K-Means con OpenMP

Gregorio Piqué

February 13, 2024

1 Introduzione

Il seguente documento costituisce il resoconto relativo al progetto intermedio del corso di *Parallel Programming for Machine Learning*, parte integrante del curriculum di Laurea Magistrale in Intelligenza Artificiale presso l'Università degli Studi di Firenze. La realizzazione del progetto ha coinvolto l'implementazione di un elaborato a scelta tra diverse opzioni, richiedendo anche la redazione di una relazione conclusiva che riflettesse e illustrasse i risultati ottenuti.

Il progetto selezionato ha richiesto l'implementazione di una versione parallela di *K-Means*[1], noto algoritmo di clustering, usando l'API *OpenMP*[2].

2 KMeans

L'algoritmo K-means è un metodo di clustering utilizzato nell'ambito del machine learning e dell'analisi dei dati. L'obiettivo principale del K-means è raggruppare un insieme di dati in cluster (o nuvole di punti), dove ogni cluster è caratterizzato da una media, chiamata centroide, che rappresenta in modo approssimato le osservazioni nel cluster [1].

Il funzionamento dell'algoritmo K-means è suddiviso in quattro fasi principali:

1. **Inizializzazione dei Centroidi:** Vengono scelti casualmente K punti come i centroidi iniziali dei K cluster (eventualmente anche scelti come punti del dataset iniziale).
2. **Assegnazione dei Punti ai Cluster:** Ogni punto nel dataset viene assegnato al cluster il cui centroide è più vicino, calcolando la distanza secondo un qualche criterio (spesso utilizzando la distanza euclidea).
3. **Aggiornamento dei Centroidi:** I centroidi dei cluster vengono aggiornati calcolando la media dei punti appartenenti a ciascun cluster.
4. **Ripetizione:** I passaggi 2 e 3 vengono ripetuti fino a che l'algoritmo converge o fino al raggiungimento del numero massimo prefissato di iterazioni.

L'algoritmo converge quando i centroidi non cambiano posizione in modo significativo tra iterazioni consecutive. L'output dell'algoritmo K-means è un insieme di cluster, ognuno caratterizzato dal suo centroide. La Figura 1 mostra il flusso esecutivo del K-Means generale, mentre la Figura 2 presenta un esempio del funzionamento dell'algoritmo implementato.

3 Metodologia

Come presentato nella precedente Sezione 2, il funzionamento dell'algoritmo K-Means prevede di iterare sui passaggi 2 e 3 di assegnazione di punti ai cluster e di aggiornamento dei centroidi fino a che l'algoritmo non converga o non si sia raggiunto il numero massimo di iterazioni prefissato. L'iniziale assegnazione dei centroidi può essere effettuata in diverse modalità, come ad esempio la scelta di coordinate casuali, o la selezione di punti dal dataset assegnando i centroidi con i punti scelti. In generale, l'inizializzazione dei centroidi utilizzando punti del dataset potrebbe offrire prestazioni migliori rispetto alla selezione di coordinate casuali. Ciò è dovuto al fatto che i centroidi vengono posizionati già in prossimità dei dati, accelerando potenzialmente la convergenza dell'algoritmo K-means. La soluzione ottenuta però potrebbe essere il risultato di un minimo locale: per questo motivo, varie librerie che implementano l'algoritmo ripetono varie volte l'esecuzione riportando alla fine il miglior risultato ottenuto (si veda la Figura 3).

L'implementazione dell'algoritmo portata avanti segue in parte questa modalità di esecuzione, ripetendo il processo di clustering K-Means per un numero predefinito (ma configurabile) di esecuzioni. Ad ogni esecuzione dell'algoritmo, i centroidi vengono riinizializzati con coordinate corrispondenti a punti selezionati casualmente dal dataset. Le soluzioni "intermedie" sono memorizzate in un vettore dedicato e, al termine del processo, viene identificata e restituita la soluzione ottimale trovata, caratterizzata dal valore complessivo di *SSE* (Sum of Squared Errors) più basso.

