Parallel Kmeans-based Image Colors Compressor

Leonardo Pagliochini[†] **and Francesco Rosnati**[†]

[†]*Master's Degree Students in* ***High-Performance Computing*** *Engineering at* ***Politecnico di Milano***
[†]*The authors contributed equally to this work*

September 6, 2024

Abstract

This project explores the *parallel* implementation of the *KMeans clustering* algorithm for the lossy *compression* of natural images. KMeans is a widely used clustering technique that partitions data into a given number K of clusters. In the context of *image compression* KMeans is employed to reduce the color palette by grouping similar colors, possibly minimizing the data required to represent the image. This study uses image compression as an illustrative application to focus on the primary objective: implementing in *C++* the KMeans algorithm using various parallel computing frameworks, in order to evaluate them against a baseline sequential version under different *compression settings*. The framework used include *OpenMP*, *MPI*, *CUDA*. Additionally, the project examines the impact of different *color models* on compression efficiency, implements a *binary encoding* of the clustered image. The goal is not to outperform established compression algorithms but to analyze the computational *efficiency* and *scalability* of the KMeans algorithm across different parallel environments.

1. Introduction

Modern era of *data-driven decision-making* and *large-scale data analysis*, still finds in clustering a fundamental technique for organizing data into meaningful structures. Clustering involves the *unsupervised classification* of patterns—such as observations, data items, or feature vectors—into groups, commonly referred to as clusters. Due to its broad appeal and utility as a critical step in exploratory data analysis, the clustering problem has been extensively studied across various fields. However, clustering poses significant challenges as a *combinatorial optimization problem*, especially in *high-dimensional spaces*. In these spaces, the "curse of dimensionality" often leads to distance metrics becoming less effective, and determining the *structure of clusters* can become computationally intensive[1].

The primary objective of clustering is to group objects with similar characteristics, and it has a wide range of applications, including feature extraction, data compression, dimensionality reduction, and vector quantization. The quality of a clustering solution is often application-dependent, varying according to the specific requirements and goals of the task at hand. Numerous clustering methods have been developed, ranging from neural network-based techniques to splitting-merging approaches and distribution-based methods. Among these, K-Means clustering still remains to this day one of the most popular centroid-based algorithms. Its popularity stems from its simplicity, computational efficiency, and effectiveness in minimizing within-cluster variance, making it a preferred choice in many practical applications [2].

While K-Means clustering has various practical applications, this study primarily focuses on exploring the parallel implementation of the K-Means algorithm in C++ using different parallel computing frameworks, including OpenMP, MPI, and CUDA. The main objective is to evaluate the computational efficiency and scalability of K-Means clustering when parallelized under different frameworks and compression settings. To illustrate the effectiveness of these parallel implementations, image compression is selected as an example application.

K-Means clustering can be employed for image compression, which involves reducing the file size of an image while preserving its visual quality. The algorithm accomplishes this by reducing the number of distinct colors in the image through a process known as *image quantization*. By identifying a specified number k of representative colors that best capture the image's overall color palette, K-Means reduces the palette from a wide range to a smaller, more manageable set. The effectiveness of K-Means in image compression lies in its ability to minimize *approximation error* by selecting the optimal k colors that represent a target image [4].

The remainder of this document is structured as follows: the Methods section details the implementation of K-Means clustering for lossy image compression, discussing various technical considerations, including color models, binary encoding schemes, and compression settings, as well as relevant libraries and code structure. The Evaluation section presents experimental results focusing on computational performance across different parallel environments. Finally, the Conclusion and Future Developments sections summarize the key insights of the study and propose potential directions for future research and development.

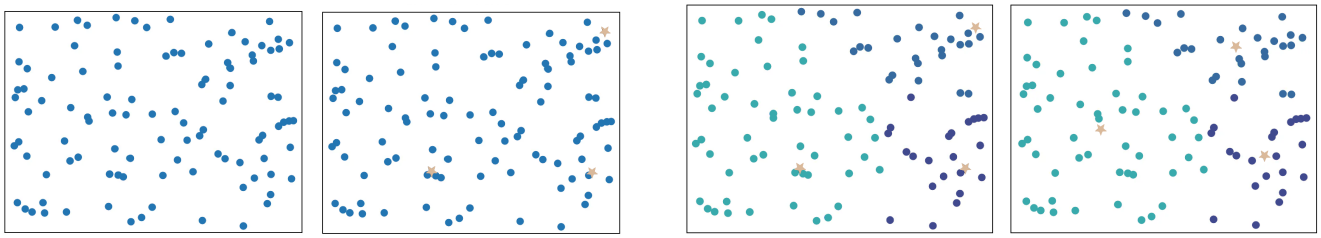
2. Methods

2.1. K-Means Algorithm

The K-Means algorithm is an *unsupervised machine learning* method used for grouping data into k clusters, where each cluster is represented by its centroid. This goal is achieved by partitioning n observations into k clusters, with each observation assigned to the cluster whose centroid is closest, as determined by a chosen distance metric.

