K-Means++

Parallelization with OpenMP

Mirko Bicchierai

Firenze, 5/7/2024

K-Means algorithm, Introduction

Algorithm steps

Initialization :

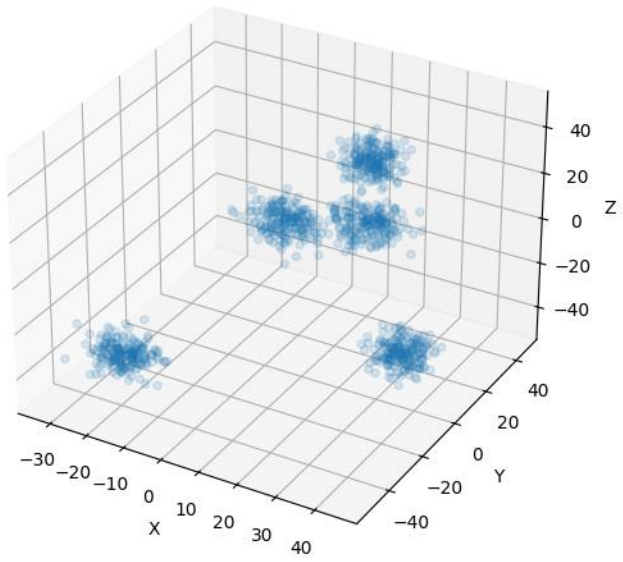
a) **Random,** Start by selecting k random point from the dataset.

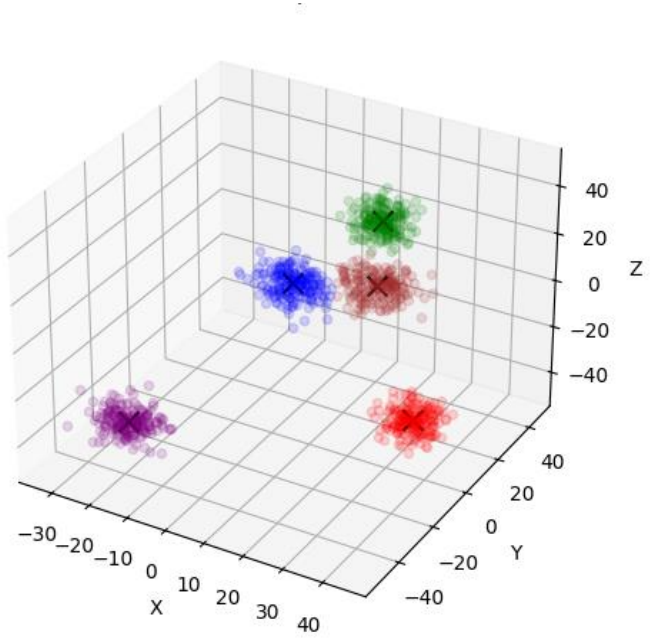
b) **K-Means++:**
   - **First Centroid,** Select the first centroid randomly from the dataset.
   - **Subsequent Centroids**, For each subsequent centroid, select a point from the dataset with a probability proportional to its squared distance from the nearest already chosen centroid.

1. **Assignment,** Assign each data point to the nearest cluster.
2. **Update centroids,** Calculate the new cluster centroids by computing the mean of all data points assigned to each cluster.
3. **Repeat,** until the centroids no longer move compared to the previous iteration.

K-Means algorithm, output

An example of kmeans algotithm output





Data genereation process

To generate the various datasets used later to test this algorithm, the '*make_blobs*' function from the '**sklearn.datasets**' library was utilized.

- For these tests, we utilized datasets generated with varying values of n = (100, 1000, ….., 10000000)
- For each n: k = (3, 5, 10, 15, 20, 25, 30, 40, 50)

```
def generate_points(n, k):
    X, _ = make_blobs(n_samples=n, n_features=3, centers=k, center_box=(-bounds, bounds), cluster_std=4, shuffle=True)
    return X
```

Hardware used and testing performance methods

- All the experiments were performed on 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz (Mobile) (8 CPU cores and 16 threads) and NVIDIA GeForce RTX 3050 Ti (Mobile)
- All the results reported below regarding the obtained speedup are calculated as the average of 100 executions of the sequential algorithm divided by the average of 100 executions of the parallel algorithm. This approach is used to ensure the most accurate results possible.
- This is except for n=10000000, where 100 executions were unmanageable in terms of time.

