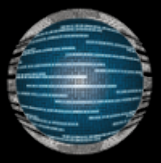


**Τμήμα Μηχανικών Η/Υ και Πληροφορικής**
*Πανεπιστήμιο Πατρών*
3 Δεκεμβρίου 2020

# Παράλληλος Προγραμματισμός
## 2ο *Εργαστήριο*

*Ανδρέας Καρατζάς*

# Περιεχόμενα


3 Κεφαλαιο 1

Ο Αλγόριθμος K-Means με OpenMP

1.1 Εισαγωγή *3*

1.2 GitHub *3*

1.3 Παραλληλοποίηση *4*

1.4 SIMD *6*

1.5 Ανάλυση Παραδοτέων *6*

1.5.1 Εισαγωγή *6*

1.5.2 Optimization *7*

1.5.3 Classes *7*

1.5.4 Η έκδοση Debug και η έκδοση Release *7*

1.5.5 Η μεταβλητή `NUM_THR` *7*

1.5.6 Η μεταβλητή `CHUNK_SIZE` *8*

1.6 Intel Profiler *9*

1.7 Scaling *10*

13 Κεφαλαιο 2

Ο κώδικας σε `C++ 17`

# 1

# Ο Αλγόριθμος K-Means με OpenMP

## 1.1 Εισαγωγή

Για τη 2η εργαστηριακή άσκηση έγινε παραλληλοποίηση του K-Means που υλοποιήθηκε σειριακά στην προηγούμενη εργασία. Η παραλληλοποίηση έγινε με OpenMP. Τα στοιχεία του συστήματος στο οποίο έγιναν οι μετρήσεις παρουσιάζονται στον Πίνακα 1.

*Πίνακας 1: Στοιχεία Συστήματος*

| | |
|---|---|
| Λειτουργικό Σύστημα | Windows 10 x64 |
| Επεξεργαστής | Intel Core i7-9750H CPU @ 2.60 GHz |
| RAM frequency | 2667 MHz |
| RAM size | 16 GB |
| IDE | Visual Studio 2019 |
| Compiler | Intel 19.1 |
| Programming Language | C++ 17 |

## 1.2 GitHub

Στα πλαίσια της εργασίας δημιουργήθηκε repository στο GitHub. Αυτό διευκολύνει τον έλεγχο του κώδικα αλλά προσφέρει και δυνατότητα επέκτασης της εργασίας στο μέλλον. Για συστήματα Linux, όπως το μηχάνημα του εργαστηρίου, ο προγραμματιστής προτείνεται να:

- Ανοίξει ένα καινούργιο παράθυρο `Terminal`
- Εκτελεί την εντολή `git clone https://github.com/andreasceid/k-means-parallel.git`
- Εκτελεί την εντολή `cd k-means-parallel`
- Εκτελεί την εντολή `chmod u+x disp_system_thread_count-ubu20.sh`
- Εκτελεί την εντολή `./disp_system_thread_count-ubu20.sh`
- Σημειώνει τον αριθμό που του επιστρέφει η εντολή

- Εκτελεί την εντολή `cd k-means-parallel`
- Εκτελεί την εντολή `vim Common.h`
- Αλλάζει την τιμή που έχει η μεταβλητή `NUM_THR` στη γραμμή 27 και την θέτει ίση με τον αριθμό που σημείωσε σε προηγούμενο βήμα (π.χ 8 για το μηχάνημα του εργαστηρίου)
- Αποθηκεύει τις αλλαγές
- Εκτελεί την εντολή `make`
- Εκτελεί την εντολή `./kmeans`

Ο μέσος αριθμός iterations είναι 24, και ο μέσος χρόνος εκτέλεσης είναι 2 λεπτά.

## 1.3 Παραλληλοποίηση

Για να παραλληλοποιηθεί ο αλγόριθμος K-Means ακολουθήθηκε το παράλληλο μοντέλο που φαίνεται στον Αλγόριθμο 1.1. Η αρχικοποίηση του συνόλου δεδομένων (dataset) σχεδιάστηκε να γίνει ταυτόχρονα με την αρχικοποίηση των κέντρων. Για να γίνει αυτό, έγινε αλλαγή στη σειριακή έκδοση του K-Means. Αντί τα κέντρα να αρχικοποιούνται με κάποιο τυχαίο διάνυσμα μέσα στο σύνολο δεδομένων, θα αρχικοποιούνται κι αυτά τυχαία. Έτσι, επιλύεται η εξάρτηση που υπήρχε μεταξύ της αρχικοποίησης του συνόλου δεδομένων και των κέντρων του K-Means. Έπειτα, η αντιστοίχηση των $N$ διανυσμάτων σε συστάδες `Classes` έγινε παράλληλα. Συγκεκριμένα, οι υπολογισμοί για την εύρεση της συστάδας που ανήκει ένα διάνυσμα $i$ γίνεται παράλληλα με τους υπολογισμούς για την εύρεση συστάδας που ανήκει ένα διάνυσμα $j$, με $i \neq j, \forall\, i, j$. Δομικά, δεν υπάρχουν άλλα σημεία τα οποία να μπορούν να παραλληλοποιηθούν. Για παράδειγμα, ο υπολογισμός του λάθους που υπάρχει ανάμεσα στα εκτιμώμενα clusters και στα πραγματικά, θα πρέπει να γίνει αφότου αντιστοιχιστούν σε κλάσεις τα διανύσματα του συνόλου δεδομένων.

**Αλγόριθμος 1.1** K-Means - Η παράλληλη έκδοση

```
1  input: Vec[N][Nv], Center[Nc][Nv], Classes[N], double threshold
2  output: Center
3  begin
4          do
5                  fork
6                          initialize Vec[N][Nv]
7                          initialize Center[N][Nv]
8                  join
9                  foreach Vec[i]:
10                         fork
11                                 Compute Class[i]
12                 join
13                 Estimate error
14         while error > threshold
15 end
```

Μια επιπλέον σκέψη ήταν να ενοποιηθούν κάτω από μία συνάρτηση οι υπολογισμοί της αντιστοίχισης διανυσμάτων σε συστάδες, και οι υπολογισμοί των καινούργιων συστάδων. Ωστόσο, αυτό κάνει αρκετά δυσνόητο το πρόγραμμα, όπως φαίνεται στον Κώδικα 1.2.1.

Κώδικας 1.3.1

```
void optimize(void)
{
  int i, j, k, cluster;
  long double distance;

#pragma omp parallel num_threads(2)
  {
#pragma omp master
    {
      memset(Classes, 0, Nc * sizeof(Classes[0]));
    }
#pragma omp single
    {
      memset(new_Center, 0.0, Nc * Nv * sizeof(new_Center[0][0]));
    }
  }

  for (i = 0; i < N; i += 1)
  {
    distance = 0.0;
    cluster = -1;

    long double min = DBL_MAX;

    for (j = 0; j < Nc; j += 1)
    {
#pragma omp parallel for num_threads(NUM_THR) reduction(+ \
                                                : distance)
      for (k = 0; k < Nv; k += 1)
      {
        distance += (Vec[i][k] - old_Center[j][k]) *
                    (Vec[i][k] - old_Center[j][k]);
      }
      if (distance < min)
      {
        min = distance;
        cluster = j;
      }
```

```
39      }
40
41      Classes[cluster] += 1;
42  #pragma omp parallel for num_threads(NUM_THR) reduction(+ \
43                                                  : new_Center[cluster])
44      for (k = 0; k < Nv; k += 1)
45      {
46        new_Center[cluster][k] += Vec[i][k];
47      }
48    }
49  }
```

Οι υπόλοιπες παραλληλοποιήσεις δεν έγιναν σε δομικό επίπεδο. Συγκεκριμένα, παραλληλοποίηση έγινε σε `for` επαναλήψεις οι οποίες με τη χρήση του OpenMP κατανέμονται στα threads του συστήματος, επιτυγχάνοντας γρηγορότερους χρόνους ολοκλήρωσης της επανάληψης.

