Parallel Computing - 2024/25
Midterm assignment
K-means clustering

Matteo Spataro
matteo.spataro@edu.unifi.it

Abstract

K-means clustering is a supervised classification algorithm that has been extensively studied and used over the years. This assignment aims to develop a sequential and a parallel version of it, so that they can be compared. After presenting the K-means algorithm and the dataset used, we will describe the implementation details, focusing on the parallelization achieved by OpenMP. We will describe the experiments carried out and we will analyze the results obtained from the two versions of the program.

1. Introduction

An algorithm under the category of partitioning clustering, K-means is based on the definition of the prototype in terms of centroid, usually representing the midpoint of a cluster. The characteristics that make it one of the most widely used algorithms in science and industry are its simplicity of implementation, its computational speed, and the fact that it always converges to a solution.

The dataset used for the experiment is entitled "Big Five Personality Test": is a 1 million answers to 50 personality items and technical information, publicly available on the Kaggle platform at the following address: `https://www.kaggle.com/datasets/tunguz/big-five-personality-test`. The file is distributed in CSV format and has a size of 93,7 MB. It collects over a million responses provided by subjects who took the "Big Five" test, one of the most widely used psychological models for personality assessment. Therefore, applying the K-means algorithm to this dataset, we will obtain K clusters, each of which identifies a personality trait.

Each column represents one of the 50 questions in the test, which could be answered by giving a value from 0 to 5. To ensure higher data quality, we manually removed some outliers, deleting all rows in which all elements had values of 0, 1, or 5. Such responses are highly anomalous and potentially indicative of insincere or random behavior on the part of participants, which would compromise the effectiveness of the clustering process.

2. Implementation

The language used for this program is **C++**. We chose to use the `std::vector` data structure because using `std::array` requires knowing the maximum size of the dataset (rows and columns) in advance, which would have limited the use of the program to a single dataset known in advance.

Since our goal is to implement an efficient version of the k-means algorithm, in order to make something simpler and easier, we made assumptions about the type of dataset. To be compatible with this program, the dataset must be contained in a file with a CSV extension, and the first row must contain the headers. In addition, each field must be convertible to a real value, otherwise the entire row will be discarded.

At last, K-means clustering algorithms allow you to choose how to initialize the centroids, but to employ a deterministic behavior, we implemented an initialization method where the first K data points are designated as initial centroids. While deterministic and computationally efficient, this approach is sensitive to input order and may yield suboptimal clustering if initial points are non-representative.

2.1. Sequential

Algorithm 1 describes the pseudocode of the sequential version of the K-means algorithm. The implementation implicitly tolerates empty clusters: centroids of clusters with *counts[c]* equal to 0 remain unchanged, avoiding division-by-zero errors. This may perpetuate suboptimal centroids but ensures numerical stability.

To quantify the proximity of an element to a centroid, we used the "squared Euclidean distance", defined in Equation 1, since the points in the dataset used belong to a Euclidean space. This choice avoids redundant square

Algorithm 1 Sequential K-means

```
1: procedure KMEANS(data, K, ϵ, maxIterations)
2:     Input:
       data: point matrix (N × D)
       K: number of clusters
       ϵ: minimum allowed centroid shift
       maxIterations: maximum iterations
3:     Output:
       centroids: list of the final K centroids
4:     Initialize centroids[1...K] ← data[1...K]
5:     iteration ← 0
6:     repeat
7:         Create arrays:
       sums[1...K][1...D] ← 0
       counts[1...K] ← 0
8:         for each point p in data do
9:             Find index c* of the centroid closest to p
10:            Assign p to cluster c*
11:            sums[c*] += p
12:            counts[c*] += 1
13:        maxChange ← 0
14:        for c = 1 to K do
15:            oldCentroid ← centroids[c]
16:            if counts[c] > 0 then
17:                centroids[c] ← sums[c] / counts[c]
18:            Δ ← distance(oldCentroid, centroids[c])
19:            if Δ > maxChange then
20:                maxChange ← Δ
21:        iteration += 1
22:    until maxChange ≤ ϵ or iteration ≥ maxIterations
23:    return centroids
```

root computations and preserves pairwise proximity relationships.

$$d^2(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{n} (x_i - y_i)^2 \tag{1}$$

Analyzing the computational complexity, for N points of dimension D and K clusters, it emerges that at each iteration of the cycle there are $\mathcal{O}(N \cdot K \cdot D)$ assignments and $\mathcal{O}(K \cdot D)$ updates of the centroids. However, since K and D are much smaller than N, the time complexity is linear in N and the number of iterations performed.

