Parallel implementation of K-means clustering algorithm

Stefano Vannucchi
Univeristà degli studi di Firenze
stefano.vannucchi@stud.unifi.it

Abstract

Central Processing Units (CPUs) have received multiple upgrades over time, since they were created. Though performance improvements for a single processor may be reaching a limit, the prophecy of Moore's Law continues to deliver improved transistor densities.

So the chip manufacturers have used this opportunity to place more than one core on a single chip, increasing CPU's capacity to execute more than one instruction at the same time.

Though this new technology is very scalable (thanks to the continuous miniaturization and the increase of the cores), it need a new programming approach.

In this article we will see two implementations, one sequential and one parallel, for solving a common data mining problem.

1. Introduction

Clustering is one of the most common data analysis technique used to get an intuition about the structure of the data. It can be defined as the task of identifying subgroups in the data such that data points in the same subgroup (or cluster) are very similar while data points in different clusters are very different.

Particularly we discuss about k-means algorithm. It is an iterative and unsupervised algorithm that tries to partition the data into K pre defined subgroups. It assign data points to a cluster such that the distance between the data points and the cluster's centroid is at the minimum. There is many different techniques to calculate the cluster's centroid; in this article will be used the mean value.

1.1. Main steps

Initialization: before start the algorithm's operations we need to initialize each cluster with random centroids values (or picking random data points from whole dataset).

Cluster assignment: we need to loop through the elements and assign each of them to the nearest cluster.

Update the centroid's value: once all the elements have been assigned we have to recalculate all the centroids values.

Definitely we have to loop through the last two steps as long as there are changes in the value of the centroids.

Due to the hard computational complexity (as we can see in the previous steps) could be a great idea to parallelize the algorithm execution. In this fashion we can exploit betters the processing skills of the modern CPUs.

The rest of the paper is organised as follows: first we shortly present the sequential version of the algorithm in the next section. In section 3 we discuss about parallelization strategies in relation to the particular programming language and library used. Finally we will examine the obtained results comparing the two versions in terms of execution time costs.

2. Sequential implementation using C++

The sequential version is written with the C++ programming language, and after a brief initialization (where data points and clusters are chosen randomly) is composed by a main *while* loop that iterate as long as there are changes (identified by a local variable named "*hasChanged*").

Below are reported some code snippets that describe the data structures used for represent clusters and single elements.

```
// Cluster data structure
class Cluster {
public:
   double centroid_x;
   double centroid_y;
   int index;
   double newCentroid_x;
   double newCentroid_y;
   int elementNo;

   explicit Cluster(int index, int x, int
       y) {
      this->index = index;
      this->centroid_x = x;
      this->centroid_y = y;
      this->newCentroid_x = 0;
      this->newCentroid_y = 0;
      this->elementNo = 0;
   }
};
```

```
// Single Element data structure
class Element{
public:
   int x;
   int y;
   int clusterId;

   explicit Element(int x, int y) {
      this->x = x;
      this->y = y;
      clusterId = 0;
   }
};
```

This is a very basic algorithm version. The advantage is its simplicity but it can not exploit the real power of all modern CPU's cores.

3. Parallel implementation using OpenMP

OpenMP is an implicit threading framework for C/C++ and Fortran which allows to simplify concurrent programming languages hiding parallelization details and easily convert a serial program into a parallel one, without restructuring the entire code.

OpenMP is also an API for shared-memory parallel programming (in fact "MP" stands for multiprocessing) and it is designed for system in which each thread or process can potentially have access to all available memory.

The OpenMP implicit threading follows the fork-join model: the programmer indicates the regions of code that can be executed in parallel and when the compiled program founds those regions the main program continues as "main thread" and multiple threads are forked, executing the parallel regions. At the end the main thread waits for other threads (like an explicit join).

OpenMP framework is perfect for problem that can be defined *embarrassingly parallel*, that is problems with few variables contention.

In this article we try to reuse the sequential code version and parallelize the main core of the program using OpenMP directives.

