Parallel Computing: Implementation of K-Means Algorithm using OpenMP

Annalisa Ciuchi
E-mail address
annalisa.ciuchi@edu.unifi.it

Abstract

In this work, the implementation of K-Means algorithm using OpenMP is presented. The main objective is to evaluate the effectiveness of the parallelized version on a CPU in improving performance compared to a sequential implementation. After a brief introduction to the functioning of K-Means Algorithm, the techniques adopted for parallelization are described. Results were then obtained on datasets of various sizes and complexities. The performance was analyzed, achieving performance improvements in terms of speedup.

1. Introduction to K-Means Algorithm

K-Means is an unsupervised clustering algorithm aimed at partitioning a dataset into k clusters, minimizing the within-cluster variance. The goal is to divide a dataset into K groups (clusters), where each data point belongs to the cluster with the nearest centroid. This process helps organize the data to identify underlying patterns or structures. K-Means is based on an iterative approach and can be summarized by the following pseudocode.

1.1. Pseudocode

Algorithm 1 K-Means

```
1: Select $K$ points as initial centroids.
2: repeat
3:     Form $K$ clusters by assigning each point to the nearest centroid.
4:     Recalculate the centroids of each cluster.
5: until Centroids changes becomes negligible.
```

1.2. Fundamental steps of the K-Mean algorthim

The K-Means algorithm involves the following fundamental steps:

1. **Initialization:** Randomly select K data points from the dataset as the initial centroids.
2. **Assignment:** For each data point in the dataset:
   - Calculate the distance to all centroids (e.g., Euclidean distance).
   - Assign the data point to the cluster represented by the nearest centroid.
3. **Update:** Recalculate the centroid of each cluster by computing the mean of all data points assigned to that cluster. The new centroid represents the updated cluster center.
4. **Convergence Check:** Repeat the *Assignment* and *Update* steps until one of the following conditions is met:
   - A predefined maximum number of iterations is reached.
   - Assignemnts no longer change.
   - Cluster centroids changes become negligible.

2. Sequential Implementation

The implementation is written in C++ and in the following sections it's been described.

2.1. Representation of Data Points

Data points are represented using a custom-defined structure called Punto (Point), which encapsulates the following:

- **Coordinates**: Stored as a std::vector<double> to represent multidimensional data.
- **Cluster ID**: An integer (int) that indicates the cluster to which the point is assigned.

2.2. Representation of Datasets and Centroids

- **Dataset**: A std::vector<Punto> is used to store the dataset.
- **Centroids**: Represented as a separate std::vector<Punto> initialized randomly.

2.3. Defined Functions

The sequential version of the K-Means algorithm includes several functions, each with a specific purpose in the clustering process. Below is a description of each function along with its implementation:

2.3.1 Sequential function

The function `sequential_kmeans` implements the K-Means clustering algorithm in a sequential manner. Below is a detailed explanation of its structure and logic.

Input Parameters:

- ds: A vector of Punto objects representing the dataset to be clustered.
- centroidi: A vector of Punto objects representing the initial centroids of the clusters.
- k: The number of clusters.

Steps:

- Initialization:
  - The iteration counter (iter) is set to 0.
  - A convergence flag (convergenza) is initialized to false.
  - A flag for the first iteration (prima_iter) is set to true.
  - A copy of the dataset from the previous iteration (precedente_ds) is created for comparison.
  - The number of points and their dimensionality are determined.
- Main Loop: The algorithm iterates until convergence.
  - Each point in the dataset is assigned to the nearest centroid based on the minimum distance using the function `assegnamento()`
  - Centroids are recalculated based on the mean position of the points assigned to each cluster using the function `aggiornamentoCentroidi()`
  - Convergence check: If it's not the first iteration, the algorithm checks whether the cluster assignments have changed compared to the previous iteration. If no changes occur, the algorithm converges. If there is no convengence, the previous dataset (precedente_ds) is updated to the current dataset, and prima_iter is set to false.

These functions are described in the next sections.

- Result: The function terminates when the cluster assignments stabilize (no changes between iterations) and it returns a tuple containing the updated dataset and final centroids.

