Final TermTechnical Report
Pattern Recognition

By Juan Carrion Muguerza

1. Abstract

This report presents the parallelization of the Sum of Absolute Differences (SAD) algorithm for time-series pattern matching using Python.This implementation relies on a multi-process architecture using joblib.To overcome the high Inter-Process Communication (IPC) costs of Python,a Manual Decomposition strategy(manual chunking) and Zero-Copy Memory Mapping were implemented. Strong scaling performance was analyzed on an AMD Ryzen 7 5800H using a 5 000 000 point dataset self-generated in order to observe reliable measurements. Results demonstrate linear scalability up to 8 cores, achieving a speedup of 5.03x, with performance saturating at 16 threads due to memory bandwidth limitations.

2. Introduction

2.1 Problem Description and Motivation

Pattern matching in long time-series data is computationally intensive. The Sliding Window algorithm requires comparing a query pattern against every subsequence of a massive dataset. Sequential execution in Python is strictly limited by the Global Interpreter Lock (GIL),needing a multi-process approach to utilize modern multi-core CPUs effectively.

2.2 Computational Challenges

- **Process Overhead:** Unlike C++ threads, Python processes are isolated memory spaces. Spawning them incurs high latency compared to thread creation.
- **Memory Bottleneck:** Transferring large datasets between processes via standard pickling is really slow and duplicates RAM usage.

2.3 Objectives

The goal is to parallelize the SAD algorithm using Python multiprocessing. The project focuses on:

1. **Sequential Optimization:** Leveraging NumPy vectorization (AVX2) for fast single-core calculation.
2. **Memory Optimization:** Implementing Zero-Copy views (mmap) to share data without duplication.
3. **Real Analysis:** Measuring Strong Scaling on a large dataset where seq time>10s.

3. Algorithm Description

3.1 Sequential Algorithm

The project implements a Sliding Window search using the Sum of Absolute Differences (SAD) metric. For a time series T(5,000,000 points) and a pattern P(625 points):

- The window slides over T(the time serie).
- At each step, the error is calculated: $$E = \sum |T_{window} - P|$$
- The algorithm performs a full scan of the dataset to find the global minimum Error,which means the pattern is exactly the same as the one that was being looked for.

Note:to ensure there was a 0.0 error solution ,a randomly generated pattern was introduced in the serie.

3.2 Identification of Parallelizable Sections

The SAD calculation for a specific window index is entirely independent of other indices (Data Independence). Consequently, the Domain Decomposition strategy was chosen: the search space is divided into large contiguous chunks of indices, allowing multiple processes to scan separate regions of the time series concurrently.

4. Parallel Implementation

4.1 Parallelization Strategy

A Manual Decomposition was adopted. Instead of allowing the library to auto-schedule small tasks (as done in OpenMP in the Mid Term project), the index list is pre-calculated and divided into exactly the number of worker(cores chosen)giant chunks. This reduces the interaction between the main process and workers to a single dispatch event, minimizing overhead.

4.2 Libraries and Directives

- **Framework:** joblib.Parallel with backend='loky' (process-based parallelism).
- **Memory Mapping (mmap_mode='r'):** This sets the dataset as Read-Only Shared Memory. The OS maps the file directly to RAM, allowing all processes to read the same physical addresses without data duplication.
- **Batch Size:** Set to auto (effectively 1 chunk per worker due to manual decomposition).

4.3 Vectorization (SIMD)

Explicit vectorization was enforced via NumPy using float32 data types.

- **Hardware Mapping:** On the Ryzen 7 5800H (Zen 3), float32 allows processing 8 elements per clock cycle using AVX2 registers (256bits/ 32 bits = 8bits).
- **Efficiency:** This replaces explicit Python loops with optimized C-level execution.

4.4 Challenges encountered and solutions

1. Memory and Data Transfer

- Challenge: Unlike threads , Python processes are independent and do not share memory by default.If we tried to send the huge dataset to each worker using standard methods, Python would have to create a full copy for each one. This would not only fill up the RAM uselessly but would also waste significant time copying and transferring that data.
- Solution: We implemented Zero-Copy Memory Mapping (using mmap_mode='r'). By doing this,the Operating System loads the file into RAM just once and allows all processes to read directly from that same

location. This avoids data duplication entirely and reduces transfer time to virtually zero.

