

Parallel K-Means

Agostino Aiezzo - 982514

September 21, 2022

1 Introduction

The objective of this project is to write a parallel version of the K-Means algorithm using OpenMP. After some preliminary operations 2, starting from a basic version of the algorithm and going through a series of modifications, attempts and tests, a sufficiently good solution has been progressively achieved 3. The results will be analyzed and discussed in a dedicated section below 4 as well as some conclusions and considerations 5.

2 Preliminary actions

Before proceeding, it is worth to spend some words about a couple of elements that contributed to support the development of this project. In particular:

- the *VectorPoint* structure which was defined and used to store and manage all the data point: based on an original structure named *Point*, all the basic operations like setting elements, getting elements, resizing vector and so on, were defined.
- the *cluster points generator* code, written in Python, to create clusters of a desired size. This data is acquired then through a *read_csv()* function in C.

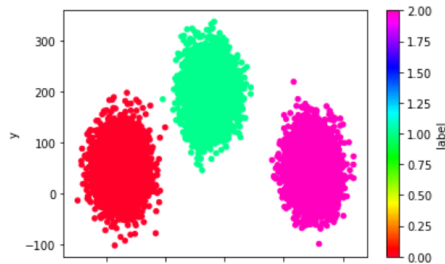


Figure 1: An example of the generated clusters

More in details, the data synthesizer allowed to test all the solutions with the same kind of dataset, the one represented in Figure 2, by just varying the size that, for our experiments, will go from 5000 to 500000 units. Of course, this was generated just once for each size, in order to fairly compare all the results.

In this way, we're able to retrieve some important performance indices by changing the number of threads and the amount of data. Ultimately, for visualizing clustering results through graphs, Python was used since in C is really difficult to plot something.

3 Solutions

As already mentioned at the beginning, starting from a basic version of the algorithm, different kinds of reasoning and omp directives were tested in order to get progressively better solutions. In fact, the objective was to decrease the general execution time.

Below, each approach is detailed, mentioning the idea behind it, why the solution works or not, advantages and disadvantages.

3.1 Solution 1

The first solution is based on the consideration that each cluster is independent from the others. As a consequence, the idea is to make the for loop that considers each centroid parallel, in order to separate all the computation of the distances. In Figure 3.1, it's possible to observe the code for this part.

The problem with

this kind of solution is that it requires to put a critical section in order to control the access of the threads to the point parameters, since the distances must be computed for each of them. As a consequence, everything is slow down and, in addition, the number of threads that can manage the for loop is limited by the number of centroids, thus parallelism cannot be completely exploited. Of course, these considerations will have consequences not only on execution times, but also on future decisions.

```
for (int i=0; i<epochs; i++)
{
    #pragma omp parallel for num_threads(num_thr)
    for(int c=0; c<centroids->size; c++)
    {
        Point centroid = VectorPoint_get(centroids,c);

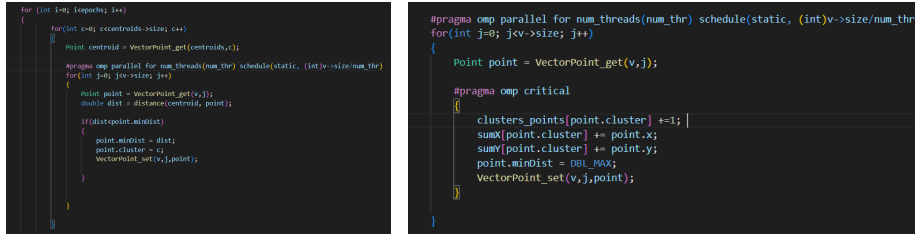
        for(int j=0; j<v->size; j++)
        {
            Point point = VectorPoint_get(v,j);
            double dist = distance(centroid, point);

            #pragma omp critical
            {
                if(dist<point.minDist) {
                    point.minDist = dist;
                    point.cluster = c;
                    VectorPoint_set(v,j,point);
                }
            }
        }
    }
}
```

Figure 2: Parallelized loop of the first solution

3.2 Solution 2

Taking into account what was previously discovered, the focus was moved directly to the points: this makes sense because, of course, data is way higher in size than the number of centroids. In particular, the for loop responsible of switching from a point to another during the computation of the distance from the centroid, has been made parallel. Thanks to this approach, it is possible for the threads to access privately to each point, avoiding, in this way, concurrency. The same reasoning is applied also to the for loop that deals with the computation of the new centroids as the ratio between the past coordinates of the centroid and the number of points that belong to the correspondent cluster.



```

for (int i=0; i<epochs; i++)
{
    for (int c=0; c<centroids->size; c++)
    {
        Point centroid = VectorPoint_get(centroids,c);
        #pragma omp parallel for num_threads(num_thr) schedule(static, (int)v->size/num_thr)
        for (int j=0; j<v->size; j++)
        {
            Point point = VectorPoint_get(v,j);
            double dist = distance(centroid, point);
            if (dist<point.minDist)
            {
                point.minDist = dist;
                point.cluster = c;
                VectorPoint_set(v,j,point);
            }
        }
    }
}

#pragma omp parallel for num_threads(num_thr) schedule(static, (int)v->size/num_thr)
for (int j=0; j<v->size; j++)
{
    Point point = VectorPoint_get(v,j);
    #pragma omp critical
    {
        clusters_points[point.cluster] +=1;
        sumX[point.cluster] += point.x;
        sumY[point.cluster] += point.y;
        point.minDist = DBL_MAX;
        VectorPoint_set(v,j,point);
    }
}

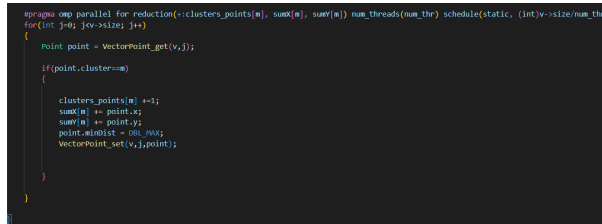
```

