Kmeans: Sequential and Parallel Version

Gianluca Giuliani
E-mail address
giuliani.gianluca1@edu.unifi.it

Abstract

K-means clustering is a fundamental machine learning technique used to partition data into K clusters. Each data point is assigned to the nearest cluster center.

This report explores the implementation of the K-means algorithm in both sequential and parallel versions using OpenMP. The parallel version, using multiple threads, can enhance computational efficiency and performance to reduce execution time and handle larger datasets.

Introduction

1. Device

The code is executed on an Intel I7 processor with 14 cores and 20 logic processors.

2. Algorithm

The K-means clustering algorithm is implemented using the following steps:

- Initialization:

  Randomly select K initial cluster centroids from the dataset.

- Assignment:

  For each data point, calculate the Euclidean distance to each centroid and assign the point to the nearest centroid, forming K clusters.

- Update:

  Recalculate the centroids by computing the mean of all data points assigned to each cluster.

- Iteration:

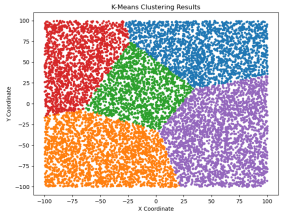

Figure 1. Results generated with 10000 points and 5 classes

  Repeat the assignment and update steps until the centroids no longer change or the maximum number of iterations is reached.

2.1. Parameters

- Number of Points
- Number of Clusters (K)
- Maximum Number of Iterations
- Number of Threads

2.2. SoA vs AoS

”AoS” and ”SoA” stand for ”Array of Structures” and ”Structure of Arrays,” respectively. They are two different ways of organizing data in memory, particularly when dealing with collections of related elements. AoS (Array of Structures) groups related data into an array of structures, while SoA (Structure of Arrays) organizes data into separate arrays based on attributes. AoS is intuitive and easy to use, while SoA can offer performance benefits in certain scenarios, especially for large datasets and optimized memory access

2.3. Code - Structure of Array

Here I report the most relevent snippets of the code of SoA parallel version. The sequential version is almost equal to parallel version.

Utility.h Definiton of the Struct.

```
struct alignas(64) Points {
    std::vector<float> x;
    std::vector<float> y;
    ...
};
```

I don't create a Cluster struct because I choose to simply use some vectors, more than other, I will implement a reduction and openMP requires arrays, still I don't know why an array inside a structure is not considered correct for openMP reductions.

SoA.h Init random centroids:

```
std::vector<float> centroidsX(k),
centroidsY(k);
#pragma omp parallel
{
std::random_device rd;
std::mt19937 gen(rd() ^
omp_get_thread_num());
std::uniform_real_distribution<float>
dis(-100.0f, 100.0f);

#pragma omp for
for (int i = 0; i < k; i++) {
    centroidsX[i] = dis(gen);
    centroidsY[i] = dis(gen);
}
}

float newCentroidsX[k] = {0.0f},
newCentroidsY[k] = {0.0f};
int counts[k] = {0};
```

I use vectors of float to record the coordinates of centroids. I will use reduction on cumulative sums in order to update centroids so I define them here.

And I use shared to make the arrays usable during reduction.

```
#pragma omp parallel shared(newCentroidsX,
newCentroidsY,
counts)
{
for (int iter = 0; iter <
maxIterations; iter++) {
...
distances calculation
update of clusters
...
}
}
```

Computation distance point - centroids

```
#pragma omp for schedule(dynamic,1000)
for (size_t i = 0; i < numPoints; i++) {
    int bestCluster = -1;
    float bestDistance =
    std::numeric_limits<float>::max();
    for (int j = 0; j < k; j++) {
        float d = distance(points, i,
        centroidsX[j], centroidsY[j]);
        if (d < bestDistance) {
            bestDistance = d;
            bestCluster = j;
        }
    }
    labels[i] = bestCluster;
}
```

This part is imbarazzingly parallelizable, the only thing where to put attenction is that evrey thread must have its own private variables to work on, using varibles I assure that.

Update centroids

```
#pragma omp for schedule(dynamic, 1000)
for (size_t i = 0; i < numPoints; i++) {
    int bestCluster = -1;
    float bestDistance =
    std::numeric_limits<float>::max();

    for (int j = 0; j < k; j++) {
        float d = distance(points, i,
        centroidsX[j], centroidsY[j]);

        if (d < bestDistance) {
            bestDistance = d;
            bestCluster = j;
        }
    }

    labels[i] = bestCluster;
}
```

I made some different tests changing the schedule: dynamic, static, guided and runtime. With runtime I score the same execution times in both sequential and parallel code. With schedule and dynamic, I noticed really small changes.

