

k -means Is All You Need

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 performing multiple times the same operation on the data, and it can greatly benefit from a parallel implementation, so as to maximize the throughput and reduce computation times. With this project, I 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 I will also discuss the results of such implementations.



Check my repository on [GitHub](#)
DavideDDB23/Parallel-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 algorithm)

```

// 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 criterion is not met do
5   // Assign each point to its closest cluster
6   for  $i$  in  $[1, |X|]$  do
7      $y_i \leftarrow \operatorname{argmin}_k (\|\mu_k - x_i\|)$ 
8   end
9   // Update the centroids based on current assignments
10  for  $k$  in  $[1, |C|]$  do
11     $\mu_k \leftarrow \operatorname{MEAN}(\{x_n : y_n = k\})$ 
12  end
13 // Return the assignments
14 return  $Y$ 

```

The algorithm consists of 3 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. 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, I base all the codebase upon the project made by 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.

I described before the overall structure of the k -means algorithm: I 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 denoted 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 changes = 0;
2 // For each point...
3 for (i = 0; i < lines; i++) {
4     class = 1;
5     minDist = FLT_MAX;
6     // For each centroid...
7     for (j = 0; j < K; j++) {
8         // Compute the distance
9         dist = euclideanDistance(&data[i*samples
10                                ], &centroids[j*samples], samples);
11         // If the distance is smallest so far,
12         // update minDist and the class of the point
13         if (dist < minDist) {
14             minDist = dist;
15             class = j+1;
16         }
17     }
18     // If the class changed, increment changes
19     // counter
20     if (classMap[i] != class) {
21         changes++;
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 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     int class = classMap[i];
4     // Increment points classified for class k
5     pointsPerClass[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[(class - 1) * samples + j]
12        += data[i * samples + j];
13    }
14 }
15 // For each centroid...
16 for (i = 0; i < K; i++) {
17     // Average all dimensions
18     for (j = 0; j < samples; j++) {
19         auxCentroids[i * samples + j] /=
20         pointsPerClass[i];
21     }
22 }
23 maxDist = FLT_MIN;
24 for (i = 0; i < K; i++) {
25     // Compute the moving distance, as a
26     // convergence check
27     distCentroids[i] = euclideanDistance(&
28     centroids[i * samples], &auxCentroids[i *
29     samples], samples);
30     if (distCentroids[i] > maxDist) {
31         maxDist = distCentroids[i];
32     }
33 }
```

II. PERFORMANCE OPTIMIZATIONS

To improve performance, several optimizations have been applied. When computing the Euclidean distance, the square root is not required for comparison purposes. Recall that for two distances d_1 and d_2 , the following holds:

$$d_1 < d_2 \Leftrightarrow d_1^2 < d_2^2,$$

so comparing squared distances yields the same result as comparing the actual distances. Thus, instead of computing:

$$\text{dist} = \sqrt{\sum_{i=1}^{\text{samples}} (\text{point}[i] - \text{center}[i])^2},$$

I calculate the squared distances using the `fmaf` function as follows:

```
1 dist = 0;
2 for(i = 0; i < samples; i++) {
3     dist = fmaf(point[i]-center[i], point[i]-
4               center[i], dist);
5 }
```

This avoids the relatively expensive square root computation. Additionally, using `fmaf` computes $(a * b) + c$ in a single, efficient step, reducing rounding errors and ensuring consistent results across all implementations.

Furthermore, since I no longer compute the square root, the convergence check must be adjusted accordingly. In the implementation, one of the convergence conditions is determined by measuring the maximum movement of the centroids between iterations. The convergence threshold is defined as `maxThreshold` (measured in the same units as the Euclidean distance). Therefore, the corresponding squared threshold used for comparing squared distances is:

$$\text{threshold}_{\text{squared}} = \text{maxThreshold}^2.$$

This ensures that the convergence condition remains equivalent to checking whether the actual maximum movement among all centroids is below `maxThreshold`.

In addition, instead of using loops in functions `zeroFloatMatrix` and `zeroIntArray`, I leverage the standard library function `memset` for more efficient memory initialization, as shown in the following code snippet:

```
1 memset(matrix, 0, rows * columns * sizeof(float));
2 memset(array, 0, size * sizeof(int));
```

These optimizations, though low-level, can result in significant performance gains in the k -means algorithm.

To develop the hybrid MPI+OpenMP implementation, I first implemented the individual MPI and OpenMP versions separately. This approach allowed me to gain a deeper understanding of their respective design choices and optimizations. In the following paragraphs, I illustrate the implementation details of both versions.

III. 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. Here I 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).

Since only rank 0 initially knows the values of `lines` (data points) and `samples` (dimensions), these must be communicated to all processes:

```
1 MPI_Bcast(&lines, 1, MPI_INT, 0, MPI_COMM_WORLD);
2 MPI_Bcast(&samples, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

User-supplied parameters (K for the number of clusters, `maxIterations`, `minChanges` and `maxThreshold`) are parsed by all ranks.

