K-Means

Progetto Long-Term

Andrea Neri

Obiettivo del progetto

- Implementazione di una versione sequenziale e due versioni parallele dell'algoritmo di clustering k-means
- Versione sequenziale e parallela con OpenMP
- Versione parallela con CUDA
- Visualizzazione grafica dei risultati ottenuti
- Valutazione performance e speedup

Clustering K-Means (1)

- Algoritmo di apprendimento non supervisionato che, dato in input un insieme di dati e un numero K di gruppi, si occupa di dividere i data points a seconda della presenza o meno di una certa somiglianza tra di loro.
- La somiglianza tra due punti può essere definita, in uno spazio a due dimensioni, dalla distanza euclidea.
- Ogni cluster avrà un punto particolare detto centroide che rappresenta il centro geometrico dell'insieme.
- Al termine dell'esecuzione dell'algoritmo tutti i punti saranno assegnati ad un solo cluster e non ci saranno punti non assegnati.

Clustering K-Means (2)

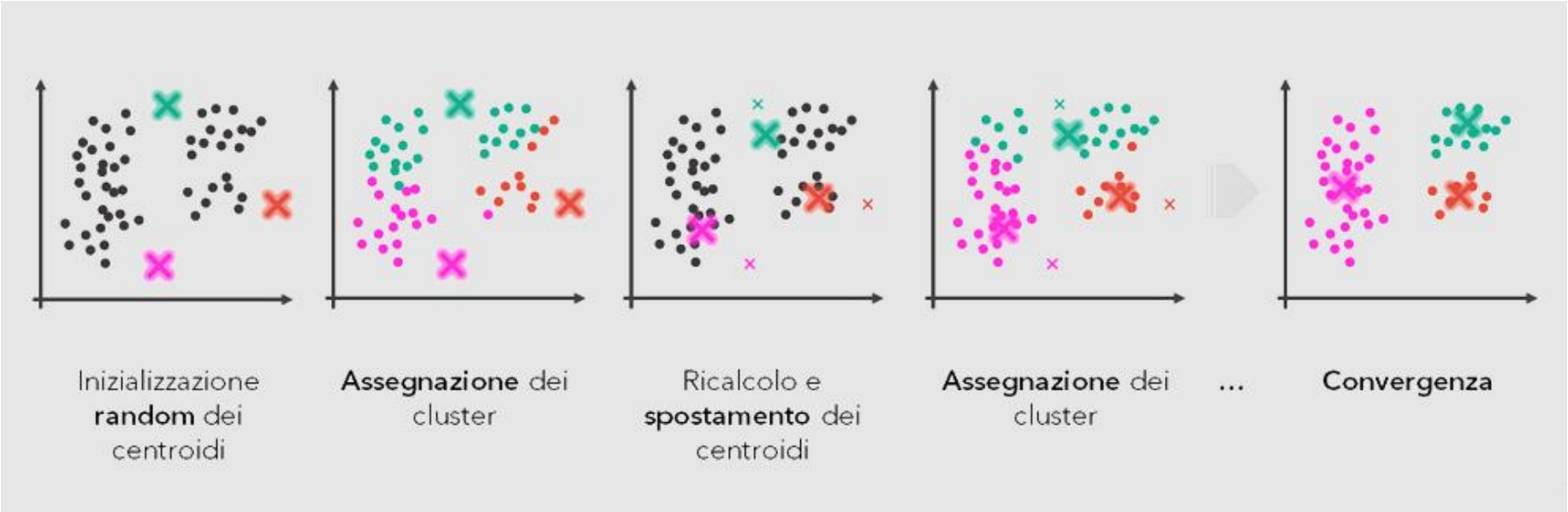
L'algoritmo può essere rappresentato dalle seguenti fasi:

1. **Inizializzazione:** si definisce il parametro K e il dataset su cui l'algoritmo andrà a lavorare;

2. **Assegnazione dei punti ad un cluster:** ogni punto del dataset viene assegnato al centroide più vicino;

3. **Ricalcolo della posizione del centroide:** si ricalcola la posizione del centroide come la media di tutti i data points appartenenti a quel cluster.