## 1.4 SIMD

Στα πλαίσια της εργασίας, ζητήθηκε να γίνει η εκμετάλλευση του επιταχυντή SIMD που διαθέτει το OpenMP. Ωστόσο, λόγω της δομής των δεδομένων αρκετές ήταν οι φορές που αυτό δεν ήταν εφικτό. Για παράδειγμα, επειδή το `Vec` που αποτελεί τη δομή του συνόλου δεδομένων του K-Means είναι τύπου `std::vector`, δε μπόρεσε να παραλληλοποιηθεί η επανάληψη αρχικοποίησής του. Η ρουτίνα υπάρχει στο αρχείο που αποστέλλεται, και το μήνυμα του μεταγλωττιστή είναι ότι δε μπορεί να μετασχηματίσει τέτοιες επαναλήψεις. Ωστόσο, η δυνατότητα `simd` αξιοποιήθηκε κατά την εύρεση της ευκλείδειας απόστασης μεταξύ 2 διανυσμάτων. Αυτό κιόλας ήταν ιδιαίτερα θετικό, καθώς η συγκεκριμένη συνάρτηση, σύμφωνα με το Profiler του GCC (gprof), η συνάρτηση αυτή κρατούσε το μεγαλύτερο ποσοστό χρόνου εκτέλεσης στο πρόγραμμα[1].

## 1.5 Ανάλυση Παραδοτέων

### 1.5.1 Εισαγωγή

Η δομή της εργασίας είναι ακριβώς ίδια με αυτήν της 1ης εργαστηριακής άσκησης. Επομένως, σε αυτό το κεφάλαιο θα αναφερθούν οι αλλαγές που έπρεπε να γίνουν στη σειριακή έκδοση έτσι ώστε να γίνει η παραλληλοποίηση.

[1] Ο Profiler είχε δείξει ότι η συνάρτηση `euclidean_diference` κατανάλωνε περίπου το 95% του χρόνου εκτέλεσης του προγράμματος.

### 1.5.2 Optimization

Η πιο βασική αλλαγή ήταν η δήλωση της μεταβλητής `new_Center` ως `std::vector<std::array<float, Nv>> new_Center(Nc)`, αντί για `std::vector<std::array<float, Nv>> new_Center`. Αυτό έγινε για να παραλληλοποιηθεί η συνάρτηση `optimize_center` της κεφαλίδας *Optimize*. Μετά από αυτήν την επεξεργασία, επιλύθηκε το πρόβλημα του *race condition* που υπήρχε λόγω της `emplace_back`, καθώς πλέον η ανάθεση τιμών γίνεται με δεικτοδότηση (indexing) χρησιμοποιώντας τη δυνατότητα `at()` που έχουν τα διανύσματα στη `C++ 17`. Επομένως, πλέον δε χρειάζεται `reserve` για τη συγκεκριμένη μεταβλητή.

### 1.5.3 Classes

Οι υπολογισμοί για τη μεταβλητή `Classes` ήταν ένα ακόμα δύσκολο κομμάτι προς παραλληλοποίηση. Συγκεκριμένα, η `Classes` είναι μια μεταβλητή τύπου `std::array<std::vector<int>, Nc>`, το οποίο κάνει αρκετά πιο ελέγξιμο το πρόγραμμα αλλά προκαλεί προβλήματα στο κομμάτι της παραλληλοποίησης. Για την επίτευξη της παραλληλοποίησης σε αυτήν τη συνάρτηση, έγιναν `private` κάποιες μεταβλητές. Τέλος, το σημείο ένθεσης στο διάνυσμα έγινε `critical`[2].

### 1.5.4 Η έκδοση Debug και η έκδοση Release

Για να διευκολυνθεί η διαδικασία του testing, δημιουργήθηκαν 2 ξεχωριστά driver files:

- `k-means-debug.cpp`: Αυτό το αρχείο κώδικα περιέχει το driver (main) σε περίπτωση που ο προγραμματιστής επιθυμεί να κάνει δοκιμές και να μελετήσει τη συμπεριφορά του προγράμματος
- `k-means-relsease.cpp`: Αυτό το αρχείο κώδικα περιέχει το driver (main) σε περίπτωση που ο προγραμματιστής επιθυμεί να κάνει benchmarking του προγράμματος και να μελετήσει τις επιδόσεις της παράλληλης έκδοσης

### 1.5.5 Η μεταβλητή `NUM_THR`

Για τη ρύθμιση του αριθμού των threads έχει οριστεί η μεταβλητή `NUM_THR`. Για το προσδιορισμό της μεταβλητής, ο προγραμματιστής προτείνεται να επιλέξει έναν ακέραιο αριθμό μεγαλύτερο του 1 και μικρότερο από τον πλήθος των logical cores που διαθέτει στο σύστημά του. Για τη διευκόλυνση του προγραμματιστή, έχουν γραφτεί 2 scripts:

[2] Έγινε και προσπάθεια να οριστεί ως `atomic`, το οποίο εκμεταλλεύεται καλύτερα τις επιμέρους υλικές ιδιότητες, αλλά ο μεταγλωττιστής δεν το αναγνώριζε ως atomic write ή ως atomic update operation.

- `disp_system_thread_count-win64`: Αυτό το Powershell script είναι φτιαγμένο για λειτουργικά συστήματα Windows 10 x64 bit. Η έξοδος είναι ένας αριθμός που αντιπροσωπεύει το συνολικό αριθμό logical cores
- `disp_system_thread_count-ubu20`: Αυτό το Bash script είναι φτιαγμένο για λειτουργικά συστήματα Ubuntu 20.04. Η έξοδος είναι ένας αριθμός που αντιπροσωπεύει το συνολικό αριθμό logical cores

Τα παραπάνω scripts υπάρχουν στο GitHub repository της εργασίας.

