

# $k$ -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

## 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 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:

```

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], &
10            centroids[j*samples], samples);
11
12         // If the distance is the lowest so far,
13         // replace it
14         if (dist < minDist) {
15             minDist = dist;
16             class = j+1;
17         }
18     }
19     // If the class is different from before, add
20     // a change to the counter
21     if (classMap[i] != class) {
22         changes++;
23     }
24     classMap[i]=class;
25 }
```

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
10        averaging
11        auxCentroids[(point_class - 1) * samples
12        + j] += data[i * samples + j];
13    }
14 }
15
16 for (i = 0; i < K; i++) {
17     for (j = 0; j < samples; j++) {
18         // Average all dimensions
19         auxCentroids[i * samples + j] /=
20         pointsPerClass[i];
21     }
22 }
23
24 maxDist = FLT_MIN;
25 for (i = 0; i < K; i++) {
26     // Compute the moving distance, as a
27     // convergence check
28     distCentroids[i] = euclideanDistance(&
29        centroids[i * samples], &auxCentroids[i *
30        samples], samples);
31     if (distCentroids[i] > maxDist) {
32         maxDist = distCentroids[i];
33     }
34 }
```

## II. PARALLELIZING WITH MPI

The Message Passing Interface (MPI) is a standardized, portable framework that enables parallel computation across distributed memory systems. These systems, typically clusters of interconnected computers, do not share a common memory. Each process operates on its local memory and must explicitly communicate with others by sending and receiving messages. MPI achieves parallelism through process-based communication, distributing both data and tasks among multiple processes.

Achieving efficient parallel performance in MPI programs requires attention to two critical factors:

- 1) Balancing the workload across processes;
- 2) Minimizing the frequency of message passing.

Both factors are crucial for scalable and high-performance parallel applications. We here show how

these optimizations are implemented in this  $k$ -means clustering algorithm. By default, the  $k$ -means clustering program follows a Globally Parallel, Locally Sequential (GPLS) model. This means that the critical initialization and termination tasks are handled by rank 0, while the looping part of the algorithm is entirely parallelized.

#### A. Design of the program

##### Initialization Phase

The initialization phase is where workload balancing decisions are made. Rank 0 reads the input data from a file (as only rank 0 has access to the file). The input data (points) is broadcast to all processes using `MPI_Scatterv`. The points remain fixed throughout the execution, enabling a single communication step to distribute data efficiently. The Centroids are randomly generated by rank 0. These centroids are then broadcasted to all processes.

##### Point assignment

Each process works on its local points with its local `classMap`. This assignment step is fully parallel, as each process handles its portion of the points independently. Assignment changes are accumulated locally and reduced globally, by using the `MPI_Allreduce` collective for checking, at the end of each iteration, the termination conditions.

##### Centroids update

Once points are assigned, each process computes its local partial sums for each centroid and the number of points in each cluster. These partial results are aggregated across all processes using `MPI_Allreduce`, ensuring globally consistent centroid values with fast communication. Each process then updates its locally assigned centroids based on the reduced global values, distributing the workload evenly among processes. The maximum distance between old and updated centroids is calculated locally and reduced globally using `MPI_Allreduce` to check for termination conditions. After these checks, an `MPI_Allgatherv` operation gathers all local centroids, preparing them for the next iteration.

##### Termination phase

In the termination phase, local point assignments (the output of the  $k$ -means algorithm) are gathered by the rank 0 process using `MPI_Gather`. Finally, all processes free their allocated memory, and the algorithm terminates.

### III. PARALLELIZING WITH OPENMP

As a mean of enhancing the performance of the  $k$ -means algorithm, we made use of the multi-thread library OpenMP. 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. We now show how we approached the parallelization of the program, and the advantages of taking certain design decisions.

#### A. OpenMP Implementation Approach

**1. Cluster Assignment:** the cluster assignment step is inherently parallelizable since each data point can be processed independently. We made use of the `#pragma omp parallel for` directive to distribute the loop iterations across multiple threads. Said directive has been accompanied by the following clauses:

- `reduction(+ : changes):` the **reduction 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;
- `schedule(dynamic, 16):` the **dynamic scheduling** 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: gathering cluster sums and averaging the centroids coordinates. Here, we addressed potential race conditions through privatization and synchronization:

- **Privatization with Local Buffers:** Each thread maintains private copies of two buffers, namely `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 in order 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 the `#pragma omp parallel for reduction(max : maxDist)` directive, 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 (i.e., `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 (i.e., `#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 per thread the `local_auxCentroids` buffer 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.

## IV. PARALLELIZING WITH CUDA

In recent years, we have seen how GPUs play crucial roles when it comes to parallelizing a program with multiple threads. Indeed, the model proposed by NVIDIA for its platforms (namely, the Single-Thread Multiple-Data model) turned out to be very efficient, by allowing notable speed-ups

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

So far, we implemented various solutions for our programs, which employed either multi-processing or multi-threading parallelism techniques, without using both approaches at the same time. However, these techniques are not mutually exclusive, and can be mixed together in order to achieve better performances. In fact, in high performance computing tasks, multi-process techniques are used within clusters to connect nodes and coordinate them, while multi-threading are used for performing computations, given the directives of the master process(es).

In this section we will show how the two multi-threading approaches that we previously considered (namely, OpenMP and CUDA) can be combined with the famous multi-processing library MPI.

### A. MPI and OpenMP

The hybrid MPI + OpenMP implementation of the  $k$ -means clustering algorithm leverages both distributed and shared memory parallelism to enhance performance and scalability. Our parallelization strategy can be described with three main key points:

#### Two-Level Parallelism

- MPI distributes data across processes (coarse-grained parallelism) with `MPI_Scatterv`, a vector scattering collective, while OpenMP parallelizes loops within each process (allowing for a much fine-grained parallelism);
- Thread safety: all the processes are initialized with `MPI_THREAD_FUNNELED`, so that to restrict MPI calls to the main thread.

#### Asynchronous Communication

The non-blocking reduce operation `MPI_Ireduce` aggregates cluster reassignment counts while centroid updates proceed, allowing to overlap communication and computation.

#### Hybrid Reductions

- OpenMP sets thread-local buffers in order to avoid intra-node races. More specifically, the `local_pointsPerClass` and the `local_auxCentroids` buffers;
- Global aggregations via `MPI_Allreduce` ensures consistent centroid states across nodes.

