ParallelKmeansImageCompressor

Generated by Doxygen 1.9.8

Chapter 1

Parallel Kmeans-based Images Compressor

This project implements a **parallel KMeans-based image colors compressor**, aimed at reducing the number of colors in a natural image while preserving its overall visual appearance. The program clusters similar colors using the **KMeans algorithm** and applies **parallel computing techniques** to compress the image through the **color quantization** technique. It supports **sequential**, **OpenMP**, **MPI**, and **CUDA** implementations to explore different levels of performance and scalability.

1.1 Authors

- **Leonardo Ignazio Pagliochini** Master's Degree student in High-Performance Computing Engineering at **Politecnico di Milano**

  GitHub: `leonardopagliochini`
  Email: `leonardoignazio.pagliochini@mail.polimi.it`

- **Francesco Rosnati** Master's Degree student in High-Performance Computing Engineering at **Politecnico di Milano**

  GitHub: `RosNaviGator`
  Email: `francesco.rosnati@mail.polimi.it`

This project was developed for the course **Advanced Methods for Scientific Computing**,
Professor: **Luca Formaggia**
Assistant Professor: **Matteo Caldana**
Politecnico di Milano

1.2 Doxygen Documentation

The documentation of the project can be found `here`.

1.3 Prerequisites

1.3.1 OpenCV C++ Library

A comprehensive library for computer vision and image processing tasks. You can refer to the `official page` to download.

1.3.2 Mpicc

A C compiler wrapper for parallel programming with the MPI library. It is advised to use `openmpi`, `official page`, as we experienced some bugs with `mpich` due to experimental version of `g++`.

1.3.3 OpenMP

A C++ API for parallel programming on shared-memory systems.

1.3.4 Boost

Boost is a versatile, cross-platform, and comprehensive collection of highly optimized, portable, reliable, and robust C++ libraries designed to enhance software development efficiency and extensibility.

1.4 Getting Started

1.4.1 Cloning repo

Standard coloning with `git clone`, no *submodules* are implemented in this repo.

1.4.2 Install dependencies

1.4.2.1 Debian based

```
# Run commands from PROJECT ROOT DIRECTORY
sudo chmod +x ./dependencyInstaller/dependencyInstallerDebianBased.sh
source ./dependencyInstaller/dependencyInstallerDebianBased.sh
```

1.4.2.2 Arch based

```
# Run commands from PROJECT ROOT DIRECTORY
sudo chmod +x ./dependencyInstaller/dependencyInstallerArchBased.sh
source ./dependencyInstaller/dependencyInstallerArchBased.sh
```

1.4.3 Compile&Run

Program can be built with or without **CUDA**, you obviously *need* `nvcc` to be able to compile with **CUDA**. To compile the project, navigate to the **project root directory** in your terminal and run the following commands.

1.4.3.1 DO NOT USE <tt>mkmodules</tt>

Program uses a version of `g++` with very recent standards, that were not supported by `mkmodules`, it is important to **unload** modules, including `gcc-glibc` in order to succesfully compile.

1.4.3.2 Without CUDA

```
# make without running
make
# make and run the menu (or simply run if already built)
make run
```

1.4.3.3 With cuda

```
# make without running (bulding also CUDA)
make cuda
# make and run the menu (building also CUDA)
make cudarun
```

1.4.3.4 Standard Run

For a standard run program will guide you to choose an *image path*, *parallel method*, *configuration settings*. See section "What to expect" for more infromation.

1.4.3.5 Debug/Preconfigured Run

If you want to avoid having to input all the information through the prompts requested by the program, you can preconfigure the options in the `.config` file.

1.5 What to expect

Once the program is started, the following screen appears, through which it is possible to compress a new image or decompress an already compressed image.

1.5.1 Compress an image

1.5.1.1 Set

If you choose the "Compress an image" option you will be asked to select:

1. **Type of parallelization** among *sequential*, *MPI*, *OpenMP* (also CUDA if `nvcc` was used during the compiling process), the type of compression, and the path of the original image.
2. **Color level**: five levels of compression, each corresponding to a certain *percentage of retained colors* (it's possible to visualize such percentages in the `.config` file).
3. **Image path**: The location of the original image file to be compressed.
4. **Three methods of compression**:
    - **Light Compression**: Preserves the most detail, recommended for smaller images where maintaining high quality is a priority. This level may take more time to process.
    - **Medium Compression**: Uses chroma subsampling to reduce image size and processing time. This is a balanced option for moderate size reduction while retaining good image quality.
    - **Heavy Compression**: Applies both chroma subsampling and resizing, significantly reducing the image size. Suitable for larger images where file size reduction is more important than retaining the highest possible quality.

1.5.1.2 Launch

After prompting all required settings the `menu` executable will exploit `boost/process` to *launch* a specific *process* (executable) relative to the chosen method, using the given *settings as arguments*.

1.5.2 Decode image

The menu will launch the `decoder` *process* (again with `boost/process`), there you will be asked to choose which `.kc` ('kmeans-compressed') to decode and visualize. Program creates list of `.kc` available to decode from the `output` folder.

1.6 Project Structure

The project is organized as follows:

1.6.0.1 Folders

- `benchmarkImages`: Images used for benchmarking the program. It can be used to test the program's performance.
- `outputs`: Contains the compressed images. After installing the program, you may notice that the outputs folder is not present. However, don't worry! It will be automatically created during the first execution of the program.
- `include`: Header files of the project. These define the classes and functions that are used in the program.
- `src`: Contains the source files of the project. These files contain the implementation of the classes and functions defined in the header files.
- `build`: Object files generated during the compilation process.
- `dependencyInstaller`: This folder contains the two scripts that can be used to install the required libraries.
- `performanceEvaluation`: Python codes used to evaluate performance and scalability of the four type of executions.

1.6.0.2 Files and Executables

- `menu`: This is the executable file generated after compiling the project. It is the main program that can be executed to compress or decompress images.
- `Makefile`: This file contains the instructions for compiling the project. It specifies the dependencies and the commands to compile the project.
- `.config`: This file contains the configuration of the program. It is used to store some hyperparameters that can be modified to change the behavior of the program.
- `Doxyfile`: Is a configuration file used by Doxygen to customize the generation of documentation from annotated source code.
- `Readme.md`: You already know that buddy ;)

1.7 How does it work?

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 (**color quantization**), possibly minimizing the data required to represent the image.

1.7.1 Kmeans

The algorithm begins with an initialization phase, where k initial cluster centers (means) $\mu_1^0, \mu_2^0, \ldots, \mu_k^0$ are chosen. After initialization, the algorithm enters an iterative phase, often referred to as **Lloyd's algorithm**, which repeatedly executes two main steps until convergence is reached.

1. Assign each data point to the nearest centroid.
2. Recompute the centroids based on the data points assigned to them.

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

1.7.2 Parallelization Techniques

The program uses several parallelization techniques to enhance performance. These techniques include:

- **OpenMP**: OpenMP is an API for parallel programming on shared-memory systems. It allows the program to parallelize the computation of the k-means algorithm by distributing the work among multiple threads.
- **MPI**: MPI is a message-passing library for parallel programming on distributed-memory systems. It allows the program to parallelize the computation of the k-means algorithm by distributing the work among multiple processes running on different nodes.
- **CUDA**: is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs (Graphics Processing Units). CUDA enables the acceleration of computationally intensive algorithms, like k-means clustering, by offloading the work to GPUs, which can process thousands of threads simultaneously. This results in a significant speedup, especially for tasks that involve large datasets and require high computational throughput.

1.7.3 What Parallelization Technique Should I Choose?

That is a really good question... as everithing in computer science **it depends**. Here you can see an overview of the execution time behaviour for increasing complexity tasks:

1.8 Report

For more details about the program, please refer to the `report`

Chapter 2

Namespace Documentation

2.1 km Namespace Reference

Main namespace for the project.

Namespaces

- namespace filesUtils
  Provides utility functions for file handling.
- namespace imageUtils
  Provides utility functions for image processing.
- namespace utilsCLI
  Provides utility functions for the command-line interface.

Classes

- class ConfigReader
  Reads and stores configuration values from a file.
- class KMeansBase
  Base class for K-means clustering algorithm.
- class KMeansCUDA
  Represents the K-means clustering algorithm using CUDA.
- class KMeansMPI
  Represents the K-means clustering algorithm using MPI.
- class KMeansOMP
  Represents the K-means clustering algorithm using OpenMP.
- class KMeansSequential
  Represents the K-means clustering algorithm.
- class Performance
  Represents the performance evaluation.
- class Point
  Represents a point in a feature space.

Functions

- __global__ void calculate_new_centroids (int *data, int *centroids, int *labels, int *counts, int n, int k, int dim)
  CUDA kernel to calculate the new centroids based on the assigned clusters.
- __global__ void average_centroids (int *centroids, int *counts, int k, int dim)
  CUDA kernel to average the calculated centroids.
- __global__ void assign_clusters (int *data, int *centroids, int *labels, int n, int k, int dim)
  CUDA kernel to assign each point to the nearest centroid.

2.1.1 Detailed Description

Main namespace for the project.

The `km` namespace encapsulates various functionalities related to data clustering, file manipulation, and image processing. It is designed to organize core utilities and algorithms used across different modules of the project.

2.1.2 Function Documentation

2.1.2.1 assign_clusters()

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

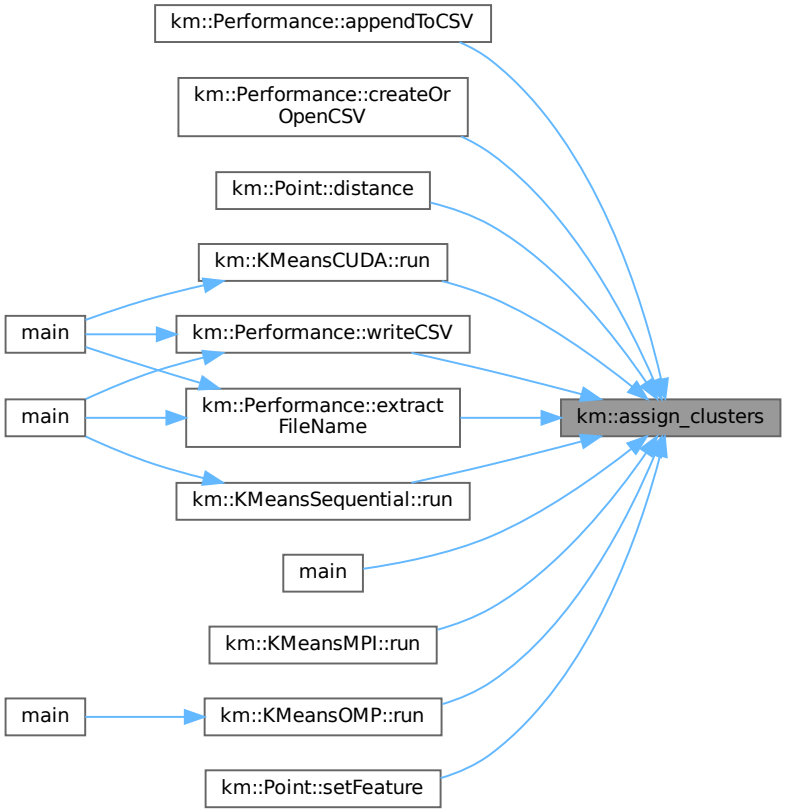
CUDA kernel to assign each point to the nearest centroid.

Parameters

data	Pointer to the data points
centroids	Pointer to the current centroids
labels	Pointer to the labels (cluster assignments)
n	Number of data points
k	Number of clusters
dim	Number of dimensions for each data point

This kernel assigns each point to the nearest centroid by calculating the Euclidean distance between each point and the centroids. The closest centroid's index is assigned to the corresponding position in the labels array. Here is the caller graph for this

function:



2.1.2.2 average_centroids()

```
__global__ void km::average_centroids (
                int * centroids,
                int * counts,
                int k,
                int dim )
```

CUDA kernel to average the calculated centroids.

Parameters

centroids	Pointer to the current centroids
counts	Pointer to the counts of points per cluster
k	Number of clusters
dim	Number of dimensions for each data point

This kernel averages the sum of the centroids from the `calculate_new_centroids` kernel by dividing the summed values by the number of points in each cluster.

2.1.2.3 calculate_new_centroids()

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

CUDA kernel to calculate the new centroids based on the assigned clusters.

Parameters

data	Pointer to the data points
centroids	Pointer to the current centroids
labels	Pointer to the labels (cluster assignments)
counts	Pointer to the counts of points per cluster
n	Number of data points
k	Number of clusters
dim	Number of dimensions for each data point

This kernel calculates the new centroids by summing the data points assigned to each centroid. The results are stored in the centroids array and the counts array records the number of points assigned to each centroid.

2.2 km::filesUtils Namespace Reference

Provides utility functions for file handling.

Functions

- auto createOutputDirectories () -> void
  Creates output directories.
- auto writeBinaryFile (std::string &outputPath, int &width, int &height, int &k, std::vector< Point > points, std::vector< Point > centroids) -> void
  Writes data to a binary file.
- auto isCorrectExtension (const std::filesystem::path &filePath, const std::string &correctExtension) -> bool
  Checks if a file has the correct extension.
- auto createDecodingMenu (std::filesystem::path &decodeDir, std::vector< std::filesystem::path > &imageNames) -> void
  Creates a decoding menu.
- auto readBinaryFile (std::string &path, cv::Mat &imageCompressed) -> int
  Reads a binary file and reconstructs the compressed image.

2.2.1 Detailed Description

Provides utility functions for file handling.

The filesUtils namespace within the km namespace offers a set of utility functions designed to handle various file operations crucial for image processing and data management. It includes functionalities to create necessary output directories, ensuring that the required directory structure is in place before any file operations are performed. The namespace provides a function to write data to a binary file, which includes parameters for the file path, image dimensions, the number of clusters, and vectors of points and centroids. This is particularly useful for saving compressed image data or related binary information. Additionally, it includes a function to verify whether a file has the correct extension, which is essential for validating file types before processing. The createDecodingMenu function facilitates the creation of a decoding menu by accepting a directory path and a vector of image names, which may be used for setting up decoding options. Lastly, the readBinaryFile function reads from a binary file to reconstruct a compressed image into an OpenCV matrix, returning the number of clusters present in the file. This set of functions is designed to streamline and manage file-related tasks, particularly in the context of image processing.

2.2.2 Function Documentation

2.2.2.1 createDecodingMenu()

```
void km::filesUtils::createDecodingMenu (
              std::filesystem::path & decodeDir,
              std::vector< std::filesystem::path > & imageNames ) -> void
```

Creates a decoding menu.

Parameters

decodeDir	Directory for decoding
imageNames	Vector of image names

Here is the call graph for this function:

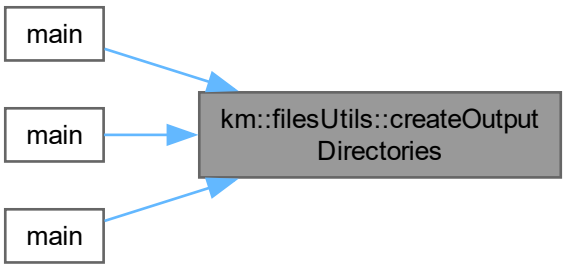
Here is the caller graph for this function:

2.2.2.2 createOutputDirectories()

```
void km::filesUtils::createOutputDirectories ( ) -> void
```

Creates output directories.

Here is the caller graph for this function:



2.2.2.3 isCorrectExtension()

```
auto km::filesUtils::isCorrectExtension (
              const std::filesystem::path & filePath,
              const std::string & correctExtension ) -> bool
```

Checks if a file has the correct extension.

Parameters

filePath	Path of the file
correctExtension	Correct extension to check

Returns

True if the file has the correct extension, false otherwise

Here is the caller graph for this function:

2.2.2.4 readBinaryFile()

```
auto km::filesUtils::readBinaryFile (
              std::string & path,
              cv::Mat & imageCompressed ) -> int
```

Reads a binary file and reconstructs the compressed image.

Parameters

path	Path of the binary file
imageCompressed	Compressed image matrix

Returns

Number of clusters

Here is the caller graph for this function:

2.2.2.5 writeBinaryFile()

```
void km::filesUtils::writeBinaryFile (
                std::string & outputPath,
                int & width,
                int & height,
                int & k,
                std::vector< Point > points,
                std::vector< Point > centroids ) -> void
```

Writes data to a binary file.

Parameters

outputPath	Path of the output file
width	Width of the image
height	Height of the image
k	Number of clusters
points	Vector of points
centroids	Vector of centroids

Here is the caller graph for this function:

2.3 km::imageUtils Namespace Reference

Provides utility functions for image processing.

Functions

- void preprocessing (cv::Mat &image, int &typeCompressionChoice)
  Performs preprocessing on an image.
- void defineKValue (int &k, int levelsColorsChoice, std::set< std::vector< unsigned char > > &different_colors)
  Defines the value of K based on the color levels choice.
- void pointsFromImage (cv::Mat &image, std::vector< Point > &points, std::set< std::vector< unsigned char > > &different_colors)
  Extracts points from an image.

2.3.1 Detailed Description

Provides utility functions for image processing.

The imageUtils namespace within the km namespace provides a suite of utility functions aimed at facilitating various image processing tasks. This namespace encompasses functions designed to preprocess images, determine the appropriate number of clusters for color-based compression, and extract points from images for further analysis. The preprocessing function is responsible for preparing an image for subsequent processing steps, adjusting it according to the specified type of compression. The defineKValue function calculates the number of clusters, or K, based on the chosen levels of colors and the distinct colors present in the image. This function helps in determining the optimal number of clusters for tasks such as color quantization. Lastly, the pointsFromImage function extracts points from the image and organizes them into a vector, using the set of distinct colors found in the image to aid in this process. These functions collectively support various aspects of image processing, ensuring efficient handling and analysis of image data.