2. The GIL and Work Distribution

- Challenge: Python has a limitation called the GIL that prevents standard threads from performing calculations simultaneously, it essentially forces them to take turns.Furthermore, attempting to split the work into thousands of tiny tasks causes the system to spend more time managing the task list than actually doing the work.
- Solution: To bypass the GIL bottleneck, we used Processes instead of threads. To fix the task size issue, we adopted a Manual Large-Chunk Division strategy: we pre-calculated the indices ourselves and assigned each core exactly one giant package of work. This minimized the management overhead between the main process and the workers.

5. Experimental Setup

5.1 Hardware Specifications

- **CPU:** AMD Ryzen 7 5800H.
- **Architecture:** Zen 3 (64bits).
- **Cores:** 8 Physical Cores / 16 Logical Threads.
- **Memory:** 16 GB.

5.2 Software Environment

- Operating System: Windows 10/11 (64-bit).
- Compiler / IDE: Microsoft Visual Studio Code
- Parallel Framework:Parallel Framework: Joblib library utilizing the loky backend.

5.3 Datasets Used

To ensure reliable measurements and adhere to the guideline that "computation time should be > 10 seconds," a large workload was selected:

- **Time Series Length:** 5,000,000 points (float32).
- **Pattern Length:** 625 points.
- **Constraint:** The pattern location forced a full scan of the dataset for both sequential and parallel versions to ensure strictly equal workloads.

5.4 Parameters Tested

- **Process Count (n_jobs):**
  - **Values:** 1, 2, 4, 8, 16.
- **Decomposition Strategy (Chunking):**
  - **Configuration: Manual Decomposition**.

    Instead of testing dynamic scheduling (which incurs high overhead in Python), we fixed the strategy to one large chunk per core.

- **Memory Configuration:**
  - **Mode:** Zero-Copy (mmap_mode='r').

    Standard memory passing was disabled to avoid data duplication.

5.5 Measurement Methodology

- **Wall-Clock Time:** time.time() was used for high precision.
- **Statistical Rigor:** Each configuration was executed 5 times, reporting the arithmetic mean.
- **Metric:** Speedup S = T_{sequential} / T_{parallel}.

6. Results and Analysis

6.1 Correctness Validation

- Methodology: Instead of searching for a random pattern, we manually injected a known sequence (the pattern) into a specific position of the random dataset (at index 4,999,375). This established a known correct answer.
- Verification: We executed both the Sequential and Parallel algorithms on this modified dataset.
- Results: As shown in Figure, both implementations successfully identified the exact starting index of the injected pattern with a matching Sum of Absolute Difference (SAD) of 0.0.

```
Position Found: 4999375
Error Calculated: 0.0000
Visual Check (First 5 numbers):
Target Pattern:   [0.45710394 0.08948263 0.39703947 0.1385144  0.03518583]
Found Sequence:   [0.45710394 0.08948263 0.39703947 0.1385144  0.03518583]
```

