Parallelization of the SLINK clustering algorithm

Alessio De Biasi (870288) Jonathan Gobbo (870506)

August 2022

1 Introduction

The purpose of this project is to implement the SLINK clustering algorithm and to apply parallelization techniques to improve its performance.

All the implementations of the clustering algorithms are written in `C++ 20` and they are compiled with the GCC G++ 12.1.1 (Red Hat) compiler, using the `-O3` and `-march=native` flags.

In particular, we tested the clustering algorithm implementations on two datasets:

- Accelerometer[1] (A), composed of 153'000 samples, each of which with 5 attributes. For the tests we have performed, we have selected only the last 3 of them;
- Generated (G), composed of 100'000 samples, each of which with 45 randomly generated real attributes uniformly distributed in the range from 0 to 99 included.

We will report the mean execution time of 3 executions of each implementation on the two datasets, run on a machine equipped with an AMD Ryzen 7 1700 CPU with 8 cores and 16 threads, 16 GB of RAM, and running Fedora Workstation 36.

In particular, we measured the time taken by each step of the pseudo-code (reported as S1, S2, S3 and S4) to be executed.

In all the tests we used two `std::vector` data structures to hold the computed values of π and λ.

Note that the reported total times include also some extra time, which is spent by the various implementations to perform the service operations like allocating and de-allocating the memory, or copying the iterators.

Moreover, for space reasons, in some tests we do not report the results obtained using 2 and 16 threads.

2 Implementations

2.1 Sequential

We first implemented the sequential version of the clustering algorithm. This implementation is just the translation in `C++` code of the pseudo-code reported in the slides.

The data is supplied as a `std::vector<double *>`, i.e., a vector of pointers to heap-allocated arrays, where each of these arrays holds the attributes of one data sample.

DS	S1	S2	S3	S4	Total
A	19 ms	1 m 30 s	37.330 s	20.271 s	2 m 28 s
G	19 ms	8 m 59 s	16.246 s	12.802 s	9 m 28 s

Table 1: Execution times of the sequential implementation

[1] `https://archive.ics.uci.edu/ml/datasets/Accelerometer`

2.2 Multithreaded distance computation

From the previous tests results, we saw that the computation of the distances (i.e., the step *2* of the pseudo-code) is the most expensive step of the algorithm.

Since each distance can be computed independently from the others, their computation can be easily made parallel by using the *OpenMP* library.

The supplied data has the same memory layout as before.

DS	S1	S2	S3	S4	Total
A2	10 ms	43.043 s	39.568 s	23.286 s	1 m 46 s
A4	11 ms	29.641 s	36.144 s	23.609 s	1 m 29 s
A8	11 ms	20.940 s	35.260 s	23.541 s	1 m 20 s
A12	10 ms	24.402 s	36.713 s	24.204 s	1 m 25s
A16	11 ms	27.201 s	47.172 s	26.257 s	1 m 41 s
G2	13 ms	4 m 12 s	15.228 s	13.855 s	4 m 41 s
G4	15 ms	2 m 23 s	15.555 s	14.881 s	2 m 53 s
G8	14 ms	1 m 34 s	15.305 s	14.977 s	2 m 04 s
G12	12 ms	1 m 43 s	17.569 s	15.308 s	2 m 16 s
G16	12 ms	1 m 42 s	20.592 s	15.668 s	2 m 18 s

Table 2: Execution times of the first parallel implementation

2.3 SSE and AVX in the computation of the distances

Since the attributes of each data sample to cluster are stored contiguously in memory, we decided to improve the previous solution using the SSE and AVX instructions when computing the distances.

Indeed, to compute the distance between two points, the algorithm perform the same instructions (difference and square) for every pair of attributes.