K-Means++ inizialization, implementation

Sequential algorithm

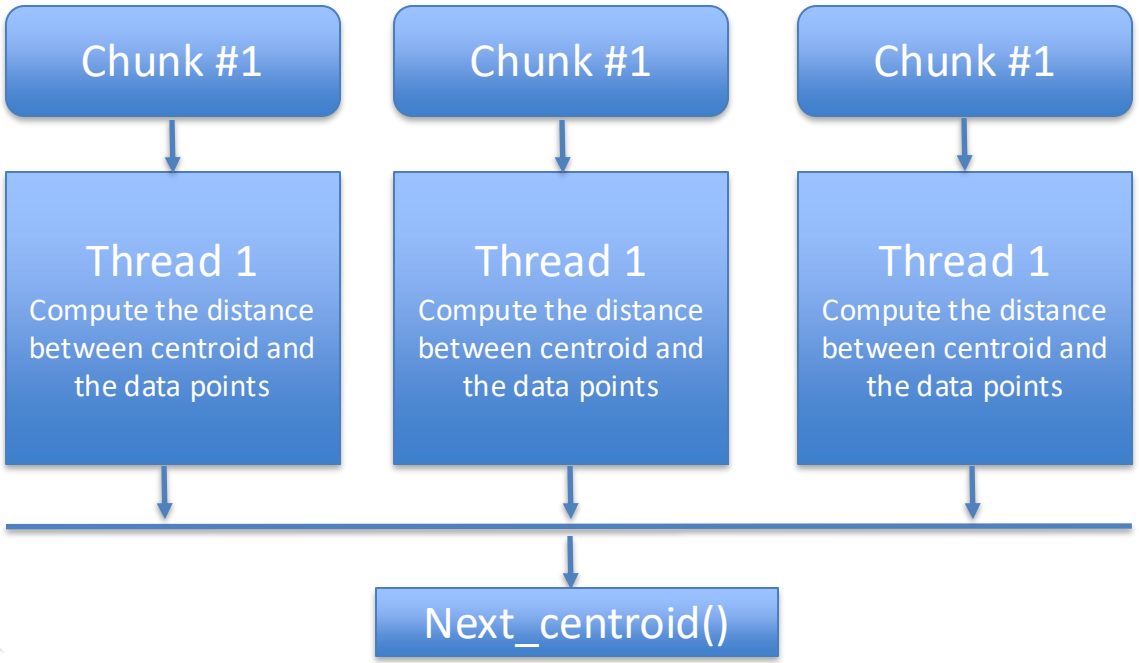
```
std::vector<Point> initialization_kmean_seq(const std::vector<Point> &data, const int k) {
    std::vector<Point> centroids;
    std::uniform_int_distribution<> distrib( a: 0,  b: data.size() - 1);
    centroids.push_back(data[distrib( &: gen)]);
    while (centroids.size() < k) {
        vector<double> distances_glob( n: data.size(),  value: numeric_limits<double>::max());
        for (size_t i = 0; i < data.size(); i++) {
            for (const Point &centroid : centroids) {
                distances_glob[i] = std::min(distances_glob[i], euclideanDistance( p1: data[i],  p2: centroid));
            }
        }
        centroids.push_back(next_centroid( dist: distances_glob, data));
    }
    return centroids;
}
Point next_centroid (const std::vector<double> &dist, const std::vector<Point> &data) {
    std::discrete_distribution<> distrib( wbegin: dist.begin(),  wend: dist.end());
    return data[distrib( &: gen)];
}
```

This inizialization method have cost of O(knk) vs random inizialization tha have a cost of O(k).

