K-means – Parallel Programming with OpenMP

Architectures and Platforms for AI
Module 2

Parsa Dahesh
parsa.dahesh@studio.unibo.it

Academic Year 2021/2022

Master degree in Artificial Intelligence
Alma Mater Studiorum - University of Bologna

1 K-means variants

In the literature there are several variants of the k-means algorithm: in this project were discussed the standard k-means and the k-medians implementations.

1.1 Standard variant

Given a dataset and the number of clusters, the algorithm executes the following steps:

1. Randomly initialize the centroid of each cluster;
2. Assign each data-point to the closest centroid by computing the euclidean distance;
3. Find the new centroids by computing the mean of the data-points of each cluster;
4. Repeat from step 2 until the final conditions are met, i.e. when the algorithm reaches a maximum number of iterations or there are no updates in the centroids' values in a single iteration.

Although the standard version is relatively fast and easy to interpret, it is quite sensitive to outliers. Therefore, a second version has been tested: the *k-medians*.

1.2 K-medians variant

The k-medians variant is very similar to the standard version but in the 3^{rd} step it computes the geometric median instead of the mean.

The main idea behind the *geometric median* is that given a set of point $x_0, ..., x_n \in \mathbf{R}^m$, find the point $y \in x_1, ..., x_n$ such that:

$$y = \underset{z}{\operatorname{argmin}} \sum_{i=0}^{n} \sum_{j=0}^{m} (x_i^j - z^j)^2$$

with $z \in x_1, ..., x_n$. In other words, the objective is to find a point y that minimizes the sum of the euclidean distances to the other points of the set.

2 Parallelization with OpenMP

In this chapter, we will focus on the main sections where a parallelization strategy was used.

2.1 Standard variant

2.1.1 Variables

The standard variant function `standard_kmeans` is defined as follows:

```
double** standard_kmeans(double** dataset, int dataset_size, int num_features, int
↪  clusters, int max_iterations, int* assignments, int early_stop)
```

and returns as output:

- `double** centroids`, a $clusters \times num\_features$ vector that represents the computed centroids;
- `int* assignments`, a vector of size $dataset\_size$ that returns the assignment of each data-point.

Other variables that are used inside the algorithm and will be later used in the computations are:

- `double** prev_centroids`, represents the centroids found in the previous iteration;
- `int* counter`, a vector of size $clusters$ that counts the number of data-points per cluster;
- `int iteration`, keeps track of the iteration number;
- `int updates`, counts the number of assignment updates in a single iteration (used for early-stopping).

2.1.2 Step 1: centroids initialization

```
double centroid_min = min_matrix(dataset, dataset_size, num_features);
double centroid_max = max_matrix(dataset, dataset_size, num_features);

#pragma omp parallel for collapse(2)
for (int i = 0; i < clusters; i++)
    for (int j = 0; j < num_features; j++)
        centroids[i][j] = random_double(centroid_min, centroid_max);
```

The centroids' initialization follows the *embarrassingly parallel pattern* and the for loops are collapsed. This is done because the assignment of `centroids[i][j]` does not require any communication with other tasks or threads.

In order to compute `centroid_min`, the following code is used:

```
double min_value;

#pragma omp parallel for collapse(2) reduction(min: min_value)
for (int i = 0; i<rows; i++)
    for (int j = 0; j<columns; j++)
        if (array[i][j] < min_value)
            min_value = array[i][j];
```

In this case, I used the *min-reduction pattern* to find the minimum value of the dataset. Respectively, the same reasoning is applied to compute `centroid_max`.

2.1.3 Step 2: assign each data-point to a centroid

```
int updates = 0;
```

```
#pragma omp parallel for reduction(+: updates)
for (int i = 0; i < dataset_size; i++) {
    int cluster_index = find_closest_centroid(dataset[i], prev_centroids, clusters,
↪   num_features);

    if (cluster_index != assignments[i]) updates += 1; // early-stopping
    assignments[i] = cluster_index;
}
```

The computations of the variables `assignment` and `update` are performed in the same for loop with the addition of *sum-reduction pattern* on the variable `update`.

We also have to count the number of data-points belonging to each cluster. To do so, the following code was used:

```
#pragma omp parallel for
for (int i = 0; i < clusters; i++){
    int my_counter = 0;

    #pragma omp parallel for reduction(+: my_counter)
    for (int j = 0; j < dataset_size; j++)
        if (assignments[j] == i) my_counter += 1;

    counter[i] = my_counter;
}
```

Besides performing the computation for each cluster in a parallel manner, it has also been used the *sum-reduction pattern* on the variable `my_counter` to count the number of data-points per cluster.

