Parallel Programming - Middle term assignment
2D K-Means (OpenMP Version)

Angelo Caponnetto

June 20, 2024

Abstract

In this work, I will study the 2D k-Means algorithm. I will compare the sequential version of the program with the parallel version. I will describe the code I've used and the obtained speed up.

1 Introduction

The assignment consists in the study of the parallel version of K-Means. Starting from k random true centroids [in the 2D square with diagonal $(0, 0), (1, 1)$], the algorithm generates N random points with deviation σ.

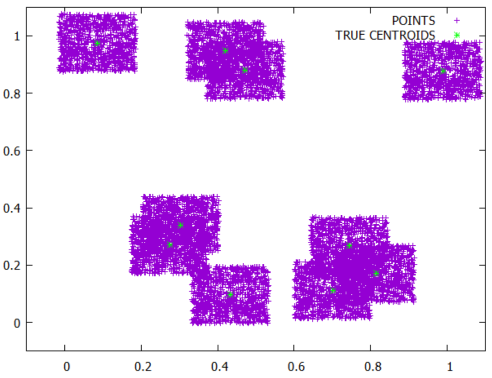

Figure 1: Points visualization

The goal is to find true centroids, starting from k random points of the dataset.

2 Data structure

Generic points and centroids are organized in structures of arrays.

```
typedef struct {
    double x[N];
    double y[N];
} PointsVec;


typedef struct {
    double x[k];
    double y[k];
    int n_points[k];
} CentroidsVec;
```

x and y are the arrays of the Cartesian coordinates; *n_points* is the array in which the algorithm saves the number of points that belong to a centroid.

3 The algorithms

In this report, two K-means algorithms will be studied: the main difference is the way in which the work is divided among all the threads. Both the algorithms are written in OpenMP: in the first one (FV) the points structure is divided in L blocks (L = active threads) and each block is manually assigned to a specific thread. In order to do that, one thread (the master) generates tasks for all the other threads.
In the second one (SV) the compiler chooses how to divide the work ('pragma omp parallel for' directive).

Algorithm 1 K-means FV(points, centroids, n_thr)

```
# pragma omp parallel num_threads(n_thr)
# pragma omp master
    **centroidsVec** master_centr[n_thr]
    **while** stop condition in False **do**
        **for** each active thread **do**
        # pragma omp task
            **int** id
            **centroidsVec** $priv\_centr \leftarrow 0$
            Divide points in n_thr-blocks
            **for** each point in block **do**
                **for** each centroid **do**
                    Find min distance
                **end for**
                Update priv_centr[closer centr]
            **end for**
            Update master_centr[id] with
                                      priv_centr data
        **end for**
        # pragma omp taskwait

        Evaluate new centroids
        centroids ← new centroids
    **end while**
```

Algorithm 2 K-means SV(points, centroids, n_thr)

```
**double** x[k], y[k]
**int** n_points[k]
# pragma omp parallel num_threads(n_thr)
**while** stop condition is False **do**
    $x[k] \leftarrow 0, y[k] \leftarrow 0, n\_points[k] \leftarrow 0$
    # pragma omp for reduction on x, y, n_points
    **for** all points **do**
        **for** all centroids **do**
            Find the min distance
        **end for**
        Update x, y, n_points of the closer centr
    **end for**
    # pragma omp barrier

    # pragma omp single
    Evaluate new centroids
    centroids ← new centroids
**end while**
```

4 Utilized CPU

Following results were collected using the CPU AMD Ryzen 7 3750H. Here we report same characteristics:

- number of cores: 4
- number of flows: 8
- Frequency: 2.3 GHz
- L1 Cache: 384 KB
- L2 Cache: 2 MB
- L3 Cache: 4 MB

5 Results

In Figure 2 and in Figure 3 time results are reported (times for 100 iterations of while loop). They were taken by changing the number of particles N and the number of active thread n_thr. Speed up curves are evaluated using times of the sequential version (only one active thread) as reference.

6 Conclusions

Plots show the performance of the algorithm, when the number of particles and the number of active thread change. As we can see, both versions of the algorithm have similar performance. Speed-up increases when n_thr $\in [1, 8]$ and it reaches the maximum for n_thr $= 8$ (speed-up ≈ 4). When n_thr > 8, overhead effects decrease performance.
Note that, in general, the first version of the algorithm achieves slightly higher performance than the second version.