2.2. Parallel

To parallelize the code, we used the **OpenMP** directives. The parallel implementation of the k-means algorithm differs from the sequential implementation in the strategy adopted for calculating the assignment of points to clusters and updating centroids. These changes are introduced in order to exploit the parallelism of multi-core systems through the use of OpenMP.

Almost all parallel regions are enclosed in the same code region given to the `#pragma omp parallel` directive, with the aim of reducing the overhead derived from the creation and synchronization of threads.

In addition, the concept of "first touch", in which memory is allocated closer to the thread in which it is first touched, has been parallelized to distribute the data across all nodes. This was done with the local variable `localFlatSums`.

2.2.1 Parallelization of the assignment phase

In sequential code, for each point in the dataset, the distance to each centroid is calculated, assigning the point to the nearest cluster and immediately updating the global sums and counts structures. This approach introduces a computational bottleneck that is eliminated in the parallel version by creating local copies of the `flatSums` (a linear version of the "sums" matrix) and counts structures for each thread created. The aim is to avoid write conflicts on shared variables by calculating distances independently and updating the local copies of each thread.

At the end of the assignment phase, the local copies are merged into the global structures using atomic operations, thanks to the `#pragma omp atomic` directive, which guarantees correctness and eliminates the contention of a shared update.

The `nowait` clause was used for assignments in order to eliminate the implicit barrier at the end of the `#pragma omp for` directive, because there is no dependency between threads. In fact, each thread initializes its own structures and then processes its own points, so it can continue as soon as it finishes the data on which to operate.

2.2.2 Parallelization of centroid updating

In the next phase, the update of all the centroids is performed in parallel using a `#pragma omp parallel for reduction(max:maxChange)`. Each thread independently updates a subset of the centroids, calculating the average of the assigned points and the squared Euclidean distance from the previous centroid position. The use of the `reduction` directive allows computing in parallel the maximum change between old and new centroids, keeping the stopping criterion unchanged from the sequential version.

3. Profiling

Profiling is a technique used to analyze the efficiency of a program. It allows us to identify the slowest parts of the code and those that could be improved through parallelization. There are several tools available, and we have

chosen to use Intel VTune, a general-purpose optimization tool.

First, we performed "Performance snapshots" for both the sequential and parallel versions of the k-means program, collecting the significant results in the Table 1.

Looking at the results, we can see how the parallel version of the code makes more effective use of double-precision floating point operations (DP GFLOPS), a sign that it makes better use of the CPU cores to perform numerical calculations simultaneously. Furthermore, parallelization with OpenMP allows for better utilization of the microarchitecture, i.e. increasing the percentage of pipeline slots used and the percentage of instructions successfully completed ("Retiring" metric).

Furthermore, the “Back-End Bound” metric confirms that the parallel version reduces the CPU bottleneck, because it reduces the percentage of time when the CPU has instructions to execute but has to wait for resources: this goes from 49.7% in the sequential version to 25.9% in the parallel version, almost halving the number of times this occurs.

It is also worth mentioning the results of the “Average DRAM Bandwidth” metric, which is the average amount of data read and written by the DRAM per second. In this field, the parallel version of the program manages to speed up reading and writing almost 10 times more than its sequential counterpart, a sign of intense and effective use of memory.

Next, we performed the hotspot analysis of the parallel version to identify which part of the code has the greatest impact on program execution time. This allowed us to visualize the histogram of actual CPU utilization, which we have shown in Figure 1. These results are obtained by running the parallel program with 12 threads and with the K parameter of the k-means algorithm set to 5.