Code:

```
std::tuple<std::vector<Punto>, std::vector<Punto>>
      sequential_kmeans(std::vector<Punto> ds,
                        std::vector<Punto> centroidi, int k) {
    int iter = 0;
    bool convergenza = false;
    bool prima_iter = true;
    std::vector<Punto> precedente_ds;

    const auto numPunti = ds.size();
    const auto dimPunti = ds[0].dimensioni.size();

    int etichetta_cluster;
    double distanza, minDistanza;
    std::vector<int> count(k);

    while (!convergenza) {
        assegnamento(ds, centroidi, k, numPunti, dimPunti,
      etichetta_cluster, distanza, minDistanza);
        aggiornamentoCentroidi(ds, centroidi, k, numPunti, dimPunti,
      count);

        iter++;

        if (!prima_iter && controlloCluster(ds, precedente_ds,
      numPunti)) {
            convergenza = true;
        } else {
            precedente_ds = ds;
            prima_iter = false;
        }
    }

    return {ds, centroidi};
}
```

Listing 1. Sequential K-Means Function

2.3.2 Cluster Assignment (`assegnamento`)

This function assigns each data point in the dataset to the nearest centroid.

Description:

- Iterates through all data points and centroids.
- Computes the Euclidean distance between each point and each centroid.
- Updates the cluster ID of each point based on the closest centroid.

Code:

```
void assegnamento(std::vector<Punto> &ds, std::vector<Punto> &centroidi,
                  int k, const unsigned long numPunti,
                  const unsigned long dimPunti, int etichetta_cluster,
                  double distanza, double minDistanza) {

    for (int i = 0; i < numPunti; i++) {
        minDistanza = std::numeric_limits<double>::max();
        for (int j = 0; j < k; j++) {
            distanza = 0;
            for (int h = 0; h < dimPunti; h++) {
                distanza += pow(ds[i].dimensioni[h] -
                centroidi[j].dimensioni[h], 2);
            }
            distanza = sqrt(distanza);
            if (distanza < minDistanza) {
                minDistanza = distanza;
                etichetta_cluster = j;
            }
        }
        ds[i].cluster_id = etichetta_cluster;
    }
}
```

Listing 2. Cluster Assignment Function

2.3.3 Centroid update (`aggiornamentoCentroidi`)

This function updates the positions of centroids recalculating each centroid's coordinates based on the mean of all data points assigned to the corresponding cluster.

Steps:

- **Initialize the count vector**: All elements of `count` (recalling the number of points assigned to each centroid) are set to 0 using std::fill (to reset the point counts for each cluster).
- **Reset Centroid Coordinates**: For each centroid, all dimensions of its `dimensioni` vector are set to 0 in order to prepare to calculate new centroid positions.
- **Accumulate Data Points into Centroids**: Iterate through each point in the dataset (`ds`): For each dimension h of the point, add the coordinate to the corresponding centroid's dimention (based on the point's `cluster_id`). Simultaneously, increment the count of points assigned to the centroid (`count[cluster_id]`).
- **Calculate New Centroid Positions**: For each centroid, calculate the mean for each dimension by divideing the sum of coordinates by the number of points assigned to the cluster (stored in `count`).

Code:

```
void aggiornamentoCentroidi(std::vector<Punto> &ds, std::vector<Punto>
        &centroidi, int k, const unsigned long numPunti, const unsigned
        long dimPunti, std::vector<int> &count) {

    std::fill(count.begin(), count.end(), 0);

    for (int j = 0; j < k; j++) {
        std::fill(centroidi[j].dimensioni.begin(),
      centroidi[j].dimensioni.end(), 0);
    }

    for (int i = 0; i < numPunti; i++) {
        for (int h = 0; h < dimPunti; h++) {
            centroidi[ds[i].cluster_id].dimensioni[h] +=
      ds[i].dimensioni[h];
        }
        count[ds[i].cluster_id]++;
    }

    for (int j = 0; j < k; j++) {
        for (int h = 0; h < dimPunti; h++) {
            centroidi[j].dimensioni[h] /= count[j];
        }
    }
}
```

Listing 3. Centroids Updates Function

2.4. Convergence Check `controlloCluster`

During the convergence check phase this function checks whether the cluster assignments of points in the current dataset (`ds`) are the same as those in the previous dataset (`precedente_ds`).

Description:

- The function loops through each point in the dataset.
- For each point, it compares the `cluster_id` in the current dataset (`ds[i]cluster_id`) with the `cluster_id` in the previous dataset (`precedente_ds[i]cluster_id`).
- If any point's `cluster_id` differs between the two datasets, the function immediately returns false, indicating that the cluster assignments have changed.
- If all points have the same `cluster_id` in both datasets, the function returns true, meaning the cluster assignments are unchanged.

