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) Funzione nearestCentroid;
      - 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 asso
```

- **Aggiornamento delle coordinate dei centroidi:**

```
for(int centroid = 0; centroid < numClusters; centroid++){
    x = newCentroidCoordinate(points, centroid, referenceCoordinate: clusters[centroid].x, whichCoordinate: 0);
    y = newCentroidCoordinate(points, centroid, referenceCoordinate: clusters[centroid].y, whichCoordinate: 1);

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

Funzione nearestCentroid

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

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

Funzione newCentroidCoordinate

```
if (whichCoordinate == 0){  // Caso coordinata x
    for(int i = 0; i < numPoints; i++){
        if(points[i].clusterId == centroid){
            newCoordinate = newCoordinate + points[i].x;
            numPointsOfCluster++;
        }
```

x → 0;
y → 1;

```
return newCoordinate / numPointsOfCluster;
```

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(x,y)
          for(int centroid = 0; centroid < numClusters; centroid++){
              x = newCentroidCoordinate(points, centroid, referenceCoordinate: clusters[centroid].x, w
              y = newCentroidCoordinate(points, centroid, referenceCoordinate: clusters[centroid].y, w

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

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 = 4;

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	CASO PARALLELO
14.58200	9.523

$$Speedup = \frac{T_S}{T_P} = \frac{14.58200}{9.523} = 1.53124$$

Confronto delle prestazioni

Tempi di esecuzione con numPoints = 10000 e numClusters = 150	
Sequenziale	14.58200
nThreads = 1	14.53600
nThreads = 2	11.17300
nThreads = 3	10.13700
nThreads = 4	9.52300
nThreads = 5	9.47200
nThreads = 6	9.15100
nThreads = 7	9.03700
nThreads = 8	**8.85500**
nThreads = 9	9.36600
nThreads = 10	9.37300
nThreads = 11	9.38100