K-Means++ inizialization

Parallelization strategy

- The inner **for** loop, which iterates through the entire dataset, has been parallelized by dividing the data points into equal chunks for analysis.
- In this way, each thread computes the minimum distances for a subset of the total points relative to the current centroids.



K-Means++ inizialization, implementation

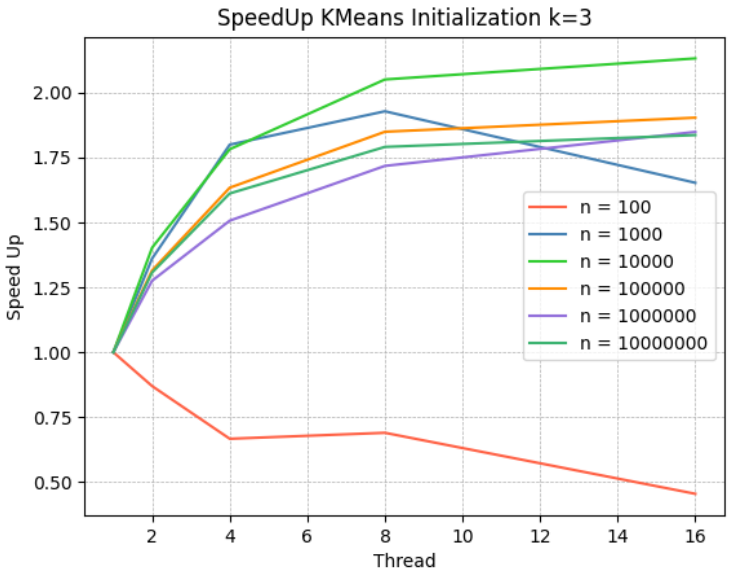
Parallel algorithm

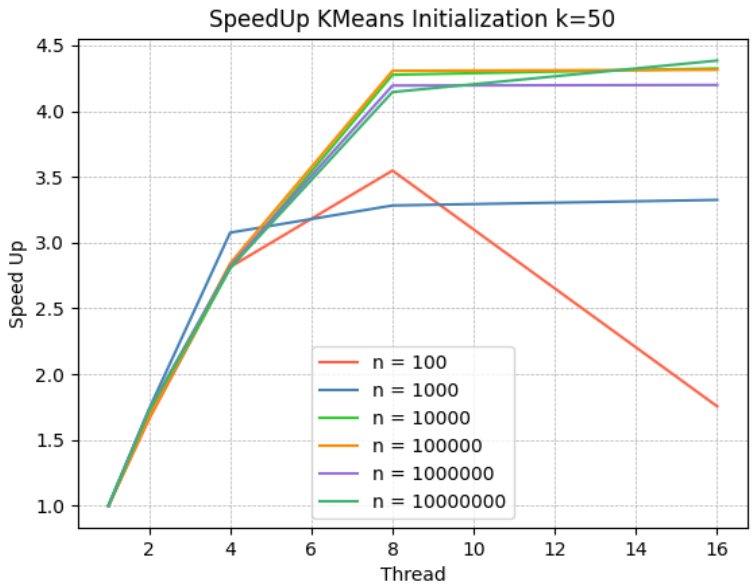
```
std::vector<Point> initialization_kmean_par(const std::vector<Point> &data, const int k, const int t) {
    std::vector<Point> centroids;
    centroids.reserve( n: k);
    std::uniform_int_distribution<> distrib( a: 0,  b: data.size() - 1);
    centroids.push_back(data[distrib( &: gen)]);
    int chunk = ceil( x: data.size() / t);
    while (centroids.size() < k) {
        vector<double> distances_glob( n: data.size(),  value: numeric_limits<double>::max());
        #pragma omp parallel for schedule(static, chunk) num_threads(t)
            for (int i = 0; i < data.size(); i++) {
                for (const Point &centroid : centroids) {
                    distances_glob[i] = std::min(distances_glob[i], euclideanDistance( p1: data[i],  p2: centroid));
                }
            }
        centroids.push_back(next_centroid( dist: distances_glob, data));
    }
    return centroids;
}
```

- This inizialization method have cost of O(k*n/t*k) , where t is the number of threads used

