive computation in a program. But the time cost of start and join thread is also comparable, so the multi-threading should only be used in case that there is too much load for serial computation. Moreover, float point error

can happen when we use different orders to compute, so we need to compute many times to ensure accuracy.

Part2

Problem statement

The program in part2 is the multiplication of the matrixes. But the dimensions of the matrices are very large, and there are 3 nested loops in this program, so the time complexity of this program is N^3. Moreover, because the huge dimensions of matrices, the data access and write speed are very slow. I have done the data access optimization by performing block-size computation, and the running time of this program improves from 76 seconds to 7.5 seconds. But there is still space for parallel computation to improve the running time.

Solution

For part2, I write two versions of the paralleled program. The first version is applying parallel computation directly. I define the number of threads to 40, and divide the dimension of matrices N into P parts. I design a new struct called MYPARAM to store the index of sub-matrices that each thread needs to solve. Then, I use the for loop to start each thread, and I use a for loop to join each thread, finally I use a for loop to aggregate the final value.

The second version of the program is a paralleled version of block-size matrices multiplication. On the base of block-size matrices multiplication, I apply parallel computation in each block. The number of threads in each block is P, and total threads of program is (P*block-size)/N.

Analysis

The running time of each version of program

Original unmodified version: 76.5 seconds

Data Optimized version: 7.5 seconds

Original unmodified threaded version: 2.4 seconds

Data Optimized threaded version: 108.2 seconds

To my surprise, when I just apply paralleled computation, it is faster than parallel + data access optimization. And I think the reason for this case is because parallel + data access optimization versions start and join too many threads, so the time of start and join threads impair the efficiency of the program. So, if we just apply parallel computation, it can achieve faster running time.

Moreover, I test different N from 100 to 300, 1000, 2000 which fit L1, L2, L3 Cache and RAM size. The running time improvement from threaded increases rapidly as the dimension of the matrix increases, because, for all matrix size, the running times are around 2 seconds. And, for N =100 or 300, the threaded version is slower than the original version. Thus, the threaded method is not always good, it should depend on the problem size and thread size.

Conclusion

The number of threads should be assigned to a reasonable number, because the start and join of the thread take a lot time. And we need to calculate the optimal number of threads to ensure the fastest running time. If the total size of the problem is small, serial computation is better than parallel computation.