# Using OpenMP to Speed Up Time Series Pattern Recognition Algorithms

Tommaso Botarelli
*tommaso.botarelli@edu.unifi.it*
7136210

## Abstract

*Time Series Pattern Recognition is a computationally intensive task that benefits from parallelization to improve performance. In this report, we show our sequential and parallel implementations to address the problem and we analyze their performances. We conduct strong and weak scaling experiments on different parallelization strategies implemented using OpenMP. Our analysis highlights the impact of system architecture on performance, demonstrating how factors such as memory hierarchy and core count influence speedup and efficiency. The results emphasize the importance of understanding the underlying system when optimizing parallel implementations and provides an analysis cue for implementing generic algorithms using OpenMP.*

## 1. Introduction

In an increasingly data-driven world, time series pattern recognition has emerged as a powerful tool for uncovering hidden trends, forecasting future events, and supporting data-driven decision-making. From finance and economics to healthcare, climate science, and beyond, the ability to detect patterns in sequential data is critical to driving innovation and efficiency.

[TODO: descrivere come è strutturato]

## 2. Time Series Pattern Recognition

*Time Series Pattern Recognition* can be of different types. In this paper we will refer with the general name of ”Time Series Pattern Recognition” to the problem of *Subsequence Matching*. This problem involves identifying a subsequence within a larger time series where a predefined distance metric is minimized with respect to a smaller reference time series. Formally, let:

$$X = \{X_1, X_2, \dots, X_N\}$$
$$x = \{x_1, x_2, \dots, x_m\}$$

where $X$ is the large time series of length $N$ and $x$ is the smaller time series of length $m$ with $m \leq N$.

The goal is to find the subsequence $X_{i:i+m-1} = \{X_i, X_{i+1}, \dots, X_{i+m-1}\}$ of $X$ that minimizes a distance metric $D(\cdot, \cdot)$ with respect to $x$. This can be expressed as:

$$i^* = \underset{i \in \{1,\dots,N-m+1\}}{\arg\min} D(x, X_{i:i+m-1})$$

where $i^*$ is the start index of the optimal subsequence and $X_{i:i+m-1}$ is the subsequence of $X$ from element in $i$ position to element in $i+m-1$ position. The distance metric $D$ could be any suitable measure.

```
compute(entireTimeSerie, timeSeriesToSearch
    ) {
  // ...
  // n = size of entire time serie
  // m = size of time series for minimizing
      distance
  for (i = 0; i < n - m; i++) {
    distance = 0;
    for (j = 0; j < m; j++) {
      distance += abs(entireTimeSerie[i +
          j] - timeSeriesToSearch[j]);
    }

    if (distance < minDistance) {
      minDistance = distance;
      minIndex = i;
    }
  }
}
```

Algorithm 1: Pseudo code for the sequential implementation

### 2.1. Distance function

In our case we define the distance function $D$ as *Sum of Absolute Differences (SAD)*. Given two time series $x, y$ of size $m$ we can define formally the SAD metric as:

$$D(x, y) = SAD(x, y) = \sum_{i=1}^{m} |x_i - y_i|$$

## 3. Sequential Algorithm

In this sections we present our implementation for the sequential algorithm. Note that the following algorithm will provide our baseline for the following experimentation.

The algorithm is a simple implementation. We loop over the big time serie and an inner loop calculates the distance metric. If the distance calculated is less than the actual *"best"* we update it and at the same time we save the index. At the end of the execution we are sure that we have found the optimal index $i^*$ defined in the previous section.

## 4. Parallelization

In this section we first discuss about the parallel programming paradigm followed in the experimentation and then provide an overview of OpenMP. Then we show in detail the parallelized algorithm.

### 4.1. Shared Memory Parallel Programming

In this work we implemented a parallel algorithm following the *Shared Memory Parallel Programming Paradigm*. Shared memory parallel programming is a type of parallel computing where multiple processors (or threads) access and manipulate a common, shared memory space. This approach allows for fast communication between threads since data is directly accessible to all threads without the need for explicit data transfer. However, it also requires careful management of memory access to avoid conflicts.