Still on rank 0, K centroids are randomly generated, these initial centroids are then broadcast so that every process starts the iterative phase with the same centroid values:

```
1 MPI_Bcast(centroids, K * samples, MPI_FLOAT, 0,
2         MPI_COMM_WORLD);
```

Since the data points remain *fixed* throughout the algorithm, it is efficient to *scatter* them just once at the beginning, using a single communication step. I use `MPI_Scatterv` to handle potentially *uneven* distributions, where the number of data points does not evenly divide by the number of processes. First, I compute `sendcounts[i]`, the number of *elements* to send to each rank i , and `displs[i]`, the displacement in the data array from where to pick these elements:

```

1 int remainder = lines % size;
2 int sum = 0;
3 for (int i = 0; i < size; ++i)
4 {
5     sendcounts[i] = (lines / size) * samples;
6     // The first 'remainder' processes get an
7     // extra point
8     if (i < remainder)
9         sendcounts[i] += samples;
10    displs[i] = sum;
11    sum += sendcounts[i];
12 }
13 // Each process then can calculate its number of
14 // assigned points
15 int local_lines = sendcounts[rank] / samples;
16 // Allocate memory for local data points and
17 // their corresponding class assignments.
18 float *local_data = (float *)calloc(local_lines *
19                                     samples, sizeof(float));
20 int *local_classMap = (int *)calloc(local_lines,
21                                     sizeof(int));
22 // Scatter data from rank 0 to all processes
23 MPI_Scatterv(data, sendcounts, displs, MPI_FLOAT,
24             local_data, sendcounts[rank], MPI_FLOAT, 0,
25             MPI_COMM_WORLD);

```

This design ensures that the program maintains a balanced workload even when the total number of points does not divide evenly by the number of processes, minimizing load imbalance. Any leftover points (remainder) are distributed one by one to the first few processes to avoid idle time, ensuring every process contributes as effectively as possible to the computation.

Point assignment

Each process works on its local subset of data points (local_lines) and maintains its own local class map (local_classmap). This assignment step is fully parallel, as each process independently determines the nearest centroid for each of its assigned points. Assignment changes are accumulated locally and then reduced globally, by using the MPI_Iallreduce collective for checking, at the end of each iteration, the termination conditions. MPI_Iallreduce is a non-blocking collective reduction operation, unlike MPI_Allreduce, which blocks until completion. MPI_Iallreduce allows computation and communication to overlap, improving performance by reducing idle time. This operation asynchronously sums up the number of local changes across all processes while allowing them to proceed with other computations.

```

1 int local_changes = 0;
2 // For each local point...
3 for (int i = 0; i < local_lines; i++)
4 {
5     int class = 1;
6     float minDist = FLT_MAX;
7     // For each centroid...
8     for (int j = 0; j < K; j++)

```

```

9     {
10         // Compute l_2 (squared, without sqrt)
11         float dist = euclideanDistance(&
12                                     local_data[i * samples], &centroids[j *
13                                     samples], samples);
14
15         // If the distance is smallest so far,
16         // update minDist and the class of the point
17         if (dist < minDist)
18         {
19             minDist = dist;
20             class = j + 1;
21         }
22     }
23 // If the class changed, increment the local
24 // change counter
25 if (local_classMap[i] != class)
26 {
27     local_changes++;
28 }
29 // Assign the new class to the point
30 local_classMap[i] = class;
31 }
32 // Gather all the changes from each process and
33 // sum them up, through non-blocking reduction
34 // operation
35 MPI_Request MPI_REQUEST;
36 MPI_Iallreduce(&local_changes, &changes, 1,
37               MPI_INT, MPI_SUM, MPI_COMM_WORLD, &
38               MPI_REQUEST);

