UNIVERSITÀ
DEGLI STUDI
FIRENZE

K-Means Clustering with Open-MP

Parallel Computing
(Mid-Term Assignment)

Davide Del Bimbo
davide.delbimbo@edu.unifi.it

09/11/2023

Introduction

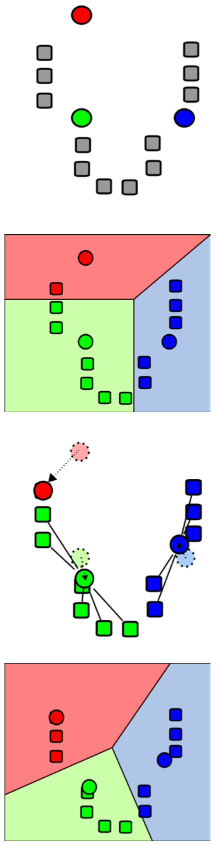
- K-Means is a clustering algorithm.
- Unsupervised learning technique.
- Identify patterns in data and divide it into **groups** or **clusters**.
- Assign each data point to one cluster:
  - Data within the same cluster are similar.
  - Data within different clusters are dissimilar.

- Dataset $\mathcal{X} = \{x_1, \dots, x_N\}$.
- Clusters $\mathcal{C} = \{\mathcal{C}_1, \dots, \mathcal{C}_K\}$:
  - Disjointed: $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset \; \forall i \neq j$.
  - **Centroid**: $\mu_k = \frac{1}{|\mathcal{C}_k|} \sum_{x \epsilon \mathcal{C}_k} x$.
- Goal: minimize the within-cluster sum-of-square criterion, called **inertia**.

$$\sum_{i=1}^{N} \min_{\mu_k \epsilon \mathcal{C}} \|x_i - \mu_k\|$$

Algorithm

1. **Initialization**: given dataset $\mathcal{X} = \{x_1, \dots, x_N\}$ of N multidimensional samples, choose K random initial cluster centroids from the dataset.
2. **Assignments**: assign each data point to the centroid of the nearest cluster based on a distance metric (Euclidean distance).
3. **Updates**: update cluster centroids by calculating the mean of all data points assigned to each cluster.
4. **Repeat**: repeat steps 2 and 3 until convergence is achieved.



- Convergence criteria:
  - No shifts in centroids (`EPSILON = 1e-6`).
  - No change in the allocation of points.
  - Reaching the minimum sum of distances.
- Maximum limit of iterations (`MAX_ITERATIONS = 500`).

- Convergence criteria:
  - No shifts in centroids (`EPSILON = 1e-6`).
  - No change in the allocation of points.
  - Reaching the minimum sum of distances.
- Maximum limit of iterations (`MAX_ITERATIONS = 500`).

- Convergence criteria:
  - No shifts in centroids (`EPSILON = 1e-6`).
  - No change in the allocation of points.
  - Reaching the minimum sum of distances.
- Maximum limit of iterations (`MAX_ITERATIONS = 500`).

Sequential implementation

- `Point`: struct representing a multidimensional data point.
    - `coordinates`: coordinates of data point (`vector<double>`).
    - `pointId`: unique identifier of data point (`int`).
    - `clusterId`: identifier of assigned cluster (`int`). Initialize to default value `-1`.
- `Centroid`: struct representing a multidimensional centroid of a cluster.
    - `coordinates`: coorindates of centroid (`vector<double>`).
    - `clusterId`: unique identifier of assigned cluster (`int`).

```cpp
// Point in multidimensional space.
struct Point {
  vector<double> coordinates; // Vector of coordinates.
  int pointId; // Identifier of the point.
  int clusterId; // Identifier of the cluster to which the point belongs.

  Point(const vector<double>& coordinates, const int pointId,
      const int clusterId);
};
```

```cpp
// Centroid in multidimensional space.
struct Centroid {
  vector<double> coordinates; // Vector of coordinates.
  int clusterId; // Identifier of the cluster.

  Centroid(const vector<double>& coordinates, const int clusterId);
};
```