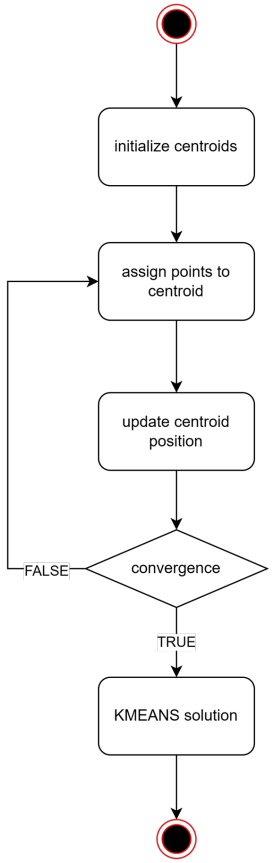
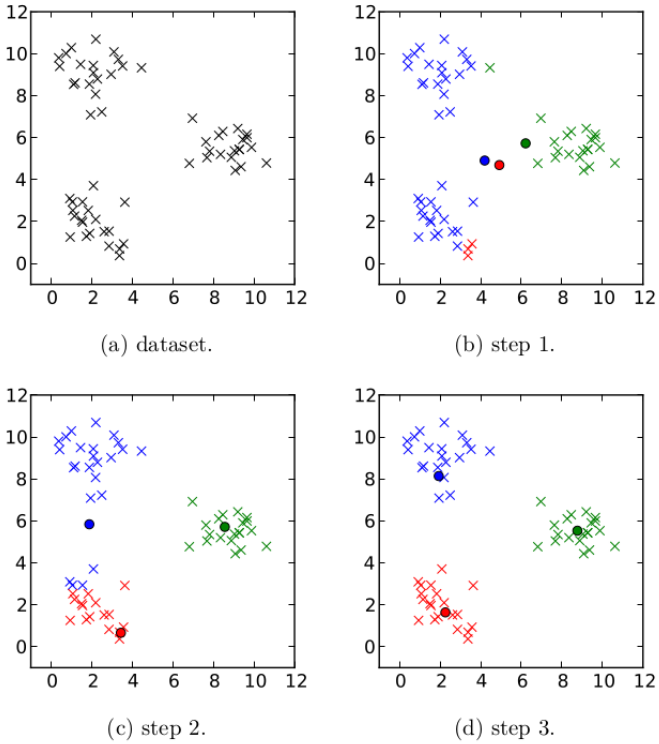


Figure 1: Flusso esecutivo di K-Means.



(a) dataset. (b) step 1.

(c) step 2. (d) step 3.

Figure 2: Esempio di clustering con K-Means.

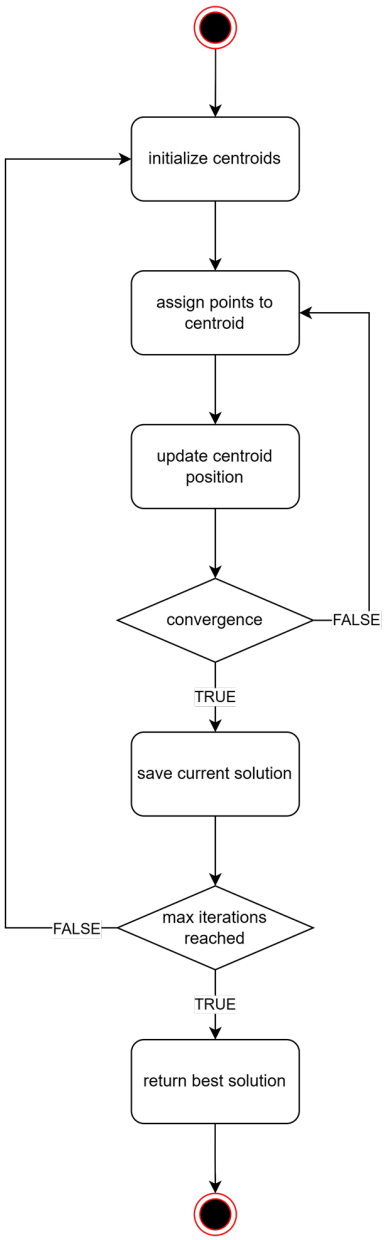


Figure 3: Flusso di K-Means con ripetizione.

4 Implementazione

Nella soluzione implementata [3], sono state sviluppate e utilizzate classi di supporto per la gestione dei dati e delle funzionalità dell'algoritmo. Le principali classi implementate sono *Points* e *Centroids* per la gestione dei punti e dei centroidi, *Solution* per memorizzare le soluzioni intermedie, *Utils* per operazioni utili al processo generale, mentre la classe *Kmeans* offre le funzionalità dell'algoritmo stesso.