The algorithm begins with an initialization phase, where k initial cluster centers (means) $\{\mu_1^{(0)}, \mu_2^{(0)}, \dots, \mu_k^{(0)}\}$ are chosen. After initialization, the algorithm enters an iterative phase, often referred to as

Figure 1. Compressed image for different k color number choices: can you tell which is the original?



(a) On the left the **initial points**, on the right the **starting random centroids**

(b) Assign points to nearest cluster, compute **new centroids**

Figure 2. Kmeans algorithm in action

Lloyd's algorithm, which repeatedly executes two main steps until convergence is reached. The first step, known as the *assignment step*, assigns each data point to the nearest centroid based on a chosen distance metric. For each observation x_j, the cluster S_i is determined such that:

$$S_i^{(t)} = \{x_j : \|x_j - \mu_i^{(t)}\|^2 \leq \|x_j - \mu_l^{(t)}\|^2, \forall l, 1 \leq l \leq k\} \quad (1)$$

The *Euclidean distance* is the most commonly used metric; however, other distance metrics, such as *Manhattan* or *cosine distance*, may be employed depending on the application's specific requirements.

Following the *assignment step*, the second step, called the *update step*, recalculates the centroids by computing the mean of all data points assigned to each cluster.

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \quad (2)$$

The algorithm continues to alternate between the *assignment* and *update steps* until convergence is achieved.

In this implementation, convergence is determined by monitoring whether *any data points change their assigned centroid between iterations*. The algorithm terminates when no data points switch clusters compared to their previous assignment, indicating that a stable clustering solution has been found.

Algorithm 1 K-Means Clustering Algorithm

1: **Input:** Dataset D, Number of clusters K, Tolerance ϵ
2: **Output:** Mean table, Assignment of each datapoint to a cluster
3: Initialize the means $\mu_1, \mu_2, \ldots, \mu_K$ randomly
4: **repeat**
5: Assign each data point to the nearest cluster based on the current means
6: Calculate new means for each cluster
7: Check if any data point has changed its assigned cluster
8: **if** No points have changed clusters **then**
9: **Output** assignment of data
10: **return**
11: **else**
12: Continue from step 4 with new means
13: **end if**
14: **until** no points change clusters between iterations

Computational complexity of the K-Means algorithm depends on the steps performed in each iteration. The *assignment step*, which computes the distance between each data point and each centroid, has a time complexity of $O(nKd)$, where n is the number of data points, K is the number of clusters, and d is the dimensionality of the data. This step is generally the most computationally intensive part of the algorithm. The *update step*, which recalculates the centroids, has a time complexity of $O(nd)$. The evaluation of the convergence criterion, which checks for changes in cluster assignments, also requires $O(n)$ computations. Due to the repeated computation of distances, the *assignment step* tends to dominate the overall computational cost [3].

The K-Means algorithm is highly scalable and well-suited for parallelization, which is a significant advantage when dealing with large datasets. By distributing the computational load across multiple processors, the algorithm can handle extensive data more efficiently, reducing overall runtime.

2.2. Kmeans-based image color compressor

Image compression is a critical process in reducing the file size of digital images while preserving as much of the original quality as possible. One effective approach for achieving this is through color quantization, which involves *reducing the number of distinct colors* used in an image.[4]

The idea behind using K-means for image color compression is to *decompose the color information of an image into a multidimensional space*, where each axis represents one of the primary colors of the chosen color model. For example, in the RGB color model, this space is three-dimensional, with the axes corresponding to the Red, Green, and Blue color channels. Each pixel in the image can be represented as a point in this three-dimensional space, with its position determined by its red, green, and blue intensity values.

This concept is not limited to RGB; it applies to other color models as well. For instance, in the CMYK model, the space would be four-dimensional, with axes representing the Cyan, Magenta, Yellow, and Key (Black) channels. Regardless of the color model, K-means clustering is used to *group pixels based on their Euclidean distance* in the corresponding color space. Each pixel is then reassigned to the nearest centroid color, *"approximating" its original color, thus achieving compression by reducing the number of distinct colors used* in the image.

How is the compression achieved?

In the standard RGB color model, each pixel is represented by $24bits$: $8bits$ each for red, green, and blue channels. However, when color quantization is applied to an image with a specified number of colors k, the representation changes significantly. Instead of needing 24 bits per pixel, the color information can be encoded using $\log_2(k)$ bits per pixel. For example, if $k = 16$, each pixel can be represented using only 4 bits, leading to a substantial reduction in the image size. This approach offers a straightforward yet effective way to achieve compression, especially when the image contains a limited range of colors or well-defined color clusters.[4]

It is important to note that K-means-based compression is a form of *lossy compression*. Unlike *lossless compression techniques*, where the original data can be perfectly reconstructed, *lossy compression* involves some level of data loss. In the context of K-means, each pixel's color is approximated by the nearest centroid among the k chosen colors. This approximation inevitably leads to a *loss of some*

color information, making the compression irreversible. The degree of perceptible data loss is often minimal when the number of clusters k is adequately chosen, but it can become noticeable if k is too low, resulting in a more significant approximation error.[4]