- `KMeans`: class representing K-Means algorithm.
    - Instantiates N data points randomly or from an input dataset.
    - Instantiates K clusters from distinct random data points.
    - Runs `KMeansIteration` method until the convergence criterion is met.
    - Provides a results plotting and a GIF animation of the execution.
    - Saves the execution time of results to a file.
- `KMeansIteration`: method representing a single iteration of K-Means algorithm.

```cpp
// Variables for the mean of the points in each cluster.
vector<std::vector<double>> clustersSum(K, vector<double>(dimensions, 0));
vector<int> clustersSize(K, 0);

// Convergence flag. Assume convergence at the beginning.
bool converged = true;

// Assign each point to the closest centroid.
for(int i = 0; i < N; i++) {
    double minDist = DBL_MAX;
    int minClusterId = -1;

    for(int j = 0; j < K; j++) {
        double dist = distance(points[i], centroids[j]);
        if(dist < minDist) {
            minDist = dist;
            minClusterId = j;
        }
    }

    // Assign the point to the closest cluster.
    points[i].clusterId = minClusterId;

    for(int dim = 0; dim < dimensions; dim++) {
        clustersSum[minClusterId][dim] += points[i].coordinates[dim];
    }
    clustersSize[minClusterId]++;
}
```

```
// Variables for the mean of the points in each cluster.
vector<std::vector<double>> clustersSum(K, vector<double>(dimensions, 0));
vector<int> clustersSize(K, 0);

// Convergence flag. Assume convergence at the beginning.
bool converged = true;

// Assign each point to the closest centroid.
for(int i = 0; i < N; i++) {
    double minDist = DBL_MAX;
    int minClusterId = -1;

    for(int j = 0; j < K; j++) {
        double dist = distance(points[i], centroids[j]);
        if(dist < minDist) {
            minDist = dist;
            minClusterId = j;
        }
    }

    // Assign the point to the closest cluster.
    points[i].clusterId = minClusterId;

    for(int dim = 0; dim < dimensions; dim++) {
        clustersSum[minClusterId][dim] += points[i].coordinates[dim];
    }
    clustersSize[minClusterId]++;
}
```

```
// Variables for the mean of the points in each cluster.
vector<std::vector<double>> clustersSum(K, vector<double>(dimensions, 0));
vector<int> clustersSize(K, 0);

// Convergence flag. Assume convergence at the beginning.
bool converged = true;

// Assign each point to the closest centroid.
for(int i = 0; i < N; i++) {
    double minDist = DBL_MAX;
    int minClusterId = -1;

    for(int j = 0; j < K; j++) {
        double dist = distance(points[i], centroids[j]);
```

```
const double distance(const Point &p, const Centroid &c) {
    double sum = 0;
    for (int dim = 0; dim < dimensions; dim++) {
        sum += (c.coordinates[dim] - p.coordinates[dim]) * (c.coordinates[dim] - p.coordinates[dim]);
    }

    return sqrt(sum);
}
```

```
        for(int dim = 0; dim < dimensions; dim++) {
            clustersSum[minClusterId][dim] += points[i].coordinates[dim];
        }
        clustersSize[minClusterId]++;
}
```

```
// Update the centroids.
for(int j = 0; j < K; j++){
    // Temporary variable for the previous centroid coordinate.
    double tmpCoordinate = 0;

    // Update the centroid of the cluster.
    for(int dim = 0; dim < dimensions; dim++) {
        // Save the previous centroid coordinate.
        tmpCoordinate = centroids[j].coordinates[dim];

        centroids[j].coordinates[dim] = clustersSum[j][dim] / clustersSize[j];

        // Check for convergence (i.e. if the centroid change position in a dimension).
        if (fabs(tmpCoordinate - centroids[j].coordinates[dim]) > EPSILON) {
            converged = false;
        }
    }
}
```

