PC-2019/20 K-means

Matteo Amatori, 7024736
University of Florence
matteo.amatori@stud.unifi.it

Abstract

For the first project in the Parallel Computing Course I decided to implement a sequential and a multithreaded version of the K-means algorithm. K-means is a clustering algorithm that, given a dataset composed by n-dimensional points and an integer K, where K is the number of desired clusters, returns a partition of the initial dataset into K partition of points.

1. Introduction

This project aims to verify the theorical aspects of parallelisation. By adopting computing techniques that make use of more than one single thread for the computation of an algorithm, the time required by a program to run is inversely proportional to the number of processing units used. In this specific project the aim is to verify these theorical 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.

2. Theorical analysis of the algorithm

Let's define a dataset D as a set of points such as

$$D = (p_1, ..., p_n)$$

where $p_1, ..., p_n$ are n-dimensional points. Let's also define a set of centroids C as a set of points that will identify the centers of the clusters

$$C = (c_1, ..., c_n)$$

K-means as an algorithm works by looping on all the points of the dataset and, for each point, looking for the closest centroid. Even though this algorithm can be used to calculate a dataset partition for n-dimensional datasets, this specific implementation uses a 2-dimensional dataset. The computation of the Euclidean distance in two dimensions is achieved by the following formula

$$dist = \sqrt{(x_{p_1} - x_{p_2})^2 + (y_{p_1} - y_{p_2})^2}$$

As the algorithm processes every point, they are assigned to the cluster represented by the closest centroid. After all points in the dataset have been analysed and assigned to the cluster represented by their closest centroid, every cluster's position is updated by computing the average of the coordinates of the points that have populated that specific cluster. Formally, given a cluster represented by a centroid

$$c_i = (x_{c_i}, y_{c_i})$$

the new coordinates of the centroid are given by

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

The dataset partition that the algorithm returns are identified by

$$S = (S_1, ..., S_j)$$

and split the dataset in such a way to minimize the within-cluster sum of squares given by

$$\arg\min_{S} \sum_{i=1}^{j} \sum_{x \in S_i} \|x - \mu_i\|^2 \qquad (1)$$

2.1. Metrics used for the analysis

In order to evaluate the performance gained by adopting parallel programming techniques, I took 10 time measurement of 3 different dataset sizes, both for the sequential and the parallel version of the algorithm. Such a number of repetitions for every possible combination of dataset size and technology used will avoid possible fluctuations in time measurements. Then, I will take an average of the time measurements of all the categories. 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

$$S_{latency} = \frac{T_S}{T_P} \tag{2}$$

where T_S and T_P represent the computation time of the sequential and the parallel algorithm.

3. Technologies used

In order to implement the K-means algorithm I choose to use C++14. This language is very versatile, as it offers Object-Oriented Programming support, while having low-level programming languages performance. Since K-means is a strongly parallelizable algorithm, I adopted OpenMP as a library for multithreaded computing. This library offers the possibility to be able to directly modify 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 fork-join thread creation and thus allowing the programmer to make use of multithreading. Regarding the CPU used, the system that I used to run the benchmarks has a Ryzen 7 3700X, with 8 physical cores with virtualization, for a total of 16 computing units. It has a base clock of 3.6GHz and a boost clock up to 4.4GHz.

4. Sequential algorithm implementation

The sequential implementation of the algorithm uses a Dataset class, which contains 3 private variables

```
vector<float> x
vector<float> y
vector<int> cluster
```

Those variables contain the dataset in the form of a Structure of Array with 3 arrays, one for the x coordinates, one for the y coordinates and one for the cluster that contains the specific point. The clusters are implemented by using a struct, with a similar shape of the Dataset class (that means, two vectors for the coordinates), but with 3 support arrays

```
float x_accumulator[NUM_CENTR]
float y_accumulator[NUM_CENTR]
int npoints[NUM_CENTR]
```

where `NUM_CENTR` is a variable that specifies how many centroids the program will run with. The first two arrays serve as accumulators for the coordinates of the points that end up being in a specific cluster, while the last one holds the number of points contained in a cluster. This was done with the aim of having an easy way to update the cluster position after every point has been assigned to the closest centroid.

Regarding the dataset, I choose to create `NUM_POINTS` random floating point values between 0 and 1000, then I randomly extracted `NUM_CENTR` points and used their coordinates as initial centroid positions.

