Parallel K-means

Luigi Cennini
E-mail address
luigi.cennini@edu.unifi.it

Abstract

This project aims to implement and compare sequential and parallel versions of the K-means algorithm using both OpenMP and CUDA. The objective is to analyze performance and measure the speedup achieved by the parallel versions compared to the sequential one.

1. System Specification

CPU: Intel i3-4005U (4 cores) 1.600GHz
GPU: NVIDIA GeForce 920M

2. Introduction to K-means

K-means clustering is an unsupervised machine learning algorithm used for partitioning a dataset into a predetermined number of clusters. The objective of K-means is to group similar data points together and discover underlying patterns or structures within the data.

At its core, K-means operates by iteratively assigning each data point to the nearest cluster centroid and then recalculating the centroids based on the mean of the data points assigned to each cluster. This process continues until the centroids no longer change significantly, or a specified number of iterations are reached.

One of the key parameters in K-means is 'k', which represents the number of clusters the algorithm should identify.

One iteration of the algorithms works in this way

1. for every point in the dataset, the point is assigned to the nearest centroid

2. centroids are updated and convergence is checked

The algorithm stops when the maximum number of iterations is reached or convergence is reached. Convergence is reached if two consecutive iteration of the algorithm don't change the centroid position (given a certain tolerance).

3. Sequential implementation

The K-Means algorithm was implemented using an array-of-structure approach.

In my implementation, the public method:

```
void fit(std::vector<Point>& dataPoints,int
    maxIteration);
```

takes a vector of data points and the maximum iteration number. For testing purpose data points are generated by a python script ($create\_dataset.py$) which generates a csv file that is imported as a $Point$ vector in c++ using the function $loadCsv$ (defined in $utils.cpp$).

At each iteration of the algorithm the centroids vector is updated, following the scheme presented in section 2.

An example of the algorithm's output is presented in Figure 1.

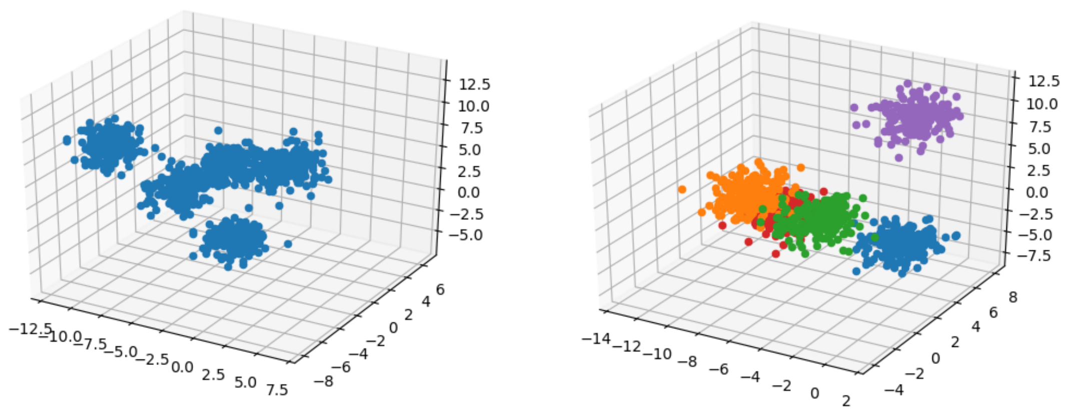


Figure 1. on the left a 1000 points dataset with 3 dimension, on the right the result of K-means with K=5

3.1. centroids initialization

The fit function of the K-means class requires the vector centroids to be initialized beforehand. Centroids can be initialized using various methods, such as the K-means++ algorithm. For the purposes of testing, I opted to initialize k centroids by randomly selecting points from the dataset. These centroids are generated alongside the dataset using the Python script $create\_dataset.py$, which produces a CSV file.

4. parallel implementation with openMP

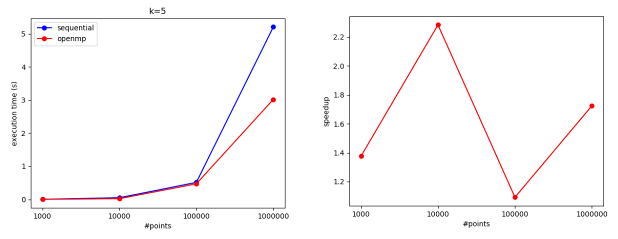
It is evident that the initial step of the algorithm outlined in section 2 consumes more time compared to the subsequent step (if $|dataPoints| >> k$).The assignment of each point to its respective cluster presents an embarrassingly parallel problem. The challenge in parallelizing this process lies in updating the clusters with the mean of the points, which introduces synchronization issues. To address this, I implemented a reduction strategy. In my approach, each thread maintains a private vector of centroids ($newCentroidsTmp$), allowing for local operations. These operations are then consolidated into global memory at the conclusion of the process. Additionally, atomics are utilized to mitigate issues of data race.

