


k-means is (really) all you need

Leonardo Biason (2045751) Alessandro Romania (2046144)

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)

```

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
5   // Assign each point to a cluster
6   for  $i$  in  $[1, |X|]$  do
7      $y_i \leftarrow \operatorname{argmin}_k (\|\mu_k - x_i\|)$ 
8   end

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

13 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 μ_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 ℓ_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 μ_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 μ_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_Iallreduce` 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 PTHREAD

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 CUDA platforms (namely, the Single-Thread Multiple-Data model) turned out to be very efficient, by allowing notable speed-ups and augmentation of the throughput. Here follows an explanation of how we decided to design a possible CUDA implementation.

A. Designing the parallel structure

As we have shown in Algorithm 1, the k -means algorithm can be logically split into two steps: the assignment step and the update step. However, while logically this division may sound reasonable, it is not appropriate for the STMD model that NVIDIA has at the core of its devices. This is because of the needed data for the two steps: the assignment step needs to work only with the data points, but can be parallelized by splitting the data into multiple parts; the update step instead needs to work with both all the data and all the centroids, so parallelizing the step as a whole becomes quite hard.

A simpler approach would be to split the update step in two parts, one that uses a fraction of the points, and the other that uses a fraction of the centroids. This would create in total three parallelization steps. However, the assignment step and the first part of the update step can be merged together, since they both need to work with part of the data. This is the reason why we decided to implement the program with two kernels, called respectively `step_1_kernel` and `step_2_kernel`, where the first kernel can be considered as points-based and the second kernel as centroids-based.

B. Program parameters and custom `atomicMax`

We decided to organize the threads in two dimensional blocks of 32×32 threads, and the blocks are instead organized in a one dimensional grid. The size of the grid depends on the called kernel: usually, we have that $|X| \gg |C|$, so it's pointless for CUDA

to reserve a grid of threads for the centroids that has the same size of the grid of the points. Indeed, the grids' single dimensions are dynamic, and are computed as follows:

$$\text{points_grid_size} = \frac{|X|}{32 \times 32} + 1$$

$$\text{centroids_grid_size} = \frac{k}{32 \times 32} + 1$$

The program makes also use of a custom function, called `custom_atomic_max()`. This function has been implemented because it allows us to perform an `atomicMax()`-like function for float numbers, which would not be normally possible with the built-it CUDA function. We here show the function as a whole:

```
1 __device__ float custom_atomic_max(float*
  value_address, float val) {
2   int* address_as_int = (int*) value_address;
3   int old = *address_as_int, assumed;
4   do {
5       assumed = old;
6       old = atomicCAS(address_as_int, assumed,
  __float_as_int(fmaxf(val, __int_as_float(
  assumed))));
7   } while (assumed != old);
8   return __int_as_float(old);
9 }
```

The idea of the function is that CUDA tries continuously to perform an `atomicCAS()` operation, which in turns performs atomically the following check:

```
old_value == to_compare ? new_value : old_value
```

The function will exit only when the value in the specified address is equal to the one that the program expects to be there, before performing the atomic transaction. This is important, so that to avoid that the function overwrites any unintentional value.

C. Analysis of the kernels

As we mentioned previously, we are making use of two kernels: `step_1_kernel` and `step_2_kernel`. In both kernels, all operations that act on the global memory are performed atomically, which avoid potential race conditions. We here show how both kernels work, alongside their code snippets.

The first kernel is called on all the data, and each point is assigned to a thread. First, a preliminary check is performed, to make sure that each thread is assigned to a valid point. While the `if` statement

may seem like a possible cause of warp divergence, it doesn't actually impact that much. Indeed, we would only discard part of the final block of kernels, while all the previous blocks are fully used.

Then, each thread will compute the ℓ_2 norm of each point for all centroids, and will store the class to which each point will be assigned into the `class_int` variable. If a change from the previous class assignment is detected, the number of changes will increase. After that, each thread will proceed to sum the coordinates of its assigned point into the auxiliary centroids matrix. The coordinates will then be averaged in the second kernel.

