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

I. Introduction

When talking about clustering and unsupervised learning, it's quite common to hear about the k-means algorithm, and for good reasons: it allows to efficiently cluster a dataset of d dimensions, and it employs the notion of convergence in order to do so. This, computationally speaking, means to repeat some operations over and over again until some stopping conditions are met.

The algorithm is not perfect though, and presents some issues:

1) the algorithm is fast in clustering, but we cannot be certain that it clusters *well*;
2) the algorithm doesn't work with non-linear clusters;
3) the initialization can make a great impact in the final result.

Many people prefer to use other clustering methods, such as the fitting of Gaussian Mixture Models. Albeit not being perfect, k-means still works well in simple, linear clusters. For the sake of this project, we are going to consider a vanilla k-means algorithm with Lloyd's initialization (the first k centroids will be selected randomly).

A. Algorithm structure

The k-means algorithm can be described with the following pseudocode, where X is the set of data points, $C = \{\mu_1, \mu_2, ..., \mu_k\}$ is the set of centroids and Y is the set of assignments:

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

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

The algorithm consists of 4 main blocks:

- the **initialization block**, where all the centroids will receive a starting, random position (as per Lloyd's method);
- the **assignment block**, where the Euclidean distance between a point and all centroids is computed, for all centroids. The point will be assigned to a cluster depending on the following operation:
$$\arg\min_k \left(\|\mu_k - x_i\|\right)$$
- 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 professors Diego García-Álvarez and Arturo Gonzalez-Escribano from the University of Valladolid [1]. 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:

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

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

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

    classMap[i]=class;
}
```

Notice the presence of the two nested **for** loops: sequentially, they would take a time of $O(|X| \cdot |C|)$, which may be optimized just by taking a simple single instruction multiple data approach (indeed, with $m > 1$ different processes or threads, it would take a time of $O\left(\frac{|X| \cdot |C|}{m}\right)$ each, which is already better than the first option).

TABLE I
RUNTIME OF THE SEQUENTIAL VERSION $(\mu \pm \sigma^2)$

Input tests		
20D	100D	100D$_2$
0.35398 ± 0.00019	1.06496 ± 0.00061	111.27844 ± 0.06059

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:

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

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

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

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

C. Performance evaluation

All the following codes have been tested on Sapienza's internal cluster, where for each node an **AMD EPYC 7301 CPU** is mounted (which has up to **64 logical cores**). In some cases, nodes with NVIDIA GPUs can be requested, and the used GPU is the **RTX Quadro 6000**, which has a compute capability of 7.5; for this reason, the CUDA program is compiled with that architecture in mind, so newer features of other CUDA architectures haven't been taken into account.

For each program, we tested the applications for 30 times on 3 different tests: 20D, 100D and 100D$_2$. The programs were launched with the following test:

```
<test_file> 30 500 0.1 0.1 <out>
```

All the measured values can be found at the end of this report in multiple tables, which will be referenced throughout the paper. Specifically, the timings are reported in the form $\mu \pm \sigma^2$, where μ stands for the mean and σ^2 for the standard deviation. In the timing plots, we haven't reported the standard deviations, due to the fact that they were not properly readable; such values can be however found in the tables.

The **speedup** $S_{d,t}$ (where T stands for the timing, d for the test and t for the number of threads/processes used) and the **efficiency** $E_{d,t}$ have also been reported. Such values have been computed with the following formulas:

$$S_{d,t} = \frac{T_{\text{serial},\, d}}{T_{\text{parallel},\, d,t}} \qquad E_{d,t} = \frac{S_{d,t}}{t}$$

Lastly, we haven't considered the Ahmdahl and Gustafson laws in the evaluation, since the vast majority of our computations have been parallelized. While there are a few operations that have been done sequentially, they are negligible with respect to the programs' workloads.

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: Balancing the workload across processes Minimizing the frequency of message passing Both factors are crucial for scalable and high-performance parallel applications. Here's how these optimizations are implemented in this k-means clustering algorithm:

A. MPI Implementation Approach

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.

1) Initialization phase: In 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.

2) Loop phase: The loop phase involves two key operations that are parallelized over different data structures:

- **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 using `MPI_Iallreduce` for checking at the and the termination conditions;
- **centroid update**: once all points are assigned, each process calculates local partial sums for centroids 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.

3) 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.

B. MPI Performance Analysis

We tested with 1, 2, 4, 8, 16, 32 and 64 MPI processes. The MPI program achieves nearly linear speedup across all input sizes up to 16 processes. For

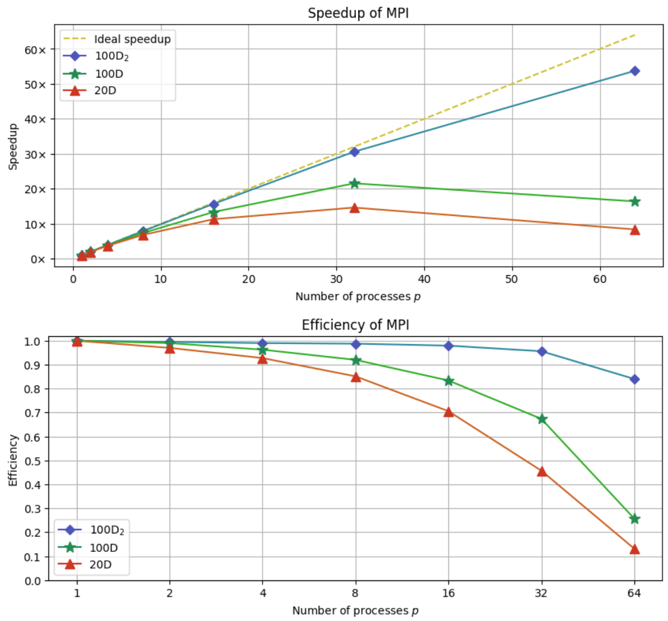

Fig. 1. Speedup (top image) and efficiency (bottom image) of the MPI application with three different tests

the largest input size ($100D_2$), the speedup continues to increase steadily up to 32 processes and maintains a near-linear trend even up to 64 processes. This indicates that for large-scale problems, the computation-to-communication ratio remains favorable, allowing the system to scale efficiently. However, for the smaller input sizes (100D and 20D), the speedup begins to decrease after 32 processes, as can be seen in figure II-A3. This drop in performances is primarily due to the increasing overhead of inter-process communication and memory exchange. As the number of processes grows, the amount of data each process handles becomes smaller, and the cost of communication dominates, making parallelism not efficient as we can see in table III for MPI.

Considering weak and strong scaling, the program exhibits both characteristics to some extent. It demonstrates strong scaling effectively up to 16 processes, particularly for larger problem sizes, where the speedup remains close to linear as more processes are added ($100D_2$). However for smaller input sizes the overhead from inter-process communication begins to outweigh the benefits of parallelism, leading to diminishing returns. In terms of weak scaling, the program performs well for larger input sizes, as it maintains relatively consistent performance when both the problem size and the number of processes increase proportionally.

The MPI implementation is well-suited for handling large-scale computations where the computation-to-communication ratio remains high. Overall, the MPI program scales efficiently, but its scalability is ultimately bounded by communication overhead at higher process counts for smaller datasets.

III. Parallelizing with PThread

POSIX Threads (**PThreads**) is an API defined by the POSIX standard, providing a powerful and flexible framework for parallel programming on shared memory systems. PThreads give developers low-level control over threads creation, synchronization, and data sharing. Each thread represents an independent flow of execution, and all threads within a process share the same memory space, making inter-thread communication efficient but also more complex to manage. PThreads require explicit management of thread lifecycle, workload division, synchronization primitives (like mutexes, condition variables, and barriers), and memory consistency. While this manual control allows for highly customized parallel solutions, it also increases the potential for bugs such as race conditions or deadlocks if not handled carefully.