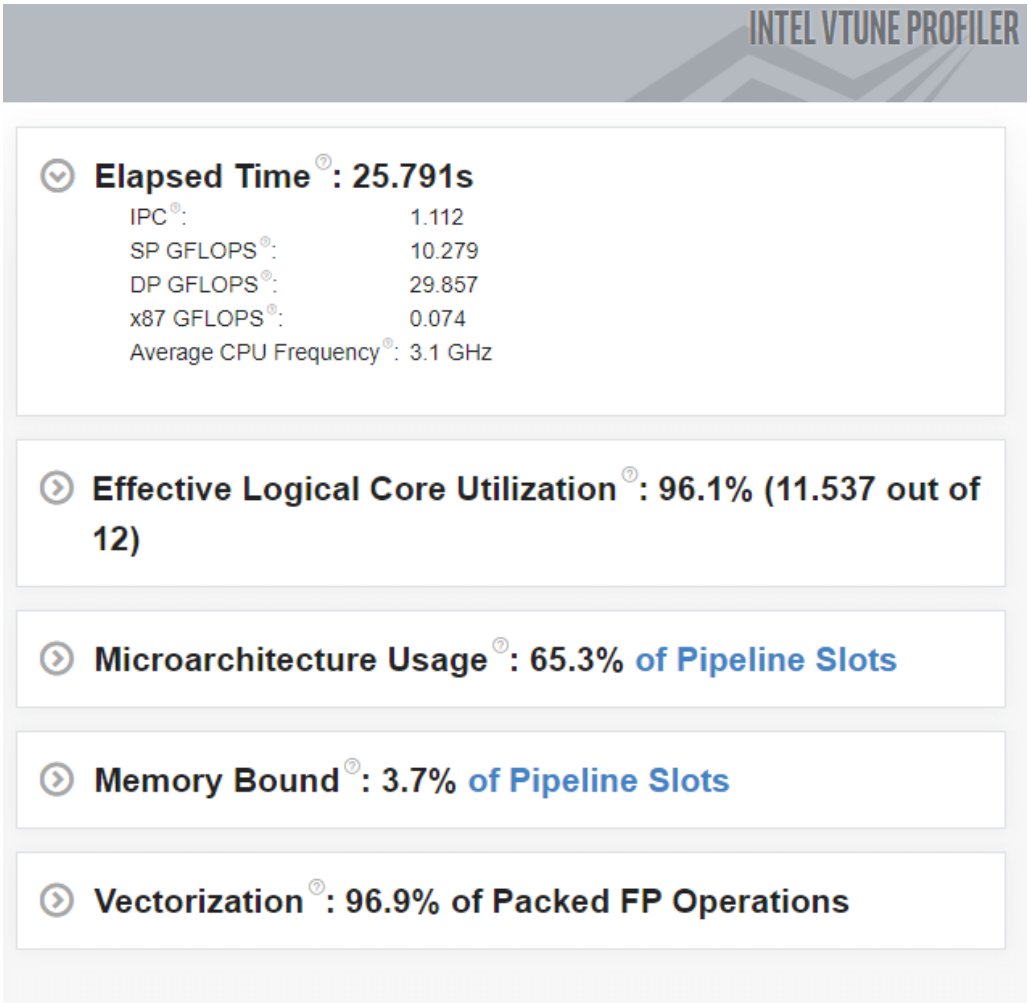
### 1.5.6 Η μεταβλητή `CHUNK_SIZE`

Η μεταβλητή `CHUNK_SIZE` υπάρχει για τη βελτιστοποίηση του παραλληλισμού στη συνάρτηση `euclidean_distance`. Συγκεκριμένα, η επανάληψη που υπάρχει στη συνάρτηση `euclidean_distance` εκτελείται 1000 φορές. Επομένως, το `chunk` που θα πρέπει να μπει στην επιλογή `schedule` που παρέχει το OpenMP θα πρέπει να αξιοποιεί με βέλτιστο τρόπο τον αριθμό των διαθέσιμων threads. Η μαθηματική σχέση που δίνει αυτήν την τιμή είναι:

$$\lceil \frac{1000}{NUM\_THR} \rceil + 1$$

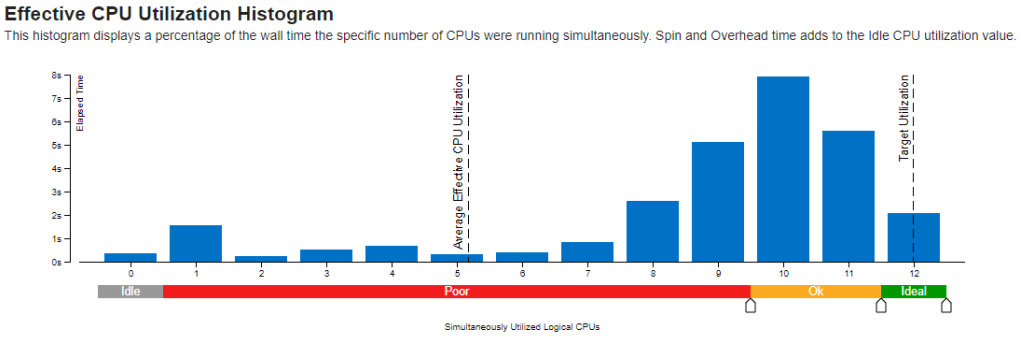
## 1.6 Intel Profiler

Τα αποτελέσματα του Profiler της Intel φαίνονται στο Σχήμα 1.1.



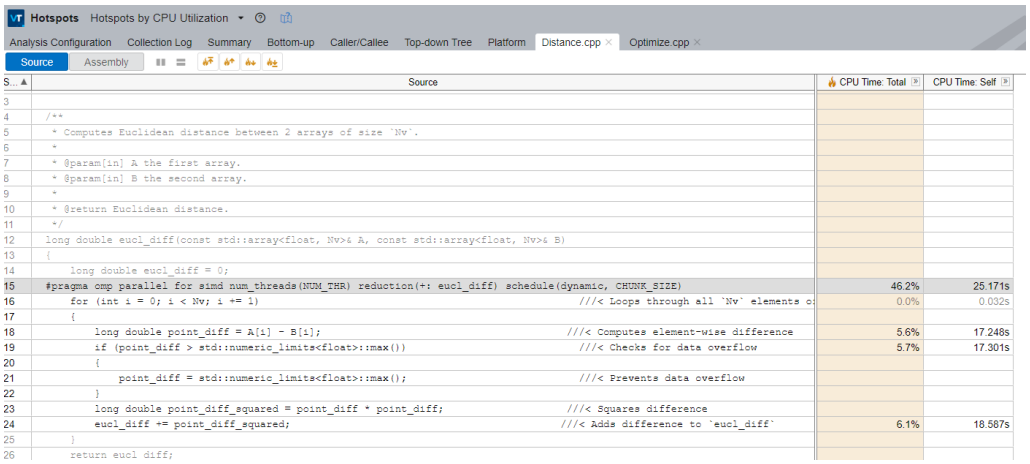

Σχήμα 1.1: Performance Snapshot από τον Profiler της Intel

Όπως φαίνεται και στο Σχήμα 1.1, η αναμονή για το fetching των δεδομένων από τη μνήμη (ποσοστό «Memory Bound») είναι αρκετά μικρή. Επίσης, έχει επιτευχθεί παραλληλοποίηση στο μεγαλύτερο ποσοστό του προγράμματος (Effective Logical Core Utilization 96.1%). Στο συγκεκριμένο μηχάνημα μάλιστα ο συνολικός χρόνος εκτέλεσης του προγράμματος είναι κοντά στα 26 δευτερόλεπτα. Επίσης, στο Σχήμα 1.2 φαίνεται το ραβδόγραμμα με το ποσοστό των Logical Cores που αξιοποιούνταν από το πρόγραμμα ταυτόχρονα.




Σχήμα 1.2: Effective CPU Utilization Histogram

Τέλος, έγινε profiling σε επίπεδο συναρτήσεων, κι όπως προέκυψε ξανά, η πιο «βαριά» συνάρτηση ήταν η `euclidean_distance`. Ωστόσο, σε αυτήν την έκδοση του K-Means το ποσοστό δεν είναι κοντά στο 90% αλλά κοντά στο 43%, όπως φαίνεται και στο Σχήμα 1.3.