DS	S1	S2	S3	S4	Total
A2	11 ms	46.922 s	32.362 s	21.116 s	1 m 40 s
A4	11 ms	34.329 s	35.250 s	23.791 s	1 m 33 s
A8	9 ms	27.252 s	35.007 s	23.633 s	1 m 26 s
A12	10 ms	27.328 s	37.252 s	24.346 s	1 m 29 s
A16	12 ms	28.596 s	47.285 s	26.313 s	1 m 42 s
G2	14 ms	4 m 29 s	16.732 s	14.733 s	5 m 00 s
G4	15 ms	2 m 33 s	19.175 s	16.085 s	3 m 08 s
G8	13 ms	1 m 39 s	19.319 s	15.894 s	2 m 15 s
G12	13 ms	1 m 43 s	18.766 s	15.457 s	2 m 18 s
G16	14 ms	1 m 39 s	21.918 s	15.986 s	2 m 17 s

Table 3: Execution times of the implementation using SSE

DS	S1	S2	S3	S4	Total
A2	8 ms	39.106 s	38.321 s	23.127 s	1 m 41 s
A4	10 ms	24.656 s	37.981 s	23.659 s	1 m 26 s
A8	8 ms	17.814 s	37.826 s	23.772 s	1 m 19 s
A12	8 ms	18.416 s	40.067 s	24.198 s	1 m 23 s
A16	10 ms	19.824 s	52.452 s	26.367 s	1 m 39 s
G2	13 ms	4 m 34 s	14.860 s	14.236 s	5 m 03 s
G4	15 ms	2 m 37 s	15.111 s	15.457 s	3 m 07 s
G8	14 ms	1 m 42 s	15.143 s	15.660 s	2 m 13 s
G12	14 ms	1 m 49 s	17.418 s	15.516 s	2 m 22 s
G16	14 ms	1 m 44 s	20.112 s	15.970 s	2 m 20 s

Table 4: Execution times of the implementation using AVX

We expected this implementation to work faster than the previous one but, as the results show, this implementation generally works slightly slower than the previous one.

Investigating using the *perf* tool, we discovered that most of the time the CPU is stalled due to a high rate of cache misses (about 80% of all the data memory accesses).

2.4 A new data structure

Therefore, we decided to change the data structure holding the data samples to cluster in favour of a more a contiguous one.

In particular, we chose to organize the data as a unique heap-allocated array, where the data samples, as well as the attributes of each of them, are stored one after the other.

DS	S1	S2	S3	S4	Total
A4	4 ms	11.770 s	34.453 s	23.509 s	1 m 10 s
A8	4 ms	7.531 s	35.604 s	23.674 s	1 m 07 s
A12	4 ms	8.277 s	37.068 s	24.300 s	1 m 10 s
G4	9 ms	44.391 s	14.319 s	15.880 s	1 m 15 s
G8	8 ms	37.005 s	14.162 s	15.937 s	1 m 07 s
G12	8 ms	37.132 s	15.271 s	15.902 s	1 m 08 s

Table 5: Execution times of the implementation using a contiguous data structure and SSE (2 and 16 threads tests omitted)

DS	S1	S2	S3	S4	Total
A4	4 ms	12.462 s	42.709 s	23.633 s	1 m 18 s
A8	4 ms	8.315 s	42.649 s	23.517 s	1 m 14 s
A12	4 ms	8.368 s	42.744 s	23.809 s	1 m 15 s
G4	10 ms	46.830 s	14.452 s	15.770 s	1 m 17 s
G8	8 ms	38.992 s	14.454 s	16.222 s	1 m 10 s
G12	8 ms	39.331 s	15.580 s	15.969 s	1 m 11 s

Table 6: Execution times of the implementation using a contiguous data structure and AVX(2 and 16 threads tests omitted)

As we can see from these results, making the data samples contiguous in memory improves significantly the performance of the algorithm.

Indeed, when a thread loads one data sample, it is very likely that, with the same memory transaction, more data samples are brought to the cache. Consequently, when the thread accesses the following data samples, they are already in the cache.