A. *PThread Implementation Approach*

For implementing the PThread version, we aimed at creating a unique function, which would contain the main loop of the algorithm. In order to do that, we made use of the PThread barriers, which can be optionally activated through a flag in the program. In order to pass the parameters to the threads, two structs have been created, one for global parameters (whose pointer is shared with all threads) and one for local parameters. In order to split the workload, each thread first gets assigned a quantity of data points and centroids; this quantity is computed as follows:

$$\texttt{local\_n} = \left\lfloor \frac{n}{t} \right\rfloor \qquad \texttt{local\_k} = \left\lfloor \frac{k}{t} \right\rfloor$$

After that, if there are any remaining data points/-centroids (so if there exists a remainder of the previous division which is not 0), they get assigned to multiple threads in a round-robin fashion. This computation is done by the program before launching the threads.

In the kernel, if a thread has a `local_n` > 0, then it can participate to the computation of the first step of the algorithm, so assigning the points to a cluster. If the thread doesn't participate, it skips the step and waits for all other threads. After the first step, a barrier will make sure that all threads have completed the tasks assigned so far; then, through a lock, the points per class and the coordinates of the auxiliary centroids that were computed locally get summed.

Similarly for the first step, only all threads with `local_k` > 0 can participate to the computation of the second step. In the second step each thread computes the new centroids, first by averaging the coordinates of the assigned auxiliary centroids, and then by computing the new distances (for the convergence check). After that, a barrier will wait for all threads to finish, and then through a second lock the convergence parameters get updated.

B. PThread Performance analysis

The PThread program was tested with 1, 2, 4, 8, 16, 32 and 64 threads. The runtimes can be observed both in figure III-B and in table IV. The speedup and the efficiency of the PThread implementation reach high values, especially with high quantities of data: indeed, the speedup is nearly linear even with 32 threads with the test $100D_2$ (by table V-C, we have 31.28×) The plots in figure III-B help highlighting the strong scaling of the application. Similarly to the MPI application, after 32 threads with the smaller tests (20D and 100D) the application performances (both the efficiency and speedup) start decreasing. The main reason of this is due to the overhead of instantiating multiple threads which do too few operations, and that need to synchronize in multiple parts of the application. We can also notice some properties of weak scalability: by the augmentation of the number of threads and of the sizes of the problem, the speedup and efficiency increase as well, and this can be ultimately noticed by the values in table V-C.

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. Clearly, this can be applied also to the k-means algorithm, and we'll now proceed to illustrate our proposed model.

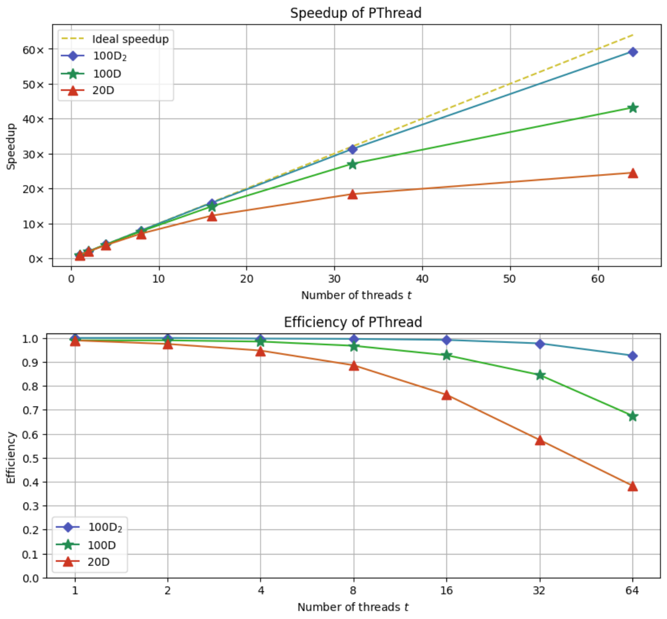

Fig. 2. Speedup (top image) and efficiency (bottom image) of the PThread application with three different tests

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, and this is because of the data needed by 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 which distributes the points over the threads, and another that distributes the centroids. This would create in total three parallelization steps. In our implementation, such steps are developed in the following three kernels: `assignment_step`, `update_step_points` and `update_step_centroids`.

B. Program parameters, custom `atomicMax` and memory management

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 is dynamically determined depending on

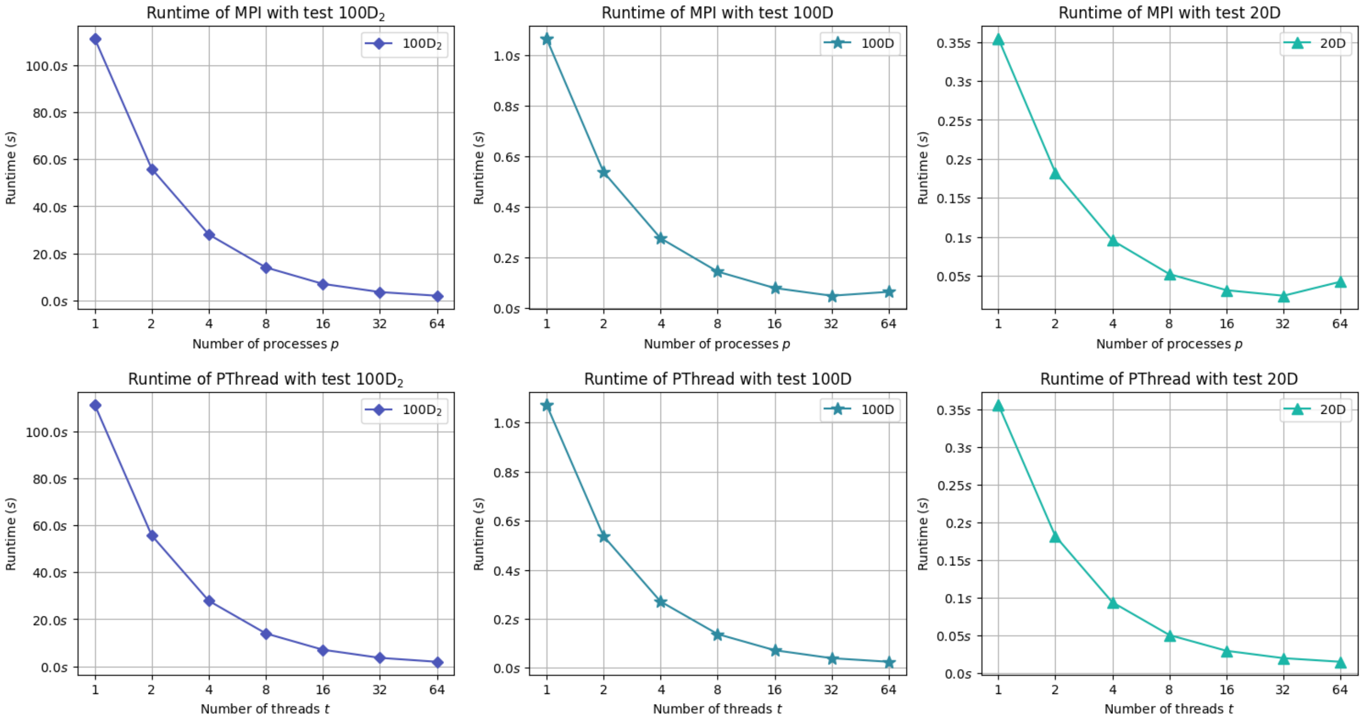

Fig. 3. Runtimes of the MPI and PThread applications with three different tests

the called kernel: usually, we have that $|X| \gg k$, 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. The grids' dimensions are computed following these formulas:

$$\texttt{points\_grid\_size} = \frac{|X|}{32 \times 32} + 1$$

$$\texttt{centroids\_grid\_size} = \frac{k}{32 \times 32} + 1$$

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:

```
__device__ float custom_atomicMax(float* value_address, float val) {
    int* address_as_int = (int*) value_address;
    int old = *address_as_int, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_int, assumed, __float_as_int(fmaxf(val, __int_as_float(assumed))));
    } while (assumed != old);
    return __int_as_float(old);
}
```

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.

Especially with CUDA, a great part of the possible sppedup that an application can achieve is given by the memory accesses. One of the many possibilities that CUDA offers, that is not instead possible with CPUs (be it with multi-processing or multi-threading programming), is to **freely manage** the **L1 cache** of the GPU (also known as the shared memory). In our implementation, we made sure to use the shared memory as much as possible, and to transfer the data to the global memory only when needed. Moreover, we made sure to use most efficiently the **memory coalescing** that CUDA does whenever retrieving data from the global memory: this has been done by "transposing" the `data` array. Normally, the `data` array would be organized as follows:

$$
\begin{array}{r|c|c|c|c|c|c|l}
\Longrightarrow & p_{1,1} & p_{1,2} & p_{1,3} & \cdots & p_{1,d-1} & p_{1,d} & \hookleftarrow_1 \\
\hookrightarrow_1 & p_{2,1} & p_{2,2} & p_{2,3} & \cdots & p_{2,d-1} & p_{2,d} & \hookleftarrow_2 \\
 & & \vdots & & \ddots & & \vdots & \\
\hookrightarrow_{n-1} & p_{n,1} & p_{n,2} & p_{n,3} & \cdots & p_{n,d-1} & p_{n,d} &
\end{array}
$$

where the notation $p_{i,j}$ stands for the j^{th} coordinate of the i^{th} point; in other words, the array was originally stored point-wise (so first all the d coordinates of the first point, then the d coordinates of the second point, and so on...). After transposing the array, it became as follows:

$$
\begin{array}{r|c|c|c|c|c|c|l}
\Longrightarrow & p_{1,1} & p_{2,1} & p_{3,1} & \cdots & p_{n-1,1} & p_{n,1} & \hookleftarrow_1 \\
\hookrightarrow_1 & p_{1,2} & p_{2,2} & p_{3,2} & \cdots & p_{n-1,2} & p_{n,2} & \hookleftarrow_2 \\
 & & \vdots & & \ddots & & \vdots & \\
\hookrightarrow_{d-1} & p_{1,d} & p_{2,d} & p_{3,d} & \cdots & p_{n-1,d} & p_{n,d} &
\end{array}
$$

thus having a data array where the **coordinates** are **stored dimension-wise** (so first the first dimension of all the points, then the second dimension, and so on...). This is preferrable since, when computing the new centroids, all threads will work "simultaneously" on the same dimension of the data, so when recovering the data coordinates from the global memory the GPU will load a burst of coordinates who belong to the same dimension together. This, as we'll show later on, **improves greatly** the speedups and timings of the application. While this may have been done also on the centroids, it is not strictly necessary, since the centroids will be already stored in the shared memory when needed.

C. Analysis of the kernels

As we mentioned previously, we are making use of three kernels: `assignment_step`, `update_step_points` and `update_step_centroids`. In all kernels, all operations that act on the global memory are **performed atomically**, which avoid potential race conditions. We here show how all three kernels work.

The first kernel is called on all the data, and each point is assigned to a thread. In this kernel, the centroids are stored in the shared memory. 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 between its assigned point and all centroids, and will store the class to which each point will be assigned into the `class_int` variable, depending on which ℓ_2 norm is the smallest. If a change from the previous class assignment is detected, the number of changes will increase.

TABLE II
PERFORMANCES OF THE CUDA VERSION $(\mu \pm \sigma^2)$