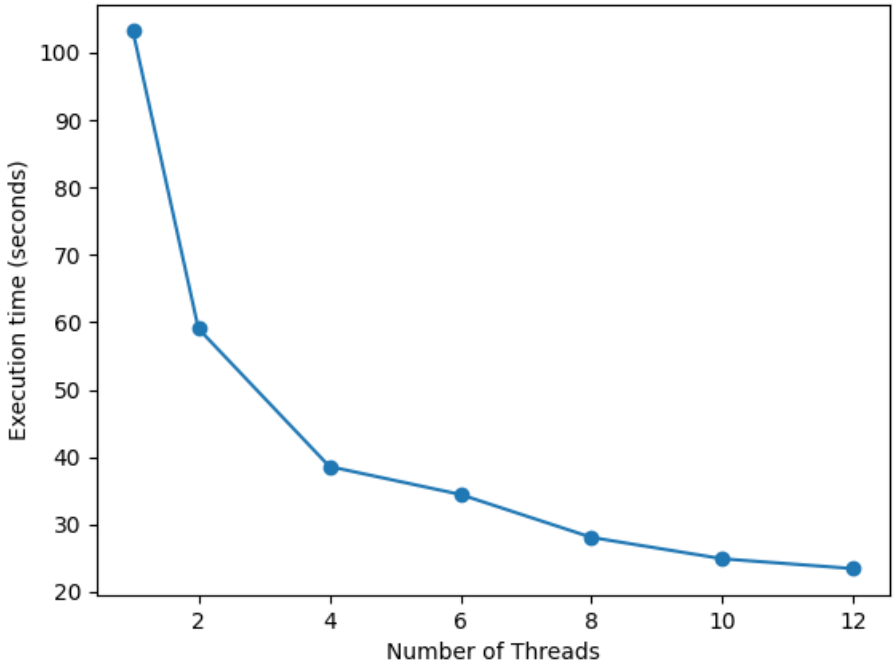



Σχήμα 1.3: K-Means Hotspot

## 1.7 Scaling

Για να αποδειχθεί πως το πρόγραμμα παραλληλοποιείται σωστά, και πως δεν υπάρχουν προβλήματα όπως το *race condition* έγινε περαιτέρω μελέτη. Συγκεκριμένα, το πρόγραμμα εκτελέστηκε αρχικοποιώντας τη μεταβλητή `NUM_THR` με όλες τις δυνατές τιμές. Έτσι, παρατίθεται στο Σχήμα 1.4 η σχετική μελέτη. Να σημειωθεί ότι η μελέτη έγινε εκτελώντας 10 φορές για την κάθε περίπτωση διαθέσιμων threads, καθώς τα δεδομένα αλλά και η αρχικοποίηση των κέντρων είναι τυχαία, οπότε πρέπει να γίνει δίκαια σύγκριση[3].

[3] Ο μέσος αριθμός επαναλήψεων της συγκεκριμένης υλοποίησης του K-Means είναι 26




Σχήμα 1.4: Convergence in Parallelism

2

# Ο κώδικας σε C++ 17

Στο κεφάλαιο αυτό επισυνάπτεται ο σχετικός κώδικας για την εργασία. Λόγω μη συμβατότητας Visual Studio και Latex κώδικα, το identation στα σχόλια είναι λάθος. Οπότε ο κώδικας παρακάτω δεν έχει όμορφη εικόνα. Επομένως, προτείνεται ο κώδικας να εξεταστεί είτε από το GitHub repository της εργασίας, είτε τοπικά από τα αρχεία που βρίσκονται στο συμπιεσμένο.

Κώδικας 2.0.1: Η κεφαλίδα Common.h

```cpp
/**
 * Common.h
 *
 * In this header file, we define the constants
 * used throughout the K-means algorithm. We also
 * include all the header files necessary to make
 * the implementation work.
 */

#pragma once

#include <iostream>                             ///< std::cout
#include <vector>                               ///< std::vector
#include <iomanip>                              ///< std::setw
#include <limits>                               ///< std::numeric_limits
#include <chrono>                               ///< std::chrono
#include <array>                                ///< std::array
#include <algorithm>                            ///< std::copy
#include <utility>                              ///< std::pair
#include <string>                               ///< std::string
#include <cstdlib>                              ///< std::abs
#include <random>                               ///< std::random_device
#include <fstream>                              ///< std::ofstream
#include <omp.h>                                ///< OpenMP Multiprocessing Programming Framework

constexpr int TEST_MODE = 0;                    ///< Set it equal to 1 to enter test mode, otherwise set it to 0
constexpr int NUM_THR = 12;                     ///< Set it to the Number of threads
constexpr int CHUNK_SIZE = (int)(1000 / NUM_THR) + 1; ///< Set it to the \ceil{\frac{Nv}{NUM_THR}}
constexpr int VERBOSITY = 3;                    ///< Sets K-means' iterface verbosity level
constexpr int MAX_LIMIT = TEST_MODE == 1 ? 1023 : 63; ///< Upper bound used in the randomly generated dataset of reals
constexpr int N = TEST_MODE == 1 ? 50 : 100000; ///< Elements in our dataset
constexpr int Nv = TEST_MODE == 1 ? 2 : 1000;   ///< Dimentions of each element
constexpr int Nc = TEST_MODE == 1 ? 5 : 100;    ///< Number of clusters, also known as `K`
constexpr double THR_KMEANS = 0.000001;         ///< Threshold used to indicate K-means convergence
```

Κώδικας 2.0.2: Η κεφαλίδα Vec.h

```cpp
/**
 * Vec.h
 *
 * In this header file, we define a function that
 * executes only in the beginning of the algorithm.
 * This function is used to generate a random dataset
 * for the K-means algorithm.
 */

#pragma once

#include "Common.h"

void vec_init(std::vector<std::array<float, Nv>>& Vec);
```

Κώδικας 2.0.3: Το αρχείο Vec.cpp

```cpp
#include "Vec.h"

/**
 * Initializes `Vec` variable.
 *
 * @param[in, out] Vec the random dataset generated for clustering using the K-means algorithm.
 *
 * @see Common.h
 *
 * @remark [<random> Engines and Distributions](https:///<docs.microsoft.com/en-us/cpp/standard-library/random?view=msvc-160#engdist)
 */
void vec_init(std::vector<std::array<float, Nv>>& Vec)
{
	std::random_device rd_Vec;                                          ///< Initializes a random generator
	std::mt19937 mt_Vec(rd_Vec());                                      ///< Uses the mt19937 engine
	std::uniform_real_distribution<float> dist_Vec(0.0, MAX_LIMIT + 0.0); ///< Generates a uniform distribution bounded by the `MAX_LIMIT` set in `Common.h` to prevent number overflow
	for (int i = 0; i < N; i += 1)
	{
		std::array<float, Nv> Elements;                                 ///< Declares a vector `Elements` that temporarily holds the vector that is to be inserted to `Vwc`
#pragma omp simd
		for (int j = 0; j < Nv; j += 1)
		{
			Elements[j] = dist_Vec(mt_Vec);                             ///< Updates contents of `Elements`
		}
		Vec.emplace_back(Elements);                                     ///< Moves `Elements` to `Vec`
	}
}
```

Κώδικας 2.0.4: Η κεφαλίδα Center.h

```cpp
/**
 * Center.h
 *
 * In this header file, we define a function that helps
 * in the random initialization of the variable `Center`.
 * This function is only used in the beginning of the
 * K-means algorithm.
 */

#pragma once

#include "Common.h"
#include "Distance.h"

void init_centers(std::vector<std::array<float, Nv>>& old_Center);
```

Κώδικας 2.0.5: Το αρχείο Center.cpp

```cpp
#include "Center.h"

/**
 * Initializes `Center` variable.
 *
 * @param[in] Vec the dataset generated.
 * @param[in, out] old_Center the random centroids generated in the beginning of the K-means algorithm.
 *
 * @remark [<random> Engines and Distributions](https:///<docs.microsoft.com/en-us/cpp/standard-library/random?view=msvc-160#engdist)
 */
void init_centers(std::vector<std::array<float, Nv>>& old_Center)
{
	std::random_device rd_Center;                                       ///< Initializes a random generator
	std::mt19937 mt_Center(rd_Center());                                ///< Uses the mt19937 engine
	std::uniform_real_distribution<float> dist_Center(0.0, MAX_LIMIT + 0.0); ///< Generates a uniform distribution bounded by the `MAX_LIMIT` set in `Common.h` to prevent number overflow
	for (int i = 0; i < Nc; i += 1)
	{
		std::array<float, Nv> Elements;                                 ///< Declares a vector `Elements` that temporarily holds the vector that is to be inserted to `old_Center`
#pragma omp simd
		for (int j = 0; j < Nv; j += 1)
		{
			Elements[j] = dist_Center(mt_Center);                       ///< Updates contents of `Elements`
		}
		old_Center.emplace_back(Elements);                              ///< Moves `Elements` to `old_Center`
	}
}
```