```

Centroids update

Once points are assigned, each process computes partial sums of point coordinates and the count of points for each centroid. These values are accumulated in two arrays: pointsPerClass and auxCentroids. These partial results are aggregated across all processes using MPI_Allreduce, ensuring globally consistent centroid values with fast communication.

```

1 // Sum the coordinate of all local points
2 for (int i = 0; i < local_lines; i++)
3 {
4     int class = local_classMap[i];
5     pointsPerClass[class - 1]++;
6     for (int j = 0; j < samples; j++)
7     {
8         auxCentroids[(class - 1) * samples + j]
9         += local_data[i * samples + j];
10    }
11 }
12 MPI_Allreduce(MPI_IN_PLACE, pointsPerClass, K,
13               MPI_INT, MPI_SUM, MPI_COMM_WORLD);
14 MPI_Allreduce(MPI_IN_PLACE, auxCentroids, K *
15               samples, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);

```

Using MPI_IN_PLACE means that the input and output of the reduction are the same arrays (pointsPerClass and auxCentroids). This avoids extra buffers and directly updates local values with the final, globally reduced results. Every process thus ends up with a consistent view of how many points and total coordinate sums for each centroid across the entire data set.

Each process then updates centroids based on the reduced global values. The maximum distance between old and updated centroids is calculated locally and reduced globally using `MPI_Allreduce` to check for termination conditions.

```
1 MPI_Allreduce(&local_maxDist, &maxDist, 1,
    MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
```

I tried also an alternative approach, where centroids were distributed among processes so that each process updated only a subset of the centroids; however, this method incurred additional communication overhead and was ultimately less efficient.

Termination phase Once the clustering iterations are complete, each process holds a local array (`local_classMap`) containing the final cluster assignments for its subset of data points. Because the data points may be unevenly distributed among processes, the `MPI_Gatherv` function is used to gather these variable-sized local arrays on the root process, thereby assembling the complete output of the *k*-means algorithm.

```
1 MPI_Gatherv(local_classMap, local_lines, MPI_INT,
    classMap, recvcunts, rdispls, MPI_INT, 0,
    MPI_COMM_WORLD);
```

Finally, all processes free their allocated memory, and the algorithm terminates.

IV. PARALLELIZING WITH OPENMP

As a means of enhancing the performance of the *k*-means algorithm, I made use of the multi-thread library OpenMP. OpenMP is a widely used API for multi-platform shared-memory parallel programming. My 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. 4 threads) is used for parallel execution.

```
1 int nThreads = 4;
2 if (argc == 8)
3 {
4     nThreads = atoi(argv[7]);
5 }
6 omp_set_num_threads(nThreads);
```

The goal is to minimize overheads such as false sharing and redundant synchronizations. Unlike distributed-memory systems (where processes do not share memory), OpenMP uses a *shared-memory* model, meaning all threads can directly access shared data. Therefore, the code design must carefully handle concurrent data updates to avoid cache conflicts and to reduce synchronization costs.

A. Design of the program

Avoiding false sharing with padded arrays.

I allocate `pointsPerClass` and `auxCentroids` in the following way:

```
1 // PADDING is set to 16 -> 64 bytes
2 int *pointsPerClass = NULL;
3 float *auxCentroids = NULL;
4
5 posix_memalign((void **)&pointsPerClass, 64,
    sizeof(int) * K * PADDING);
6 posix_memalign((void **)&auxCentroids, 64,
    sizeof(float) * K * samples);
```

The `pointsPerClass` array uses extra padding (`PADDING`) so that each cluster's count occupies a separate cache line, preventing *false sharing*. False sharing occurs when multiple threads update different elements of the same cache line, causing unnecessary invalidations in each thread's cache. By ensuring each cluster's assignments counter is aligned to a different cache line, I minimize that performance penalty. The memory alignment via `posix_memalign(..., 64, ...)` ensures that these arrays start on a 64-byte boundary. This alignment minimizes cache line splits and reduces false sharing, thereby enhancing cache efficiency and overall performance.

Parallel Region and Iterative Loop

After initialization, the code enters a single `#pragma omp parallel` region, avoiding the overhead of repeatedly forking and joining threads on each iteration.

```
1 #pragma omp parallel
```

Initially, I explicitly declared each variable's scope because I used the default (`none`) clause, as a best practice. However, to improve code readability and clarity, I decided to omit these explicit declarations since OpenMP's default behavior treats variables declared outside the parallel region as shared—and that is exactly what I need.

Point Assignment

Inside the parallel region, each iteration begins with all threads participating in the *assignment* of data points to the closest centroid.

```
1 #pragma omp for reduction(+ : changes) schedule(
    static)
2 for (int i = 0; i < lines; i++)
3     ...
```

- `#pragma omp for`: Splits the loop across threads in the current team.
- `reduction(+ : changes)`: Each thread accumulates its own local count of reassignments,

and OpenMP sums them at the end of the loop into a single global total.

