Capici Alessandro

1-Analysis of the serial algorithm

1.1-What is a clustering algorithm?

A clustering algorithm is a method used in order to process data into clusters. There are several ways to do so and the chosen algorithm for the project is the K-Means.

1.2-K-Means

The K-Means algorithm starts with a given number of points (N) and a given number of clusters (K). The score subdivides the dataset into K different clusters. In order to do this, I initialize K of the N points as centroids (a centroid is the center of a cluster).The distance from every point to every centroid is iteratively calculated in order to bind the point to the nearest cluster. At the end of each iteration the coordinates of each centroid is updated and this operation is done using the following formula:

$$C_i = \frac{1}{|S_i|} * \sum_{x_i \in S_i} x_i$$

C is the centroid, *i* is referred to the *i-th* coordinate(x,y,z).
S is the sum of the points *belonging to the centroid*.
X is the coordinate *i-th of the point*.

If the **stop criteria** are not satisfied it's computed again the iteration, which means:

1. Calculate which cluster the points belong to.
2. Update the centroids.

1.2.1 Stop criteria

The exit conditions are represented by one of the following options:

1. There is no datapoint that changes the cluster from one iteration to another.
2. We reach the maximum number of iterations.
3. The error is small enough.

2- A-priori study of available parallelism

2.1-Serial implementation

It was decided to represent ***points*** and ***centroids*** as structs.

```
typedef struct {
    int ID_point;
    int ID_cluster;
    float coordinates[DIMENSIONS];
}point;
```

```
typedef struct {
    int ID_cluster;
    int count_points;
    float coordinates[DIMENSIONS];
    int sum_coordinates[DIMENSIONS];
}centroid;
```

I took the decision of representing points and centroid with different structures because they have different attributes and different meanings.
The main functions implemented are the following:

- *findEuclidianDistance();*
- *processClusterSerial();*
- *replaceCentroid();*

The most interesting function to analyze is processClusterSerial. This function is in charge of finding which cluster every point belongs to , updating the centroids and monitoring when the stop criteria are satisfied.
The main part of the function is a while loop, which counts the number of iterations as we described below. Inside this while loop there is a for loop that calculates for each point the distance from all centroids and matches the point with the nearest cluster. At the end of the for, but still inside the while loop, the centroids will be updated.
Below is reported the pseudo code processClusterSerial:

```
processClusterSerial {
        While(stop Conditions) {
                resetSumCoordinate{}
                Pragma parallel{
                        pragma for {
                                for(All The points) {
                                        for(All The Clusters) {
                                                findEuclidianDistance();
                                                If(the distance is the smallest) {
                                                        match the point with the centroid
                                                }
                                        }
                                }
                        }
                        Pragma critical
                        sumCoordinates();
                }
                }
                pragma parallel for
                replaceCentroid();
        }
}
```

It was clear from the beginning that the for loop that calculates the distances has independent iterations one from another , so this is a section(the red one) that can be parallelized very well.The section of code containing the while loop(the blue one), on the other hand, cannot be parallelized as there is a dependence between cycles due to the updating of the coordinates of the centroids.

2.2-Prior Speed-up

$$Amdahl\ law: Speedup = \frac{1}{(1-Parallelizable\ code)+\frac{Parallelizable\ code}{number\ of\ threads}}$$

In order to better use the profiling tools, I decided to put most of the code inside functions so that they could be visualized and profiled by the various tools used. Some of these functions were then removed in the parallel version, to avoid continual program counter changes, the relative jumps and therefore a loss of pipeline efficiency.
To calculate the speed-up I decided to rely on a tool called valgrind.

2.2.1-Valgrind

Valgrind is a profiling and memory debugging tool. It provides various tools, including callgrind which takes care of showing how many functions are called, the time taken by each function and the machine instructions executed for each function.
The steps to get this information were the following:

- Compiling the source by setting the -g flag to the gcc compiler. This flag produces debugging information in the operating system's native format.
- *valgrind -tool=callgrind ./serial* : I use the callgrind tool on my executable. The output result will be saved in a file with integer extension .PID eg callgrind.out.1987
- *callgrind_annotate callgrind.out.1987 -inclusive=yes --tree=both* : I annotate the result of the out file with various representation flags, including function display and hierarchy.
- after which I filter the result with grep to get just the lines I am looking for.