Κώδικας 2.0.6: Η κεφαλίδα Class.h

```h
/**
 * Class.h
 *
 * In this header file, we a function that assigns
 * the vectors found in the variable `Vec` to a
 * corresponding `Class` or more commonly known as cluster.
 */

#pragma once

#include "Common.h"
#include "Distance.h"

void compute_classes(const std::vector<std::array<float, Nv>>& Vec, const std::vector<std::array<float, Nv>>& old_Center, std::array<std::vector<int>, Nc>& Classes);
```

Κώδικας 2.0.7: Το αρχείο Class.cpp

```cpp
#include "Class.h"

/**
 * Computes `Class` variable.
 *
 * @param[in] Vec the dataset generated.
 * @param[in] old_Center the current centroids.
 * @param[in, out] Classes the classes corresponding to the dataset.
 *
 * @note This function calls `eucl_diff` from Distance.h
 *
 * @see [K-means clustering Algorithm](https:///<medium.com/@jaredchilders_38839/k-means-clustering-algorithm-4334db89bdf3)
 */
void compute_classes(const std::vector<std::array<float, Nv>>& Vec, const std::vector<std::array<float, Nv>>& old_Center, std::array<std::vector<int>, Nc>& Classes)
{
#pragma omp parallel for num_threads(NUM_THR) schedule(dynamic, 10)
	for (int i = 0; i < Nc; i += 1)                                         ///< Clears contents of `Classes` variable
	{
		Classes[i].clear();                                                 ///< Prevents garbage processing
		Classes[i].reserve((int)N / Nc);                                    ///< Optimizes array with high probability
	}
	int argmin_idx = -1;                                                    ///< Initializes \argmin index
	long double argmin_val = std::numeric_limits<long int>::max() + 0.0;    ///< Initializes \argmin value
#pragma omp parallel for simd num_threads(NUM_THR) firstprivate(argmin_idx, argmin_val) schedule(dynamic, 1000)
	for (int i = 0; i < N; i += 1)                                          ///< Loop through `Vec`
	{
		for (int j = 0; j < Nc; j += 1)                                     ///< Loop through `old_Center`
		{
			long double temp_eucl_dist = eucl_diff(Vec.at(i), old_Center.at(j));///< Compute Euclidean distance between parsed `Vec` element and parsed `old_Center` element
			if (argmin_val > temp_eucl_dist)                                ///< Checks if this Euclidean distance is so far the smallest
			{
				argmin_val = temp_eucl_dist;                                ///< Updates \argmin value
				argmin_idx = j;                                             ///< Updates \argmin index
			}
		}
#pragma omp critical
		Classes[argmin_idx].emplace_back(i);
		argmin_idx = -1;                                                    ///< Sets \argmin index for the next loop
		argmin_val = std::numeric_limits<unsigned long int>::max();         ///< Sets \argmin value for the next loop
	}
}
```

Κώδικας 2.0.8: Η κεφαλίδα Optimize.h

```h
/**
 * Optimize.h
 *
 * In this header file, we define a function which
 * computes the error of the `Center` variable with
 * respect to the `Vec` variable.
 */

#pragma once

#include "Common.h"

void optimize_center(const std::vector<std::array<float, Nv>>& Vec, std::vector<std::array<float, Nv>>& new_Center, const std::array<std::vector<int>, Nc>& Classes);
```

Κώδικας 2.0.9: Το αρχείο Optimize.cpp

```cpp
#include "Optimize.h"

/**
 * Computes `Center` variable after `Classes` correction.
 *
 * @param[in] Vec the dataset.
 * @param[in] Classes the classes corresponding to the dataset.
 * @param[in, out] new_Center the new clusters of the K-means algorithm.
 */
void optimize_center(const std::vector<std::array<float, Nv>>& Vec, std::vector<std::array<float, Nv>>& new_Center, const std::array<std::vector<int>, Nc>& Classes)
{
#pragma omp parallel for num_threads(NUM_THR) schedule(dynamic, 10)
	for (int i = 0; i < Nc; i += 1)                                         ///< Loops through all clusters
	{
		std::array<float, Nv> Element = { 0.0 };                            ///< Initializes `Element` array which holds a cluster vector
		for (int j = 0; j < Classes[i].size(); j += 1)                      ///< Loops through `Vec` vectors
		{
#pragma omp simd
			for (int k = 0; k < Nv; k += 1)                                 ///< Loops through vector elements
			{
				Element[k] += (float)(Vec.at(Classes[i].at(j))[k] / Classes[i].size());///< Computes vector element
			}
		}
		new_Center.at(i) = Element;                                         ///< Updates `new_Center` variable
	}
}
```

Κώδικας 2.0.10: Η κεφαλίδα Distance.h

```h
/**
 * Distance.h
 *
 * In this header file, we define some functions which
 * help with the computations needed to define whether
 * K-means has converged or not.
 */

#pragma once

#include "Common.h"

long double eucl_diff(const std::array<float, Nv>& src, const std::array<float, Nv>& dst);
long double convergence(const std::vector<std::array<float, Nv>>& curr_Center, const std::vector<std::array<float, Nv>>& prev_Center);
long double normalize_convergence(const long double curr_iter_conv, const long double prev_iter_conv, const int iter_counter);
```

Κώδικας 2.0.11: Το αρχείο Distacne.cpp

```cpp
#include "Distance.h"

/**
 * Computes Euclidean distance between 2 arrays of size `Nv`.
 *
 * @param[in] A the first array.
 * @param[in] B the second array.
 *
 * @return Euclidean distance.
 */
long double eucl_diff(const std::array<float, Nv>& A, const std::array<float, Nv>& B)
{
	long double eucl_diff = 0;
#pragma omp parallel for simd num_threads(NUM_THR) reduction(+: eucl_diff) schedule(dynamic, CHUNK_SIZE)
	for (int i = 0; i < Nv; i += 1)                                    ///< Loops through all `Nv` elements of the given arrays
	{
		long double point_diff = A[i] - B[i];                          ///< Computes element-wise difference
		if (point_diff > std::numeric_limits<float>::max())            ///< Checks for data overflow
		{
			point_diff = std::numeric_limits<float>::max();                ///< Prevents data overflow
		}
		long double point_diff_squared = point_diff * point_diff;      ///< Squares difference
		eucl_diff += point_diff_squared;                               ///< Adds difference to `eucl_diff`
	}
	return eucl_diff;
}

/**
 * Checks if K-means has converged.
 *
 * @param[in] curr_Center the current `Center` variable .
 * @param[in] prev_Center the old `Center` variable.
 *
 * @return Euclidean distance between the 2 vectors.
 *
 * @note if the Euclidean distance between 2 points is not greater than 1.0, then it is normalized to 0.
 *       This is to optimize speed, prevent overfitting, and solve the precision error carried throughout the $Nc \times Nv = 1{,}000{,}000$ additions.
 */
long double convergence(const std::vector<std::array<float, Nv>>& curr_Center, const std::vector<std::array<float, Nv>>& prev_Center)
{
	long double convergence_sum = 0;                                   ///< Initializes `convergence_sum` variable used to store the total additive difference between `curr_Center` and `prev_Center`
	for (int i = 0; i < Nc; i += 1)                                    ///< Loops through all centers
	{
		long double tmp_eucl_d = eucl_diff(curr_Center.at(i), prev_Center.at(i));///< Computes element-wise Euclidean distance
		convergence_sum += tmp_eucl_d > 1.0 ? tmp_eucl_d : 0.0;        ///< Optimizes Convergence if error is close to defined threshold
		if (convergence_sum > std::numeric_limits<double>::max())      ///< Checks for number overflow
		{
			convergence_sum = std::numeric_limits<double>::max();          ///< Prevents number overflow
		}
	}
	return convergence_sum;
}

/**
 * Normalizes the error between current `Center` variable and old `Center` variable.
 *
 * @param[in] curr_iter_conv the current `Center` error.
 * @param[in] prev_iter_conv the old `Center` error.
 * @param[in] iter_counter the loop number.
 *
 * @return normalized Euclidean distance between centers.
 *
 * @note The computation of the normalization formula is explained below:
 *       if the current loop error is less or equal to the number of clusters, then $normalized = 0.0$
 *       if the current iteration of the optimization loop is less or equal to 2, then $normalized = 1.0$
 *       else, $normalized = \frac{|curr_iter_conv - prev_iter_conv|}{\arg\max curr_iter_conv, prev_iter_conv}$
 */
long double normalize_convergence(const long double curr_iter_conv, const long double prev_iter_conv, const int iter_counter)
{
	return curr_iter_conv > Nc && iter_counter > 2 ? std::abs(curr_iter_conv - prev_iter_conv) / std::max(curr_iter_conv, prev_iter_conv) :
		iter_counter <= 2 ? 1.0 : 0.0;
}
```

