PC-2018/19 Course Project Template

Davide Nesi
davide.nesi@stud.unifi.it

Abstract

In this paper we will compare the performance of OpenMP and serial approach. For this comparison we decided to use a common problem in data analytics: given a timeseries and a pattern retrive the point where the distance between the timeseries values and the pattern values is the minimum. In this paper we will show and discuss our results for variable datasets length.

1. Introduction

Time series is a series of data indexed by time. Probably it is the most common type of representation for real world sensors and measures. The necessity to find and retrieve where the time series is more similar to a selected pattern is a way to find meaning and forecast possible scenarios. This problem is common in finance and usually it involves quite large datasets and multiple runs for different patterns. For the comparison we intended to bring to attention we decided to use different implementation of a serial approach to the problem in C++ and a parallel version for each approach. For the parallel approach we decided to use OpenMP. Doing so we were able to easily confron naf solution to more ingenious ones.

1.1. Pattern in time series

We decided to use this problem for our benchmarks for its properties. Superficially analyzing the problem we can deduce that given p the length of the pattern P and t the lenght of the time series T the problem can be written as in (1).

$$\min_{i \in \phi} \sum_{n=i}^{i+p} |T(n) - P(n-i)| \qquad (1)$$
$$\phi = [0, n-p]$$

We can see how the number of operations required to solve the problem is defined by (2)

$$(t-p) * p \simeq O(p^2) \qquad (2)$$

so this problem is definetly intresting. Given (1) we can notice how the iterations of the problems are completly independent on each other. This property is very important for simple parallelization. We can label this problem as an embarrassingly parallel problem. The search of minimum is not a complex part and we omitted in the complexity estimation.

1.2. C++ and OpenMP

We decided to use C++ as base programming language. This decision is based upon common diffusion of this language and the OpenMP support. OpenMP is an API that supports multiple programming languages, and in particular we can use C++ support to show differences between parallel and serial approach without changing context. OpenMP allows us to test different C++ implementation and to test how good and bad coding behaviour can really affect performance. For this reason we provided different implementations for both serial and parallel approach.

2. Tools

For this paper we used both programming and data analysis tools: the codes written in C++ and linked to OpenMP is written inside CLion IDE, the data obtained from the multiple runs is then analyzed and displayed using MATLAB. All the test are performed on a computer equipped with an Intel i7-8700k. The CPU used has 6 cored and 12 threads. The external workload on the machine while performing the tests is zero.

2.1. CLion

CLion is a great IDE for C++ and for many other languages. In order to obtain full use of OpenMP we needed to modify the CMake file to include OpenMP. This was a simple and painless solution. Compilation runs smoothly and without import errors.

2.2. MATLAB

MATLAB is probably the most used numerical computing enviroment and offers a lot of tools for data analysis and representation. We used MATLAB to estimate the attendability of our datas and to display the results from multiple runs.

3. Implementation

We decided to offer more than one implementation for both serial and parallel approach to better represent how small changes with both OpenMP and C++ only can have a big impact on performance. To be sure about result correctness we decided to check results for every execution. In the next subsection we will discuss all theimplementation we propose.

3.1. Serial C++ implementations

For our first implementation we just coded the problem as presented, creating a vector for all the distances and updating it step by step during the execution. Than we seached for the minimum inside the vector created. The second implementation is very similar to the first one but we decided to invert the order of the loops to create the vector of differences. These first two approaches are probably the most intuitive ones. In the third implementation we decided to use a variable to store the diffrerences and then copy the variable value inside the vector. This approach shows us how a small change can affect performances. The last serial approach we used completly avoids the vector creation process. In fact during the creation of the distances only the minimum is saved with the corresponding index.

Implementation	Execution time
0 simple	112929.278 ms
1 inverted simple	112977.147 ms
2 tmp var	91323.974 ms
3 no resVector	91290.520 ms

Table 1. Serial implementations example runs with pattern lenght of 10^3 and time series length of 10^7.

3.2. Parallel OpenMP implementations

The first two parallel implementations are the simpler ones and are very similar to the first two serial implementations. The third implementation uses a temporary value without storing it in a vector. The fourth implementation specifies to OpenMP the number of threads to use and uses the OpenMP barrier. The fifth one is the implementation uses the best practice for this case having the barrier and the longest iteration as the first loop. The sixth implementation is an OpenMP reduction test.

4. Results

The results of this tests were very clear. In Table (1) we can see how among the serial implementations the use of the temporary value increased the performance and lowered the execution time.

