K-Means – Parallel Computing

Armand Palla - 7042287

Abstract

I have chosen this topic because we have studied the K-Means algorithm in the course “Multivariate Analysis and Statistical Learning” and it is a known concept for me. This mid-term paper focuses on the study and implementation of the k-means algorithm both in sequential and parallel version. The implementation was done in C++ and for the parallel part I have used OpenMP. K-means is a clustering algorithm composed by n-dimensional points and an integer K, where K is the number of desired clusters. K-Means is a clustering algorithm which has the task of dividing the points of space into K groups on the characteristics of the points themselves. The ultimate goal is to have the points that belong to the same class on the same cluster (they must therefore be similar and have a small distance between them), while those that are on different clusters must be of different classes (therefore they must not be similar and have a relatively large distance). For simplicity, points generated in this report are randomly chosen in 2D space and also to avoid race condition, the process when two or more threads can access shared data and they try to change it at the same time.

1. Introduction

In this project I want to describe better the difference between a sequential program and a parallel program. Specifically, the aim of this report is to verify these theoretical aspects by comparing the execution time of the K-means algorithm with different datasets sizes when using only one core of computation and when using multiple ones. By adopting different computing techniques, we make use of more than one single thread for the execution and computation of the algorithm.

2.How the algorithm works

Clustering is an unsupervised machine learning technique, with several valuable applications in different fields. Among all the unsupervised learning algorithms, clustering via k-means might be one of the simplest and most widely used algorithms. Briefly speaking, k-means clustering aims to find the set of k clusters such that every data point is assigned to the closest centroid, and the sum of the distances of all such assignments is minimized. The main idea is to define K centers, one for each cluster. These centers should be placed in a cunning way because different locations cause different results. So, the better choice is to place them as much as possible far away from each other.

Algorithmic steps for k-means clustering

Let P = $\{p_1,p_2,p_3,\ldots\ldots,p_n\}$ be the set of data points and C = $\{c_1,c_2,\ldots\ldots,c_k\}$ be the set of centers.

1) Randomly select *'k'* cluster centers.

2) Calculate the distance between each data point and cluster centers.

Here we use the **Euclidean** method to find the distance between data points and cluster centroids but there are also some other methods which we can apply to, such as: Karl Pearson distance, Minkowsky distance, Manhattan distance etc. We have used the **Euclidean** method for simplicity.

$$dist = \sqrt{(x_{p1} - x_{p2})^2 + (y_{p1} - y_{p2})^2}$$

3) Assign the data point to the cluster center whose distance from the cluster center is the minimum of all the cluster centers.

4) Recalculate the new cluster centers in order to take the new coordinates of the centroid such as:

$$x_{c_i} = \frac{\sum_{i=1}^{k} x_{p_i}}{k} \quad \text{and} \quad y_{c_i} = \frac{\sum_{i=1}^{k} y_{p_i}}{k}$$

5) Recalculate the distance between each data point and new obtained cluster centers.

6) If no data point was reassigned then stop, otherwise repeat from step 3).

3.Technology

I have used the C++17 to implement the K-Means algorithm. This is a high-level language and offers very large support for Object-Oriented Programming. Since KMeans is a strongly parallelizable algorithm, I adopted OpenMP as a library for the parallel part. This is a library that offers a lot of possibility to modify directly the sequential version of the algorithm in order to make the code run on multiple threads instead of a single one. This is achieved by the use of some specific OpenMP compiler directives, such as **"#pragma omp parallel"** which acts with the same principle of a forkjoin thread creation and thus allowing the programmer to make use of multithreading. OpenMP consists of a set of "compiler #pragmas" that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism. The processor that I have used to run the algorithm is a Intel® Core™ i5-5200 CPU and has a base clock of 2.20GHz and a boost clock of 2.19 GHz.

4.Sequential Algorithm

I have initialized the algorithm using a class called Datapoint with 3 private variables as below:

```
private:
    vector<float> a;
    vector<float> b;
    vector<int> group;
```

The idea of these rows is to represent **a**, **b** and **group** as a structure with 3 Arrays. Vector **a** and **b** will keep the position of the points and in the other hand **group** will be the cluster in which these points are included. The **groups** are implemented by using a struct in the same way as Datapoint adding also three others support arrays such as:

```
float a_accumulator[centers_nr];
float b_accumulator[centers_nr];
int point_nr[centers_nr];
```

The first two arrays serve as accumulator variables in order to take the value or the coordinates of the points after being in a specific cluster."centers_nr" is a variable that specifies how many centroids the program will run with or to understand better is the number of **K.** It is a value given by the user at the initial of program. This is the weak point of this algorithm because we do not know exactly how much will be the optimal K. There are some direct methods and statistical testing methods to calculate the optimal K, but in our example “centers_nr” is a value initialized by us. The last array holds the number of points contained in a cluster. I have created the datapoint set depending on **Points** variable and at the initial I have initialised it with value 5000. It takes random floating point values between 0 and 5000 ,then I choose again “centers_nr” in a randomly way using their coordinates as initial center(centroid) positions.