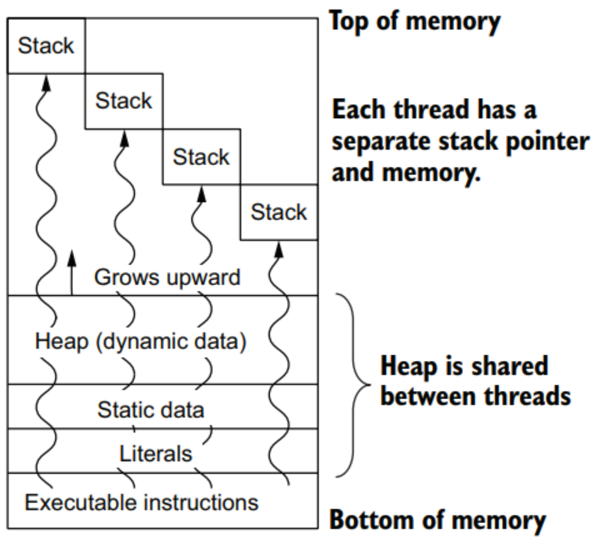

Figure 1: *The threaded memory model [1]*

### 4.2. OpenMP

OpenMP is one of the most widely supported open standards for threads and shared-memory parallel programming [1].

OpenMP is a direct implementation of the shared memory parallel programming paradigm. It allows multiple threads to access and manipulate the same memory space, making it ideal for tasks that require frequent data sharing and low-latency communication between threads. Through its simple directive-based approach, OpenMP abstracts much of the complexity involved in thread management, synchronization, and workload distribution.

```
compute(entireTimeSerie, timeSeriesToSearch
    ) {
 //...
  #pragma omp parallel for private(
      distance) shared(minDistance)
      schedule(<scheduling-type>)
  for (int i = 0; i < n - m; i++) {
     distance = 0;
     #pragma omp simd
     //compute distance: line 7-9 of
         sequential

     #pragma omp critical
     if (distance < minDistance) {
        minDistance = distance;
        minIndex = i;
     }
  }
}
```

Algorithm 2: Pseudo code for the simple parallel implementation. Scheduling type can be choosen from several options see [5]

OpenMP gives the possibility to control how threads are distributed to processors (using `OMP_PROC_BIND` environment variable). It helps manage thread affinity, which can impact performance by reducing cache misses and improving memory locality.

We can select the value to be (see [4] for a complete list):

- `false`, no binding is applied to the threads. Threads are allowed to migrate freely across available processors.
- `close`, threads will be packed onto a single socket first before using another one. Once a socket is full, additional threads are placed on the next socket.
- `spread`, OpenMP assigns each thread to a separate physical core first (threads are evenly distributed across the sockets).

### 4.3. Parallel Algorithm

OpenMP allows for a simple parallelization of a sequential code using `#pragma omp for [options/clauses]` (Algorithm 2). Even if this can be sufficient in the majority of cases we will show that the best performance can be reached by combining `#pragma omp parallel` sections and standard `for` loop. This can be done by dividing the problem domain manually (Algorithm 3).

```
compute(entireTimeSerie, timeSeriesToSearch) {
 //...
  #pragma omp parallel shared(globalMinDistance, globalMinIndex)
  {
     int threadId = omp_get_thread_num();
     int numThreads = omp_get_num_threads();
     int startIndex = n * threadId / numThreads;
     int endIndex = n * (threadId + 1) / numThreads;
     if (endIndex == n){
        endIndex -= m;
     }
     //...
     for (i = startIndex; i < endIndex; i++) {
        // standard for loop search
        each thread save its best result in 'minDistance'
     }
     #pragma omp critical
     {
        if (minDistance < globalMinDistance) {
           globalMinDistance = minDistance;
           globalMinIndex = minIndex;
        }
     }
  }
}
```

Algorithm 3: Pseudo code for the 'manual' parallel implementation. The domain of the problem is divided in the code and the code is in a `parallel` section to reach maximum performance

Another algorithm that we provide is Algorithm 4. As we will show this implementation is necessary to reach better performance than the simple one.