Among the parallel implementations proposed the best one is the implementation with barrier and temporary value. Reduction has actually a bad impact on the performance as seen in Table (2).

Coparing parallel and serial approach in solving the given task we can see how even the most basic parallel implementation has a significative

Implementation	Execution time
0 simple	17663.895 ms
1 inverted simple	16715.396 ms
2 tmp var	13000.138 ms
3 explicit threads	13181.506 ms
4 tmp var	12988.320 ms
5 reduction	22719.429 ms

Table 2. Parallel implementations example runs with pattern lenght of 10^3 and time series length of 10^7.

boost in performance compared with a serial one. Even when the dataset is very small the parallel implementation is practically always better than the serial one. In Figure (1) we can se how varying the size of the pattern and the size of the time series affects the time of execution. In Figure(1) the upper surface is generated by the time of execution of a serial implementation and the surface on the bottom is generated by the time of execution of a parallel implementation. The whole top surface took 1000 runs to be completed and around 8.9 hours. The lower surface took 1000 runs to be completed and around 0.9 hours. That means that globally we achieved a speed up of around 9.9 times.

5. Conclusions

In the end we can conclude that the parallel solutions we proposed have much better performance compared to serial ones. And even with very few data there is no real reason to go with a serial implementation.

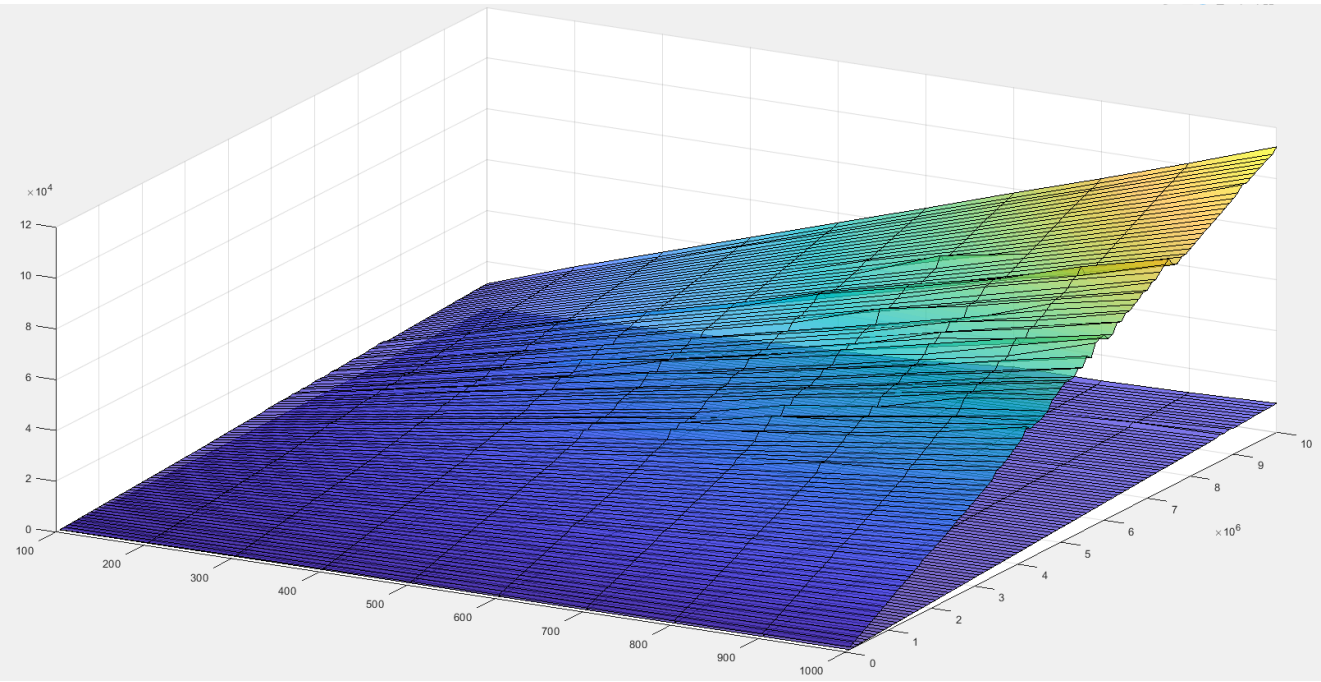

Figure 1. The two surface created by time of executions of one of the serial implementations (on top) and one of the parallel implementations (on bottom).