Report Kmeans
Parallel Programming for Machine Learning project

Chisci Marco
marco.chisci@edu.unifi.it

Abstract

This report details the parallelization of a sequential K-means clustering algorithm using OpenMP. Its goal is to measure the speedup achieved by parallelizing K-means and compare different parallel implementations. The report first introduces the datasets used, followed by an overview of the K-means algorithm and its sequential and parallel implementations. The experiments evaluate execution times for varying configurations, such as different dataset sizes, point dimensions, numbers of clusters and threads.

1. Introduction

This report presents the work I did to parallelize a previously sequential version of K-means clustering, utilizing OpenMP.

The following sections will contain:

- Section 2: the points datasets used in the experiments
- Section 3: an overview of K-means clustering
- Section 4: detailed explanation of both the sequential and parallel implementations of the clustering algorithm.
- Section 5: a comprehensive analysis of the experiments to measure the time required to run the clustering algorithm under various configurations.
  The primary objective of this section is to assess the potential speedup gained from the parallelized version of the clustering process.

2. Used Data

To test the K-means clustering implementation, these simple 2D clusters were used: Kaggle datasets.
These datasets provide a straightforward way to verify the accuracy of the algorithm on well-defined clusters.

For testing with larger datasets of varying number of points and dimensions, *generate_random_csv* in 'generateData.h/cpp' was used. This function allows the creation of csv files containing randomly generated coordinates based on the specified number of points and dimensions, especially larger and more complex datasets than those easily found online.

3. K-means

K-means clustering is an unsupervised machine learning algorithm used to group an unlabeled dataset into distinct clusters.
Its goal is to divide the set of data points into k groups so that the data points within each group are more similar to each other than to those in other clusters (Fig. 1).

The algorithm works as follows:

- Randomly initialize k points, called means or cluster centroids.
- Assign each data point to the nearest cluster based on its distance to the clusters' centroids.
- Update the centroid coordinates by calculating the average of the points currently in each cluster.
- Repeat the second and third steps for a given number of iterations.

The choice of initial centroids can significantly influence the resulting clusters.
If a centroid is initialized as an outlier, it might end up with no points associated with it, and more than one cluster might end up linked with a single centroid. Similarly, multiple centroids might be initialized within the same cluster, leading to poor clustering results.

To mitigate these issues, K-means++ ensures a smarter initialization of the centroids and improves the quality of the clustering:

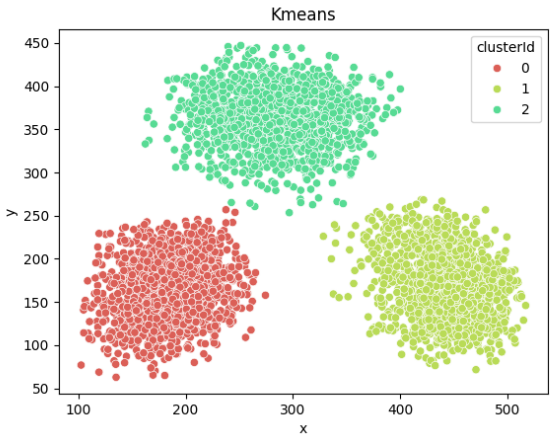

Figure 1. K-means++ clustering, 3 clusters

- Randomly select the first centroid from the data points.
- For each data point, calculate its distance from the nearest centroid.
- Choose the next centroid as the data point with the maximum distance from the previously selected centroids.
- Repeat the second and third steps until k centroids have been selected.

4. Code

4.1. Code structure

The following files, available in the GitHub repository, will be utilized in the upcoming experiments.

