Efficient Time Series Search: Sequential and parallel Approaches Using and Python's Joblib and asyncio

Dylan Fouepe
E-mail address
dylan.fouepe@edu.unifi.it

Abstract

This project explores methods for searching a given time series within a larger dataset of time series using both sequential and parallel computing approaches. The parallel implementations utilize the Joblib and asyncio libraries in Python. The study aims to evaluate the performance improvements achieved through parallelization and to identify the most effective strategies for time series analysis

1. Introduction

Time series analysis is crucial in various fields, including finance, weather forecasting, and biomedical signal processing. A common problem within this domain is the search for a specific time series pattern within a larger set of time series data. Traditional sequential search algorithms can be slow and inefficient, especially with the increasing volume of data. This project aims to address this challenge by exploring both sequential and parallel search algorithms. Using the OpenMP Joblib and asyncio for parallelization in Python, we seeek to enhance the search process's efficiency and speed. The study will provide a comprehensive evaluation of the performance gains achieved through parallel computing and offer practical insights into the implementation of efficient time series search algorithms.

1.1. Time series

Time series data consist of sequential observations indexed by time, crucial for analyzing trends, patterns, and dependencies in various domains such as finance, weather forecasting, and healthcare. These data sequences enable researchers and analysts to discern temporal behaviors, predict future trends, and derive meaningful insights from historical data. In this project, the focus lies on developing efficient algorithms to search for specific patterns within extensive time series datasets. Traditional sequential search algorithms sequentially evaluate each data point, which can be inefficient and time-consuming for large datasets. Conversely, parallel search algorithms break down the search task into smaller sub-tasks, processed concurrently across multiple processors or threads, thereby significantly enhancing search speed and efficiency.

1.2. Enhancing Time Series Search Efficiency with Python's Joblib/Asyncio

In Python, the Joblib library is utilized for its simplicity and versatility in parallel computing tasks. Joblib provides tools for efficient execution of parallel tasks, particularly beneficial for data-intensive operations such as time series analysis. Asyncio complements Python's asynchronous programming capabilities, enabling non-blocking, concurrent execution of I/O-bound tasks. This concurrency helps in efficiently managing time series data access and processing, enhancing overall performance and responsiveness in search operations. By evaluat-

ing the effectiveness of both sequential and parallel approaches in time series search, this project aims to provide valuable insights into optimizing search algorithms for large-scale time series datasets. The comparative analysis not only benchmarks performance gains achieved through parallel computing but also identifies best practices and strategies for efficient time series analysis and pattern recognition.

1.3. The Data

The M4 dataset is a collection of time series data used in the M4 competition, which aims to evaluate forecasting and pattern recognition methods. This dataset, available for download at `https://github.com/Mcompetitions/M4-methods/tree/master/Dataset`, includes a diverse set of time series with various frequencies, such as monthly, quarterly, and yearly. To benchmark pattern recognition algorithms in Python, it is often necessary to prepare a specific subset of this dataset based on length and quality criteria.

The process of extracting and cleaning the data involves several key steps. First, we randomly select a sample of time series from the M4 dataset. Next, we clean the data by removing any `NaN` (not a number) values to ensure the integrity of the data used for benchmarking. After cleaning, we check the length of each time series. If a series is shorter than a defined threshold, we extend it by appending zero values to reach the desired length. This approach allows us to generate a homogeneous dataset suitable for comparative testing of various pattern recognition methods.

1.4. The Benchmark results

In our benchmarking experiments, we utilized a machine running Ubuntu 20.04 with an Intel Core i7-1165G7 processor. The Python 3 interpreter was employed for executing our pattern recognition algorithms, and PyCharm was used as the integrated development environment (IDE). This setup provided a solid foundation for evaluating the performance of parallelized pattern recognition algorithms. Our primary focus was on two critical aspects: the impact of increasing the number of threads on performance and the effect of dataset size on speedup.

1.4.1 Benchmark number of threads

We conducted a benchmark to analyze how the number of threads affects the performance of our pattern recognition algorithms. The tests revealed that increasing the number of threads up to eight significantly enhanced the speedup. As results show:

1. NumThreads Speedup:
2. 1 1.0
3. 2 1.7
4. 4 3.0
5. 8 3.3
6. 16 3.1

As we increased from a single thread to eight threads, we observed a proportional improvement in execution time, demonstrating that our algorithm effectively leveraged parallel processing. This result is consistent with the theoretical expectation that adding more threads reduces the computation time by distributing tasks across multiple cores. Beyond eight threads, however, the performance gains plateaued, indicating that the processor's core count and thread management might have reached their optimal utilization.

1.5. Benchmark dataset size

In a separate benchmark focusing on dataset length, we investigated how varying the size of the dataset influenced the speedup of our pattern recognition algorithms.

1. Size Speedup
2. 100 0.7
3. 500 3.1
4. 1000 3.7

5. 5000 3.4
6. 10000 3.2

The results indicated that as the length of the dataset increased, the observed speedup improved. This is because larger datasets provided more opportunities for parallelization, which enhanced the algorithm's efficiency. Smaller datasets, on the other hand, offered fewer parallelizable tasks, leading to less noticeable improvements. This trend underscores the importance of dataset size in parallel computing; larger datasets tend to benefit more from parallel processing due to increased workload distribution.

1.6. Conclusion

Overall, the benchmarking results demonstrate that parallelizing pattern recognition algorithms effectively improves performance, particularly with the appropriate number of threads and sufficiently large datasets. Our experiments show that using up to eight threads can yield substantial performance gains, aligning with the capabilities of modern multi-core processors. Additionally, the performance benefits of parallel processing are more pronounced with larger datasets, which offer more opportunities for distributing computational tasks. These findings highlight the significance of optimizing both thread utilization and dataset size to achieve the best results in parallel computing for pattern recognition tasks.