The analysis reveals an effective CPU utilization of **63,9%** and that simultaneous usage of the 12 cores in this CPU persists for **2,5** seconds, i.e. almost the entire execution time of the program. On the other hand, short intervals suffer from underutilized execution, likely at the beginning and end of parallel regions and during `atomic` flushes.

In addition, Intel VTune's hotspot profiling enabled an analysis that led to modify of the parallel code. By experimenting with the number of threads, we were able to confirm that inserting the `schedule(dynamic)` clause in the `#pragma omp for` directives allows for much greater exploitation of the 12 logical cores of the CPU in question. These results are shown in Figure 2: above are the results of using the `schedule(static)` clause, while below are the results of the `schedule(dynamic)` clause.

In conclusion, our VTune analysis confirms that the parallel k-means implementation effectively leverages hardware threads, but is limited by `atomic` operations and reduction at the end of each iteration.

4. Experiments and results

The experimentation carried out in this project aims to quantify the time benefit of using parallelization through OpenMP directives. To this end, a sufficiently large dataset called “Big Five Personality Test” was chosen on which to run the programs. A rough removal of outliers and null rows was applied to the dataset, resulting in a size of 93,7 MB and containing 1.010.579 rows of 50 columns.

To compile the program we used "x64 Native Tools Command Prompt for VS 2022" with the instruction

```
"cl /std:c++20 /openmp:llvm /O2
Kmeans.cpp /Fe:kmeans.exe /EHsc"
```

where "`/O2`" tells the compiler to apply a series of complex optimizations to make the executable code as fast as possible: it has been observed that it reduces the execution time of the entire program by a factor of ~ 6.

The experiment we conducted aimed to vary the number of threads involved in parallelization, comparing each execution time with the results obtained from the sequential version of the k-means algorithm. The number of threads was varied from 2 to 24 with step size of two, while the number K of clusters was left unchanged, with K equal to 5.

The results obtained are shown in Figure 3, where each execution time is given by the average of 10 identical executions. These results show that the clustering time decreases rapidly with the increase of threads in execution, demonstrating how well the calculation can be parallelized. The optimum is achieved with 12 threads, which coincides with the number of logical cores of the CPU on which the experiment was performed.

As we can see more clearly from the zoom of the curve shown in Figure 4, there are no further benefits beyond that number of threads because the available cores become saturated and the overhead also increases.

In line with the results obtained so far, Figure 5 shows the speedup curve of the parallel k-means as the number of available threads varies. The speedup of the parallel program with p threads is given by the formula

$$S_p = \frac{t_s}{t_p} \tag{2}$$

where t_s is the time in seconds obtained from the sequential version of the program, while t_p is the time in seconds obtained from the parallel program with p threads.

Metric	Sequential k-means	Parallel k-means
Elapsed Time	30,522 s	8,467 s
IPC	1,525	1,179
DP GFLOPS	2,361	17,953
Logical Core Utilization	9,3 % (1,114/12)	89,2 % (10,708/12)
Physical Core Utilization	18,3 % (1,097/6)	91,1 % (5,467/6)
Microarchitecture Usage	37,3 %	56,2 %
Retiring	37,3 %	56,2 %
Front-End Bound	6,3 %	12,7 %
Bad Speculation	6,7 %	5,2 %
Back-End Bound	49,7 %	25,9 %
Memory Bound	20,5 %	8,0 %
Core Bound	29,2 %	17,8 %
Memory Breakdown	20,5 %	8,0 %
Cache Bound	13,9 %	17,6 %
DRAM Bound	7,2 %	6,1 %
Average DRAM Bandwidth	1,877 GB/s	11,883 GB/s

Table 1. Intel VTune performance metrics for sequential and parallel k-means implementations.