2.5 Make the insertion of a new point parallel

After having made parallel the computation of the distances, we focused on parallelizing the next most expensive step of the algorithm, which is the insertion of a new point to the dendrogram (i.e., the step *3* of the pseudo-code).

However, there is no chance to make it parallel because, when the element at index i is analyzed, the algorithm requires that all the elements before i have already been analyzed.

Therefore, if we give to each thread an element to analyze in a round-robin fashion, then there will be always a thread working while all the other threads will be blocked waiting for the active thread to complete its work.

2.6 Make the rearrangement of the structure of the dendrogram parallel

Therefore, we decided to try to parallelize the third most expensive step of the algorithm, which is the rearrangement of the structure of the dendrogram after a new point is added (i.e., the step *4* of the pseudo-code).

Since each iteration of the `for` loop is independent from the others, making it parallel is almost trivial if we use the *OpenMP* library.

Like the previous implementation, data is supplied as a unique heap-allocated array.

For space reasons, we omitted the results of the implementation when using SSE instructions.

DS	S1	S2	S3	S4	Total
A4	4 ms	10.599 s	42.538 s	8.939 s	1 m 02 s
A8	4 ms	5.836 s	43.255 s	5.659 s	54.760 s
A12	4 ms	6.895 s	45.326 s	4.489 s	56.720 s
G4	9 ms	46.341 s	15.961 s	6.888 s	1 m 09 s
G8	8 ms	38.337 s	16.110 s	3.792 s	58.254 s
G12	8 ms	39.039 s	17.603 s	3.408 s	1 m 00 s

Table 7: Execution times of the implementation executing in parallel the computation of the distances (using threads and AVX instructions) as well as the rearrangement of the structure of the dendrogram (2 and 16 threads tests omitted)

2.7 Devising a new algorithm

The last most expensive step of the algorithm is the initialization π and λ (i.e., the step *1* of the pseudo-code). However, as the results show, even if we make it parallel, we will gain only few milliseconds.

Therefore, to improve further the performance of the algorithm, we tried devising a new one. This new algorithm builds incrementally the dendrogram like the pseudo-code does, but in a different way.

In particular, when the algorithm adds to the dendrogram a new data sample d, it first computes the closest sample c, and their distance l. Then, the algorithm traverses the path on the dendrogram from the leaf representing c to the root.

Indeed, since c is the closest sample to d, then d must be linked to one of the clusters containing c, and such a cluster can be found only on that path. In particular, this cluster is represented by the last edge ε in the path with height smaller than l. Therefore, the algorithm just needs to add a new edge linking ε to d, and then to link the new edge to the parent of ε.

Then, the algorithm continues traversing the path so to possibly update the height of the remaining edges, because adding d to a cluster may have made it closer to the cluster it joins to.

Representing the dendrogram as a binary tree using a linked structure, each edge of the path to traverse is just a pair of nodes. Since the dendrogram is not rearranged, each of these pairs can be analyzed independently. This implies that several threads may analyze such pairs in parallel requiring almost no synchronization. One of these threads will add the new edge, while all the others will possibly update the heights of the edges.

However, there are some cases where the dendrogram is actually rearranged. Indeed, suppose that the cluster $\{a, b\}$ joins the cluster $\{e, f\}$. If the data sample to add d is closer to b, then it will be added to the first cluster, and this may make the data sample e closer to the cluster $\{a, b, d\}$ more than to f.

Thus, adding the data sample d to the dendrogram may require to rearrange some of the edges. This implies that the edges in the path to traverse may change while the threads are traversing it. To avoid inconsistencies, we require much more synchronization between the threads, which may force them to work sequentially, hence reducing the performance.

Therefore, we abandoned this algorithm, and we tried to implement some micro-optimizations so to improve the performance the algorithm we have used so far.

2.8 Partial sums micro-optimization

To compute the distance between two data samples, the previous implementations using SSE and AVX instructions first load from memory 2 and 4 attributes respectively. Then, they compute the square of the pairwise attributes difference, and they accumulate the result on a stack-allocated variable, which requires to horizontally sum all the values in the extended register. All these four operations are repeated until no more attributes are left.