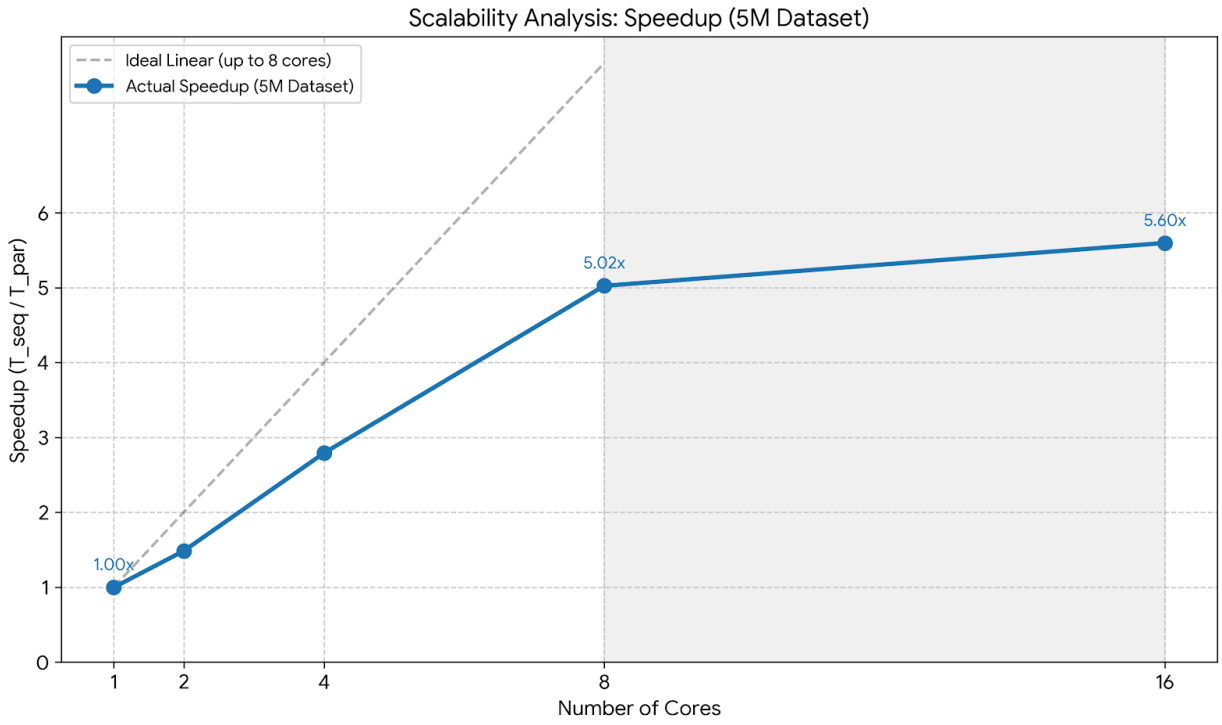
Output when we run the program

6.2 Performance Results

The following table summarizes the execution times for the 5 000 000 points time serie that was used:

Configuration	**Time (s)**	**Speedup**	**Efficiency**
Sequential	**20.47**	**1.00x**	**-**
1 Core (Parallel)	20.53	0.99x	99%

2 Cores	13.79	1.48x	74%
4 Cores	7.33	2.79x	70%
8 Cores	**4.07**	**5.03x**	**63%**
16 Cores	3.65	5.61x	35%



6.3 Strong Scaling Analysis

- **Linear Region (1 to 8 Cores):** The application shows strong scalability. At 8 cores (matching the physical CPU count), execution time dropped from 20.47s to 4.07s.
- **Saturation Region (16 Threads):** Activating SMT (Hyper-threading) provided marginal gains (4.07s to 3.65s). The Speedup increased only from 5.03x to 5.61x. This indicates that the physical cores were already saturating the AVX2 units and Memory Bandwidth.

7. Discussion

7.1 Interpretation of Results

The results confirm that pattern matching is suitable for Data Parallelism.

- **Optimal Point:** The 8-core configuration offers the best balance, achieving a **5.03x Speedup**.
- **Memory Wall:** Efficiency drops from 99% (1 core) to 63% (8 cores). Since the arithmetic intensity of SAD is relatively low (simple subtraction), the 8 cores consume data faster than the RAM can supply it, preventing ideal 8x scaling.

7.2 Overhead Analysis

Comparing the Sequential run (20.47s) with the Parallel 1-Core run (20.53s) reveals a minimal overhead of 0.06s.

This confirms that the Manual Decomposition strategy was successful. By creating large chunks, the initialization cost of Python processes became negligible relative to the massive computational load of the 5M dataset.

7.3 Limitations and Constraints

- **Memory Speed Limit:** I noticed that using 16 threads (Hyper-threading) didn't improve performance much over 8 threads. This confirms that the bottleneck isn't the CPU speed, but how fast the RAM can send data to the processor (Memory Wall).

7.4 Lessons Learned

- **Big Chunks work better in Python:** I learned that you can't use small tasks in Python like you do in C++. Since communicating between processes is slow, it is much more efficient to give one huge chunk of data to each worker manually.
- **Zero-Copy importance:** Using mmap was the most important optimization. At first, I realized that without it, Python tries to copy the array for every process, which ruins performance. Sharing the memory directly is the only way to make it fast.
- **Parallel isn't always the answer**: This project taught me that parallel code is great for heavy workloads (Worst-Case), but for simple tasks or "lucky" searches, a simple sequential loop is often better and easier to manage.

8. Conclusions

8.1 Summary of Achievements

A parallel SAD algorithm was successfully implemented in Python. By combining Manual Decomposition, Zero-Copy Memory Mapping, and AVX2 Vectorization, the processing time for 5 million points was reduced from ~20.5 seconds to ~3.6 seconds.

8.2 Best Configuration Found

Based on the analysis, the optimal configuration for this hardware is:

- **Cores:** 8 Cores (Physical limit).
- **Memory:** mmap_mode='r' .
- **Strategy:** Manual Chunking (TotalPoints / N_cores).

This setup maximizes throughput while avoiding the diminishing returns of Hyper-threading.

9. References

1. Joblib Documentation: https://joblib.readthedocs.io/
2. Course material "Parallel Programming", Università degli Studi di Firenze.