# $k$-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, $C = \{\mu_1, \mu_2, ..., \mu_k\}$ is the set of centroids and $Y$ is the set of assignments:

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

```
// Initialize the centroids
for $k$ in $[1, |C|]$ do
    $\mu_k \leftarrow$ a random location in the input space
end
while convergence hasn't been reached do
    // Assign each point to a cluster
    for $i$ in $[1, |X|]$ do
        $y_i \leftarrow \text{argmin}_k (\|\mu_k - x_i\|)$
    end
    // Compute the new position of each centroid
    for $k$ in $[1, |C|]$ do
        $\mu_k \leftarrow \text{MEAN}(\{x_n : z_n = k\})$
    end
end
// Return the centroids
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:
$$\underset{k}{\arg\min}\left(\|\mu_k - x_i\|\right)$$
- 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 professors 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:

```c
int cluster;
// For each point...
for(i = 0; i < points_number; i++) {
    class = 1;
    minDist = FLT_MAX;
    // For each cluster...
    for(j = 0; j < K; j++) {
        // Compute the distance
        dist = l2_norm(&data[i*samples], &centroids[j*samples], samples);

        // If the distance is the lowest so far, replace it
        if(dist < minDist) {
            minDist = dist;
            class = j+1;
        }
    }

    // If the class is different from before, add a change to the counter
    if(classMap[i] != class) {
        changes++;
    }

    classMap[i]=class;
}
```

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:

```c
// For each point...
for (i = 0; i < lines; i++) {
    point_class = classMap[i];
    // Add 1 to the points classified for class k
    pointsPerClass[point_class - 1] += 1;

    // For each dimension...
    for(j = 0; j < samples; j++) {
        // ...add it to a table for summing and averaging
        auxCentroids[(point_class - 1) * samples + j] += data[i * samples + j];
    }
}

for (i = 0; i < K; i++) {
    for (j = 0; j < samples; j++) {
        // Average all dimensions
        auxCentroids[i * samples + j] /= pointsPerClass[i];
    }
}

maxDist = FLT_MIN;
for (i = 0; i < K; i++) {
    // Compute the moving distance, as a convergence check
    distCentroids[i] = euclideanDistance(&centroids[i * samples], &auxCentroids[i * samples], samples);
    if (distCentroids[i] > maxDist) {
        maxDist = distCentroids[i];
    }
}
```

## II. Parallelizing with MPI

## III. Parallelizing with OpenMP

## IV. Parallelizing with CUDA

## V. Interlacing Multi-processing with Multi-threading

### A. MPI and OpenMP

### B. MPI and CUDA

## VI. Performance Analysis

## VII. Conclusions