The result file performed on a total of 10000 points and 3 centroids is the following (I assumed that the instructions were linearly linked by the number of centroid points and iterations)

- Total instruction : 3278557
- *processClusterSerial*: 3276086
  that are subdivided in:
  - *findEuclidianDistance: 1724787*
  - *sumCoordinates:720000*
  - *replaceCentroids:104000*

The remaining lines of code (*1724787+720000+104000 is not equal to* 3276086 of ProcessClusterSerial) are composed of the external while loop and of the search of the minimum “check if”. These parts have not been inserted in any function, but they are still parallelized. Therefore only the initialization of the variables and the while loop are not parallelized, so I can assume that the total number of instructions is about equal to processClusterSerial function. So by adding all the parallelizable parts I will obtain the speed-up.
At this point it is necessary to make a clarification: since sumCoordinate will be inserted (see point 3) in a critical section, the percentage of parallelized code cannot be calculated . So I get a speed-up range that varies between:

- ≈ 4 in case all threads want to access the critical section at the same time thus obtaining a serial execution
- $\approx 23,7$in case there is never a queue to access the critical section

Obviously these are extreme cases whose probability of verification is very low.
In this way, the speed-up does not take into account the time lost for any cache.
Since I made this clarification, I decided to also use another profiling tool called gprof. Using gprof, we can see how in the output file the functions that take up most of the time and memory out of the whole program are the *findEuclidianDistance* and *processClusterSerial* .

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 47.11      0.40     0.40        1   400.48   851.01  processClusterSerial
 26.50      0.63     0.23 24000000     0.00     0.00  findEuclideanDistance3D
 25.91      0.85     0.22  8000000     0.00     0.00  sum_coordinates
  0.59      0.85     0.01     2400     0.00     0.00  replaceCentroid
  0.00      0.85     0.00        1     0.00     0.00  initializeCentroids
  0.00      0.85     0.00        1     0.00   851.01  kMeanSerial
  0.00      0.85     0.00        1     0.00     0.00  read_file3D
```

In this file the various percentages of use of the functions are evident.
From this file I can see that the sum of the times of the functions that can be parallelized is equivalent to 0,8499 s which, divided by the whole time, gives me the percentage of the parallel part which is equivalent to $\approx 0,9998823$ if the critical

section is completely parallelized, and $\approx 0,7411764$ if the critical section is completely serialized.

Again i obtained a range:

- ≈ 4 in case all threads want to access the critical section at the same time thus obtaining a serial execution
- ≈ 24in case there is never a queue to access the critical section

These results are consistent with the analysis of the machine instructions made before.

3-OpenMP parallel implementation

The parallel implementation is the following:

```
int processClusterParallel(int N_points, int K, point *data_points, centroid *centroids, int *num_iterations) {

    *num_iterations = 0;
    bool isChanged = true; // this variable is shared

    int i,j; //private index
    double min_distance, current_distance;
    int p_sum_coordinates_matrix[N_CENTROIDS][DIMENSIONS] = {}; // 'p' means 'private' for each thread
    int p_sum_points[N_CENTROIDS] = {};
    while(*num_iterations < MAX_ITERATIONS && isChanged) {
        int e, s;
        isChanged=false;
        for(e = 0; e < K; e++){
            centroids[e].count_points = 0;
                for(s = 0; s < DIMENSIONS; s++){
                    centroids[e].sum_coordinates[s] = 0;
                }
        }

        #pragma omp parallel shared(isChanged) firstprivate(i,j, min_distance, current_distance, p_sum_coordinates_matrix, p_sum_points)
        {