- **PointsSoA.h/cpp**: define and implement the class MultiDimensionalPointArray, where the points are stored as Structure of Arrays (SoA), and the Cluster class.
- **KmeansSoA.h/cpp**: implementation of the algorithm as a class with parameters such as the number of clusters (k) and the number of clustering iterations. This class manages all the points, clusters, and provides three different methods for running k-means clustering: one sequential and two parallel.
- **plotKmeans.py**: python code designed to plot the clustered points.
- **testKmeansSoA.cpp**: the executable file that tests and compares the three K-means implementations using different sets of points, varying numbers of clusters and iterations.

```
// Initializing clusters, select the first cluster
int index = getRandomIndex( size: (int)points.getNumPoints(), seed: 111);
// cluster numbered from 0 to k-1
points.setClusterIdByIndex(index, value: 0);
Cluster cluster( clusterId: 0, centroid: indexToCoordinates(index));
clusters.push_back(cluster);

std::vector<double>distances;
// kmeans++ initialization for the other clusters
for (int i = 1; i < K; i++){
    distances.clear();
    // find the closest existing cluster to each point, select the furthest
    for (int p =0; p<points.getNumPoints(); p++){
        double dist = 0.0;
        // iterate through all the previously found centroids
        for (int j = 0; j < i; j++) {
            double  sum = 0.0;
            for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                sum += pow( x: clusters[j].getCentroidCoordByPos( pos: dim) - points( pointIndex: p, dimensionIndex: dim), y: 2.0);
            }
            dist += sqrt( x: sum);
        }
        distances.push_back(dist);
    }
    // select the point that is furthest from all the other clusters as a new centroid
    index = std::max_element( first: distances.begin(), last: distances.end())-distances.begin();
    points.setClusterIdByIndex(index, value: i);
    Cluster cluster( clusterId: i, centroid: indexToCoordinates(index));
    clusters.push_back(cluster);
}
```

Figure 2. K-means++ sequential initialization

4.2. PointsSoA.h/cpp

These files define and implement the MultiDimensionalPointArray and Cluster classes.

The MultiDimensionalPointArray class organizes points using a Structure of Arrays (SoA) approach. When provided with a dataset of input points, its constructor determines the number of points and their dimensionality. Each coordinate of a point and cluster ID is stored in its own separate array, improving cache utilization and memory alignment, as the same coordinates from different points are stored contiguously in memory.

The Cluster class consists of a cluster ID, the centroid's coordinates, and the necessary getter and setter methods for managing these attributes.

4.3. KmeansSoA.h/cpp

These files contain the KMeansSoA class. Each KMeansSoA object has its points stored in a MultiDimensionalPointArray attribute, along with its clusters and the total number of clustering iterations.
The class provides three methods: *runSeq* runs a sequential version of K-means, while the other two are two different versions of parallel K-means using OpenMP.

runSeq initializes the clusters for the K-means++ algorithm (Fig. 2) and then iterates through the clustering steps (Fig. 3) as described in Section Sec. 3.

Both *runPar* and *runParPrivate* use the same parallelized initialization process (Fig. 4).
For each cluster (except the first), the for cycle that calculates the distance of each point from all the previous clusters' centroids is parallelized.

```
    // Initialize the sums of each cluster's coordinates and their number of points
    std::vector<std::vector<double> > sums( n: K, value: std::vector<double>( n: points.getNumDimensions()));
    std::vector<int> nPoints( n: K);

    for (int iter = 1; iter<=iters; iter++)
    {
        //printf( "Iteration %d/%d \n", iter, iters);

        // Calculate each point's new cluster ID and accumulate the sums of each cluster's coordinates
        for (int p =0; p<points.getNumPoints(); p++)
        {
            int currentClusterId = points.getClusterIdByIndex( index: p);
            int nearestClusterId = getNearestClusterId( index: p);

            if (currentClusterId != nearestClusterId)
            {
                points.setClusterIdByIndex( index: p,  value: nearestClusterId);
            }
            for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                sums[nearestClusterId][dim] += points( pointIndex: p, dimensionIndex: dim);
            }
            nPoints[nearestClusterId]++;
        }

        // Recalculate the centroid of each cluster, and reset sums and nPoints
        for(int c=0; c<K ; c++) {
            for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                clusters[c].setCentroidByPos( pos: dim,  val: sums[c][dim]/nPoints[c]);
                sums[c][dim]=0;
            }
            nPoints[c]=0;
        }
    }
```