Parallel implementation

- Uses the SoA (Structure of Arrays) architecture.
- Uses linearized vectors to represent multidimensional matrices.
- Uses OpenMP pragmas to enable multithreading and parallelized code:
    - Parallelization of the first loop, as it requires high cost for large datasets ($O(N \times K \times D)$).
    - No parallelization of the second loop, since it has a small cost for tipical values of K ($O(K \times D)$). It may result in overhead costs.
    - Vectorization of distance calculation to enable SIMD processing.

```
// Points in multidimensional space using SoA architecture.
struct Points {
  const int size; // Number of points.
  double* coordinates; // Array of coordinates of all dimensions.
  int* pointsIds; // Array of points identifiers.
  int* clustersIds; // Array of clusters identifiers to which the points belong.

  Points(const int size, double* coordinates, int* pointsIds,
      int* clustersIds);

  ~Points();
};
```

```
// Centroids in multidimensional space using SoA architecture.
struct Centroids {
  const int size; // number of clusters.
  double* coordinates; // Array for coordinates of all dimensions.
  int* clustersIds; // Array for clusters identifiers.

  Centroids(const int size, double* coordinates, int* clustersIds);

  ~Centroids();
};
```

```c
// Variables for the mean of the points in each cluster.
double clustersSum[K * dimensions] = {0}; // Sum of coordinates of points in each cluster.
int clustersSize[K] = {0}; // Number of points in each cluster.

// Convergence flag. Assume convergence at the beginning.
bool converged = true;

// Assign each point to the closest centroid.
#pragma omp parallel for schedule(static) default(none) \
     shared(points, centroids, clustersSum, clustersSize, converged)
for(int i = 0; i < N; i++) {
    double minDist = DBL_MAX;
    int minClusterId = -1;

    for(int j = 0; j < K; j++) {
        double dist = distance(i, j);
        if(dist < minDist) {
            minDist = dist;
            minClusterId = j;
        }
    }

    // Update the identifier of the cluster.
    points.clustersIds[i] = minClusterId;

    for(int dim = 0; dim < dimensions; dim++) {
        #pragma omp atomic
        clustersSum[minClusterId + K * dim] += points.coordinates[i + N * dim];
    }

    #pragma omp atomic
    clustersSize[minClusterId]++;
}
```

```
// Variables for the mean of the points in each cluster.
double clustersSum[K * dimensions] = {0}; // Sum of coordinates of points in each cluster.
int clustersSize[K] = {0}; // Number of points in each cluster.

// Convergence flag. Assume convergence at the beginning.
bool converged = true;

// Assign each point to the closest centroid.
#pragma omp parallel for schedule(static) default(none) \
     shared(points, centroids, clustersSum, clustersSize, converged)
for(int i = 0; i < N; i++) {
    double minDist = DBL_MAX;
    int minClusterId = -1;

    for(int j = 0; j < K; j++) {
        double dist = distance(i, j);
        if(dist < minDist) {
            minDist = dist;
            minClusterId = j;
        }
    }

    // Update the identifier of the cluster.
    points.clustersIds[i] = minClusterId;

    for(int dim = 0; dim < dimensions; dim++) {
        #pragma omp atomic
        clustersSum[minClusterId + K * dim] += points.coordinates[i + N * dim];
    }

    #pragma omp atomic
    clustersSize[minClusterId]++;
}
```

```
// Variables for the mean of the points in each cluster.
double clustersSum[K * dimensions] = {0}; // Sum of coordinates of points in each cluster.
int clustersSize[K] = {0}; // Number of points in each cluster.

// Convergence flag. Assume convergence at the beginning.
bool converged = true;

// Assign each point to the closest centroid.
#pragma omp parallel for schedule(static) default(none) \
     shared(points, centroids, clustersSum, clustersSize, converged)
for(int i = 0; i < N; i++) {
    double minDist = DBL_MAX;
    int minClusterId = -1;

    for(int j = 0; j < K; j++) {
        double dist = distance(i, j);
```

