PC-2024/25 Midterm Assignment

D'Alò Gabriele
gabriele.dalo@edu.unifi.it

Abstract

This assignment is about pattern recognition in a time series of data. The main goal is to evaluate the difference in performances between sequential and parallel implementations.

1. Introduction

1.1. Assignment

In this assignment we want to carry out pattern recognition on a time series of data, using the SAD metric as the matching criterion. It is required to implement the algorithm sequentially and in parallel in order to evaluate the difference in the performances of the two implementations.
It is also required to experiment with various cases, such as searching for time series of different lengths or with the use of different techniques for parallelization.

1.2. Objectives

The objective of this experiment is to evaluate how the performances of a simple algorithm vary when parallelism is used and to see what are the different ways in which the latter can be implemented.

1.3. Development environment

All experiments were carried out on a machine with a macOS Sonoma operating system with a 2,6 GHz Intel Core i7 6 core processor and a 16 GB 2400 MHz DDR4 memory. The code was mainly written using C++ as the programming language and using the OpenMP library to implement parallelization, CLion was used as IDE and clang as compiler.

2. Implementation

2.1. Structure

Two different projects have been created to experiment with, one that uses AoS (Array of Structures) and one that uses SoA (Structure of Arrays), in order to see how the performances vary using different techniques for arranging the data in memory.
In addition to these two projects that performs the experiments there are also further scripts:

- TimeSeriesGenerator.py: Python script that generates the time series of data on which we will perform pattern recognition;
- plot_results.py: a simple Python script that allows us to plot our results generated at the end of the experiments;

2.2. General overview

Let us now describe the various written codes in more details.
As mentioned, the TimeSeriesGenerator.py script allows you to generate a time series of data, it does so using mockseries (`https://github.com/cyrilou242/mockseriesf`). The data generated are not realistic, but represent the increase in the average annual temperature recorded over the years, taking a period of 2024 years as a reference and acquiring the temperature value daily. Fixed values for the temperature increase were set and a linear trend was assumed (which is not realistic). For purely experimental purposes we are not interested in the data being realistic, so it was decided to create a specific script and not to use ready-made datasets so as to be able to create new

data if necessary, *e.g.* increasing the span of years. The other python script is a simple piece of code that takes a .csv file, in which the results of our experiments are saved, and plots the data to show how the speedup and efficiency of our code varies as the number of threads used for parallelization increases. Remember that:

$$Speedup = \frac{Execution\ time\ sequential}{Execution\ time\ parallel}$$

$$Efficiency = \frac{Speedup}{Number\ of\ threads}$$

As already mentioned, there are two versions of the project, Midterm-Assignment_AoS and Midterm-Assignment_SoA, which codes are essentially the same except for the way time series are stored in memory, which is only relevant for the analysis of the final results, so they will be treated as a single project in the discussion that follows.
Furthermore, in order to make the experiments repeatable, it was decided to take as the pattern to search for a smaller time series acquired from the initial time series, which is perturbed by a certain amount. In particular, the last N values of the time series are taken, with N varying depending on the experiments.

2.3. Main project

Let's now move on to the main part, that is, the one that deals with carrying out the sperimentation. The project is designed to be able to be executed in 2 modes, depending on the input entered by the user. In addition to the mode that carries out experiments automatically, there is also an option to allow a user to perform customized experiments. In the case of customized experiments, the user can specify the number of threads to use for parallelization, the length of the time series (the pattern) to search for and whether to carry out the experiment several times (with the same data) in order to calculate an average on the results.
In the case of automatic experimentation, a number of repetitions to be carried out has been set (100), the parallelization is carried out with different numbers of threads (1, 2, 4, 6, 8, 10, 12) and with different lengths for the pattern to look for (1, 10, 50, 100, 500, 1000). For each possible combination, speedup and efficiency results are calculated and saved. The program execution flow is as follows:

1. The time series on which the pattern recognition will be performed is acquired from the CSV file (generated using the appropriate script);
2. For each fixed length of the pattern and for each number of threads the experiment is repeated several times and an average of the execution times is taken. The pattern is first acquired from the end of the time series and perturbed, after which the pattern recognition is carried out in 4 versions: sequential and with 3 different parallel implementations (see 2.4);
3. For each time series to be searched, we keep track of the version that results in the best speedup and the best efficiency as the number of threads varies, and the best results are saved in a CSV file (see 3).

