


# $k$ -means Is All You Need

Leonardo Biason (2045751)   Alessandro Romania (2046144)   Davide De Blasio (2082600)

**Abstract**—The  $k$ -means algorithm is a well known clustering algorithm, which is often used in unsupervised learning settings. However, the algorithm requires to perform multiple times the same operation on the data, and it can greatly benefit from a parallel implementation, so that to maximize the throughput and reduce computation times. With this project, we propose some possible implementations, based on some libraries that are considered to be the de-facto standard when it comes to writing multithreaded or parallel code, and we will discuss also the results of such implementations

Sapienza, ACSAI, Multicore Programming

 Check our repository [on GitHub](#)  
ElBi21/PSEM-kmeans

## I. INTRODUCTION

When talking about clustering and unsupervised learning, it's quite common to hear about the  $k$ -means algorithm, and for good reasons: it allows to efficiently cluster a dataset of  $d$  dimensions, and it employs the notion of convergence in order to do so. This, computationally speaking, means to repeat some operations over and over again until some stopping conditions are met.

The algorithm is not perfect though, and presents some issues:

- 1) the algorithm is fast in clustering, but we cannot be certain that it clusters *well*;
- 2) the algorithm doesn't work with non-linear clusters;
- 3) the initialization can make a great impact in the final result.

Many people prefer to use other clustering methods, such as the fitting of Gaussian Mixture Models. Albeit not being perfect,  $k$ -means still works well in simple, linear clusters. For the sake of this project, we are going to consider a vanilla  $k$ -means algorithm with Lloyd's initialization (the first  $k$  centroids will be selected randomly).

### A. Algorithm structure

The  $k$ -means algorithm can be described with the following pseudocode, where  $X$  is the set of data points,  $C = \{\mu_1, \mu_2, \dots, \mu_k\}$  is the set of centroids and  $Y$  is the set of assignments:

---

#### Algorithm 1: $k$ -means (Lloyd's initialization)

---

```

// Initialize the centroids
1 for  $k$  in  $[1, |C|]$  do
2    $\mu_k \leftarrow$  a random location in the input space
3 end

4 while convergence hasn't been reached do
   // Assign each point to a cluster
5   for  $i$  in  $[1, |X|]$  do
6      $y_i \leftarrow \operatorname{argmin}_k (\|\mu_k - x_i\|)$ 
7   end

   // Compute the new position of each centroid
8   for  $k$  in  $[1, |C|]$  do
9      $\mu_k \leftarrow \operatorname{MEAN}(\{x_n : z_n = k\})$ 
10  end
11 end

// Return the centroids
12 return  $Y$ 

```

---

The algorithm consists of 4 main blocks:

- the **initialization block**, where all the centroids will receive a starting, random position (as per Lloyd's method);
- the **assignment block**, where the Euclidean distance between a point and all centroids is computed, for all centroids. The point will be assigned to a cluster depending on the following operation:

$$\operatorname{argmin}_k (\|\mu_k - x_i\|)$$

- the **update block**, where the position of the centroids is updated, and the new position of a centroid  $\mu_k$  is equal to the mean of all the data points positions belonging to cluster  $k$

### B. Sequential Code Bottlenecks

For implementing the  $k$ -means algorithm, we will base all the codebase upon the project made from pro-

fessors Diego García-Álvarez and Arturo Gonzalez-Escribano from the University of Valladolid. The code shown in this subsection is taken from their project, although slightly adapted for giving enough context in the code snippets.

We described before the overall structure of the  $k$ -means algorithm: we will now proceed to examine its two main bottlenecks. As we can see from Algorithm 1, we have two main blocks that may cause performance issues: the **assignment block** and the **update block**.

The first **for** block in the **initialization step** does not represent a major bottleneck, since it just needs to assign a random location to each of the  $K$  centroids. It can be parallelized, but it won't help as much as parallelizing the two steps mentioned before.