2.3.2 Function Documentation

2.3.2.1 defineKValue()

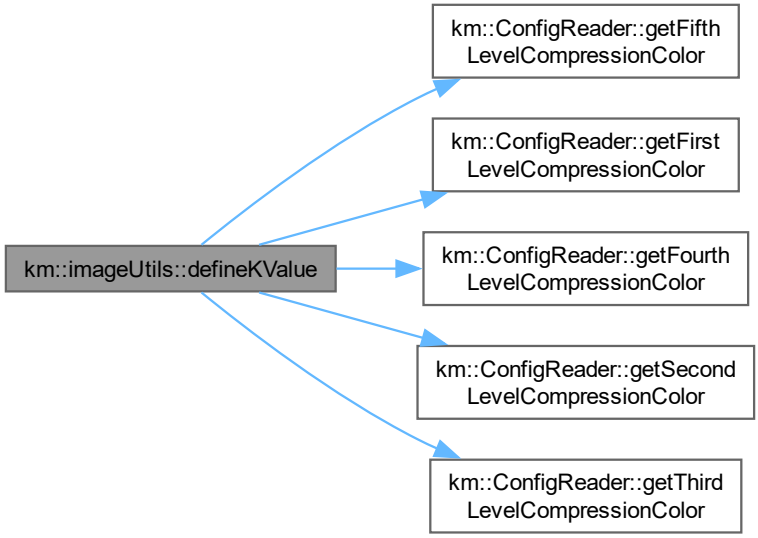
```
void km::imageUtils::defineKValue (
              int & k,
              int levelsColorsChoice,
              std::set< std::vector< unsigned char > > & different_colors )
```

Defines the value of K based on the color levels choice.

Parameters

k	Value of K
levelsColorsChoice	Levels of colors choice
different_colors	Set of different colors in the image

Here is the call graph for this function:



Here is the caller graph for this function:

2.3.2.2 pointsFromImage()

```
void km::imageUtils::pointsFromImage (
              cv::Mat & image,
              std::vector< Point > & points,
              std::set< std::vector< unsigned char > > & different_colors )
```

Extracts points from an image.

Parameters

image	Input image
points	Vector of points
different_colors	Set of different colors in the image

Here is the caller graph for this function:

2.3.2.3 preprocessing()

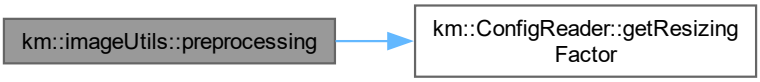
```
void km::imageUtils::preprocessing (
                cv::Mat & image,
                int & typeCompressionChoice )
```

Performs preprocessing on an image.

Parameters

image	Input image
typeCompressionChoice	Type of compression choice

Here is the call graph for this function:



Here is the caller graph for this function:

2.4 km::utilsCLI Namespace Reference

Provides utility functions for the command-line interface.

Functions

- void mainMenuHeader ()

  Displays the main menu header.
- void decoderHeader ()

  Displays the decoder header.
- void workDone ()

  Displays the work done message.
- void printCompressionInformations (int &originalWidth, int &originalHeight, int &width, int &height, int &k, size_t &different_colors_size)

  Prints the compression information.
- void displayDecodingMenu (std::string &path, std::vector< std::filesystem::path > &imageNames, std::filesystem::path &decodeDir)

  Displays the decoding menu.

2.4.1 Detailed Description

Provides utility functions for the command-line interface.

The utilsCLI namespace within the km namespace provides a collection of utility functions for enhancing command-line interface(CLI) interactions. The mainMenuHeader function displays the main menu header, while decoderHeader shows the header for the decoder section. The workDone function outputs a completion message to indicate that work has been finished. The printCompressionInformations function prints detailed compression data, including the original and compressed image dimensions, the number of clusters, and the count of different colors. Lastly, the displayDecodingMenu function presents a menu for decoding, showing image names and the path of the decoding directory. These functions facilitate user interaction and provide essential information during CLI operations.

2.4.2 Function Documentation

2.4.2.1 decoderHeader()

```
void km::utilsCLI::decoderHeader ( )
```

Displays the decoder header.

Here is the caller graph for this function:

2.4.2.2 displayDecodingMenu()

```
void km::utilsCLI::displayDecodingMenu (
                std::string & path,
                std::vector< std::filesystem::path > & imageNames,
                std::filesystem::path & decodeDir )
```

Displays the decoding menu.

Parameters

path	Path of the directory containing the compressed images
imageNames	Vector of image names
decodeDir	Path of the decoding directory

Here is the caller graph for this function:

2.4.2.3 mainMenuHeader()

```
void km::utilsCLI::mainMenuHeader ( )
```

Displays the main menu header.

Here is the caller graph for this function:

2.4.2.4 printCompressionInformations()

```
void km::utilsCLI::printCompressionInformations (
                int & originalWidth,
                int & originalHeight,
                int & width,
                int & height,
                int & k,
                size_t & different_colors_size )
```

Prints the compression information.

Parameters

originalWidth	Original width of the image

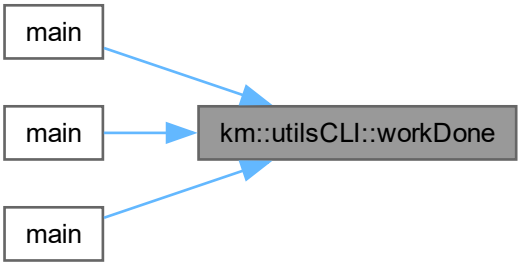
Here is the caller graph for this function:

2.4.2.5 workDone()

```
void km::utilsCLI::workDone ( )
```

Displays the work done message.

Here is the caller graph for this function:



Chapter 3

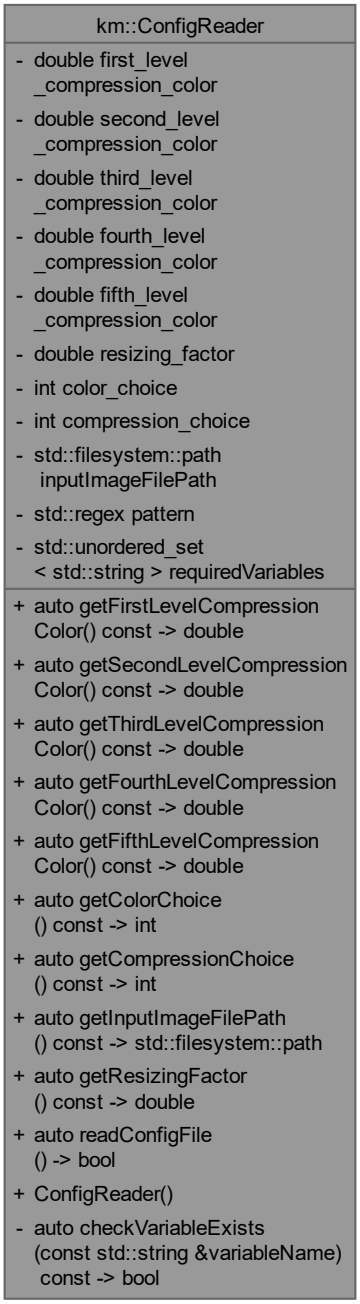
Class Documentation

3.1 km::ConfigReader Class Reference

Reads and stores configuration values from a file.

`#include <configReader.hpp>`

Collaboration diagram for km::ConfigReader:



Public Member Functions

- auto getFirstLevelCompressionColor () const -> double

  Gets the first level compression color value.
- auto getSecondLevelCompressionColor () const -> double

  Gets the second level compression color value.
- auto getThirdLevelCompressionColor () const -> double

Gets the third level compression color value.

- auto getFourthLevelCompressionColor () const -> double
  Gets the fourth level compression color value.
- auto getFifthLevelCompressionColor () const -> double
  Gets the fifth level compression color value.
- auto getColorChoice () const -> int
  Gets the color choice.
- auto getCompressionChoice () const -> int
  Gets the compression choice.
- auto getInputImageFilePath () const -> std::filesystem::path
  Gets the input image file path.
- auto getResizingFactor () const -> double
  Gets the resizing factor.
- auto readConfigFile () -> bool
  Reads the configuration file.
- ConfigReader ()

Private Member Functions

- auto checkVariableExists (const std::string &variableName) const -> bool

Private Attributes

- double first_level_compression_color = 0.
  First level compression color value.
- double second_level_compression_color = 0.
  Second level compression color value.
- double third_level_compression_color = 0.
  Third level compression color value.
- double fourth_level_compression_color = 0.
  Fourth level compression color value.
- double fifth_level_compression_color = 0.
  Fifth level compression color value.
- double resizing_factor = 0.
  Resizing factor.
- int color_choice = 0
  Color choice.
- int compression_choice = 0
  Compression choice.
- std::filesystem::path inputImageFilePath
  Input image file path.
- std::regex pattern
  Regular expression pattern.
- std::unordered_set< std::string > requiredVariables = {}
  Set of required variables.

3.1.1 Detailed Description

Reads and stores configuration values from a file.

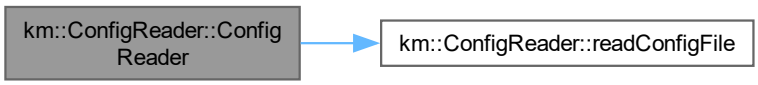
The ConfigReader class, located within the km namespace, is designed to handle the reading and storage of configuration settings from a file. This class is particularly focused on managing settings for image processing and compression. It holds various parameters such as compression color values for different levels, resizing factors, color choices, and compression choices, which are essential for tailoring the behavior of image processing operations. The class also manages the input image file path, allowing it to reference the specific files needed for processing. A regular expression pattern is included for validating or extracting configuration details, and a set of required variables is maintained to ensure that all necessary configuration options are present. The class provides several getter methods to access these stored settings, ensuring that they can be easily retrieved by other parts of the application. Additionally, it includes a method to read and validate the configuration file, ensuring that all required parameters are correctly set up before proceeding with any image processing tasks.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 ConfigReader()

```
km::ConfigReader::ConfigReader ( )
```

Here is the call graph for this function:



3.1.3 Member Function Documentation

3.1.3.1 checkVariableExists()

```
auto km::ConfigReader::checkVariableExists (
              const std::string & variableName ) const -> bool  [private]
```

3.1.3.2 getColorChoice()

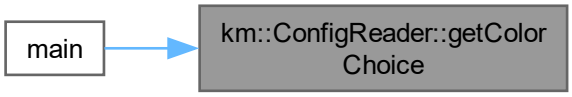
```
auto km::ConfigReader::getColorChoice ( ) const -> int
```

Gets the color choice.

Returns

Color choice

Here is the caller graph for this function:



3.1.3.3 getCompressionChoice()

```
auto km::ConfigReader::getCompressionChoice ( ) const -> int
```

Gets the compression choice.

Returns

Compression choice

3.1.3.4 getFifthLevelCompressionColor()

```
auto km::ConfigReader::getFifthLevelCompressionColor ( ) const -> double
```

Gets the fifth level compression color value.

Returns

Fifth level compression color value

Here is the caller graph for this function:

3.1.3.5 getFirstLevelCompressionColor()

```
auto km::ConfigReader::getFirstLevelCompressionColor ( ) const -> double
```

Gets the first level compression color value.

Returns

First level compression color value

Here is the caller graph for this function:

3.1.3.6 getFourthLevelCompressionColor()

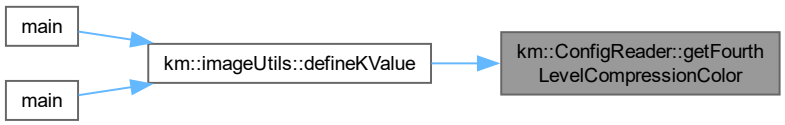
```
auto km::ConfigReader::getFourthLevelCompressionColor ( ) const -> double
```

Gets the fourth level compression color value.

Returns

Fourth level compression color value

Here is the caller graph for this function:



3.1.3.7 getInputImageFilePath()

```
auto km::ConfigReader::getInputImageFilePath ( ) const -> std::filesystem::path
```

Gets the input image file path.

Returns

Input image file path

3.1.3.8 getResizingFactor()

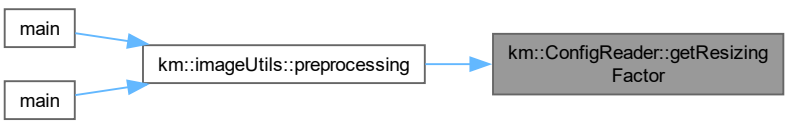
```
auto km::ConfigReader::getResizingFactor ( ) const -> double
```

Gets the resizing factor.

Returns

Resizing factor

Here is the caller graph for this function:



3.1.3.9 getSecondLevelCompressionColor()

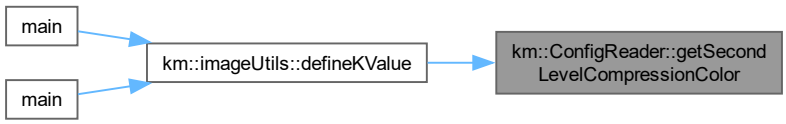
```
auto km::ConfigReader::getSecondLevelCompressionColor ( ) const -> double
```

Gets the second level compression color value.

Returns

Second level compression color value

Here is the caller graph for this function:



3.1.3.10 getThirdLevelCompressionColor()

```
auto km::ConfigReader::getThirdLevelCompressionColor ( ) const -> double
```

Gets the third level compression color value.

Returns

Third level compression color value

Here is the caller graph for this function:

3.1.3.11 readConfigFile()

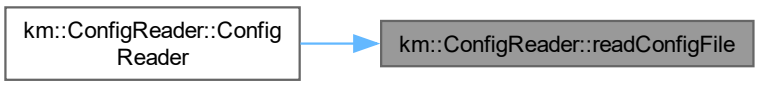
```
auto km::ConfigReader::readConfigFile ( ) -> bool
```

Reads the configuration file.

Returns

True if the configuration file is read successfully, false otherwise

Here is the caller graph for this function:



3.1.4 Member Data Documentation

3.1.4.1 color_choice

```
int km::ConfigReader::color_choice = 0  [private]
```

Color choice.

3.1.4.2 compression_choice

```
int km::ConfigReader::compression_choice = 0  [private]
```

Compression choice.

3.1.4.3 fifth_level_compression_color

```
double km::ConfigReader::fifth_level_compression_color = 0.  [private]
```

Fifth level compression color value.

3.1.4.4 first_level_compression_color

```
double km::ConfigReader::first_level_compression_color = 0.  [private]
```

First level compression color value.

3.1.4.5 fourth_level_compression_color

```
double km::ConfigReader::fourth_level_compression_color = 0.  [private]
```

Fourth level compression color value.

3.1.4.6 inputImageFilePath

```
std::filesystem::path km::ConfigReader::inputImageFilePath  [private]
```

Input image file path.

3.1.4.7 pattern

```
std::regex km::ConfigReader::pattern  [private]
```

Regular expression pattern.

3.1.4.8 requiredVariables

```
std::unordered_set<std::string> km::ConfigReader::requiredVariables = {}  [private]
```

Set of required variables.

3.1.4.9 resizing_factor

```
double km::ConfigReader::resizing_factor = 0.  [private]
```

Resizing factor.

3.1.4.10 second_level_compression_color

```
double km::ConfigReader::second_level_compression_color = 0.  [private]
```

Second level compression color value.

3.1.4.11 third_level_compression_color

```
double km::ConfigReader::third_level_compression_color = 0.  [private]
```

Third level compression color value.

The documentation for this class was generated from the following files:

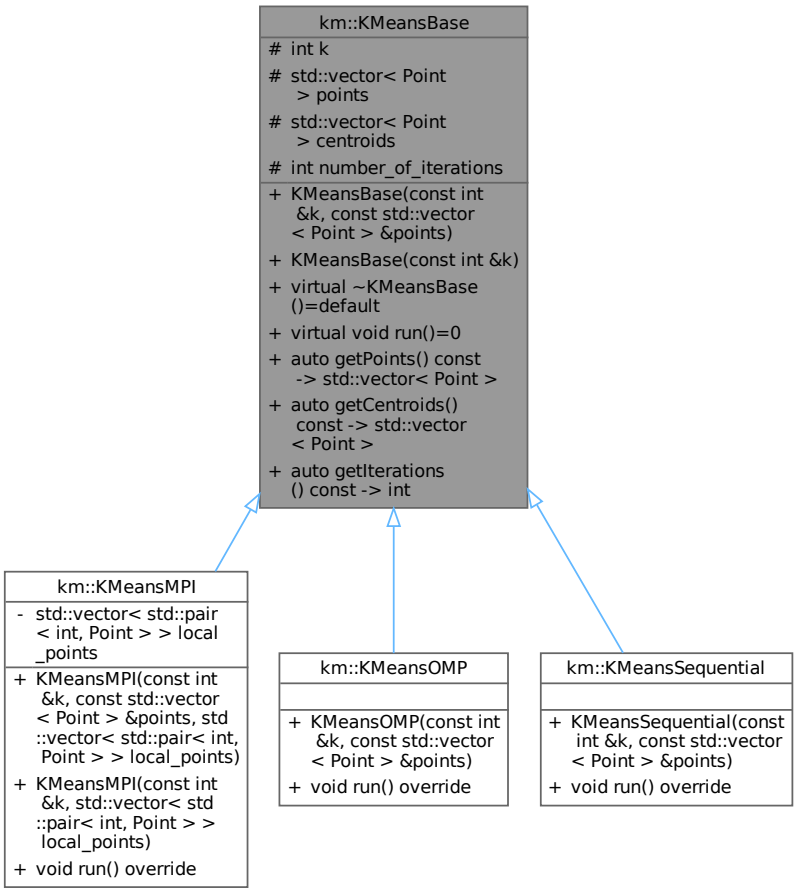
- include/configReader.hpp
- src/configReader.cpp

3.2 km::KMeansBase Class Reference

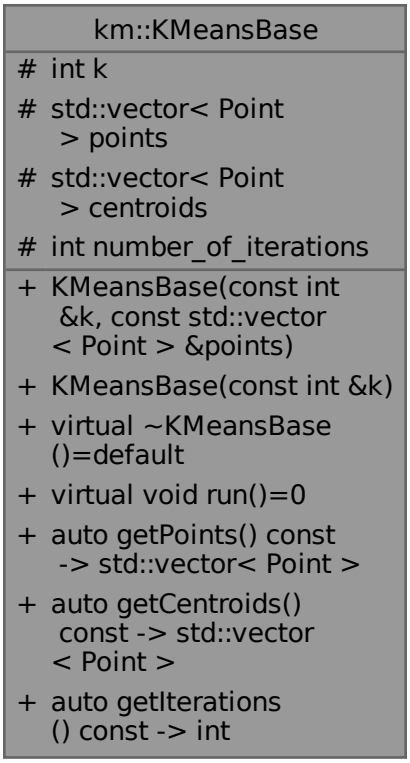
Base class for K-means clustering algorithm.