4.1 Classi

Points

La classe *Points* consente la gestione dei punti e include campi come *totalPoints*, indicante il numero totale di punti considerati, *posX* e *posY* che rappresentano le coordinate in due dimensioni dei punti, *ids* la sequenza di identificatori dei punti (una sequenza ordinata crescente di interi), e *centroidsIds* che associa ciascun punto al centroide corrispondente. Oltre a costruttori e distruttori, i metodi a disposizione sono *addPoint* e *addPoints*, i quali permettono di aggiugnere uno o più punti. La classe gestisce i dati rappresentanti i punti del problema con un layout di tipo *SoA* (Structure of Arrays). La Figura 4 presenta la struttura della classe *Points*.

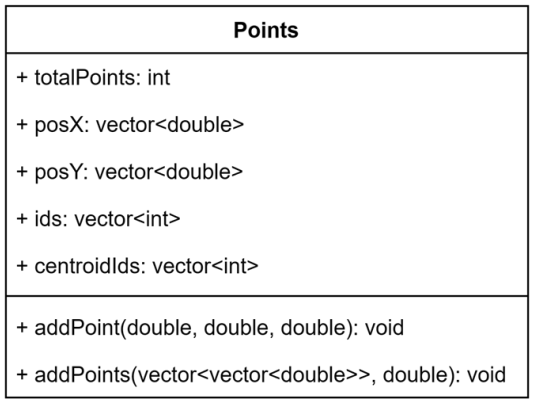


Figure 4: Classe Points.

Centroids

La classe *Centroids* permette di gestire i centroidi. Include campi quali *totalCentroids*, che indica il numero totale di centroidi, *posX* e *posY* che rappresentano le loro coordinate in due dimensioni e *ids* la sequenza di identificatori dei centroidi (una sequenza ordinata crescente di interi). Oltre a costruttori e distruttori, i metodi a disposizione sono *addCentroid* per l'aggiunta di centroidi e *clear* che permette di rimuovere tutti i centroidi attualmente considerati, per poi riinizializzarli in modo random (si veda il funzionamento dell'algoritmo alla Sezione 3). La classe gestisce i dati rappresentanti i centroidi con un layout di tipo *SoA* (Structure of Arrays). La Figura 5 presenta la struttura della classe *Centroids*.

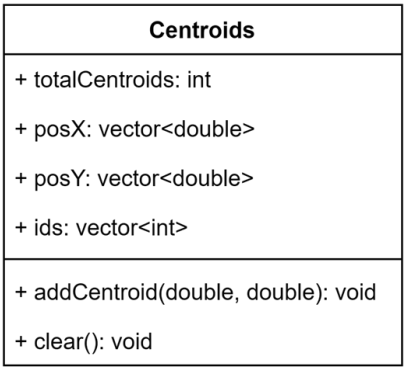


Figure 5: Classe Centroids.

Kmeans

La classe *Kmeans* fornisce le funzionalità dell'algoritmo K-Means. Oltre ai campi come *points* e *centroids*, include *xCoordMax* e *yCoordMax*, che rappresentano il valore massimo delle coordinate di punti e centroidi. Altri campi significativi sono *maxIterations*, che determina quando interrompere l'algoritmo nel caso in cui non converga, e *kmeansExecutions*, che definisce il numero totale di esecuzioni di K-Means da effettuare e quindi quante volte ripetere l'algoritmo riinizializzando i centroidi.

Oltre ai costruttori e distruttori, gli unici metodi disponibili sono *runParallel()* e *runSequential()*, utilizzati per avviare l'esecuzione dell'algoritmo rispettivamente in modo parallelo (specificando il numero di thread da utilizzare) o sequenziale. L'implementazione delle due versioni è identica, ad eccezione dei comandi per creare e gestire i thread in quella parallela. Come accennato nella Sezione 3, la soluzione sviluppata esegue l'algoritmo K-Means più volte. Per ciascuna esecuzione, vengono utilizzati diversi centroidi di partenza al fine di ottenere soluzioni potenzialmente differenti. La soluzione finale restituita è quella considerata migliore, valutata utilizzando l'SSE come metrica che ne definisce la bontà. La Figura 6 presenta la struttura della classe *Kmeans*.