Figure 3. K-means++ sequential clustering

```
    std::vector<double>distances;
    std::vector<int>indices;
    // kmeans++ initialization for the other clusters
    for (int i = 1; i < K; i++){
        distances.clear();
        indices.clear();
        // find the closest existing cluster to each point, select the furthest
#pragma omp parallel default(none) shared(distances,i, points, clusters, indices)  num_threads(threads)
        {   std::vector<double>privateDistances;
            std::vector<int>privateIndices;
            #pragma omp for
            for (int p = 0; p < points.getNumPoints(); p++) {
                double dist = 0.0;
                // iterate through all the previously found centroids
                for (int j = 0; j < i; j++) {
                    double sum = 0.0;
                    for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                        sum += pow( x: clusters[j].getCentroidCoordByPos( pos: dim) - points( pointIndex: p,  dimensionIndex: dim),  y: 2.0);
                    }
                    dist += sqrt( x: sum);
                }
                privateDistances.push_back(dist);
                // collect the local indices to know the original index of each distance
                privateIndices.push_back(p);
            }
            // synchronize all the threads, collect all the distances and their original indices
            #pragma omp critical
            {
                distances.insert( position: distances.end(),  first: privateDistances.begin(),  last: privateDistances.end());
                indices.insert( position: indices.end(),  first: privateIndices.begin(),  last: privateIndices.end());
            }
        }
        // select the point that is the furthest from all the other clusters as a new centroid
        index = std::max_element( first: distances.begin(),  last: distances.end()) - distances.begin();
        points.setClusterIdByIndex( index: indices[index],  value: i);
        Cluster cluster( clusterId: i,  centroid: indexToCoordinates( index: indices[index]));
        clusters.push_back(cluster);
    }
```

Figure 4. K-means++ parallelized initialization

Each thread is assigned a subset of points to compute their distances independently. To synchronize them, a critical section is used after the distances calculations, where each thread stores its results in a shared variable. This shared variable is used to determine the point farthest from the previous centroids.
Additionally, a shared vector of indices is maintained to map each distance to its corresponding point's original index.

After the initialization, *runPar* parallelizes both loops in the clustering process (Fig. 5).
Each thread is assigned a subset of points and determines their new cluster ID assignments. The points' coordinates are then summed and stored in the shared matrix

```
    for (int iter = 1; iter<=iters; iter++)
    {   //printf( "Iteration %d/%d \n", iter, iters);

#pragma omp parallel default(none) shared(points, clusters, sums, nPoints) num_threads(threads)
        {   // Calculate each point's new cluster ID and accumulate the sums of each cluster's coordinates
            // each threads takes a part of the points
            #pragma omp for
            for (int p = 0; p < points.getNumPoints(); p++) {
                int currentClusterId = points.getClusterIdByIndex( index: p);
                int nearestClusterId = getNearestClusterId( index: p);

                if (currentClusterId != nearestClusterId) {
                    points.setClusterIdByIndex( index: p,  value: nearestClusterId);
                }
                for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                // synchronize access to the shared sums matrix
                    #pragma omp atomic update
                    sums[nearestClusterId][dim] += points( pointIndex: p,  dimensionIndex: dim);
                }
                // synchronize access to the shared nPoints vector
                #pragma omp atomic update
                nPoints[nearestClusterId]++;
            }

            // Recalculate the centroid of each cluster and reset sums and nPoints
            #pragma omp for
            for (int c = 0; c < K; c++) {
                for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                    clusters[c].setCentroidByPos( pos: dim,  val: sums[c][dim] / nPoints[c]);
                    sums[c][dim] = 0;
                }
                nPoints[c] = 0;
            }
        }
    }
```

Figure 5. K-means++ parallelized clustering

($number of clusters$ x $points' dimensions$) while the number of points assigned to each cluster is stored in a shared vector. To manage concurrent access to the shared matrix and vector, OpenMP's atomic update instruction is used for synchronization.
After the coordinate's sums and the number of points of each cluster have been calculated, the second for cycle is also parallelized. This time, there is no need to synchronize the threads, since each thread handles a different cluster, calculating the new centroid coordinates and resetting the shared variables independently.

runParPrivate has a different approach to synchronize access to the shared variables (Fig. 6).
Each thread maintains its own private copy of the sums matrix and the number of points vector. Synchronization occurs only after each thread completes its calculations, at which point the private variables are summed to the shared variables using an atomic update. This approach minimizes the use of atomic operations during computations, reducing potential interruptions between threads.