            #pragma omp for
            for(i = 0; i < N_points; i++) {
                min_distance = __DBL_MAX__; // min_distance is assigned the largest possible double value
                //tie the point with the centroid
                for(j = 0; j < K; j++) {
                    current_distance = sqrt(pow((double) (data_points[i].coordinates[0] - centroids[j].coordinates[0]), 2)+
                                            pow((double) (data_points[i].coordinates[1] - centroids[j].coordinates[1]), 2)+
                                            pow((double) (data_points[i].coordinates[2] - centroids[j].coordinates[2]), 2));
                    if(current_distance < min_distance) {
                        min_distance = current_distance;
                        //assign the ID of the cluster as the number of the centroid
                        data_points[i].ID_cluster = j;
                        isChanged = true;
                    }
                }
                // update the number of point for the centroid
                p_sum_points[data_points[i].ID_cluster]++;

                //start the sum
                for(j = 0; j < DIMENSIONS; j++) {
                    p_sum_coordinates_matrix[data_points[i].ID_cluster][j] += data_points[i].coordinates[j];

                }

            }
            //sum the sum coordinate' s contribution by all threads
            #pragma omp critical
            for(i = 0; i < N_CENTROIDS; i++){
                centroids[i].count_points += p_sum_points[i];
                for(j = 0; j < DIMENSIONS; j++){
                    centroids[i].sum_coordinates[j] += p_sum_coordinates_matrix[i][j];
                }
            }

        }

        //recenter centroid based on its points
        #pragma omp parallel for
        for(e = 0; e < K; e++) {
            replaceCentroid(&centroids[e]);
        }

        (*num_iterations)++;
    }

    return 0;
}
```

The while loop, as mentioned, could not be divided into independent iterations, due to the updating of the coordinates of the centroids. So the first pragma was inserted immediately after the while, where the section to be parallelized included the search for the minimum distance and the sum of the coordinates.

When threads are instantiated, the default static scheduling assigns each thread a chunk of the same size. Assigning the chunk there are some problems of sharing the support variables used for the correct execution of the program.

The isChanged variable is shared among all threads since it represents one of the exit criteria and all threads can change it.
Regarding the internal variables of the threads,the private ones are respectively : *i*. and *j*. , the indexes of the cycles, the variables useful to assign the minimum distance (*min distance, current_distance*) and the support variables useful to calculate the sum of the points and the sum of the coordinates (*p_sum_coordinates_matrix, p_sum_points*).
When the sum of the coordinates has to be saved, problems of simultaneous access to the same resource arise so if they're not managed with a **critical pragma** they could lead to the calculation of a wrong result and therefore to a wrong output.
This was the first and only parallelization step shown since the parallelization of the initial coordinate (resetCoordinate), the for loop and the function replaceCentroid have too few iterations to give a substantial speed-up.
The advantage of this implementation is certainly the simplicity of reading and developing the parallelization. On the other hand, an intrinsic drawback in the structure of the algorithm is the fact that every time the while ends a loop, all the threads are destroyed and the next loop is recreated by adding overheads to each iteration that prevent the theoretical speed-up from being reached.

4-Testing and debugging

To debug the parallel code,I decided to rely on some python libraries that implement the algorithm. The results obtained respectively from the parallel version and the python implementation were the following.
10K points , 3 centroids:
python result:

```
[[-439.36734694 -696.75510204 -452.48979592]
 [ 678.69642857   36.39285714  120.83928571]
 [-311.22340426 -678.19148936  334.0212766 ]]
```

gcp result:

```
-438.731903 -670.351318 -465.574615
662.178223 30.636364 120.679825
-287.183258 -660.888672 329.243134
```

200K points , 3 centroids:
python:

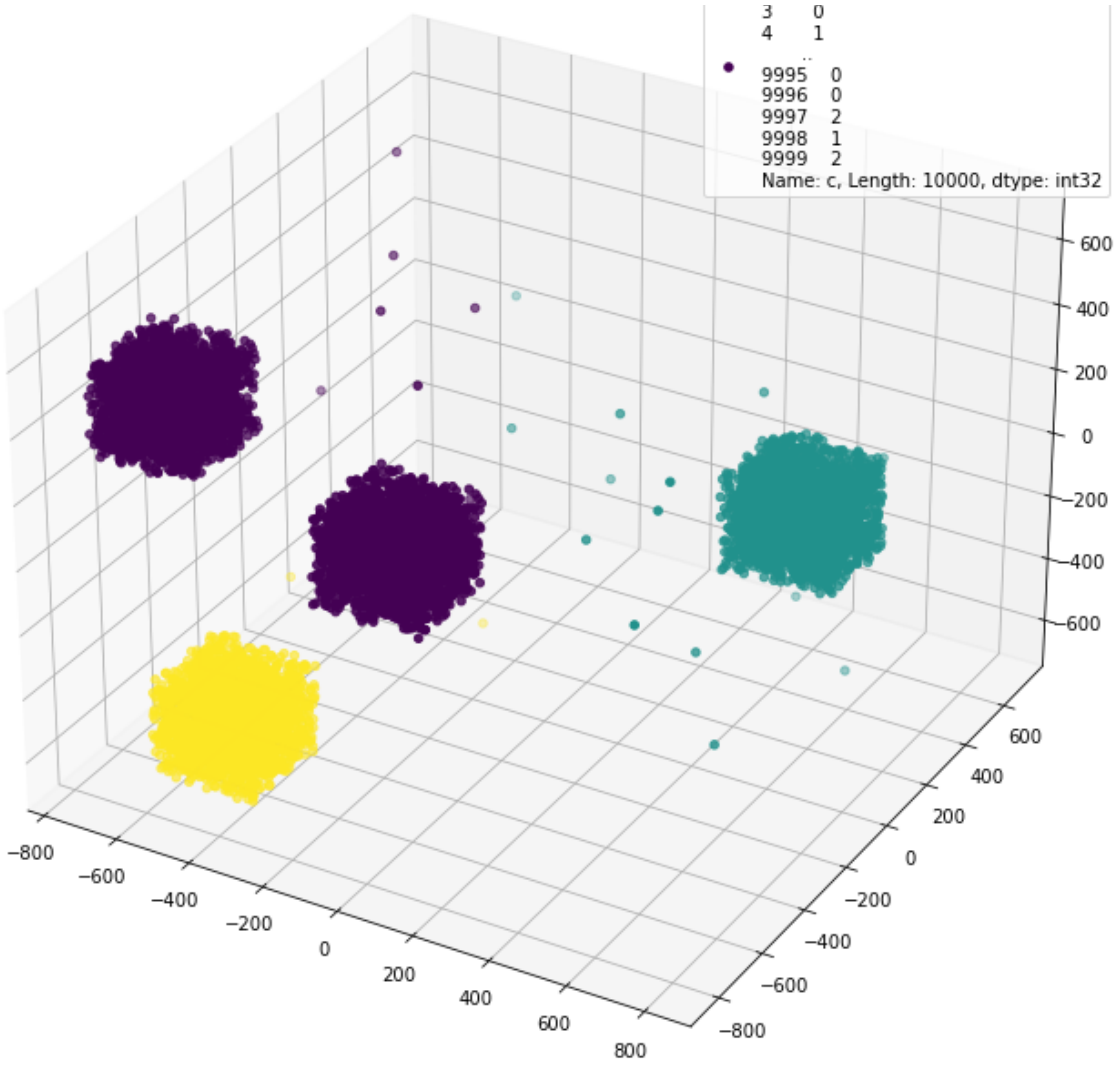
```
[[ 480.10909091 -449.67272727  466.78181818]
[-459.0625     -302.40625      30.28125   ]
[ 510.10416667  644.0625      -52.97916667]]
```

gcp:

```
500.073669 -449.341278 464.025330
-455.669006 -304.008636 27.443785
483.302979 635.789062 -43.864044
```

I noticed that there are some small differences between the coordinates of the centroids, but these differences depend both on the initialization of the centroids, on the stop criteria and

also on the algorithm chosen. However, despite these discrepancies between the two programs, the results are consistent.
In this case, a comparison was made only on the parallel algorithm, since the results of the serial are consistent with the results of the parallel.
Below is a graphical representation of the file clustering with 10K points obtained through a python program.



5-Performance analysis

5.1-Speed-up analysis

The performances were measured on multiple datasets. I decided to show only results on large enough datasets, in order to better appreciate all the advantages deriving from parallelization. For the calculations and performance tests through the google cloud platform, I instantiated e2-custom (24 vCPU, 12 GB RAM). The measurements as the cores changed were carried out up to 24 cores (which are the

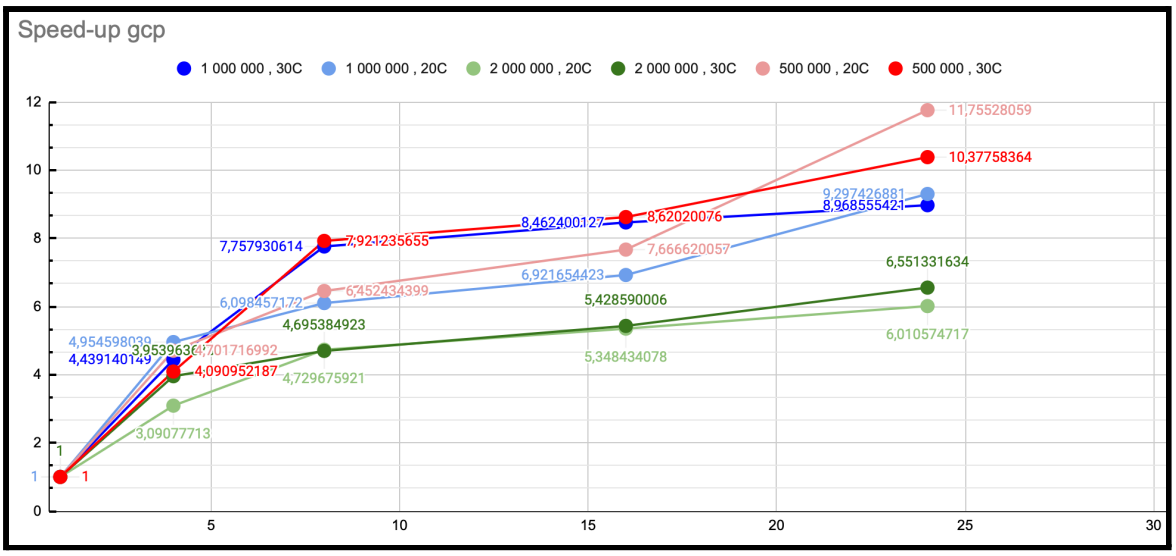
actual ones of my machine) in order to assign 1 thread to each core, avoid the virtualization of some threads and, consequently, reduce possible overhead. Since I chose large datasets as a base,I decided to try various measurements by always increasing the size. In particular through a python program I created 3 datasets, respectively of 500K, 1M and 2M of three-dimensional points. The result was reported in the following table:

GCP 24 core							
n° CENTROIDI	N° ITERAZIONI	Punti	1	4	8	16	24
20	800	500K	387,96411	82,515411	60,126781	50,604322	33,00339
20	800	1M	783,13042	158,061343	128,414515	113,142086	84,230877
20	800	2M	1749,815167	566,140842	369,96513	327,164015	291,122771
30	800	5K	567,181884	138,643	71,602703	65,79683	54,654523
30	800	1M	1142,577889	257,387208	147,278694	135,018183	127,398208
