Contents

K-means

Elia Belli 2006305, Federico Fantozzi 2047034

1 Introduction

K-means is an iterative algorithm for dividing a set of objects into K groups, called clusters. In this document, we present 4 parallel implementations of the algorithm created with the following technologies:

- MPI
- OpenMP
- CUDA
- MPI + OpenMP

2 Algorithm

Analyzing the sequential version of the algorithm, we can distinguish 3 main sections within each iteration:

1. **Point classification**: for each point, the cluster of belonging is identified based on the nearest centroid, and the number of changes from the previous iteration is taken into account (termination condition).
2. **Centroid recalculation**: based on the point classification just performed, the coordinates of the centroids are updated as the average of the positions of the points belonging to their cluster.
3. **MaxDist calculation**: the displacement of each centroid from the previous iteration is calculated, and the maximum is taken (termination condition).

2.1 Possible modifications

Below are some modifications to the base algorithm that we considered but ultimately discarded:

- In point classification, we are interested in calculating the nearest centroid, but to do so, it is sufficient to apply the Pythagorean theorem $a^2 = b^2 + c^2$ and compare the squares of the distances. In fact, if $d_1 < d_2 \Rightarrow d_1^2 < d_2^2$, this allows us to avoid the calculation of $sqrt()$, which is generally a costly operation. This modification resulted in a different output from the base version due to different approximations.

- Inspired by the K-means++ variant, we thought of applying a heuristic for the initial centroid selection to converge in fewer iterations. This modification was discarded because the base algorithm we considered always uses the same set of initial centroids.

3 Parallel Implementations

The algorithm has a complexity of $O(lines \cdot samples \cdot K)$ in each iteration, with *lines* generally $>> K$ and *samples*, so the main focus was to scale with the number of points.

3.1 MPI

3.1.1 Input Partitioning

The points are partitioned among the processes such that each process works on a contiguous block of points of size $|points|/|processes|$. If there are remaining points, they are distributed among the processes in a round-robin manner.
Each process will work on the assigned block for all iterations, without needing to know the assignment of other blocks. Only upon reaching the termination condition, through the collective `Allgather`, the process with rank 0 will retrieve the portions of $classMap$ from the various processes and write them to the output file.
Centroids are also partitioned in the same way, so each process performs steps (2) and (3) only for its own.

Partitioning the data this way allows a process to proceed to the next step of the algorithm using only the data it calculated itself in the previous step. Furthermore, this organization enables asynchronous transmission of the necessary data for the subsequent synchronization points during the transition from one step to another.

3.1.2 Communication

After completing step (1), a process performs two `Iallreduce` operations: one on $pointsPerClass$, needed to complete the mean calculation, and another on *changes*. Meanwhile, the process can continue summing the coordinates of its points in $auxCentroids$.

Now we have a synchronization point: an `Allreduce` is performed so that each process has the data from *auxCentroids*, and it waits for the previously started `Iallreduce` on *pointsPerClass* to be completed, so that each process can proceed with dividing the coordinates of the centroids assigned to it.

At the end of step (2), each process has already calculated the coordinates for its own centroids. Therefore, we can perform an `Iallgather` on *auxCentroids*, overlapping it with step (3), instead of performing it separately at the end. Since step (3) depends on both the values of *centroids* and *auxCentroids*, the result of the `Iallgather` must be collected into a temporary array so as not to influence the execution of step (3).

At the end of step (3), the processes are synchronized, and the termination condition of the algorithm is calculated. Before proceeding, a `wait` on the `Iallgather` of *auxCentroids* and a `memcpy` ensure that the data related to the new positions of the centroids are correctly calculated.

3.1.3 Performance

Strong Scaling: we performed measurements for strong scaling on various inputs, doubling the size at each test to observe how the program behaved on both small and large inputs.

- According to **Amdahl's law**, every program has portions that cannot be parallelized, so by increasing the number of processes we reach a plateau.
- Once the plateau is reached, efficiency starts to decrease due to the overhead of communication between processes, in line with the **Universal Scalability Model**[1], which extends Amdahl's law to account for bandwidth and latency limits.

Weak Scaling: The scaled speedup follows **Gustafson's law** up to 16 processes with efficiency above 90%, then it starts to drop for the following reasons:

- As the input size increases, the amount of data to transfer increases, thus latency in communication grows.
- As the input increases, data access occurs in lower-level caches.