Metric	Input tests		
	20D	100D	$100D_2$
Runtime^{1a}	0.05633 ± 0.02272	0.12794 ± 0.03329	8.46844 ± 1.29596
Speedup^{1a}	6.28×	8.32×	13.14×
Runtime^{1b}	0.038851 ± 0.00758	0.03343 ± 0.00402	1.25401 ± 0.00367
Speedup^{1b}	9.11×	31.86×	88.74×
Runtime^{2b}	0.00545 ± 0.00006	0.00993 ± 0.00004	0.50741 ± 0.00337
Speedup^{2b}	35.92×	61.9×	126.83×
Runtime^{3b}	0.00522 ± 0.00008	0.00873 ± 0.00008	0.16875 ± 0.00018
Speedup^{3b}	67.82×	121.99×	659.44×

$*^a$: **without** a correct handling of memory coalescing + **without** using the shared memory
$*^b$: **with** a correct handling of memory coalescing + **using** the shared memory

$*^1$: executed on an **RTX Quadro 6000**
$*^2$: executed on a **GeForce RTX 4060**
$*^3$: executed on a **GeForce RTX 5080**

The second kernel will have a point assigned to each thread, and each thread will increment, in the `points_per_class` array, the value in the position corresponding to the class assigned to the point. After that, the coordinates of the point are added to the `aux_centroid` array, so that to create, on the new step, the new centroid from the average of the coordinates. Here, both the `points_per_class` and `aux_centroids` arrays are stored in shared memory, and only after a thread computes its contribution to the final array the result in shared memory gets copied over to the global memory (also here it is done through atomic operations, so that to avoid concurrency problems).

The third kernel, `update_step_centroids`, firstly ensure that all threads are mapped to a valid centroid; then all threads will compute, for each dimension of the centroids, the average coordinates of the new centroid. Once done, each thread will then perform the ℓ_2 norm between the previous centroid and the new one, so that to compute the

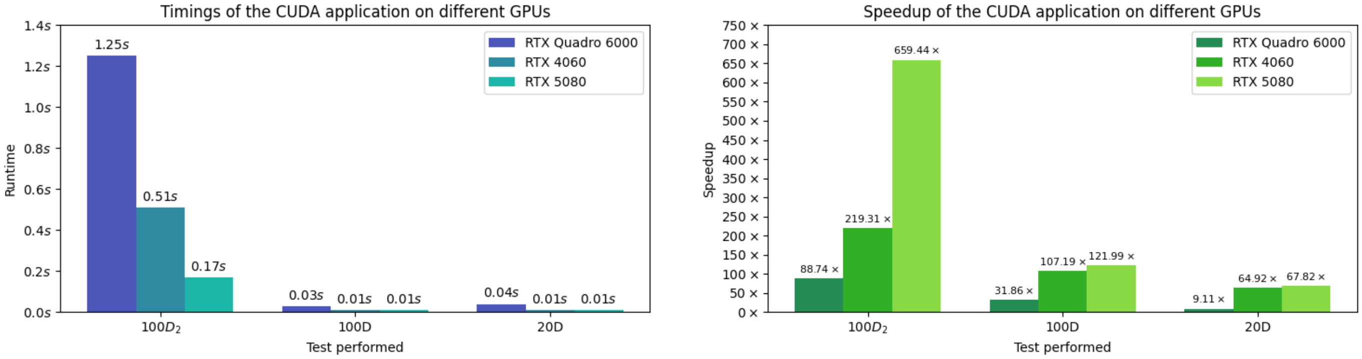

Fig. 4. Runtimes and speedups of the CUDA application on different GPUs

`max_distance` variable, needed for convergence. Once again, the variable is first stored in shared memory, and it's then stored in global memory via the use of the `custom_atomicMax()` function, which we illustrated previously.

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

D. CUDA Performance analysis

The CUDA version performed as can be seen from table IV-C (where both runtime and speedup are reported). As we mentioned in the previous section, the **correct handling** of the **memory** and **exploitation** of the **memory bursts** allows to reach high speedups of the application: the difference can be seen for the rows with the $*^{1a}$ and $*^{1b}$ apices (the former denotes a version of the application that doesn't handle memory correctly, the latter instead does). We can see that, on the same system, a correct memory handling could grant, with this application, an improvement on the speedup of ~ 76×.

We will show however that the speedup is **not** the **most impactful statistic** for determining the performance of a CUDA application: in order to show this, we also sampled the performance of the application in other systems with other NVIDIA GPUs, such as an RTX 4060 (marked by $*^2$) and an RTX 5080 (marked by $*^3$). In such cases, the speedup reported is relative to the sequential timings obtained on the cluster mentioned in section *1.C*. The reason why we compared the CUDA timings to a single sequential version is to simulate how the code would run if we just changed the GPU and left the rest of the system intact (of course, the test would be appropriate if we could directly change the hardware on the cluster, but this wasn't a possibility). The table shows moreover how the speedup varies greatly depending on the GPU is has been executed on, reaching higher numbers on newer architectures. This can also be better seen from figure IV-C.