- `schedule(static)`: Assigns each thread a contiguous block of iterations, which is efficient when each iteration takes roughly the same time. I tried different scheduling strategies (static, dynamic, guided) with different chunk size, but static is the one that provided the best performance.

Since data and centroids are shared, each thread can directly read them. Only `classMap` is modified by each thread, but no synchronization is necessary because each iteration *writes* to a distinct index (`classMap[i]`).

Centroid update

In this step, the centroids are recalculated based on the current assignments. This involves two main phases:

```
1 #pragma omp for schedule(static) reduction(+ :
    pointsPerClass[ : (K * PADDING)],
    auxCentroids[ : (K * samples)])
2 for (int i = 0; i < lines; i++) {
3     ...
4 }
```

Reduction Clause: `reduction(+ : ...)` with array sections ensures each thread uses its own local copy of `pointsPerClass` and `auxCentroids`. At the end of the loop, OpenMP combines these local arrays element-wise into the global arrays.

```
1 #pragma omp for reduction(max : maxDist) schedule
    (static)
2 for (int i = 0; i < K; i++) {
3     ...
4 }
```

Reduction Clause: `reduction(max : maxDist)`: each thread computes its own local maximum distance, and OpenMP takes the maximum of these values at the end of the loop, as the global `maxDist`.

Termination Phase

In each iteration, one thread (via `#pragma omp single`) checks the termination conditions. If any condition is met, a shared boolean flag (`terminate`) is set to 1. In the next iteration, a barrier ensures that every thread has completed its previous iteration and sees the updated value of `terminate` before proceeding. All threads check this flag, and if it evaluates to true, break out of the loop, thereby exiting the parallel region.

```
1 #pragma omp barrier
2 if (terminate)
3 {
4     break; // Exit the while loop in all threads
5 }
```

Finally, the master thread writes out the results and frees allocated memory.

V. 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-Instruction Multiple-Threads model) turned out to be very efficient, by allowing notable speed-ups and augmentation of the throughput. Here follows an explanation of how I decided to design a possible CUDA implementation.

A. Designing the parallel structure

As I 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 SIMT 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 and can be parallelized by splitting the data into multiple parts, but 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 into 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 on the data points. This is the reason why I decided to implement the program with two kernels:

- `step_1_kernel` (points-based)
- `step_2_kernel` (centroids-based)

B. Program parameters and custom `atomicMax`

I configure the first kernel, `step_1_kernel`, with a two-dimensional block of 32×32 threads (i.e., 1024 threads per block). Since each thread processes a single data point, the total number of blocks is computed as:

$$\left\lceil \frac{|X|}{32 \times 32} \right\rceil,$$

This ensures a one-to-one mapping from threads to data points, and it leverages coalesced memory accesses when reading the points.

For the second kernel, `step_2_kernel`, since we usually have $K \ll |X|$, I use a one-dimensional grid to process the centroids where each thread processes one centroid (if its ID is below K). If there are K centroids:

$$\text{threads_per_block} = 256, \quad \text{blocks} = \left\lceil \frac{K}{256} \right\rceil$$

The program makes also use of a custom function, called `custom_atomic_max()`. This function has been implemented because it allows me to perform an `atomicMax()`-like function for float numbers, which would not be normally possible with the built-in CUDA function. Here I 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()` (atomic compare-and-swap) operation, which in turn performs atomically the following check:

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

The function only exits when the value in the specified address is equal to the expected value prior to performing the transaction. This behavior prevents race conditions that could occur when multiple threads attempt to update `maxDistance` simultaneously.

C. Analysis of the kernels

As I mentioned previously, I am making use of two kernels: `step_1_kernel` and `step_2_kernel`. In both kernels, every update to global memory is performed atomically, thereby avoiding potential race conditions. Below, I describe how each kernel works.