`#include <kMeansBase.hpp>`

Inheritance diagram for km::KMeansBase:



Collaboration diagram for km::KMeansBase:



Public Member Functions

- KMeansBase (const int &k, const std::vector< Point > &points)

  Constructor for KMeansBase.
- KMeansBase (const int &k)

  Constructs a KMeansBase object only with the specified number of clusters.
- virtual ~KMeansBase ()=default

  Virtual destructor for KMeansBase.
- virtual void run ()=0

  Runs the K-means clustering algorithm.
- auto getPoints () const -> std::vector< Point >

  Gets the poinots.
- auto getCentroids () const -> std::vector< Point >

  Gets the centroids.
- auto getIterations () const -> int

  Gets the number of iterations.

Protected Attributes

- int k

  Number of clusters.

- std::vector< Point > points

  Vector of points.
- std::vector< Point > centroids

  Vector of centroids.
- int number_of_iterations

  Number of iterations.

3.2.1 Detailed Description

Base class for K-means clustering algorithm.

The KMeansBase class, part of the km namespace, serves as a foundational class for implementing the K-means clustering algorithm. It is designed to manage and execute the clustering process, providing a base for derived classes to build upon with specific implementations. The class includes several key functionalities: it allows for the construction of an object with either a predefined number of clusters and a set of points or just the number of clusters. The run method, which is a pure virtual function, must be implemented by any derived class to execute the K-means algorithm. This structure ensures that the base class can provide the essential setup and data management, while specific clustering logic is handled by subclasses. The KMeansBase class also includes methods to retrieve the points used for clustering, the centroids calculated by the algorithm, and the number of iterations the algorithm has undergone. These methods provide access to the internal state of the clustering process, enabling users to inspect and analyze the results. Protected member variables include the number of clusters, the points to be clustered, the centroids resulting from the clustering process, and the count of iterations performed, allowing derived classes to access and manipulate these values as needed.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 KMeansBase() [1/2]

```
km::KMeansBase::KMeansBase (
            const int & k,
            const std::vector< Point > & points )
```

Constructor for KMeansBase.

Parameters

k	Number of clusters
points	Vector of points

3.2.2.2 KMeansBase() [2/2]

```
km::KMeansBase::KMeansBase (
            const int & k )
```

Constructs a KMeansBase object only with the specified number of clusters.

Parameters

k	The number of clusters.

3.2.2.3 ~KMeansBase()

```
virtual km::KMeansBase::~KMeansBase ( )  [virtual], [default]
```

Virtual destructor for KMeansBase.

3.2.3 Member Function Documentation

3.2.3.1 getCentroids()

```
std::vector< km::Point > km::KMeansBase::getCentroids ( ) const -> std::vector<Point>
```

Gets the centroids.

Returns

Vector of centroids

Here is the caller graph for this function:

3.2.3.2 getIterations()

```
int km::KMeansBase::getIterations ( ) const -> int
```

Gets the number of iterations.

Returns

Number of iterations

Here is the caller graph for this function:

3.2.3.3 getPoints()

```
std::vector< km::Point > km::KMeansBase::getPoints ( ) const -> std::vector<Point>
```

Gets the poinots.

Returns

Vector of points

Here is the caller graph for this function:

3.2.3.4 run()

```
virtual void km::KMeansBase::run ( )  [pure virtual]
```

Runs the K-means clustering algorithm.

Implemented in km::KMeansMPI, km::KMeansOMP, and km::KMeansSequential.

3.2.4 Member Data Documentation

3.2.4.1 centroids

```
std::vector<Point> km::KMeansBase::centroids  [protected]
```

Vector of centroids.

3.2.4.2 k

```
int km::KMeansBase::k  [protected]
```

Number of clusters.

3.2.4.3 number_of_iterations

```
int km::KMeansBase::number_of_iterations  [protected]
```

Number of iterations.

3.2.4.4 points

```
std::vector<Point> km::KMeansBase::points  [protected]
```

Vector of points.

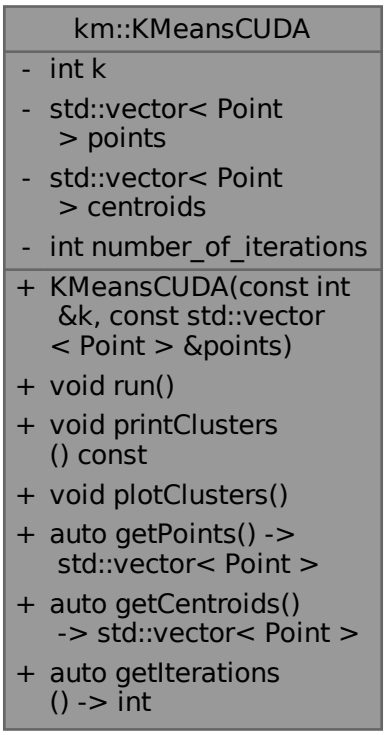
The documentation for this class was generated from the following files:

- include/kMeansBase.hpp
- src/kMeansBase.cpp

3.3 km::KMeansCUDA Class Reference

Represents the K-means clustering algorithm using CUDA.

Collaboration diagram for km::KMeansCUDA:



Public Member Functions

- KMeansCUDA (const int &k, const std::vector< Point > &points)
  Constructor for KMeans.
- void run ()
  Runs the K-means clustering algorithm using CUDA.
- void printClusters () const
  Prints the clusters.
- void plotClusters ()
  Plots the clusters.
- auto getPoints () -> std::vector< Point >
  Gets the points.
- auto getCentroids () -> std::vector< Point >
  Gets the centroids.
- auto getIterations () -> int
  Gets the number of iterations.

Private Attributes

- int k
  Number of clusters.
- std::vector< Point > points
  Vector of points.
- std::vector< Point > centroids
  Vector of centroids.
- int number_of_iterations
  Number of iterations.

3.3.1 Detailed Description

Represents the K-means clustering algorithm using CUDA.

The KMeansCUDA class, located in the km namespace, is designed to implement the K-means clustering algorithm using CUDA for enhanced performance through parallel processing on GPUs. This class extends the functionality of traditional K-means clustering by leveraging CUDA to accelerate computations, making it suitable for large datasets and complex clustering tasks. The class provides a constructor that initializes the number of clusters and the vector of points to be clustered. It includes a run method that executes the K-means algorithm on the GPU, performing the clustering operations efficiently by taking advantage of parallel processing capabilities. Additionally, it offers methods to print and plot the clusters, allowing users to visualize the results of the clustering process. The getPoints, getCentroids, and getIterations methods provide access to the internal state of the clustering, including the input points, the resulting centroids, and the number of iterations the algorithm has undergone, respectively. This design ensures that users can both run and analyze the K-means clustering process using CUDA for improved performance.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 KMeansCUDA()

```
km::KMeansCUDA::KMeansCUDA (
                const int & k,
                const std::vector< Point > & points )
```

Constructor for KMeans.

Parameters

k	Number of clusters
points	Vector of points

3.3.3 Member Function Documentation

3.3.3.1 getCentroids()

```
auto km::KMeansCUDA::getCentroids ( ) -> std::vector<Point>
```

Gets the centroids.

Returns

Vector of centroids

Here is the caller graph for this function:

3.3.3.2 getIterations()

```
auto km::KMeansCUDA::getIterations ( ) -> int
```

Gets the number of iterations.

Returns

Number of iterations

Here is the caller graph for this function:

3.3.3.3 getPoints()

```
auto km::KMeansCUDA::getPoints ( ) -> std::vector<Point>
```

Gets the points.

Returns

Vector of points

Here is the caller graph for this function:

3.3.3.4 plotClusters()

```
void km::KMeansCUDA::plotClusters ( )
```

Plots the clusters.

3.3.3.5 printClusters()

```
void km::KMeansCUDA::printClusters ( ) const
```

Prints the clusters.

3.3.3.6 run()

```
void km::KMeansCUDA::run ( )
```

Runs the K-means clustering algorithm using CUDA.

Here is the call graph for this function:

Here is the caller graph for this function:

3.3.4 Member Data Documentation

3.3.4.1 centroids

```
std::vector<Point> km::KMeansCUDA::centroids  [private]
```

Vector of centroids.

3.3.4.2 k

```
int km::KMeansCUDA::k  [private]
```

Number of clusters.

3.3.4.3 number_of_iterations

```
int km::KMeansCUDA::number_of_iterations  [private]
```

Number of iterations.

3.3.4.4 points

```
std::vector<Point> km::KMeansCUDA::points  [private]
```

Vector of points.

The documentation for this class was generated from the following files:

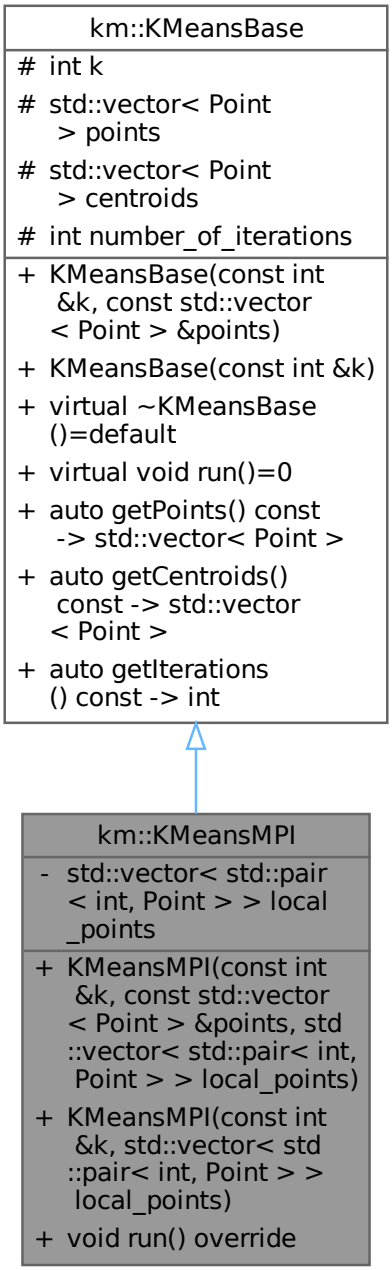
- include/kMeansCUDA.cuh
- src/kMeansCUDA.cu

3.4 km::KMeansMPI Class Reference

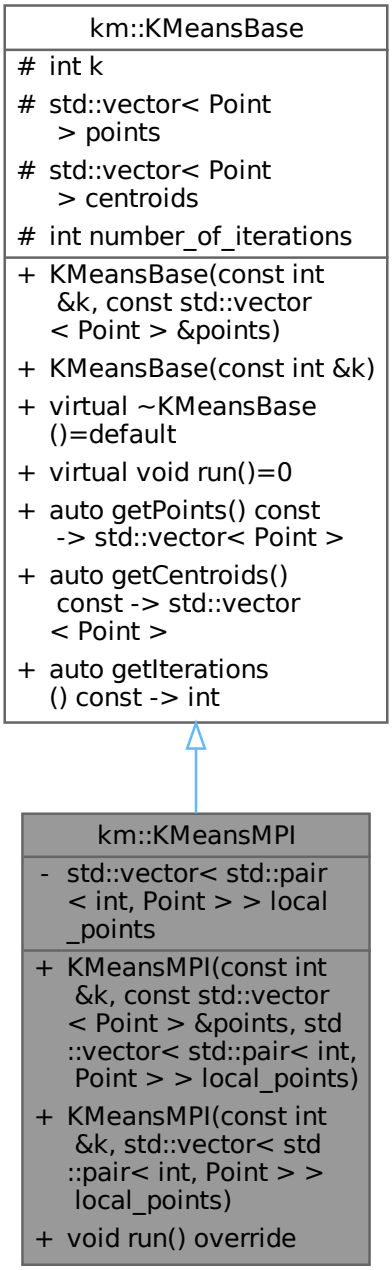
Represents the K-means clustering algorithm using MPI.

`#include <kMeansMPI.hpp>`

Inheritance diagram for km::KMeansMPI:



Collaboration diagram for km::KMeansMPI:



Public Member Functions

- KMeansMPI (const int &k, const std::vector< Point > &points, std::vector< std::pair< int, Point > > local_points)

  Constructor for KMeansMPI.
- KMeansMPI (const int &k, std::vector< std::pair< int, Point > > local_points)

  Constructor for KMeansMPI.
- void run () override

  Runs the K-means clustering algorithm using MPI.

Public Member Functions inherited from km::KMeansBase

- KMeansBase (const int &k, const std::vector< Point > &points)
  Constructor for KMeansBase.
- KMeansBase (const int &k)
  Constructs a KMeansBase object only with the specified number of clusters.
- virtual ~KMeansBase ()=default
  Virtual destructor for KMeansBase.
- auto getPoints () const -> std::vector< Point >
  Gets the poinots.
- auto getCentroids () const -> std::vector< Point >
  Gets the centroids.
- auto getIterations () const -> int
  Gets the number of iterations.

Private Attributes

- std::vector< std::pair< int, Point > > local_points
  Vector of local points.

Additional Inherited Members

Protected Attributes inherited from km::KMeansBase

- int k
  Number of clusters.
- std::vector< Point > points
  Vector of points.
- std::vector< Point > centroids
  Vector of centroids.
- int number_of_iterations
  Number of iterations.

3.4.1 Detailed Description

Represents the K-means clustering algorithm using MPI.

The KMeansMPI class, located in the km namespace, is designed to implement the K-means clustering algorithm using MPI (Message Passing Interface) for parallel and distributed computing. This class extends the base KMeansBase class to enable clustering operations across multiple processes, leveraging MPI to handle large-scale data and computational tasks more efficiently. The class includes two constructors: one that initializes the number of clusters, a vector of points, and a vector of local points distributed across MPI processes; and another that initializes only the number of clusters and local points. The run method, overridden from KMeansBase, is responsible for executing the K-means clustering algorithm using MPI, coordinating the clustering process across different processes in a distributed computing environment. The local_points member variable holds the points assigned to each MPI process, enabling the parallel execution of clustering tasks. This class is designed to handle clustering in a distributed setting, allowing for efficient processing of large datasets by distributing the workload across multiple computing nodes.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 KMeansMPI() [1/2]

```
km::KMeansMPI::KMeansMPI (
            const int & k,
            const std::vector< Point > & points,
            std::vector< std::pair< int, Point > > local_points )
```

Constructor for KMeansMPI.

Parameters

k	Number of clusters
points	Vector of points

3.4.2.2 KMeansMPI() [2/2]

```
km::KMeansMPI::KMeansMPI (
            const int & k,
            std::vector< std::pair< int, Point > > local_points )
```

Constructor for KMeansMPI.

Parameters

k	Number of clusters

3.4.3 Member Function Documentation

3.4.3.1 run()

```
void km::KMeansMPI::run ( )  [override], [virtual]
```

Runs the K-means clustering algorithm using MPI.

Implements km::KMeansBase.

Here is the call graph for this function:

3.4.4 Member Data Documentation

3.4.4.1 local_points

```
std::vector<std::pair<int, Point> > km::KMeansMPI::local_points  [private]
```

Vector of local points.

The documentation for this class was generated from the following files:

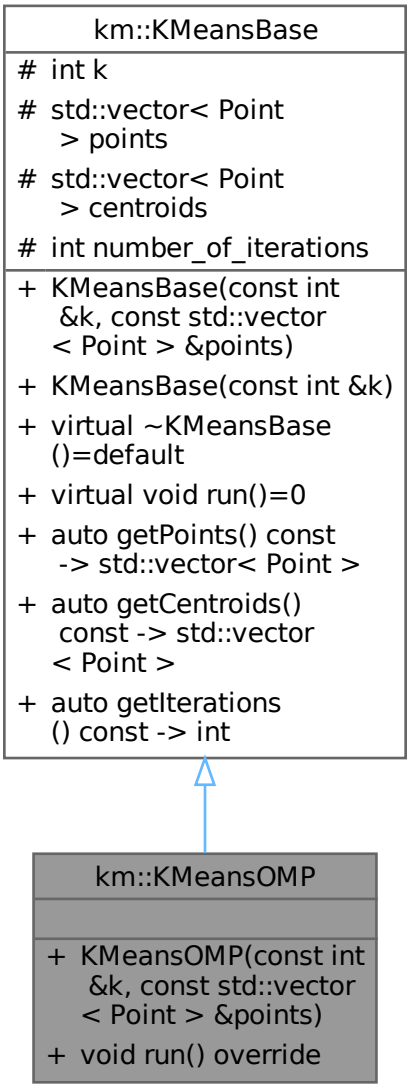
- include/kMeansMPI.hpp
- src/kMeansMPI.cpp

3.5 km::KMeansOMP Class Reference

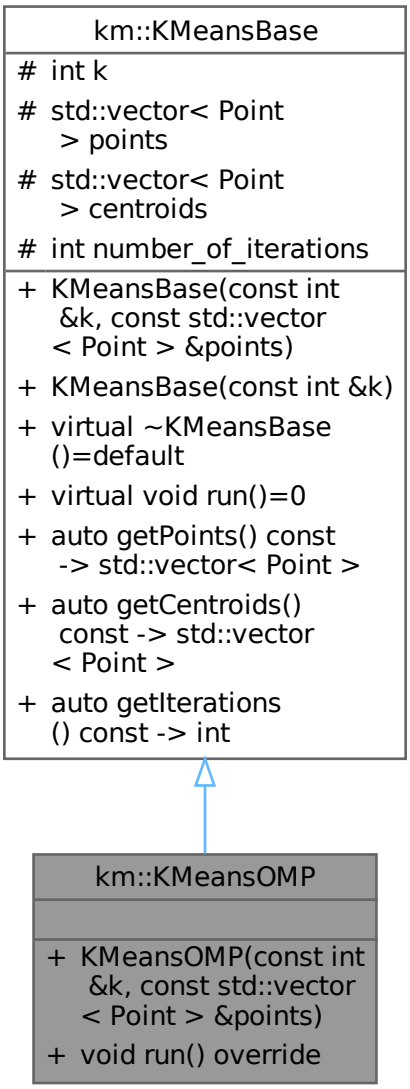
Represents the K-means clustering algorithm using OpenMP.