Figure 3: Parallelized loops of the second solution

Unfortunately, also in this case, a critical section must be defined. It may happen, in fact, that two threads with 2 different points (that have been classified with the same cluster) can access contemporaneously to the same position of the count vector. This makes everything slower, of course, but still better than the previous approach.

3.3 Solution 3

The third solution avoids to use the critical section in the second part of the algorithm just by applying the reduction mechanism when computing the new centroids. This required to upset the original code a little. In Figure 3.3, the code is shown.



```

#pragma omp parallel for reduction(+:clusters_points[n], sumX[n], sumY[n]) num_threads(num_thr) schedule(static, (int)v->size/num_thr)
for (int j=0; j<v->size; j++)
{
    Point point = VectorPoint_get(v,j);
    if (point.cluster==n)
    {
        clusters_points[n] +=1;
        sumX[n] += point.x;
        sumY[n] += point.y;
        point.minDist = DBL_MAX;
        VectorPoint_set(v,j,point);
    }
}

```

Figure 4: Parallelized loop of the third solution

As we will see in a while 4, this last approach can be considered the best and most efficient so far, despite further improvements could be made.

4 Results

We're going to discuss, now, about the results of each solution. In particular, we're going to focus mainly on two measurements: *execution* execution time and *scalability/efficiency* in the form of *strong scale efficiency*. For testing deeply all the solutions, what we did is to consider a combination of different thread numbers and sizes for the dataset. For each combination, in order to have a more precise result, the execution time was computed as the mean value over 1000 repetitions.

4.1 Result 1

In Figure 5 results of the first solution are depicted. What we can notice is that, in general, there's no improvements at all when using multi threads, since the average execution time is higher than the single thread case. There's just **one case** in which more threads perform good: if we try to compute the speedup, we get $S(2) = 1.23$ and an amount of reduced work of about 60%. In general, this would be good, but due to the other negative results, this case should be considered more an exception than anything else.

k-meansl	1	2	4	6	8
5000	0.016358	0.020114	0.019697	0.019985	0.019872
15000	0.046379	0.057589	0.057474	0.057712	0.065879
100000	0.338469	0.468249	0.437861	0.389608	0.443094
500000	1.730340	1.399268	2.293763	1.870761	1.812756

Figure 5: First solution execution times

Summing up, this approach is not good not only because of the inefficiency of the approach itself, but also, and mainly I would say, because of the critical section which makes threads very slow and randomize their execution.

4.2 Result 2

In Figure 6 a decidedly better and more coherent situation can be observed. What we can deduce is, in fact, that the **best results** are achieved by using 4 or 6 threads. Making some computations with the correspondent values, a speed up between 16-48% and a strong scale efficiency between 20-32% are achieved.

k-means2	1	2	4	6	8
5000	0.017440	0.015703	0.015002	0.015395	0.016250
15000	0.048957	0.044512	0.037897	0.038014	0.043519
100000	0.329659	0.284511	0.234058	0.221751	0.231441
500000	1.507543	1.617504	1.280326	1.234042	1.276086

Figure 6: First solution execution times

The road taken is certainly better than the previous one, but, as we will see shortly, there are even better results in the last solution.

4.3 Result 3

Observing Figure 7, we can easily determine that this is so far the best solution. Computing speed up and strong scale efficiency according to the **best cases**, we have, respectfully, a mean value of 50% for the first parameter, and a range of values between 38-55% for the second one.

k-means3	1	2	4	6	8
5000	0.018318	0.016410	0.017410	0.019099	0.017444
15000	0.054158	0.040689	0.034732	0.036429	0.036045
100000	0.325401	0.264318	0.238621	0.186850	0.182028
500000	1.262901	0.968262	0.816142	0.927593	0.879923

Figure 7: First solution execution times

5 Conclusions

In general, results are good and, in some cases, impressive, especially in the last solution. However, they could be further improved, for example, by using a real dataset or more sophisticated omp directives. Other possibilities could be to adopt a different composition of the algorithm as well as an approach that tries to reduce the overhead in the creation of the threads. In any case, we have a demonstration that using parallelism in applications with a lot of data, as is clustering, is really determinant to reduce time and exploit resources.