An inherent aspect of the K-means algorithm is the randomness involved in the initialization of centroids, which can lead to variability in the results. The initial placement of centroids affects the convergence of the algorithm and the final clusters. As a result, different runs of the algorithm with the same k may produce slightly different compressed images.[4]

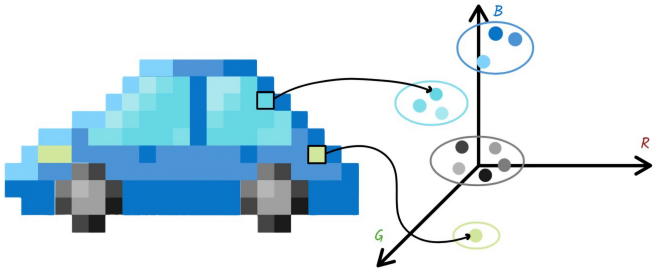


Figure 3. Image color "decomposition"

The importance of luminance

While initial experiments were conducted using the RGB color model, it was later found that the `YCbCr` color space provided significantly better results for lossy compression in this scenario. YCbCr separates the image into luminance (Y) and chrominance (Cb, Cr) components. The focus on luminance helps preserve the perceived image quality, while chrominance data can be reduced more aggressively without substantial visual degradation. Additionally, the nonlinearity of YCbCr better supports compression, as human vision is more sensitive to changes in brightness than in color, allowing for more efficient reduction of color information.

In the YCbCr model, the pixel data is transformed from the RGB space, with subsequent clustering and compression performed on the transformed data. The algorithm reduces the number of distinct colors by identifying representative colors in the YCbCr space and reassigning pixels to the nearest centroid.

3. Implementation

The implementation of the *Parallel Kmeans-based Image Compressor* (`PKIC`) application is structured to provide a flexible and user-friendly interface for compressing and decompressing images using the available methods. The core of the application revolves around a menu-driven system that allows the user to decide whether to compress or decompress an image, choose between different compression settings, and select the specific parallel method to be used for encoding. This modular design facilitates the execution of different processes to test the different implementations. The following subsections detail the main components of the system's implementation.

3.1. Implementation of K-means Clustering Classes

The K-means clustering algorithm is implemented using a base class, multiple derived classes, and a separate CUDA class for GPU-based computation.

Base Class: `KMeansBase`

The `KMeansBase` class serves as the foundational class for the K-means clustering algorithm. It is part of the `km` namespace[1]. This class provides core functionalities required to manage and execute the clustering process.

The `KMeansBase` class includes two constructors: the first constructor initializes the object with a specified number of clusters `k` and a vector of `points` representing the dataset to be clustered. The second constructor initializes the object with only the number of clusters, allowing the dataset to be specified later.

The class provides several methods for accessing the clustering results:

```
[[nodiscard]] auto getPoints() const
                -> std::vector<Point>;

[[nodiscard]] auto getCentroids() const
                -> std::vector<Point>;

[[nodiscard]] auto getIterations() const
                -> int;
```

These methods return the dataset points, the computed centroids, and the number of iterations performed by the algorithm, respectively. Note that the `[[nodiscard]]` flag is used to suggest to the compiler that the return value of the function should not be ignored by the caller.

Protected member variables include:

```
int k;
std::vector<Point> points;
std::vector<Point> centroids;
int number_of_iterations;
```

These variables store the number of clusters, the dataset points, the centroids of the clusters, and the iteration count. Derived classes have direct access to these members to facilitate their specific implementations.

The class defines a pure virtual function:

```
virtual void run() = 0;
```

This function must be implemented by any derived class to execute the K-means algorithm, thereby making `KMeansBase` an abstract class.

Derived Classes Constructors

Several classes inherit from `KMeansBase` to provide specific implementations of the K-means algorithm.

For both `KMeansSequential` and `KMeansOMP`, the constructors simply use a member initializer list to call the base class constructor of `KMeansBase`. This straightforward approach ensures that both classes inherit the base initialization logic, setting up the number of clusters and dataset points directly through the base class, without additional modifications or initializations.

In contrast, `KMeansMPI` requires a more nuanced initialization approach due to its use in distributed computing environments with MPI. The root process utilizes a constructor that initializes the number of clusters and all the data points. In contrast, the other processes are initialized with a default constructor and later receive their assigned data chunks from the root process.

The `run()` Method in Derived Classes

The `run()` method, initially declared as a pure virtual function in the `KMeansBase` class, is overridden in each derived class to implement the K-means clustering algorithm according to the specific computational strategy of that class.

In `KMeansSequential`, the `run()` method is defined to execute the K-means algorithm iteratively. It updates the centroids and reassigns points to the nearest cluster in each iteration until convergence is achieved.

For `KMeansOMP`, the `run()` method is also overridden but is adapted to utilize OpenMP for parallelization. It incorporates OpenMP directives to parallelize the updating of centroids and the reassignment of points across multiple threads.

[1] The `km` namespace serves as the main container for core functionalities of the project.