`#include <kMeansOMP.hpp>`

Inheritance diagram for km::KMeansOMP:



Collaboration diagram for km::KMeansOMP:



Public Member Functions

- KMeansOMP (const int &k, const std::vector< Point > &points)

  Constructor for KMeansOMP.
- void run () override

  Runs the K-means clustering algorithm using OpenMP.

Public Member Functions inherited from km::KMeansBase

- KMeansBase (const int &k, const std::vector< Point > &points)

  Constructor for KMeansBase.
- KMeansBase (const int &k)

Constructs a KMeansBase object only with the specified number of clusters.

- virtual ~KMeansBase ()=default

  Virtual destructor for KMeansBase.

- auto getPoints () const -> std::vector< Point >

  Gets the poinots.

- auto getCentroids () const -> std::vector< Point >

  Gets the centroids.

- auto getIterations () const -> int

  Gets the number of iterations.

Additional Inherited Members

Protected Attributes inherited from km::KMeansBase

- int k

  Number of clusters.

- std::vector< Point > points

  Vector of points.

- std::vector< Point > centroids

  Vector of centroids.

- int number_of_iterations

  Number of iterations.

3.5.1 Detailed Description

Represents the K-means clustering algorithm using OpenMP.

The KMeansOMP class, part of the km namespace, is designed to implement the K-means clustering algorithm using OpenMP, a parallel programming model for shared-memory architectures. This class extends the KMeansBase class to utilize OpenMP for parallelizing the clustering process, which can significantly speed up computations by leveraging multi-core processors. The class features a constructor that initializes the number of clusters and the vector of points to be clustered. The run method, which overrides the base class method, is responsible for executing the K-means clustering algorithm with parallelization support provided by OpenMP. This allows the algorithm to handle clustering operations more efficiently by distributing computational tasks across multiple threads. By integrating OpenMP, the KMeansOMP class aims to enhance the performance of the K-means clustering algorithm, making it suitable for processing larger datasets and improving computational efficiency in environments with multi-core CPUs.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 KMeansOMP()

```
km::KMeansOMP::KMeansOMP (
            const int & k,
            const std::vector< Point > & points )
```

Constructor for KMeansOMP.

Parameters

k	Number of clusters
points	Vector of points

3.5.3 Member Function Documentation

3.5.3.1 run()

```
void km::KMeansOMP::run ( )  [override], [virtual]
```

Runs the K-means clustering algorithm using OpenMP.

Implements km::KMeansBase.

Here is the call graph for this function:

Here is the caller graph for this function:

The documentation for this class was generated from the following files:

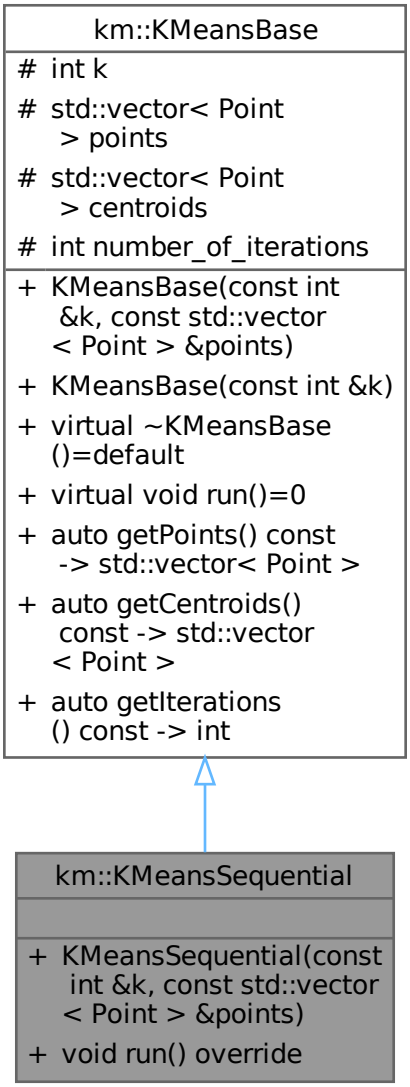
- include/kMeansOMP.hpp
- src/kMeansOMP.cpp

3.6 km::KMeansSequential Class Reference

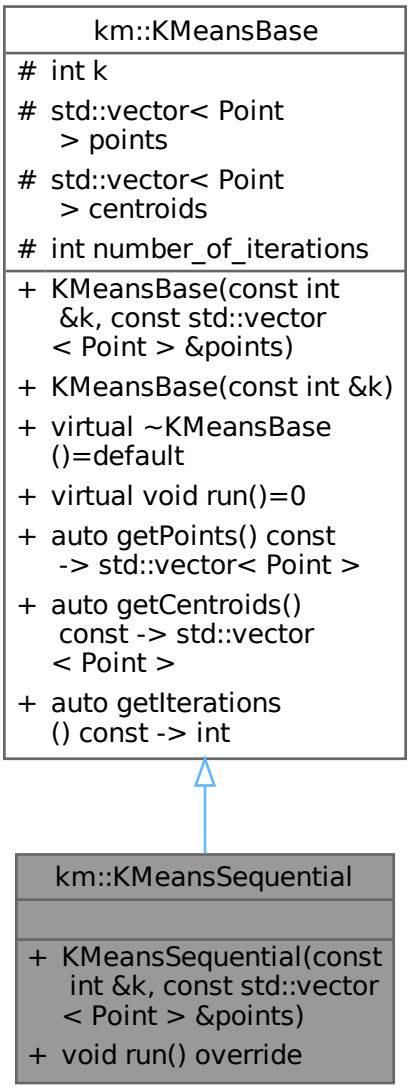
Represents the K-means clustering algorithm.

```
#include <kMeansSequential.hpp>
```

Inheritance diagram for km::KMeansSequential:



Collaboration diagram for km::KMeansSequential:



Public Member Functions

- KMeansSequential (const int &k, const std::vector< Point > &points)

  Constructor for KMeansSequential.
- void run () override

  Runs the K-means clustering algorithm.

Public Member Functions inherited from km::KMeansBase

- KMeansBase (const int &k, const std::vector< Point > &points)

  Constructor for KMeansBase.
- KMeansBase (const int &k)

Constructs a KMeansBase object only with the specified number of clusters.

- virtual ~KMeansBase ()=default

  Virtual destructor for KMeansBase.

- auto getPoints () const -> std::vector< Point >

  Gets the poinots.

- auto getCentroids () const -> std::vector< Point >

  Gets the centroids.

- auto getIterations () const -> int

  Gets the number of iterations.

Additional Inherited Members

Protected Attributes inherited from km::KMeansBase

- int k

  Number of clusters.

- std::vector< Point > points

  Vector of points.

- std::vector< Point > centroids

  Vector of centroids.

- int number_of_iterations

  Number of iterations.

3.6.1 Detailed Description

Represents the K-means clustering algorithm.

The KMeansSequential class, within the km namespace, provides a straightforward implementation of the K-means clustering algorithm. This class extends the KMeansBase class to implement the algorithm in a sequential manner, meaning that it performs all computations in a single-threaded, non-parallel fashion. The class includes a constructor that initializes the number of clusters and the vector of points to be clustered. The run method, which overrides the virtual method from KMeansBase, is responsible for executing the K-means clustering algorithm in a sequential, step-by-step process. This implementation is suitable for environments where parallel processing is not available or necessary, and it provides a foundational approach to K-means clustering that can be used for benchmarking or as a baseline for more complex implementations. The KMeansSequential class serves as a basic and direct implementation of K-means clustering, focusing on clarity and correctness of the algorithm in a non-parallelized context.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 KMeansSequential()

```
km::KMeansSequential::KMeansSequential (
            const int & k,
            const std::vector< Point > & points )
```

Constructor for KMeansSequential.

Parameters

k	Number of clusters
points	Vector of points

3.6.3 Member Function Documentation

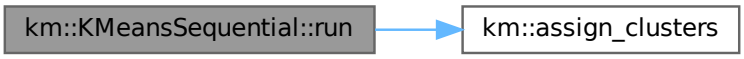
3.6.3.1 run()

```
void km::KMeansSequential::run ( )  [override], [virtual]
```

Runs the K-means clustering algorithm.

Implements km::KMeansBase.

Here is the call graph for this function:



Here is the caller graph for this function:

The documentation for this class was generated from the following files:

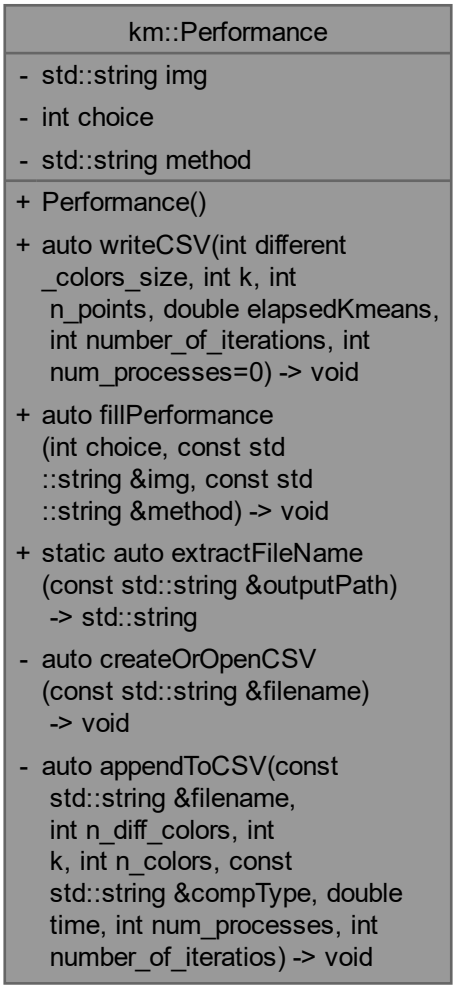
- include/kMeansSequential.hpp
- src/kMeansSequential.cpp

3.7 km::Performance Class Reference

Represents the performance evaluation.

```
#include <performanceEvaluation.hpp>
```

Collaboration diagram for km::Performance:



Public Member Functions

- Performance ()

  Default constructor.
- auto writeCSV (int different_colors_size, int k, int n_points, double elapsedKmeans, int number_of_iterations, int num_processes=0) -> void

  Writes performance data to a CSV file.
- auto fillPerformance (int choice, const std::string &img, const std::string &method) -> void

  Fills the performance data.

Static Public Member Functions

- static auto extractFileName (const std::string &outputPath) -> std::string

  Extracts the file name from the output path.

Private Member Functions

- auto createOrOpenCSV (const std::string &filename) -> void

  Creates or opens a CSV file.
- auto appendToCSV (const std::string &filename, int n_diff_colors, int k, int n_colors, const std::string &compType, double time, int num_processes, int number_of_iteratios) -> void

  Appends performance data to the CSV file.

Private Attributes

- std::string img

  Image name.
- int choice {}

  Choice of performance evaluation.
- std::string method

  Method used.

3.7.1 Detailed Description

Represents the performance evaluation.

The Performance class in the km namespace is designed for evaluating and recording the performance of clustering algorithms. It includes methods to write performance data to a CSV file, extract file names from paths, and fill in performance metrics based on various criteria. The class has a default constructor and methods for writing data to a CSV file, such as writeCSV for recording performance metrics, and fillPerformance for populating evaluation data. Private methods handle file operations, including creating or opening CSV files and appending data. The class manages internal details like image names and evaluation choices for performance analysis.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 Performance()

```
km::Performance::Performance ( )  [default]
```

Default constructor.

3.7.3 Member Function Documentation

3.7.3.1 appendToCSV()

```
void km::Performance::appendToCSV (
            const std::string & filename,
            int n_diff_colors,
            int k,
            int n_colors,
            const std::string & compType,
            double time,
            int num_processes,
            int number_of_iteratios ) -> void  [private]
```

Appends performance data to the CSV file.

Parameters

filename	Name of the CSV file
n_diff_colors	Number of different colors
k	Number of clusters
n_colors	Number of colors
compType	Compression type
time	Elapsed time
num_processes	Number of processes

Here is the call graph for this function:

3.7.3.2 createOrOpenCSV()

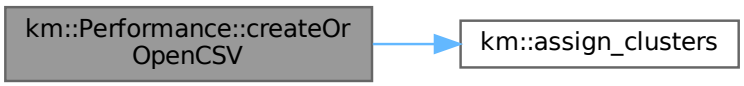
```
void km::Performance::createOrOpenCSV (
            const std::string & filename ) -> void  [private]
```

Creates or opens a CSV file.

Parameters

filename	Name of the CSV file

Here is the call graph for this function:



3.7.3.3 extractFileName()

```
auto km::Performance::extractFileName (
            const std::string & outputPath ) -> std::string  [static]
```

Extracts the file name from the output path.

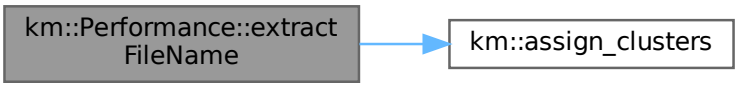
Parameters

outputPath	Output path

Returns

Extracted file name

Here is the call graph for this function:



Here is the caller graph for this function:

3.7.3.4 fillPerformance()

```
void km::Performance::fillPerformance (
                int choice,
                const std::string & img,
                const std::string & method ) -> void
```

Fills the performance data.

Parameters

choice	Choice of performance evaluation
img	Image name
method	Method used

Here is the caller graph for this function:

3.7.3.5 writeCSV()

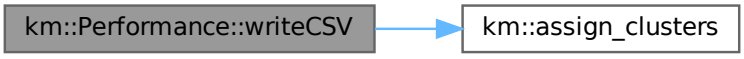
```
void km::Performance::writeCSV (
                int different_colors_size,
                int k,
                int n_points,
                double elapsedKmeans,
                int number_of_iterations,
                int num_processes = 0 ) -> void
```

Writes performance data to a CSV file.

Parameters

different_colors_size	Number of different colors
k	Number of clusters
n_points	Number of points
elapsedKmeans	Elapsed time for K-means clustering
num_processes	Number of processes (optional, default=0)

Here is the call graph for this function:



Here is the caller graph for this function:

3.7.4 Member Data Documentation

3.7.4.1 choice

```
int km::Performance::choice {}  [private]
```

Choice of performance evaluation.

3.7.4.2 img

```
std::string km::Performance::img  [private]
```

Image name.

3.7.4.3 method

```
std::string km::Performance::method  [private]
```

Method used.

The documentation for this class was generated from the following files:

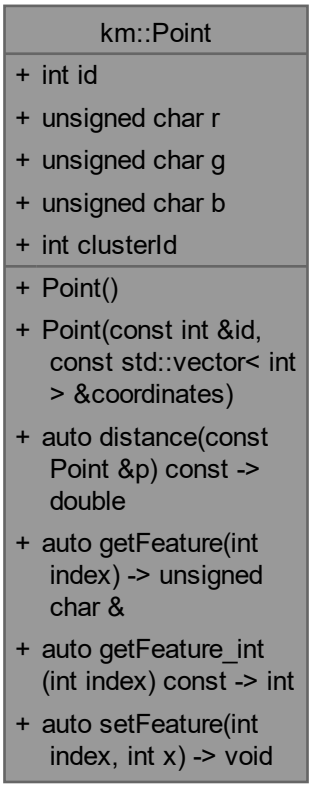
- include/performanceEvaluation.hpp
- src/performanceEvaluation.cpp

3.8 km::Point Class Reference

Represents a point in a feature space.

`#include <point.hpp>`

Collaboration diagram for km::Point:



Public Member Functions

- Point ()
  Constructor for Point.
- Point (const int &id, const std::vector< int > &coordinates)
  Constructor for Point.
- auto distance (const Point &p) const -> double
  Calculates the distance between this point and another point.
- auto getFeature (int index) -> unsigned char &
  Gets a feature value at the specified index.
- auto getFeature_int (int index) const -> int
  Gets a feature value as an integer at the specified index.
- auto setFeature (int index, int x) -> void
  Sets a feature value at the specified index.

Public Attributes

- int id {0}
  ID of the point.
- unsigned char r {0}
  Red component.
- unsigned char g {0}
  Green component.
- unsigned char b {0}
  Blue component.
- int clusterId {-1}
  ID of the cluster the point belongs to.

3.8.1 Detailed Description

Represents a point in a feature space.

The Point class in the km namespace represents a point in a feature space, with attributes including an ID, RGB color components, and a cluster ID. It features a default constructor and a parameterized constructor for initializing points with specific IDs and coordinates. The class includes methods to compute the distance between two points, retrieve and set feature values, and access feature values as integers. These functionalities facilitate the manipulation and analysis of points within clustering algorithms and other feature-based computations.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 Point() [1/2]

```
km::Point::Point ( )  [default]
```

Constructor for Point.