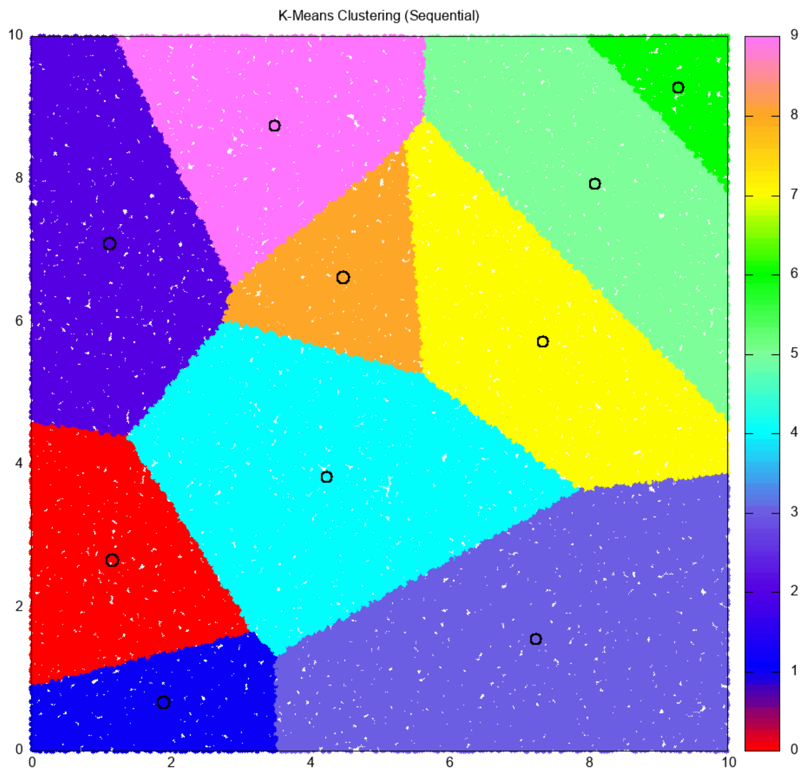
```
const double distance(const int pointId, const int centroidId) {
    double sum = 0;
    #pragma omp simd reduction(+:sum)
    for (int dim = 0; dim < dimensions; dim++) {
        sum += (centroids.coordinates[centroidId + K*dim] - points.coordinates[pointId + N*dim]) *
               (centroids.coordinates[centroidId + K*dim] - points.coordinates[pointId + N*dim]);
    }

    return sqrt(sum);
}
```

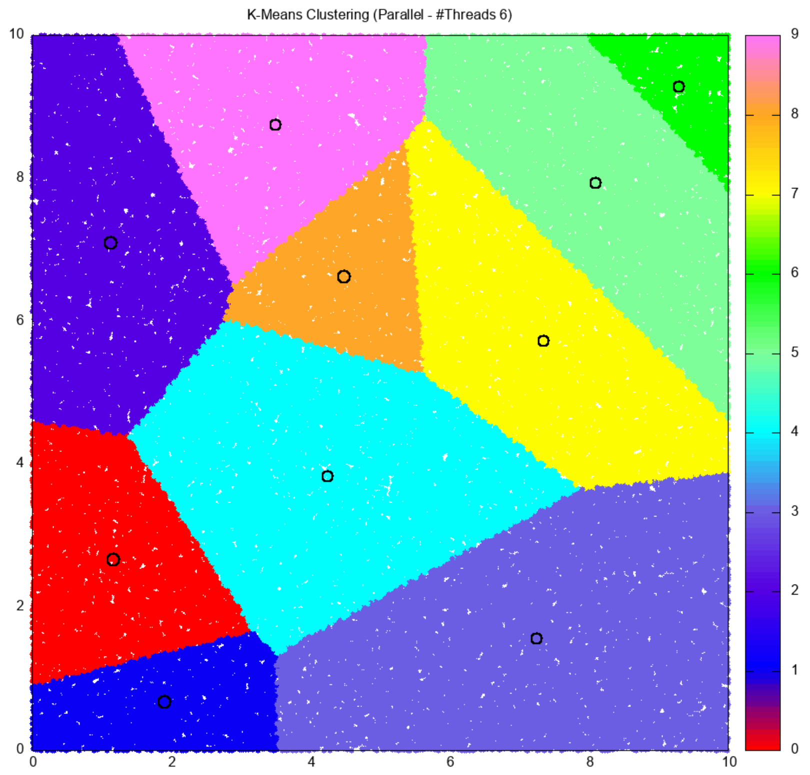
```
        #pragma omp atomic
        clustersSum[minClusterId + K * dim] += points.coordinates[i + N * dim];
    }

    #pragma omp atomic
    clustersSize[minClusterId]++;
```

Correctness



Animation of sequential execution of the K-Means algorithm on a dataset of 100.000 two-dimensional data points, organized in 10 clusters. The execution time was $5.52s$.