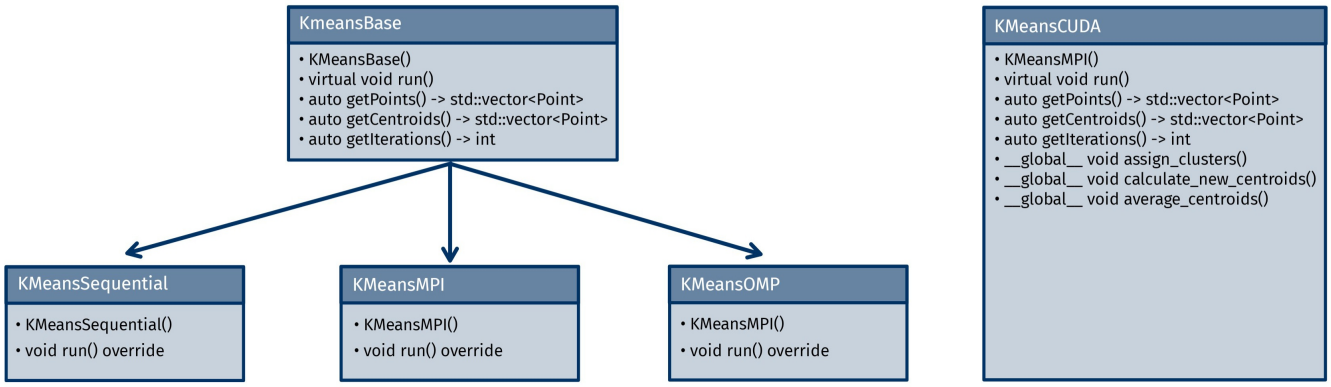


Figure 4. Code Overview

In the case of `KMeansMPI`, the `run()` method is implemented to perform clustering across multiple processes in a distributed computing environment. Unlike the other implementations, `KMeansMPI` must explicitly manage communication among processes. The method begins by distributing chunks of the dataset from the root process to all other processes. During each iteration of the algorithm, processes compute partial results (such as partial sums for centroid calculations) based on their local data. After local computations, the `run()` method employs MPI communication functions to aggregate results from all processes and distribute updated centroid information back to each process.

Further details on how the `run()` methods are optimized for parallel frameworks like OpenMP and MPI, along with their respective performance benefits, are discussed in the *parallel framworks* box ahead.

CUDA Implementation: `KMeansCUDA`

The `KMeansCUDA` class, defined in `kMeansCUDA.cuh`, implements the K-means algorithm using CUDA for GPU acceleration. Unlike the previous classes, `KMeansCUDA` does not derive from `KMeansBase` due to the specialized nature of CUDA programming.

The constructor mimics the one in `KMeansBase`: it initializes the class with the number of clusters and the dataset. The `run()` method performs the K-means algorithm on the GPU, using CUDA kernels to parallelize the computations for centroid updates and point assignments. Here are the kernels used:

```
__global__ void calculate_new_centroids
    (int *data, int *centroids, int *labels
        , int *counts, int n, int k, int dim);

__global__ void average_centroids
    (int *centroids, int *counts, int k,
        int dim);

__global__ void assign_clusters
    (int *data, int *centroids, int *labels,
        int n, int k, int dim);
```

The `run()` function in this CUDA-based KMeans algorithm manages the iterative process of clustering data points by leveraging the power of the GPU. It begins by preparing the input data, converting the points and centroids into simple arrays that are compatible with GPU processing. Once the data is ready, memory is allocated on the GPU for the points, centroids, labels, and counts, which track how many points belong to each centroid. At each iteration, the function invokes the three CUDA kernels to parallelize the computationally intensive tasks across many threads. The first kernel, `assign_clusters()`, calculates the Euclidean distance between each data point and all centroids, assigning each point to the nearest centroid. Once all points have been assigned to a cluster, the second kernel, `calculate_new_centroids()`, accumulates the feature values of the points in each cluster to update the centroids' positions. Finally, the `average_centroids()` kernel computes the new centroid positions by averaging these accumulated values. Once the algorithm completes, the final cluster labels and centroids are copied back to the CPU, and the GPU memory is freed.

Parallel frameworks

OpenMP implements a shared memory model and a multiple thread program to spawn parallel computation. Random points are picked as initial means to start the algorithm, then the parallel region is entered. Each point is assigned to its nearest cluster by a group of threads operating in parallel, followed by a sequential step to compute new means. If the algorithm has not converged then the parallel threads are spawned again with the new means.

MPI is the implementation for a distributed memory platform. The deployment is a Master-Slave model where sequential steps are performed by the master process and the slaves do the parallel work. The major difference in the distributed memory implementation from the shared memory implementation is that the master needs to perform explicit send and receive calls to distribute the data among available processes and to collect resulting assignments from the processes. In every iteration, the slaves send the index array containing the cluster assignment for each point and the master broadcasts new means to all slaves after updating means and checking for convergence. The master decides to terminate when the desired convergence is reached.

CUDA: this is a heterogeneous implementation consisting of CPU as host and GPU as device. The initialization is done on the host, then the host copies all the points to the GPU's global memory. The GPU performs the required calculations to form the index array containing the nearest cluster for each point in parallel. The index array is copied back to the host memory. The host then calculates new means and copies them back to the device if convergence is not reached.