The first micro-optimization we have implemented accumulates the partial sums in an extended register instead of in a stack-allocated variable, in order to perform the horizontal sum only once at the end of the computation, and to avoid writing to memory each partial result.

The new implementation combines this micro-optimization with all the parallelization strategies we have implemented so far. The data is still supplied as a unique heap-allocated array.

Also in this case, for space reasons, we omitted the results of the implementation when using SSE instructions.

DS	S1	S2	S3	S4	Total
A4	4 ms	10.174 s	41.862 s	8.887 s	1 m 01 s
A8	4 ms	5.820 s	42.804 s	5.592 s	54.227 s
A12	4 ms	6.868 s	45.697 s	4.526 s	57.102 s
G4	11 ms	41.854 s	17.229 s	5.854 s	1 m 05 s
G8	10 ms	37.064 s	16.846 s	3.590 s	57.515 s
G12	9 ms	37.005 s	17.897 s	3.414 s	58.330 s

Table 8: Execution times of the first micro-optimized implementation using AVX (2 and 16 threads tests omitted)

2.9 Square roots micro-optimization

The last micro-optimization we have implemented extends the previous one by using the square of the distances instead of the distances themselves. The computation of the square roots is performed only at end of the execution of the algorithm and only on the computed values of λ.

Indeed, ignoring possible rounding errors, this solution works because the following property holds for non-negative values (as the distances are):

$$\sqrt{n^2} \geq \sqrt{m^2} \iff n^2 \geq m^2$$

DS	S1	S2	S3	S4	S5	Total
A2	4 ms	10.961 s	37.824 s	12.933 s	190 μs	1 m 02 s
A4	4 ms	6.641 s	41.826 s	8.989 s	112 μs	57.466 s
A8	4 ms	3.903 s	42.693 s	5.592 s	58 μs	52.199 s
A12	5 ms	4.603 s	45.701 s	4.549 s	69 μs	54.864 s
A16	5 ms	4.067 s	51.38 s	4.222 s	52 μs	59.340 s
G2	9 ms	1 m 5 s	15.592 s	9.995 s	113 μs	1 m 30 s
G4	9 ms	43.109 s	16.769 s	6.865 s	66 μs	1 m 07 s
G8	9 ms	36.077 s	16.618 s	3.834 s	35 μs	56.544 s
G12	9 ms	36.747 s	17.697 s	3.430 s	48 μs	57.890 s
G16	5 ms	36.383 s	20.554 s	3.100 s	38 μs	1 m 01 s

Table 9: Execution times of the second micro-optimized implementation using SSE

DS	S1	S2	S3	S4	S5	Total
A2	4 ms	11.439 s	37.881 s	13.146 s	216 μs	1 m 02 s
A4	4 ms	7.106 s	42.571 s	8.888 s	115 μs	58.576 s
A8	4 ms	4.147 s	42.281 s	5.607 s	61 μs	52.046 s
A12	5 ms	5.271 s	44.500 s	4.595 s	71 μs	54.376 s
A16	5 ms	4.386 s	50.834 s	4.346 s	52 μs	59.578 s
G2	9 ms	53.895 s	16.303 s	10.436 s	124 μs	1 m 21 s
G4	10 ms	41.312 s	16.961 s	6.453 s	67 μs	1 m 05 s
G8	8 ms	37.104 s	16.829 s	3.715 s	35 μs	57.663 s
G12	8 ms	37.015 s	17.697 s	3.430 s	47 μs	58.427 s
G16	9 ms	37.781 s	20.467 s	3.212 s	38 μs	1 m 01 s

Table 10: Execution times of the second micro-optimized implementation using AVX

In the previous tables, S5 reports the time taken to compute the square roots only.