Κώδικας 2.0.12: Η κεφαλίδα Interface.h

```cpp
/**
 * interface.h
 *
 * In this header file, we define some functions which
 * implement a basic User Interface (UI) throughout the
 * execution of the K-means algorithm.
 */

#pragma once

#include "Common.h"

void multidimentional_float_vector_interface(const std::vector<std::array<float, Nv>>& Obj, std::string Obj_name);
void multidimentional_int_array_interface(const std::array<std::vector<int>, Nc>& Obj, std::string Obj_name);
void progress_interface(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark);
void kmeans_progress(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark, const int verbose);
void kmeans_termination(const std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_results, const long double iter_conv);
```

Κώδικας 2.0.13: Το αρχείο Interface.cpp

```cpp
#include "Interface.h"

/**
 * Prints the contents of a vector of arrays.
 *
 * @param[in] Obj the vector of arrays.
 * @param[in] obj_name the name of the variable.
 *
 * @note This function helps in printing the contents of the variables `Vec`, `old_Centers` and `new_Centers`.
 */
void multidimentional_float_vector_interface(const std::vector<std::array<float, Nv>>& Obj, std::string Obj_name)
{
	for (int i = 0; i < Obj.size(); i += 1)
	{
		for (int j = 0; j < Obj.at(i).size(); j += 1)
		{
			std::cout << std::setw(11) << Obj_name << "[" << i << "][" << j << "]:\t" << std::setw(18) << Obj.at(i)[j] << " ";
		}
		std::cout << std::endl;
	}
	std::cout << std::endl << std::endl;
}

/**
 * Prints the contents of an array of vectors.
 *
 * @param[in] Obj the array of vectors.
 * @param[in] obj_name the name of the variable.
 *
 * @note This function helps in printing the contents of the variable `Classes`.
 */
void multidimentional_int_array_interface(const std::array<std::vector<int>, Nc>& Obj, std::string Obj_name)
{
	for (int i = 0; i < Obj.size(); i += 1)
	{
		for (int j = 0; j < Obj[i].size(); j += 1)
		{
			std::cout << Obj_name << "[" << i << "][" << j << "]: " << std::setw(4) << Obj[i].at(j) << " ";
			if (j % 6 == 0 && j != 0)
			{
				std::cout << std::endl;
			}
		}
		std::cout << std::endl << std::endl;
	}
	std::cout << std::endl << std::endl;
}

/**
 * Prints the progress of the K-means algorithm.
 *
 * @param[in] iter_counter the current loop number.
 * @param[in] iter_conv the current loop error.
 * @param[in] norm_iter_conv the current normalized loop error.
 * @param[in] loop_benchmark the current loop benchmark
 */
void progress_interface(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark)
{
	std::cout << "Iteration: " << std::setw(2) << iter_counter
		<< "\tError: " << std::setw(std::numeric_limits<float>::max_digits10) << iter_conv
		<< "\tNormalized error: " << std::setw(std::numeric_limits<float>::max_digits10) << norm_iter_conv
		<< "\tLoop checkpoint: " << std::setw(std::numeric_limits<double>::max_digits10) << loop_benchmark.count();
	TEST_MODE == 1 ? (std::cout << std::endl << std::endl << std::endl) : std::cout << std::endl;
}

/**
 * Calls `progress_interface` function to print K-means progress.
 *
 * @param[in] iter_counter the current loop number.
 * @param[in] norm_iter_conv the current normalized loop error.
 * @param[in] iter_conv the current loop error.
 * @param[in] loop_benchmark the current loop benchmark
 * @param[in] verbose the frequency used for activating the interface
 *                if equal to 0, then `progress_interface` is called every 4 loops
 *                if equal to 1, then `progress_interface` is called every 3 loops
 *                if equal to 2, then `progress_interface` is called every 2 loops
 *                if equal to 3, then `progress_interface` is called in every loop
 *
 * @see `progress_interface`
 */
void kmeans_progress(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark, const int verbose)
{
	switch (verbose)
	{
	case 0: if (iter_counter % 4 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	case 1: if (iter_counter % 3 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	case 2: if (iter_counter % 2 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	case 3: if (iter_counter % 1 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	default:
		break;
	}
}

/**
 * Prints the resuts of the K-means algorithm.
 *
 * @param[in] bench_results the benchmarking results.
 * @param[in] iter_conv the final normalized K-means error.
 *
 * @see `Benchmark.h`
 */
void kmeans_termination(const std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_results, const long double iter_conv)
{
	std::cout << std::endl
		<< std::setw(18) << "[TOTAL TIME] " << std::setw(15) << bench_results.first.count()
		<< " {IN SECONDS}" << std::endl
		<< std::setw(18) << "[K-MEANS BENCH] " << std::setw(15) << bench_results.second.count()
		<< " {IN SECONDS}" << std::endl
		<< std::setw(18) << "[CONVERGENCE] " << std::setw(15) << iter_conv << std::endl;
}
```

Κώδικας 2.0.14: Η κεφαλίδα Benchmark.h

```h
/**
 * Benchmark.h
 *
 * In this header file, we define some functions that help us
 * benchmark some crucial parts of the K-means algorithm. Using
 * these functions, we can benchmark the K-means runtime, as well as
 * how long did it take for our implementation to converge.
 */

#pragma once

#include "Common.h"

std::chrono::time_point<std::chrono::system_clock> init_benchmark(void);
std::chrono::time_point<std::chrono::system_clock> bench_convergence(void);
std::chrono::duration<double> bench_loop(std::chrono::time_point<std::chrono::system_clock>, std::chrono::time_point<std::chrono::system_clock>);
std::chrono::time_point<std::chrono::system_clock> terminate_bench(void);
std::pair<std::chrono::duration<double>, std::chrono::duration<double>> benchmark_results(std::chrono::time_point<std::chrono::system_clock> start, std::chrono::time_point<std::chrono::system_clock> loop_benchmark, std::chrono::time_point<std::chrono::system_clock> end);
```