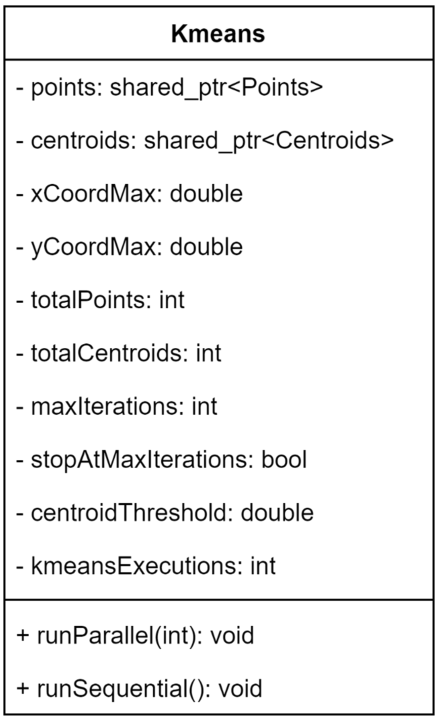


Figure 6: Classe Kmeans.

Solution

Vi è infine *Solution*, che fornisce una struttura per salvare le soluzioni intermedie dell'algoritmo, rappresentate dai campi *sse*, *centroids* e *clusterIds*. Di queste verrà successivamente ritornata la soluzione ottimale trovata, ovvero con valore di SSE minore. La Figura 7 presenta la struttura della classe *Solution*.

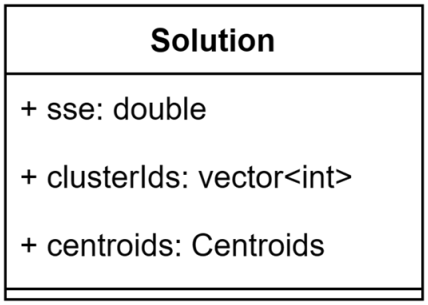


Figure 7: Classe Solution.

4.2 Codice Sorgente

Questa sezione presentata il codice sorgente delle parti principali dell'algoritmo. In particolare, vengono riportati e commentati i metodi *runParallel()* della classe *Kmeans* per l'esecuzione stessa dell'algoritmo, e *makeBlobs()* della classe *Utils*, che permette di generare cluster circolari di punti. L'intero codice sorgente sviluppato è pubblicamente disponibile al repository `https://github.com/Pikerozzo/openmp-kmeans`.

runParallel()

Il metodo *runParallel()* consente di avviare l'esecuzione della versione parallelizzata dell'algoritmo. Dopo aver definito il numero di thread da utilizzare, il metodo inizializza alcune variabili e oggetti di supporto. Tra questi, vi è *solutions*, un vettore di oggetti di tipo *Solution* (consultare la Sezione 4.1), con una dimensione pari al numero di esecuzioni di K-Means che verranno successivamente effettuate, ognuna delle quali genererà una soluzione ammissibile. Altri oggetti istanziati sono *centroids* e *points*, oggetti manipolati per trovare le soluzioni alle singole esecuzioni, *executionInitCentroids* utilizzato per la selezione dei centroidi iniziali, una serie di variabili e flag di convergenza e dei vettori usati per il calcolo delle posizioni dei centroidi. Il metodo definisce poi l'inizio della sezione parallela generando tanti thread quanti sono specificati con la clausola *num_threads(numThreads)*. Il vettore *solutions* è condiviso tra tutti i thread, così come le variabili che rappresentano i punti, i centroidi e le variabili e flag di convergenza. Vengono poi impostate le variabili e flag di convergenza con una sezione *#pragma omp single nowait* e selezionati i centroidi di partenza per l'attuale esecuzione dell'algoritmo (step parallelizzato con *#pragma omp for*). Successivamente inizia la ripetizione dei passaggi 2 e 3 dell'algoritmo, rispettivamente di assengamento dei punti ai cluster più vicini e aggiornamento delle coordinate dei centroidi (si veda la Sezione 2). Il primo dei due passaggi viene parallelizzato con la direttiva *#pragma omp for*. Una volta identificato il centroide più vicino al punto attualmente considerato, vengono aggiornati i vettori utilizzati successivamente per il calcolo della nuova posizione dei centroidi.