Parameters

features_size	Number of features

3.8.2.2 Point() [2/2]

```
km::Point::Point (
            const int & id,
            const std::vector< int > & coordinates )
```

Constructor for Point.

Parameters

id	ID of the point
coordinates	Coordinates of the point

3.8.3 Member Function Documentation

3.8.3.1 distance()

```
auto km::Point::distance (
            const Point & p ) const -> double
```

Calculates the distance between this point and another point.

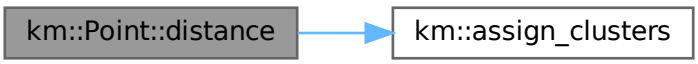
Parameters

p	Other point

Returns

Distance between the points

Here is the call graph for this function:



3.8.3.2 getFeature()

```
auto km::Point::getFeature (
            int index ) -> unsigned char &
```

Gets a feature value at the specified index.

Parameters

index	Index of the feature

Returns

Feature value

3.8.3.3 getFeature_int()

```
auto km::Point::getFeature_int (
            int index ) const -> int
```

Gets a feature value as an integer at the specified index.

Parameters

index	Index of the feature

Returns

Feature value as an integer

3.8.3.4 setFeature()

```
void km::Point::setFeature (
               int index,
               int x ) -> void
```

Sets a feature value at the specified index.

Parameters

index	Index of the feature
x	Feature value

Here is the call graph for this function:

3.8.4 Member Data Documentation

3.8.4.1 b

```
unsigned char km::Point::b {0}
```

Blue component.

3.8.4.2 clusterId

```
int km::Point::clusterId {-1}
```

ID of the cluster the point belongs to.

3.8.4.3 g

```
unsigned char km::Point::g {0}
```

Green component.

3.8.4.4 id

```
int km::Point::id {0}
```

ID of the point.

3.8.4.5 r

```
unsigned char km::Point::r {0}
```

Red component.

The documentation for this class was generated from the following files:

- include/point.hpp
- src/point.cpp

Chapter 4

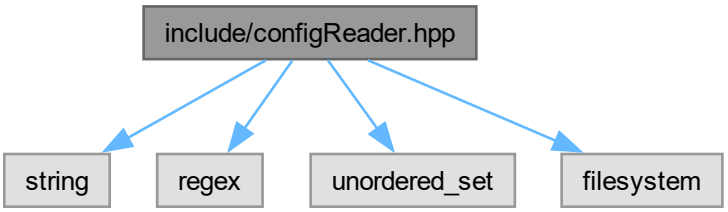
File Documentation

4.1 include/configReader.hpp File Reference

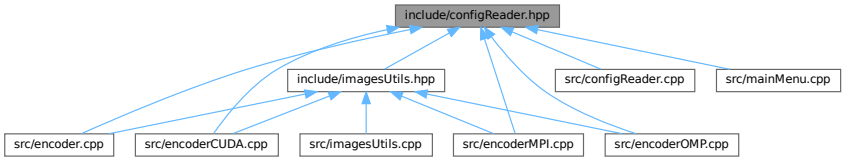
ConfigReader class declaration.

```
#include <string>
#include <regex>
#include <unordered_set>
#include <filesystem>
```

Include dependency graph for configReader.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class km::ConfigReader

  Reads and stores configuration values from a file.

Namespaces

- namespace km

  Main namespace for the project.

4.1.1 Detailed Description

ConfigReader class declaration.

4.2 configReader.hpp

Go to the documentation of this file.

```
00001
00006 #ifndef CONFIG_READER_HPP
00007 #define CONFIG_READER_HPP
00008
00009 #include <string>
00010 #include <regex>
00011 #include <unordered_set>
00012 #include <filesystem>
00013
00014 namespace km
00015 {
00028     class ConfigReader
00029     {
00030     private:
00031         double first_level_compression_color = 0.;
00032         double second_level_compression_color = 0.;
00033         double third_level_compression_color = 0.;
00034         double fourth_level_compression_color = 0.;
00035         double fifth_level_compression_color = 0.;
00036         double resizing_factor = 0.;
00037         int color_choice = 0;
00038         int compression_choice = 0;
00039         std::filesystem::path inputImageFilePath;
00040         std::regex pattern;
00041         std::unordered_set<std::string> requiredVariables = {};
00042
00043         [[nodiscard]] auto checkVariableExists(const std::string &variableName) const -> bool;
00044
00050     public:
00051         [[nodiscard]] auto getFirstLevelCompressionColor() const -> double;
00052
00057         [[nodiscard]] auto getSecondLevelCompressionColor() const -> double;
00058
00063         [[nodiscard]] auto getThirdLevelCompressionColor() const -> double;
00064
00069         [[nodiscard]] auto getFourthLevelCompressionColor() const -> double;
00070
00075         [[nodiscard]] auto getFifthLevelCompressionColor() const -> double;
00076
00081         [[nodiscard]] auto getColorChoice() const -> int;
00082
00087         [[nodiscard]] auto getCompressionChoice() const -> int;
00088
00093         [[nodiscard]] auto getInputImageFilePath() const -> std::filesystem::path;
00094
00099         [[nodiscard]] auto getResizingFactor() const -> double;
00100
00105         [[nodiscard]] auto readConfigFile() -> bool;
00106
00107         ConfigReader();
00108     };
00109 } // namespace km
00110
00111 #endif // CONFIG_READER_HPP
```

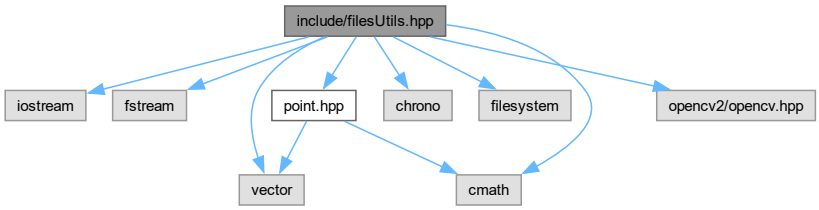
4.3 include/filesUtils.hpp File Reference

Utility functions for file handling.

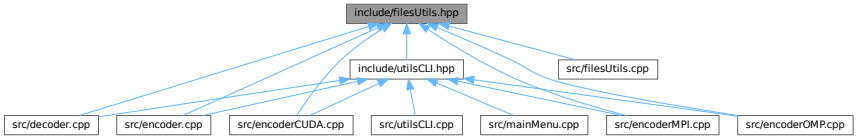
```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <chrono>
```

```
#include <filesystem>
#include <point.hpp>
#include <opencv2/opencv.hpp>
```

Include dependency graph for filesUtils.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace km::filesUtils

  Provides utility functions for file handling.
- namespace km

  Main namespace for the project.

Functions

- auto km::filesUtils::createOutputDirectories () -> void

  Creates output directories.
- auto km::filesUtils::writeBinaryFile (std::string &outputPath, int &width, int &height, int &k, std::vector< Point > points, std::vector< Point > centroids) -> void

  Writes data to a binary file.
- auto km::filesUtils::isCorrectExtension (const std::filesystem::path &filePath, const std::string &correctExtension) -> bool

  Checks if a file has the correct extension.
- auto km::filesUtils::createDecodingMenu (std::filesystem::path &decodeDir, std::vector< std::filesystem::path > &imageNames) -> void

  Creates a decoding menu.
- auto km::filesUtils::readBinaryFile (std::string &path, cv::Mat &imageCompressed) -> int

  Reads a binary file and reconstructs the compressed image.

4.3.1 Detailed Description

Utility functions for file handling.

4.4 filesUtils.hpp

Go to the documentation of this file.

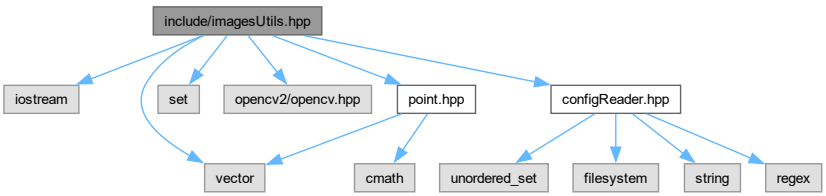
```
00001
00006 #ifndef FILESUTILS_HPP
00007 #define FILESUTILS_HPP
00008
00009 #include <iostream>
00010 #include <fstream>
00011 #include <vector>
00012 #include <cmath>
00013 #include <chrono>
00014 #include <filesystem>
00015 #include <point.hpp>
00016 #include <opencv2/opencv.hpp>
00017
00031 namespace km
00032 {
00033     namespace filesUtils
00034     {
00038         auto createOutputDirectories() -> void;
00039
00049         auto writeBinaryFile(std::string &outputPath, int &width, int &height, int &k, std::vector<Point> points,
      std::vector<Point> centroids) -> void;
00050
00057         auto isCorrectExtension(const std::filesystem::path &filePath, const std::string &correctExtension) -> bool;
00058
00064         auto createDecodingMenu(std::filesystem::path &decodeDir, std::vector<std::filesystem::path> &imageNames) ->
      void;
00065
00072         auto readBinaryFile(std::string &path, cv::Mat &imageCompressed) -> int;
00073     };
00074 }
00075
00076 #endif // FILESUTILS_HPP
```

4.5 include/imagesUtils.hpp File Reference

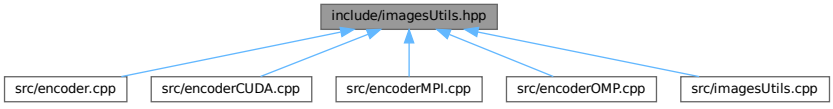
Utility functions for image processing.

```
#include <iostream>
#include <vector>
#include <set>
#include <opencv2/opencv.hpp>
#include <configReader.hpp>
#include <point.hpp>
```

Include dependency graph for imagesUtils.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace km::imageUtils

  Provides utility functions for image processing.
- namespace km

  Main namespace for the project.

Functions

- void km::imageUtils::preprocessing (cv::Mat &image, int &typeCompressionChoice)

  Performs preprocessing on an image.
- void km::imageUtils::defineKValue (int &k, int levelsColorsChoice, std::set< std::vector< unsigned char > > &different_colors)

  Defines the value of K based on the color levels choice.
- void km::imageUtils::pointsFromImage (cv::Mat &image, std::vector< Point > &points, std::set< std::vector< unsigned char > > &different_colors)

  Extracts points from an image.

4.5.1 Detailed Description

Utility functions for image processing.

4.6 imagesUtils.hpp

Go to the documentation of this file.

```
00001
00006 #ifndef IMAGEUTILS_HPP
00007 #define IMAGEUTILS_HPP
00008
00009 #include <iostream>
00010 #include <vector>
00011 #include <set>
00012 #include <opencv2/opencv.hpp>
00013 #include <configReader.hpp>
00014 #include <point.hpp>
00015
00028 namespace km
00029 {
00030     namespace imageUtils
00031     {
00037         void preprocessing(cv::Mat& image, int& typeCompressionChoice);
00038
00045         void defineKValue(int& k, int levelsColorsChoice, std::set<std::vector<unsigned char»& different_colors);
00046
00053         void pointsFromImage(cv::Mat& image, std::vector<Point>& points, std::set<std::vector<unsigned char»&
      different_colors);
00054     };
00055
00056 }
00057
00058 #endif // IMAGEUTILS_HPP
```

4.7 include/kmDocs.hpp File Reference

Documentation for the km namespace.

Namespaces

- namespace km

  Main namespace for the project.

4.7.1 Detailed Description

Documentation for the km namespace.

This file provides comprehensive documentation for the km namespace, which includes utilities for clustering algorithms, file handling, image processing, etc. The km namespace serves as the main container for core functionalities and tools used throughout the project.

4.8 kmDocs.hpp

Go to the documentation of this file.

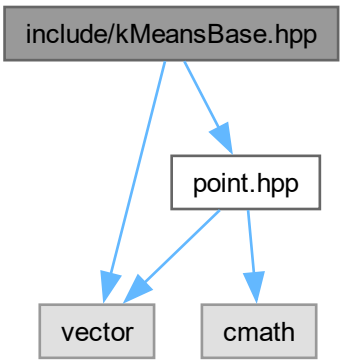
```
00001
00017 namespace km {
00018     // This file is for Doxygen documentation purposes only.
00019 }
```

4.9 include/kMeansBase.hpp File Reference

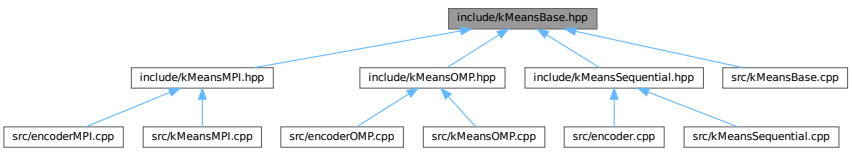
Base class for K-means clustering algorithm.

```
#include <vector>
#include "point.hpp"
```

Include dependency graph for kMeansBase.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class km::KMeansBase

  Base class for K-means clustering algorithm.

Namespaces

- namespace km

  Main namespace for the project.

4.9.1 Detailed Description

Base class for K-means clustering algorithm.

4.10 kMeansBase.hpp

Go to the documentation of this file.

```
00001
00006 #ifndef KMEANS_BASE_HPP
00007 #define KMEANS_BASE_HPP
00008
00009 #include <vector>
00010 #include "point.hpp"
00011
00012 namespace km
00013 {
00029     class KMeansBase
00030     {
00031
00032     public:
00038         KMeansBase(const int &k, const std::vector<Point> &points);
00039
00045         KMeansBase(const int &k);
00046
00050         virtual ~KMeansBase() = default;
00051
00055         virtual void run() = 0;
00056
00061         [[nodiscard]] auto getPoints() const -> std::vector<Point>;
00062
00067         [[nodiscard]] auto getCentroids() const -> std::vector<Point>;
00068
00073         [[nodiscard]] auto getIterations() const -> int;
00074
00075     protected:
00076         int k;
00077         std::vector<Point> points;
00078         std::vector<Point> centroids;
00079         int number_of_iterations;
00080     };
00081 } // namespace k
00082
00083 #endif // KMEANS_BASE_HPP
```

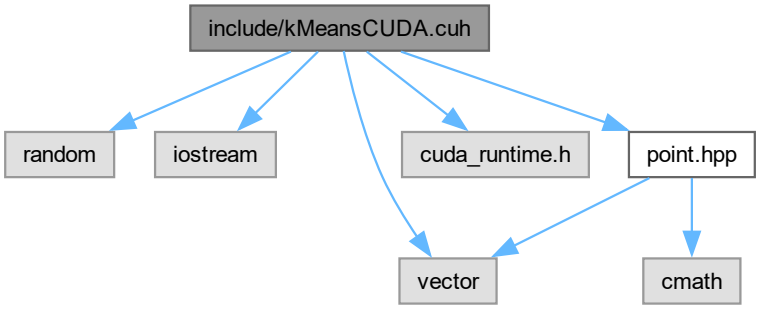
4.11 include/kMeansCUDA.cuh File Reference

Implementation of the K-means clustering algorithm using CUDA.

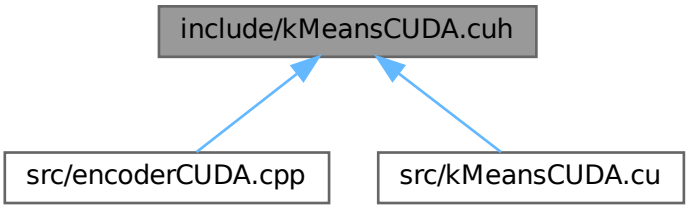
```
#include <random>
#include <iostream>
#include <vector>
#include <cuda_runtime.h>
```

```
#include <point.hpp>
```

Include dependency graph for kMeansCUDA.cuh:



This graph shows which files directly or indirectly include this file:



Classes

- class km::KMeansCUDA

  Represents the K-means clustering algorithm using CUDA.

Namespaces

- namespace km

  Main namespace for the project.

Functions

- __global__ void km::calculate_new_centroids (int *data, int *centroids, int *labels, int *counts, int n, int k, int dim)

  CUDA kernel to calculate the new centroids based on the assigned clusters.
- __global__ void km::average_centroids (int *centroids, int *counts, int k, int dim)

  CUDA kernel to average the calculated centroids.
- __global__ void km::assign_clusters (int *data, int *centroids, int *labels, int n, int k, int dim)

  CUDA kernel to assign each point to the nearest centroid.

4.11.1 Detailed Description

Implementation of the K-means clustering algorithm using CUDA.

4.12 kMeansCUDA.cuh

Go to the documentation of this file.

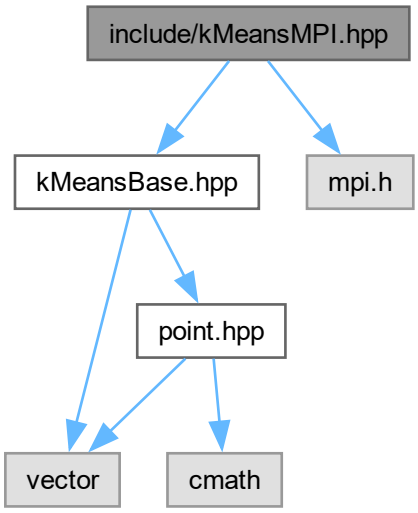
```
00001
00008 #ifndef KMEANS_CUDA_HPP
00009 #define KMEANS_CUDA_HPP
00011
00012 #include <random>
00013 #include <iostream>
00014 #include <vector>
00015 #include <cuda_runtime.h>
00016 #include <point.hpp>
00017
00018
00019 namespace km
00020 {
00033     __global__ void calculate_new_centroids(int *data, int *centroids, int *labels, int *counts, int n, int k, int dim);
00034
00044     __global__ void average_centroids(int *centroids, int *counts, int k, int dim);
00045
00057     __global__ void assign_clusters(int *data, int *centroids, int *labels, int n, int k, int dim);
00058
00059
00060
00073     class KMeansCUDA
00074     {
00075     public:
00082         KMeansCUDA(const int &k, const std::vector<Point> &points);
00083
00087         void run();
00088
00092         void printClusters() const;
00093
00097         void plotClusters();
00098
00103         auto getPoints() -> std::vector<Point>;
00104
00110         auto getCentroids() -> std::vector<Point>;
00111
00116         auto getIterations() -> int;
00117
00118     private:
00119         int k;
00120         std::vector<Point> points;
00121         std::vector<Point> centroids;
00122         int number_of_iterations;
00123     };
00124 } // namespace km
00125
00126
00128 #endif // KMEANS_CUDA_HPP
00129
```

4.13 include/kMeansMPI.hpp File Reference

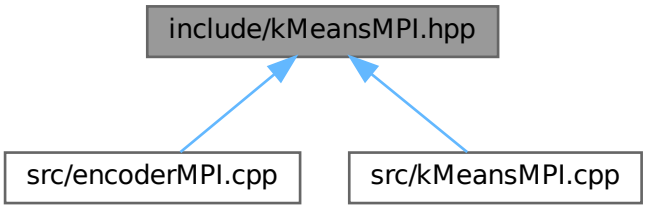
Implementation of the K-means clustering algorithm using MPI.

```
#include "kMeansBase.hpp"
#include <mpi.h>
```

Include dependency graph for kMeansMPI.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class km::KMeansMPI

  Represents the K-means clustering algorithm using MPI.