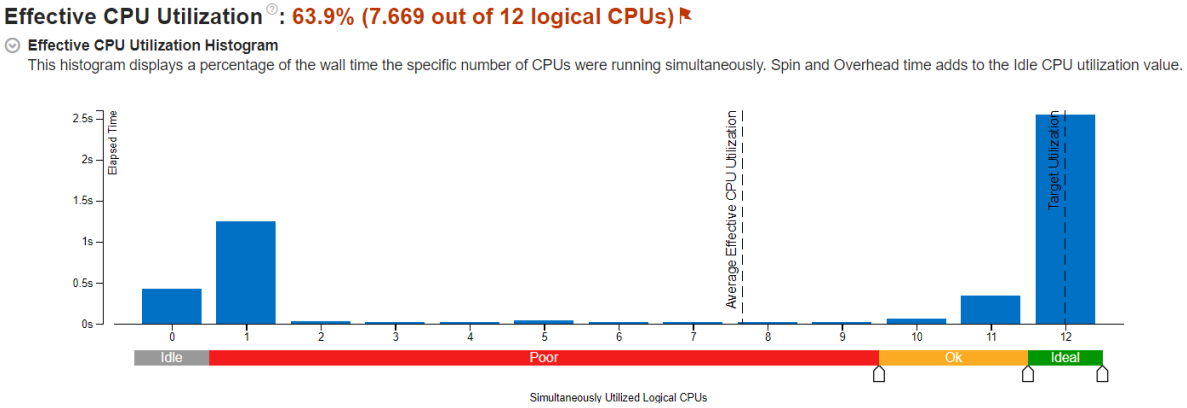

Figure 1. Effective CPU utilization histogram of parallel k-means.

5. Conclusions

The implementation of a parallel version of the K-means algorithm using OpenMP showed that parallelization is an effective strategy for significantly reducing execution times on large datasets. The approach adopted minimized conflicts between threads through local data structures and the use of the `reduction` directive, lowering the overhead from synchronization and exploiting the capabilities of the CPU microarchitecture.

Analysis using Intel VTune confirmed the improvement in performance metrics, highlighting more efficient memory usage and increased utilization of the CPU's physical and logical cores. In particular, it was observed that the use of the `schedule(dynamic)` clause further increase scalability and efficiency, allowing the program to better adapt to the characteristics of the underlying hardware.

The experimental results showed that the parallel algorithm scales effectively up to the number of available logical cores, beyond which the benefits diminish due to resource saturation and increased management overhead. The observed speedup shows behavior in line with theoretical expectations, confirming the effectiveness of parallelism applied to the most computationally expensive phases of the K-means algorithm.

Ultimately, this study demonstrates that, even though it is a relatively simple clustering algorithm, the conscious adoption of parallelization techniques allows for significant performance improvements, making it even more effective for the classification of large datasets in real-world contexts.

