Mid Term Presentation: K-means Algorithm

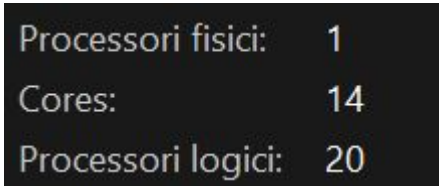
Parallel Programming

Gianluca Giuliani

Luogo e data arial

Dispositivo

Il dispositivo su cui viene eseguito il codice contiene un processore intel i7 con 14 cores e 20 processori logici



Algoritmo

- Inizializzazione
  Inizializzo casualmente i centroidi dei cluster
- Assegnazione
  Assegna ogni punto al centroide più vicino.
- Aggiornamento
  Calcola nuovi centroidi come la media dei punti assegnati.

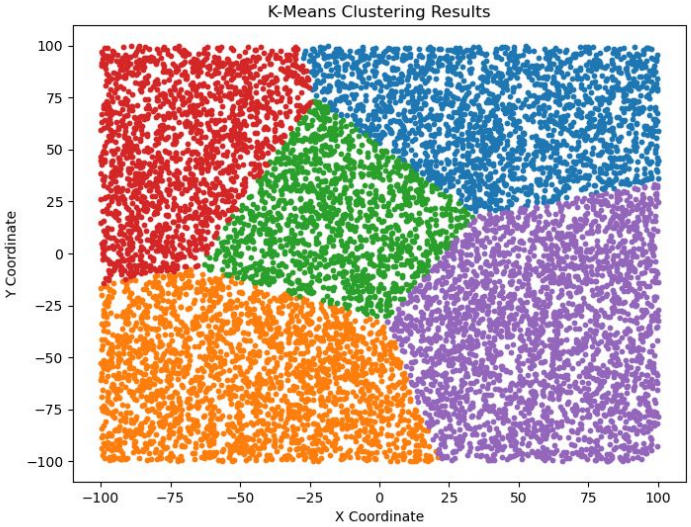
Ripeto Assegnazione e Aggiornamento per un certo numero, maxIteration, volte.

Algoritmo

- Inizializzazione
  - Inizializzo casualmente i centroidi dei cluster
- Assegnazione
  - Assegna ogni punto al centroide più vicino.
- Aggiornamento
  - Calcola nuovi centroidi come la media dei punti assegnati.

Ripeto Assegnazione e Aggiornamento per un certo numero, maxIteration, volte.

Risultati



Kmenas con 1000 punti e 5 cluster

Parametri

- Numero di Punti
- Numero di Cluster
- Numero massimo di iterazioni
- Numero di threads

SoA vs AoS

- In SoA, Structure of Arrays, i dati sono separati in array distinti, uno per ogni attributo.

  Questo mi permette di ottimizzare operazioni SIMD e favorisce il parallelismo

- In AoS ogni elemento è una struttura che rappresenta un oggetto completo.

  Di contro avrò un accesso alla cache più lento.

Code Snippets: AoS - Cluster Point.h

```
class Cluster {
    float x, y;
    float sum_x, sum_y;
    int size;

public:
    Cluster(float x_val = 0, float y_val = 0){
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<> dis(-100.0, 100.0);

        x = dis(gen);
        y = dis(gen);
        sum_x= 0;
        sum_y= 0;
        size = 0;
    }

    void updateCenter() {
        if (size > 0) {
            x = sum_x / size;
            y = sum_y / size;
        }
    }

    void reset() {
        sum_x = 0;
        sum_y = 0;
        size = 0;
    }

    void updateCenter(float sum_x, float sum_y, int count) {
        if (count > 0) {
            x = sum_x / count;
            y = sum_y / count;
        }
    }
```

```
class Point {
    float x, y;
    int cluster_id;
```

Code Snippets: AoS - Cluster Point.h

```
class Point {
    float x, y;
    int cluster_id;
```