```
1 __global__ void step_1_kernel(float* data, float*
  centroids, int* points_per_class, float*
  aux_centroids, int* class_map, int*
  changes_return) {
2
3   // Compute global thread index
4   int thread_index = (blockIdx.y * gridDim.x *
  blockDim.x * blockDim.y) + (blockIdx.x *
  blockDim.x * blockDim.y) + (threadIdx.y *
  blockDim.x) + threadIdx.x;
5
6   if (thread_index < gpu_n) {
7       int class_int = class_map[thread_index];
8       float min_dist = FLT_MAX;
9
10      // For each centroid...
11      for (int centroid = 0; centroid < gpu_K;
  centroid++) {
12          float distance = 0.0f;
13
14          // Compute the euclidean distance
15          euclideanDistance(&data[thread_index *
  gpu_d], &centroids[centroid * gpu_d], gpu_d,
  &distance);
16
17          // If distance is smaller, replace the
  distance and assign new class
18          if (distance < min_dist) {
19              min_dist = distance;
20              class_int = centroid + 1;
21          }
22      }
23
24      // If the class is different, add one change
  and write new class
25      if (class_map[thread_index] != class_int) {
26          atomicAdd(changes_return, 1);
27      }
28
29      // Map the value to the class map
30      class_map[thread_index] = class_int;
31
32      int class_assignment = class_map[thread_index];
33      int point_index = class_assignment - 1;
34
35      atomicAdd(&(points_per_class[point_index]),
  1);
36
37      for (int dim = 0; dim < gpu_d; dim++) {
38          int index = point_index * gpu_d + dim;
39          atomicAdd(&aux_centroids[index], data[
  thread_index * gpu_d + dim]);
40      }
41  }}
```

The second kernel performs the same check as `step_1_kernel` on each thread, to ensure that all threads are mapped to a valid centroid. After that, all threads will compute, for each dimension of the centroids, the average coordinate. Once computed, each thread will then perform the ℓ_2 norm between the previous centroid and the new one, so that to compute the `max_distance` variable, needed for convergence. Finally, via the use of the `custom_atomic_max()` function, the gratest distance is stored in memory.

