Time Series Pattern Recognition with OpenMP

[B024314] Parallel Computing Exam

Tommaso Botarelli

23/04/2025

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Da un secolo, oltre.

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

Introduction

1 Time Series Pattern Recongnition

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.
Applications:

- Finance and economics
- Healthcare
- . . .

Detect patterns in sequential data is critical to driving innovation and efficiency.

Definition

Time Series Pattern Recognition 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:

$$X = \{X_1, X_2, \ldots, X_N\}$$

$$x = \{x_1, x_2, \ldots, x_m\}$$

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

Goal

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

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

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.

Distance Function

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:

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

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

Implementation

2 Sequential

A simple and intuitive sequential implementation has been developed to test the performance of the parallel algorithms.

```
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;
      }
  }
}
```

Table of Contents

3 Parallelization

Shared Memory Parallel Programming

3 Parallelization

Shared memory parallel programming is a type of parallel computing where multiple processors (or threads) access and manipulate a common, shared memory space. Characteristics:

- Fast communication between threads
- It requires careful management of memory access to avoid conflicts and memory bottlenecks

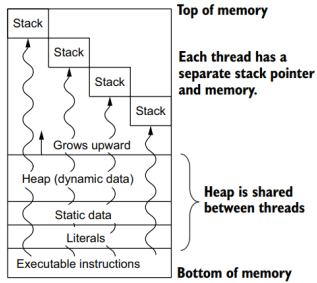
OpenMP

3 Parallelization

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

It allows multiple threads to access and manipulate the same memory space.

Through its simple directive-based approach, OpenMP abstracts much of the complexity involved in thread management, synchronization, and workload distribution.



Threads Position

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

- `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).

Implementations

3 Parallelization

OpenMP allows for a simple parallelization of a sequential code using `#pragma omp for [options/clauses]`.
The best performance can be reached by combining `#pragma omp parallel` sections and standard `for` loop.

In the work three different implementations have been built:

- *Simple*
- *Smart*
- *Manual*

Simple

3 Parallelization

The first implementation is built by using the standard OpenMP for loop parallelization.

```
1  compute(entireTimeSerie, timeSeriesToSearch) {
2   //...
3     #pragma omp parallel for private(distance) shared(minDistance) schedule(<type>)
4     for (int i = 0; i < n - m; i++) {
5         distance = 0;
6         #pragma omp simd
7         //compute distance: line 7-9 of sequential
8
9         #pragma omp critical
10        if (distance < minDistance) {
11            minDistance = distance;
12            minIndex = i;
13        }
14    }
15 }
```

The critical section is accessed only if the actual distance is less than the actual minimum. This helps reducing the contention between threads.

```
1  compute(entireTimeSerie, timeSeriesToSearch) {
2   //...
3     #pragma omp parallel for private(distance) reduction(min:minDistance) schedule(<type>)
4     for (int i = 0; i < n - m; i++) {
5         distance = 0;
6         #pragma omp simd
7         //compute distance: line 7-9 of sequential
8
9         if (distance < minDistance) {
10            #pragma omp critical
11            {
12                if (distance < minDistance) {
13                //..
14                }
15            }
16        }
17    }
18 }
```

Manual calculation of the index allows to *minimize* the critical sections accesses and push to maximum performances

The number of accesses to the critical section is equal to the number of threads $\rightarrow$ reduced waiting time

```
1  compute(entireTimeSerie, timeSeriesToSearch) {
2   //...
3     #pragma omp parallel shared(globalMinDistance, globalMinIndex)
4     {
5         int threadId = omp_get_thread_num();
6         int numThreads = omp_get_num_threads();
7         int startIndex = n * threadId / numThreads;
8         int endIndex = n * (threadId + 1) / numThreads;
9         if (endIndex == n){
10            endIndex -= m;
11        }
12        //...
13        for (i = startIndex; i < endIndex; i++) {
14            // standard for loop search
15            each thread save its best result in 'minDistance'
16        }
17        #pragma omp critical
18        {
19            if (minDistance < globalMinDistance) {
20                globalMinDistance = minDistance;
21                globalMinIndex = minIndex;
22            }
23        }
24    }
25 }
```

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

4 Experimentation

Experimental Environment

All the experimentation has been conducted in a dual-socket *Intel Xeon Silver 4314* system with 32 physical cores and 64 threads.

L1 and L2 cache are private for each core but L3 is shared between the same socket.

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

Experimental Setup

4 Experimentation

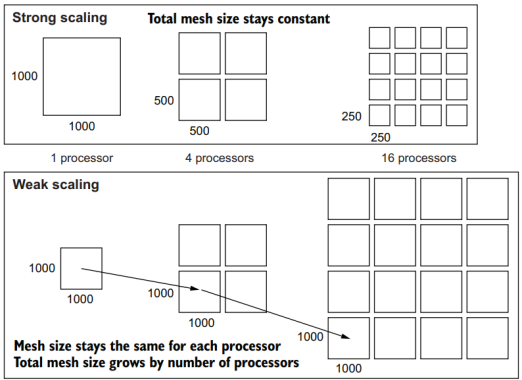
Two performances tests have been performed in this work: **strong** and **weak scaling**.

