K-Means — Random Maze Solver

Parallel Programming - Final Presentation

Angelo Caponnetto

June 21 2024

UNIVERSITÀ
DEGLI STUDI
FIRENZE

- K-Means 2D
  - Intoduction
  - Data structure
  - The Algorithms (OpenMP)
  - Results

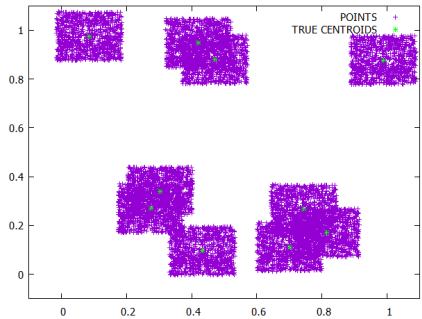
- Random Maze Solver
  - Introduction
  - Data Structure
  - The Algorithm (CUDA)
  - Results

K-Means - Introduction

1 K-Means 2D

K-Means is an unsupervised learning method that allows you to find subgroups in a dataset. In this work K-Means 2D is studied.

Starting from k random true centroids, the algorithm generates N random points with deviation σ.



The goal is to find true centroids.

Data Structure

1 K-Means 2D

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.

The Algorithms

1 K-Means 2D

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

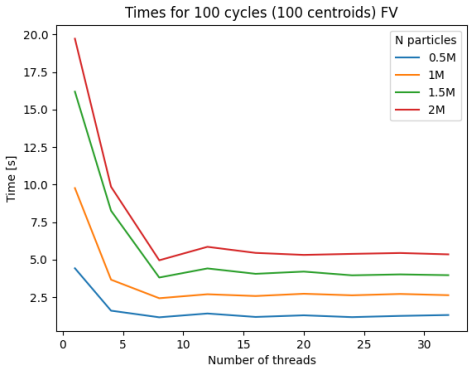
```
1:  # pragma omp parallel num_threads(n_thr)
2:  # pragma omp master
3:      centroidsVec master_centr[n_thr]
4:      while stop condition in False do
5:          for each active thread do
6:          # pragma omp task
7:              int id
8:              centroidsVec priv_centr ← 0
9:              Divide points in n_thr-blocks
10:             for each point in block do
11:                 for each centroid do
12:                     Find min distance
13:                 end for
14:                 Update priv_centr[closer centr]
15:             end for
16:             Update master_centr[id] with
17:                                 priv_centr data
18:         end for
19:         # pragma omp taskwait
20:
21:         Evaluate new centroids
22:         centroids ← new centroids
23:     end while
```

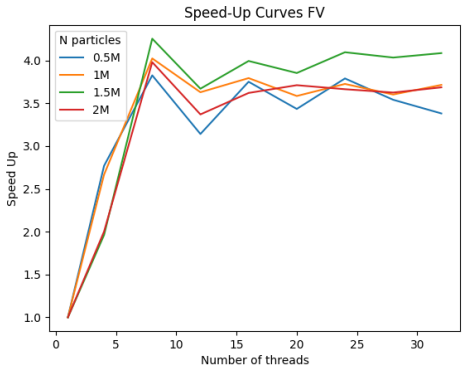
Algorithm 2 K-means SV(points, centroids, n_thr)

```
1:  double x[k], y[k]
2:  int n_points[k]
3:  # pragma omp parallel num_threads(n_thr)
4:  while stop condition is False do
5:      x[k] ← 0, y[k] ← 0, n_points[k] ← 0
6:      # pragma omp for reduction on x, y, n_points
7:      for all points do
8:          for all centroids do
9:              Find the min distance
10:         end for
11:         Update x, y, n_points of the closer centr
12:     end for
13:     # pragma omp barrier
14:
15:     # pragma omp single
16:     Evaluate new centroids
17:     centroids ← new centroids
18: end while
```

Results (Algorithm 1)