[1] Universal Scalability Model

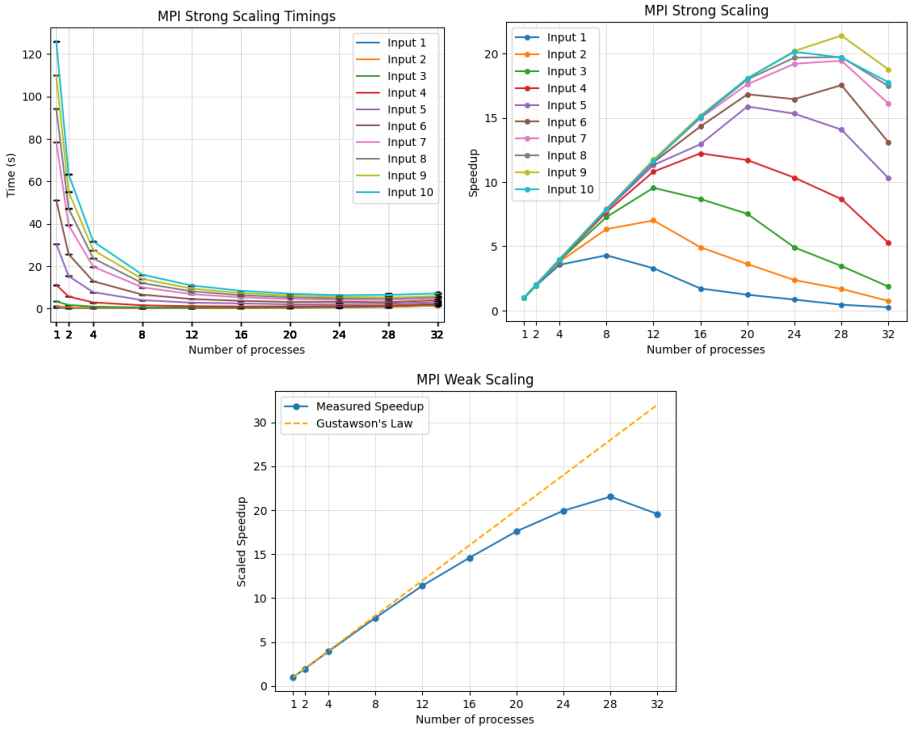

Figure 1: Measurements performed on input with points of 100 dimensions and parameters 100 150 0.01 0.01

	Number of Points									
Processes	3.125	6.250	12.500	25.000	37.500	50.000	62.500	75.000	87.5000	100.000
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
2	0.96	0.98	0.99	0.99	1.00	0.99	1.00	1.00	1.00	0.99
4	0.89	0.95	0.98	0.98	0.99	0.99	0.99	0.99	0.99	0.99
8	0.54	0.80	0.91	0.96	0.98	0.98	0.99	0.99	0.99	0.99
12	0.27	0.58	0.80	0.90	0.94	0.96	0.97	0.97	0.98	0.97
16	0.11	0.31	0.54	0.76	0.81	0.90	0.94	0.94	0.95	0.94
20	0.06	0.18	0.38	0.60	0.80	0.84	0.88	0.90	0.90	0.90
24	0.04	0.10	0.20	0.43	0.64	0.69	0.80	0.82	0.84	0.84
28	0.02	0.06	0.12	0.31	0.50	0.63	0.70	0.70	0.76	0.70
32	0.01	0.02	0.06	0.16	0.32	0.41	0.50	0.55	0.60	0.55

Table 1: Efficiency derived from the measurements in 1

3.2 OpenMP

The OpenMP version was developed starting from an optimized version of the sequential code, initially following the ideas used in the MPI version.

We chose to use a single parallel region that encloses the entire `do-while`, thus avoiding the creation and destruction of threads at each iteration.

3.2.1 Input Partitioning

Unlike MPI, where input partitioning was handled manually, in OpenMP, the workload distribution among threads is automated using `pragma` directives.
By using the default scheduling, which corresponds to `static` with a `chunk size` of $|points|/|threads|$, the load is evenly distributed among the threads, replicating the same partitioning strategy adopted in the MPI version.

3.2.2 Synchronization and Reductions

Unlike MPI, we have the advantage of shared memory, which allows us to reduce synchronization points and eliminate the overhead of communication between processes.

Synchronizations: the only synchronization points are between the sum and the division for the calculation of $auxCentroids$, that is, when the threads transition from working on their points to their centroids, and after the calculation of $maxDist$.
Using a `pragma omp single`, a single thread is responsible for evaluating the termination conditions and preparing the shared arrays for the next iteration.

