K-means clustering with OpenMP

Dragos Tanasa
E-mail address
dragos.tanasa@stud.unifi.it

Abstract

K-Means is a popular unsupervised learning algorithm used both in data mining and machine learning. Its primary objective is to partition a dataset into a user-defined number of clusters facilitating the discovery patterns within the data. In modern times datasets have grown significantly in size comprising a vast number of observations and high dimensionality. Consequently the demand for faster and more efficient ways to apply these algorithms has become imperative.

Future Distribution Permission

1. Introduction

In this paper an exploration of the K-Means algorithm is presented detailing its theoretical foundation and operational principles. Subsequently a will be introduced a parallelization strategy with OpenMP, a widely-used parallel programming framework designed to facilitate shared-memory multiprocessing. By effectively distributing the computational workload across multiple cores, the parallelized K-Means implementation seeks to reduce the overall execution time, enabling faster and more efficient clustering of large-scale datasets.

2. K-Means

The K-Means algorithm is an unsupervised learning technique used for clustering data points into K distinct groups or clusters. It aims to find cluster centers (centroids) in such a way that data points within the same cluster are more similar to each other than to those in other clusters. This mathematicly translates to minimizing the within-cluster sum of squares (WCSS) (i.e. variance). Formally, given a set of observation $\mathbf{x_1}...\mathbf{x_n}$ we want to find

$$arg\min_S \sum_{i=1}^{k} \sum_{x_i \in S_i} \|\mathbf{x} - \mu_\mathbf{i}\|^2 \tag{1}$$

Where

$$\mu_\mathbf{i} = \frac{1}{\|S_i\|} \sum_{\mathbf{x} \in S_i} \mathbf{x} \tag{2}$$

2.1. K-Means algorithm

Here's a brief explanation of the K-Means algorithm:

1. **Initialization** : Choose K initial cluster centers from the dataset. These centroids will act as the representatives of the clusters.

2. **Assignment**: Assign each data point to the nearest cluster center based on a distance metric, commonly the Euclidean distance.

3. **Update**: Calculate the new cluster centers (centroids) by computing the mean of all data points assigned to each cluster. The mean represents the ”center” of the cluster.

4. **Repeat**: Iterate steps 2 and 3 until convergence. Convergence occurs when the cluster assignments no longer change significantly, or when a specified maximum number of iterations is reached.

The complexity is:

$$O(n \times d \times k \times iter)$$

Where n is the cardinality of d-dimensional datapoints, k is the number of clusters and $iter$ is the number of iteration required to converge.

2.2. Sequential implementation in C++

```
std::vector<Point> new_centroids =
    std::vector<Point>(k);
int cluster_cardinality[k];

for(int i = 0; i < epochs; i++){
   for(int j = 0; j < k; j++){
      new_centroids[j].to_zero(-1);
      cluster_cardinality[j] = 0;
   }
   for (Point& p: points){
      double distance = DBL_MAX;
      for(int i = 0; i < k; i++){
         if (euclidean_dist(p,
             centroids[i]) < distance) {
            distance =
                euclidean_dist(p,
                centroids[i]);
            p.cluster = i;
         }
      }
      new_centroids[p.cluster] += p;
      cluster_cardinality[p.cluster]++;
   }
   for(int i = 0; i < k; i++){
      new_centroids[i] /=
          cluster_cardinality[i];
   }
   centroids = new_centroids;
}
```

The code employs the Point class, which functions as a struct containing the coordinates of a point along with its associated cluster ID. This class is further equipped with operator overloads aimed at enhancing the code's readability.

During each iteration, the algorithm initializes `new_centroids`, a vector of points, and `cluster_cardinality`, an array, to zero. The algorithm then proceeds to iterate through each point, determining the cluster that is closest to the point in question. Upon identifying the i-th centroid as the nearest one, the algorithm adds the coordinates of the point to the i-th element of `new_centroids`. Simultaneously, it increments the i-th element of `cluster_cardinality` by one. This process continues for all points in the dataset.

A coordinate-wise division is performed of the vectors within `new_centroids` by their corresponding cardinalities. This operation yields the updated coordinates for the new centroids, effectively recalculating their positions based on the mean of the points within each cluster.

3. Initialization

One of the major drawbacks of the K-Means algorithm is its sensitivity to the initial clustering initialization. This sensitivity affects both the convergence speed and, more significantly, the convergence itself. Unfortunate initializations can lead to highly suboptimal solutions. Two types of initialization are commonly used:

- **Random initialization**: k random points are selected from the dataset. This approach suffers from the issues mentioned earlier and is not very reliable despite its efficiency (constant time complexity).