3.2. Menu System

The menu system is the central component through which users interact with the `PKIC` application. Upon launching the program, users are presented with a menu that allows them to choose between two main operations: *encoding* (compressing) an image or *decoding* (decompressing) a previously compressed file. If the user chooses to encode an image, the menu further prompts for various compression settings and the choice of parallel method to use among the previously described.

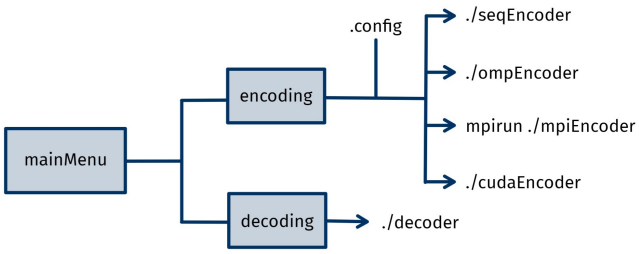


Figure 5. Menu System

One all the preferences are set the application uses `boost::process` library, which enables the *menu* to spawn a different process for each compression method. The use of `boost::process` was crucial in this implementation due to the need to handle different execution contexts: the *MPI-based encoder must be launched with* `mpirun`, while the other methods (sequential, OpenMP, and CUDA) are executed as standard processes without `mpirun`.

3.3. Compression Choices

The compression settings in this application allow users to fine-tune the image compression process by determining the amount of original color information to retain in the final output. Users can select a desired level of color preservation, which controls the *percentage of the initial colors to maintain* in the compressed image. This level of control is particularly useful for achieving a balance between image quality and file size.

In addition to color preservation levels, the application provides several compression methods, each offering different degrees of compression intensity. These methods range from lower compression levels, which aim to maintain high image quality with minimal reduction in data size, to heavier compression, which significantly reduces the image size by aggressively applying *chroma subsampling* and *resizing* techniques. Chroma subsampling is a method that reduces the amount of color information in the image by averaging nearby color values, which is less noticeable to the human eye due to the way we perceive color versus brightness. Resizing, on the other hand, reduces the pixel dimensions of the image to achieve further size reduction. [2]

The application offers the following three compression levels:

- **Light Compression:** This level preserves the most detail and is recommended for smaller images where maintaining high quality is a priority. It may take more time to process due to the minimal reduction in data.
- **Medium Compression:** At this level, chroma subsampling is applied to reduce the image size and the time required for computation, making it a balanced option for moderate size reduction while still retaining good image quality.
- **Heavy Compression:** This level applies both chroma subsampling and resizing, significantly reducing the image size. It is suggested for larger images where file size reduction is more critical than retaining the highest possible quality.

Users can set these options manually through the main menu of the application. The ‘main.cpp‘ file serves as the main entry point for the Image Compressor application, initializing necessary components, reading configuration settings, and providing a command-line interface for users to choose between image compression and decompression options. The application prompts users to select the desired compression level and color preservation settings if they are not predefined in the `.config` file. The `ConfigReader` class automatically reads and applies settings from the `.config` file. If a setting is defined in the file, it will be used. This modular approach allows for partial configuration, meaning users can specify only the settings they want to pre-configure.

[2] Both chroma subsampling and resizing are executed through internal OpenCV routines, such library is also used to read and load the image into the program.

3.4. Binary Encoding

Following the application of the k-means clustering algorithm, the next step is to efficiently encode the clustered data into a binary format. As previously mentioned, compression is achieved by storing only the color information of the cluster centroids, with all other pixel data being redundant. The process capitalizes on the fact that each pixel's color can be represented by its cluster membership rather than its full RGB value, which substantially reduces the data size.

In this encoding process, *metadata* are stored at the beginning of the binary file, containing essential information such as the image dimensions and the number of clusters k. These metadata is crucial for reconstructing the image during decoding. Instead of storing each pixel's full color information, we store only the color information of each cluster centroid. The pixel data is then represented by the cluster membership index, which requires significantly fewer bits.

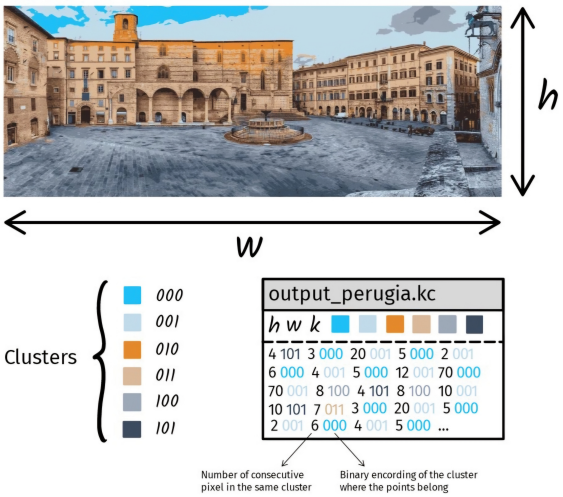


Figure 6. Encoding mechanism

Byte-level representation

To implement binary encoding as previously described it's necessary to convert data into a *byte-level representation* before writing it to the binary file. This is crucial because binary files operate at the byte level, meaning all data—whether integer values or more complex structures—must be converted into sequences of bytes. In the `km::filesUtils::writeBinaryFile` function, the image dimensions (width, height) and the number of clusters k are first converted to a 16-bit unsigned integer format (‘uint16_t‘). This ensures that the data is appropriately sized and formatted for compact storage.