## 5. Experimentation

In this section we show the experimental setup. We first introduce the experimental environment in which all the tests were conducted. Then a detailed description of the tests performed and the data used will be shown.

### 5.1. Performance Tests: Strong vs. Weak Scaling

To collect information about the performance of a parallel code we often use two tests: *Weak* and *Strong* scaling (see Figure 2).

- Strong scaling test measures performance when the problem size is fixed, and the number of threads/processors increases. Ideal strong scaling means the execution time should decrease proportionally as more threads are added.

- Weak scaling test measures performance when the workload per thread is kept constant while increasing the total number of threads. Ideal weak scaling means the execution time remains constant as the problem size grows in proportion to the number of threads.

Both strong and weak scaling assume to have a dataset larger than the cache memory. Indeed the real performance scaling can be viewed only by deleting all the optimization derived from a dataset stored entirely in cache.

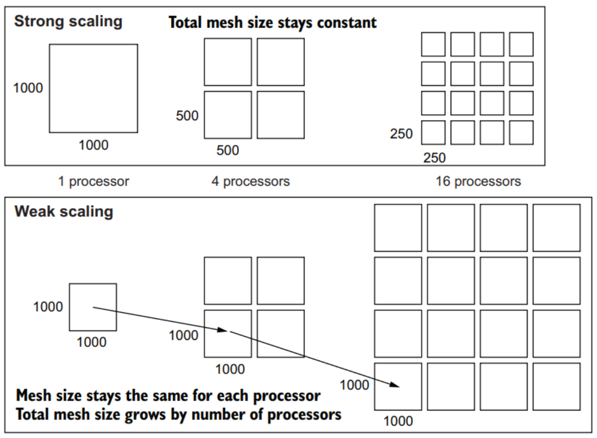
Strong scaling
Total mesh size stays constant
1000
1000
500
500
250
250
1 processor
4 processors
16 processors
Weak scaling
1000
1000
1000
1000
1000
1000
Mesh size stays the same for each processor
Total mesh size grows by number of processors

Figure 2: *Graphical representation of strong vs weak scaling (Image from [1])*

### 5.2. Experimental Environment

All the experimentation has been conducted in a dual-socket Intel Xeon Silver 4314 system with 32 physical cores and 64 threads. The entire info about the system are shown in Table 1.

| Name | Value |
|---|---|
| Clock rate [max] | 2.4 [3.4] GHz |
| # Sockets | 2 |
| Cores | 32 (16 per socket) |
| # Logical Processors | 64 (2 per core) |
| L1d (Data cache) | 1.5 MiB (48 KiB per core) |
| L1i (Instruction cache) | 1 MiB (32 KiB per core) |
| L2 cache | 40 MiB (1.25 MiB per core) |
| L3 cache | 48 MiB (24 MiB per socket) |
| # NUMA nodes | 2 |

Table 1: Hardware architecture used for the experimentation. Note that L1 and L2 cache are private for each core but L3 is shared between the same socket.

## 5.3. Experimental Setup

In this work we made performance analysis using the tests already defined in 5.1 (strong and weak scaling). The scaling has been performed in various scenarios varying the bind of threads (false/close/spread) and the scheduling type (dynamic/static) as shown in Table 2.

```
compute(entireTimeSerie, timeSeriesToSearch
    ) {
  //...
    #pragma omp parallel for private(
        distance) reduction(min:minDistance)
        schedule(<scheduling-type>)
    for (int i = 0; i < n - m; i++) {
       distance = 0;
       #pragma omp simd
       //compute distance: line 7-9 of
           sequential

       if (distance < minDistance) {
          #pragma omp critical
          {
             if (distance < minDistance) {
             //..
             }
          }
       }
    }
}
```