In order to better analyze the application, we carried out, thanks to the Nsight Compute profiler, an analysis of the **roofline model** of our application, which has been reported in figure IV-D. We were only able to retrieve the roofline model from the systems with the RTX 4060 and 5080 GPUs. The profiler measured the performance of the L1 and VRAM memory. From the model we can see that the application is still memory bound (both related to the L1 and VRAM memory) with both GPUs; while the performance is still high, in order to improve the application it could be useful to perform more operations on the data, so that to move the application from the memory-bound region to the compute-bound one. Moreover, an improvement that could be made is to use more the data stored in the L1 cache; however, given the nature of the algorithm, only a few operations are performed on the different data used (be it the array of points, the array of centroids, etc...), so it becomes hard to increase the arithmetic intensity without drastically changing the algorithm.

V. Interlacing Multi-processing and 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

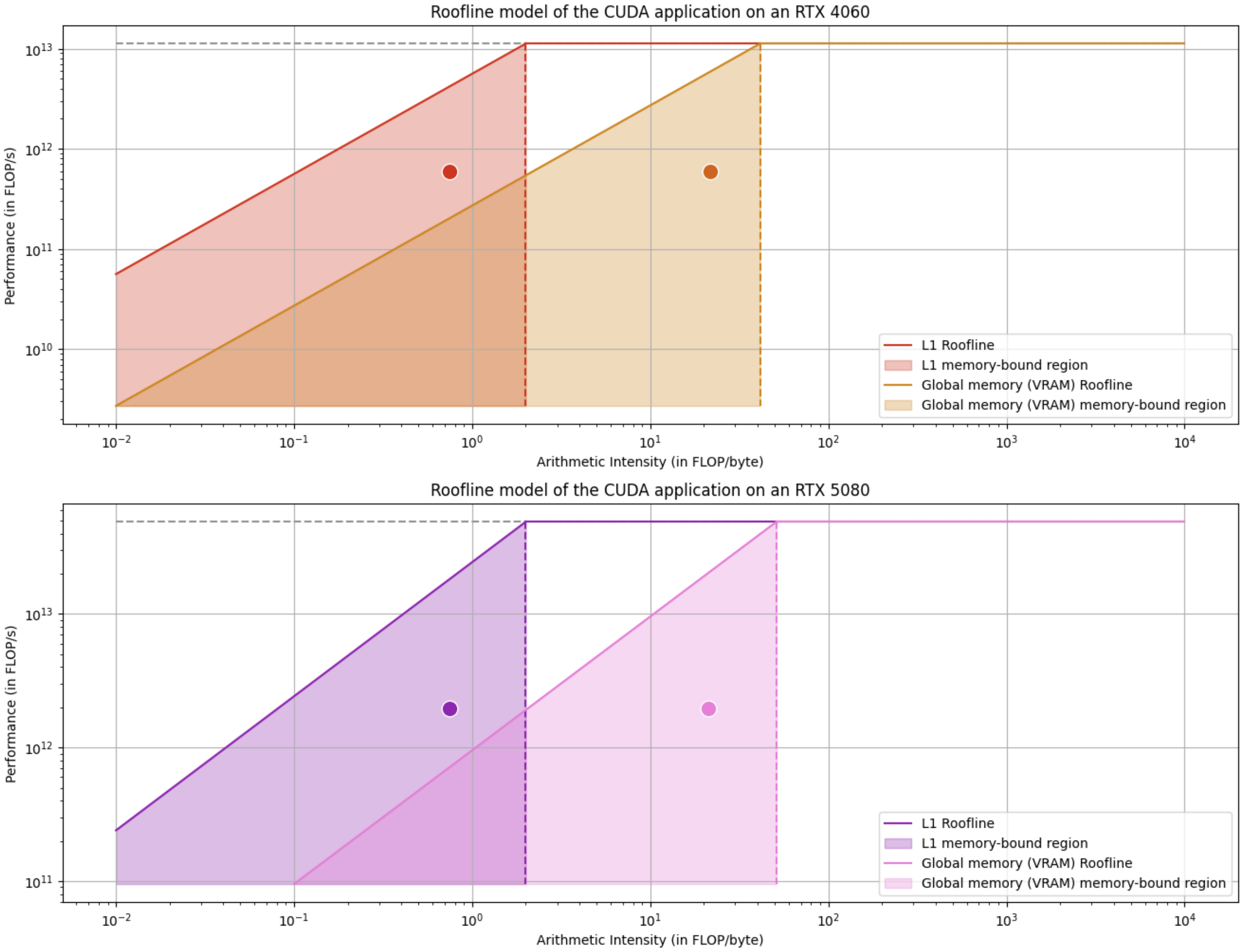

Fig. 5. Roofline model of the CUDA application on an RTX 4060 GPU (on top) and on a RTX 5080 GPU (on bottom)

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 multiprocessing library MPI, and this will be done thanks to the OpenMP library.

The **Open Multi-Processing** (**OpenMP**) API is a standardized and portable framework designed to simplify parallel programming on shared memory systems, such as multi-core processors within a single machine. It enables efficient execution by allowing multiple threads to access a shared memory space. Threads can operate on either shared data or private copies, depending on the task. OpenMP achieves parallelism through simple compiler directives, allowing developers to parallelize sections of code without the complexity of manually managing threads, synchronization, or workload distribution.

We implemented a version of the k-means algorithm that leverages both MPI and OpenMP to maximize performance. This hybrid approach is commonly used in cluster environments consisting of multiple nodes without shared memory. Within each node, OpenMP takes advantage of the shared-memory multiprocessor architecture, while MPI handles communication and data distribution across nodes.

A. *Implementation approach*

The k-means clustering program follows a Globally Parallel, Locally Sequential (GPLS) model. In this approach, the critical initialization and termination tasks are handled by rank 0, while the core looping part of the algorithm is fully parallelized. MPI is used to distribute the data and computation across nodes,

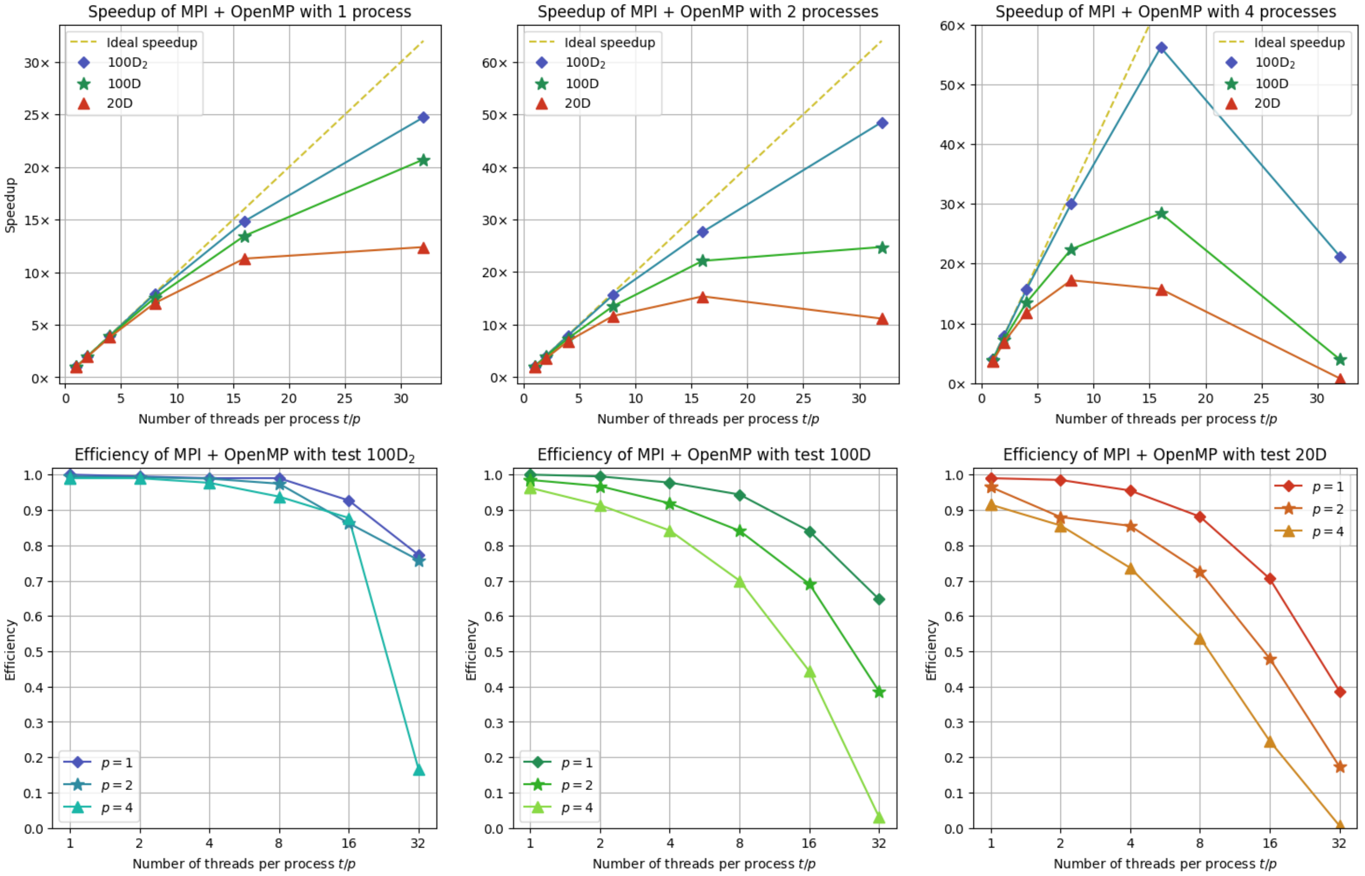

Fig. 6. Comparison of the speedup of the MPI + OpenMP program (on top) and the efficiency (on bottom), on three different input tests. Each plot compares the program on a set amount of processes

and within each node, OpenMP is used to parallelize the workload across multiple threads.

1) Initialization phase: In the initialization phase is where workload balancing decisions are made. We initialize MPI with `MPI_Init_thread` using `MPI_THREAD_FUNNELED` to ensure thread safety for MPI calls, then we set the number of OpenMP threads per MPI process using `omp_set_num_threads`. 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.

2) Loop phase: The loop phase involves two key operations that are parallelized over different data structures:

- **point assignment**: each process works on its local points with its local `classMap`. This assignment step is fully parallel, as each MPI process handles its portion of the points. Each MPI process independently spawns OpenMP threads by using