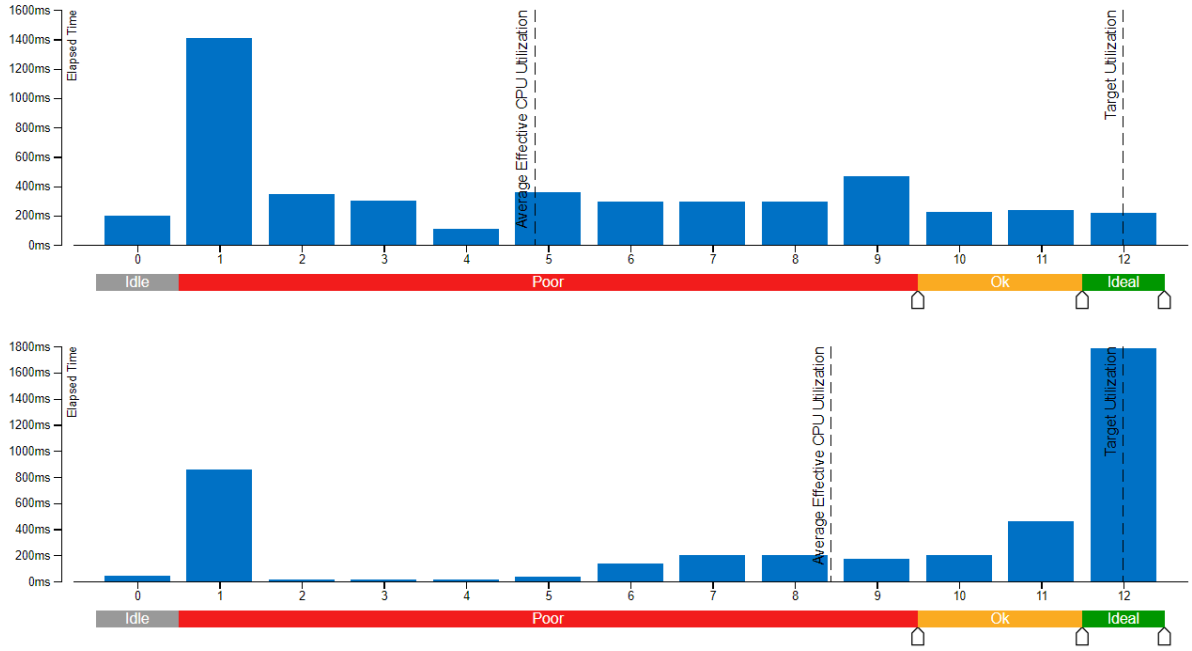

Figure 2. Above, the CPU histogram of using `schedule(static)`. Below, the CPU histogram of using `schedule(dynamic)`.

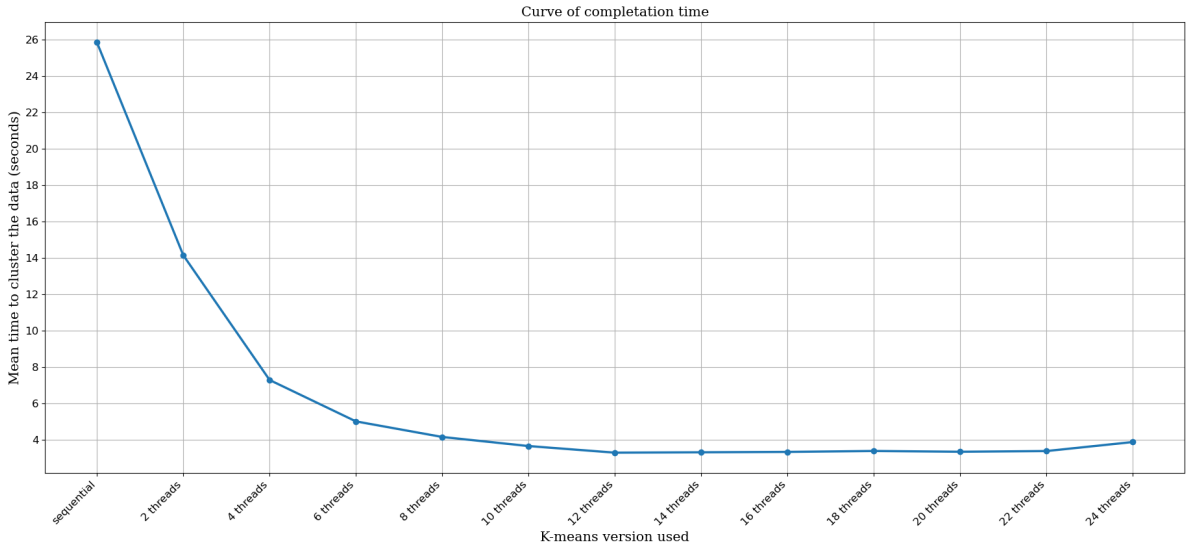

Figure 3. Curve of the mean clustering time varying the number of threads.