Reductions: within the parallel region, we use `pragma omp for` with the `reduction` clause (and the `nowait` clause if synchronization is unnecessary) to perform reductions on $auxCentroids$ and $pointsPerClass$.
We noticed that using large values for K and $samples$ leads to stack overflow, due to the fact that OpenMP uses the stack to create private thread support arrays. To overcome this problem, we implemented a reduction by allocating private arrays on the heap, but we had to introduce atomic operations. The analysis was then conducted on a much smaller number of centroids, so we decided to return to the initial version due to the slowdown caused by the atomics.

3.2.3 Performance

For the performance analysis of OpenMP, we decided to limit ourselves to the number of available physical cores, without utilizing hyperthreading.

Cache Usage: we analyzed cache usage using `perf` on a local CPU with 4 physical cores. Specifically, we wanted to understand how many cache misses we had and the possible cause.

We observed an increase in `L1_dcache_load_misses` and `LLC_load_misses` proportional to the increase in input size (with threads fixed at 4), so we proposed two hypotheses:

- **False Sharing**: the shared memory portions *data* and *centroids* cannot cause this phenomenon since they are used only for reading and never modified. Meanwhile, writing to *auxCentroids* happens privately and only then is the reduction performed, so this hypothesis was discarded.
- **Limited Cache**: performance tends to deteriorate as the input size increases because it is no longer possible to keep all the data in cache. This was more evident locally, with L1 caches of 64 KiB, L2 caches of 512 KiB, and L3 caches of 3 MiB, which forced nearly constant accesses to DRAM. On the other hand, considering a CPU from the cluster and input size 10, weighing 40 MB, it can reside entirely in the L3 cache of 128 MiB, and when partitioned across 32 threads, it fits almost entirely into the L1 cache of 1 MiB.

This observation also applies to the MPI version, as the cause is not shared memory. Furthermore, it helps us understand that a limiting factor for our program is the amount of available cache. Therefore, a distributed version of the program is more suitable if we want to process even larger inputs.

Strong Scaling: since the program works with shared memory, it does not follow the Universal Scalability Model as MPI does, showing better strong scaling. The plateau caused by Amdahl's law is less evident, which demonstrates that the sequential portion during computation is very low, and the problem is highly parallelizable.

Weak Scaling: the scaled speedup follows **Gustafson's law** with efficiency above 90% up to 28 threads.

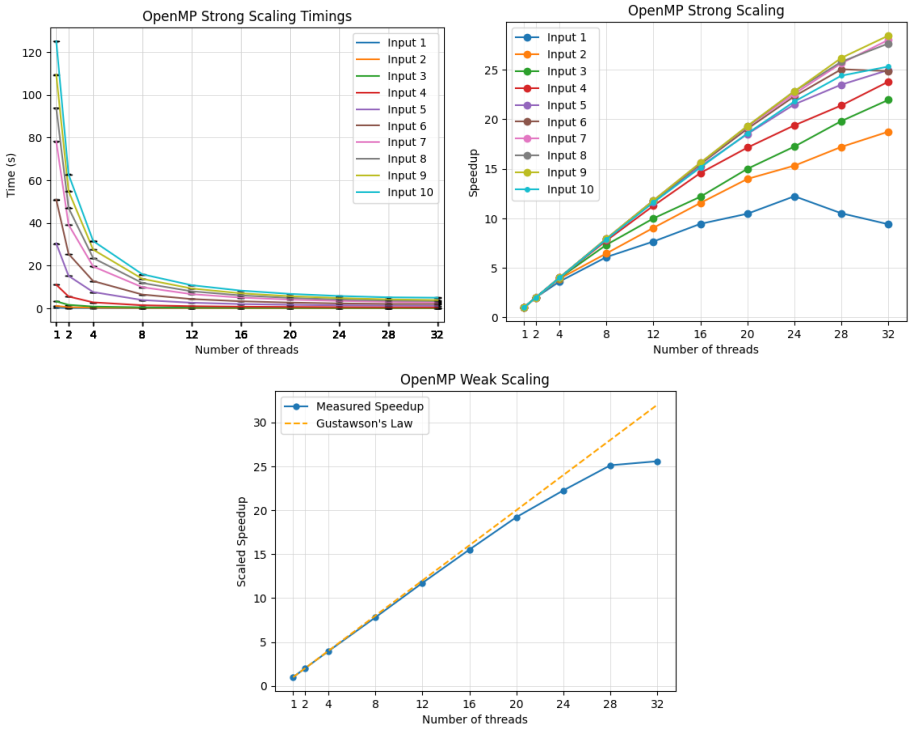

Figure 2: Measurements performed on input with points of 100 dimensions and parameters 100 150 0.01 0.01