2.4. Code versions

The sequential code that performs pattern recognition works as follows:

- We start from the beginning of the time series and consider a window of the length of the pattern to be searched for;
- The SAD (Sum of Absolute Differences) between the window in the time series and the pattern to search for is calculated. We check whether this SAD is better (less) than the best SAD found up to that moment and if it is, that is considered the new best SAD. We also keep track of the values in the time series that correspond to the best match;
- The window is scrolled by 1 and the new SAD is calculated and so on until there are no more windows to check;

We also keep track of the number of comparisons made to ensure that it is the same for everyone, this would not be strictly necessary but is also done for educational purposes.
The parallel code was implemented in three different ways so we can see which one performs best on our problem. The first version it's one of the simplest version of parallelization we can think of, the default

OpenMP implementation for splitting data to threads is used. Each thread performs local pattern recognition on its own data, locally keeping track of the best SAD, the values that allow us to obtain it and the number of checks performed, after which a critical section (to avoid race conditions) verify which is the best SAD and add the number of local checks to get the total.
The second version, instead, explicitly assigns data chunks to the various threads, this could help reduce scheduling overhead and achieve greater efficiency in memory management.
However, we would like to avoid using critical sections whenever possible, as these introduce synchronization delays, the calculation of the total number of comparisons can be carried out with atomic, avoiding having to perform it in the critical section, but the same thing cannot be done for the calculation of the minimum SAD (OpenMP doesn't have a method to do this). In the third version, atomic is used to calculate the total number of checks and a double check of the minimum SAD is used to reduce the number of times the critical section is accessed. In particular, what happens is that each thread checks whether its local SAD is less than the global one, if it isn't then it is not necessary to enter the critical section since even if the global one were updated it would still be less than the local one, if instead is less than it enters the critical section and the check is performed again as someone else may have updated the local value before the actual entry into the critical section.

2.5. Implementation choices

Looking at the problem text, three things come to mind that it might make sense to parallelize:

- **Acquiring the time series from the CSV**: it makes sense since our time series is quite large (almost 1 million data), however the data can only be read from a CSV sequentially, therefore it is necessary first to store the entire content of the file into a string from which we can then separate and read the data in parallel;
- **Generation of the pattern to recognize**: even if we wanted to generate it randomly instead of acquiring it from the time series, it would not make much sense to do it in parallel as the patterns we are considering have a fairly small size (the maximum is 1000 elements) and therefore there would be a worsening of the performances due to the parallelization overhead and inefficiency;
- **Pattern recognition**: it makes sense to do it in parallel since, as mentioned, we have a time series containing a lot of data and therefore we have many operations to perform.

Checks have also been included in the code to verify that the various versions of pattern recognition return the same results. Furthermore, one thing that has been noticed is that there are cases in which the parallel implementation requires so little time to perform some operations that this time is considered 0, therefore, to avoid problems (such as division by 0) it was decided to consider this time as an arbitrarily low value i.e. 10^{-8}. Looking at the code you can also notice that the sequential time is recalculated every time a different number of threads is used, this is not calculated only once at the beginning because it has been noticed that otherwise you have inconsistent results, due to the fact that the performances seems to get worse over time as the number of executions increases (this may be due to things like my PC temperature rising, for example), in particular as the number of threads increases.

3. Resutls

Let's finally discuss the results of our experiments. Looking at all the results you can see that the implementation with SoA is slightly better in terms of execution time than the one with AoS, this mainly depends on the fact that the pattern recognition is carried out only on the values of the time series (not on the dates) therefore the fact that they are contiguous in memory in the case of the SoA allows us to access them more quickly (the spatial locality is exploited). Also we can then notice that there is not one version that always seems to be better than the others, it depends on the length of the pattern to search but we observe that the one with the explicit declaration of the chucks seems to be the best more often in the case of using AoS while in the case of SoA it's the one in which atomic is also used.