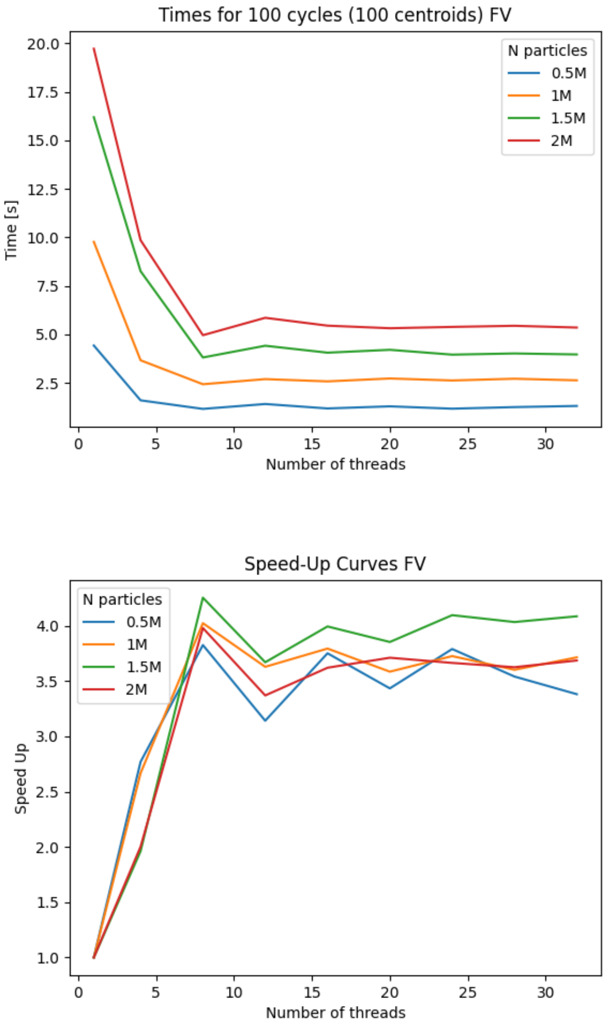

Figure 2: Times and speed-up of the FV algorithms.

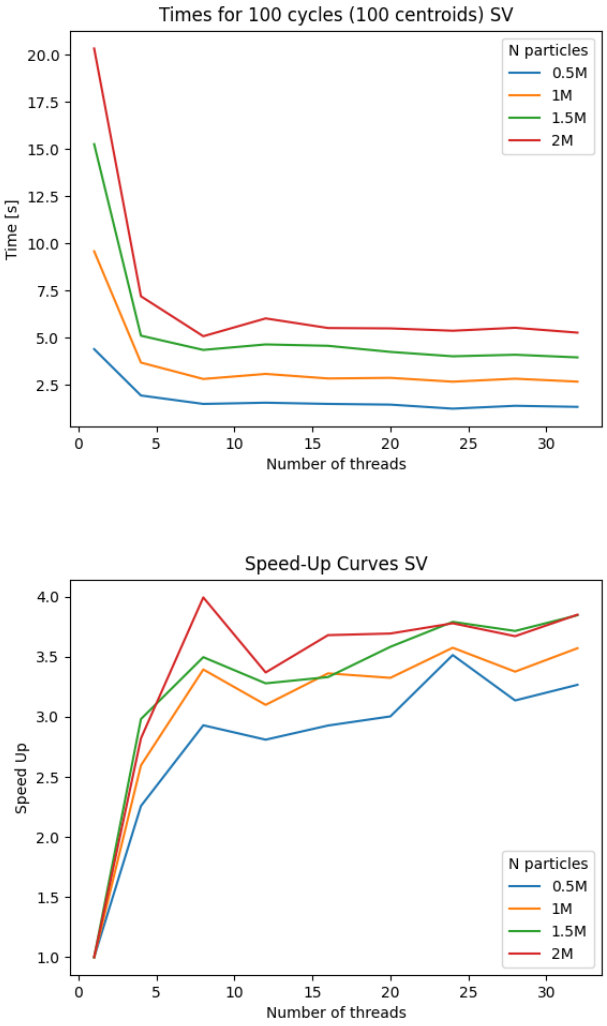

Figure 3: Times and speed-up of the SV algorithms.