Κώδικας 2.0.15: Το αρχείο Benchmark.cpp

```cpp
#include "Benchmark.h"

/**
 * Launches benchmark.
 *
 * @return the starting timepoint of the K-means algorithm.
 */
std::chrono::time_point<std::chrono::system_clock> init_benchmark(void)
{
        std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_clock::now();
        return start;
}

/**
 * Launches K-means optimization loop benchmark.
 *
 * @return the starting timepoint of the K-means optimization loop.
 *
 * @see kmeans.cpp
 */
std::chrono::time_point<std::chrono::system_clock> bench_convergence(void)
{
        std::chrono::time_point<std::chrono::system_clock> loop_benchmark = std::chrono::system_clock::now();
        return loop_benchmark;
}

/**
 * Computes per-loop benchmark.
 *
 * @param[in] startpoint loop starting time point
 * @param[in] endpoint loop ending time point
 *
 * @return optimization loop benchmark.
 *
 * @see kmeans.cpp
 */
std::chrono::duration<double> bench_loop(std::chrono::time_point<std::chrono::system_clock> startpoint, std::chrono::time_point<std::chrono::system_clock> endpoint)
{
        return endpoint - startpoint;
}

/**
 * Stops K-means benchmark.
 *
 * @return the ending timepoint of the K-means algorithm.
 */
std::chrono::time_point<std::chrono::system_clock> terminate_bench(void)
{
        std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_clock::now();
        return end;
}

/**
 * Computes K-means benchmarking results.
 *
 * @param[in] start the starting timepoint of the K-means algorithm.
 * @param[in] loop_benchmark the starting timepoint of the K-means optimization loop.
 * @param[in] end the ending timepoint of the K-means algorithm.
 *
 * @return a pair with the total execution time of the K-means algorithm in seconds
 *                and the total time in seconds that the algorithm needed to converge.
 */
std::pair<std::chrono::duration<double>, std::chrono::duration<double>> benchmark_results(
        const std::chrono::time_point<std::chrono::system_clock> start,
        const std::chrono::time_point<std::chrono::system_clock> loop_benchmark,
        const std::chrono::time_point<std::chrono::system_clock> end)
{
        std::chrono::duration<double> elapsed_seconds = end - start;
        std::chrono::duration<double> k_means_benchmark = end - loop_benchmark;
        std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_res;
        bench_res.first = elapsed_seconds;
        bench_res.second = k_means_benchmark;
        return bench_res;
}
```

Κώδικας 2.0.16: Η κεφαλίδα Validation.h

```h
/**
 * Validation.h
 *
 * In this header file, we define some functions which
 * help the developer validate the data computed.
 * This is done by exporting the data to CSV files.
 */

#pragma once

#include "Common.h"

void export_multidimentional_float_vector(const std::vector<std::array<float, Nv>> Obj, std::string filename);
void export_multidimentional_integer_array(const std::array<std::vector<int>, Nc> Obj, std::string filename);
void track_kmeans_progress(int iter_counter, long double norm_iter_conv);
void export_kmeans_progress(std::string filename);
```

Κώδικας 2.0.17: Το αρχείο Validation.cpp

```cpp
#include "Validation.h"

static std::vector<std::pair<int, long double>> kmeans_progress;            ///< Variable used to store K-means progress

/**
 * Exports the contents of a vector of arrays to a CSV file.
 *
 * @param[in] Obj the vector of arrays.
 * @param[in] filename the name of file.
 *
 * @note This function helps in exporting the contents of the variables `Vec`, `old_Centers` and `new_Centers`.
 */
void export_multidimentional_float_vector(const std::vector<std::array<float, Nv>> Obj, std::string filename)
{
	std::ofstream export_stream;                                            ///< Defines an output file stream
	export_stream.open("./data/" + filename + ".csv");                      ///< Associates `export_stream` with a CSV file named after the `filename` variable
	for (int i = 0; i < Obj.size(); i += 1)                                 ///< Loops through `Obj`
	{
		for (int j = 0; j < Obj.at(i).size(); j += 1)                       ///< Loops through the elements of the array inside `Obj` that is being parsed
		{
			export_stream << Obj.at(i)[j] << (j < Obj.at(i).size() - 1 ? "," : "");///< Exports element of that array to the `export_stream` file stream
		}
		export_stream << std::endl;
	}
	export_stream.close();                                                  ///< Closes file stream
}

/**
 * Exports the contents of an array of vectors into a CSV file.
 *
 * @param[in] Obj the array of vectors.
 * @param[in] filename the name of file.
 *
 * @note This function helps in printing the contents of the variable `Classes`.
 */
void export_multidimentional_integer_array(const std::array<std::vector<int>, Nc> Obj, std::string filename)
{
	std::ofstream export_stream;                                            ///< Defines an output file stream
	export_stream.open("./data/" + filename + ".csv");                      ///< Associates `export_stream` with a CSV file named after the `filename` variable
	for (int i = 0; i < Obj.size(); i += 1)                                 ///< Loops through `Obj`
	{
		for (int j = 0; j < Obj[i].size(); j += 1)                          ///< Loops through the elements of the vector inside `Obj` that is being parsed
		{
			export_stream << Obj[i].at(j) << (j < Obj.at(i).size() - 1 ? "," : "");///< Exports element of that vector to the `export_stream` file stream
		}
		export_stream << std::endl;
	}
	export_stream.close();                                                  ///< Closes file stream
}

/**
 * Updates `kmeans_progress` variable, which will be used to export data about K-means progress.
 *
 * @param[in] iter_counter the current loop number.
 * @param[in] norm_iter_conv the current normalized loop error.
 */
void track_kmeans_progress(int iter_counter, long double norm_iter_conv)
{
	kmeans_progress.push_back(std::make_pair(iter_counter, norm_iter_conv));
}

/**
 * Exports the contents of the `kmeans_progress` variable into a CSV file.
 *
 * @param[in] filename the name of file.
 */
void export_kmeans_progress(std::string filename)
{
	std::ofstream export_stream;                                            ///< Defines an output file stream
	export_stream.open("./data/" + filename + ".txt");                      ///< Associates `export_stream` with a CSV file named after the `filename` variable
	for (int i = 0; i < kmeans_progress.size(); i += 1)                     ///< Loops through `kmeans_progress`
	{
		export_stream <<
			"Iteration: " << kmeans_progress.at(i).first <<
			"Convergence: " << kmeans_progress.at(i).second << std::endl;   ///< Exports an element of `kmeans_progress` vector to the `export_stream` file stream
	}
	export_stream.close();                                                  ///< Closes file stream
}
```

Κώδικας 2.0.18: Η κεφαλίδα kmeans.h

```
/**
 * kmeans.h
 *
 * In this header file, we include all the
 * custom made header files used in the
 * K-means algorithm.
 */

#pragma once

#include "Vec.h"
#include "Center.h"
#include "Class.h"
#include "Common.h"
#include "Optimize.h"
#include "Distance.h"
#include "Interface.h"
#include "Benchmark.h"
#include "Validation.h"
```