3.1. Performances evaluation

The first thing we can notice is that the time for reading data from the CSV decreases when parallelization is applied, and reaches its minimum when 6 threads are used, this is probably because reading data is a very light operation and therefore the data is still too little to be read efficiently with more threads (*e.g.* with 12 threads).
To calculate the best number of threads to use we cannot only take into account the speedup because we could have cases in which we have a high speedup but a very low efficiency, what is done is that a score is calculated for each number of threads given by product of speedup and efficiency and the use of the number of threads that has the highest score is considered better. From this we see how the best number of threads to use is always 12 even if it has less efficiency when you have a small size for the pattern to search for. Clearly this can vary depending on what we are interested in, *e.g.* if we are more interested in efficiency at the expense of some speedup it might be better to use fewer threads. Another thing we can observe is that the best speedup occurs in the case of patterns with length 100.

3.2. Graphs

To conclude, let's see some of the graphs generated from our results (not all are reported but only some relevant ones).
Let's start by observing the results in the case of a small dimensional pattern with 10 elements. From the graphs below we can see that the speedup increases almost linearly as the number of threads increases while the efficiency decreases and then increases again, that's because the worst number of threads to use seems to be 8, furthermore, in this case slightly better speedup and efficiency results can be observed in the case of AoS. The fact that with 8 threads there is a worsening of the expected performance could be due to various factors, such as the affinity of the threads to the processor. With 8 threads, you could have a distribution in which multiple threads compete for the resources of a single physical core, with 10 or 12 threads instead, the system could redistribute the threads more evenly, improving performances.

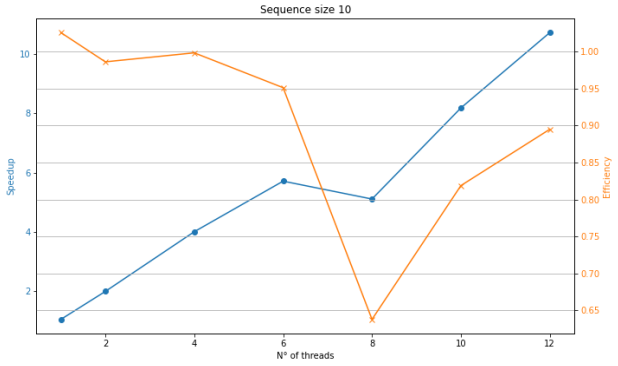

Figure 1. Pattern size = 10 with AoS

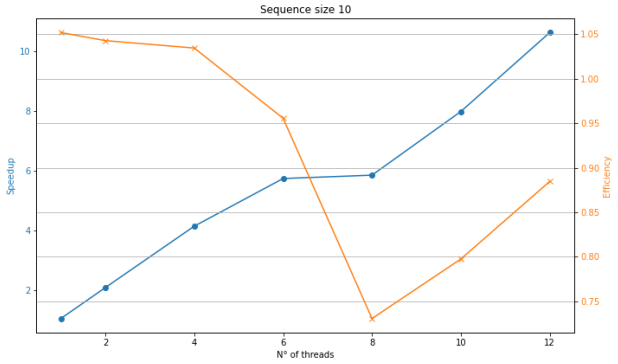

Figure 2. Pattern size = 10 with SoA

Instead, taking a pattern of slightly larger size (100) we observe that, as already mentioned, we get a better speedup as we get to have it linear or even superlinear in the case of AoS. In fact, the increase in speedup is almost linear as the number of threads increases, however, from an efficiency point of view there is still a drastic drop when 8 threads are used.