Algorithm 4: Pseudo code for the "smarter" simple parallel implementation

Then for each combination it has been analyze a strong scaling with increasing number of threads $(1, 2, 4, 8, 16, 32, 64)$ and weak scaling with increasing domain size (Table 3). For the strong scaling the data size is chosen to be $17'088'000$ floating point values (this gives a size of $\approx$ 65 MiB bigger value than the L3 cache).

| Algorithm | Threads Bind |
|---|---|
| $Seq$ | - |
| $Simple$ | false\|close\|spread |
| $Smart$ | false\|close\|spread |
| $Manual$ | false\|close\|spread |

Table 2: Types of algorithms analyzed. $Seq$ is Algorithm 1, $Simple$ is Algorithm 2, $Smart$ is Algorithm 4 and $Manual$ is Algorithm 3

For each experiment the execution time of the algorithms has been taken from an average of 10 executions. The size of the sequence to search ($x$ in Section 2) is chosen to be 1000 floating point values fixed for each tests.

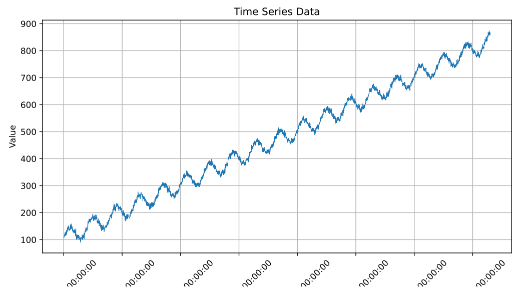

Figure 3: *Example of time series generated with Mockseries*

## 5.4. Data Collection

The data used for the experimentation has been created artificially using a Python library called `mockseries` [3]. This library provide a simple interface for the creation of user defined pseudo random time series (see Figure. 3). For the purpose of this work create artificial data of user defined length is a big advantage.

| # threads | # elements | Data size in MiB [MB] |
|---|---|---|
| 1 | 13672800 | 52 [54] |
| 2 | 27345600 | 104 [109] |
| 4 | 54696960 | 208 [218] |
| 8 | 109382400 | 417 [437] |
| 16 | 218764800 | 834 [874] |
| 32 | 437529600 | 1668 [1749] |
| 64 | 875059200 | 3337 [3500] |

Table 3: Weak scaling data sizes: number of float points in the entire domain

## 6. Results

In this sections all the results of the experimentation will be provided. We first show the results of *strong* and *weak*

| Algo | Threads | Avg. Execution Time [ms] | | | Speedup | | | Efficiency | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *False* | *Close* | *Spread* | *False* | *Close* | *Spread* | *False* | *Close* | *Spread* |
| $Seq$ | 1 | 20292 | - | - | 1 | - | - | 100% | - | - |
| $Smart$ | 2 | 10623 | 10783 | 10429 | 1.91 | 1.88 | 1.95 | 96% | 94% | 97% |
| | 4 | 5503 | 5484 | 5285 | 3.69 | 3.70 | 3.84 | 92% | 93% | 96% |
| | 8 | 3068 | 2876 | 2732 | 6.61 | 7.06 | 7.43 | 83% | 88% | 93% |
| | 16 | 1709 | 1546 | 1508 | 11.87 | 13.13 | 13.46 | 74% | 82% | 84% |
| | 32 | 1606 | 1116 | 1695 | 12.64 | 18.18 | 11.97 | 39% | 57% | 37% |
| | 64 | 2671 | 1814 | 2725 | 7.60 | 11.19 | 7.45 | 12% | 17% | 12% |
| $Manual$ | 2 | 9933 | 10436 | 10118 | 2.04 | 1.94 | 2.01 | 102% | 97% | 100% |
| | 4 | 5551 | 5170 | 5005 | 3.66 | 3.92 | 4.05 | 91% | 98% | 101% |
| | 8 | 3330 | 2615 | 2498 | 6.09 | 7.76 | 8.12 | 76% | 97% | 102% |
| | 16 | 1484 | 1372 | 1315 | 13.67 | 14.79 | 15.43 | 85% | 92% | 96% |
| | 32 | 768 | 765 | 740 | 26.42 | 26.53 | 27.42 | 83% | 83% | 86% |
| | 64 | 568 | 629 | 646 | 35.73 | 32.26 | 31.41 | 56% | 50% | 49% |