Κώδικας 2.0.19: Το αρχείο k-means-release.cpp

```
#include "kmeans.h"

/**
 * Implements K-means clustering algorithm.
 *
 * @return 0, if the executable was terminated normally.
 *
 */
int main(void)
{
	std::chrono::time_point<std::chrono::system_clock> loop_benchmark, prev_loop_benchmark, start;///< Initializes loop-wise benchmark variables
	std::cout.precision(std::numeric_limits<float>::max_digits10);                                ///< Sets output precision for long doubles
	prev_loop_benchmark = start = init_benchmark();                                               ///< Starts custom benchmark
	std::vector<std::array<float, Nv>> Vec;                                                       ///< Initializes the `Vec` variable
	std::array<std::vector<int>, Nc> Classes;                                                     ///< Initializes `Classes` variable, which holds all cluster pointers
	std::vector<std::array<float, Nv>> old_Center;                                                ///< Initializes `old_Center` variable, which holds centroids
	std::vector<std::array<float, Nv>> new_Center(Nc);                                            ///< Initializes the `new_Center` variable, which holds optimized centroids
	long double iter_conv = std::numeric_limits<double>::infinity();                              ///< Initializes `iter_conv` variable, which holds the sum of distances betweens `Vec` and `new_Center`
	long double norm_iter_conv = std::numeric_limits<double>::infinity();                         ///< Initializes `norm_iter_conv` variable, which holds the normalized value of `iter_conv`
#pragma omp parallel num_threads(2)
	{
#pragma omp master
		{
			Vec.reserve(N);                                                                   ///< Reserves `N` blocks of memory to increase speed
			vec_init(Vec);                                                                    ///< Fills `Vec` with random real numbers
		}
#pragma omp single
		{
			old_Center.reserve(Nc);                                                           ///< Reserves `Nc` blocks of memory to increase speed
			init_centers(old_Center);                                                         ///< Fills `old_Center` with random vectors from `Vec`
		}
	}
	std::chrono::time_point<std::chrono::system_clock> optimization_benchmark = init_benchmark(); ///< K-means optimization loop benchmark
	for (int iter_counter = 1; norm_iter_conv > THR_KMEANS; iter_counter += 1)                    ///< K-means optimazation loop
	{
		std::chrono::duration<double> bench_res =
			bench_loop(prev_loop_benchmark, loop_benchmark = init_benchmark());           ///< Benchmarks loop
		kmeans_progress(iter_counter, iter_conv, norm_iter_conv, bench_res, VERBOSITY);       ///< Outputs K-means progress
		compute_classes(Vec, old_Center, Classes);                                            ///< Re-computes `Classes` contents
		optimize_center(Vec, new_Center, Classes);                                            ///< Computes error of current clusters and stores it to the variable `new_Center`
		long double prev_iter_conv = iter_conv;                                               ///< Stores previous `iter_conv` to use it to normalize convergence
		iter_conv = convergence(new_Center, old_Center);                                      ///< Computes convergence of current K-means loop
		norm_iter_conv = normalize_convergence(iter_conv, prev_iter_conv, iter_counter);      ///< Normalizes `iter_conv` to avoid overfitting
		old_Center.swap(new_Center);                                                          ///< Swaps `old_Center` with `new_Center` to start the next loop
	}
	std::chrono::time_point<std::chrono::system_clock> end = terminate_bench();               ///< Initializes the endpoint of K-means benchmark
	std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_res =
		benchmark_results(start, optimization_benchmark, end);                            ///< Stores benchmark results
	kmeans_termination(bench_res, norm_iter_conv);                                            ///< Outputs benchmark results
	return 0;
}
```

Κώδικας 2.0.20: Το αρχείο k-means-debug.cpp

```cpp
#include "kmeans.h"

/**
 * Implements K-means clustering algorithm.
 *
 * @return 0, if the executable was terminated normally.
 *
 * @note This is the debug version
 *               The debug version prints out results and helps the tester
 *               to explore the different steps of the algorithm.
 */
int main(void)
{
	std::chrono::time_point<std::chrono::system_clock> loop_benchmark, prev_loop_benchmark, start; ///< Initializes loop-wise benchmark variables
	std::cout.precision(std::numeric_limits<float>::max_digits10);                                 ///< Sets output precision for long doubles
	prev_loop_benchmark = start = init_benchmark();                                                ///< Starts custom benchmark
	std::vector<std::array<float, Nv>> Vec;                                                        ///< Initializes the `Vec` variable
	std::array<std::vector<int>, Nc> Classes;                                                      ///< Initializes `Classes` variable, which holds all cluster pointers
	std::vector<std::array<float, Nv>> old_Center;                                                 ///< Initializes `old_Center` variable, which holds centroids
	std::vector<std::array<float, Nv>> new_Center(Nc);                                             ///< Initializes the `new_Center` variable, which holds optimized centroids
	long double iter_conv = std::numeric_limits<double>::infinity();                               ///< Initializes `iter_conv` variable, which holds the sum of distances betweens `Vec` and `new_Center`
	long double norm_iter_conv = std::numeric_limits<double>::infinity();                          ///< Initializes `norm_iter_conv` variable, which holds the normalized value of `iter_conv`
#pragma omp parallel num_threads(2)
	{
#pragma omp master
		{
			Vec.reserve(N);                                                                ///< Reserves `N` blocks of memory to increase speed
			vec_init(Vec);                                                                 ///< Fills `Vec` with random real numbers
		}
#pragma omp single
		{
			old_Center.reserve(Nc);                                                        ///< Reserves `Nc` blocks of memory to increase speed
			init_centers(old_Center);                                                      ///< Fills `old_Center` with random vectors from `Vec`
		}
	}
#pragma omp parallel num_threads(4)
	{
#pragma omp master
		{
			multidimentional_float_vector_interface(Vec, "Vec");                           ///< Outputs `Vec` contents
		}
#pragma omp single
		{
			export_multidimentional_float_vector(Vec, "Vec");                              ///< Exports `Vec` contents to CSV file
		}
#pragma omp single
		{
			multidimentional_float_vector_interface(old_Center, "old_Center");             ///< Outputs `old_Center` contents
		}
#pragma omp single
		{
			export_multidimentional_float_vector(old_Center, "old_Center");                ///< Exports `old_Center` contents to CSV file
		}
	}
	std::chrono::time_point<std::chrono::system_clock> optimization_benchmark = init_benchmark();  ///< K-means optimization loop benchmark
	for (int iter_counter = 1; norm_iter_conv > THR_KMEANS; iter_counter += 1)                     ///< K-means optimazation loop
	{
		std::chrono::duration<double> bench_res =
			bench_loop(prev_loop_benchmark, loop_benchmark = init_benchmark());    ///< Benchmarks loop
#pragma omp parallel num_threads(3)
		{
#pragma omp master
			{
				kmeans_progress(iter_counter, iter_conv, norm_iter_conv, bench_res, VERBOSITY);  ///< Outputs K-means progress
			}
#pragma omp single
			{
				track_kmeans_progress(iter_counter, norm_iter_conv);                     ///< Stores K-means progress
			}
#pragma omp single
			{
				compute_classes(Vec, old_Center, Classes);                               ///< Re-computes `Classes` contents
			}
		}
#pragma omp parallel num_threads(3)
		{
#pragma omp master
			{
				optimize_center(Vec, new_Center, Classes);                               ///< Computes error of current clusters and stores it to the variable `new_Center`
			}
#pragma omp single
			{
				multidimentional_int_array_interface(Classes, "Classes");                ///< Outputs `Classes` contents
			}
#pragma omp single
			{
				export_multidimentional_integer_array(Classes, "Classes" + std::to_string(iter_counter));///< Exports `Classes` contents to CSV file
			}
		}
		long double prev_iter_conv = iter_conv;                                          ///< Stores previous `iter_conv` to use it to normalize convergence
#pragma omp parallel num_threads(3)
		{
#pragma omp master
			{
				iter_conv = convergence(new_Center, old_Center);                 ///< Computes convergence of current K-means loop
				norm_iter_conv = normalize_convergence(iter_conv, prev_iter_conv, iter_counter);  ///< Normalizes `iter_conv` to avoid overfitting
			}
#pragma omp single
			{
				multidimentional_float_vector_interface(new_Center, "new_Center");       ///< Outputs `new_Center` contents
			}
#pragma omp single
			{
				export_multidimentional_float_vector(new_Center, "new_Center" +
					std::to_string(iter_counter));                                   ///< Exports `new_Center` contents to CSV file
			}
		}
		old_Center.swap(new_Center);                                             ///< Swaps `old_Center` with `new_Center` to start the next loop
		multidimentional_float_vector_interface(old_Center, "old_Center");       ///< Outputs `old_Center` contents
	}
	std::chrono::time_point<std::chrono::system_clock> end = terminate_bench();          ///< Initializes the endpoint of K-means benchmark
	std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_res =
		benchmark_results(start, optimization_benchmark, end);                   ///< Stores benchmark results
	kmeans_termination(bench_res, norm_iter_conv);                                   ///< Outputs benchmark results
	export_kmeans_progress("kmeans_results");                                        ///< Exports K-means progress into a CSV file
	return 0;
}
```