Clustering K-Means (3)



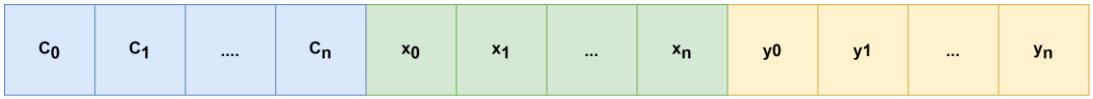
Rappresentazione di punti e cluster

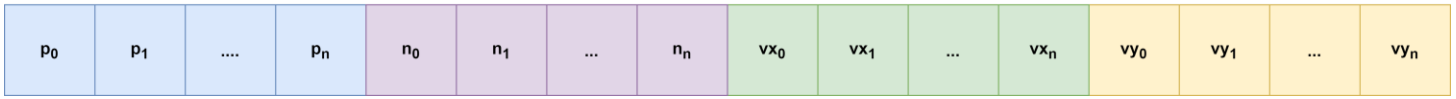
Un punto in uno spazio a 2 dimensioni può essere rappresentato utilizzando 3 elementi:

- c: il cluster a cui verrà assegnato
- x: la coordinata nello spazio relativa all'asse x
- y: la coordinata nello spazio relativa all'asse y

Un cluster in uno spazio a 2 dimensioni può essere rappresentato utilizzando 4 elementi:

- p: definisce il punto che rappresenta il centroide
- n: il numero di punti assegnato al cluster
- vx: valore che rappresenta la somma delle componenti di ogni singolo punto del cluster lungo l'asse x (valore utilizzato per il ricalcolo del centroide)
- vy: valore che rappresenta la somma delle componenti di ogni singolo punto del cluster lungo l'asse y (valore utilizzato per il ricalcolo del centroide)





Calcolo della distanza

- Distanza euclidea:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
float euclideanDistance(float x1, float y1, float x2, float y2) {
        return sqrt(pow((x1-x2),2) +pow((y1-y2),2));
}
```

- Distanza di Manhattan:

$$d(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

```
float manhattanDistance(float x1, float y1, float x2, float y2) {
        return abs(x1-x2)+abs(y1-y2);
}
```

Implementazione con OpenMP

Assegnamento dei punti ai cluster

```
void assignPointToCluster(float* points, float* clusters) {
#pragma omp parallel for default(none) shared(points, clusters) num_threads(THREAD_NUMBER) schedule(static)
  for (int index=0; index < POINT_NUMBER; index++) {
    float x_cluster, y_cluster = 0;
    float x_point = points[1 * POINT_NUMBER + index];
    float y_point = points[2 * POINT_NUMBER + index];
    unsigned int best_fitting = 0;
    float old_distance = FLT_MAX;
    float actual_distance = 0;

    for (int i = 0; i < CLUSTER_NUMBER; i++) {
      unsigned int centroid_index = (int) clusters[0 * CLUSTER_NUMBER + i];
       x_cluster = points[1 * POINT_NUMBER + centroid_index];
       y_cluster = points[2 * POINT_NUMBER + centroid_index];

       if (DISTANCE == 0) {
          actual_distance = euclideanDistance(x_point, y_point, x_cluster, y_cluster);
       } else {
          actual_distance = manhattanDistance(x_point, y_point, x_cluster, y_cluster);
       }
        if (actual_distance < old_distance) {
          best_fitting = i;
          old_distance = actual_distance;
       }
    }
    points[0 * POINT_NUMBER + index] = (float) best_fitting;
#pragma omp atomic
    clusters[1*CLUSTER_NUMBER+best_fitting]=clusters[1*CLUSTER_NUMBER+best_fitting]+1;
  }
}
```

Ricalcolo del centroide

```
void calculateValue(float* points, float* clusters){
#pragma omp parallel for default(none) shared(clusters,points) num_threads(THREAD_NUMBER) schedule(static)
        for (int i = 0; i < POINT_NUMBER; i++) {
                unsigned int cluster_n=points[0 * POINT_NUMBER + i];
        #pragma omp atomic
                clusters[2*CLUSTER_NUMBER+cluster_n=clusters[2*CLUSTER_NUMBER+cluster_n]+points[1*POINT_NUMBER+i];

        #pragma omp atomic
                clusters[3*CLUSTER_NUMBER+cluster_n=clusters[3*CLUSTER_NUMBER+cluster_n]+points[2*POINT_NUMBER+i];
        }
}


void recomputeCentroid(float* points, float* clusters) {

#pragma omp parallel for default(none) shared(clusters,points) num_threads(THREAD_NUMBER) schedule(dynamic)
        for (int i = 0; i < CLUSTER_NUMBER; i++) {
                float centroid_x = clusters[2*CLUSTER_NUMBER+i]/clusters[1*CLUSTER_NUMBER+i];
                float centroid_y = clusters[3*CLUSTER_NUMBER+i]/clusters[1*CLUSTER_NUMBER+i];

                unsigned int centroid_index=(int) clusters[0*CLUSTER_NUMBER+i];
                points[1*POINT_NUMBER+centroid_index]= centroid_x;
                points[2*POINT_NUMBER+centroid_index]= centroid_y;

        }
}
```

Rimozione dei punti dal cluster

```
void removePoint(float* clusters) {
#pragma omp parallel for default(none)shared(clusters)num_threads(THREAD_NUMBER)
   for (int i = 0; i < CLUSTER_NUMBER; i++) {
      clusters[1*CLUSTER_NUMBER+i] = 0;
      clusters[2*CLUSTER_NUMBER+i] = 0;
      clusters[3*CLUSTER_NUMBER+i] = 0;
   }
}
```

Implementazione con CUDA

Assegnamento dei punti ai cluster