- **KMeans++**: k points are selected maximizing the distance between them. The first point is selected randomly, the second is randomly selected with probability proportional to the distance from the first one and so on. This leads to more reliable and efficient clustering results.

3.1. Random initialization implementation

The random initialization is straightforward. We randomly pick k points from the list of Points.

```
std::vector<Point> centroids;
std::random_device rand_dev;
std::mt19937 gen(rand_dev());
std::uniform_int_distribution<> distrib(0,
    points.size());
int i = 0;
while (centroids.size()<k) {
   Point new_centroid =
       points[distrib(gen)];
   if (!contatins(centroids,
       new_centroid)) {
      new_centroid.cluster=i;
```

```
        i++;
        centroids.push_back(new_centroid);
    }
}
return centroids;
```

3.2. K-means++ implementation

The K-means++ algorithm, on the other hand, is more complex but lends itself well to parallelization. The idea is to iterate through each point and calculate its distance with respect to the closest already chosen centroid. The new centroid is chosen based on a weighted probability distribution, with the weights being the distances previously calculated.

Here is shown only the core functionality of the initialization.

```
while (centroids.size() < k) {
    std::vector<double>
        distances(points.size(),
        std::numeric_limits<double>::max());
    #pragma omp parallel for
        num_threads(threads)
    for (size_t i = 0; i < points.size();
        ++i) {
        for (const Point &centroid :
            centroids) {
            double distance =
                euclidean_dist(points[i],
                centroid);
            distances[i] =
                std::min(distances[i],
                distance);
        }
    }
    Point nextCentroid =
        next_centroid(points, distances);
    centroids.push_back(nextCentroid);
}
return centroids;
```

4. Parallelization strategy

It's worth pointing out that K-means falls into the category of embarrassingly parallel problmes, each point can be in fact assigned to the closest centroid independently from the others and that make it a perfect candidate for parallelization.

4.1. Main strategy

The most significant expense in the sequential implementation is associated with the loop responsible for iterating through each point and assigning it to every cluster. This operation alone incurs a cost of $O(n \times d \times k)$. The update phase of the algorithm, costing $O(k)$, remains relatively negligible in comparison to the rest of the algorithm.

Turning to the parallel implementation, the primary objective is to reduce computation time through the distribution of work across various CPUs. The aim is to achieve a complexity of $O\left(\frac{n \times d \times k}{N} \times \text{iter}\right)$, where N represents the number of utilized CPUs by parallelizing the main loop mentioned. In order to achive that a **reduction like strategy was employed** . First two new **private** variables were defined: `tmp_new_centroids` and `tmp_cluster_cardinality`. Each core operates on its chunk of points in a manner analogous to the sequential algorithm.

```
#pragma omp parallel num_threads(threads)
{
    std::vector<int>
        tmp_cluster_cardinality(k, 0);
    std::vector<Point> tmp_new_centroids(k,
        Point());
    #pragma omp for nowait schedule(static,
        points_per_thread)
    for (Point& p: points) {
        double distance = euclidean_dist(p,
            centroids[0]);
        p.cluster = 0;
        for (int i = 1; i < k; i++) {
            double tmp = euclidean_dist(p,
                centroids[i]);
            if (tmp < distance) {
                distance = tmp;
                p.cluster = i;
            }
        }
        tmp_new_centroids[p.cluster] += p;
        tmp_cluster_cardinality[p.cluster]++;
    }
```

The schedule policy chosen is `static`. The dataset is divided into equal parts. A `nowait` clause was added since there is no need for threads to synchronize before entering the critical section. On the contrary, it could improve performance if threads finish their workload asynchronously.

Subsequently we enter a crtical region in which each thread adds the its contribution to the `new_centroids` and `cluster_cardinality` arrays.

```
#pragma omp critical
{
   for (int i = 0; i < k; i++) {
      new_centroids[i] +=
          tmp_new_centroids[i];
      cluster_cardinality[i] +=
          tmp_cluster_cardinality[i];
   }
}
```

The code then procedes as the sequential one.