K-Means++ inizialization, results

Speed Up obtained: number of data points

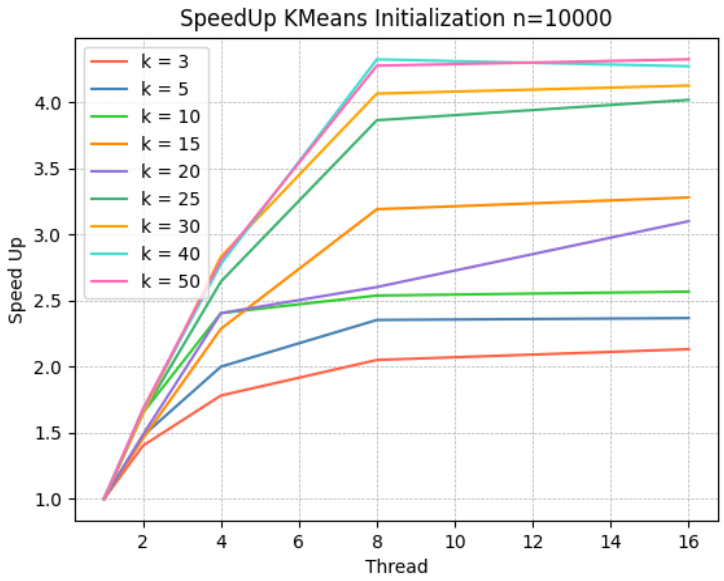


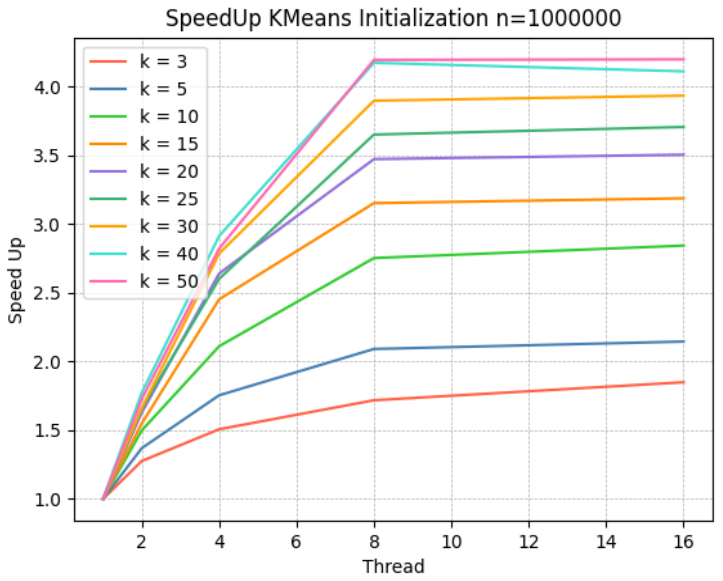


- For small values of n, this approach is generally not advisable, as the overhead of management outweighs the benefits.

K-Means++ inizialization, results

Speed Up obtained: number of clusters





- As expected, the speedup achieved increases roughly linearly with the increase in K.

Vectorization

Implementation and results

In addition to the parallelization approach discussed, a vectorization approach for distance computation was also tested.

```
double euclideanDistance(const Point& p1, const Point& p2) {
    double dist = 0;
    #pragma omp simd
    for (int i = 0; i < DIM; i++)
        dist += (p1.coordinate[i] - p2.coordinate[i]) * (p1.coordinate[i] - p2.coordinate[i]);
    return sqrt( x: dist);
}
```

- This approach did not yield any advantages in terms of speedup. It was tested with point dimensions of 3 and 5; going beyond these dimensions made visualizing the output too complex.
- This result also applies to the following section, where the parallel implementation of the K-Means algorithm will be discussed.