In each test all the algorithms shown have been compared with three different threads configuration: **false, close, spread** bind.

For each tests an average of 10 executions has been taken.

C++ and **CMake**

Compilation: GCC with −O3 enabled for best optimization and vectorization



Strong Scaling Configuration

With strong scaling, the problem size remains fixed and the number of threads is increased.
Strong scaling is used to understand whether the implemented algorithm is robust with respect to increasing the number of threads.
Ideally we would like the execution time to decrease proportionally to the increase in the number of threads.

The domain size for this configuration is chosen to be bigger than L3 cache to have a good comparison between the algorithms and real world speedup.

- $|X| = 17'088'000$ floating points value (≈ 65 Mib)
- $|x| = 1000$ floating points value

Weak Scaling Configuration

With weak scaling you scale the problem size proportionally to the number of threads Weak scaling allows us to see whether a certain algorithm is robust to increasing problem domain size. Ideally we would like to see a constant execution time as the number of threads and problem size increase.

# 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]

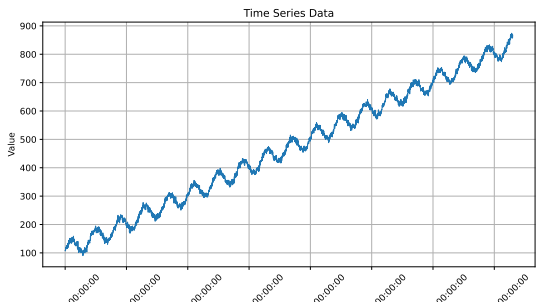
Data Collection

4 Experimentation

The data has been created artificially using a Python library called `mockseries`.

Advantages:

- Simple interface for the creation of user defined pseudo random time series
- Data size defined by the user



UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

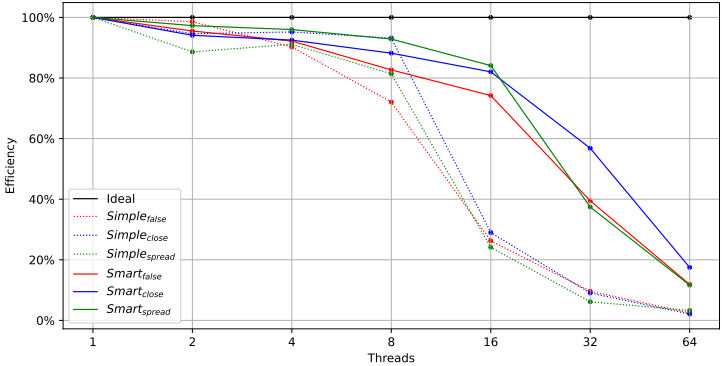
Table of Contents

5 Results

Strong Scaling: *Simple* vs *Smart*

5 Results

- Overall performances of *Smart* are better than *Simple*
- Different threads bind configuration don't add particular advantages in the results
- Big drop in performances after 16 threads in all threads bind configuration



Strong Scaling: Why?

5 Results

Smart provide a more optimized way for access the critical section:

- Addition of $reduction(min : minDistance)$ and double `if` statement
- `minDistance` is updated in a more optimized way
- The time losed in waiting to get the access to the variable is smaller

Big drop from 16 to 32 threads explained by a combination of:

- Shift from 1 to 2 physical cpu utilization
- Inefficient data organization

Strong Scaling: *Smart* vs *Manual*

5 Results

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%

Strong Scaling: Why?

5 Results

Superlinear performance for a low number of threads is explained by better cache utilization.

The increase in overall performances is explained by a combination of:

- Better threads memory organization: the data is closer to the threads and less L1 cache misses are expected
- Better division of the data between the threads
- Better parallel section: more control over the threads work

`spread` is better for almost all the configuration because $Manual$ allows to a better memory management and spreding the thread in two NUMA nodes allow to better cache utilization (all the cache is used).

The final big drop in efficiency is due to memory bottleneck and threads contention.

Weak Scaling

5 Results

Manual provide good weak scaling capabilities.

This confirms:

- Good workload division between the threads
- Small contention between the threads
- `spread` allows to use all the memory and allows to reach best performances

Algo	Threads	Avg. Execution Time [ms]			Efficiency		
		False	*Close*	*Spread*	*False*	*Close*	*Spread*
Sequential	1	15743	-	-	100%	-	-
Manual	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%

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Table of Contents

▶ Conclusion

What we can learn from the experimentation and results?

- *Manual* implementation reach the best performances as we could expect
- `spread` provides better results with respect to the others bind mode
- The experimentation provides a good overview of the capabilities of *OpenMP*
- To reach the maximum performance a knowledge of the system and a ”deep” understanding of *OpenMP* is required

Problem

- The system can be accessed by multiple users $\rightarrow$ possible measurement error (reduced by multiple run)
- Does this help `false` bind to reach better results?

Future Development: the experimentation can be conducted in a more controlled system

UNIVERSITÀ DEGLI STUDI FIRENZE
Da un secolo, oltre.

Time Series Pattern Recognition with OpenMP

Thank you for listening!

Any questions?