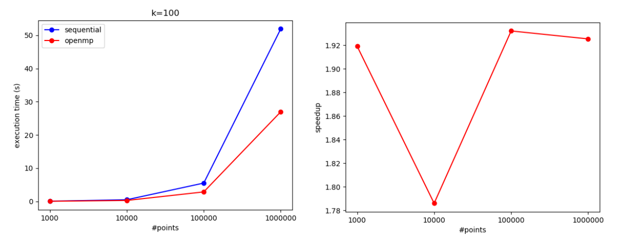
Since we need to perform reduction operations on an entire array of values, we need to manually implement a reduction operation. The following outlines the procedure for doing so:

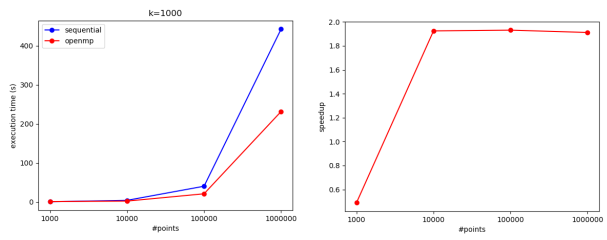
```
#pragma omp parallel
{
    newCentroidsTmp = []
#pragma omp for schedule(static)
    for (point p in dataPoints){
        //assign cluster to p
        //update newCentroidsTmp
    }
    //perform global newCentroids update
}
```

5. speedup analysis

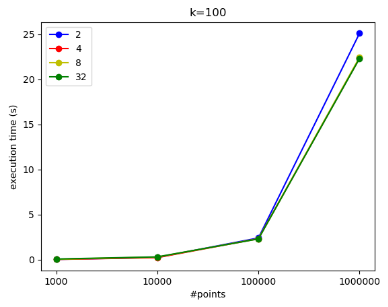
For the speedup analysis sequential and parallel execution times are compared and no stop condition is set. Dataset are generated with increasing number of points and same centroids are used for both the sequential and parallel run. Here we can see the time and speedup for $K = 5, 100, 1000$ clusters.







We can see that the maximum speedup we get is about 2.3. We can also test how varying the number of threads per block affects the execution time.



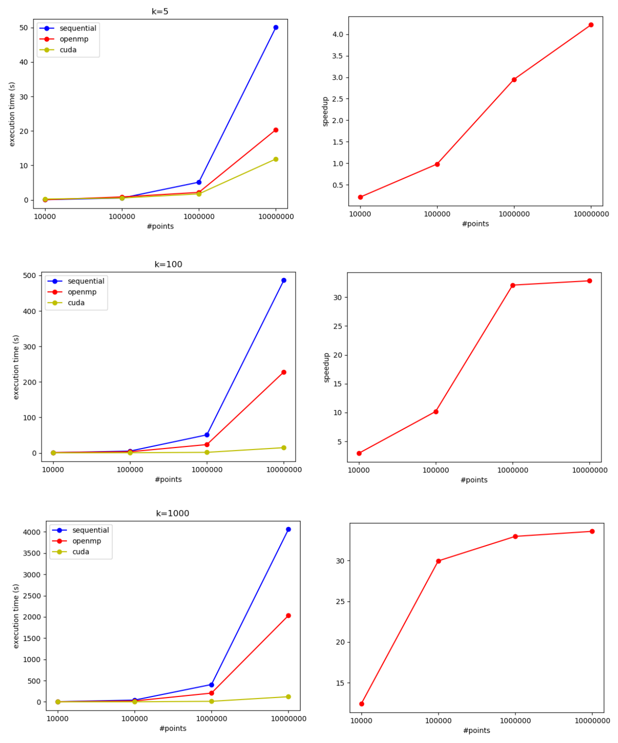
In this case, with a number k=100 of centroids, we can observe that having more than 4 threads doesn't improve performances.

6. CUDA

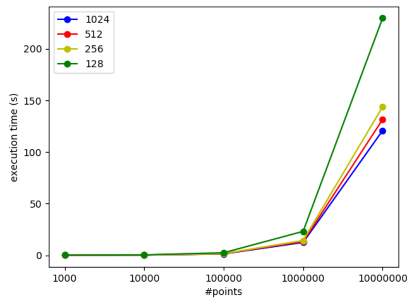
The CUDA implementation of the k-means algorithm follows a similar approach to the OpenMP version, utilizing reduction to minimize the number of atomic operations conducted in global memory. I implemented reduction within the $centroidAssignAndUpdate$ kernel. Initially, this kernel generates shared centroids (stored in the $newCentroids\_shared$ vector) and proceeds to assign points to their nearest centroids. As this assignment process is embarrassingly parallel, synchronization is unnecessary, with each GPU thread handling a distinct point independently. Following the assignment of points, their corresponding cluster labels are stored in the $clusterLabel\_device$ vector. Subsequently, the shared centroids needs to be updated based on the collected information. Atomic operations are employed to prevent data races during this process. Upon completion of this section of code execution by all threads within the block, the global centroid coordinates are updated. Once again, atomic operations are used to safeguard against data races.

7. Speedup analysis

Here speedup for CUDA version is shown using datasets up to 10^7 points and $k = 5, 100, 1000$ centroids.



We can see that the maximum speedup for the CUDA implementation is more than 30. We can also test how varying the number of threads per block affects the execution time.



In this case, with a number k=1000 of centroids, we can observe that the shortest execution time, as the number of points in the dataset increases, is achieved with 1024 threads per block.