The second **for** block represents the **assignment step**, which is, unlike the initialization step, computationally expensive: for each point, the algorithm will have to compute the euclidean distance (here onwards denotes as  $\ell_2$ ) between said point and all centroids  $\mu_k \in C$ , and select the lowest distance. This will determine the cluster of the point. In a C program, this may be accomplished with the following piece of code:

```

1 int cluster;
2 // For each point...
3 for(i = 0; i < points_number; i++) {
4     class = 1;
5     minDist = FLT_MAX;
6     // For each cluster...
7     for(j = 0; j < K; j++) {
8         // Compute the distance
9         dist = l2_norm(&data[i*samples], &
            centroids[j*samples], samples);
10
11         // If the distance is the lowest so far,
            replace it
12         if(dist < minDist) {
13             minDist = dist;
14             class = j+1;
15         }
16     }
17
18     // If the class is different from before, add
            a change to the counter
19     if(classMap[i] != class) {
20         changes++;
21     }
22
23     classMap[i]=class;
24 }
```

Notice the presence of the two nested **for** loops: sequentially, they would take a time of  $O(|X| \cdot |C|)$ , which may be optimized just by taking a simple single instruction multiple data approach (indeed,

with  $m > 1$  different processes or threads, it would take a time of  $O\left(\frac{|X| \cdot |C|}{m}\right)$  each, which is already better than the first option).

The third **for** loop represents the update step, which also is computationally expensive: we would need to perform the mean of the coordinates of all the points belonging to a cluster  $\mu_k$ . This implies that all the coordinates of the points must be first summed, and then averaged on the number of points being classified to  $\mu_k$ . An implementation in the C language would look like the following:

```

1 // For each point...
2 for (i = 0; i < lines; i++) {
3     point_class = classMap[i];
4     // Add 1 to the points classified for class k
5     pointsPerClass[point_class - 1] += 1;
6
7     // For each dimension...
8     for(j = 0; j < samples; j++) {
9         // ...add it to a table for summing and
            averaging
10         auxCentroids[(point_class - 1) * samples
            + j] += data[i * samples + j];
11     }
12 }
13
14 for (i = 0; i < K; i++) {
15     for (j = 0; j < samples; j++) {
16         // Average all dimensions
17         auxCentroids[i * samples + j] /=
            pointsPerClass[i];
18     }
19 }
20
21 maxDist = FLT_MIN;
22 for (i = 0; i < K; i++) {
23     // Compute the moving distance, as a
            convergence check
24     distCentroids[i] = euclideanDistance(&
            centroids[i * samples], &auxCentroids[i *
            samples], samples);
25     if (distCentroids[i] > maxDist) {
26         maxDist = distCentroids[i];
27     }
28 }
```

## II. PARALLELIZING WITH MPI

### III. PARALLELIZING WITH OPENMP

To enhance the performance of the  $k$ -means algorithm, we employed OpenMP for parallelization. OpenMP is a widely-used API for multi-platform shared-memory parallel programming. Our primary goal was to leverage multiple CPU cores to expedite the computationally intensive parts of the algorithm: the **cluster assignment** and **centroid update** steps. The number of threads can be optionally specified as a parameter; if not provided, a default value (i.e. 8 threads) is used for parallel execution.

## A. OpenMP Implementation Approach

**1. Cluster Assignment:** The cluster assignment step is inherently parallelizable since each data point can be processed independently. We utilized the `#pragma omp parallel for` directive to distribute the loop iterations across multiple threads.

- **Reduction Clause:** The `reduction(+ : changes)` clause safely aggregates the total number of cluster reassignments across threads. This avoids race conditions by creating thread-private copies of `changes` that are combined after the loop.
- **Dynamic Scheduling:** The `schedule(dynamic, 16)` clause improves load balancing by allowing threads to dynamically claim "chunks" of 16 iterations.