2.4. Code - Array of Structure

Cluster Point.h Here i define my classes

```
class Point {
    float x, y;
    int cluster_id;
...
   }
};


class Cluster {
    float x, y;
    float sum_x, sum_y;
    int size;
    ...

Cluster(float x_val = 0, float y_val = 0){

        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<>
            dis(-100.0, 100.0);

        x = dis(gen);
        y = dis(gen);
        sum_x= 0;
        sum_y= 0;
        size = 0;
    }
    void updateCenter() {
        if (size > 0) {
            x = sum_x / size;
            y = sum_y / size;
        }
    }
};
```

AoS.h In the file are implemented the parallel and sequential version of the code.

I only report snippets of parallel version.

Cluster Initialization

```
#pragma omp for
for (int i = 0; i < num_clusters; ++i) {
    clusters[i] =
    Cluster();
}
```

Here i simply initialize centroids randomly.

Distance calculation inside a loop on iteration:

```
#pragma omp parallel for schedule(dynamic, 1000)
for (size_t i = 0; i < points.size(); ++i) {
float min_distance =
    std::numeric_limits<float>::max();
    int best_cluster = -1;

for (size_t j = 0; j < clusters.size(); ++j) {
    float distance = points[i].distanceTo
    (clusters[j].getX(), clusters[j].getY());

    if (distance < min_distance) {
        min_distance = distance;
        best_cluster = j;
    }
}
points[i].cluster_id = best_cluster;
// Thread-safe because `i`
// is private to each thread
        }

float* cluster_sum_x = new float[clusters.size()]();
float* cluster_sum_y = new float[clusters.size()]();
int* cluster_count = new int[clusters.size()]();

...

delete[] cluster_sum_x;
delete[] cluster_sum_y;
delete[] cluster_count;
```

Vector initialization for reduction I will use later.

Accumulation: Distance calculation inside a loop on iteration:

```
#pragma omp parallel for reduction(+:
cluster_sum_x[:num_clusters],
cluster_sum_y[:num_clusters],
cluster_count[:num_clusters])
    for (size_t i = 0; i < points.size(); ++i) {

        int cluster_id = points[i]
            .cluster_id;
        cluster_sum_x[cluster_id] +=
            points[i].x;
        cluster_sum_y[cluster_id] +=
            points[i].y;
        cluster_count[cluster_id]++;
    }
```

Update center:

```
#pragma omp parallel for
for (size_t i = 0; i < clusters.size(); ++i) {

clusters[i].updateCenter(cluster_sum_x[i],
cluster_sum_y[i], cluster_count[i]);
        }
```

3. Dataset

Dataset file consists in a csv file with random float numbers generated with

`create_dataset.py`

. It is possible to change the file and change in main.cpp and main2.cpp the value of variable dataset.

4. Experiments for SoA

The metric used to compare the performances of the sequential algorithm with the OpenMP is the speedup, computed as

```
Speedup = ExecutionTime_seq /
    ExecutionTime_paral
```

4.1. Varying the number of points

What I will do is executing kmeans, clustering 10000, 10000000, 100000000 points.

5. Results

5.1. SoA

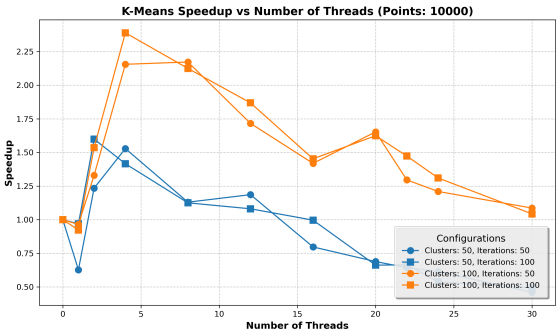

Figure 2. SoA Experiment varying number of Threads with 10000 points

As we can see in figure 2, the speedup is not so good with number of points that is small. In fact, if the computation increases, with number of cluster = 100, the speedup is better. Actually the overhead is slowing the code. As we can see in figure 3, the speedup increase with the increase of number of points, now it is actually linear. The max speedup is 10. One last note, the number of points, to fully use all the threads, is still too low, in fact there is a little decrease of the curves.

If I decide to use more threads, e.g. 30, than my logic processor i see that the performances start to decrease.