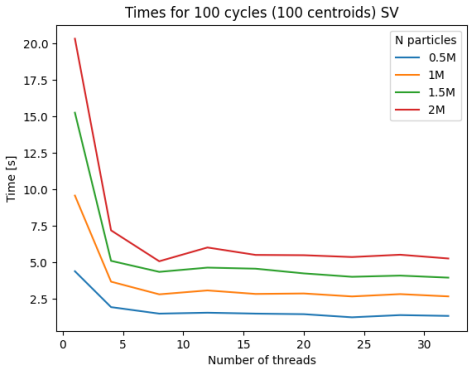
1 K-Means 2D

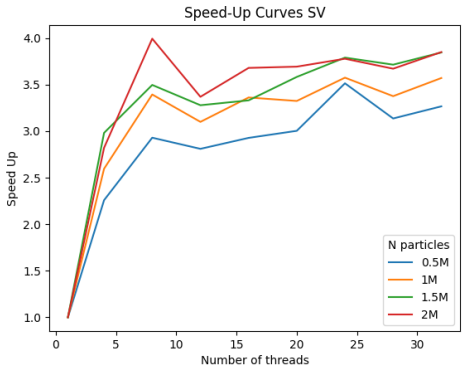




Results (Algorithm 2)

1 K-Means 2D

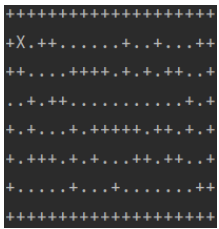




RMS - Introduction

2 Random Maze Solver

The program generates a maze in the form:



- Dot represents corridor
- Plus symbol represents wall
- X represents the starting position

In order to find the exit, the algorithm generates N particles (initialized on the starting position) and for each of them, at each step, it randomly chooses an adjacent tile among the allowed ones (the ones with dot inside).

Data Structure

2 Random Maze Solver

The data structure the algorithm uses is an array organized in this way:

POS	VAL	
.	.	
.	.	
i	x_m	Tile coordinates
i + 1	y_n	
i + 2	n_adjac	Number of adjacent corridors of (x_m, y_n)
i + 3	x_1	Coordinates adjacent corridor 1
i + 4	y_1	
i + 5	x_2	Coordinates adjacent corridor 2
i + 6	y_2	
i + 7	x_3	Coordinates adjacent corridor 3
i + 8	y_3	
i + 9	x_4	Coordinates adjacent corridor 4
i + 10	y_4	
.	.	
.	.	

- If a tile does not have four adjacent corridors, the excess positions in the array will be filled with zeroes
- Tiles in data structure are sorted by rows ((0,0), (1,0), (2,0) ...)

UNIVERSITÀ DEGLI STUDI FIRENZE

The Algorithm

2 Random Maze Solver

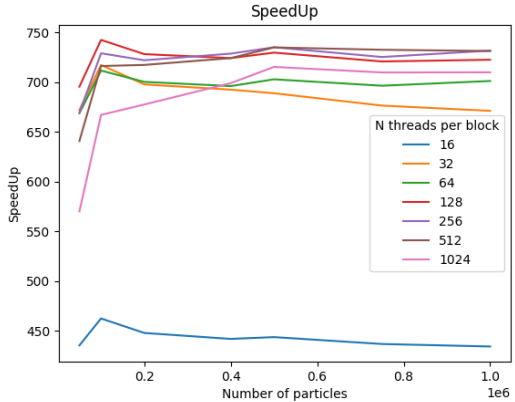
Algorithm 2 _global_RandSolver(x_{array}, y_{array}, (x_{ext}, y_{ext}), flag, N, d_lin _maze)

1: **int** $idx \leftarrow$ thread id
2: **if** $idx < N$ **then**
3: **short** $x \leftarrow x_{array}[idx]$, $y \leftarrow y_{array}[idx]$
4: **short** n_rand
5: **short** $firstNeigPos \leftarrow$ first initialization
6: **int** $n\_steps \leftarrow 0$
7:
8: **while** $flag \neq 1$ **and** $n\_steps < max\_steps$ **do**
9: $n\_rand \leftarrow n \in [0, n_{adjac} - 1]$
10: $x \leftarrow d\_lin\_maze[firstNeigPos + 2n\_rand]$
11: $y \leftarrow d\_lin\_maze[firstNeigPos +$
12: $2n\_rand + 1]$
13: Update $firstNeigPos$
14: $n\_steps \; += 1$
15:
16: **if** $(x, y) = (x_{exit}, y_{exit})$ **then** $flag = 1$
17: **end if**
18: **end while**
19: $x_{array}[idx] \leftarrow x$, $y_{array}[idx] \leftarrow y$
20: **end if**

Results

2 Random Maze Solver

Speed-Up curves are evaluated using times for a single upload for each particle (maze 300×300).



Results

2 Random Maze Solver

Results are collected with: $BlockDim = 256$ $d_{Man}(P_{start}, P_{exit}) = 1.5L.$