Implementazione sequenziale e parallela del K-means

Guido Ciardi 7090798
guido.ciardi@stud.unifi.it

Esame orale di Parallel Computing

Università degli studi di Firenze

Firenze, 12/01/2023

Pseudocodice algoritmo di clustering K-means

Input: Dataset di "numPoints" punti e "numClusters" centroidi;
Output: Insieme di "numClusters" clusters;

Steps:
Per ogni punto p_i del dataset considerato:
Ripeti:
1. Calcola la distanza da ogni punto ad ogni centroide;
2. Inserisci tale punto all'interno del cluster del centroide ad esso più vicino;

Per ogni centroide:
   a) Aggiorna le sue coordinate facendo la media delle rispettive coordinate di ogni punto presente nel cluster attualmente considerato;

Finchè:
Le coordinate dei centroidi non cambiano più;
(Criterio di convergenza)

Codice sequenziale

1. Inizializzazione dei punti/centroidi;
2. Funzione k_means:
   a) Aggiornamento dele coordinate dei centroidi;
      - Calcolo distanza Euclidea;

   b) Funzione newCentroidCoordinate;

3. Stampa dei risultati;

Inizializzazione di punti/centroidi

```
point point_Initialization(double x, double y){
    point p = {x, y, .clusterId: -1};
    return p;
}
```

- Caso punto standard → clusterId = -1;
- Caso punto centroide → clusterId = numero del cluster

```
typedef struct{
    double x;
    double y;
    int clusterId;
}point;
```

Logica di memorizzazione dei punti usata: **Arrays of Structures (AoS)**

Funzione K_means

- **Condizione di convergenza:**

```
while(convergence == 0){
```

- **Calcolo del centroide più vicino al punto attuale:**

```
for (int i = 0; i < numPoints; i++) { // Scansione dei punti

    centroidIndex = nearestCentroid( actualPoint: points[i], clusters);

    points[i].clusterId = centroidIndex; // Il punto "i" viene assc
```

- **Fase di aggiornamento delle coordinate dei centroidi:**

```
if(newX != clusters[centroid].x || newY != clusters[centroid].y){
    clusters[centroid].x = newX;
    clusters[centroid].y = newY;
    convergence = 0;
}
```

Funzione nearestCentroid

```
int nearestCentroid(point actualPoint, point clusters[]){
    double distance;
    int centroidIndex = 0;

    // Confronto del punto col primo centroide per inizializzare la distanza
    double minDistance = euclideanDistance( a: actualPoint,  b: clusters[0]);
    // minDistance viene usata per mantenersi la distanza minore ed individu
    // appartenenza di un determinato punto


    for (int centroid = 1; centroid < numClusters; centroid++) { // Scansion
        distance = euclideanDistance( a: actualPoint,  b: clusters[centroid]);

        if (distance < minDistance) {
            minDistance = distance;
            centroidIndex = centroid;
        }

    }

    return centroidIndex;
}
```

Aggiornamento delle coordinate dei centroidi

```
for(int centroid = 0; centroid < numClusters; centroid++){
    newX = 0;
    newY = 0;
    numPointsOfCluster = 0;


    for(int i = 0; i < numPoints; i++){
        if(points[i].clusterId == centroid){
            newX = newX + points[i].x;
            newY = newY + points[i].y;
            numPointsOfCluster++;
        }
```

```
if(numPointsOfCluster != 0){
    newX = newX / numPointsOfCluster;
    newY = newY / numPointsOfCluster;
}
else{
    newX = clusters[centroid].x;
    newY = clusters[centroid].y;
}
```

```
if(newX != clusters[centroid].x || newY != clusters[centroid].y){
    clusters[centroid].x = newX;
    clusters[centroid].y = newY;
    convergence = 0;
}
```

Esempio di output in un'esecuzione sequenziale

```
Cluster numero: 0
Lista dei punti associati al centroide (1.000000, 5.000000):
(0.000000, 6.000000)  (1.000000, 5.000000)  (2.000000, 4.000000)

Cluster numero: 1
Lista dei punti associati al centroide (3.500000, 2.500000):
(3.000000, 3.000000)  (4.000000, 2.000000)

Cluster numero: 2
Lista dei punti associati al centroide (5.000000, 1.000000):
(5.000000, 1.000000)

Tempo di esecuzione dell'algoritmo: 0.019000
```

Codice parallelo

Introduzione della macro: `#define nThreads 8`

Inizializzazione parallela dei punti:

```
#pragma omp parallel num_threads(nThreads)
    {
#pragma omp for schedule(auto) nowait
        for(int i = 0; i < numPoints; i++){
            points[i] = point_Initialization( x: i,  y: numPoints - i);
        }
        // Si utilizza nowait perchè i due cicli sono indipendenti e non
        // La schedule(auto) decide se schedulare/distribuire i valori t

#pragma omp for schedule(auto)
        for(int i = 0; i < numClusters; i++){
            clusters[i] = point_Initialization( x: (i * numClusters) % (n
            clusters[i].clusterId = i;
        }
    }
```

Funzione k_means parallela

- ***Calcolo del cluster di appartenenza per ciascun punto:***

```
#pragma omp parallel num_threads(nThreads)
        {
#pragma omp for schedule(auto) private(centroidIndex)
            for (int i = 0; i < numPoints; i++) { // Scansione dei punti

                centroidIndex = nearestCentroid( actualPoint: points[i], clusters);

                points[i].clusterId = centroidIndex; // Il punto "i" viene asso

            }
```

- ***Aggiornamento delle coordinate dei centroidi:***

```
#pragma omp for schedule(auto) private(newX, newY)
            for(int centroid = 0; centroid < numClusters; centroid++){
                newX = 0;
                newY = 0;
                numPointsOfCluster = 0;


                for(int i = 0; i < numPoints; i++){
                    if(points[i].clusterId == centroid){
                        newX = newX + points[i].x;
                        newY = newY + points[i].y;
```

Stampe finali parallelizzate

```
#pragma omp parallel for schedule(auto) num_threads(nThreads)
        for(int point = 0; point < numPoints; point++) {
            if(points[point].clusterId == i){
                printf("(%f, %f)  ", points[point].x, points[point].y);
            }
        }
    }
```

Confronto delle prestazioni

numPoints = 10000; numClusters = 150;

(Per il caso parallelo anche:) nThreads = 8;

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	CASO PARALLELO
14.582	2.006

$$Speedup = \frac{T_S}{T_P} = \frac{14.582}{2.006} = 7.2692$$

Aumento del numero di centroidi

Nota: La parallelizzazione viene fatta per centroidi, quindi all'aumentare del numero di centroidi si avrà un aumento dello speedup.

numPoints = 10000; numClusters = 200; (Per il caso parallelo anche:) nThreads = 8;

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	**CASO PARALLELO**
46.489	3.895

$$Speedup = \frac{T_S}{T_P} = \frac{46.489}{3.895} = 11.93556$$

Confronto delle prestazioni

Tempi di esecuzione con numPoints = 10000 e numClusters = 200	
Sequenziale	46.489
nThreads = 2	17.689
nThreads = 4	11.288
nThreads = 6	8.423
nThreads = 8	**3.895**
nThreads = 10	9.965