`step_1_kernel` (Points-based)

The first kernel is called on all the data, and each point is assigned to a thread. The kernel first copies the current centroids from global memory into dynamic shared memory (`sharedCentroids`) for faster access. It then initializes block-level arrays

`blockSums` and `blockCounts` to accumulate partial results without the overhead of frequent global memory writes. A preliminary check ensures that each thread is processing a valid data point. While the `if` statement may seem like a possible cause of warp divergence, it only affects threads in the final block, leaving most blocks fully utilized. At the end of the kernel, each block's partial results are atomically added to the global accumulators. A key point is that if the point's assigned class changes, a block-level counter (`blockChanges`) is updated atomically.

```
1 __global__ void step_1_kernel(float *data, float
  *centroids, int *globalCounts, float *
  globalSums, int *classMap, int *
  changes_return)
2 {
3   ...
4   // Allocate dynamic shared memory:
5   extern __shared__ char sharedBuffer[];
6   float *sharedCentroids = (float *)
  sharedBuffer; // K * samples
7   float *blockSums = sharedCentroids + K *
  samples; // K * samples
8   int *blockCounts = (int *) (blockSums + K *
  samples); // K
9
10  __shared__ int blockChanges;
11
12  // Copy centroids into shared memory
13  for (int i = tid; i < K * samples; i +=
  blockSize)
14  {
15      sharedCentroids[i] = centroids[i];
16  }
17  ...
18  __syncthreads();
19
20  // Each thread processes one data point.
21  if (idx < lines)
22  {
23      ...
24      // Update block-level accumulators
25      int accum_idx = class_idx - 1;
26      atomicAdd(&blockCounts[accum_idx], 1);
27      for (int j = 0; j < samples; j++)
28      {
29          atomicAdd(&blockSums[accum_idx *
  samples + j], point[j]);
30      }
31  }
32  __syncthreads();
33
34  // One thread (tid==0) per block updates the
  global accumulators.
35  if (tid == 0)
36  {
37      // Add local changes to the global
  changes counter
38      atomicAdd(changes_return, blockChanges);
39
40      // For each centroid...
41      for (int c = 0; c < K; c++)
42      {
43          atomicAdd(&globalCounts[c],
  blockCounts[c]);
44          for (int j = 0; j < samples; j++)
45          {
46              atomicAdd(&globalSums[c * samples
  + j], blockSums[c * samples + j]);
47  }
```

step_2_kernel (Centroids-based)

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 squared distance between the previous centroid position and the new one, so as to compute the `dist` variable, needed for convergence. Finally, via the use of the `custom_atomic_max()` function, the greatest distance is stored in memory in `maxDistance`.

```
1 __global__ void step_2_kernel(float *globalSums,
2   float *centroids, int *globalCounts, float *
3   maxDistance)
4 {
5   int c = blockIdx.x * blockDim.x + threadIdx.x;
6   // Each thread updates one centroid if c < K
7   if (c < K)
8   {
9     ...
10    // Update maximum distance using a custom
11    atomic max for floats
12    custom_atomic_max(maxDistance, dist);
13  }
```

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

VI. INTERLACING MULTI-PROCESSING WITH MULTI-THREADING

So far, I implemented various solutions for the 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 is used for performing computations, given the directives of the master process(es).

In this section I will show how the multi-threading approach that I previously considered, namely OpenMP, 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. My parallelization strategy can be described with two main key points:

Two-level parallelism

- *MPI Data Distribution (Coarse-Grained)*: Each process receives a subset of data points (`local_lines`) via an `MPI_Scatterv` operation. This means every process operates on a fraction of the overall data points, allowing the algorithm to scale across multiple nodes.
- *OpenMP Loop Parallelism (Fine-Grained)*: Within each MPI process, I further exploit shared-memory parallelism using OpenMP. Multiple threads work in parallel on local loops (e.g. assigning data points to centroids, computing partial sums, ...), which speeds up computations on multicore CPUs.

Thread Safety

To ensure correctness while mixing MPI calls and OpenMP threading, I initialized MPI with `MPI_THREAD_FUNNELED`. This approach restricts MPI function calls to the *main* thread, avoiding data races or undefined behavior that can occur if multiple threads make MPI calls simultaneously.

In each iteration, the workflow combines both MPI collectives and OpenMP parallel loops:

Point Assignment

Data is already distributed across processes by `MPI_Scatterv`. Each process holds its local partition in `local_lines`. Threads in each process assign their local points to the nearest centroid in parallel:

```
1 int local_changes = 0;
2 #pragma omp parallel for reduction(+:
3   local_changes) schedule(static)
4 for (int i = 0; i < local_lines; i++)
5 {
6   ...
7 }
```

Then, the local changes are combined across all processes with a non-blocking collective:

```
1 MPI_Iallreduce(&local_changes, &changes, 1,
2   MPI_INT, MPI_SUM, MPI_COMM_WORLD, &request);
```

Centroid Update

Each process computes partial sums of the coordinates for points assigned to each cluster and counts the assignments using two arrays:

- `auxCentroids` accumulates coordinate sums for each cluster.