Later, when writing cluster information to the binary file, the cluster memberships are represented as 8-bit unsigned integers (‘uint8_t‘). The choice of ‘uint8_t‘ is intentional; it ensures that each membership value occupies exactly one byte, aligning with the binary file's need for byte-aligned data. This conversion process not only aligns with the file's storage requirements but also reduces the file size, as fewer bits are used to represent the same information.

Further compression is achieved through run-length encoding (`RLE`). When pixels belonging to the same cluster are encountered

consecutively, only the start of the run and its length (count) are stored. In the code, this is handled by writing the run length (number of consecutive pixels with the same cluster membership) and the cluster ID to the buffer. Using ‘uint8_t‘ for both the count and cluster ID allows for efficient storage, with each run occupying just two bytes regardless of its length. However, this design imposes certain limitations: the maximum number of clusters that can be indexed is limited to 256, and the maximum number of consecutive pixels with the same color that can be represented in a single RLE run without repeating the cluster ID is also capped at 256.

These limitations were deliberately accepted after evaluating the benchmark images, which exhibit variability in both size and color count. It was determined that, for all cases tested, these constraints did not pose any problems. Removing these restrictions would have resulted in significantly larger file sizes after encoding, diminishing the storage efficiency of the method. Therefore, the decision to impose these limits was made to balance between file size and flexibility.

3.5. Utilities Overview

The utility functions within the `PKIC` application are strategically organized into three distinct namespaces: `km::filesUtils`, `km::imageUtils`, and `km::utilsCLI`. Each namespace is designed to handle specific aspects of file management, image processing, and command-line interface operations, respectively, ensuring that the application maintains a modular and clean structure.

C++ static&public "classes" through namespaces

In C++, organizing utility functions in namespaces is considered good practice when those functions are "static" in nature, meaning they do not require maintaining any state and are not tied to any particular object instance. Essentially, namespaces can be viewed as a way to create "static classes" without the overhead and structure of class instantiation. This design choice has several advantages.

First, it reduces overhead because namespaces do not require memory allocation or the management of constructors and destructors, unlike static classes that might involve additional syntactic and semantic complexities. Second, namespaces offer greater simplicity and flexibility in grouping related functions under a common domain without the rigidity of class definitions, which is particularly useful in applications focused on functional operations, such as image processing or file management.

The `km::filesUtils` namespace focuses on file handling tasks essential for managing the data flow of the application. It includes functions for creating the output directory, writing and reading binary files, verifying file extensions, and generating the decoding menu. These utilities ensure the application can efficiently manage file operations, particularly when handling compressed image data in binary form.

The `km::imageUtils` namespace is dedicated to various image processing tasks required by the application. This namespace provides functions for preprocessing images to adjusts images according to specified compression settings, setting the number of clusters for color-based compression based on user choice, and extract point form image[3] to be able to process them.

The `km::utilsCLI` namespace is designed to enhance the command-line interface (CLI) experience, providing a range of functions that support user interaction. These utilities manage the display of menu headers, completion messages, and detailed compression information, as well as presenting decoding options to the user. By centralizing these operations, `km::utilsCLI` maintains a user-friendly interface that guides users through the compression and decompression processes.

[3] Extraction exploits `openCV`

3.6. Decoder

The decoder component of the system is responsible for reading the binary-encoded files produced by the encoding process and reconstructing the compressed image. It takes the binary format file and decodes it to regenerate the image with the preserved essential features. This allows users to visualize the compressed image and to save it in a `.jpg` format.

4. Experimental results

4.1. Experimental Setup

To facilitate an in-depth evaluation of the scalability and performance of the different parallel implementations, we conducted our experiments on specific hardware configurations. The GPU used is an NVIDIA GeForce MX150, featuring 384 CUDA cores and a base clock speed of 1468 MHz. For CPU-based implementations, we utilized an Intel Core i7-12700H, which includes 14 cores (6 performance cores and 8 efficiency cores) with a base clock speed of 2.3 GHz (for the performance cores) and a maximum turbo frequency of up to 4.7 GHz.

It is important to note that the application was not tested in a distributed system environment. In such a setup, using MPI could have significantly enhanced performance, particularly if MPI was employed for inter-node communication while OpenMP or CUDA was utilized for fine-grain parallelism within each node.

4.2. Performance Analysis with Varying Task Complexity

In this section, we examine how the compression times for each implementation vary as the complexity of the task increases, focusing on identifying which method is most effective in different scenarios. Specifically, we analyze the impact of changing the number of data points on the performance of each implementation.