4.4. plotKmeans.py

This file includes the code that generates color-coded plots of all the points based on their final cluster IDs.
It is used to produce 2D visualizations of clustering (Fig. 1) to demonstrate the accuracy of the K-means algorithm.

4.5. testKmeansSoA.cpp

This is the executable file that compares the three different k-means implementations with different datasets,

```
    for (int iter = 1; iter<=iters; iter++)
    {
        //printf( "Iteration %d/%d \n", iter, iters);
#pragma omp parallel default(none) shared(points, clusters, sums, nPoints) num_threads(threads)
        {   // Each thread has its own private sums and nPoints
            std::vector<std::vector<double> > sumsPrivate( n: K, value: std::vector<double>( n: points.getNumDimensions()));
            std::vector<int> nPointsPrivate( n: K);

            // Calculate each point's new cluster ID and accumulate the private sums of each cluster's coordinates
            // each threads has a part of all the points
            #pragma omp for nowait
            for (int p = 0; p < points.getNumPoints(); p++) {
                int currentClusterId = points.getClusterIdByIndex( index: p);
                int nearestClusterId = getNearestClusterId( index: p);

                if (currentClusterId != nearestClusterId) {
                    points.setClusterIdByIndex( index: p, value: nearestClusterId);
                }
                for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                    sumsPrivate[nearestClusterId][dim] += points( pointIndex: p, dimensionIndex: dim);
                }
                nPointsPrivate[nearestClusterId]++;
            }

            // Accumulate all the private sums and nPoints, synchronize after each threads finishes accumulating the private sums
            for (int c = 0; c < K; c++) {
                for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                #pragma omp atomic update
                    sums[c][dim] += sumsPrivate[c][dim];
                }
            #pragma omp atomic update
                nPoints[c] += nPointsPrivate[c];
            }

            // wait until all the sums and nPoints have been calculated
            #pragma omp barrier

            // Recalculate the centroid of each cluster and reset sums and nPoints
            #pragma omp for
            for (int c = 0; c < K; c++) {
                for (int dim = 0; dim < points.getNumDimensions(); dim++) {
                    clusters[c].setCentroidByPos( pos: dim, val: sums[c][dim] / nPoints[c]);
                    sums[c][dim] = 0;
                }
                nPoints[c] = 0;
            }
        }
    }
```

Figure 6. K-means++ private parallelized clustering

numbers of clusters and threads. Each method is executed multiple times, and the average execution times and speedups are computed.

The results are saved in a txt file.

To validate the correctness of the parallel versions, the final cluster IDs assigned to each point by both the sequential and parallel K-means algorithms are compared.

5. Experiments and Results

All the following experiments were conducted on the SSH server 'papavero.dinfo.unifi' using CLion as the IDE and g++ as the compiler.

Unless otherwise specified, each experiment uses a default configuration of 64 threads, 8 clusters, and 100 clustering iterations.

5.1. Different number of clusters

The number of clusters has an impact on the initialization execution time. Each cluster, beyond the first, adds a for loop that iterates through all the points.

Fig. 7 displays the speedups of the two parallel K-means++ implementations as the number of clusters increases.
With fewer clusters, the shared sums matrix is smaller ($numberofclusters$ x $points'dimensions$). This means

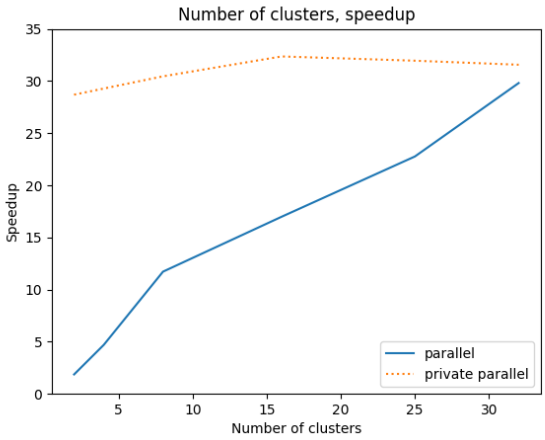

Figure 7. Speedups of the two parallel K-means++ implementations as the number of clusters changes