Table 4: Strong scaling results: comparison between $Smart$ and $Manual$

scaling respectively and then we provide a global discussion of the them.

## 6.1. Strong Scaling

In a first phase, after the execution time of the sequential algorithm was known, we perform an evaluation between two different parallel implementation. The comparison was taken between Algorithm 2 and Algorithm 4. The slightly differences between the codes lead to a huge different in practical performance.

As we can see clearly in Figure 4, even if for each algorithms we can observe a drop in the efficiency starting from $8/16$ threads respectively the overall performance of $SmartSPar$ is better than $SPar$. The differences in performance for this type of algorithm can be explained in the smarter way in which we have access to the critical section. If for $SPar$ we have a contention on the variable `minDistance` between the threads for each loop, in $SMartSPar$ the contention between the threads is decreased by the optimization of the clause $reduction(min : minDistance)$ and more important the `if` statement for the `minDistance` before access the critical section. This provide a more optimized way to update this variable instead of losing time waiting for the release of the critical section. As we can see this behavior is more clear with an increasing number of threads. This is obvious because the contention increase with the number of threads (note that the system has 64 virtual cores).

As we've already seen even if OpenMP is a simple library to use, reaching good performance is not so straight forward. We can push our optimization higher by implementing a *manual* parallelization (Algorithm 3). The results of the comparison are shown in Table 4.

We can clearly see that the performance of $MPar$ are better in general and more robust to the scaling. The best performance can be reach with `bind = false` configuration even if with 64 threads the efficiency is low. The huge drop in performance from 32 to 64 threads are likely to be caused by contention between the threads that at this point are placed in the two sockets. The fact that the threads are spread in two different sockets cause the latency and the drop in performance from 32 to 64 threads.

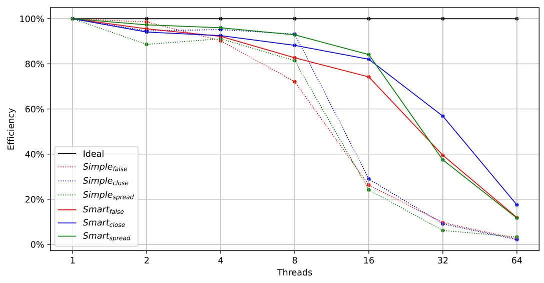

Figure 4: *Comparison of the efficiency between Algorithm 2 and Algorithm 4*

From the table we can see that the best performance until 64 threads are reached by $MPar$ with `bind = spread`. With spread, OpenMP assigns each thread to a separate physical core first. This avoids cache contention and maximizes memory bandwidth per core because the L1 and L2 cache are dedicated only to one thread instead of 2. Indeed this provides an experimental speedup greater than the ideal one for $< 16$ threads.

This because `spread` avoids cache contention and maximizes memory bandwidth per core. With 32 threads all the core is running a thread and this slow down the performance. With 64 threads the advantage of spread is vanished

| Algo | Threads | Avg. Execution Time [ms] | | | Efficiency | | |
|---|---|---|---|---|---|---|---|
| | | *False* | *Close* | *Spread* | *False* | *Close* | *Spread* |
| $Seq$ | 1 | 15743 | - | - | 100% | - | - |
| $MPar$ | 2 | 15249 | 15922 | 15238 | 103% | 99% | 103% |
| | 4 | 15431 | 16062 | 15347 | 102% | 98% | 103% |
| | 8 | 15686 | 16171 | 15474 | 100% | 97% | 102% |
| | 16 | 16521 | 16939 | 16434 | 95% | 93% | 96% |
| | 32 | 18107 | 18877 | 18087 | 87% | 83% | 87% |
| | 64 | 19339 | 19440 | 19385 | 81% | 81% | 81% |

Table 5: Weak scaling results

because each core has now two different threads and cache contention become higher and higher.

