PC-2023/24 Course Project Template

Elia Matteini 7148467
E-mail address
elia.matteini@edu.unifi.it

Filippo Zaccari 7148301
E-mail address
filippo.zaccari@edu.unifi.it

Abstract

This paper presents two different implementations of k-means algorithm using Euclidean distance.
The purpose of this paper is to analyze and compares a C sequential version with two different parallel versions of the algorithm, focusing on the efficiency of parallelization achieved through pragma directives valuating parameters like threads, cores and speedUp. We used pragma OpenMP on CPU and pragma OpenACC on GPU.

1. Introduction

K-Means clustering is an unsupervised learning algorithm, is a method for grouping elements into K clusters. It aims to assign each element to the cluster with the nearest mean distance or centroid. The purpose of the K-Means algorithm is to minimize the sum of distances between the points and their respective cluster's centroid.

The algorithm is made up of two parts:

- Generating points and centroids
- Cluster finding
  - Evaluating the distance
  - Updating clusters based on the mean distance

2. Workflow

The very beginning step of this project was a research about k-means clustering, usage and scope. When we gained sufficient information about the subject, as first approach we developed a sequential version that works on 2D coordinates. Clearly it was almost impossible to check results manually with a large amount of data, the easiest way to see if the algorithm works has been to plot the data into a Cartesian plane with a simple Python script.
K-means algorithm therefore generate a certain number of points and centroids (using casual numbers), execute the k-means clustering and writes points and centroids on a CSV file, on the other side the python script read the CSV file and plot all the data into a Cartesian plane.
Once the sequential k-means algorithm was consolidated we proceeded to develop the parallel version.
Finally, when both the algorithm worked, we've done tests to compare the parallel k-means versions with sequential one.

2.1. Sequential K-means

This section explains how the code works, focusing on the most relevant parts for the parallelization. The sequential version is written in C++, we used the programming construct Array of Structures to represent the system, where a point is represented as an object of the class Point. The initial part of the code is responsible for setting up the environment, randomly generating points and centroids to

allow the second part of the algorithm to work properly.
The second part is the main part of the code. It's composed of a main loop, that iterate until the system reach the equilibrium, and two separate loops nested inside. The first loop calculates the Euclidean distance between points and all clusters, assigning all points to the nearest cluster. This part is implemented as follows:

```
for (auto &point: points) {
    int cluster = -1;
    double minDistance = __DBL_MAX__;
    for (int i = 0; i < centroidsNumber; i++)
        {
        int d = point.evaluateDistance(
            centroids.at(i));
        if (d < minDistance) {
            minDistance = d;
            cluster = i;
        }
    }
    if(cluster != point.getCluster()) {
        point.setCluster(cluster);
    }
    sumXYcount[3 * point.getCluster()] +=
        point.getX();
    sumXYcount[3 * point.getCluster() + 1] +=
        point.getY();
    sumXYcount[3 * point.getCluster() + 2]++;
}
```

The second for loop go through all clusters and calculates the mean distance among all points in the cluster.

```
     for (int i = 0; i < centroidsNumber; i++)
          {
        double averageX = sumXYcount[3 * i] /
            (int) sumXYcount[3 * i + 2];
        double averageY = sumXYcount[3 * i +
            1] / (int) sumXYcount[3 * i + 2];
        if (centroids.at(i).getX() != averageX
            ) {
            centroids.at(i).setX(averageX);
            control = true;
        }
        if (centroids.at(i).getY() != averageY
            ) {
            centroids.at(i).setY(averageY);
            control = true;
        }
```

2.2. Parallel k-means using OpenMP

OpenMP is an API for shared-memory parallel programming, it's a set of compiler directives, library routines, and environment variables, it instruct the compiler on how to parallelize the code. [2]

Before using openMP we optimized the code to make it faster and easier to parallelize. Except for the class Point, all the things from OOP were removed, then we removed the implicit for to allow openMP to parallelize it. The first for loop showed above has been parallelized using the instruction:
`#pragma omp parallel for reduction(+:sumXYCount [:3*centroidsNumber])`
with reduction on the array used to save the sum of the points and avoid race condition.

```
    #pragma omp parallel for reduction(+:
        sumXYCount[:3*centroidsNumber])
    for (int i = 0; i < numPoints; i++) {
        double minDistance = __DBL_MAX__;
        int cluster = -1;
        for (int j = 0; j < centroidsNumber; j
            ++) {
            int d = pow(points[i].x -
                centroids[j].x, 2) + pow(
                points[i].y - centroids[j].y,
                2);
            if (d < minDistance) {
                minDistance = d;
                cluster = j;
            }
        }
        if(cluster != points[i].cluster) {
            points[i].cluster = cluster;
            changed = true;
        }
        sumXYCount[3 * cluster] += points[i].x
            ;
        sumXYCount[3 * cluster + 1] += points[
            i].y;
        sumXYCount[3 * cluster + 2]++;
    }
```