**2. Centroid Update:** The centroid update step involves two parallel phases: accumulating cluster sums and normalizing centroids. Here, we addressed potential race conditions through privatization and synchronization:

- **Privatization with Local Buffers:** Each thread maintains private copies of `local_pointsPerClass` (count of points assigned to the cluster) and `local_auxCentroids` (cumulative sum of data points for each cluster). This eliminates races during local accumulation, as threads operate on isolated data.
- **Critical Section for Global Aggregation:** After local accumulation, a `#pragma omp critical` region safely merges thread-local results into global `pointsPerClass` and `auxCentroids`. While critical sections incur synchronization overhead, they are used sparingly here once per thread to minimize contention.
- **Parallel Mean Calculation:** The final centroid normalization loop (`#pragma omp parallel for`) parallelizes the division by `pointsPerClass`, avoiding race conditions since each cluster is processed independently.

An alternative approach that we experimented with uses the reduction feature to eliminate the need for explicit synchronization.

```

1 #pragma omp parallel for reduction(+ :
  pointsPerClass[ : K], auxCentroids[ : K *
    samples])
2     for (i = 0; i < lines; i++)
3     {
4         int class = classMap[i] - 1;
5         pointsPerClass[class]++;
6         for (int j = 0; j < samples; j++)
7         {

```

```

8             auxCentroids[class * samples + j]
9             += data[i * samples + j];
10        }

```

The reduction clause automates the process of merging thread-local results into shared variables. By reducing both `pointsPerClass` and `auxCentroids`, the need for a critical section is removed, potentially lowering synchronization overhead. However, we chose the original approach with explicit privatization and critical regions to demonstrate proficiency with OpenMP synchronization and race condition management.

The maximum centroid displacement (`maxDist`) is computed serially. While this loop could be parallelized using `#pragma omp parallel for reduction(max : maxDist)`, we retained the serial approach for simplicity, as its computational cost is negligible compared to other steps and because the number of clusters  $k$  is typically small, the overhead associated with parallelization outweighs the potential performance gains.

## B. Solutions Implemented

- **Reduction for Scalars:** The `changes` variable uses a reduction clause, which is more efficient than atomic operations for scalar summation. OpenMP handles private copies and post-loop merging automatically.
- **Privatization and Critical Sections:** For array/matrix updates (e.g., `auxCentroids`), thread-local buffers reduce the need for fine-grained synchronization. The critical section ensures safe aggregation with minimal overhead, as it is invoked only once per thread.
- **Atomic Operations vs. Critical Regions:** While atomic operations (e.g., `#pragma omp atomic`) could replace the critical section for scalar increments (`pointsPerClass`), they would be inefficient for matrix additions (`auxCentroids`) due to repeated fine-grained locking. Privatization strikes a better balance between correctness and performance.

## C. Performance Considerations

- **Critical Section Overhead:** The single critical section per thread during centroid aggregation has negligible cost compared to the computational work, as merging local buffers is a minor operation.
- **Memory Efficiency:** Privatizing `local_auxCentroids` per thread increases

memory usage proportionally to the number of threads. However, this is manageable for moderate thread counts and cluster sizes.

- **Dynamic Scheduling Trade-off:** While dynamic scheduling improves load balancing, it introduces overhead for chunk management. A chunk size of 16 was empirically chosen to balance parallelism and scheduling latency.

#### *D. Conclusion*

The OpenMP-based parallelization of the  $k$ -means algorithm effectively mitigates race conditions through a combination of reduction clauses, thread-local privatization, and strategic critical sections. These strategies not only ensured the correctness of the computations but also optimized performance by minimizing synchronization overhead.

### IV. PARALLELIZING WITH CUDA

#### V. INTERLACING MULTI-PROCESSING WITH MULTI-THREADING

##### *A. MPI and OpenMP*

##### *B. MPI and CUDA*

### VI. PERFORMANCE ANALYSIS

### VII. CONCLUSIONS