4.2. Vectorization

Additionaly, we can use vectorization when calculating the distance. This could increase the performance of our code especially when dealing with high dimensional data. Vectorizaton in this case would be achived by simply exploiting the omp directive that OpenMP provides

```
double euclidean_dist(const Point& p1,
    const Point& p2){
   double distance= 0;
   #pragma omp simd
   for (int i=0; i<DIM; i++)
      distance += (p1.coordinates[i] -
          p2.coordinates[i]) *
          (p1.coordinates[i] -
          p2.coordinates[i]);
   return distance;
}
```

Despite working in theory this approach doesn't seem to improve the performance of out algorithm in the case presented as we will se in Section 6.2

5. Algorithm Correctness

Algorithm Correctness was checkd by visually plotting the result of the clusterization using 2 and 3 dimensional datasets.

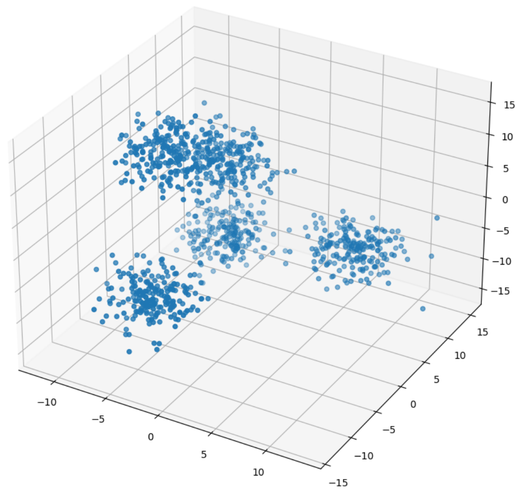
Figure 1. Initial cluseters

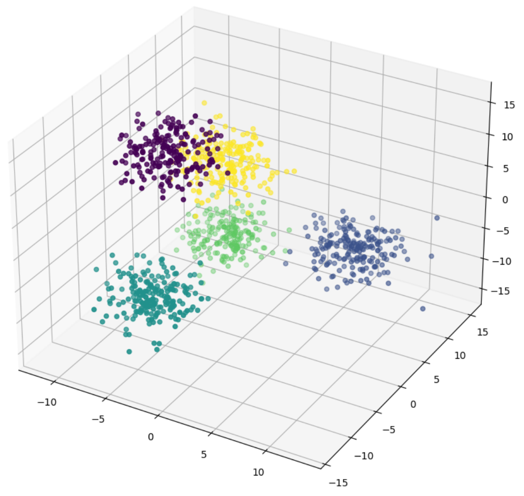
Figure 2. After clustering.

6. Setup

Experiments were performed on AMD Ryzen7 4800H (8 CPU cores and 16 threads) and NVIDIA GeForce RTX 2060 Mobile. The code was compiled using g++ 11.4.0 on Ubuntu 22.04.

Test were perfomed on synthetic data generate with `make_blobs` function of Scikit-learn framework

7. Results

7.1. Convergence

The convergence criterion involves two criteria: one checks if the max number of iterations

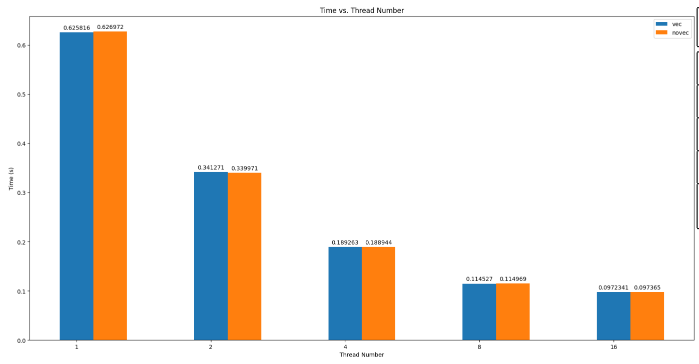

Figure 3. Execution time of vectorized vs unvectorized code.

has been reached and the other check if the different between the position of the centroids in two subsequent iteration is below a certain treshold. For the speedup analysis only the first criterion was chosen fixing the number of iterations at 100 since the second one is higly dependent on the initialization.

7.2. Effect of vectorization

As anticipated in Section 3.2, vectorization doesn't seem to help in our particular case. The experiment involved running the K-means algorithm 100 times with different thread numbers on a dataset of 10,000 observations with 10 clusters. Each point belongs to a 10-dimensional space; a higher-than-normal dimensionality was chosen to enhance the potential effects of vectorization, if they were to manifest. The results are shown in Figure 3. The result doesn't seem to improve even when the vectorization is manually implemented using AVX2 instructions. This lack of improvement could potentially be attributed to our usage of an Array of Structures approach for handling point coordinates rather than a Structure of Arrays.

