Parallel K-means

Ludovico Granata

September 2022

1 Introduction

The goal of this document is to present the work done to implement a parallel version of the popular cluster algorithm "**K-means**" using OpenMP[1]. The implementation will be tested on point cloud data, set of 3D points in the space, representing objects or shapes. In particular we will take scan of real objects from the Stanford 3D Scanning Repository[2].
The document will first analyze the algorithm, then it will describe the parallel implementation and conclude with a performance evaluation.

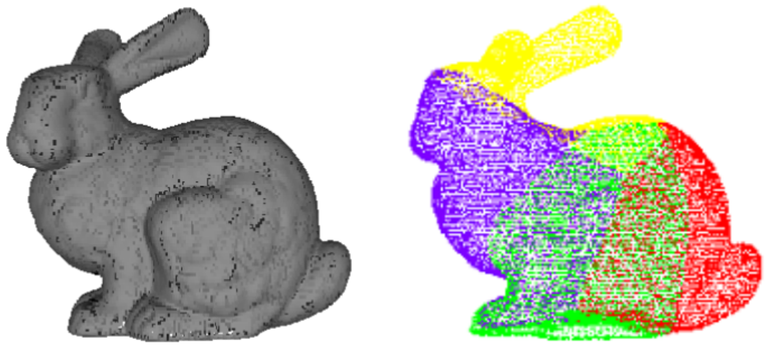

Figure 1: On the left a point cloud depicting a bunny. On the right the application of the K-Means algorithm (**K**=4) on the point cloud.

2 Algorithm Analysis

The **K-means** algorithm partitions a set of values, usually, d-dimensional real vectors, in k groups, by computing k representatives, named centroids: the val-

[1]https://www.openmp.org/
[2]http://graphics.stanford.edu/data/3Dscanrep/

ues in input that share the same nearest centroid belong to the same computed partition. The **K-means** algorithm is composed by three main phases, the first phase is computed only one time, while the phase 2 and 3 are repeated until specific conditions are met :

Algorithm 1 K-means

```
Initialization()                                  ▷ Phase 1
while conditions do
    NearestCentroidAssignment()                   ▷ Phase 2
    UpdateCentroids()                             ▷ Phase 3
end while
```

The first phase is the initialization phase, the position of each of the **K** centroids is computed by following the *Forgy method* meaning by randomly choosing **K** points from the dataset as initial position.
In the second phase each data point is assigned to the nearest centroid, following a specific metric, in this case we apply *L2* distance.
Finally in the third phase the position of each centroid is updated as the means of the assigned points.
The algorithm stops when the position of the new centroids do not change.
The execution of each phase needs the output of the previous executed phase, so in a parallel version of the algorithm we can act on the single phase but we can't run the phases altogether.

Algorithm 2 Parallel K-means

```
parallel {
Initialization()                                  ▷ Phase 1
}
while conditions do
    parallel {
    NearestCentroidAssignment()                   ▷ Phase 2
    }
    parallel {
    UpdateCentroids()                             ▷ Phase 3
    }
end while
```

3 Parallel implementation

In the following we will present a parallelize version of each phase described in the previous chapter.

3.1 Phase 1: Centroids initialization

In the first phase we randomly select a point and we take it from the dataset with the function `getRow()` and we initialize the centroid with the `setRow` function. Since we repeat this process inside a for loop for **K** times it can be easily parallelized with the `omp parallel for` directive. Since each job has the same computational effort we can use a static schedule.

```
#pragma omp parallel for schedule(static,schedule)
  for (i = 0; i < K; i++){
    setRow(&centroids, i, DIM, getRow(points, random_int(0,
    n_points), DIM));
  }

```

3.2 Phase 2: Nearest centroid

After entering the while loop instead of creating and terminating threads each time we need them, we can create a pool of threads that can be reused. The default behaviour is to have shared variables, we change it to private only for variables that will be used to iterate in a loop.

```
  while (finish == 0)  {
  #pragma omp parallel private(i,j,p,k)
    {
      NearestCentroidAssignment();
      UpdateCentroids();
    }
   }

```

To assign each point to the nearest centroid we compute, for each point, the distance with each centroid. Then we compute the closer centroid for each point and save the result. The computation of each distance is independent from the others and so it can be parallelized. We use the `omp for` directive to produce a job for each point. Since we have an internal loop over the **K** centroids we need to specify that the `j` iteration variable is private, otherwise every thread will be able to incrementing it, resulting in race condition.

```
#pragma omp for private(j) schedule(static,schedule)
      for (i = 0; i < n_points; i++)
      {
        for (j = 0; j < K; j++)
            setValue (& distances , i, j, K, computeDistance (
    getRow (points , i, DIM ), getRow ( centroids , j, DIM), DIM));
        cluster[i] = argmin(distances, i, K);
      }