  ```
  #pragma omp parallel for
  ```

  which parallelize the loop over local points and track local cluster assignment changes with `reduction(+: local_changes)`. Assignment changes are then reduced globally using `MPI_Iallreduce` for checking at the and the termination conditions;
- **centroid update**: after points are assigned to clusters, each MPI process computes local partial sums for the centroids and the number of points in each cluster. Within each process, OpenMP threads are spawned using `#pragma omp parallel` for to parallelize the loop over local points. This is done with the following reduction clause, so that to ensure thread-safe accumulation:

  ```
  reduction(+: auxCentroids[:K*D],
            pointsPerClass[:K])
  ```

Once local computations are complete, the partial results from all MPI processes are aggregated using MPI_Allreduce, ensuring globally consistent centroid values with fast communication.

After collecting the distances to update centroid positions, each MPI process is responsible for a subset of centroids and performs local updates based on the globally reduced values. To ensure efficient parallelism within each process, OpenMP is used to parallelize the computation of centroid displacements. Specifically, a pragma omp parallel for reduction(max: local_maxDist) directive is used so that multiple threads can compute the maximum displacement among local centroids in parallel.Once the local maximum displacement (local_maxDist) is computed via OpenMP, an MPI_Allreduce operation with MPI_MAX is used to determine the global maximum displacement (global_maxDist) across all processes. This value is then used to evaluate convergence based on defined termination conditions, such as maximum number of iterations, minimal centroid movement (threshold), or no significant changes. Following the convergence check, each process uses MPI_Allgatherv to gather all updated local centroids into a global centroid array. This ensures that all processes have the complete set of centroids for the next iteration, maintaining consistency across the distributed environment.

3) 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.

B. MPI + OpenMP Performance analysis

When running the program with 1 MPI process, the setup effectively becomes a pure OpenMP implementation. In this case, we observe near-linear scaling up to 16 threads. At 32 threads, performance begins to drop slightly in efficiency, though it still scales well-especially for the larger input sizes (100D and $100D_2$). This drop is likely due to false sharing, a common memory access issue in multi-threaded programs. Although false sharing could be mitigated by padding shared data structures, we chose not to apply this optimization, since the main focus of this application is to scale across multiple MPI processes.

When using 2 or 4 MPI processes, we notice a similar scaling trend. Specifically: with 2 MPI processes and 32 threads total (16 threads per process) running 100D and $100D_2$ tests, performance remains efficient; with 4 MPI processes and 16 threads total (4 threads per process) running 100D and 100D2 tests also maintain good efficiency. In both cases, the parallel workload reaches 64 total threads, which seems to be peak for the given problem sizes. However, when increasing to 4 MPI processes with 32 threads each (totaling 128 threads), performance drops completely. This is likely due to the increased overhead of thread creation and inter-process communication, which becomes less efficient for smaller input sizes. This behaviour can be observed from figure IV-D and table V-C. At this scale, the computational workload per thread is no longer enough to justify the overhead, leading to reduced speedup and efficiency. This is not due to the OpenMP implementation as we saw that it works effectively with 1 MPI process.

With 1 MPI process (pure OpenMP), strong scaling is evident up to 16 threads, with diminishing returns at 32 threads due to false sharing. With 2 and 4 MPI processes, strong scaling remains effective, especially for 100D and 100D2, indicating that adding more resources (threads/processes) improves runtime up to a point. However, at 128 total threads (4 MPI × 32 threads), strong scaling breaks down. The problem size per thread becomes too small, and overhead (thread management + inter-process communication) dominates. In terms of weak scaling, the program performs well for larger input sizes, as it maintains relatively consistent performance when both the problem size and the number of processes and threads increase proportionally.

C. Conclusion