30	800	2M	2537,662082	641,802073	540,458796	467,462468	387,350576

With the relative speed ups:

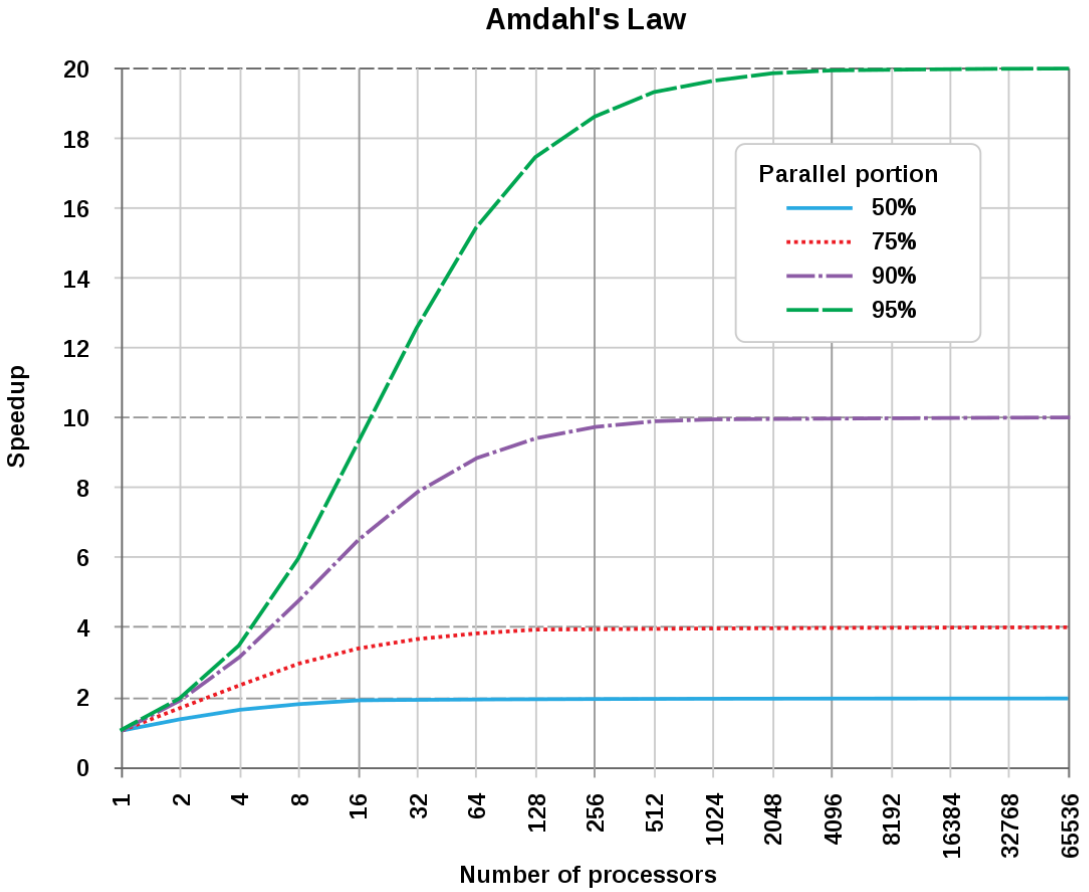
S1	S4	S8	S16	S24
1	4,701716992	6,452434399	7,666620057	11,75528059
1	4,954598039	6,098457172	6,921654423	9,297426881
1	3,09077713	4,729675921	5,348434078	6,010574717
1	4,090952187	7,921235655	8,62020076	10,37758364
1	4,439140149	7,757930614	8,462400127	8,968555421
1	3,953963673	4,695384923	5,428590006	6,551331634

Graphically:



As we can see the serial execution times (with the same centroids) are proportional. So we can see that it is the change in the file size that causes the speed-up to vary more, not the increase in centroids.

All the time measurements, as the number of threads increase, tend to decrease. Looking at the speed-up results we can see that they are in line with the theoretical ones expected. Given the percentage of the parallelized program the result should converge with about 1024 threads if we follow the Amdahl chart.



5.2-Size analysis

As the dataset gets larger, the speed-up decreases, this effect is due to the number of cache misses that occur because of the size of the data.
The datasets used were 3:

- 500K points:6.1 MB
- 1M points:12.8 MB
- 2M points: 24.8 MB

The CPU memory hierarchy is divided as follows:

- private L1d and L1i caches each 32K 8-way
- L2 (256K 8-way)and L3 (46080K 20-way) shared

Assuming empty caches and aligned data I should have respectively (rough estimate):

points	size(MB)	L1 size(MB)	L2 size(MB)	L3 size(MB)	L1 miss	L2 miss	L3 miss
500K	6,1	0,03125	0,25	45	196	25	1
1M	12,8	0,03125	0,25	45	410	52	1
2M	24,8	0,03125	0,25	45	794	100	1

From this table we see that the number of cache misses increases and this entails a considerable extra execution time. If we want to consider the parallel version the size of the dataset should be divided by the number of threads used since each thread is assigned by the *parallel for pragma* a chunk of equal size.