2.10 A new data structure for the sequential implementation

To compute the speed-ups of all the previous parallel implementations, we need to also execute the sequential implementation with the data supplied as a unique heap-allocated array.

DS	S1	S2	S3	S4	Total
A	4 ms	27.978 s	28.764 s	20.663 s	1 m 17 s
G	16 ms	2 m 24 s	16.307 s	13.705 s	2 m 54 s

Table 11: Execution times of the sequential implementation using a contiguous data structure to store the data samples

As we can see, this memory layout drastically improves the performance of the sequential implementation, even without using any parallelization technique.

Indeed, storing all the data samples contiguously in memory greatly improves the use of the cache, as we observed using the *perf* tool.

2.11 Data layouts comparison

Finally, we report the results of the tests we have performed on the last parallel implementation of the clustering algorithm, but supplying the data as a `std::vector<double *>` (i.e., a vector of heap allocated arrays each holding the attributes of one data sample) like in the first parallel implementation.

In the following tables, S5 reports the time taken to compute the square roots only.

DS	S1	S2	S3	S4	S5	Total
A2	9 ms	42.708 s	41.007 s	17.475 s	170 μs	1 m 41 s
A4	6 ms	34.021 s	42.132 s	9.814 s	97 μs	1 m 26 s
A8	6 ms	29.379 s	42.149 s	6.243 s	51 μs	1 m 18 s
A12	6 ms	29.705 s	43.549 s	5.491 s	72 μs	1 m 18 s
A16	7 ms	30.903 s	48.340 s	5.206 s	54 μs	1 m 24 s
G2	9 ms	2 m 52 s	17.576 s	10.547 s	118 μs	3 m 20 s
G4	9 ms	1 m 47 s	19.144 s	6.986 s	66 μs	2 m 13 s
G8	9 ms	1 m 24 s	18.963 s	4.404 s	35 μs	1 m 48 s
G12	10 ms	1 m 31 s	20.374 s	4.026 s	49 μs	1 m 56 s
G16	10 ms	1 m 31 s	22.532 s	3.765 s	39 μs	1 m 57 s

Table 12: Execution times of the second micro-optimized implementation using SSE and `std::vector<double*>`

DS	S1	S2	S3	S4	S5	Total
A2	7 ms	33.418 s	39.571 s	15.797 s	181 μs	1 m 29 s
A4	7 ms	20.032 s	40.853 s	9.820 s	95 μs	1 m 11 s
A8	7 ms	13.387 s	42.220 s	5.761 s	61 μs	1 m 01 s
A12	8 ms	16.225 s	44.615 s	4.966 s	68 μs	1 m 05 s
A16	9 ms	19.212 s	50.916 s	4.934 s	52 μs	1 m 15 s
G2	14 ms	2 m 27 s	16.368 s	10.641 s	126 μs	2 m 54 s
G4	13 ms	1 m 35 s	17.435 s	7.007 s	69 μs	1 m 59 s
G8	13 ms	1 m 22 s	17.152 s	4.432 s	35 μs	1 m 44 s
G12	13 ms	1 m 30 s	20.023 s	4.109 s	53 μs	1 m 54 s
G16	14 ms	1 m 29 s	21.576 s	3.801 s	42 μs	1 m 55 s

Table 13: Execution times of the second micro-optimized implementation using AVX and `std::vector<double*>`

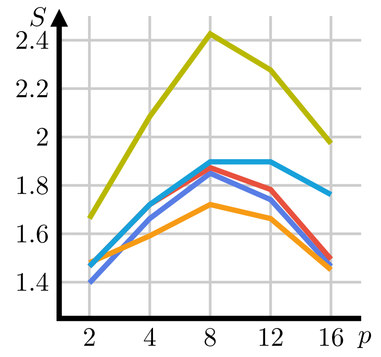
2.12 Speed-up

When storing the data samples in a `std::vector<double*>`, we reached a speed-up of 2.43 for the *Accelerometer* dataset and 5.46 for the *Generated* one, both executing the AVX parallel implementation described in 2.11 with 8 threads.

