OpenMP Pattern Recognition

Alessandro Bianco
alessandro.bianco1@edu.unifi.it

Abstract

This report outlines the implementation details of a pattern recognition algorithm applied to time series data. The algorithm is based on the SAD (Sum of Absolute Differences) metric and aims to identify the time series in the dataset that most closely matches a given query. The dataset is generated using a Python script that leverages the mockseries library. Both the SOA (Structure of Arrays) and AOS (Array of Structures) representations will be implemented in sequential and parallel versions, along with an analysis of how parallelization impacts different sections of the algorithm.

1. Introduction

The goal of this project is to analyze the performance of parallelization applied to a search problem over datasets of varying sizes, where increasing the data volume also increases computational complexity and resource requirements. Specifically, the aim is to evaluate the efficiency of the algorithm's sequential versions and to examine how different data layouts, SOA and AOS, affect the results. In the parallel implementation, two distinct approaches will be compared: one in which the outer loop that iterates over the time series is parallelized, and another where only the inner loop, responsible for comparing the elements of a single series with the query, is parallelized. This comparison will help assess how the choice of parallelization strategy impacts the overall performance of the algorithm.

2. Pattern recognition problem

The problem of pattern recognition applied to time series consists in identifying, within a set of temporal sequences, the one that most closely resembles a reference sequence, known as the query. In this project, the comparison between the query and the dataset series is performed using the SAD (Sum of Absolute Differences) metric, which computes the sum of the absolute differences between corresponding elements of the two sequences. The lower the SAD value, the greater the similarity between the analyzed series and the query. The formula for the SAD metric between a series $S = [s_1, s_2, \ldots, s_n]$ and a query $Q = [q_1, q_2, \ldots, q_n]$ is as follows:

$$\text{SAD}(S, Q) = \sum_{i=1}^{n} |s_i - q_i|$$

3. Dataset generation

The dataset used in this project is generated using the mockseries library. A Python script is implemented to take as input the number of time series to generate, the length of each series, and the length of the query.

The time series generation algorithm creates synthetic sequences by combining three components: a growing linear trend, a weekly sinusoidal seasonality, and Gaussian noise. For each requested series, it generates hourly timestamps starting from January 1, 2021, computes the sum of the three components for each timestamp, and truncates the result to the desired length. This process is repeated to generate *NUM_SERIES* independent sequences of *SERIES_LENGTH* points

each, plus a query of *QUERY_LENGTH* points, saving all results in separate CSV files.

4. Sequential algorithm

Below is the pseudocode for the basic sequential implementation of the search algorithm using the SAD metric.

The core logic of the algorithm consists of two nested loops: an outer loop that iterates over each time series in the dataset, and an inner loop that scans every possible comparison window. For each window, the sum of absolute differences with respect to the query is computed, and the minimum value obtained for each series is retained. At the end, the algorithm stores the minimum SAD value found for each series and returns both the vector of all SAD values and the index of the series with the overall lowest SAD — that is, the one most similar to the query.

Algorithm 1 SAD-Based Pattern Recognition

Require: Dataset D, N time series, Query Q of length L
Initialize $bestIndex \leftarrow 0$, $bestSAD \leftarrow \infty$
Initialize $sadValues[1 \dots N] \leftarrow \infty$
for $i \leftarrow 1$ to N **do**
  for each subsequence S of $D[i]$ of length L **do**
    $sad \leftarrow \sum_{k=1}^{L} |S[k] - Q[k]|$
    if $sad < sadValues[i]$ **then**
      $sadValues[i] \leftarrow sad$
    end if
  end for
  if $sadValues[i] < bestSAD$ **then**
    $bestSAD \leftarrow sadValues[i]$
    $bestIndex \leftarrow i$
  end if
end for

4.1. SoA implementation

In the SoA implementation, the series values are stored using a vector of vectors of double elements, where the outer vector is indexed by time instant, and for each instant it contains a vector of the values of all series at that time. The data are therefore organized by time instants, so that all the values at time t are stored contiguously in the subvector indexed by t. Access to the values is provided by the $getValue$ method, which returns the element $timePoints[timeIndex][seriesIndex]$. This structure ensures that operations involving multiple series at the same time read contiguous data, while inevitably penalizing sequential accesses within a single series.