Il secondo passo viene anch'esso parallelizzato con la direttiva *#pragma omp for*, ma iterando in questo caso sui centroidi. Durante questa fase, vengono aggregati i risultati ottenuti dai vari thread sulle posizioni dei punti associati al centroide corrente, eseguendo una media delle posizioni dei punti per individuare le potenziali nuove coordinate dei centroidi. Tali coordinate vengono aggiornate solo se la nuova posizione determina uno spostamento considerevole, superiore a un valore minimo definito con *centroidThresholdSquared*. La clausola *reduction(+:totCentroidsMoved)* è utilizzata per ottenere il numero totale di centroidi che si sono mossi durante l'iterazione corrente. Successivamente, viene calcolato il SSE della soluzione attuale iterando sulla lista di punti tramite la direttiva *#pragma omp for reduction(+:currSse)*.

Le due operazioni principali (assegnamento dei punti ai cluster e aggiornamento della posizione

dei centroidi) vengono ripetute fino a quando non viene soddisfatta una condizione di arresto. Questa condizione è verificata da un singolo thread (utilizzando la direttiva *#pragma omp single*), il quale controlla se è stato raggiunto il numero massimo di iterazioni o se nessuno dei centroidi si è spostato tra un'iterazione e l'altra. Nel caso in cui queste condizioni vengano soddisfatte, la soluzione corrente viene salvata all'interno del vettore *solutions*. Una barriera finale (definita con *#pragma omp barrier*) permette di sincronizzare i thread prima del reset delle variabili e delle flag di convergenza, e prima dell'inizio dell'esecuzione successiva.