Code:

```
bool controlloCluster(std::vector<Punto> ds, std::vector<Punto>
      precedente_ds, int numPunti) {
    for (int i = 0; i < numPunti; i++) {
        if (ds[i].cluster_id != precedente_ds[i].cluster_id) {
            return false;
        }
    }
    return true;
}
```

Listing 4. Convergence Check Function

3. Parallel Implementation

The nature of K-Means makes it well-suited for parallelization, as many of its operations can be performed independently for each point or cluster.

- For example, in the case of assignment, the operations performed for each point are independent of one another. Therefore, starting from each point, the distances to each cluster can be calculated, and the point can be assigned to the closest cluster in parallel.
- Similarly, for the update step, the sum of the coordinates of the points assigned to each cluster can be calculated in parallel. Each coordinate of the centroids is then updated independently by dividing the accumulated sum by the number of points.

3.1. Use of OpenMP Directives

3.1.1 #pragma omp parallel

Compared to the previously shown sequential function (`sequential_kmeans`), in the parallel version of K-Means, a parallel section was added within the while loop using the directive `#pragma omp parallel`.

```
. . .
    while (!convergenza) {
#pragma omp parallel num_threads(8) default(none) \
    private(pDist, pMinDist, etichetta_cluster) \
    shared(numPunti, k, dimPunti, ds, centroidi, count)
. . .
```

Listing 5. pragma omp parallel

This directive creates a group of threads that work in parallel. Within the specified block, each thread executes a portion of the assigned work.

Thread-local variables (declared as private) have been specified to ensure that each thread can operate independently and avoids conflicts.
Shared variables (declared as shared) are also specified to ensure that all threads can access common data needed for computation.
In the directive, a specific scheduling type was not specified, as the default scheduling (static) was chosen. This decision was made because the workload is balanced (the work assigned to each thread is evenly distributed), given that the operations within the loop have similar execution times and there are no significant differences in the work performed between iterations.

3.1.2 #pragma omp for

This directive automatically divides a loop among the available threads. As each point can be analyzed independently to determine the closest cluster, dividing the loop allows the threads to process different points simultaneously. For this reason, the directive was used in the outermost for loop related to calculating the closest cluster, assigning a subset of points to each thread. In addition, within a #pragma omp parallel block, the #pragma omp for clause divides a loop among the already created threads, without needing to recreate the thread team. This approach eliminates the overhead associated with repeatedly creating and destroying threads, which would be inefficient.

```
...
#pragma omp for
            for (int i = 0; i < numPunti; i++) {
                pMinDist = std::numeric_limits<double>::max();
                for (int j = 0; j < k; j++) {

                    pDist = 0;
                    for (int h = 0; h < dimPunti; h++) {
                        pDist += pow(ds[i].dimensioni[h] -
       centroidi[j].dimensioni[h], 2);
                    }
                    pDist = sqrt(pDist);
                    if (pDist < pMinDist) {
                        pMinDist = pDist;
                        etichetta_cluster = j;
                    }
                }
                ds[i].cluster_id = etichetta_cluster;
            }
...
```

Listing 6. pragma omp for - assignment

Another use is during the initialization/reset of the shared variable related to the number of points in each cluster. This is because resetting the point count for each cluster is a simple and independent operation for each cluster.

```
...
    #pragma omp for
            for (int j = 0; j < k; j++) {
                count[j] = 0;
            }
...
```

Listing 7. pragma omp for - reset

Finally this directive is also been used in the outermost for loop during the update phase.

```
...
#pragma omp for
            for (int i = 0; i < numPunti; i++) {
                for (int h = 0; h < dimPunti; h++) {
...
```

Listing 8. pragma omp for - update

3.1.3 #pragma omp atomic

The #pragma omp atomic directive ensures that an operation on a shared variable is executed atomically, preventing race conditions. Multiple threads may attempt to update the same centroid coordinate simultaneously. This directive was used in the update phase both to accumulate the coordinates of the centroids and to update the point counter. Using atomic ensures that updates are performed one at a time, preserving data consistency.

```
#pragma omp atomic
                    centroidi[ds[i].cluster_id].dimensioni[h] +=
       ds[i].dimensioni[h];
                }
#pragma omp atomic
                count[ds[i].cluster_id]++;
            }
```

Listing 9. pragma omp atomic

The division, used in the final phase of calculating the new centroids, is a read-modify-write operation. Therefore, it is necessary to use atomic, as race conditions may occur if multiple threads attempt to execute this sequence simultaneously.

```
...
#pragma omp atomic
                    centroidi[j].dimensioni[h] /= count[j];

...
```