Below we can see some code snippets that describe the idea used for this implementation.

```
#pragma omp for
  for (int i = 0; i < elementsNo; i++) {
    int cluster_id = clusters[0].index;
    double min_distance =
        getDistance(elements[i],
        clusters[0]);
    for (int j = 1; j < clustersNo; j++) {
      double distance =
          getDistance(elements[i],
          clusters[j]);
      if (distance < min_distance) {
        min_distance = distance;
        cluster_id = clusters[j].index;
      }
    }
    elements[i].clusterId = cluster_id;
  }
```

Above is reported the first step parallelized using a *pragma omp for* directive. This mean that each step of the external loop are executed in parallel. Therefore to each thread is assigned a data point and it has to compare its distance with every cluster's centroid. Before finishing the singole iteration step, the thread assign the data point to the nearest cluster.

```
#pragma omp for
  for (int i = 0; i < clustersNo; i++) {
    clusters[i].newCentroid_x = 0;
    clusters[i].newCentroid_y = 0;
    clusters[i].elementNo = 0;
    for (int j = 0; j < elementsNo; j++) {
      if (elements[j].clusterId ==
          clusters[i].index) {
        clusters[i].newCentroid_x +=
            elements[j].x;
        clusters[i].newCentroid_y +=
            elements[j].y;
        clusters[i].elementNo ++;
      }
    }
    if (clusters[i].elementNo > 0) {
      clusters[i].newCentroid_x =
          clusters[i].newCentroid_x /
          clusters[i].elementNo;
      clusters[i].newCentroid_y =
          clusters[i].newCentroid_y /
          clusters[i].elementNo;
    }
    if (clusters[i].centroid_x !=
        clusters[i].newCentroid_x ||
        clusters[i].centroid_y !=
        clusters[i].newCentroid_y) {
      clusters[i].centroid_x =
          clusters[i].newCentroid_x;
      clusters[i].centroid_y =
          clusters[i].newCentroid_y;
      #pragma omp critical
      {
        hasChanged = true;
      };
    }
  }
```

In this step is performed the computation of the cluster's centroid. We have another *pragma omp for* directive that assign to every thread the calculation for a single cluster.

We can also notice the presence of a *pragma omp critical* directive, that means only one thread a time can enter and modify the value of the shared *hasChanged* variable. (As we have explained before, that variable is used to determine the end of the main loop)

4. Results and comparison between sequential and parallel versions

The parallelization of an algorithm is a great technique to better exploit the power of a modern computer; but first we have to well study the problem and the techniques to apply in order to achieve an effective improvement in terms of temporal costs.

This is mainly due to overhead introduced with the program parallelization. That higher cost is caused by both creation/management of threads and mutex usage to guarantee mutually exclusive access to shared variables (costs that we do not have to support in case of serial program).

Generally the parallelization is very efficient when we have a lot of data to work on and when the contention between shared variable si very low.

Below are reported some stats that show the obtained results in our context.

No of elements	speedup
1,000	1.40
10,000	1.95
100,000	1.98
1,000,000	2.66
10,000,000	2.71

Table 1. k-means 4 clusters

No of elements	speedup
1,000	1.90
10,000	2.10
100,000	2.76
1,000,000	2.83

Table 2. k-means 10 clusters

The attempts have been made on a computer with an i7 and 4 cores (physical + ht). The results show a speedup obtained by comparison of

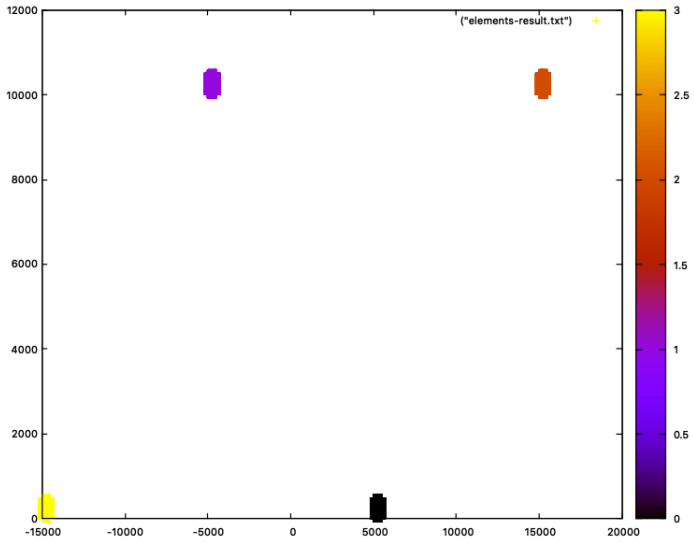

Figure 1. Graphical cluster representation using Gnuplot library.

No of elements	speedup
1,000	2.01
10,000	2.83
100,000	2.87

Table 3. k-means 100 clusters

sequential and parallel (with 8 threads) executions of the program.

Have been carried out several tests with a growing number of elements and clusters (in such a way as to make execution more complex). As we can see from the tables, is performed an increasing speedup that achieve a sub-linear trend; non reaching the ideal value equal to the number of cpu's cores.

5. References

1. Lin, Snyder - Principles of Parallel Programming
2. Marco Bertini - Parallel Programming course slides
3. C++ programming language
4. OpenMP library for C++
5. Gnuplot library