```

3.3 Phase 3: Update of centroid's positions

The computation of the new centroids can be done in two steps, first we sum all the values of each cluster saving the number of element for each of them, then

we divide the total sum by the number of members.
Since we want to sum the points for each centroid, we can save the result in a array `clusterSum`, and we can apply a reduction, in this way OpenMP will managed autonomously the access to the array.
To divide each sum by the number of member of each cluster we have two innested loops that iterate over the number of cluster and number of dimension. Since the two loops are perfectly nested we can use the `collapse(2)` keyword. There isn't an imbalance of the work assigned to each job so we can use a static schedule.

```
#pragma omp for reduction(+:clusterSum) private(i) schedule(static,
    schedule)
      for (p = 0; p < n_points; p++)
      {
        for (i = 0; i < DIM; i++){
          clusterSum[cluster[p]][i] += getValue(points, p, i, DIM);
        }
        clusterSum[cluster[p]][DIM]++;
      }

#pragma omp for collapse(2) schedule(static,schedule)
      for (i = 0; i < K; i++)
      {
        for (k = 0; k < DIM; k++)
          clusterSum[i][k] = clusterSum[i][k] / clusterSum[i][DIM];
      }

```

finally we have to check the termination condition: first we assign 1 to the `finish` variable then if there is at least one centroid position that has been changed, we assigned to it value 0. We assign the value 1 using the directive `omp single` to make it execute only by the first thread that is available.

```
#pragma omp single
      finish = 1;

#pragma omp for schedule(static,schedule)
      for (i = 0; i < K; i++)
      {
        if (equal(clusterSum[i], getRow(centroids, i, DIM), DIM) ==
     1)
          finish = 0;
        setRow(&centroids, i, DIM, clusterSum[i]);
      }

```

4 Performance evaluation

To evaluate the performance of the parallelized version of K-means we first do a strong performance analysis and then a weak performance analysis.
All the results are computed using the University of Bologna computational resources:

- Intel Xeon quad-core CPU
- 44 GB of RAM

All the results are computed three times and averaged together.

4.1 Strong performance analysis

Here we analyse how much faster a given problem can be solved with `p` workers instead of one. We compute two index, the speedup and the strong scaling efficiency, that is just the speedup divided by the number of threads. The red line is the ideal case.

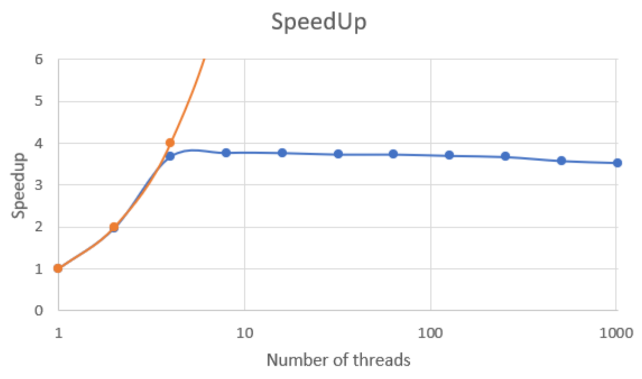

Figure 2: Speedup with 500.000 points and K=100

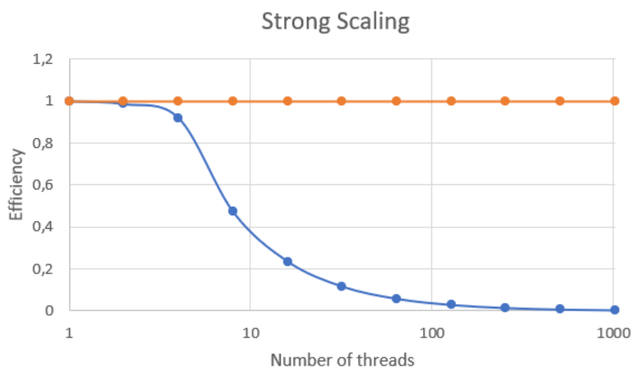

Figure 3: Strong Scaling Efficiency with 500.000 points and K=100

We can see that the efficiency (and speedup) are almost ideal below four thread. This was an expected result considering that the used CPU is quad-core.

4.2 Weak performance analysis

The complexity of the K-means algorithm is:

$$\mathcal{O}(N \times K \times DIM \times I)$$

with:

- **N** total number of points in the dataset
- **K** number of cluster
- **DIM** dimensionality of the datapoint
- **I** number of iteration to reach the end

To compute the Weak coefficient we have to double the amount of work every time we double the amount of threads, in other words the work for each thread should remain the same. Since we cannot change the number of iteration of the algorithm or the number of dimension of the data points, to double the amount of work we have to act on the number of points and the number of clusters.So it is easy to see that if we want to double the amount of work we will have:

$$N_i = \sqrt{2} \times N_{i-1} \ \texttt{and} \ K_i = \sqrt{2} \times K_{i-1}$$

where $i = 2^p$ with p being the number of threads, starting with p=1.

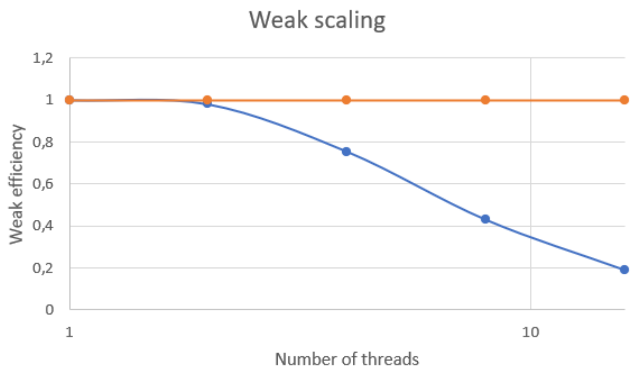

Figure 4: Weak scaling coefficient

Also with the weak scaling coefficient we have very good performance below four threads, while for higher number of threads the efficiency slowly go towards zero.