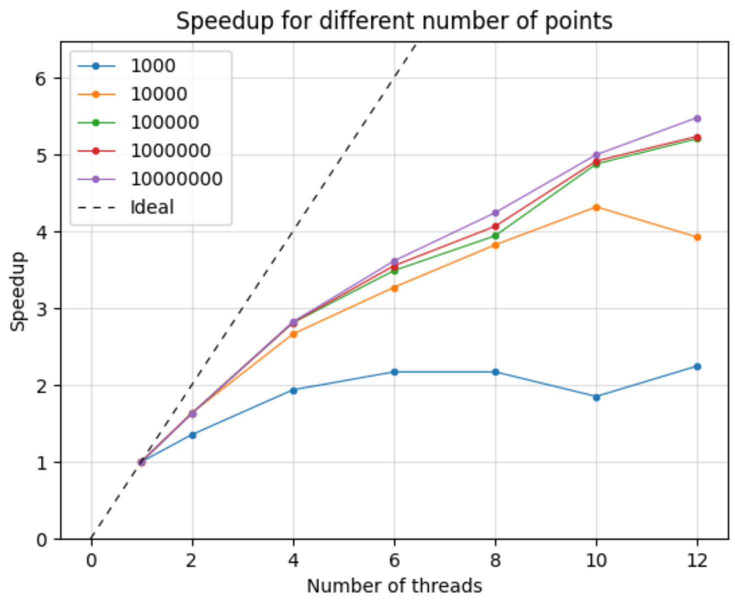
Animation of parallel execution with 6 threads of the K-Means algorithm on a dataset of 100.000 two-dimensional data points, organized in 10 clusters. The total execution time was $1.34s$.

Performance

- Analysis of speedup as the number of data points, number of clusters and data point dimensions change:
  1. Dataset of 1.000, 10.000, 100.000, 1.000.000, 10.000.000 two-dimensional data points, organized in 10 clusters.
  2. Dataset of 10.000 two-dimensional data points, organized in 2, 5, 10, 50, 100 clusters.
  3. Dataset of 10.000 data points with 2, 3, 5, 10, 20 dimensions, organized in 10 clusters.
- Comparison of AoS and SoA architectures.

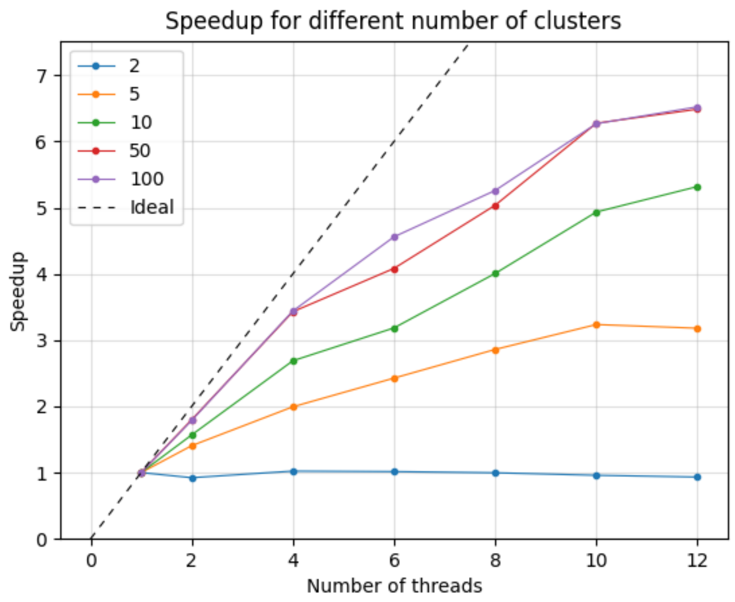
- Speed as the number of points changes:

	2 Threads	4 Threads	6 Threads	8 Threads	10 Threads	12 Threads
1.000	1,35	1,94	2,17	2,17	1,85	2,25
10.000	1,64	2,65	3,27	3,83	4,32	3,92
100.000	1,63	2,81	3,49	3,94	4,88	5,21
1.000.000	1,63	2,81	3,55	4,07	4,92	5,24
10.000.000	1,63	2,82	3,62	4,24	5,00	**5,48**



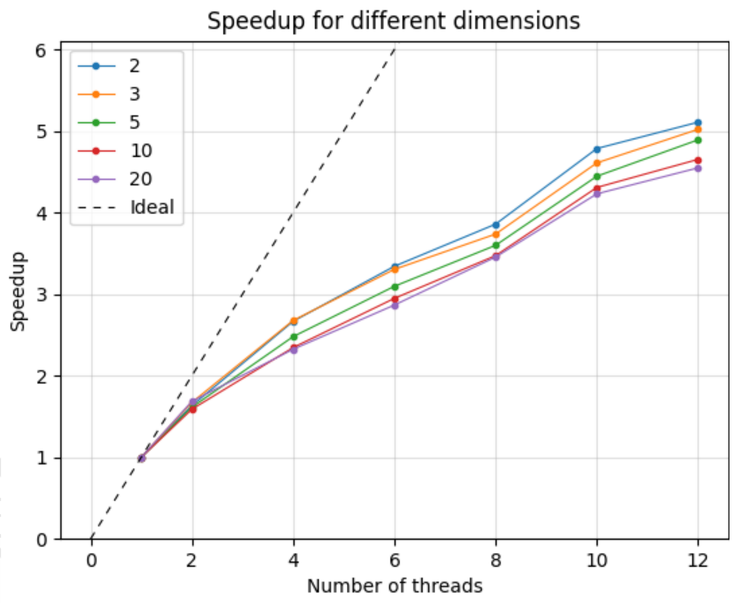
- Speed as the number of clusters changes:

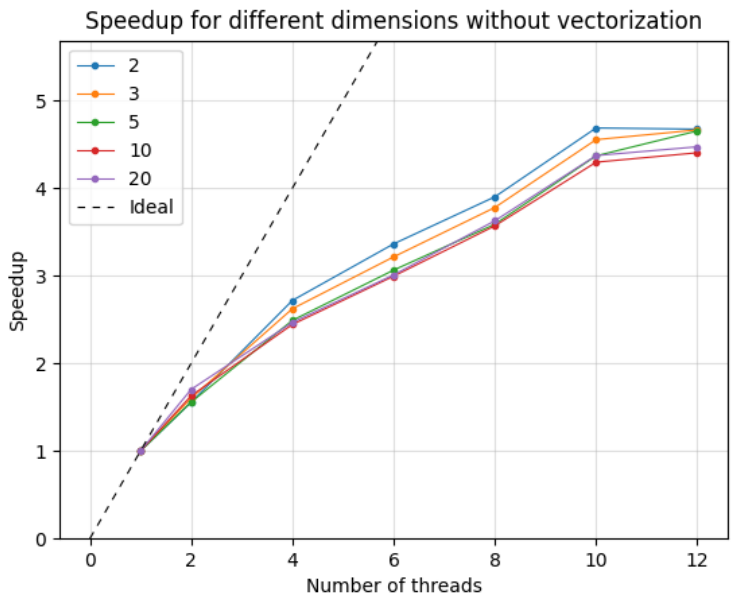
	2 Threads	4 Threads	6 Threads	8 Threads	10 Threads	12 Threads
2	0,92	1,02	1,02	1,00	0,96	0,93
5	1,41	1,99	2,43	2,86	3,24	3,18
10	1,57	2,69	3,18	4,01	4,93	5,32
50	4,80	3,43	4,08	5,04	6,28	6,49
100	1,79	3,44	4,56	5,26	6,27	**6,52**