K-Means implementation

Sequential algorithm

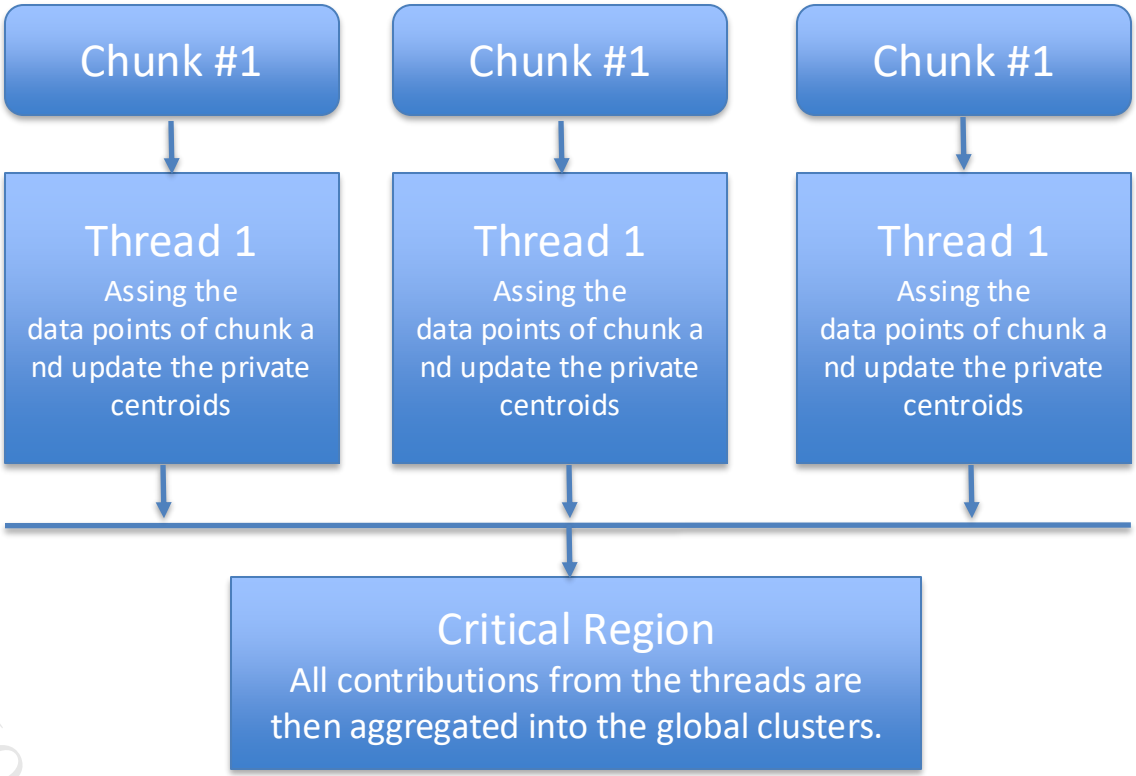
- The assignment has a cost of O(dnk).
- The update phase has a cost of O(k).

```
vector<Point> kMeans(vector<Point> &data, vector<Point> &centroids, int k, int maxIterations) {
    vector<Point> newCentroids = std::vector<Point>( n: k,  value: Point());
    for (int iter = 0; iter < maxIterations; ++iter) {
        vector<int> counts( n: k,  value: 0);
        newCentroids = std::vector<Point>( n: k,  value: Point());
        for (Point &pt: data) {
            double minDistance = distance( p1: pt,  p2: centroids[0]);
            pt.actualCentroid = 0;
            for (int i = 0; i < k; i++) {
                double x = distance( p1: pt,  p2: centroids[i]);
                if (x < minDistance) {
                    minDistance = x;
                    pt.actualCentroid = i;
                }
            }
            newCentroids[pt.actualCentroid] += pt;
            counts[pt.actualCentroid]++;
        }
        for (int i = 0; i < k; i++) {
            newCentroids[i] /= counts[i];
        }
        if (areEqual( vec1: centroids,  vec2: newCentroids))
            return centroids;
        centroids = newCentroids;
    }
    return centroids;
}
```

K-Means

Parallelization strategy

- The inner **for** loop, which iterates through all the points in the dataset, has been parallelized.
- In this way, each thread works on a chunk of data, assigning points to local clusters and then updating these local clusters.