The sum of the points will be done by all threads independently so the reduction is necessary to save the sum of all the points. The operations inside the loop are fully independent from each other, so we can easily parallelize the cycle with the instruction
`#pragma omp parallel for`

```
    #pragma omp parallel for
    for (int j = 0; j < centroidsNumber; j++)
        {
        centroids[j].x = sumXYCount[3 * j] /
            sumXYCount[3 * j + 2];
        centroids[j].y = sumXYCount[3 * j + 1]
             / sumXYCount[3 * j + 2];
    }
}
```

2.3. Parallel k-means using CUDA OpenACC

The OpenACC provides a set of compiler directives (pragmas), library routines and environment variables that can be used to write parallel Fortran, C and C++ programs that run on accelerator devices including GPUs and CPUs. It's similar to OpenMP, it's an implicit accelerated HPC programming framework. [1]

As done before, in this case we adapt the structure of the code to make it friendly to the GPU cache, so we changed the structure of the code from AoS to SoA. To minimize the data transfer between host and device, the array used to calculate the sum of the points has been created inside the GPU, points and centroids have been transferred from host to device using the instruction:

```
#pragma acc data
copyin(points.x[:numPoints],
points.y[:numPoints], cen-
troidsNumber, numPoints,
points.clusters[:numPoints],
centroids.x[:centroidsNumber],
centroids.y[:centroidsNumber],
changed) create(sumXYCount
[:centroidsNumber*3])
```

The parallelization using openACC is similar to openMP, the first loop has been parallelized using the instruction:

```
#pragma acc parallel loop
gang worker vector re-
duction(|:changed) re-
duction(+:sumXYCount
[:3*centroidsNumber])
present(points.x[:numPoints],
points.y[:numPoints], cen-
troids.x[:centroidsNumber],
centroids.y[:centroidsNumber],
points.clusters[:numPoints],
sumXYCount[:3*centroidsNumber],
changed)
```

As was done in openMP, we applied the reduction to the array used to sum the points and reduction to the variable "changed" using the pipe to OR each iteration between them. In this case we use the directive `present` to specify that the data are already loaded on the device and is no needed to copy again from the host.

```
    #pragma acc parallel loop gang worker
        vector reduction(|:changed) reduction
        (+:sumXYCount[:3*centroidsNumber])
        present(points.x[:numPoints], points.y
        [:numPoints], centroids.x[:
        centroidsNumber], centroids.y[:
        centroidsNumber], points.clusters[:
        numPoints], sumXYCount[:3*
        centroidsNumber], changed)
    for (int i = 0; i < numPoints; i++) {
        double minDistance = DBL_MAX;
        int cluster = -1;
        for (int j = 0; j < centroidsNumber; j
            ++) {
            int d = pow(points.x[i] -
                centroids.x[j], 2) + pow(
                points.y[i] - centroids.y[j],
                2);
            if (d < minDistance) {
                minDistance = d;
                cluster = j;
            }
        }
        if (cluster != points.clusters[i]) {
            points.clusters[i] = cluster;
            changed = 1;
        }
        sumXYCount[3 * cluster] += points.x[i
            ];
        sumXYCount[3 * cluster + 1] += points.
            y[i];
        sumXYCount[3 * cluster + 2]++;
#pragma acc update self(changed)
}
```

To synchronize the variable `changed` between device and host, we used the instruction `#pragma acc update self(changed)` to ensure the correct execution of the `if` below.

The operations inside the second loop are

fully independent of each other and easily parallelizable.
To use the array to calculate the new centroid's positions we used the instruction `#pragma acc parallel loop firstprivate(sumXYCount, centroidsNumber)` as follow in the code:

```
if (changed) {
    #pragma acc parallel loop gang worker
        vector firstprivate(sumXYCount)
    for (int j = 0; j < centroidsNumber; j++)
        {
        centroids.x[j] = sumXYCount[3 * j] /
            sumXYCount[3 * j + 2];
        centroids.y[j] = sumXYCount[3 * j + 1]
             / sumXYCount[3 * j + 2];
    }
}
```

To avoid overhead transferring data from device to host we copied only the result of the operations, including clusters and centroids position with the instruction:
`#pragma acc data copyout(points.clusters[0:numPoints], centroids.x[0:centroidsNumber], centroids.y[0:centroidsNumber])`

3. Speedup

To test the programs we used a Python script to execute the sequential versions and compare the results with the parallel version. We used both PC to test openMP, to test openACC we used only the Asus PC due to lack of nvidia video card on the other PC.