Namespaces

- namespace km

  Main namespace for the project.

4.13.1 Detailed Description

Implementation of the K-means clustering algorithm using MPI.

4.14 kMeansMPI.hpp

Go to the documentation of this file.

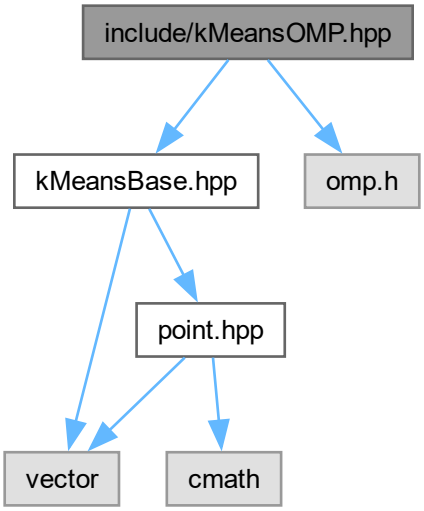
```
00001
00006 #ifndef KMEANS_MPI_HPP
00007 #define KMEANS_MPI_HPP
00008
00009 #include "kMeansBase.hpp"
00010 #include <mpi.h>
00011
00012 namespace km
00013 {
00026     class KMeansMPI : public KMeansBase
00027     {
00028     public:
00034         KMeansMPI(const int &k, const std::vector<Point> &points, std::vector<std::pair<int, Point» local_points);
00035
00040         KMeansMPI(const int &k, std::vector<std::pair<int, Point» local_points);
00041
00045         void run() override;
00046
00047     private:
00048         std::vector<std::pair<int, Point» local_points;
00049     };
00050 } // namespace k
00051
00052 #endif // KMEANS_MPI_HPP
```

4.15 include/kMeansOMP.hpp File Reference

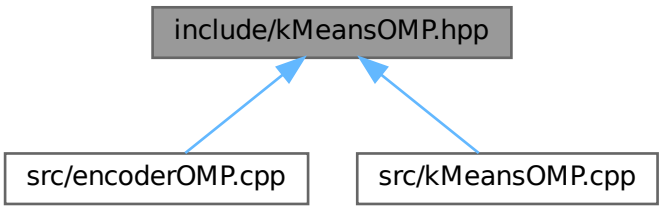
Implementation of the K-means clustering algorithm using OpenMP.

```
#include "kMeansBase.hpp"
#include <omp.h>
```

Include dependency graph for kMeansOMP.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class km::KMeansOMP

  Represents the K-means clustering algorithm using OpenMP.

Namespaces

- namespace km

  Main namespace for the project.

4.15.1 Detailed Description

Implementation of the K-means clustering algorithm using OpenMP.

4.16 kMeansOMP.hpp

Go to the documentation of this file.

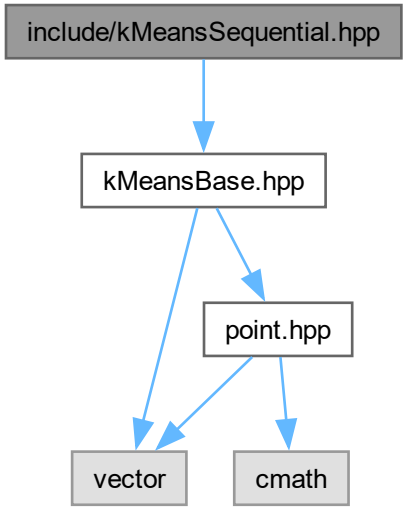
```
00001
00006 #ifndef KMEANS_OMP_HPP
00007 #define KMEANS_OMP_HPP
00008
00009 #include "kMeansBase.hpp"
00010 #include <omp.h>
00011
00012 namespace km
00013 {
00026     class KMeansOMP : public KMeansBase
00027     {
00028     public:
00034         KMeansOMP(const int &k, const std::vector<Point> &points);
00035
00039         void run() override;
00040     };
00041 } // namespace km
00042
00043 #endif // KMEANS_OMP_HPP
```

4.17 include/kMeansSequential.hpp File Reference

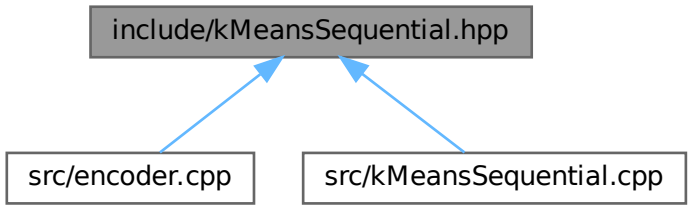
Implementation of the K-means clustering algorithm.

```
#include "kMeansBase.hpp"
```

Include dependency graph for kMeansSequential.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class km::KMeansSequential

  Represents the K-means clustering algorithm.

Namespaces

- namespace km

  Main namespace for the project.

4.17.1 Detailed Description

Implementation of the K-means clustering algorithm.

4.18 kMeansSequential.hpp

Go to the documentation of this file.

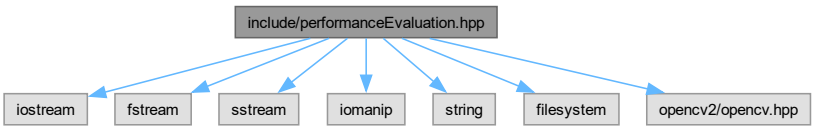
```
00001
00006 #ifndef KMEANS_SEQUENTIAL_HPP
00007 #define KMEANS_SEQUENTIAL_HPP
00008
00009 #include "kMeansBase.hpp"
00010
00011 namespace km
00012 {
00025     class KMeansSequential : public KMeansBase
00026     {
00027     public:
00033         KMeansSequential(const int &k, const std::vector<Point> &points);
00034
00038         void run() override;
00039     };
00040 }
00041
00042 #endif // KMEANS_SEQUENTIAL_HPP
```

4.19 include/performanceEvaluation.hpp File Reference

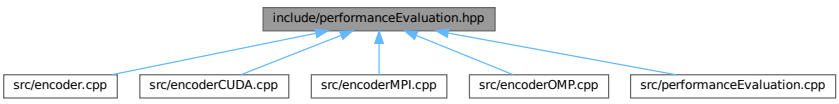
Performance evaluation class.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <string>
#include <filesystem>
#include <opencv2/opencv.hpp>
```

Include dependency graph for performanceEvaluation.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class km::Performance

  Represents the performance evaluation.

Namespaces

- namespace km

  Main namespace for the project.

4.19.1 Detailed Description

Performance evaluation class.

4.20 performanceEvaluation.hpp

Go to the documentation of this file.

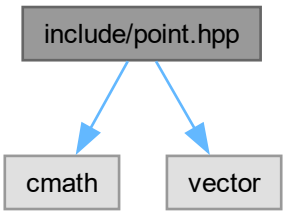
```
00001
00006 #ifndef PERFORMANCE_HPP
00007 #define PERFORMANCE_HPP
00008
00009 #include <iostream>
00010 #include <fstream>
00011 #include <sstream>
00012 #include <iomanip>
00013 #include <string>
00014 #include <filesystem>
00015 #include <opencv2/opencv.hpp>
00016
00017 namespace km
00018 {
00030     class Performance
00031     {
00032     public:
00036         Performance();
00037
00046         auto writeCSV(int different_colors_size, int k, int n_points, double elapsedKmeans, int number_of_iterations,
      int num_processes = 0) -> void;
00047
00053         static auto extractFileName(const std::string &outputPath) -> std::string;
00054
00061         auto fillPerformance(int choice, const std::string &img, const std::string &method) -> void;
00062
00063     private:
00068         auto createOrOpenCSV(const std::string &filename) -> void;
00069
00080         auto appendToCSV(const std::string &filename, int n_diff_colors, int k, int n_colors, const std::string
      &compType, double time, int num_processes, int number_of_iteratios) -> void;
00081
00082         std::string img;
00083         int choice{};
00084         std::string method;
00085     };
00086 }
00087
00088 #endif // PERFORMANCE_HPP
```

4.21 include/point.hpp File Reference

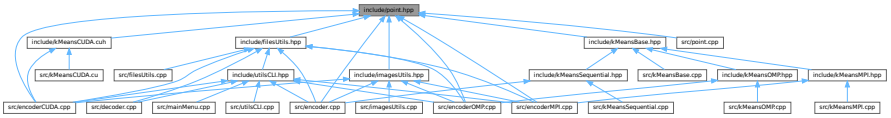
Point class representing a point in a feature space.

```
#include <cmath>
#include <vector>
```

Include dependency graph for point.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class km::Point

  Represents a point in a feature space.

Namespaces

- namespace km

  Main namespace for the project.

4.21.1 Detailed Description

Point class representing a point in a feature space.

4.22 point.hpp

Go to the documentation of this file.

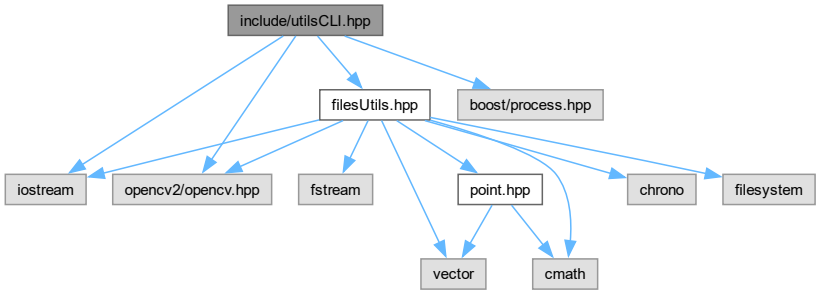
```
00001
00006 #ifndef POINT_HPP
00007 #define POINT_HPP
00008
00009 #include <cmath>
00010 #include <vector>
00011
00012 namespace km
00013 {
00023     class Point
00024     {
00025     public:
00026         int id{0};
00027         unsigned char r{0};
00028         unsigned char g{0};
00029         unsigned char b{0};
00030         int clusterId{-1};
00031
00036         Point();
00037
00043         Point(const int &id, const std::vector<int> &coordinates);
00044
00050         [[nodiscard]] auto distance(const Point &p) const -> double;
00051
00057         auto getFeature(int index) -> unsigned char &;
00058
00064         [[nodiscard]] auto getFeature_int(int index) const -> int;
00065
00071         auto setFeature(int index, int x) -> void;
00072     };
00073 } // namespace km
00074
00075 #endif // POINT_HPP
```

4.23 include/utilsCLI.hpp File Reference

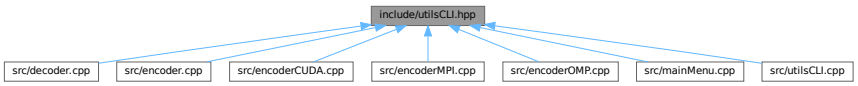
Utility functions for the command-line interface.

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <filesUtils.hpp>
#include <boost/process.hpp>
```

Include dependency graph for utilsCLI.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace km::utilsCLI

  Provides utility functions for the command-line interface.
- namespace km

  Main namespace for the project.

Functions

- void km::utilsCLI::mainMenuHeader ()

  Displays the main menu header.
- void km::utilsCLI::decoderHeader ()

  Displays the decoder header.
- void km::utilsCLI::workDone ()

  Displays the work done message.
- void km::utilsCLI::printCompressionInformations (int &originalWidth, int &originalHeight, int &width, int &height, int &k, size_t &different_colors_size)

  Prints the compression information.
- void km::utilsCLI::displayDecodingMenu (std::string &path, std::vector< std::filesystem::path > &imageNames, std::filesystem::path &decodeDir)

  Displays the decoding menu.

4.23.1 Detailed Description

Utility functions for the command-line interface.

4.24 utilsCLI.hpp

Go to the documentation of this file.

```
00001
00006 #ifndef UTILSCLI_HPP
00007 #define UTILSCLI_HPP
00008
00009 #include <iostream>
00010 #include <opencv2/opencv.hpp>
00011 #include <filesUtils.hpp>
00012 #include <boost/process.hpp>
00013
00025 namespace km
00026 {
00027     namespace utilsCLI
00028     {
00029
00033         void mainMenuHeader();
00034
00038         void decoderHeader();
00039
00043         void workDone();
00044
00055         void printCompressionInformations(int &originalWidth, int &originalHeight, int &width, int &height, int &k,
      size_t &different_colors_size);
00056
00063         void displayDecodingMenu(std::string &path, std::vector<std::filesystem::path> &imageNames,
      std::filesystem::path &decodeDir);
00064     };
00065 }
00066
00067 #endif // UTILSCLI_HPP
```

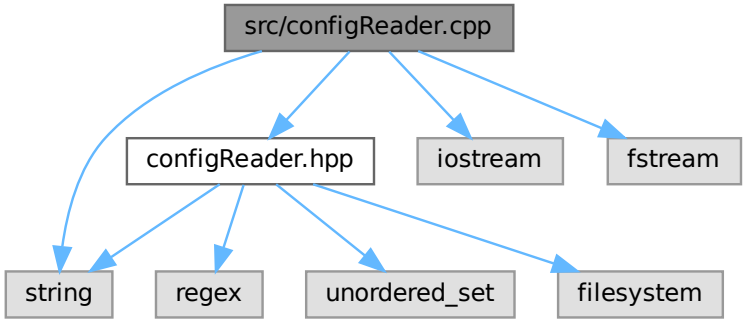
4.25 README.md File Reference

4.26 src/configReader.cpp File Reference

Implementation of the ConfigReader class for reading and parsing configuration files.

```
#include <configReader.hpp>
#include <iostream>
#include <fstream>
#include <string>
```

Include dependency graph for configReader.cpp:



4.26.1 Detailed Description

Implementation of the ConfigReader class for reading and parsing configuration files.

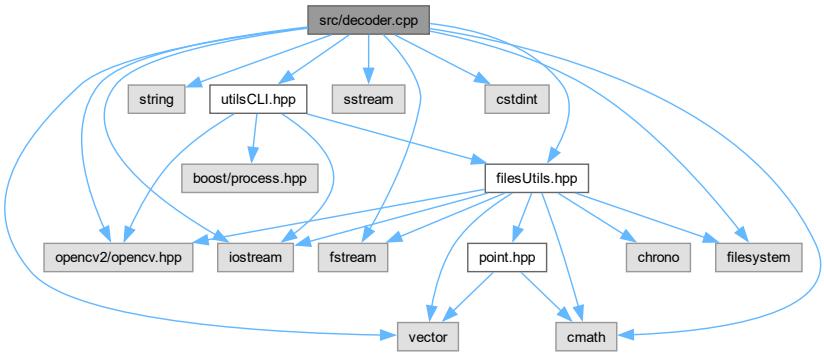
This source file contains the implementation of the ConfigReader class, which provides functionalities for reading, validating, and storing configuration settings from a .config file. The class is designed to handle configuration variables related to image processing, such as compression colors and resizing factors, and stores them for use in other parts of the application.

4.27 src/decoder.cpp File Reference

The main program for decoding .kc files generated by the Encoder.