K-Means, implementation (1/2)

Parallel algorithm

```
vector<Point> kMeans(vector<Point> &data, vector<Point> &centroids, int k, int maxIterations, int threads) {
    int block = ceil( x: data.size() / threads);
    vector<Point> newCentroids( n: k, value: Point());
    vector<int> counts( n: k, value: 0);
    for (int iter = 0; iter < maxIterations; ++iter) {
        newCentroids = vector<Point>( n: k, value: Point());
        counts = vector<int>( n: k, value: 0);
        #pragma omp parallel num_threads(threads)
        {
            vector<int> tmp_counts( n: k, value: 0);
            vector<Point> tmp_newCentroids( n: k, value: Point());
            #pragma omp for nowait schedule(static, block)
            for (Point &pt: data) {
                double minDistance = distance( p1: pt, p2: centroids[0]);
                pt.actualCentroid = 0;
                for (int j = 1; j < k; j++) {
                    double dist = distance( p1: pt, p2: centroids[j]);
                    if (dist < minDistance) {
                        minDistance = dist;
                        pt.actualCentroid = j;
                    }
                }
                tmp_newCentroids[pt.actualCentroid] += pt;
                tmp_counts[pt.actualCentroid]++;
            }
```

K-Means, implementation (2/2)

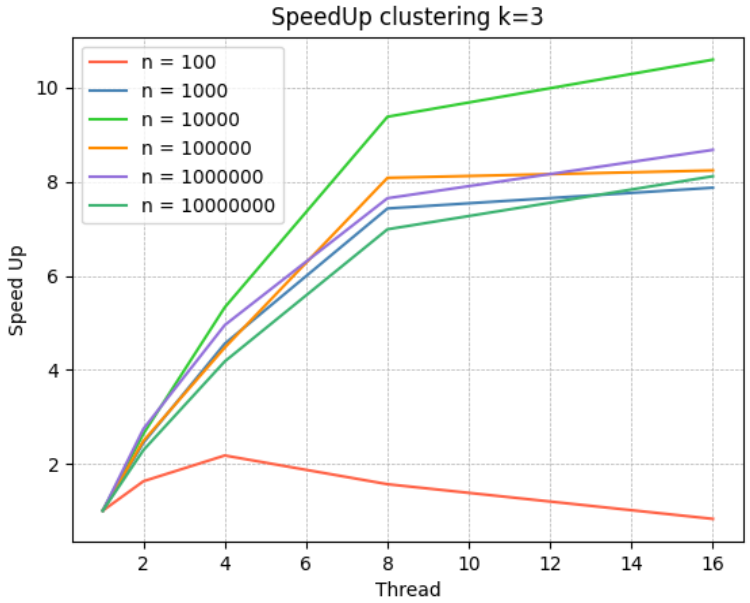
Parallel algorithm

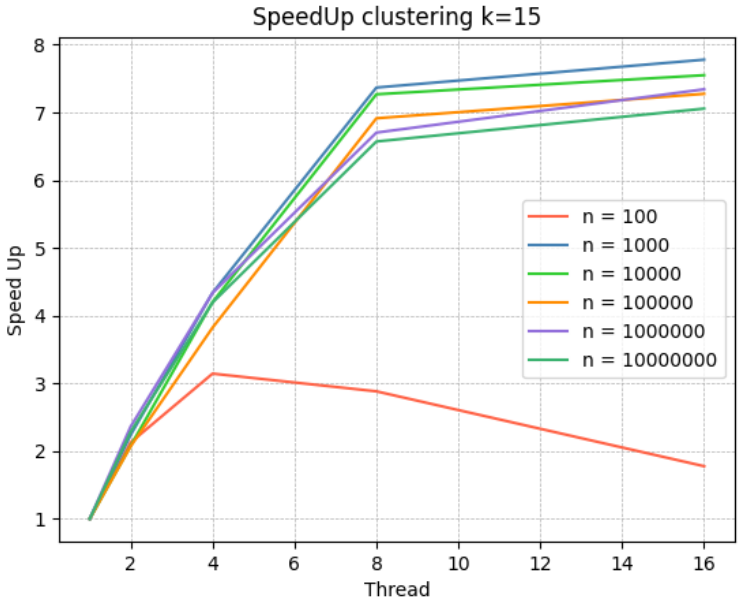
```
            #pragma omp critical
            {
                for (int i = 0; i < k; i++) {
                    newCentroids[i] += tmp_newCentroids[i];
                    counts[i] += tmp_counts[i];
                }
            }
        }
        for (int i = 0; i < k; i++) {
            newCentroids[i] /= counts[i];
        }
        if (areEqual( vec1: centroids,  vec2: newCentroids)) {
            return centroids;
        }
        centroids = newCentroids;
    }
    return centroids;
}
```