- A padded array `pointsPerClass` (with extra space defined by `PADDING`) holds the number of points in each cluster.

Within each process, these updates are performed in parallel using OpenMP:

```
1 #pragma omp parallel for schedule(static)
   reduction (+:pointsPerClass[ : (K * PADDING)
   ], auxCentroids[ : (K * samples)])
2 for (int i = 0; i < local_lines; i++)
3 {
4     ...
5 }
```

Because the padded `pointsPerClass` array is not stored contiguously in memory, its values are copied into a contiguous array (`pointsPerClassContig`) before performing the MPI reduction. This contiguous layout is necessary for efficient use of `MPI_Allreduce`:

```
1 for (int i = 0; i < K; i++) {
2     pointsPerClassContig[i] = pointsPerClass[i *
   PADDING];
3 }
4
5 MPI_Allreduce(MPI_IN_PLACE, pointsPerClassContig,
   K, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

Similarly, the `auxCentroids` array is reduced across all processes:

```
1 MPI_Allreduce(MPI_IN_PLACE, auxCentroids, K *
   samples, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

After the reductions, the contiguous results are copied back into the padded array:

```
1 for (int i = 0; i < K; i++) {
2     pointsPerClass[i * PADDING] =
   pointsPerClassContig[i];
3 }
```

Each process then updates its copy of the centroids array (which represents the global centroids) and computes the maximum centroid movement (`local_maxDist`) using an OpenMP loop:

```
1 float local_maxDist = 0.0f;
2 #pragma omp parallel for reduction(max:
   local_maxDist) schedule(static)
3 for (int i = 0; i < K; i++)
4 {
5     ...
6 }
```

Finally, a global reduction using `MPI_Allreduce` is performed to compute the maximum centroid movement across all processes:

```
1 MPI_Allreduce(&local_maxDist, &maxDist, 1,
   MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
```

Termination phase Once the clustering iterations are complete, each process holds a local array (`local_classMap`) containing the final cluster assignments for its subset of data points. These arrays are gathered on the root process using the `MPI_Gatherv` collective:

```
1 MPI_Gatherv(local_classMap, local_lines, MPI_INT,
   classMap, recvcunts, rdispls, MPI_INT, 0,
   MPI_COMM_WORLD);
```

Finally, all processes free their allocated memory, and the algorithm terminates.

Avoiding False Sharing and Why `auxCentroids` is *Not* Aligned

- *Padded pointsPerClass*: To avoid false sharing when multiple OpenMP threads increment cluster counter, each cluster's assignments counter is padded so that each resides on a separate 64-byte cache line. This is done via `posix_memalign(...)` with the `PADDING` factor.
- *No 64-Byte Alignment for auxCentroids*: In the pure OpenMP version, explicitly aligning `auxCentroids` on 64-byte boundaries was beneficial because threads frequently updated its coordinates. However, in the hybrid MPI+OpenMP implementation, the update pattern combined with MPI reductions makes explicit alignment less critical. Empirical tests showed that the overhead of alignment slightly *increased* computation time, so I opted to allocate `auxCentroids` using standard `malloc`.

VII. PERFORMANCE ANALYSIS


I will now proceed to show the results for each implementation. The evaluation of performances considers the computation time of the parallel versions relative to the sequential one, as well as the speedup, efficiency, and both weak and strong scaling behaviors.

The parameters given to the program were the following:

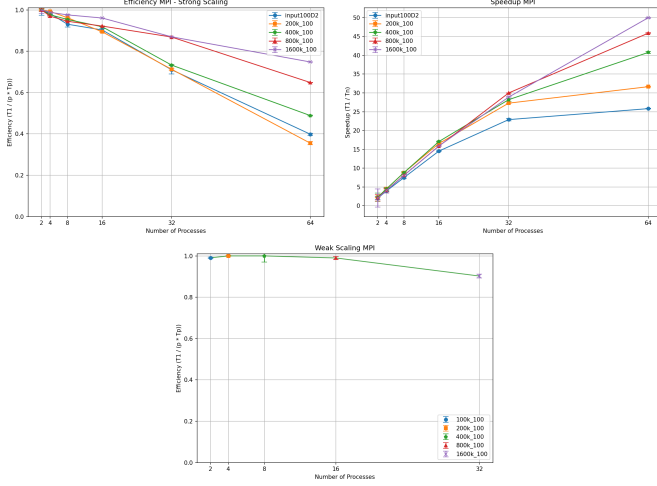
```
<program> <input> 20 5000 1 0.0001 <output>
```

To better analyze the performance of the different implementations, I generated additional point clouds, not included in the test files. These point clouds were randomly generated by analyzing the biggest test file available (100k points with 100 dimensions), identifying the minimum and maximum value over all dimensions of all points, and picking from a uniform distribution in that range. The files generated have 100k (100k_100), 200k (200k_100), 400k (400k_100), 800k (800k_100) and 1.6M (1600k_100) points and 100 dimensions.