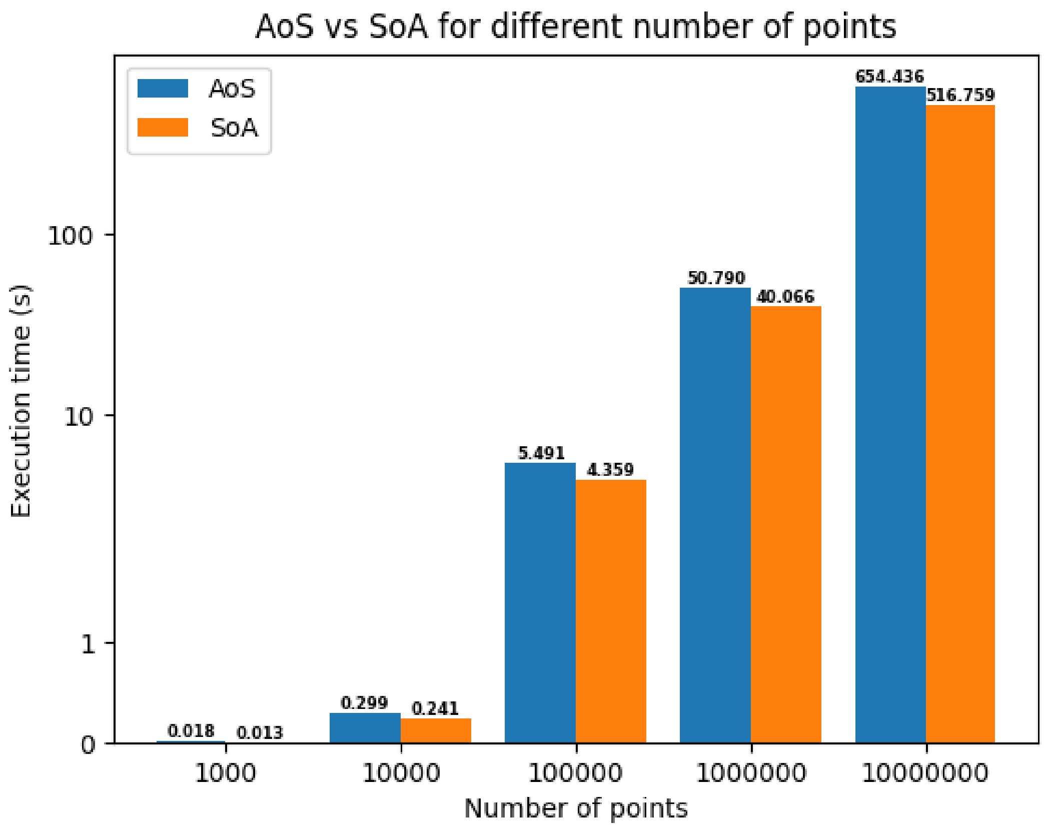
- Speed as the dimensions changes:

	2 Threads	4 Threads	6 Threads	8 Threads	10 Threads	12 Threads
2	1,63	2,67	3,34	3,86	4,78	**5,11**
3	4,67	2,68	3,30	3,74	4,61	5,02
5	1,61	2,48	3,10	3,60	4,45	4,89
10	1,59	2,35	2,95	3,47	4,31	4,65
20	1,69	2,32	2,87	3,45	4,23	4,55





- AoS vs SoA:


AoS vs SoA for different number of points
AoS
SoA
Execution time (s)
Number of points
0
1
10
100
1000
10000
100000
1000000
10000000
0.018
0.013
0.299
0.241
5.491
4.359
50.790
40.066
654.436
516.759

Conclusions

- Parallel code results in higher performance than sequential code.
- OpenMP parallelization is particularly advantageous, enabling a significant improvement in speedup.
- SoA architecture reduces the execution time.
- Vectorization provides a slight improvement in high-dimensional computation.