	100 000	500 000	1 Million
Parallel, 2D	4.92606	5.06141	6.46353
Private, 2D	25.0359	28.1786	29.8376
Parallel, 4D	6.69915	7.47719	6.75568
Private, 4D	28.658	29.4665	30.6216
Parallel, 8D	12.7733	8.79724	13.1395
Private, 8D	30.9999	30.9922	32.1643

Table 1. Speedups achieved by the two parallelized K-means++ methods as the dataset size and point dimensions change

that it is more likely that multiple threads attempt to access the same matrix cell with an atomic update. This makes the *runPar* method slower, a problem that the private version avoids.
As the number of clusters grows, the sequential method's workload also increases, making parallelization more beneficial as the overhead of thread management is outweighed by the reduction in execution time. Additionally, the shared sums matrix is larger and the chances of more than one threads wanting to access the same cell are lower. This is the reason why the performances of *runPar* method improve with the number of clusters. The private version of the parallelized K-means has the first advantage, but experiences longer synchronization times as more clusters lead to larger shared variables and longer for loops.

5.2. Different datasets sizes and points dimensions

Tab. 1 shows the speedups achieved by the two parallelized K-means++ methods (one using more atomic updates and the other using thread-private variables) as the dataset size and point dimensions vary.

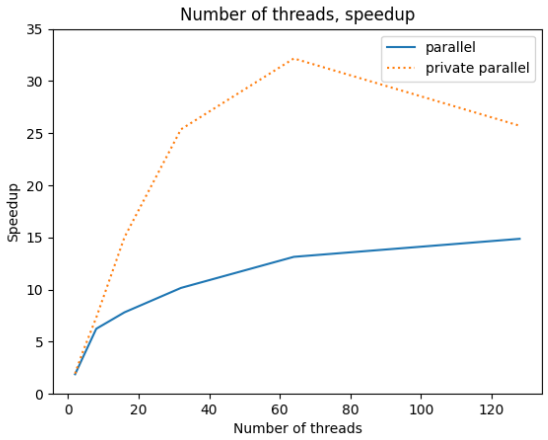

Figure 8. Speedups of the two parallelized K-means++ methods as the number of threads changes

Three main trends are observable:

1. The private parallel version consistently outperforms the other method. This is primarily due to the use of 8 clusters, which causes frequent thread interruptions when updating shared variables in the non-private version, as explained earlier.

2. As the point's dimensions grow, so does the speedup. Points with more coordinates imply more operations. This reduces the relative impact of thread initialization time on the overall execution, as more time is spent in the parallelized sections of the search algorithm. Moreover, smaller dimension imply smaller shared variables, increasing the change of threads interrupting each other.

3. Larger datasets produce similar effects, for the same reasons as with higher dimensions. There could be multiple reasons that explain the few cases where an increase in the number of points does not mean a higher speedup. Each method is run multiple times, different runs might experience more or fewer conflicts between threads. The same goes for different datasets. For instance, a dataset where most points belong to the same cluster can lead to slower speedups due to more threads attempting to access the same memory location simultaneously.

5.3. Different number of threads

The final experiment changes the number of threads used to search for 8 clusters in the dataset that contains 1 million 8-dimentional points.

In Fig. 8 the private method shows an almost linear speedup increase as the number of threads rises, peaking at 64 threads. Beyond this point, speedup plateaus, indicating that 64 threads sufficiently utilize all available CPU resources. Adding more threads leads only introduces additional overhead for thread creation, management, termination, and synchronization, which offsets the benefits of a faster execution.
The results are different with the other parallelized method. While more threads do reduce execution time, they also increase the likelihood of threads interrupting each other. As a result, speedup continues to rise gradually, but at a slower rate as the thread count grows.