4.2. Aos implementation

In this case, in contrast to the SoA version, the data are stored using the $Sample$ struct, which contains the information for a single series, and thus the outer vector contains the collection of various series represented by $Sample$. This structure has the advantage of keeping the samples of the same series contiguous in memory, optimizing sequential reads, but it remains less suited for parallel processing.

5. Parallelization

We now proceed to implement the parallelized versions of the sequential algorithms. Specifically, two parallelized variants will be developed for both the AoS and SoA implementations. The aim is to investigate how parallelizing two different loops within the same algorithm can affect performance and determine the most suitable usage scenarios. In particular, we will independently parallelize: (i) the outer for loop, which iterates over the various time series in the dataset, and (ii) the inner for loop, which scans each individual time series to compare it against the query. For simplicity, we will refer to the parallelization of the outer loop as *Outer*, and the parallelization of the inner loop, which operates along a single series, as *Inner*.

5.1. Outer parallelization

The parallelization logic is applied using the OpenMP directive *#pragma omp parallel for* on the outer loop that iterates over the time series. Each created thread is responsible for computing the SAD for a subset of the time series in the dataset. Each thread calculates its own local *minSad* for the assigned series, and the global *bestSad* is efficiently handled via the *reduction(min:bestSad*) clause. This clause pre-

vents race conditions by combining the results from all threads at the end of the loop. The assignment of the *bestIndex*, however, requires explicit control, which is managed within a critical section using the *#pragma omp critical* directive. This ensures that only one thread at a time can update the value, thus preventing race conditions and ensuring data consistency.

5.2. Inner parallelization

In the Inner version, the outer loop iterating over the series remains sequential, while the parallelization directive is applied to the inner loop instead. This means that for each individual time series, a team of threads is spawned to handle the iterations of the inner loop, which iterates over all possible positions for the SAD computation. Unlike the outer version, it is not necessary to apply critical sections for updating the global minimum, as this is performed sequentially by the main thread after all calculations for the current series have been completed.

6. Testing

In this section, the tests performed will be presented and analyzed by testing how the various implementations discussed previously behave in different test cases depending on the size and structure of the datasets provided.

Below are the technical specifications of the machine on which the tests were performed:

Parameter	Specification
OS	Ubuntu 22.04.5 LTS x86_64
CPU	Intel i7-8565U @ 1.8 GHz
Cores per socket	4
Threads per core	2

Table 1: Specifications of the system

6.1. Test suite description

The tests to be carried out aim to determine which data structure can offer better performance for each parallelization strategy, and how the size of the datasets to be analyzed may affect the effectiveness of the strategies (Inner vs. Outer). Based on these observations, the algorithms will be tested on four different types of datasets, divided into two main categories:

- *Outer parallelization focused*: Verify the effectiveness of outer parallelization when there are many series to process in parallel, with reduced synchronization overhead.

# Series	Series length	Query length
1000	100 points	50 points
5000	100 points	50 points

Table 2: Outer parallelization focused dataset sizes

- *Inner parallelization focused*: Test inner parallelization on very long series, where parallelism over the time steps should become more advantageous.

# Series	Series length	Query length
10	5000 points	50 points
5	10000 points	50 points

Table 3: Inner parallelization focused dataset sizes

6.2. Test results and analysis

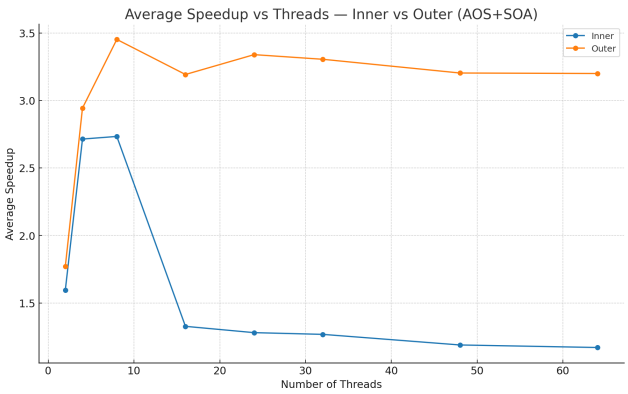

Figure 1: Inner vs Outer avg speedup/threads

From this chart we can see that the Outer parallelization scales better than the Inner version. At low thread counts the speedups increase, but by 8 threads we already observe the knee where performance begins to decline; in particular, there is a sharp collapse of the Inner variants, and