5. Parallel algorithm implementation

As previously mentioned, the parallel implementation shares many parts with the sequential version. One major issue that can occurr while computing using multiple threads that run at the same time, is the possibility for race conditions.

Race conditions occurr 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 occurr

	Sequential version (CPU - 1 thread)			Parallel version (CPU - OpenMP 16 threads)		
Measurements	100K Pts. [ms]	500K Pts. [ms]	2M Pts. [ms]	100K Pts. [ms]	500K Pts. [ms]	2M Pts. [ms]
1	111	553	2212	14	46	250
2	244	658	2211	12	43	153
3	111	600	2256	14	55	147
4	132	606	2230	10	44	158
5	279	607	2211	14	47	162
6	114	605	2219	14	40	155
7	243	692	2198	8	40	164
8	227	605	2227	13	49	146
9	229	592	2209	14	51	151
10	175	680	2204	9	48	154
Mean	186.5	619.8	2217.7	12.2	46.3	164
Std. dev.	65.2	43.1	16.6	2.4	4.8	30.8

Table 1. K-means performance analysis

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. Based on the implementation of the centroids explained in the previous chapter, the process of addition of 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 npoints 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. This means that there is no predictable outcome 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 prevents 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 `NUM_POINTS` is a way bigger value than `NUM_CENTR`.

6. Analysis of performance

The tests have been conducted with the aim of evaluating the performance gain derived from using a multi-threaded approach compared to a single-threaded one. The metric that is used to evaluate the gain is the Speedup, which is shown in formula (2). The code has been set to run, both in sequential and parallel mode, with 3 different dataset sizes: 100000, 500000 and 2000000 points.

In addition to this, The code has been set to run, when in parallel mode, with 2, 4, 8, and 16 threads, in order to appreciate the different speedups that could derive from using more threads to compute the data. Table 1 shows the results achieved when the algorithm was running on a single CPU thread against the same algorithm running with the parallel implementation with OpenMP on 16 threads. Using the formula (2) it is easy to get the mean values of the measurements taken and obtain the Speedup value.

	Dataset dimension		
Nr. Threads used	100K Pts.	500K Pts.	2M Pts.
1	1.00	1.00	1.00
2	3.28	2.24	1.99
4	5.50	3.27	2.83
8	13.23	9.16	8.21
16	15.29	13.39	13.52

Table 2. Speedup analysis

Here I present some results regarding the speedup obtained when using 16 threads

$$S_{latency} = \frac{T_S}{T_P} = \frac{186.5ms}{12.2ms} = 15.29$$

$$S_{latency} = \frac{T_S}{T_P} = \frac{619.8ms}{46.3ms} = 13.39$$

$$S_{latency} = \frac{T_S}{T_P} = \frac{2217.7ms}{164ms} = 13.52$$

By analysing the speedup values presented in Table 2, it can be deduced that, when the size of the dataset is not big, the speedup tends to maintain a curve that stays above the linear speedup ratio. The other two curves, as shown in Figure 1, tend to remain lower than the first one. Another interesting topic about this graph is that the points that correspond to 8 thread used are far higher than the ones that correspond to 4 threads used. They tend to remain well above the trendline, and all of them are above the linear speedup ratio.

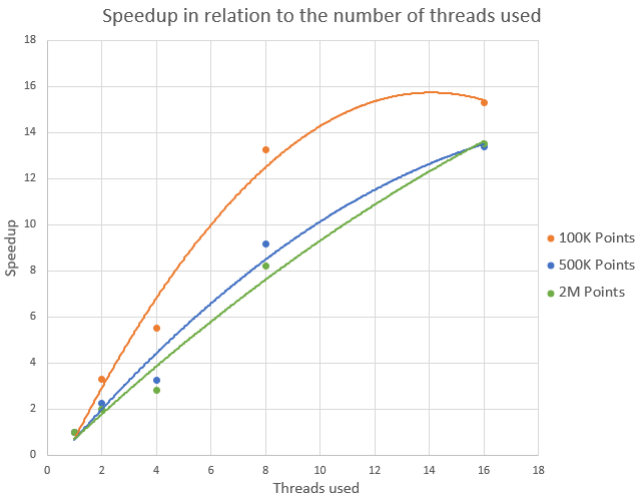

Figure 1. Speedup graph

7. Conclusions

With this project has been possible to prove the theorical aspects of the parallel programming paradigm that allows a machine to compute more data in the same amount of time.