The hybrid MPI + OpenMP implementation is well-suited for mid-to-large-scale parallel configurations, where the workload per thread remains significant. Overall, the program demonstrates efficient scaling, particularly up to 64 total threads. However, its scalability is ultimately limited by thread management and communication overhead at higher thread counts, especially when the problem size does not increase proportionally.

References

[1] K-means assignment. https://trasgo.infor.uva.es/sdm_downloads/k-means/. Accessed: 2025-01-29.

TABLE III
AVERAGE EXECUTION TIMES OF MPI $(\mu \pm \sigma^2)$

Processes	**Input files (in dimensions)**		
	20D	100D	100D$_2$
1	0.35482 ± 0.00039	1.06557 ± 0.00096	111.25387 ± 0.11351
2	0.18259 ± 0.00432	0.53813 ± 0.00434	55.92949 ± 0.11749
4	0.09535 ± 0.00171	0.27635 ± 0.00172	28.07182 ± 0.12835
8	0.052 ± 0.00045	0.14474 ± 0.0007	14.08863 ± 0.05404
16	0.03135 ± 0.00039	0.07982 ± 0.00039	7.10323 ± 0.02457
32	0.02421 ± 0.00625	0.04941 ± 0.01561	3.63824 ± 0.01364
64	0.04227 ± 0.00405	0.06498 ± 0.00777	2.06993 ± 0.01525

TABLE IV
AVERAGE EXECUTION TIMES OF PTHREAD $(\mu \pm \sigma^2)$

Threads	**Input files (in dimensions)**		
	20D	100D	100D$_2$
1	0.35589 ± 0.00116	1.07345 ± 0.00179	111.49448 ± 0.08208
2	0.18134 ± 0.00051	0.5372 ± 0.00025	55.72216 ± 0.02585
4	0.09344 ± 0.0002	0.27037 ± 0.00011	27.87802 ± 0.0107
8	0.04993 ± 0.00015	0.13759 ± 0.000005	13.96211 ± 0.01513
16	0.029 ± 0.00015	0.07174 ± 0.00024	7.01285 ± 0.00178
32	0.01924 ± 0.00026	0.03934 ± 0.00015	3.5571 ± 0.00333
64	0.01443 ± 0.00048	0.02464 ± 0.00028	1.87593 ± 0.00213

TABLE V
AVERAGE EXECUTION TIMES OF MPI + OPENMP $(\mu \pm \sigma^2)$

Processes	**Threads**	**Input files (in dimensions)**		
		20D	100D	100D$_2$
1	1	0.35709 ± 0.00048	1.06757 ± 0.0008	111.25387 ± 0.11351
	2	0.18002 ± 0.0003	0.53609 ± 0.00047	55.93932 ± 0.0679
	4	0.09277 ± 0.00033	0.27239 ± 0.00061	28.08522 ± 0.06347
	8	0.05018 ± 0.00046	0.14105 ± 0.00067	14.04603 ± 0.02584
	16	0.03136 ± 0.0005	0.07924 ± 0.00436	7.5028 ± 0.09568
	32	0.02858 ± 0.00497	0.05141 ± 0.00269	4.50071 ± 0.18373
2	1	0.18372 ± 0.00448	0.53968 ± 0.00436	55.92949 ± 0.11749
	2	0.1006 ± 0.02559	0.27507 ± 0.0028	28.04651 ± 0.06814
	4	0.05179 ± 0.001	0.14482 ± 0.00225	14.07429 ± 0.04251
	8	0.0305 ± 0.00054	0.0791 ± 0.00166	7.13597 ± 0.02573
	16	0.02306 ± 0.00071	0.04817 ± 0.00093	4.0312 ± 1.48805
	32	0.03185 ± 0.00557	0.04304 ± 0.0034	2.2939 ± 0.12168
4	1	0.09674 ± 0.0014	0.27689 ± 0.00169	28.07182 ± 0.12835
	2	0.05165 ± 0.00107	0.14576 ± 0.00346	14.04567 ± 0.04577
	4	0.03005 ± 0.00056	0.07905 ± 0.00093	7.11934 ± 0.07096
	8	0.02056 ± 0.00046	0.04754 ± 0.00203	3.70751 ± 0.12491
	16	0.02249 ± 0.00256	0.03749 ± 0.00331	1.98168 ± 0.05976
	32	0.45262 ± 0.01723	0.26914 ± 0.00238	5.24488 ± 0.03644

TABLE VI
SPEEDUPS OF MPI AND PTHREAD

MPI

Processes	Input files (in dimensions)		
	20D	100D	100D$_2$
1	0.998×	0.999×	1.0×
2	1.939×	1.979×	1.99×
4	3.713×	3.854×	3.964×
8	6.807×	7.358×	7.898×
16	11.291×	13.341×	15.666×
32	14.618×	21.554×	30.586×
64	8.373×	16.389×	53.759×

PTHREAD

Threads	Input files (in dimensions)		
	20D	100D	100D$_2$
1	0.995×	0.992×	0.998×
2	1.952×	1.982×	1.997×
4	3.788×	3.939×	3.992×
8	7.089×	7.74×	7.97×
16	12.206×	14.845×	15.868×
32	18.395×	27.07×	31.283×
64	24.537×	43.226×	59.319×

TABLE VII
SPEEDUPS AND EFFICIENCY OF MPI-OPENMP

SPEEDUPS

Processes	Threads	Input files (in dimensions)		
		20D	100D	100D$_2$
1	1	0.991×	0.998×	1.0×
	2	1.966×	1.987×	1.989×
	4	3.815×	3.91×	3.962×
	8	7.054×	7.55×	7.922×
	16	11.287×	13.439×	14.832×
	32	12.385×	20.714×	24.725×
2	1	1.927×	1.973×	1.99×
	2	3.519×	3.872×	3.968×
	4	6.835×	7.354×	7.907×
	8	11.606×	13.463×	15.594×
	16	15.353×	22.109×	27.604×
	32	11.114×	24.745×	48.511×
4	1	3.659×	3.846×	3.964×
	2	6.854×	7.306×	7.923×
	4	11.78×	13.473×	15.63×
	8	17.22×	22.401×	30.014×
	16	15.743×	28.407×	56.154×
	32	0.782×	3.957×	21.217×

EFFICIENCY

Processes	Threads	Input files (in dimensions)		
		20D	100D	100D$_2$
1	1	0.99	1.0	1.0
	2	0.985	0.995	0.995
	4	0.955	0.9775	0.99
	8	0.88125	0.94375	0.99
	16	0.70562	0.84	0.92688
	32	0.38719	0.64719	0.7725
2	1	0.965	0.985	0.995
	2	0.88	0.9675	0.9925
	4	0.855	0.91875	0.98875
	8	0.72562	0.84125	0.97438
	16	0.47969	0.69094	0.8625
	32	0.17359	0.38656	0.75797
4	1	0.915	0.9625	0.99
	2	0.85625	0.91375	0.99
	4	0.73625	0.84188	0.97688
	8	0.53812	0.7	0.93781
	16	0.24594	0.44391	0.87734
	32	0.00609	0.03094	0.16578

TABLE VIII
EFFICIENCY OF MPI AND PTHREAD

MPI

Processes	**Input files (in dimensions)**		
	20D	100D	$100D_2$
1	1.0	1.0	1.0
2	0.97	0.99	0.995
4	0.9275	0.9625	0.99
8	0.85125	0.92	0.9875
16	0.70562	0.83375	0.97938
32	0.45688	0.67344	0.95594
64	0.13078	0.25609	0.84

PTHREAD

Threads	**Input files (in dimensions)**		
	20D	100D	$100D_2$
1	0.99	0.99	1.0
2	0.975	0.99	1.0
4	0.9475	0.985	0.9975
8	0.88625	0.9675	0.99625
16	0.76312	0.92812	0.99188
32	0.57469	0.84594	0.9775
64	0.38344	0.67547	0.92688