	Number of Points									
Threads	3.125	6.250	12.500	25.000	37.500	50.000	62.500	75.000	87.5000	100.000
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4	0.90	0.97	0.99	0.99	0.99	1.00	1.00	0.99	0.99	0.99
8	0.76	0.81	0.92	0.97	0.98	0.98	0.99	0.99	0.99	0.99
12	0.64	0.75	0.83	0.94	0.97	0.97	0.98	0.98	0.98	0.98
16	0.59	0.72	0.76	0.91	0.95	0.96	0.98	0.97	0.98	0.98
20	0.52	0.70	0.75	0.86	0.92	0.95	0.96	0.97	0.97	0.97
24	0.51	0.64	0.72	0.81	0.89	0.93	0.94	0.95	0.95	0.95
28	0.38	0.61	0.71	0.76	0.84	0.89	0.92	0.92	0.92	0.94
32	0.29	0.59	0.69	0.74	0.78	0.77	0.88	0.86	0.86	0.89

Table 2: Efficiency derived from the measurements in 2

3.3 CUDA

The CUDA program is divided into 3 kernels to ensure synchronization between blocks, as it is necessary for all threads to finish executing the previous one before starting the next.

3.3.1 Kernel 1: kmeansClassMap

Each thread is assigned a point, on which it performs the following operations:

1. Calculates the class of the point, updating $classMap$ in global memory and $localPointsPerClass$ in shared memory.
2. Checks if the class has changed from the previous iteration; if so, it updates a counter $localChanges$ in shared memory: updating data first in shared memory, although it still requires an atomic sum, reduces the wait time for accessing the critical section as only threads from the block compete for the write, and access is faster.
3. Sums the coordinates of the point to the corresponding center in $auxCentroids$ in global memory: due to the randomness of the classification, performing a local sum for the block would require having the entire $auxCentroids$ in shared memory, but this is only possible for small values of K and $samples$.

Once all threads in the block have completed the steps, the local sum of $pointsPerClass$ and $localChanges$ are transferred to $pointsPerClass$ and $changes$ in global memory.

Shared Memory: the first version of the program worked on $centroids$ and $data$ in global memory. Since $centroids$ is accessed by all threads, we decided to fully load it into shared memory at the start of the kernel, reducing accesses to global memory by a factor of $blockSize$ times: despite the speedup, this approach is limited by the shared memory size, which saturates quickly as K and $samples$ increase. For this reason, we switched to **tiling** with the same effect but much lower shared memory occupancy.
At this point, we had shared memory available to reduce accesses to $data$ in global memory. Each thread works on a single point, so each thread is assigned a portion of shared memory to load it. Since $blockSize$ points need to be loaded into shared memory, it is necessary to adjust the block size appropriately. This is done by a host-side function at the start of the program. Using this technique, we reduced each thread's accesses to global memory by $(K \cdot samples)\times$, achieving a $2\times$ speedup for the entire program.

Tiling: in the classification of its point, each thread calculates the distance to each center in the same order. This allows every iteration (on the centroids in the kernel) to load a centroid into shared memory from which all must calculate the distance, and then replace it with the next one in the following iteration.

This requires synchronization within the block, slightly slowing down the execution compared to the previous version but bypassing the shared memory limitation.
Since the tiling is designed such that each thread in the block loads one coordinate of the centroid into shared memory, there can be cases where many threads do not contribute to the loading (e.g., if $blockSize = 128$ and $samples = 2$, 126 threads are idle). For this reason, using a `tileSize` parameter in the kernel, we considered loading a number of centroids into shared memory that maximizes the threads used, thus reducing the number of synchronizations. This idea was eventually discarded because it required the use of the modulo operator, which we found inefficient on GPUs.

3.3.2 Kernel 2: kmeansCentroidsDiv

Once all blocks have executed Kernel 1, we can complete the calculation of the average by dividing the centroid coordinates by the corresponding points per class.