3.1. Hardware

Hardware specs of the machines where the code has been tested are:

Laptop System: ASUSTeK
product: TUF Gaming FX505DV
Distro: Ubuntu 22.04.4 LTS
Kernel: 6.5.0-21-generic x86_64
CPU: quad core (8 thread) AMD Ryzen 7 3750H 2.3 Ghz
Memory: 16 GB
GPU: NVIDIA GeForce RTX 2060
Cores: 1920
Memory: 6GB GDDR6
Architecture: Turing

Laptop System: LENOVO ThinkPad X280
Distro: Debian GNU/Linux trixie/sid
Kernel: 6.6.15-amd64
CPU: dual core (4 threads) model: Intel Core i5-7300U 64 bits 2.6 Ghz
Memory: 8 GB

3.2. Measurements

To calculate speedUp and efficiency we used a script python to take measurements of the software executions. The python script execute sequential and parallel k-means varying the number of threads used for each execution. During the executions, the script take measure of the amount of time used by sequential version and parallel version.

To compute speedup and efficiency we used the following formulas:

$$S_p = t_s/t_p$$

$$E_p = S_p/p$$

where S_p represent the speedup calculated and p is the number of cores, t_s is the sequential execution time and t_p the parallel one.
Tests were been executed generating the same points and centroid setting the same seed for random data generation for each test.

3.2.1 OpenMP

In the graphs below it's possible to see how the speedup changes depending on the number of the threads. Each test is identified by a different color depending on the number of centroids.