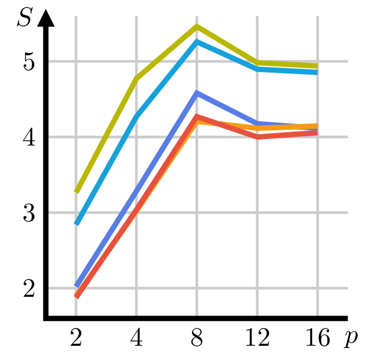
Instead, when storing the data samples in a unique array, we obtained a speed-up of 1.48 for the *Accelerometer* dataset exeuting the AVX parallel implementation described in 2.9 with 8 threads, while we obtained a speed-up of 3.08 for the *Generated* dataset executing the SSE parallel implementation described in 2.9 with 8 threads.

All the parallel techniques we have applied achieve better speed-ups when the data samples are composed of many attributes, since we exploit the parallelization of the computation of the distances, which is the most expensive step of the algorithm. Moreover, we get the highest speed-ups when using 8 threads, since we are exploiting exactly all the available CPUs.

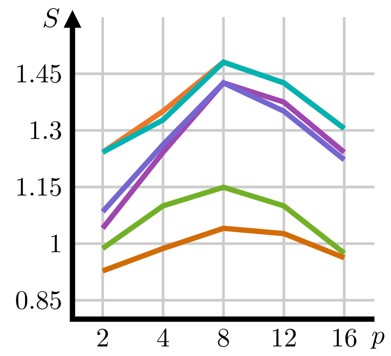
In conclusion, among all the scheduling policies offered by the *OpenMP* library, we have found that the `static` one is the most effective in parallelizing the `for` loops.



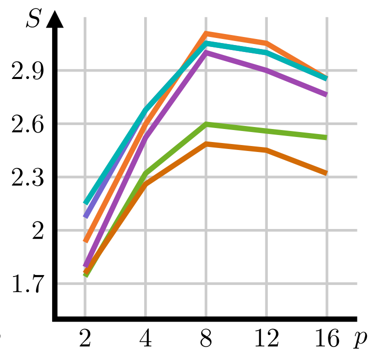
Accelerometer dataset with data layout `std::vector<double *>`



Generated dataset with data layout `std::vector<double *>`



Accelerometer dataset with unique array data layout



Generated dataset with unique array data layout

🟦 2.2 🟧 2.3 SSE 🟥 2.3 AVX 🟩 2.4 SSE 🟫 2.4 AVX 🟪 2.6

🟪 2.8 🟧 2.9 SSE 🟦 2.9 AVX 🟦 2.11 SSE 🟨 2.11 AVX

Speed-ups varying dataset and data samples layout

3 How to use the library

We organized all the sequential and parallel implementations of the clustering algorithm into two different classes, namely the `SequentialClustering` and `ParallelClustering` classes.

All these implementations are able to process data samples with any number of attributes, as long it is the same for all the samples.

The `main-sample.cpp` file in the `test/src` directory contains an example illustrating how to use the library.

3.1 `ParallelClustering` class

This class, defined in `ParallelClustering.h` header file, offers several utility methods and constants allowing the programmer to easily deal with aligned memory. In particular, the `computeSseDimension` and `computeAvxDimension` static methods allow to compute the number of `double` elements each data sample must be composed of, taking into account also possible paddings.

However, the `cluster` static method is the most important one offered by this class. It clusters the given data samples using the specified distance computation algorithm (see 3.1.3 for the list of available algorithms), which is specified as a template argument so to compile only the code needed to execute the requested algorithm. The method allows also to specify the number of threads to use in each of the steps that can be executed in parallel, namely:

1. The computation of the distance between two data samples;
2. The operations needed to rearrange the structure of the dendrogram after a new point is added (i.e., the step *4* of the pseudo-code);
3. The computation of the square roots of the distances stored in λ.