Initially, we thought of assigning one centroid to each thread, so each would perform a single access to global memory to retrieve the value in $pointsPerClass$ and save it in a register, reducing global memory accesses by $samples\times$. Since the number of centroids is generally low, this led to the execution of only a few blocks, so most of the SMs were inactive during this phase.
Therefore, it was decided to distribute the coordinates rather than the centroids to the threads, so each performs the division of coordinates starting from its $globID$ with an offset of $gridSize$, achieving a $3\times$ speedup for the kernel compared to the previous version.

3.3.3 Kernel 3: kmeansMaxDist

After calculating the new centroid coordinates, we need to calculate the termination condition for the minimum centroid shift.

Each thread is assigned a centroid for which it calculates the shift from the previous iteration. At this point, it must compare it with the maximum and update it if it is greater, which requires the use of an atomic max operation performed first on a shared memory variable and finally in global memory at the end of the kernel.

AtomicMax: CUDA does not provide an `atomicMax` operation for floats, but it can be implemented using `atomicCAS`, which allows checking if the value has been modified (compare) and swapping it with the new one if it has remained the same. In our case, the value replaced is the new maximum calculated from the last read value.

3.3.4 Performance

Below, we report the speedup relative to the sequential version, measured on an NVIDIA Quadro RTX6000 from the cluster and an NVIDIA GeForce MX920, which we later used for kernel analysis.

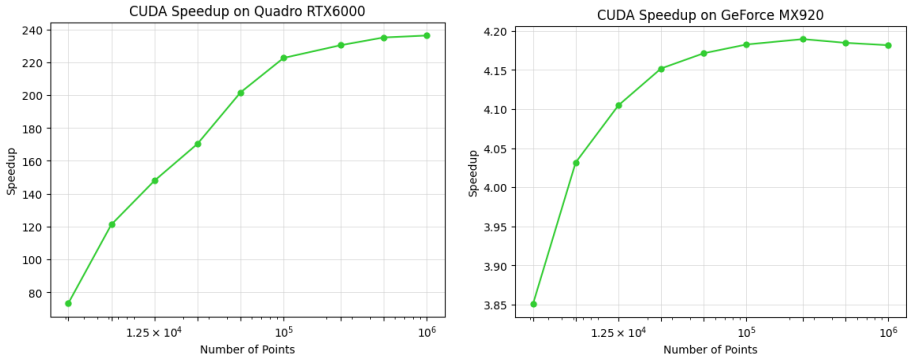

Figure 3: Speedup measured on input from 3.125 to 1.000.000 points (100D) with parameters 100 150 0.01 0.01

3.4 MPI + OpenMP

This version directly follows from the previous MPI and OpenMP versions, which, based on the same ideas, were well suited for merging.

3.4.1 Process Distribution

For each node used, we have one MPI process and 32 threads, in order to fully utilize the cores, leveraging shared memory for computation and using MPI only for communication.
We used the `SERIALIZED` mode and introduced `omp single` at the synchronization points, so that only the first thread that reaches them executes the MPI call, while the others wait until the communication is completed in case it is blocking.
The rest of the program's structure remained the same as in the OpenMP version.

3.4.2 Performance

We ran the tests by doubling the number of nodes, from 1 to 32.

Although distributing the data across multiple nodes allows us to leverage more computational power, solving the problem at the cache limit, the bottleneck becomes communication between nodes, which can be improved by increasing the bandwidth or modifying the topology.

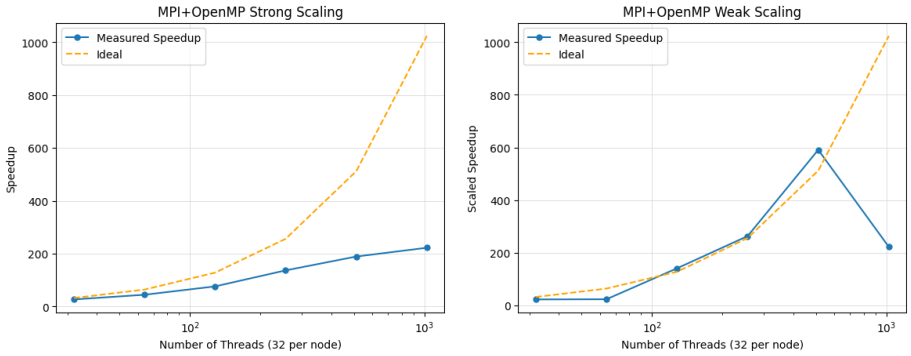

Figure 4: Strong Scaling measured on input of 1.000.000 points (100D)
Weak Scaling measured on input from 31.250 to 1.000.000 points (100D)
with parameters 100 150 0.01 0.01