```

1 __global__ void step_2_kernel(float*
  centroids_table, float* centroids, int*
  points_per_class, float* max_distance) {
2 // Index of the thread
3 int thread_index = (blockIdx.y * gridDim.x *
  blockDim.x * blockDim.y) + (blockIdx.x *
  blockDim.x * blockDim.y) +
4   (threadIdx.y * blockDim.x
5   + threadIdx.x);
6
7 if (thread_index < gpu_K) {
8   float distance;
9   // For each dimension...
10  for (int d = 0; d < gpu_d; d++) {
11    centroids_table[thread_index * gpu_d + d]
12    /= (float) points_per_class[thread_index];
13    // Compute Euclidean distance ( $\ell_2$  norm)
14    // to check for maximum distance
15    distance += pow((centroids[thread_index *
16    gpu_d + d] - centroids_table[thread_index *
17    gpu_d + d]), 2);
18  }
19  // Perform sqrt of distance
20  // distance = sqrt(distance);
21  // Exchange atomically, disregard old value
22  custom_atomic_max(max_distance, distance);
23 }

```

After executing the second kernel, the program continues repeating in loop the two kernels until one of the convergence conditions is met.

V. INTERLACING MULTI-PROCESSING WITH MULTI-THREADING WITH MPI + OPENMP

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 multi-threading approaches can be combined with the famous multi-processing library MPI.

With MPI, we noticed that by augmenting the number of processes, the speed-up continued to grow. However, as expected, after a given number of processes we don't have anymore a linear scaling of the performances. With the input file `input100D.inp`, which has $|X| = 10^4$, we start observing a non-linear increase already at $p = 5$, while with input file `input100D2.inp`, which has $|X| = 10^5$, we stop increasing linearly after $p = 6$. We can see this behaviour in Figure 1. This is because the overhead of exchanging data starts to become tangible, affecting negatively the performances. The parameters given to the program were the following:

<program> <input> 40 5000 1 000.1 <output>

For the sole exception that with `input2D2.inp` the number of centroids was 10, as $|X_{2D}| = 20$.

A. OpenMP

B. CUDA

Regarding the CUDA implementation, we show in Figure 2 its performances, compared to the ones of the sequential version. For this comparison, two different test inputs have been used, so that to better show the capabilities of this version. We can notice how the sequential version takes, in most of the cases, higher times for finishing executing. The only exception to this is with the input `input2D2.inp`, where $|X|$ is incredibly small. In that case, there is a great overhead, given by the memory movements needed by CUDA.

However, the difference can be seen with files containing a greater amount of data: the sequential version on input file `input100D2.inp` takes large amount of data, with both of the tests. On Table II are listed the average timings for both versions on 30 samples each, and as we can see, the advantage of using CUDA against the sequential counterpart becomes evident when the number of data gets over the order of 10^4 (indeed, the input file `input20D.inp` contains 10,000 points). At the end, CUDA is capable of reaching speedups of roughly $15.41\times$ over the sequential version:

$$\frac{17.353409s}{1.171914s} + \frac{74.396411s}{4.647139s} \approx 15.41$$

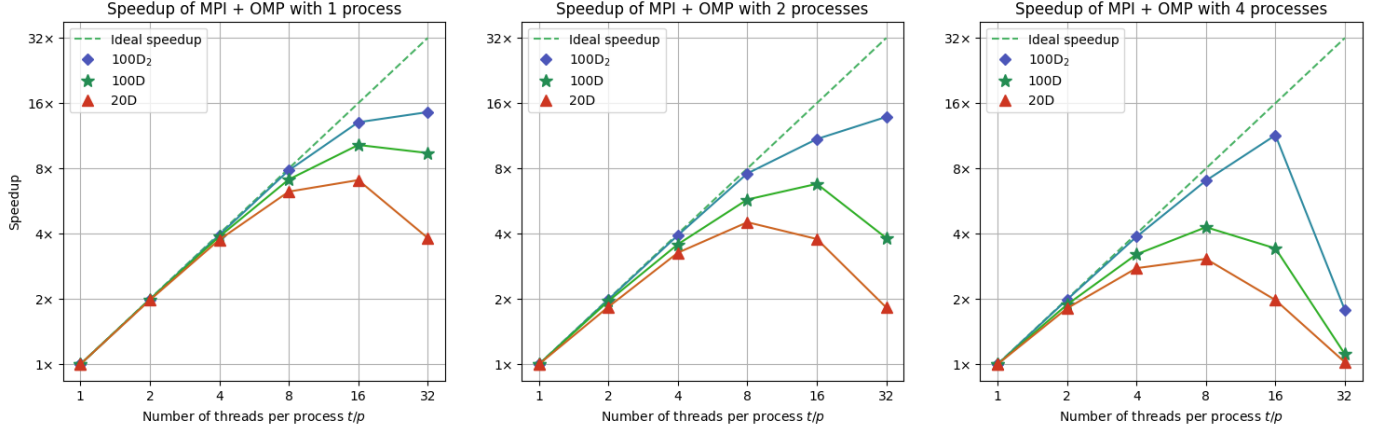


TABLE I
AVERAGE EXECUTION TIMES OF MPI + OMP

Process	Threads	Input files (in dimensions)		
		20D	100D	100D ₂
1	1			

A: <program> <input> 40 5000 1 0.0001 <output> **B:** <program> <input> 30 500 0.1 0.1 <output>

TABLE II
AVERAGE EXECUTION TIMES OF CUDA AND SEQUENTIAL VERSIONS

Test	Platform	Input files (in dimensions)					
		2D	2D ₂	10D	20D	100D	100D ₂
A	Sequential	0.008275s	0.000017s	0.002507s	0.106904s	0.432722s	17.353409s
	CUDA	0.000674s	0.000129s	0.001720s	0.012662s	0.055614s	1.171914s
B	Sequential	0.021466s	0.000051s	0.006098s	0.232680s	0.812164s	74.396411s
	CUDA	0.002292s	0.000151s	0.002440s	0.087071s	0.055614s	4.647139s

A: <program> <input> 40 5000 1 0.0001 <output> **B:** <program> <input> 30 500 0.1 0.1 <output>

C. MPI + OpenMP

D. MPI + PThreads

VI. CONCLUSIONS