The class assignments produced by the parallel implementations are identical to those of the sequential version, confirming that every point is correctly

classified. Each experiment has been run 5 times. Detailed results, entire distribution of timings and more plots of the tests are available [on GitHub](#) .

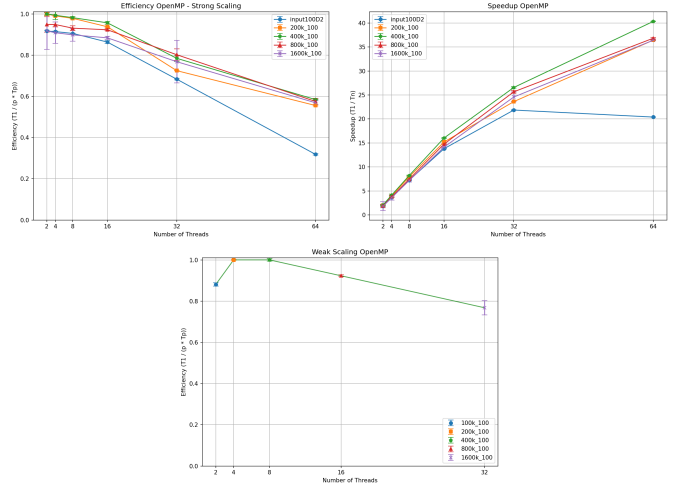
A. MPI



With MPI, I noticed that by augmenting the number of processes, the *speedup* continued to grow. However, as expected, after a given number of processes, performance no longer scales linearly. With the input file `input100D2.inp`, which has $|X| = 10^5$, I start observing a non-linear increase after $p = 16$, while with input file `1600k100.inp`, which has $|X| = 1600000$, it stops increasing linearly after $p = 32$, even though for $p = 64$ I still obtain a speed up of ~ 50 , which is quite good. This is because the overhead of exchanging data starts to become tangible, negatively affecting the performances. In terms of *weak scaling*, the efficiency remains close to ideal when the problem size is increased proportionally with the number of processes, meaning that the program is weakly scalable. Under *strong scaling*, the efficiency is high at lower process counts but drops after $p = 32$ as communication overhead becomes significant.

B. OpenMP

Using OpenMP, I observe that as the number of threads increases, the *speed-up* also rises. However, beyond a certain point, performance stops scaling linearly. For the smaller dataset `input100D2.inp`, the speed-up curve begins to flatten around $t = 16$, whereas for the dataset `400k100.inp` near-linear scaling holds up until $t = 32$. Even at $t = 64$, I still observe a speed-up of ~ 40 , which is not quite bad. This drop-off in linearity stems from the growing overhead



of synchronization and shared-memory contention as more threads are introduced.

In terms of *weak scaling*, efficiency remains close to ideal when the workload increases proportionally with the thread count. However, under *strong scaling*, the efficiency is high at lower thread counts but begins to decline once synchronization overhead becomes dominant, particularly beyond $t = 32$.

C. CUDA

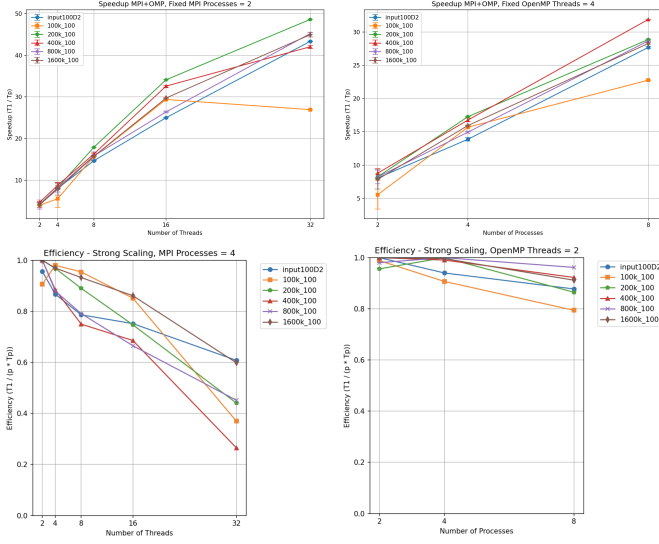
Here I show a table comparing sequential and CUDA computation time and relative speedup for all test files.