5.Parallel algorithm

The parallel implementation shares many parts with the sequential version. One major issue that can occur while computing using multiple threads that run at the same time is the possibility for race conditions. Race conditions occur when two different sources try to write to a specific memory location at the same time. In this specific context of the K-means algorithm, race conditions can occur when the dataset is processed by multiple threads at a time, and two different threads try to add two different points to the same cluster. The process of addition a specific point to a cluster is achieved by increasing the two accumulator values with the coordinates of the point and then by increasing the specific **point_nr** value by 1. **The main problem** resides in the fact that the struct that holds all the information about the clusters is shared among all threads and this means that there is no predictable result when two threads try to add two different points to the same cluster. In order to solve this problem I created a thread-local variable which is a copy of the shared struct that holds all the informations about the clusters, as well as two thread-local variables that hold information about the closest centroid from the currently processed point. Every thread-local variable will be updated by each thread indipendently. This assures that every thread has his local copy of a struct that holds all the accumulators with the coordinates of the points that are in that specific cluster.
When a specific thread has finished to process the dataset points, he will then proceed to update the cluster struct that is shared among all threads within a critical section, which is a portion of the code that assures that only one thread runs the code within the critical section at a time. This gives us unpredictable results when processing the update to the centroids coordinates. The performance is not heavly impacted by the critical section because the amount of time that each thread stays in that specific portion of code is significantly short than the amount of time that each thread spends by processing each point, since **Points** is a way bigger value than **centers_nr.**

6.The performance

Computers are constructed out of various components, one of them being the CPU which stands for the Central Processing Unit. The piece of hardware is also referred to as a processor, core processor, or a microprocessor [7], and is essential for a computer's functionality. Its task is to receive hardware and software instructions and process them by performing calculations or operations and ensures to satisfy their request. CPUs are fabricated with ingrained cores and the amount can vary from one single core to multiple cores. A core is a processing unit that resides within a CPU and the number of cores decides how many logical processors a CPU has. CPUs with two cores are called dual-cores as in my case and have thereof two logical processors. Each core is independent of one another and can perform calculations and operations individually and simultaneously, but there exists situations where some work together using a specified shared set of data. Cores are seen as the "brains" [8] of a CPU, working by handling the multiple received instructions. The software instructions which the CPU operates, originates from processes. A process is an application program under execution by the CPU and it can vary in complexity depending on which program is being executed momentarily. In a multi-processing environment, multiple processes can be active at the same time, thus, increasing the number of instructions to the CPU.

The results were carried out by comparing the times completion of the two versions by varying the number of cluster and the number of points. Recall that the machine with which it is tested the program is a 2 core machine.

The next step will be to compare the sequential and the parallel version measurements in order to analyze the time difference between the two implementations, by using the Speedup metric given by

S(latency) = TS/TP where TS and TP represent the computation time of the sequential and the parallel algorithm.

	Sequential version [ms]			**Parallel version [ms] (4 threads)**		
Computation	1000 Points	10000 Points	100000 Points	1000 Points	10000 Points	100000 Points
1	218	2388	16686	19	227	2283
2	279	2732	16282	16	184	2403
3	183	2751	18034	16	173	2418
4	153	2295	20877	16	175	2574
5	208	2403	17052	17	276	2549
Mean	**208.2**	**2513.8**	**17786.2**	**16.8**	**207**	**2445.4**

In order to evaluate the performance gained by adopting parallel programming techniques, I took 5 time computation of 3 different dataset sizes, both for the sequential and the parallel version of the algorithm. Dataset sizes are: 1000, 10000, 100000 and the number of threads are 4.

I am presenting some results regarding the speedup obtained when using 16 threads:

S(latency) =TS/TP=208.2/16.8= **12.4ms**

S(latency) =TS/TP=2513.8/207= **12.1ms**

S(latency) =TS/TP=17786.2/2445.4= **7.27ms**

7.Conclusions

The main reason that we do parallel programming is to execute code efficiently, since parallel programming saves time and allows the execution of applications in a shorter time. Parallel programming goes beyond the limits imposed by sequential computing, which is often constrained by physical and practical factors that limit the ability to construct faster sequential computers. The solutions to certain problems are represented more naturally as a collection of simultaneously executing tasks. This means that parallel programming techniques can save the software developer work in some situations by allowing the developer to directly implement data structures and algorithms developed by researchers.

In our project the main intention is to use more data at the same time using parallel computing.

8.References

[1] https://it.wikipedia.org/wiki/K-means
[2] https://www.youtube.com/watch?v=_aWzGGNrcic
[3] https://it.wikipedia.org/wiki/OpenMP
[4] https://www.youtube.com/watch?v=_1QNzaWPYOE
[5] https://healthcare.ai/step-step-k-means-clustering/
[6] https://github.com