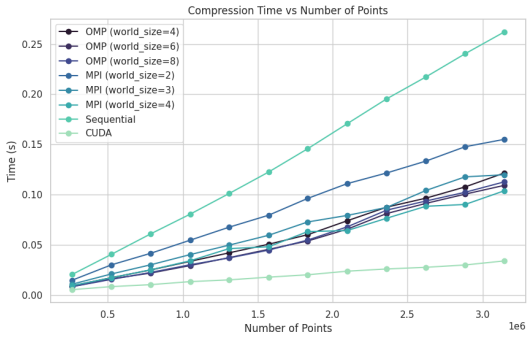


Figure 7. Execution time comparison

Sequential (light blue) performs the worst, which is no news, with the compression time increasing significantly as the number of points increases. This outcome is expected, as the sequential approach runs on a single thread and lacks any form of parallelism. Without the ability to distribute the workload across multiple cores or threads, the sequential method processes each operation one at a time. As a result, the compression time increases significantly with larger datasets. This linear growth in processing time highlights the limitations of a single-threaded approach when dealing with computationally intensive tasks, such as compression.

CUDA (light green) consistently shows the best performance (lowest time) across all data sizes. This is expected as CUDA leverages GPU acceleration, which is highly optimized for parallel tasks like compression. By distributing computational tasks across thousands of GPU cores, CUDA can execute a huge number of operations concurrently. The linearity and repetitive nature of compression tasks make them well-suited to this parallel processing approach, as the workload can be evenly distributed without much inter-dependency

between tasks. This leads to a significant reduction in processing time compared to traditional CPU-based processing, which typically involves fewer cores and thus less parallelism.

OMP and MPI seem to perform similarly, though the exact world sizes provide slight variations in performance. As the world size increases (more threads or processes), there is generally a slight improvement in performance.

4.3. Scalability

Scalability is a critical measure of how the implementation behaves while the number of available resources increases, an overview of the difference between "weak" and "strong" is given further on in the proper box.

Weak Scalability

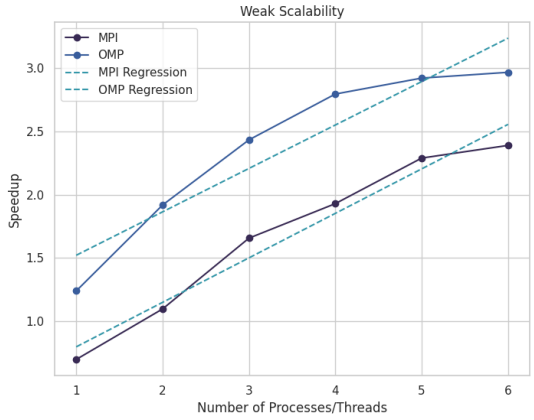


Figure 8. Weak Scalability

OMP generally demonstrates better scalability compared to MPI, as indicated by its higher speedup values across all tested configurations. OMP's speedup increases steadily, nearing a speedup of 3.0 at six threads, while MPI shows a more modest improvement, achieving a maximum speedup of slightly above 2.0. The regression lines indicate that both methods exhibit a trend of diminishing returns as the number of processes or threads increases.

Strong Scalability

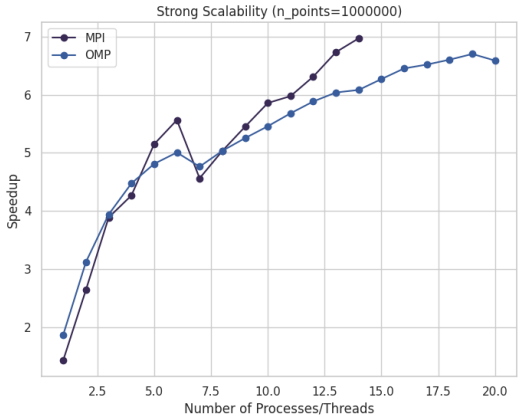


Figure 9. Strong Scalability

In this strong scalability scenario with a fixed workload size of 1,000,000 points, MPI generally outperforms OMP, achieving higher speedup values as the number of processes/threads increases. Both MPI and OMP demonstrate an initial steep increase in speedup, particularly noticeable up to around 5 processes/threads. Beyond this point, the speedup growth for OMP tends to stabilize, showing a more gradual increase and eventually plateauing around a speedup of 6.0 as the number of threads approaches 20.

In contrast, MPI continues to scale more effectively with additional processes, achieving a maximum speedup of about 7.0. However, there are noticeable fluctuations in the MPI curve due to the fact that the CPU used has 6 performance cores and 8 efficency cores. The stronger scalability of MPI compared to OMP suggests that MPI may be more suitable for workloads that can benefit from a larger number of processes, particularly in scenarios requiring higher levels of parallelism.

Strong Scalability VS Weak Scalability

To understand the concepts of weak and strong scalability in the context of parallel computing.

Weak scalability (based on *Gustafson's law*) refers to an algorithm's ability to maintain constant performance as both the problem size and the number of processing elements increase proportionally. Formally, if:

- P is the number of processing elements (processors),
- $T(P, N)$ is the runtime of the algorithm with P processors and problem size N,

then weak scalability is achieved if:

$$T(P, N) \approx T(1, \frac{N}{P})$$

This implies that the runtime remains relatively stable when both the number of processors and the problem size increase proportionally.