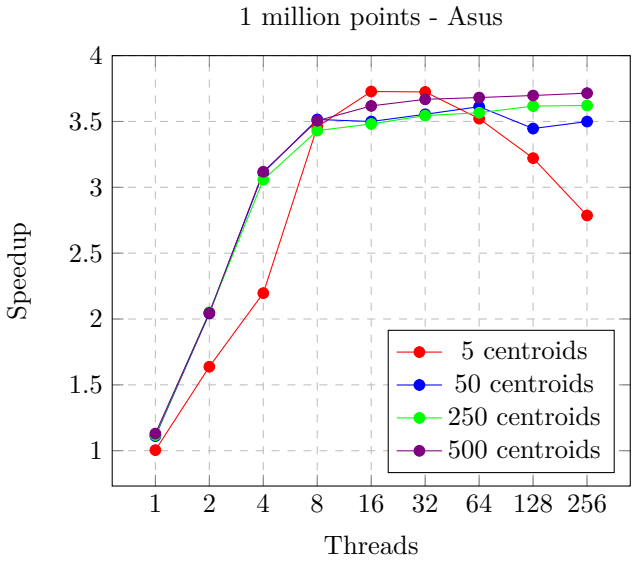


Figure 1: SpeedUp 1 million points ASUS PC

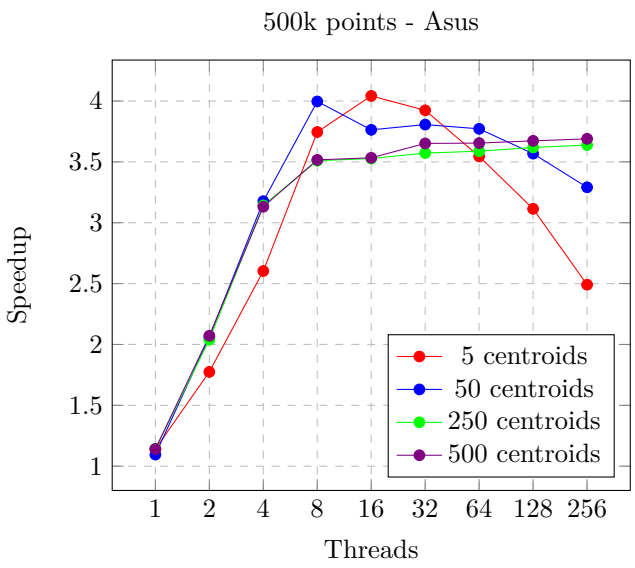


Figure 2: SpeedUp 500k points ASUS PC

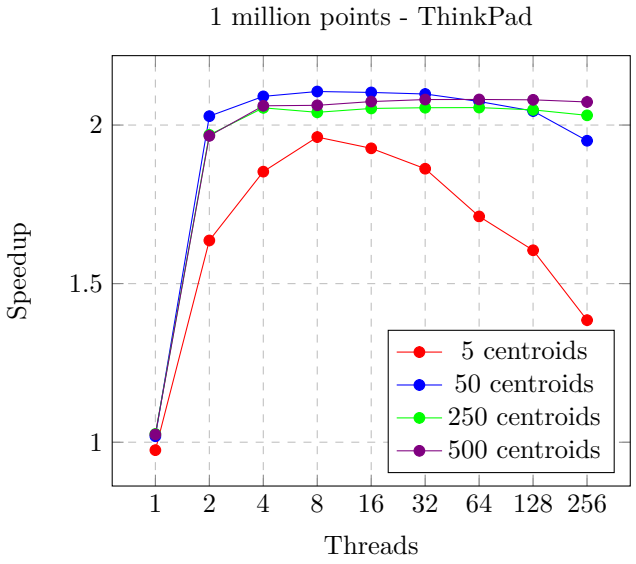


Figure 3: SpeedUp 1 million points ThinkPad PC

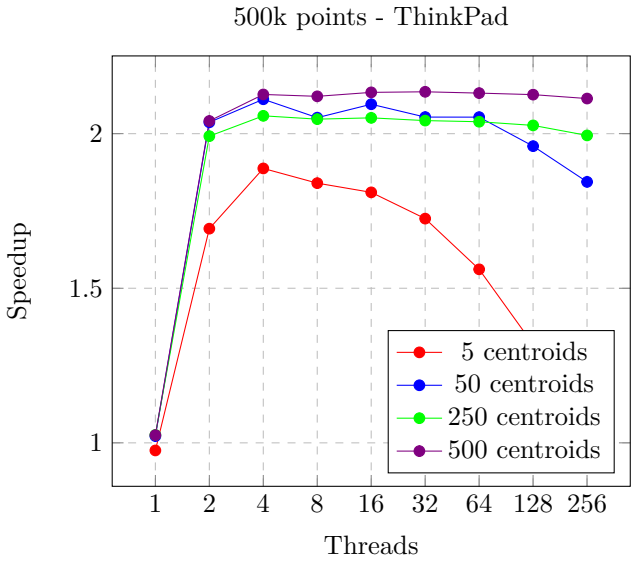
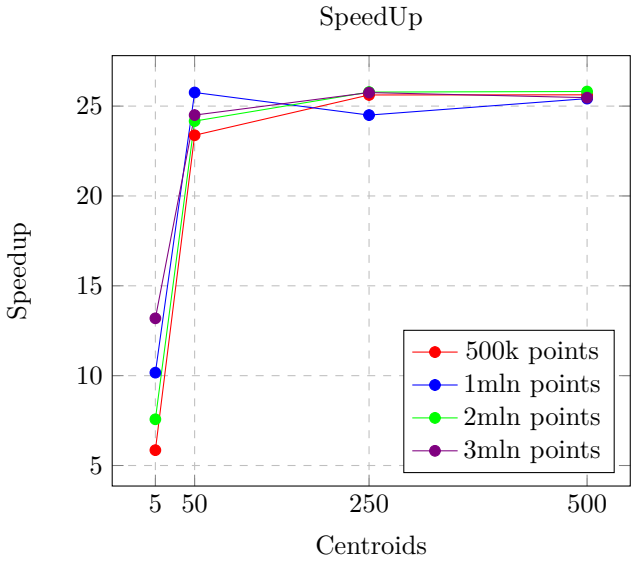


Figure 4: SpeedUp 500k points ThinkPad PC

As we can see in the graphs, the best speedup is reached when the number of threads used by the program is equal to the number of logical threads available on the PC. When the number of threads increase above the number of logical threads available on the machine the speedup starts to decrease. With 5 centroids we can see that the speedup decrease faster than the other cases, this happens because of the overhead to manage threads in front of a few centroids.

3.2.2 OpenACC

In the graph below we can see the results obtained executing the program on GPU using openACC pragmas.



4. Conclusion

As we can see in the graphs, with openMP there is a speedup that reach the number of cores. In addition there is a relation between speedup and centroids number, with a low number of centroids we have a lower speedup, on the other hand, with a higher number of centroids we have a higher speedup.
Due to the number of cores, the speed achieved on the GPU is higher compared to the CPU. The considerations about the number of centroids are the same for the CPU.

References

[1] OpenACC. Openacc organization. URL: `https://www.openacc.org/`.

[2] OpenMP. Openmp api. URL: `https://www.openmp.org/`.