AoS–Inner even slows down relative to the sequential baseline. The causes are consistent with the algorithm's architecture: Outer is almost embarrassingly parallel, since each thread processes different series and requires very little synchronization, whereas Inner incurs significant overhead because the work must be split within the same series, degrading locality and increasing barriers and false sharing. We also note that the plateau of the Outer curves after 8–16 threads indicates that the kernel is memory-bound, so beyond this threshold the memory bandwidth becomes saturated.

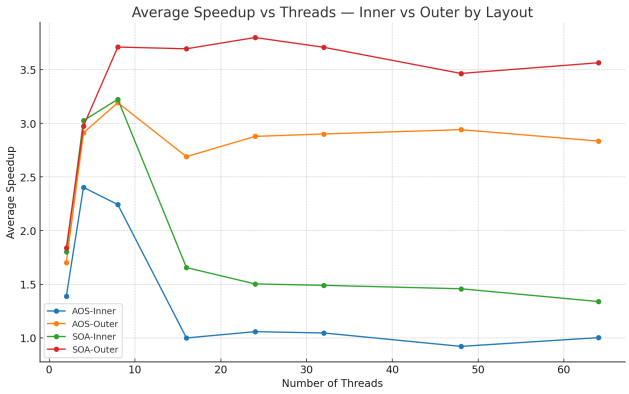

Figure 2: SoA vs AoS avg speedup/threads

In the comparison between SoA and AoS, the average speedup is consistently higher with the SoA version at all levels of parallelism, with a larger peak than AoS. This occurs because SoA favors contiguous loads that are exploited more effectively as the thread count increases, whereas AoS tends to perform optimally in the sequential setting.

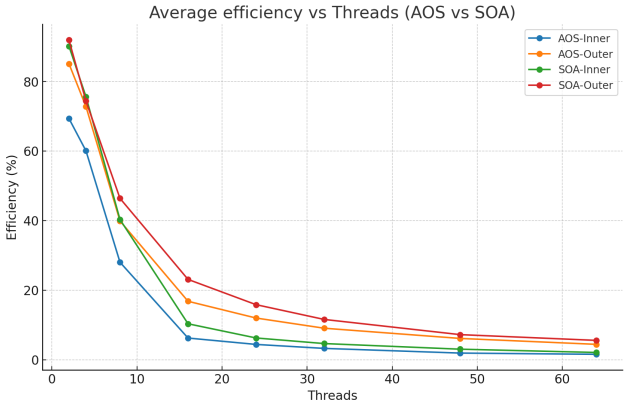

Figure 3: Average efficency vs threads

From the efficiency chart, we observe a rapid decline in efficiency as the number of threads grows; all curves decrease monotonically, in line with Amdahl's law. This confirms that parallelization overheads become increasingly dominant as parallelism increases, reducing efficiency. We also note that Outer maintains higher efficiency than Inner, precisely because Outer parallelization is essentially embarrassingly parallel (each thread works on different series), while Inner splits a single series across threads.

7. Conclusions

The experimental evidence shows that the Outer strategy scales much better than Inner: assigning independent time series to different threads minimizes synchronization, whereas splitting the same series across threads (Inner) introduces barriers, increases contention and false sharing, and degrades data locality, occasionally making AoS-Inner even slower than the sequential baseline. Comparing data layouts, SoA is systematically more efficient in parallel settings thanks to contiguous, cache-friendly accesses, while AoS tends to be more competitive only in the sequential case. The speedup curves exhibit a clear plateau beyond 8–16 threads, indicating a memory-bandwidth bound regime; as predicted by Amdahl's law, efficiency decreases monotonically with the thread count—remaining higher for Outer than Inner, but quickly converging to low values—so the marginal benefit of adding threads

becomes negligible. These results reinforce that there is no single "best" parallelization strategy: one must tailor the approach to the workload and goal to avoid poorly performing code. In our setting, the initial hypothesis is confirmed: Inner parallelization is generally inferior due to its higher coordination overhead, while Outer offers the most reliable performance gains.