```
1 #pragma omp parallel // Thread-local
   accumulation
2 {
3     int *local_pointsPerClass = calloc(K, sizeof(
        int));
4     float *local_auxCentroids = (float *)calloc(K
        * samples, sizeof(float));
5     // ...
6 }
7
8 // Global sync
9 MPI_Allreduce(MPI_IN_PLACE, pointsPerClass, K,
    MPI_INT, MPI_SUM, MPI_COMM_WORLD);
10 MPI_Allreduce(MPI_IN_PLACE, auxCentroids, K *
    samples, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

A major concern while designing this application was also ensuring that an efficient **data distribution** system was in place. We show, in the following two key points, how we tackled this problem:

#### Balanced Workloads:

- The `MPI_Scatterv` collective allows to split data into chunks, adjusted via two parameters: `sendcounts` and `displs`. Said collective is capable of handling uneven divisions, ensuring balanced workloads even when the total data points (`lines`) are not evenly divisible by the number of processes.
- Each process computes locally the updates for a subset of centroids. These partial updates are then gathered from all processes and re-distributed to ensure that every process has the complete and updated set of centroids for the next iteration, using `MPI_Allgatherv`. We report here the full collective call:

```
1 MPI_Allgatherv(local_centroids, local_k *
    samples, MPI_FLOAT, centroids,
    centroid_sendcounts, centroid_displs,
    MPI_FLOAT, MPI_COMM_WORLD);
```

#### Final Result Aggregation

After convergence, the `MPI_Gatherv` collective reconstructs the global cluster assignments on the root process. Each process sends its own `local_classMap` (containing classifications for its local data subset), and the root process assembles them into `classMap` using the precomputed `recvcounts` and `rdispls` offsets. Below the code snippet, which shows how the `MPI_Gatherv` call was assembled, we give more precise definitions regarding the two offsets aforementioned:

```
1 // Gather local classifications into root's
   classMap
2 MPI_Gatherv(local_classMap, local_lines, MPI_INT,
    classMap, recvcounts, rdispls, MPI_INT, 0,
    MPI_COMM_WORLD);
```

- `recvcounts`: array specifying the number of elements received from each process (matches `sendcounts` from initial scatter);
- `rdispls`: offsets in `classMap` where each process' data is stored.

These offsets ensures efficient reconstruction of the global result without requiring intermediate storage on worker processes.

#### Memory Efficiency:

- Each process stores only its local data (`local_points`) and centroid subset (`local_centroids`).
- Thread-local buffers minimize shared memory contention without atomic operations.

How did we approach **synchronization and termination**?

**Convergence Check:**

- OpenMP's `reduction(max:local_maxDist)` computes per-process centroid displacements.
- `MPI_Reduce` finds the global maximum displacement for termination decisions.

**Termination Broadcast**

Root process evaluates conditions and broadcasts `terminate flag` via `MPI_Bcast`, ensuring synchronized loop exits.

*B. MPI and CUDA*

VI. PERFORMANCE ANALYSIS

VII. CONCLUSIONS