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <cmath>
#include <cstdint>
#include <filesystem>
#include <filesUtils.hpp>
#include <utilsCLI.hpp>
```

Include dependency graph for decoder.cpp:



Functions

- auto main () -> int

  Main function for the Decoder program.

4.27.1 Detailed Description

The main program for decoding .kc files generated by the Encoder.

This program provides a command-line interface for decoding images that have been compressed by the Encoder. It allows users to load a compressed image, convert it to a viewable format, and optionally save it as a .jpg file. The program makes use of OpenCV for image processing and the custom filesUtilsand utilsCLI libraries for handling file operations and user interactions. The Decoder program reads a .kc compressed file, decodes it into an image matrix, converts the color space for display, and allows the user to save a decompressed copy as a .jpg image. The program provides a simple command-line interface for selecting files and configuring output options.

4.27.2 Function Documentation

4.27.2.1 main()

```
auto main ( ) -> int
```

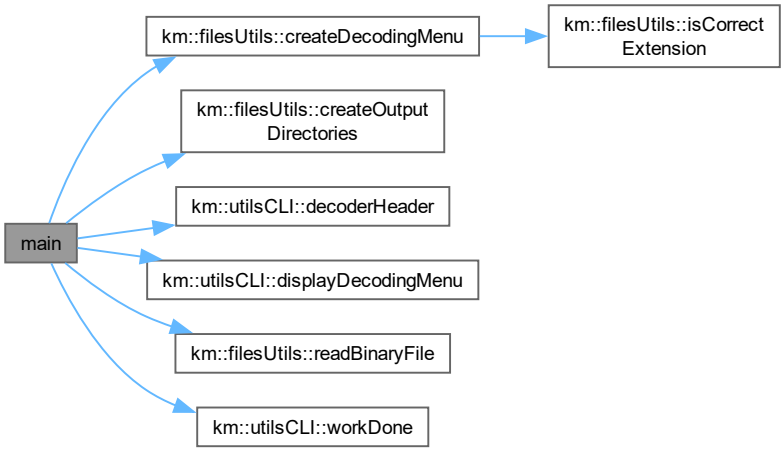
Main function for the Decoder program.

This function initializes the Decoder program, presents a menu for selecting the compressed image file, decodes the selected file, converts it for display, and provides the option to save the decoded image as a .jpg file.

Returns

Returns 0 on successful execution.

Here is the call graph for this function:



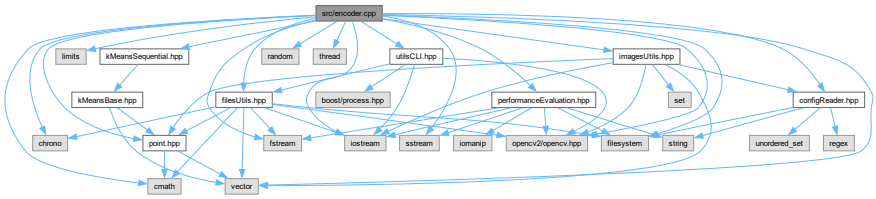
4.28 src/encoder.cpp File Reference

Main entry point for the image compression application.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
#include <filesystem>
#include <fstream>
#include <sstream>
#include <random>
#include <thread>
#include <chrono>
#include <point.hpp>
#include <kMeansSequential.hpp>
#include <utilsCLI.hpp>
#include <imagesUtils.hpp>
#include <filesUtils.hpp>
```

```
#include <opencv2/opencv.hpp>
#include <configReader.hpp>
#include <performanceEvaluation.hpp>
```

Include dependency graph for encoder.cpp:



Functions

- auto main (int argc, char ∗argv[ ]) -> int

  Main function for the image compression application.

4.28.1 Detailed Description

Main entry point for the image compression application.

This program compresses an image using the K-means clustering algorithm in sequential mode. It reads input parameters from the command line, processes the image, applies compression, and saves the compressed image to a binary file. The program also evaluates performance metrics and writes them to a CSV file. The application uses OpenCV for image processing and custom libraries for compression and file handling.

4.28.2 Function Documentation

4.28.2.1 main()

```
auto main (
              int argc,
              char * argv[ ] ) -> int
```

Main function for the image compression application.

This function initializes the program, processes input arguments, reads the input image, performs preprocessing, applies K-means clustering for image compression, and saves the compressed image to a binary file. It also evaluates and logs the performance metrics.

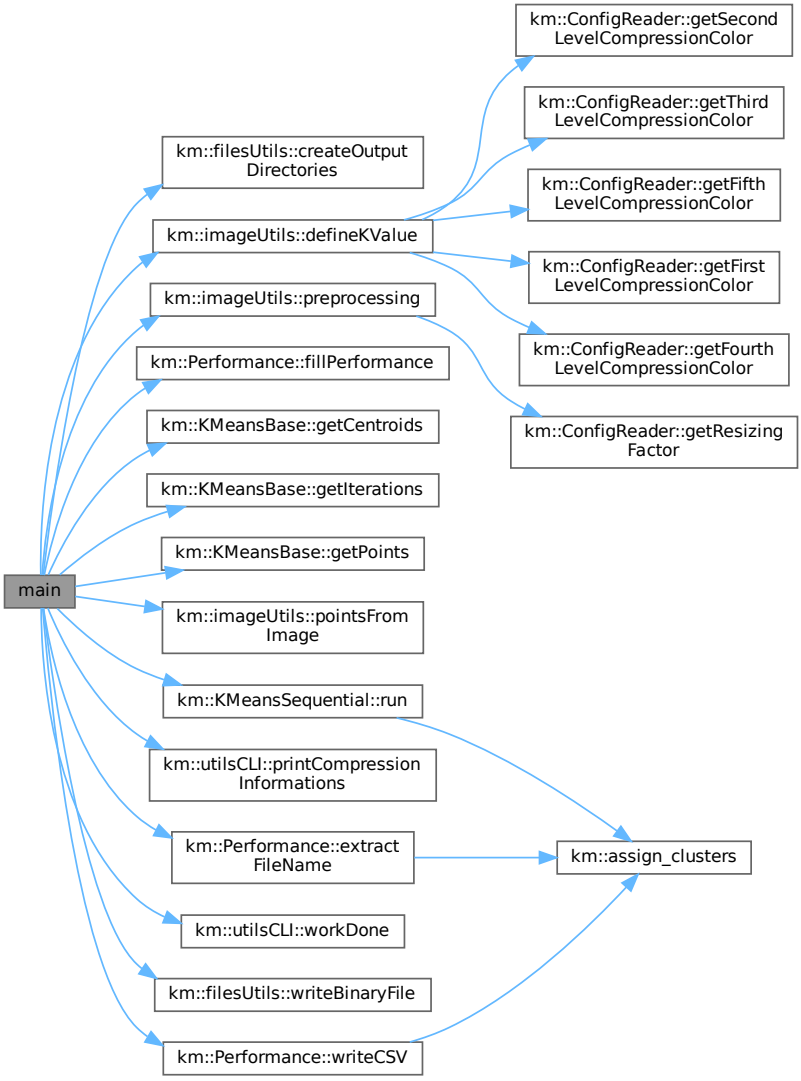
Parameters

argc	The number of command-line arguments.
argv	The array of command-line arguments.

Returns

Returns 0 on successful execution, or 1 if an error occurs.

Here is the call graph for this function:



4.29 src/encoderCUDA.cpp File Reference

Main entry point for the CUDA-based image compression application.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
#include <filesystem>
#include <fstream>
#include <sstream>
```

```
#include <random>
#include <thread>
#include <chrono>
#include <point.hpp>
#include <kMeansCUDA.cuh>
#include <utilsCLI.hpp>
#include <imagesUtils.hpp>
#include <filesUtils.hpp>
#include <opencv2/opencv.hpp>
#include <configReader.hpp>
#include <performanceEvaluation.hpp>
```

Include dependency graph for encoderCUDA.cpp:

Functions

- int main (int argc, char ∗argv[ ])

  Main function for the CUDA-based image compression application.

4.29.1 Detailed Description

Main entry point for the CUDA-based image compression application.

This program compresses an image using the K-means clustering algorithm, leveraging CUDA for GPU acceleration. It reads input parameters from the command line, processes the image, applies compression using the GPU, and saves the compressed image to a binary file. The program also evaluates performance metrics and logs them to a CSV file. The application uses OpenCV for image processing and custom libraries for compression, file handling, and performance evaluation.

4.29.2 Function Documentation

4.29.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function for the CUDA-based image compression application.

This function initializes the program, processes input arguments, reads the input image, performs preprocessing, applies K-means clustering using CUDA for image compression, and saves the compressed image to a binary file. It also evaluates and logs the performance metrics.

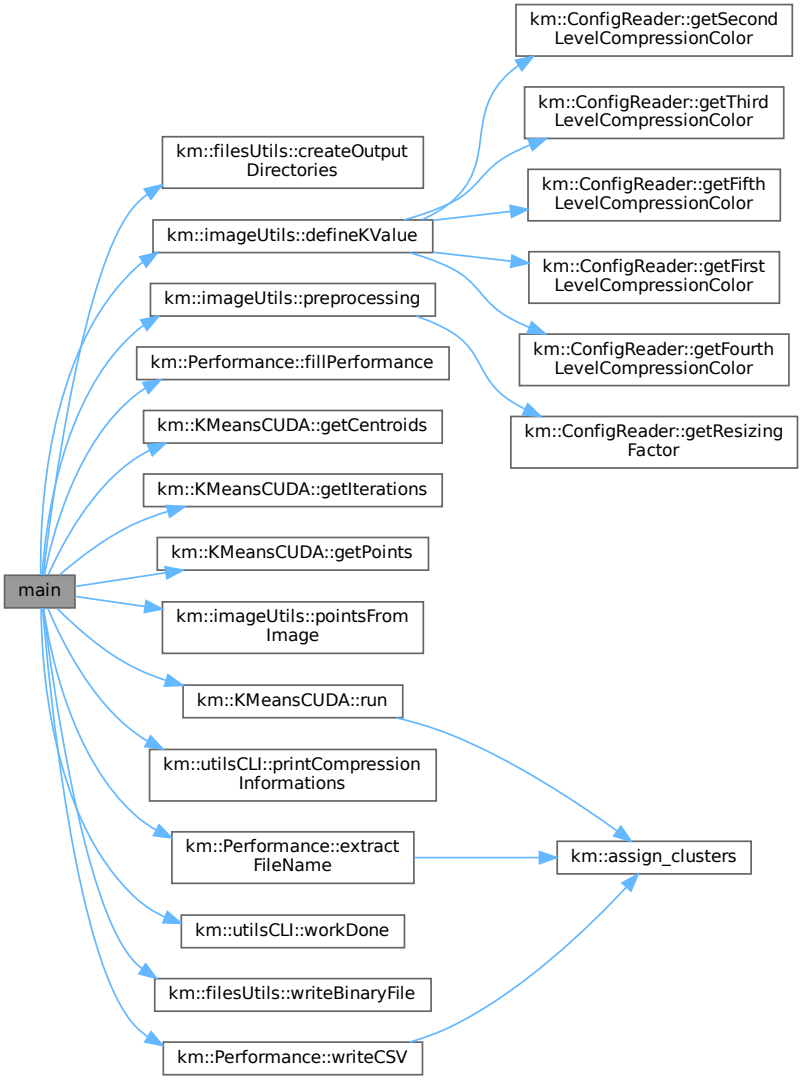
Parameters

argc	The number of command-line arguments.
argv	The array of command-line arguments.

Returns

Returns 0 on successful execution, or 1 if an error occurs.

Here is the call graph for this function:



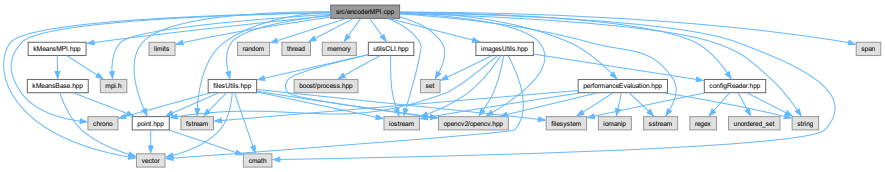
4.30 src/encoderMPI.cpp File Reference

Main entry point for the MPI-based image compression application.

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <limits>
#include <fstream>
```

```
#include <sstream>
#include <random>
#include <thread>
#include <memory>
#include <set>
#include <chrono>
#include <point.hpp>
#include <kMeansMPI.hpp>
#include <configReader.hpp>
#include <utilsCLI.hpp>
#include <imagesUtils.hpp>
#include <filesUtils.hpp>
#include <mpi.h>
#include <performanceEvaluation.hpp>
#include <span>
```

Include dependency graph for encoderMPI.cpp:



Functions

- auto main (int argc, char ∗argv[ ]) -> int

  Main function for the MPI-based image compression application.

4.30.1 Detailed Description

Main entry point for the MPI-based image compression application.

This program compresses an image using the K-means clustering algorithm, leveraging CUDA for GPU acceleration. It reads input parameters from the command line, processes the image, applies compression using the GPU, and saves the compressed image to a binary file. The program also evaluates performance metrics and logs them to a CSV file. The application uses OpenCV for image processing and custom libraries for compression, file handling, and performance evaluation.

4.30.2 Function Documentation

4.30.2.1 main()

```
auto main (
              int argc,
              char * argv[] ) -> int
```

Main function for the MPI-based image compression application.

This function initializes the program, processes input arguments, reads the input image, performs preprocessing, applies K-means clustering using CUDA for image compression, and saves the compressed image to a binary file. It also evaluates and logs the performance metrics.

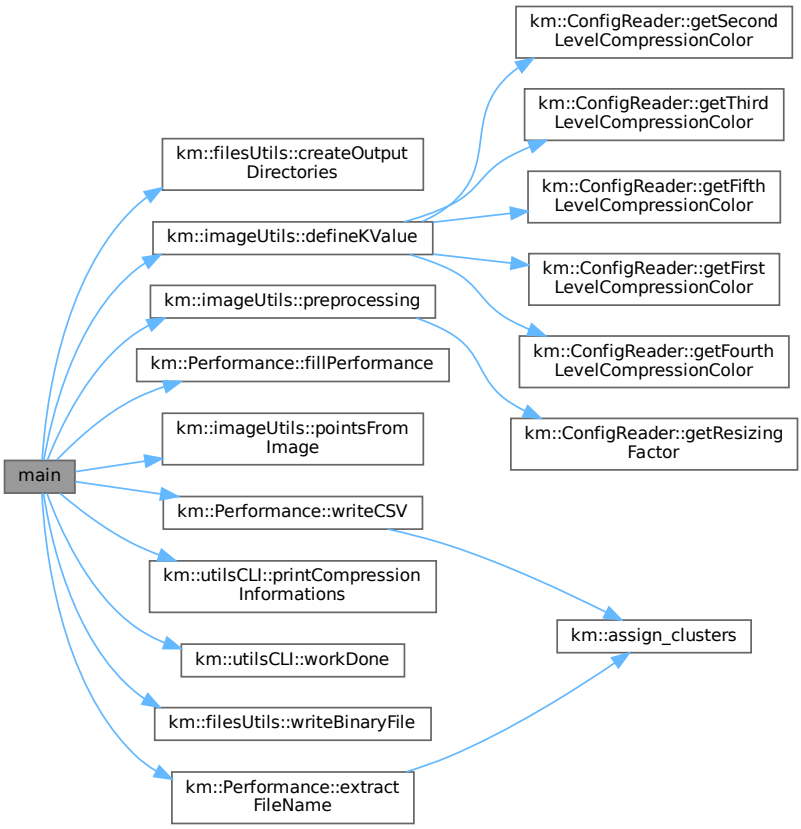
Parameters

argc	The number of command-line arguments.
argv	The array of command-line arguments.

Returns

Returns 0 on successful execution, or 1 if an error occurs.

Here is the call graph for this function:



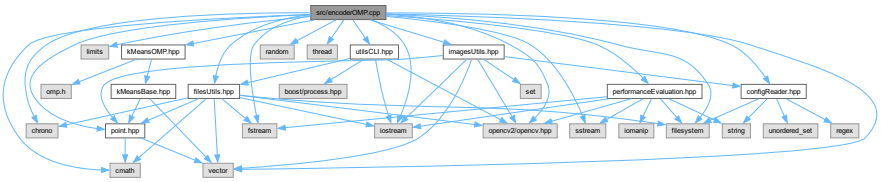
4.31 src/encoderOMP.cpp File Reference

Main entry point for the OpenMP-based image compression application.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
#include <filesystem>
#include <fstream>
#include <sstream>
#include <random>
```

```
#include <thread>
#include <chrono>
#include <point.hpp>
#include <kMeansOMP.hpp>
#include <utilsCLI.hpp>
#include <imagesUtils.hpp>
#include <filesUtils.hpp>
#include <opencv2/opencv.hpp>
#include <configReader.hpp>
#include <performanceEvaluation.hpp>
```

Include dependency graph for encoderOMP.cpp:



Functions

- auto main (int argc, char ∗argv[ ]) -> int

  Main function for the OpenMP-based image compression application.

4.31.1 Detailed Description

Main entry point for the OpenMP-based image compression application.

This program compresses an image using the K-means clustering algorithm with OpenMP for parallel processing across multiple threads. It reads input parameters from the command line, processes the image, applies compression, and saves the compressed image to a binary file. The program also evaluates performance metrics and writes them to a CSV file. The application uses OpenCV for image processing and custom libraries for compression, file handling, and performance evaluation.

4.31.2 Function Documentation

4.31.2.1 main()