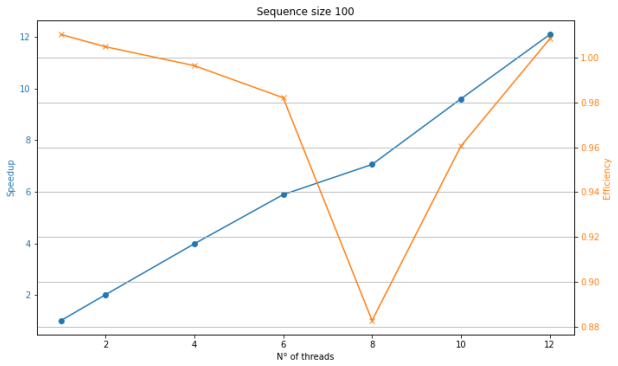

Figure 3. Pattern size = 100 with AoS

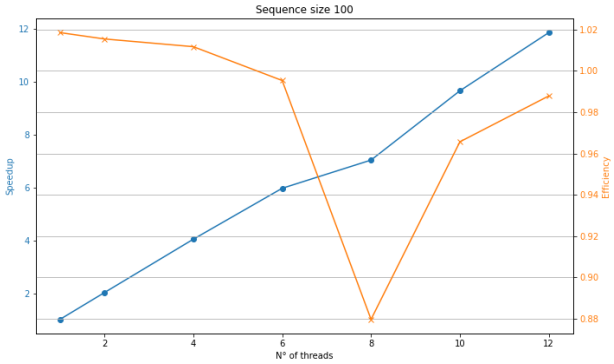

Figure 4. Pattern size = 100 with SoA

Finally, testing with a larger size such as 1000, we observe that the increase in speedup is now practically linear, while the efficiency decreases slightly between 1 and 6 threads and then drops significantly with 8 and decreases again for 12 threads (observing the measurement scale, however, we can see that the efficiency in general is better than the previous case).

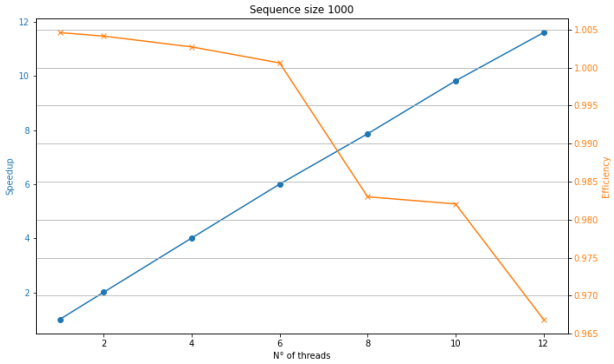

Figure 5. Pattern size = 1000 with AoS

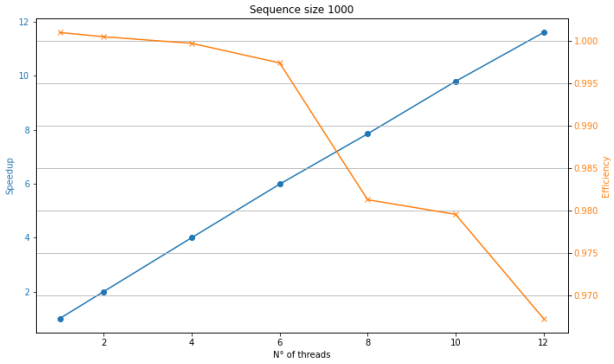

Figure 6. Pattern size = 1000 with SoA

This particular behavior when using 8 or 10 threads could be due to the fact that the machine on which the experiment was performed is a machine with 6 physical cores that uses hyper-threading and therefore has 12 logical cores. So, up to 6 threads each thread works on a dedicated physical core, making the most of available hardware resources. When you use 8 or 10 threads, some physical cores start running two logical threads at the same time, this introduces overhead due to sharing physical core resources such as: L1/L2 caches, ALU, memory bandwidth, ecc..., as a result the additional benefit of parallelization is reduced. When you go to 12 threads, all physical cores run two logical threads, taking full advantage of hyper-threading.
To conclude, from the results we can observe how this problem is particularly well suited to being parallelized, and, if we have enough data to work with, we can exploit all the cores at our disposal very well.