Test File	Seq	CUDA	Speedup
100D2	57.35	0.95	60.20
100D	1.05	0.03	35.99
20D	0.34	0.005	63.30
10D	0.01	0.001	8.55
2D	0.01	0.001	10.78
100k_100	20.30	0.35	57.54
200k_100	52.81	0.76	69.85
400k_100	99.26	1.46	67.94
800k_100	195.09	3.16	61.75
1600k_100	286.46	4.76	60.15

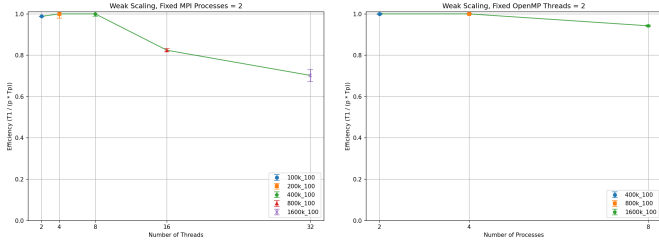
The sequential version takes significantly longer for computation. The speedup achieved by CUDA is ~ 60 for most test files. However, for small files (such as 10D and 2D), the speedup is noticeably lower. This is due to the overhead introduced by CUDA, primarily from memory transfers and kernel launch operations, which become more significant when processing smaller datasets.

D. MPI + OpenMP

I have calculated the total number of processes/threads as $p \times t$.



In both scenarios—whether I fix the number of threads and vary the number of processes, or vice versa—larger datasets achieve higher speedups and maintain better efficiency as the number of processes/threads increases. This is because the substantial computational load of larger inputs better offsets the overhead of communication and synchronization. Conversely, smaller datasets exhibit diminishing speedup more quickly, with efficiency declining sharply as processes/threads increase significantly.

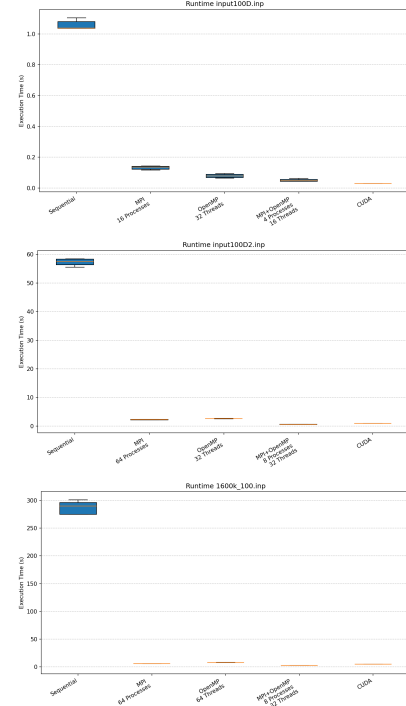


For fixed $p = 2$, increasing the number of OpenMP threads maintains efficiency close to ideal up to $t = 16$, beyond which it begins to decline due to overhead.

On the other hand, for fixed $t = 2$, increasing the number of MPI processes maintains an ideal efficiency for all the combinations, meaning the program is weakly scalable.

E. Overall Comparisons

The sequential version consistently exhibits the longest runtime, while the parallel approaches substantially reduce execution time. In the following Figure, boxplots illustrate the distribution of runtimes across all implementations (Sequential, MPI, OpenMP, Hybrid MPI+OpenMP, and CUDA) for three of the test files.



VIII. CONCLUSIONS

This project made me realize the great potential of parallelizing sequential programs to drastically reduce computation time. GPU programming is the future. Although writing programs in CUDA is clearly more difficult than parallelizing a program on the CPU using OpenMP or MPI, its advantages are endless. I can only imagine the impact that advancements in GPU technology will have on AI and LLM development in the coming years.

I would say that CUDA and the cluster were the sources of my nightmares and sleepless nights over the past few weeks, yet they also pushed me to achieve satisfactory results.

I encountered several problems during implementation due to hardware limitations. I have a Mac with Apple Silicon, which is not ideal for parallel programming because it does not support various debugging libraries. Moreover, it has only 4 performance cores and 4 efficiency cores and lacks an NVIDIA GPU, making it not useful for testing. For this reason, I was forced to use the cluster, which was not always operational. In the end, however, I managed to resolve all the issues.