```
auto main (
              int argc,
              char * argv[] ) -> int
```

Main function for the OpenMP-based image compression application.

This function initializes the program, processes input arguments, reads the input image, performs preprocessing, applies K-means clustering using OpenMP for image compression, and saves the compressed image to a binary file. It also evaluates and logs the performance metrics.

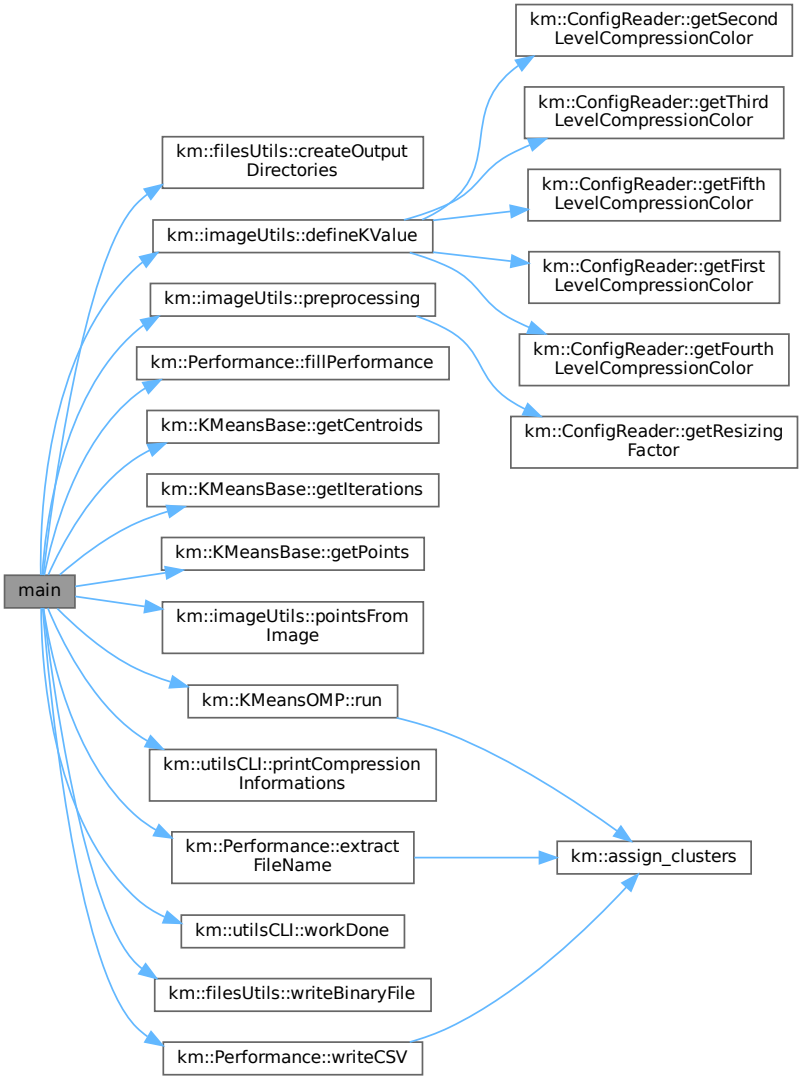
Parameters

argc	The number of command-line arguments.
argv	The array of command-line arguments.

Returns

Returns 0 on successful execution, or 1 if an error occurs.

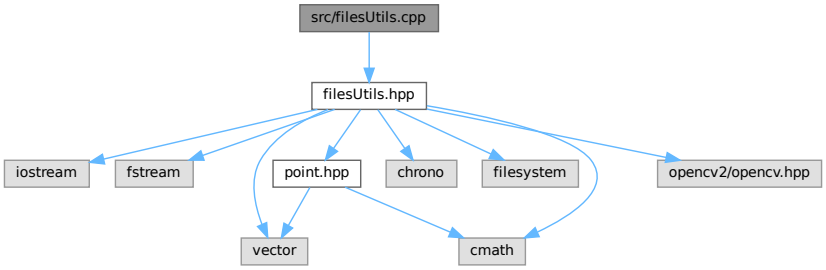
Here is the call graph for this function:



4.32 src/filesUtils.cpp File Reference

Utility functions for file operations.

`#include <filesUtils.hpp>`
Include dependency graph for filesUtils.cpp:



4.32.1 Detailed Description

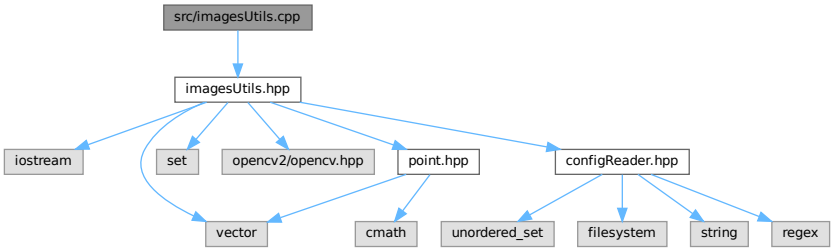
Utility functions for file operations.

This file contains implementations of various utility functions used for file handling, including creating output directories, writing and reading binary files, checking file extensions, and managing decoding menus for the image compression application.

4.33 src/imagesUtils.cpp File Reference

Utility functions for image processing.

`#include <imagesUtils.hpp>`
Include dependency graph for imagesUtils.cpp:



4.33.1 Detailed Description

Utility functions for image processing.

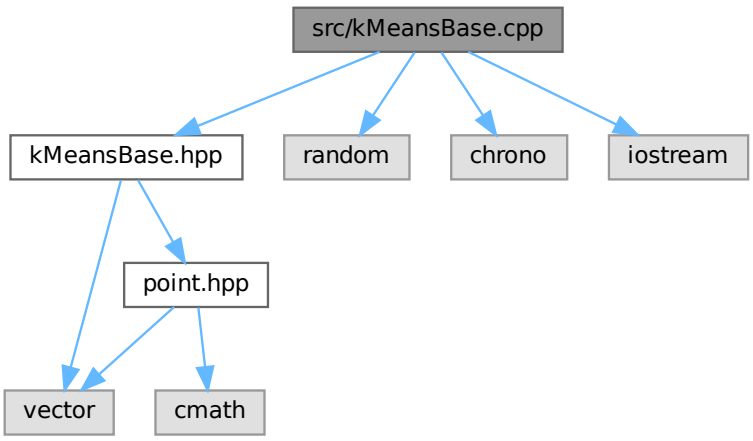
This file contains implementations of utility functions used for preprocessing images, defining compression levels, and extracting data points from images for the image compression application.

4.34 src/kMeansBase.cpp File Reference

Base class implementation for the K-means clustering algorithm.

```
#include "kMeansBase.hpp"
#include <random>
#include <chrono>
#include <iostream>
```

Include dependency graph for kMeansBase.cpp:



4.34.1 Detailed Description

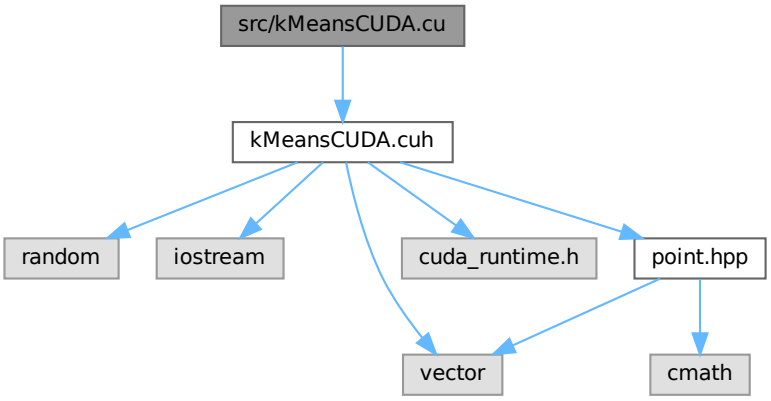
Base class implementation for the K-means clustering algorithm.

This file contains the implementation of the `KMeansBase` class, which serves as the base class for different versions of the K-means clustering algorithm. It provides fundamental functionalities such as initialization of centroids, getters for points, centroids, and the number of iterations.

4.35 src/kMeansCUDA.cu File Reference

CUDA implementation of the K-means clustering algorithm.

```
#include <kMeansCUDA.cuh>
```
Include dependency graph for kMeansCUDA.cu:



4.35.1 Detailed Description

CUDA implementation of the K-means clustering algorithm.

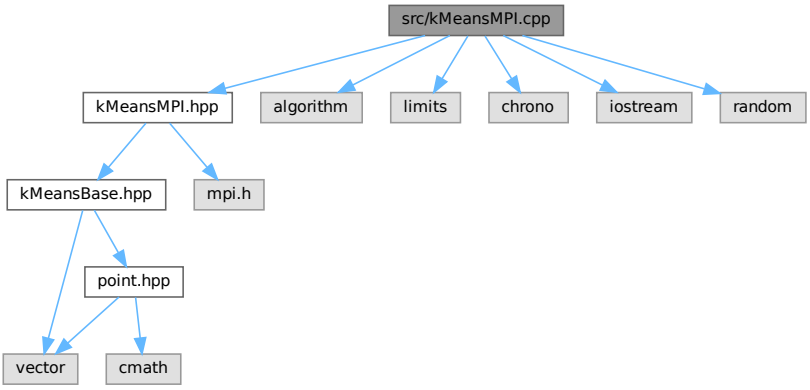
This file contains the implementation of the `KMeansCUDA` class, which leverages CUDA to perform the K-means clustering algorithm on the GPU. The class includes functions for initializing centroids, assigning clusters, calculating new centroids, and averaging centroids, all using CUDA kernels for parallel processing. This implementation allows for efficient processing of large datasets by utilizing GPU acceleration.

4.36 src/kMeansMPI.cpp File Reference

MPI implementation of the K-means clustering algorithm.

```
#include "kMeansMPI.hpp"
#include <algorithm>
#include <limits>
#include <chrono>
#include <iostream>
#include <random>
```

Include dependency graph for kMeansMPI.cpp:



4.36.1 Detailed Description

MPI implementation of the K-means clustering algorithm.

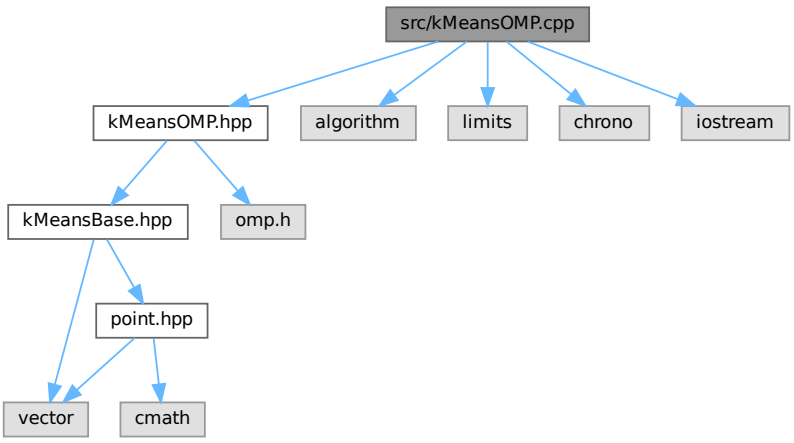
This file contains the implementation of the `KMeansMPI` class, which uses MPI (Message Passing Interface) for parallel processing to perform the K-means clustering algorithm across multiple processors. The class handles tasks such as initializing centroids, distributing data, computing new centroids, and synchronizing results among different MPI processes. This implementation is designed for efficient distributed computation in a high-performance computing environment.

4.37 src/kMeansOMP.cpp File Reference

OpenMP implementation of the K-means clustering algorithm.

```
#include "kMeansOMP.hpp"
#include <algorithm>
#include <limits>
#include <chrono>
#include <iostream>
```

Include dependency graph for kMeansOMP.cpp:



4.37.1 Detailed Description

OpenMP implementation of the K-means clustering algorithm.

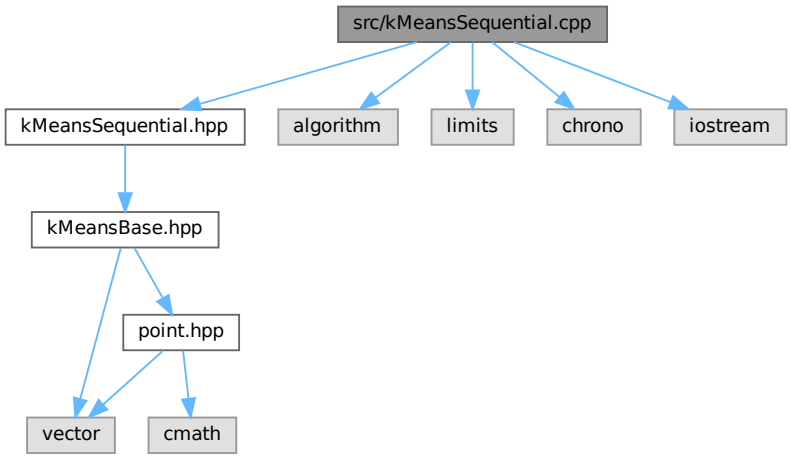
This file contains the implementation of the `KMeansOMP` class, which utilizes OpenMP for parallel processing to perform the K-means clustering algorithm using multiple threads. The class includes methods for initializing centroids, updating cluster assignments, recalculating centroids, and synchronizing results across threads. This implementation leverages multi-threading to enhance performance in shared-memory environments.

4.38 src/kMeansSequential.cpp File Reference

Sequential implementation of the K-means clustering algorithm.

```
#include "kMeansSequential.hpp"
#include <algorithm>
#include <limits>
#include <chrono>
#include <iostream>
```

Include dependency graph for kMeansSequential.cpp:



4.38.1 Detailed Description

Sequential implementation of the K-means clustering algorithm.

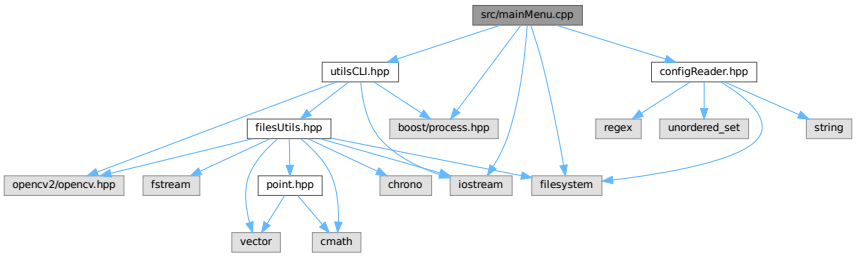
This file contains the implementation of the `KMeansSequential` class, which performs the K-means clustering algorithm in a sequential manner without parallelization. The class handles tasks such as initializing centroids, assigning points to clusters, recalculating centroids, and determining convergence. This implementation is intended for environments where parallel processing is not available or needed.

4.39 src/mainMenu.cpp File Reference

Main entry point for the Image Compressor application.

```
#include <iostream>
#include <utilsCLI.hpp>
#include <filesystem>
#include <configReader.hpp>
#include <boost/process.hpp>
```

Include dependency graph for mainMenu.cpp:



Functions

- auto main () -> int

  Main function that runs the Image Compressor application.

4.39.1 Detailed Description

Main entry point for the Image Compressor application.

This file contains the main function that initializes the Image Compressor application. It reads configuration settings, displays a command-line interface for users to choose between image compression and decompression, and handles user input to run the appropriate compression or decompression processes.

4.39.2 Function Documentation

4.39.2.1 main()

```
auto main ( ) -> int
```

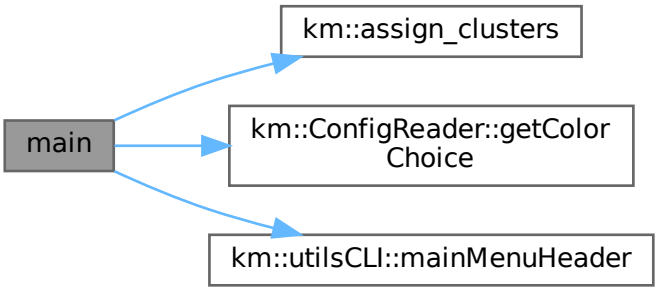
Main function that runs the Image Compressor application.

This function initializes necessary components, reads configuration settings, and provides a command-line interface for users to choose between image compression and decompression options. It handles user inputs to determine the desired operation and calls the appropriate functions or external processes to execute the selected option.

Returns

Returns 0 on successful execution.

Here is the call graph for this function:

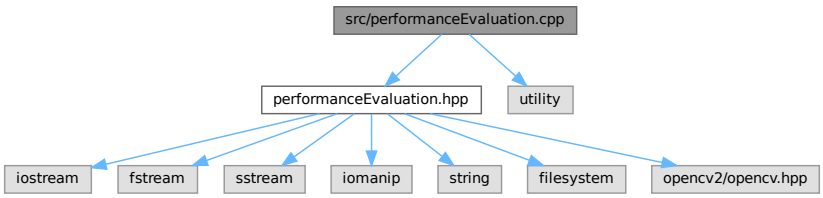


4.40 src/performanceEvaluation.cpp File Reference

Utility functions for evaluating and recording the performance of image compression algorithms.

```
#include <performanceEvaluation.hpp>
#include <utility>
```

Include dependency graph for performanceEvaluation.cpp:



4.40.1 Detailed Description

Utility functions for evaluating and recording the performance of image compression algorithms.

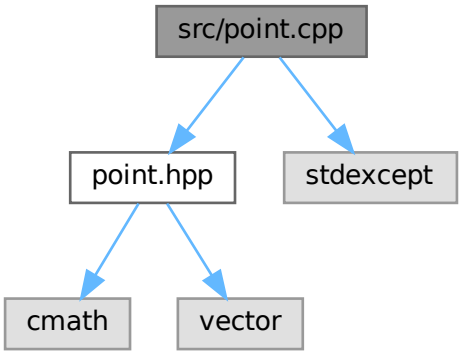
This file contains the implementation of the `Performance` class, which provides functionalities to record performance metrics such as time taken for compression, number of iterations, and other relevant statistics. The class includes methods to fill performance data, write results to a CSV file, and manage the creation and appending of data to the CSV file for easy analysis.

4.41 src/point.cpp File Reference

Implementation of the Point class for image compression algorithms.

```
#include <point.hpp>
#include <stdexcept>
```

Include dependency graph for point.cpp:



4.41.1 Detailed Description

Implementation of the Point class for image compression algorithms.

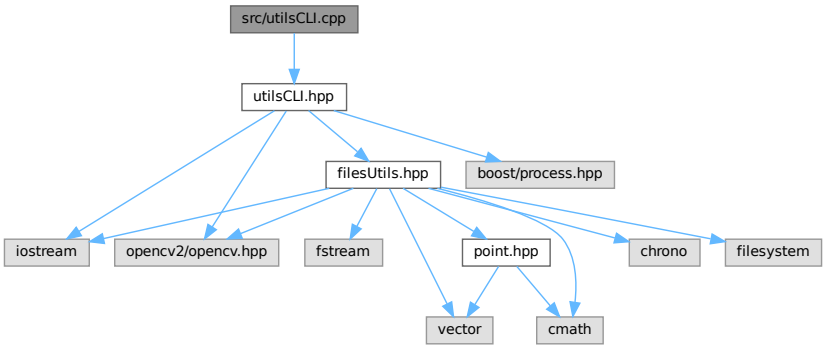
This file contains the implementation of the `Point` class, which represents a pixel or data point in a color space for use in image compression algorithms. The class provides methods to access and modify color features, calculate distances between points, and manage cluster assignments. It is a core component for K-means clustering and similar algorithms.

4.42 src/utilsCLI.cpp File Reference

Utility functions for command-line interface operations.

```
#include <utilsCLI.hpp>
```

Include dependency graph for utilsCLI.cpp:



4.42.1 Detailed Description

Utility functions for command-line interface operations.

This file contains implementations of utility functions that facilitate command-line interactions for the Image Compressor application. It includes functions to display headers, manage user inputs, and provide feedback during compression and decompression processes.

Index