```
__global__ void assignPointToCluster(float* points, float* clusters) {
        unsigned int point_n = threadIdx.x + blockIdx.x * blockDim.x;
        if (point_n < POINT_NUMBER) {
                float x_cluster, y_cluster = 0;
                float x_point = points[1 * POINT_NUMBER + point_n];
                float y_point = points[2 * POINT_NUMBER + point_n];
                unsigned int best_fitting = 0;
                float old_distance = FLT_MAX;
                float actual_distance = 0;

                for (int i = 0; i < CLUSTER_NUMBER; i++) {
                        unsigned int centroid_index = clusters[0 * CLUSTER_NUMBER + i];
                        x_cluster = points[1 * POINT_NUMBER + centroid_index];
                        y_cluster = points[2 * POINT_NUMBER + centroid_index];

                        if (DISTANCE == 0) {
                                actual_distance = euclideanDistance(x_point, y_point, x_cluster, y_cluster);
                        }
                        else
                        {
                                actual_distance= manhattanDistance(x_point, y_point, x_cluster, y_cluster);
                        }
                        if (actual_distance < old_distance) {
                                best_fitting = i;
                                old_distance = actual_distance;
                        }
                }
                points[0 * POINT_NUMBER + point_n] = best_fitting;
                atomicAdd(&clusters[1 * CLUSTER_NUMBER + best_fitting], 1);
        }
}
```

Ricalcolo del centroide

```
__global__ void calculateValue(float* points, float* clusters) {
        unsigned int point_n = threadIdx.x+ blockIdx.x*blockDim.x;
        if (point_n < POINT_NUMBER) {
                unsigned int cluster_n = points[0* POINT_NUMBER+point_n];
                atomicAdd(&clusters[2*CLUSTER_NUMBER+ cluster_n], points[1*POINT_NUMBER+ point_n]);
                atomicAdd(&clusters[3*CLUSTER_NUMBER +cluster_n], points[2*POINT_NUMBER+ point_n]);
  }
}
```

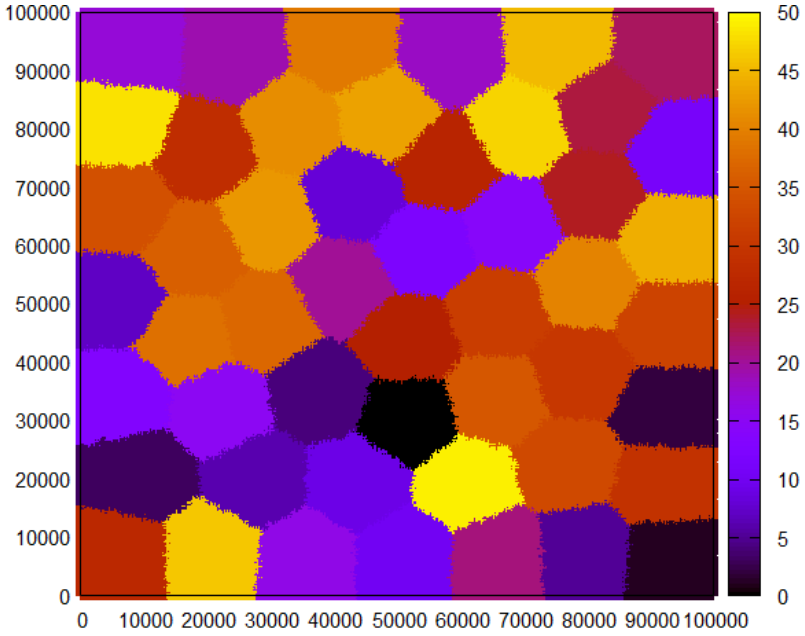
```
__global__ void recomputeCentroid(float* points, float* clusters) {
        unsigned int cluster_n=threadIdx.x+blockIdx.x* blockDim.x;
        float centroid_x=clusters[2*CLUSTER_NUMBER+ cluster_n]/clusters[1*CLUSTER_NUMBER+cluster_n];
        float centroid_y=clusters[3*CLUSTER_NUMBER+ cluster_n]/clusters[1*CLUSTER_NUMBER+cluster_n];
        unsigned int cluster_index=(unsigned int)clusters[0 *CLUSTER_NUMBER+cluster_n];
        points[1*POINT_NUMBER+cluster_index]=centroid_x;
        points[2*POINT_NUMBER+cluster_index]=centroid_y;
}
```

Rimozione dei punti dal cluster

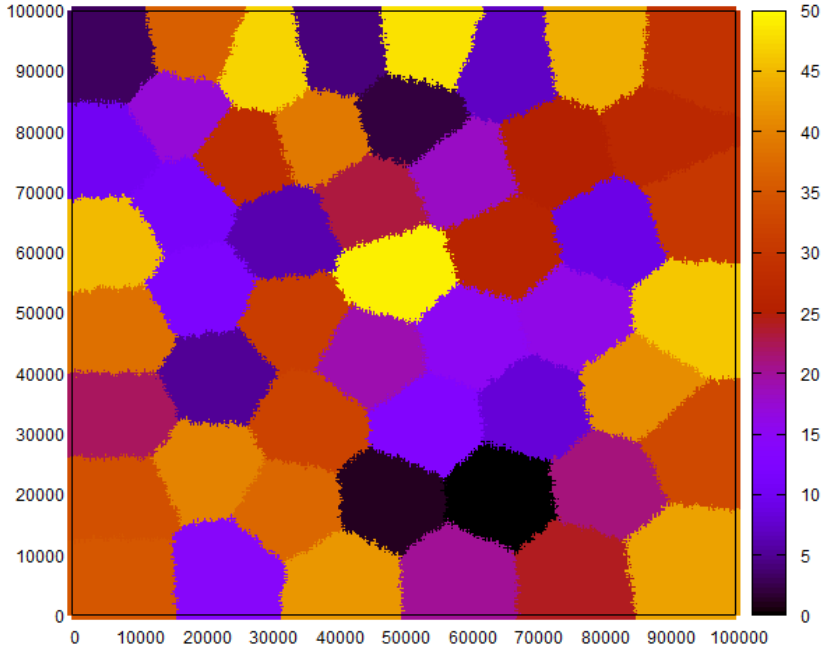
```
__global__ void removePoint(float* clusters) {
        unsigned int cluster_n=threadIdx.x + blockIdx.x*blockDim.x;