Listing 10. pragma omp atomic

3.1.4 #pragma omp collapse

The #pragma omp collapse(n) directive combines multiple nested loops into a single "virtual" loop to enable better parallelization. This directive was used in the final step for calculating the coordinates of the new centroids. Instead of parallelizing only the outer loop (over clusters), the collapse directive allows parallelization of the inner loop (over dimensions) as well, increasing the total number of iterations to distribute among the threads.

```
#pragma omp for collapse(2)
            for (int j = 0; j < k; j++) {
                for (int h = 0; h < dimPunti; h++) {
#pragma omp atomic
                    centroidi[j].dimensioni[h] /= count[j];
                }
            }
```

Listing 11. pragma omp collapse

It is also been used during the reset phase of the centroids

```
#pragma omp for collapse(2)
            for (int j = 0; j < k; j++) {
                for (int h = 0; h < dimPunti; h++) {
                    centroidi[j].dimensioni[h] = 0;
                }
            }
```

Listing 12. pragma omp collapse

3.1.5 Convergence check

The convergence check is not parallelized, as parallelizing it would introduce greater overhead than the benefits gained. This operation is computationally lightweight and linear with respect to the number of points. Also its parallelization would require iterating over the entire vector regardless, to decide the final result of the check. In contrast, the sequential version can stop as soon as the first difference between ds and precedente_ds is found.

```
...
if (!primaIter && controlloClusterOmp(ds, precedente_ds, numPunti)) {
            convergenza = true;
        } else {
            precedente_ds = ds;
            primaIter = false;
        }
...
```

Listing 13. convergence

```

bool controlloClusterOmp(std::vector<Punto> ds, std::vector<Punto>
       precedente_ds, int numPunti){
    for (int i = 0; i < numPunti; i++){
        if(ds[i].cluster_id != precedente_ds[i].cluster_id){
            return false;
        }
    }
    return true;
}
```

Listing 14. convergence

3.2. Code

```
std::tuple<std::vector<Punto>,std::vector<Punto>>
       openmp_kmeans(std::vector<Punto> ds, std::vector<Punto> centroidi,
       int k) {

    bool convergenza = false, primaIter = true;
    std::vector<Punto> precedente_ds;
    //Numero di punti del dataset
    auto numPunti = ds.size();
    //Dimensione dei punti
    auto dimPunti = ds[0].dimensioni.size();

    int etichetta_cluster;
    double pMinDist, pDist;
    std::vector<int> count(k);

    while (!convergenza) {

#pragma omp parallel num_threads(8) default(none) \
    private(pDist, pMinDist, etichetta_cluster)\
    shared(numPunti, k, dimPunti, ds, centroidi, count)
        {
#pragma omp for
            for (int i = 0; i < numPunti; i++) {
                pMinDist = std::numeric_limits<double>::max();
                for (int j = 0; j < k; j++) {

                    pDist = 0;
                    for (int h = 0; h < dimPunti; h++) {
                        pDist += pow(ds[i].dimensioni[h] -
       centroidi[j].dimensioni[h], 2);
                    }
                    pDist = sqrt(pDist);
                    if (pDist < pMinDist) {
                        pMinDist = pDist;
                        etichetta_cluster = j;
                    }
                }
                ds[i].cluster_id = etichetta_cluster;
            }

//reset variabile shared num punti cluster count
#pragma omp for
            for (int j = 0; j < k; j++) {
                count[j] = 0;
            }
//reset dei centroidi
#pragma omp for collapse(2)
            for (int j = 0; j < k; j++) {
                for (int h = 0; h < dimPunti; h++) {
                    centroidi[j].dimensioni[h] = 0;
                }
            }
//calcolo somme e num totale punti in ogni cluster
#pragma omp for
            for (int i = 0; i < numPunti; i++) {
                for (int h = 0; h < dimPunti; h++) {
#pragma omp atomic
                    centroidi[ds[i].cluster_id].dimensioni[h] +=
       ds[i].dimensioni[h];
                }
#pragma omp atomic
                count[ds[i].cluster_id]++;
            }

#pragma omp for collapse(2)
            for (int j = 0; j < k; j++) {
                for (int h = 0; h < dimPunti; h++) {
#pragma omp atomic
                    centroidi[j].dimensioni[h] /= count[j];
                }
            }
        }

        if (!primaIter && controlloClusterOmp(ds, precedente_ds,
       numPunti)) {
            convergenza = true;
        } else {
            precedente_ds = ds;
            primaIter = false;
        }
    }
    return {ds,centroidi};
}
```

Listing 15. parallel_kmeans

4. Tests and Results

The evaluation of the K-Means algorithm, implemented in both sequential and parallel versions using OpenMP, was conducted on an Intel i9-13900HX CPU.
Initial centroids were randomly selected and identical for both versions to ensure a fair comparison.