Also `bind = close` has quite good performance. With close, OpenMP assigns threads to adjacent CPU cores first, meaning that Threads will be packed onto a single socket first before using the second socket. Once a socket is full, additional threads are placed on the next socket. This approach keeps threads physically close, which reduces inter-socket communication but increases local contention. In our case from Table 4 we can see a big drop in performance from 16 to 32 threads. This because we pass from using a single (and full) physical socket to both socket adding some latency due access data on the other socket's memory.

Finally we can say that this implementation of the algorithm is memory bound because of the huge drop in performance when we use all the threads available.

### 6.2. Weak Scaling

For weak scaling we selected a dataset of the size shown in Table 3. This selection provides a workload greater than L3 cache for each thread so we can experiment the true speedup.

The results of this experimentation are limited to the best case of the previous section ($ManP$) and are shown in Table [5. As before the table shows all the results for three different bind selection.

As we could expect the results show that `spread` and `false` reach the best performance. This behaviour can be explained by the fact that with `close` the threads are distributed in the same socket first. So initially the performance can have a slow down due to memory congestion in the same socket.

With `spread` the threads are distributed in two sockets and with `false` the OS distributes them based on the system status. The performance of `false` can be explained by the fact that the experimentation is conducted not in a free environment so give the OS the possibility to optimize the distribution based on the actual core occupation.

`spread` provides the best performance due to improved memory utilization. Indeed with `spread` we use from the beginning all the sockets available. This provides less I/O congestion because all the socket memory managements are used from the first moment.

So we can finally say that the performances reached by $Manual$ are good even with a huge scaling of $64\times$ the base case and the efficiency of the code stay high with incresing the domain size.

## 7. Conclusion

As we've already discussed the results reflect what we could expect from the system architecture used in experimentation.

From the work done, one can learn how just using `#pragma omp parallel for` is not enough to achieve performant code. It is therefore often necessary to write ad hoc code to achieve maximum performance.

Another important fact that can be learned from this work is that although OpenMP appears to be an easy-to-use library, achieving high performance and performance in strong and weak scaling requires a thorough understanding of the system architecture and the problem being addressed.

This is especially true in the case of systems with multiple NUMA nodes. Indeed, in these systems, as has been shown, the memory utilization component plays a critical factor in achievable performance.

### 7.1. Future Improvements

The main issue with the experimentation is the lack of a dedicated system. Since it was carried out in a system whose conditions were not of absolute absence of other tasks the results may lead to differences from the real ones.

This issue, although mitigated by the multiple execution of the algorithms and the consideration of averaging times, could be addressed in future experimentation by having a dedicated and free system at hand.

### 7.2. Additional Materials

The code and the experimentation results are available in GitHub (see [2]).

## References

[1] Yuliana Zamora Robert Robey. *Parallel and High Performance Computing*. 2020.

[2] Tommaso Botarelli. *Parallel Computing Project*. URL: `https://github.com/TommasoBotarelli/ParallelComputing_UNIFI_Project/tree/master/pattern_recognition`.

[3] cyrilou242. *Mockseries*. URL: `https://github.com/cyrilou242/mockseries`.

[4] Documentation OpenMP. *OPENMP API Specification*. URL: `https://www.openmp.org/spec-html/5.0/openmpse52.html`.

[5] Documentation OpenMP. *Worksharing-Loop Construct*. URL: `https://www.openmp.org/spec-html/5.0/openmpsu41.html`.