        clusters[1 * CLUSTER_NUMBER+ cluster_n]=0;

        clusters[2 * CLUSTER_NUMBER+ cluster_n]=0;

        clusters[3 * CLUSTER_NUMBER+ cluster_n]=0;
}
```

Grafici a confronto

5 milioni di punti e 50 cluster



OpenMP



CUDA

Risultati OpenMP

Versione sequenziale

cluster number	10	15	20	30	50	100
point number						
5000	15,63	31,25	31,25	31,25	46,88	93,75
50000	125,01	156,26	188,57	266,70	437,54	860,48
500000	1315,88	1488,84	1870,74	2929,62	4308,55	8373,58
5000000	13440,10	15801,70	20016,80	28244,10	45006,20	85179,70
10000000	25895,60	31957,70	39570,60	56504,00	88307,00	172595,00

Versione parallela (12 thread)

cluster number	10	speedup	15	speedup	20	speedup	30	speedup	50	speedup	100	speedup
point number												
5000	31,25	0,50	31,25	1,00	31,25	1,00	46,88	0,67	31,25	1,50	31,25	3,00
50000	218,77	0,57	172,98	0,90	140,64	1,34	157,34	1,70	156,26	2,80	187,51	4,59
500000	1640,66	0,80	1693,30	0,88	1568,25	1,19	1410,59	2,08	1332,77	3,23	1579,29	5,30
5000000	16652,10	0,81	17634,10	0,90	15567,60	1,29	14782,40	1,91	14487,80	3,11	16968,70	5,02
10000000	33039,40	0,78	34338,70	0,93	29679,30	1,33	26522,30	2,13	25520,40	3,46	30037,50	5,75

Risultati CUDA

Versione sequenziale OpenMP

cluster number	10	15	20	30	50	100
point number						
5000	15,63	31,25	31,25	31,25	46,88	93,75
50000	125,01	156,26	188,57	266,70	437,54	860,48
500000	1315,88	1488,84	1870,74	2929,62	4308,55	8373,58
5000000	13440,10	15801,70	20016,80	28244,10	45006,20	85179,70
10000000	25895,60	31957,70	39570,60	56504,00	88307,00	172595,00

Versione parallela CUDA (256 thread per block)

cluster number	10	speedup	15	speedup	20	speedup	30	speedup	50	speedup	100	speedup
point number												
5000	8,79	1,78	12,02	2,60	12,29	2,54	10,73	2,91	15,94	2,94	17,68	5,30
50000	18,44	6,78	27,91	5,60	33,11	5,69	35,11	7,60	51,25	8,54	89,55	9,61
500000	114,69	11,47	157,94	9,43	175,45	10,66	237,81	12,32	307,78	14,00	586,91	14,27
5000000	981,43	13,69	1198,61	13,18	1290,92	15,51	1840,52	15,35	2874,60	15,66	5288,10	16,11
10000000	1666,71	15,54	2173,28	14,70	2412,61	16,40	3545,59	15,94	5553,89	15,90	10232,50	16,87

Risultati CUDA