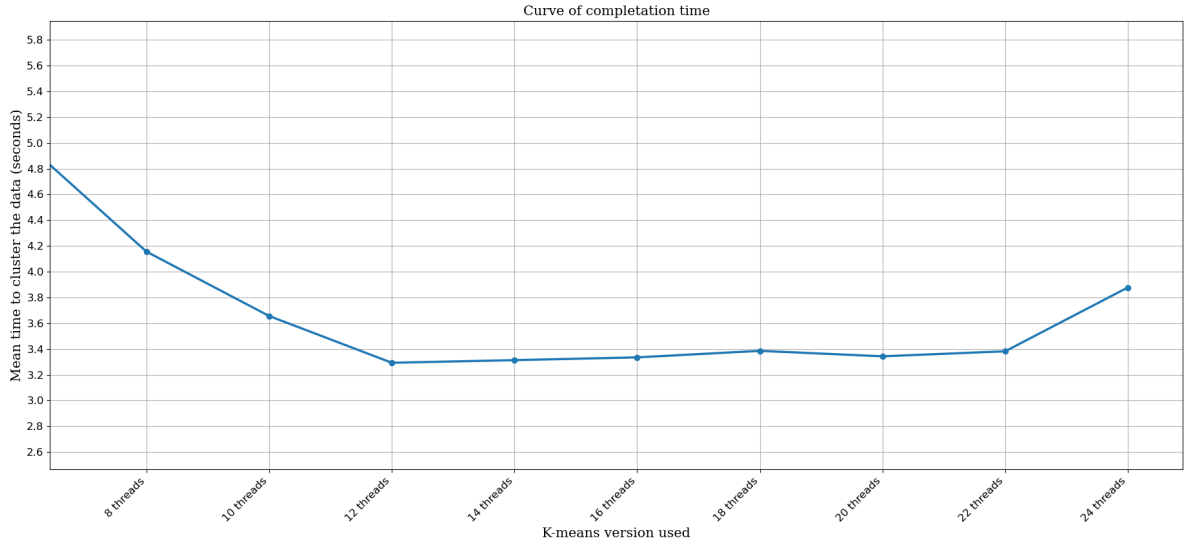

Figure 4. Zoom of the mean clustering time: 12 threads gives the best result.

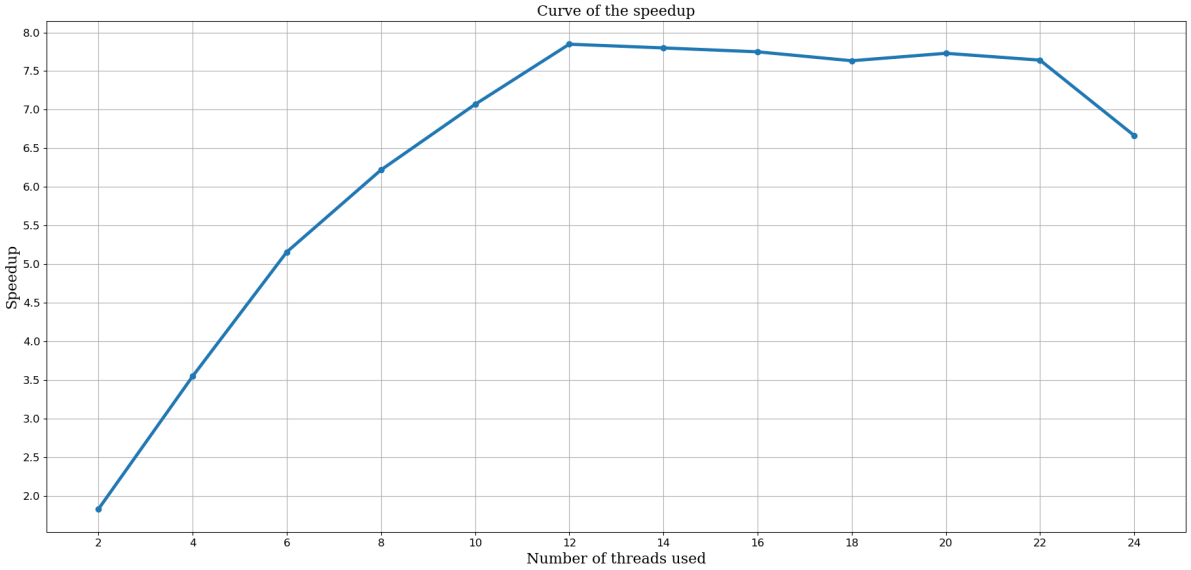

Figure 5. Speedup curve of the parallel k-means varying the number of threads.