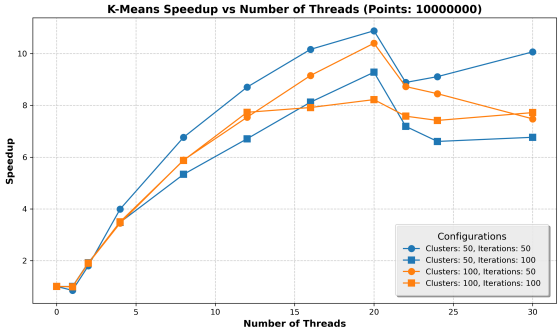

Figure 3. SoA Experiment varying number of Threads with 10000000 points

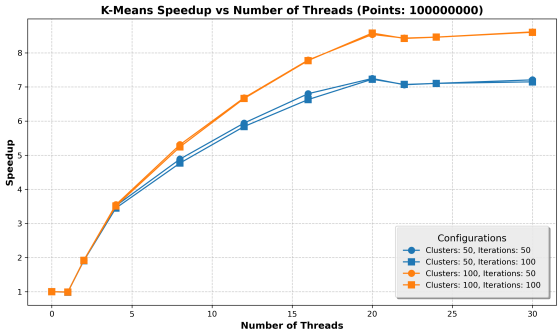

Figure 4. SoA Experiment varying number of Threads with 100000000 points

5.2. AoS

The experiment done in SoA section are repeated using AoS code.

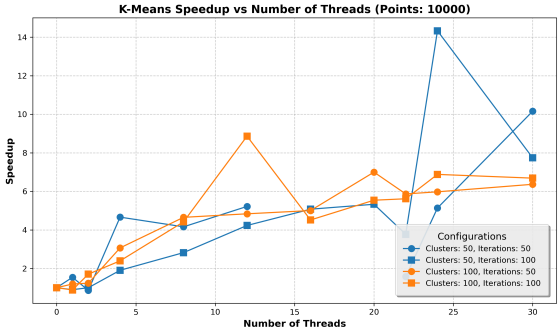

Figure 5. AoS Experiment varying number of Threads with 10000 points

5.3. SoA vs AoS

The following plots shows the execution times of SoA and Aos version. In generale SoA outperforms AoS version of the code.

As we can see, the execution times, when the

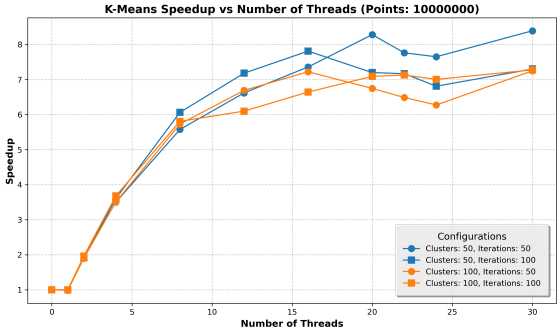

Figure 6. AoS Experiment varying number of Threads with 10000000 points

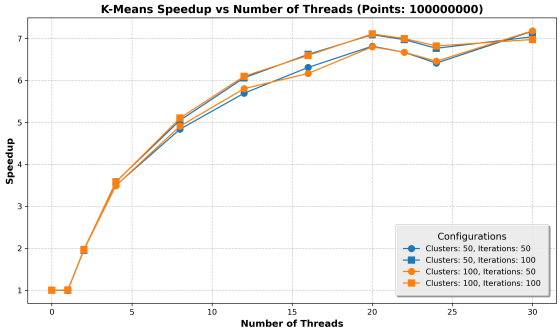

Figure 7. AoS Experiment varying number of Threads with 100000000

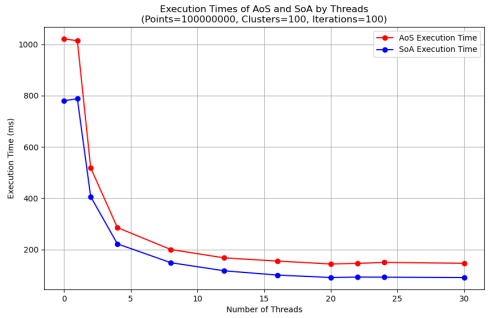

Figure 8. Execution times of AoS and SoA version in the same configuration test.

operation to compute consists in a lot of points, the execution times of SoA version are always better.

Here i summarize the results with a groupby.

5.4. Conclusions

The reports describes how to implement AoS and SoA versions of K means algorithm. About performances it is possible confirm that SoA version outperforms Aos kmeans.

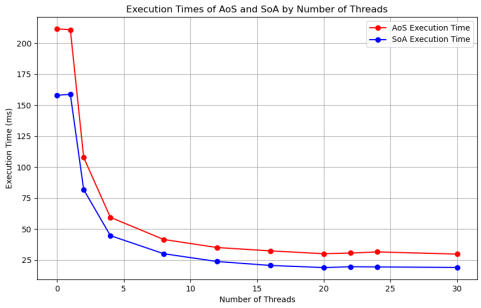

Figure 9. Execution times of AoS and SoA version with mean values.