2.1.4 Step 3: compute the new centroids

```
#pragma omp parallel for schedule(dynamic,1) collapse(2)
for (int j = 0; j < num_features; j++)
    for (int k = 0; k < clusters; k++) {
        if (counter[k] <= 0) continue;

        double sum = 0.0;
        #pragma omp parallel for reduction(+: sum)
        for (int i = 0; i < dataset_size; i++)
            if (assignments[i] == k)
                sum += dataset[i][j];

        centroids[k][j] = sum/counter[k];
    }
```

To compute the value of the new centroids, we first collapse the for loops iterating over the features and the clusters. A *dynamic scheduling* is used to overcome unbalanced workloads caused by the `if` condition. Then, we can focus on the sum of the data-points for each feature

by exploiting the *sum-reduction pattern* over the variable `sum`. Once computed, this variable is divided by the counter of its cluster and assigned to the corresponding centroid's feature.

2.2 K-medians variant

2.2.1 Variables

The signature of `median_kmeans` is the same as `standard_kmeans` (see section 2.1.1). The same applies to the variables that are used in the algorithm, with the addition of `double*** data_assignments`, a $clusters \times data\_size \times num\_features$ vector that saves the data-points of each cluster. This variable was introduced to ease the computation of the median.

Note: steps 1 and 2 of the algorithm will be skipped since they are very similar to the k_means variant and do not introduce major changes in the parallelization strategies.

2.2.2 Step 3: compute the centroids

First of all, we need to define the function that is used to compute the sum of the distances between a single `point` and all the other `points`. As usual, we exploit the *sum-reduction pattern* on the variable `sum`. The code is implemented as follows:

```
double distance_sum(double* point, double** points, int size, int dim) {
    double sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < size; i++)
        sum += euclidean_distance(point, points[i], dim);

    return sum;
}
```

We can finally compute the geometric median for each cluster:

```
for (int i = 0; i< clusters; i++){
    double min_dist = INFINITY;

    double my_dist;
    int index;

    #pragma omp parallel private(my_dist, index)
    {
        #pragma omp for reduction(min:min_dist)
        for (int j = 0; j < counter[i]; j++) {
            double distance = distance_sum(data_assignments[i][j],
↪   data_assignments[i], counter[i], num_features);

            if (distance < min_dist) {
                min_dist = distance;
                my_dist = distance;
                index = j;
```

```
            }
        }

        #pragma critical
        if (my_dist == min_dist)
            memcpy(centroids[i], data_assignments[i][index], num_features *
↪   sizeof(double));
    }
}
```

Being an argmin function, using the *reduction pattern* is not enough. Besides computing the minimum distance (stored in a shared variable `min_distance`), we also have to store the corresponding data-point. To do so, each thread will have the local variables `my_dist` and `index` that store respectively the minimum distance found by that single thread and the index of the data-point.

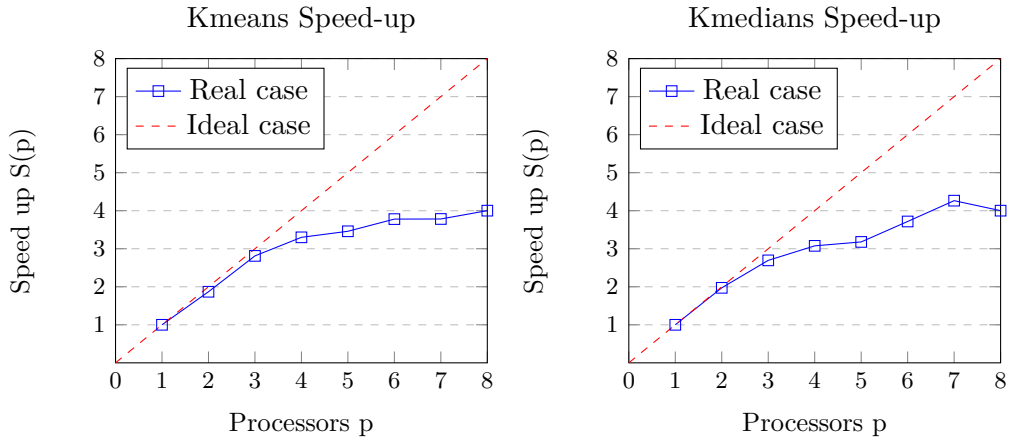
Then, in a `#pragma critical` section, we will compare the variables `my_dist` and `min_dist` to find the thread that is storing the correct data-point index that corresponds to the geometric median.