Note that the parallelization of such steps can be enabled or disabled at compile time by using the template parameters of the `ParallelClustering` class.

3.1.1 Data samples memory layout

The `cluster` static method allows a certain flexibility in the memory layout of the data samples. In particular, the method accepts two main organizations:

1. A contiguous region of memory, where the data samples, as well as the attributes of each sample, are stored one after the other.
   The method accepts either an iterator iterating over that region of memory, or directly the data structure holding it. In this case, the `cluster` method takes care of extracting the iterator by calling the `begin` method or, if present, the `cbegin` one;
2. A data layout organized in two levels.
   The first level is merely a container that holds the data samples to cluster (or the iterators iterating over them).
   This container is required to be randomly accessible, but it is not required to be contiguous in memory.
   Each element of the second level, instead, is a data structure (or an iterator iterating over it) holding a contiguous region of memory that stores the attributes of one data sample.
   For both levels, the `cluster` method accepts either the iterator over the data structure, or the data structure itself. In this last case, the method takes care of calling the `begin` method, or the `cbegin` one if present, to obtain an iterator iterating over the data structure.

3.1.2 π and λ memory layouts

The `cluster` static method allows also a certain flexibility in the memory layout of π and λ.

In particular, each of them can be any randomly accessible data structure, or an iterator iterating over it. In the former case, the method takes care of calling `begin` method to retrieve the random access iterator iterating over the specified data structure.

Both the data structures holding the values of π and λ must be properly sized so that all the values computed by the `cluster` method can fit into them.

3.1.3 Distance computation algorithms

The `cluster` static method allows the programmer to use different algorithms to compute the distance between two data samples. In particular:

- `CLASSICAL` computes the Euclidean distances without using any SIMD instruction;
- `SSE` and `AVX` compute the distances using SSE and AVX instructions respectively;
- `SSE_OPTIMIZED` and `AVX_OPTIMIZED` compute the distances using SSE and AVX instructions respectively, and they keep the partial sums in an extended register instead of storing them into a stack-allocated variable (see 2.8);
- `SSE_OPTIMIZED_NO_SQUARE_ROOT` and `AVX_OPTIMIZED_NO_SQUARE_ROOT`, which act like `SSE_OPTIMIZED` and `AVX_OPTIMIZED` respectively, but they compute the square of the distances, without applying the square root operation (see 2.9).

All these constants are defined in the `DistanceComputers` enumeration, defined in the `DistanceComputers.h` header file.

3.2 `SequentialClustering` class

This class, defined in the `SequentialClustering.h` header file, provides only one method, the `cluster` static method, which executes the sequential implementation of the clustering algorithm on the specified data samples.

3.2.1 Data samples memory layout

The `cluster` static method supports the same data layouts as the `cluster` method of the parallel implementations (see 3.1.1).

In addition, if the specified data structure (or iterator) is organized in two levels, then this method does not require the first level to be randomly accessible, but only to be at least input iterable (or an input iterator).

3.2.2 π and λ memory layouts

The `cluster` static method supports exactly the same data layouts for both π and λ as the `cluster` method of the parallel implementations (see 3.1.2).

3.3 `Timer` class

If the `TIMERS` macro is defined, both the sequential and parallel implementations of the clustering algorithm will measure, using a timer, the time taken to execute each step of the algorithm.

In particular, each step is measured by a different timer with a specific identifier, where `0` measures the time taken to execute all the service operations, while `1` to `4` measure the time taken to execute the corresponding steps of the pseudo-code.

In addition, only for the parallel implementations run with the distance computation algorithms `SSE_OPTIMIZED_NO_SQUARE_ROOT` and `AVX_OPTIMIZED_NO_SQUARE_ROOT`, the timer with identifier `5` measures the time taken to compute the square root of each value stored in λ.

The `Timer` class, defined in the `Timer.h` header file, offers the `print` and `printTotal` static methods to print to the console the times measured by each of these timers.