Strong scalability (based on *Amdahl's law*) measures how the performance of an algorithm improves when increasing the number of processing elements, while keeping the problem size N fixed. Strong scalability is characterized by a significant reduction in runtime as more processors are added. Mathematically, it can be expressed as:

$$T(P, N) \approx \frac{T(1, N)}{P}$$

However, according to *Amdahl's law*, this ideal speedup is not fully achievable due to the fixed serial portion of the algorithm that cannot be parallelized. This serial fraction limits the overall performance improvement, as it cannot be divided across P processors. Amdahl's law is given by the formula:

$$S(P) = \frac{1}{(1-\alpha) + \frac{\alpha}{P}} \xrightarrow{P \to \infty} \frac{1}{1-\alpha}$$

where $S(P)$ is the speedup with P processors, and α represents the fraction of the program that is parallelizable

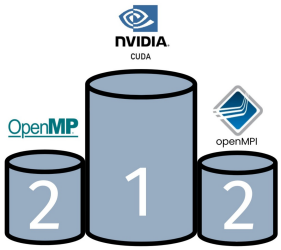


Figure 10. Ranking of parallelization techniques

4.4. Compression Result

In this section, we present the results of applying the k-means based image compression algorithm across various test images.The following figure illustrates the comparison between the original uncompressed image (on the left) and the corresponding compressed image generated by our implementation (on the right).

Figure 11. Original and Compressed Version of Lena

With this level of compression, the resulting image closely resembles the original, yet the file size is significantly reduced from 478 KB to 154 KB. With a very high level of compression, the image ends up weighing 27.3 KB (Figure 12)

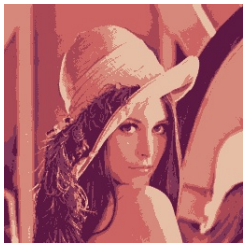


Figure 12. Super Compressed Version of Lena

Since we employed Run-Length Encoding (RLE) for compression, images featuring contiguous regions of similar colors experience considerable advantages. RLE is particularly effective in these cases because it encodes consecutive pixels of the same color as a single value and a count.

5. Conclusion

In the present work, we have demonstrated how K-Means, specifically applied to image compression, can be studied and implemented within parallel computing frameworks to analyze efficiency and performance. Our study focused on identifying which framework performs better in specific environments, such as single-node systems without distributed computing or clusters.

As anticipated, CUDA consistently delivered superior performance, especially as the problem size increased. In contrast, both OpenMP and MPI experienced significant performance degradation as the problem grew larger, primarily due to resource exhaustion. These frameworks exhaust available CPU resources much earlier than GPU resources, leading to diminished performance.

Unexpectedly, MPI performed comparably to OpenMP in many scenarios, with both frameworks exhibiting similar limitations. These limitations are primarily due to the restricted number of cores, especially when using efficiency cores. The performance drops significantly not only because these cores are inherently "less performant" but also because occupying them impacts the overall system's ability to manage resources effectively, leading to operational inefficiencies.

Future Developments

In future developments, parallelization can be leveraged not only to improve the runtime of K-Means but also to enhance the initialization phase. A better choice of initial centroids often leads to more accurate clustering results. This can be achieved by running several iterations of the algorithm multiple times in parallel, with different random initializations. Each run would be evaluated based on inter-cluster distance metrics, such as maximizing the minimum distance between centroids or minimizing intra-cluster variance. The best-performing initialization would then be selected as the starting point for the full K-Means run. This entire process can be fully parallelized using MPI or OpenMP.

Another potential area of exploration is the evaluation of various heuristics and data structures that are typically used to improve K-Means performance. Techniques such as KD-trees or Ball-trees, commonly employed for efficient nearest-neighbor searches, could be analyzed to determine which perform better in parallel environments, and which are less suited for parallelization.

Finally, testing K-Means on a distributed system with significantly higher computational loads would be an important next step. By utilizing multiple nodes, each equipped with GPUs (or even multiple GPUs per node), the scalability and performance of the algorithm could be thoroughly assessed in a distributed parallel setting.

If you arrived here...

... here's a little secret for you! In **Figure 1**, none of the images are actually the original. Each one was compressed with a different choice of color retention, with all of them containing less than 10% of the original colors. **Quite the surprise, isn't it?**

References

[1] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review", *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[2] J. Bhimani, M. Leeser, and N. Mi, "Accelerating k-means clustering with parallel implementations and gpu computing", in *2015 IEEE high performance extreme computing conference (HPEC)*, IEEE, 2015, pp. 1–6.

[3] M. Capó, A. Pérez, and J. A. Lozano, "An efficient approximation to the k-means clustering for massive data", *Knowledge-Based Systems*, vol. 117, pp. 56–69, 2017.

[4] S. Charmot, "Clear, visual explanation of k-means for image compression with gifs", *Towards Data Science*, 2023. [Online]. Available: https://towardsdatascience.com/clear-and-visual-explanation-of-the-k-means-algorithm-applied-to-image-compression-b7fdc547e410.

Contact us

You can contact us through these methods.

leonardoignazio.pagliochini@mail.polimi.it
francesco.rosnati@mail.polimi.it