Code Snippets: AoS AoS.h

```
std::vector<Cluster> clusters(num_clusters);
for (int i = 0; i < num_clusters; ++i) {
    clusters[i] = Cluster();
}

#pragma omp parallel for
for (int iter = 0; iter < max_iterations; ++iter) {
    // Assignment Step
    #pragma omp parallel for schedule(dynamic, 1000)
    for (size_t i = 0; i < points.size(); ++i) {
        float min_distance = std::numeric_limits<float>::max();
        int best_cluster = -1;

        for (size_t j = 0; j < clusters.size(); ++j) {
            float distance = points[i].distanceTo(clusters[j].getX(), clusters[j].getY());
            if (distance < min_distance) {
                min_distance = distance;
                best_cluster = j;
            }
        }
        points[i].setClusterId(best_cluster);
    }

    float* cluster_sum_x = new float[clusters.size()]();
    float* cluster_sum_y = new float[clusters.size()]();
    int* cluster_count = new int[clusters.size()]();
```

Code Snippets: AoS AoS.h

```
// Reset the clusters before accumulation
#pragma omp parallel for
for (size_t i = 0; i < clusters.size(); ++i) {
    clusters[i].reset();
}

// Accumulate
#pragma omp parallel for reduction(+:cluster_sum_x[:clusters.size()], cluster_sum_y[:clusters.size()], cluster_count[:clusters.size()])
for (size_t i = 0; i < points.size(); ++i) {
    int cluster_id = points[i].getClusterId();

    cluster_sum_x[cluster_id] += points[i].getX();
    cluster_sum_y[cluster_id] += points[i].getY();
    cluster_count[cluster_id]++;
}

// Update the cluster
#pragma omp parallel for
for (size_t i = 0; i < clusters.size(); ++i) {
    clusters[i].updateCenter(cluster_sum_x[i], cluster_sum_y[i], cluster_count[i]);
}

delete[] cluster_sum_x;
delete[] cluster_sum_y;
delete[] cluster_count;
```

Code Snippets: SoA Utility.h

```
struct alignas(64) PointsSoA {
    std::vector<float> x;
    std::vector<float> y;

    PointsSoA(size_t numPoints) : x(numPoints), y(numPoints) {}

    float getX(size_t index) const { return x[index]; }
    float getY(size_t index) const { return y[index]; }

    Codeium: Refactor | Explain | Generate Function Comment | ×
    void set(size_t index, float xVal, float yVal) {
        x[index] = xVal;
        y[index] = yVal;
    }
};
```

Code Snippets: SoA SoA.h

```
std::vector<float> centroidsX(k), centroidsY(k);
#pragma omp parallel
{
std::random_device rd;
std::mt19937 gen(rd() ^ omp_get_thread_num());
std::uniform_real_distribution<float> dis(-100.0f, 100.0f);

#pragma omp for
for (int i = 0; i < k; i++) {
    centroidsX[i] = dis(gen);
    centroidsY[i] = dis(gen);
}
}
```

Code Snippets: SoA SoA.h

```
#pragma omp parallel shared(newCentroidsX, newCentroidsY, counts)
{
    for (int iter = 0; iter < maxIterations; iter++) {

        // Parallelize point-to-centroid assignment
        #pragma omp for schedule(static,500)
        for (size_t i = 0; i < numPoints; i++) {
            int bestCluster = -1;
            float bestDistance = std::numeric_limits<float>::max();
            for (int j = 0; j < k; j++) {
                float d = distance(points, i, centroidsX[j], centroidsY[j]);
                if (d < bestDistance) {
                    bestDistance = d;
                    bestCluster = j;
                }
            }
            labels[i] = bestCluster;
        }
```

Code Snippets: SoA SoA.h

```
        // Parallelize centroid update with reduction
        #pragma omp for reduction(+:newCentroidsX, newCentroidsY, counts)
        for (size_t i = 0; i < numPoints; i++) {
            int cluster = labels[i];
            newCentroidsX[cluster] += points.getX(i);
            newCentroidsY[cluster] += points.getY(i);
            counts[cluster]++;
        }

        // Update centroids after parallel computation
        #pragma omp for
        for (int j = 0; j < k; j++) {
            if (counts[j] > 0) {
                centroidsX[j] = newCentroidsX[j] / counts[j];
                centroidsY[j] = newCentroidsY[j] / counts[j];
            }
        }
    }
}
```

didascalia arial regular 12 pt allineamento a sinistra o giustificato didascalia arial regular 12 pt allineamento a sinistra o giustificato didascalia arial regular 12 pt allineamento a sinistra o giustificato