- In this mode, the algorithm has a cost of O(d * n/t * k)
- The update always has a cost of O(k)

K-Means, results

Speed Up obtained: number of data points

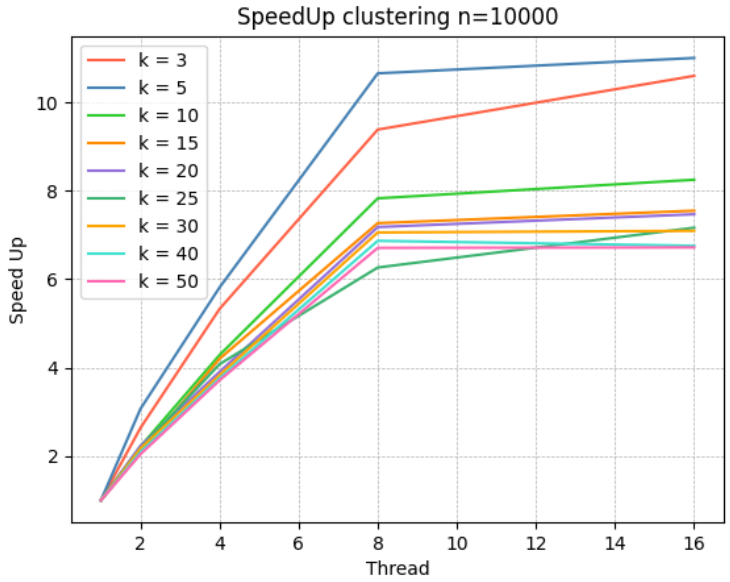


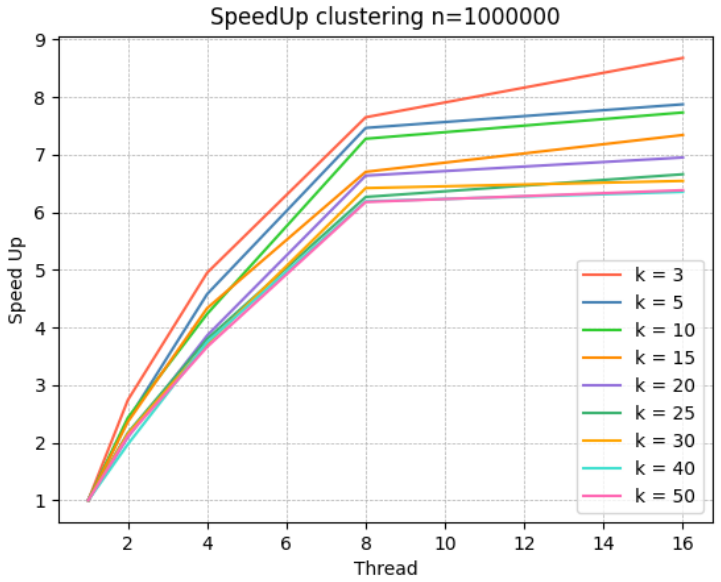


- Even in this case, for small values of n, this approach is not worthwhile.
- Except for cases with small k, the results align with expectations: for larger n, we generally achieve greater speedup.

K-Means, results

Speed Up obtained: number of clusters





- For higher values of k, the speedup tends to stabilize between 7 and 8.
- In general, the highest recorded speedup value is 11, achieved with 16 threads, n=10000 and k=5.