7.3. Speedup analysis: number of observation

The first thing we want to test is the speed up of the parallel code with respect to the number of cores and the dataset cardinality. 5 different datasets of 3-dimensional points containing 10 clusters where generated, the dataset cardinality was respectivly 1000, 5000, 10000, 50000 and 100000. The K-means algorithm was then runned

	2 Thread	4 Thread	8 Thread	16 Thread
1000	1.31	2.42	2.63	1.44
5000	2.07	5.47	**9.63**	8.55
10000	1.23	2.70	7.50	8.76
50000	1.53	2.36	3.48	5.72
100000	1.77	2.96	4.76	6.61

Table 1. Speed Up

using 1, 2, 4, 8 and 16 threads. For each combination (thread number, cluster cardinality) the algorithm was runned 100 times and the results averaged.

The result of the experiment is shown in Table 1.

A graphical representation of the speed up can be seen in Figure 4.

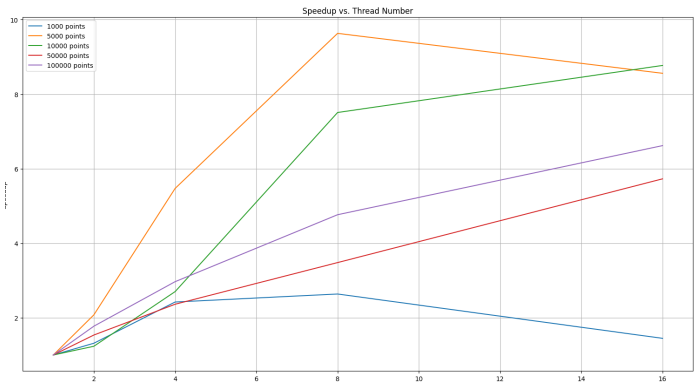

Figure 4. SpeedUp graph

The highest speed-up recorded is 9.63. Interestingly, both of the smallest datasets peaked at 8 threads. It is plausible that this behavior is influenced by the fact that with 16 threads, the individual K-means iteration is faster than the overhead introduced by spawning and managing all 16 threads. Additionally, the overhead cost could also explain why we measure different speed-ups for different cluster cardinalities when using the same number of threads: the cost of spawning and managing n threads is constant with respect to the number of points, but it is proportionally lower for each iteration when dealing with bigger datasets.

	Random	K-means++
mean time	3.59e-05 s	0.02 s

Table 2. Speed Up

7.4. Speedup analysis: number of clusters

With a setup similar to the one illustrated before, the effect of the number of clusters was tested. Having fixed a dataset size of 10,000 observations we observed what happens with respect to cluster numbers 2, 3, 5, 10, 20, and 50. The results are illustrated in Figure 5.

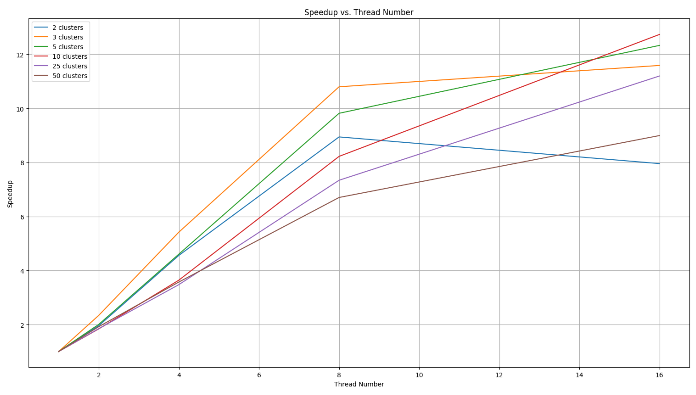

Figure 5. SpeedUp graph

As expected, the speed-up increases with the number of cores, with the exception of the 2-cluster case, where we observe a decrease in speed-up when going from 8 to 16 clusters. This can be explained in the same way we did before.

7.5. Kmeans++

All previous test were executed using random initialization since the focus was on understanding the speedup of the kmeans algorithm and helped speeding the profiling process. The K-means++ initialization is infact order of magnitude slower than the Random initialization as we can see in Table 2

7.6. Conclusion

In conclusion, this project delved into the intricacies of the K-Means algorithm and its parallelization using the OpenMP framework. We explored its theoretical foundations, operational steps, and initialization methods. By leveraging parallelization, we effectively reduced execution times for large datasets. Our thorough analysis validated the accuracy and efficiency of the parallelized implementation. This work contributes to the optimization of algorithms for big data applications and underscores the significance of parallel programming techniques in enhancing computational efficiency.