3 Results

The algorithms were tested on a 1500×4 float dataset and run with the following configuration: Intel i7-6700HQ Quad-core; RAM: 8GB; OS: Windows 10 – Ubuntu 22.04 WSL

The values in the graphs have been computed by averaging the results of 3 tests per number of processors.

3.1 Strong scaling

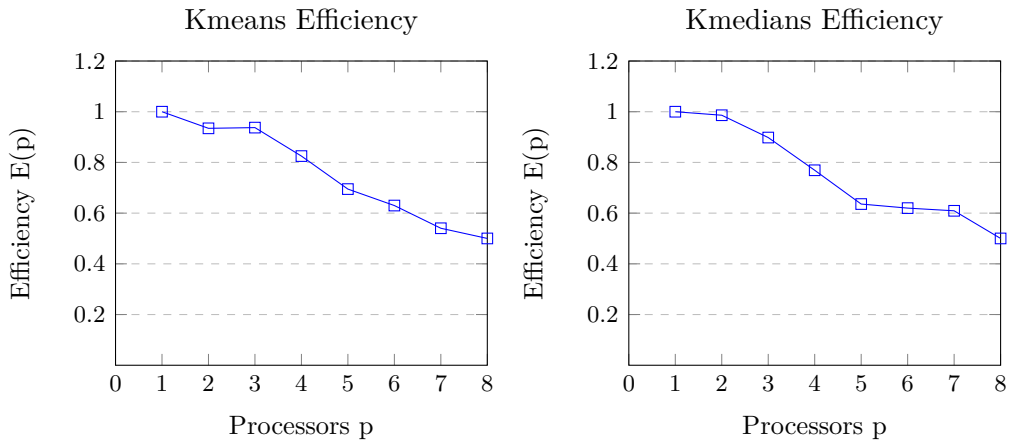


The values in the graphs have been obtained with the following equations:

$$S(p) = \frac{T_{parallel}(1)}{T_{parallel}(p)} \qquad E(p) = \frac{T_{parallel}(1)}{p * T_{parallel}(p)}$$

where $T_{parallel}(p)$ indicates the execution time of the parallel program with p processors.

By analyzing the speed-up and the strong scaling efficiency of the algorithms, we can see a high efficiency (≥ 0.8) up to 4 processors, with slow decay from 5 threads, although the standard *kmeans* seems to be more efficient before reaching hyper-threading.

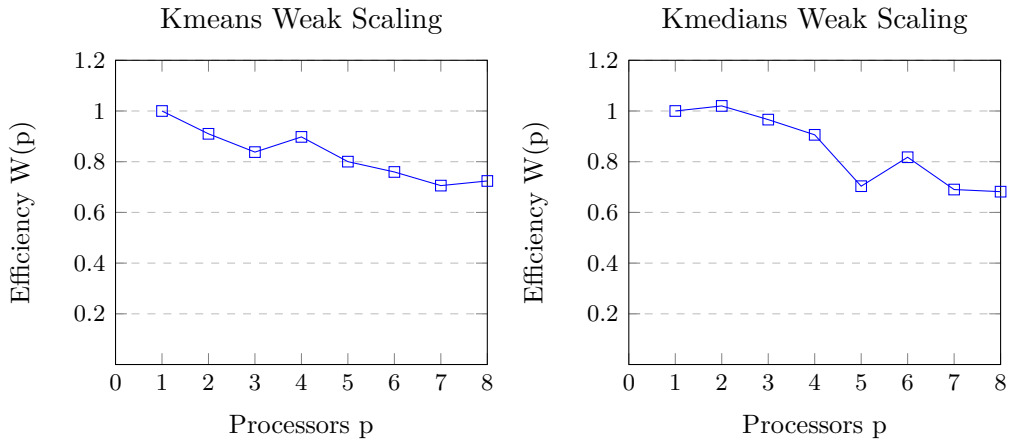


3.2 Weak scaling

First of all, we have to analyze the complexity of the two algorithms. In the *kmeans* case, we have a complexity of $O(N^4)$ with N being the dataset size, and the input size is computed as $\sqrt{p}N \times \sqrt{p}M$ with M being the number of features.

While in the *kmedians* case, the complexity is $O(N^5)$ and the input size is computed as $\sqrt[3]{p}N \times \sqrt[3]{p}M$. To define these equations, we have assumed that the number of iterations and cluster is constant, and we can only adjust the dimensions of the input dataset.

The weak scaling efficiency has been computed with $W(p) = T_1/T_p$ where T_p represents the time required to complete p work units with p processors



We can observe a high efficiency (≥ 0.8) before dropping with a number of threads larger than 4 although the *kmedians* variant seems to be more efficient before reaching hyper-threading.