4.1. Datasets

The tests were performed on datasets generated both randomly and synthetically, as well as on real-world datasets.

- The random datasets were generated by varying the number of points and dimensions, while the algorithm was tested with varying numbers of clusters (10, 100, and 1000).
- The real-world dataset used was letters. This dataset represents the 26 letters of the English alphabet.
- synthetic datasets included a1, a2, a3, birch1, and s1.
  - a1, a2, a3: Synthetic 2-d data with increasing number of clusters (k). There are 150 vectors per cluster.
  - birch1: Synthetic 2-d data with N=100,000 vectors and k=100 clusters in regular grid structure.
  - s1: s-set contains synthetic 2-d data with N=5000 vectors and k=15 Gaussian clusters with different degree of cluster overlap. s1 contains well-separated clusters with uniform density.

  These Datasets can be found here http://cs.joensuu.fi/sipu/datasets/

4.2. Results

The execution time and speed-up relative to the sequential version were measured.
The results indicate that the speed-up increases with the number of clusters and dimensions due to the growing workload. The speedup generally increases with k and this is expected because searching for a higher number of clusters (higher k values) increases computational complexity, providing the threads with more work to execute in parallel.

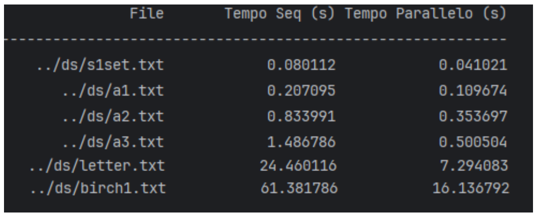
In particular we can notice that for larger datasets and increasing number of clusters, with 4 threads the speedup approaches the theoretical maximum. This occurs because the computational workload is sufficiently high to fully utilize the available threads, reducing the relative impact of overhead.

Datasets:	1000x10			1000x100			10000x10			10000x100		
k	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup
10	0.1030	0.0457	2.2520	0.2925	0.1328	2.2023	9.3354	3.8867	2.4018	13.8604	5.8996	2.3493
100	0.2755	0.0919	2.0973	1.1317	0.3549	3.1882	63.4136	17.7199	3.5787	128.9821	38.2602	3.3712
1000	0.4588	0.1281	3.5802	1.7881	0.5319	3.3617	95.6736	28.5661	3.3481	189.5835	59.0072	3.2128

Table 1. Random datasets with times (sequential, parallel) and Speedup using 4 threads.

Dataset	**Dimensions**	k	**Speedup**
s1set	5000×2	15	1.953774968
a1	150×2	20	1.887787568
a2	150×2	35	2.357968369
a3	150×2	50	2.970763572
letters	20000×16	26	3.353547211
birch1	100000×2	100	3.803494051

Table 2. Synthetic and real datasets speedup 4 threads



```
             File       Tempo Seq (s) Tempo Parallelo (s)
------------------------------------------------------------
    ../ds/s1set.txt           0.080112            0.041021
       ../ds/a1.txt           0.207095            0.109674
       ../ds/a2.txt           0.833991            0.353697
       ../ds/a3.txt           1.486786            0.500504
   ../ds/letter.txt          24.460116            7.294083
   ../ds/birch1.txt          61.381786           16.136792
```

Figure 1. Synthetic datasets execution time

However If the goal is to maximize computational speed for large datasets and high k values, using 8 threads gives the best results [Table 3].

Datasets:	1000x10	1000x100	1000x1000	10000x10	10000x100
k=10	1.923171	2.499623	1.885809	3.989404	3.096025
k=100	3.062129	3.739437	3.606545	4.878448	5.268425
k=1000	4.567809	4.561711	5.514172	5.238903	4.699418

Table 3. Random datasets Speedup with 8 threads

Instead, for smaller-scale problems, 4 threads might already be sufficient, limiting resource consumption without significantly sacrificing performance. That mean If the dataset is very small or k is low, the difference between 4 and 8 threads might be marginal, making 4 threads more efficient in terms of cost-benefit ratio.

Both random, synthetic and real-world datasets show consistent trends in execution time and speed-up, validating the robustness of the parallel approach.