Fuori dalla sezione parallela, viene poi ricercata la migliore soluzione tra quelle presenti nel vettore *solutions* e restituita come soluzione finale dell'algoritmo.

```
1  void Kmeans::runParallel(int numThreads) {
2
3      // initialize current centroids and points
4    Centroids centroids{ totalCentroids };
5    Points points = *p;
6
7    // vector of kmeans solutions
8    vector<Solution> solutions(kmeansExecutions);
9
10   // variables for convergence check
11   bool notConverged = true;
12   int it = 0;
13   int totCentroidsMoved = 0;
14   double currSse = 0.0;
15   double centroidThresholdSquared = centroidThreshold * centroidThreshold;
16
17   // vectors to store centroid positions and counts for each thread
18   vector<double> centroidPositionsX(totalCentroids * numThreads);
19   vector<double> centroidPositionsY(totalCentroids * numThreads);
20   vector<int> centroidCounts(totalCentroids * numThreads);
21
22   // initialize centroids with random points
23   mt19937_64 gen(seed);
24   vector<int> indices{ points.ids };
25   vector<int> executionInitCentroids(kmeansExecutions * totalCentroids);
26   for (int run = 0; run < kmeansExecutions; ++run)
27   {
28     shuffle(indices.begin(), indices.end(), gen);
29     for (int i = 0; i < totalCentroids; i++) {
30       executionInitCentroids[run * totalCentroids + i] = indices[i];
31     }
32   }
33
34   // start of parallel section
35 #pragma omp parallel shared(centroids, points, solutions, executionInitCentroids,
       notConverged, it, currSse, totCentroidsMoved, centroidPositionsX,
       centroidPositionsY, centroidCounts) default(none) num_threads(numThreads)
36   {
37     int threadID = omp_get_thread_num();
38
39     double pointX, pointY;
40     int closestCentroidId;
41     double closestCentroidDist;
42     double deltaX, deltaY;
43     double distance;
44     bool assignToCentroid;
45
46     int index;
47     double x, y, count;
48     double newX, newY;
49
50     int centroidId;
51
52
53     // run kmeans for each set of initial centroids
54     for (int run = 0; run < kmeansExecutions; ++run)
55     {
56       // reset convergence variables for current run
```

```
#pragma omp single nowait
      {
        totCentroidsMoved = 0;
        currSse = 0;
        notConverged = true;
        it = 0;
      }

      // set initial centroids for current run
#pragma omp for
      for (int i = 0; i < totalCentroids; i++)
      {
        index = executionInitCentroids[run * totalCentroids + i];
        centroids.posX[i] = points.posX[index];
        centroids.posY[i] = points.posY[index];
        centroids.ids[i] = i;
      }

      // kmeans run start
      while (notConverged)
      {
        // assign points to closest centroid
#pragma omp for
        for (int i = 0; i < totalPoints; i++)
        {
          pointX = points.posX[i];
          pointY = points.posY[i];

          closestCentroidId = 0;
          closestCentroidDist = -1;

          // find closest centroid to point
          for (int j = 0; j < totalCentroids; j++)
          {
            deltaX = centroids.posX[j] - pointX;
            deltaY = centroids.posY[j] - pointY;
            distance = deltaX * deltaX + deltaY * deltaY;
            assignToCentroid = distance < closestCentroidDist || closestCentroidDist <
                0;

            // update closest centroid if distance is smaller
            if (assignToCentroid) {
              closestCentroidDist = distance;
              closestCentroidId = centroids.ids[j];

              points.centroidIds[i] = closestCentroidId;
            }
          }

          // sum the position of current point associated to its centroid, for each
              thread
          centroidPositionsX[closestCentroidId + totalCentroids * threadID] += pointX;
          centroidPositionsY[closestCentroidId + totalCentroids * threadID] += pointY;
          centroidCounts[closestCentroidId + totalCentroids * threadID]++;
        }

        // update centroid positions, reduction on total moved centroids
#pragma omp for reduction(+:totCentroidsMoved)
        for (int i = 0; i < totalCentroids; i++)
        {
          x = 0;
          y = 0;
          count = 0;
          // sum the coordinates of points associated to a centroid, for each thread
          for (int j = 0; j < numThreads; j++)
          {
            index = j * totalCentroids + i;
            x += centroidPositionsX[index];
            y += centroidPositionsY[index];
            count += centroidCounts[index];
          }
```

```
126
127           // update centroid position if there are points associated to it
128           if (count > 0) {
129             newX = x / count;
130             newY = y / count;
131
132             deltaX = newX - centroids.posX[i];
133             deltaY = newY - centroids.posY[i];
134             distance = deltaX * deltaX + deltaY * deltaY;
135
136             if (distance > centroidThresholdSquared) {
137               centroids.posX[i] = newX;
138               centroids.posY[i] = newY;
139
140               totCentroidsMoved++;
141             }
142           }
143         }
144
145         // find SSE of current solution
146 #pragma omp for reduction(+:currSse)
147         for (int i = 0; i < totalPoints; i++)
148         {
149           centroidId = points.centroidIds[i];
150           deltaX = centroids.posX[centroidId] - points.posX[i];
151           deltaY = centroids.posY[centroidId] - points.posY[i];
152           currSse += sqrt(deltaX * deltaX + deltaY * deltaY);
153         }
154
155         // check stop conditions: no centroids moved or max iterations reached
156 #pragma omp single
157         {
158           // if no centroids moved or max iterations reached, save solution
159           if (totCentroidsMoved == 0 || (stopAtMaxIterations && it == maxIterations))
                {
160             solutions[run] = Solution{ currSse, centroids, points.centroidIds };
161             notConverged = false;
162           }
163
164           // update iteration counter, reset solution results and helper variables
165           totCentroidsMoved = 0;
166           currSse = 0;
167           it++;
168
169           std::fill(centroidPositionsX.begin(), centroidPositionsX.end(), 0);
170           std::fill(centroidPositionsY.begin(), centroidPositionsY.end(), 0);
171           std::fill(centroidCounts.begin(), centroidCounts.end(), 0);
172         }
173       }
174       // kmeans run end
175
176 #pragma omp barrier
177     }
178   }
179   // end of parallel section
180
181   // find best solution
182   auto bestSolution = min_element(solutions.begin(), solutions.end(), [](const
          Solution& sol1, const Solution& sol2) {
183     return sol1.sse < sol2.sse;
184     });
185
186   *c = bestSolution->centroids;
187   p->centroidIds = bestSolution->clusterIds;
188 }
```

makeBlobs()

Viene riportata la funzione *makeBlobs()* della classe *Utils*. Questa imita il comportamento dell'omonima funzione presente nella libreria *scikit-learn* di Python, e consente di creare un dataset di punti, creando un numero definito di cluster più o meno circolari con posizioni casuali. Ogni cluster è composto da

una quantità variabile di punti, i quali presentano una certa variabilità rispetto al centro del cluster. La funzione restituisce, alla fine, le coordinate dei punti generati. La Figura 8 mostra un esempio di risultato della funzione.

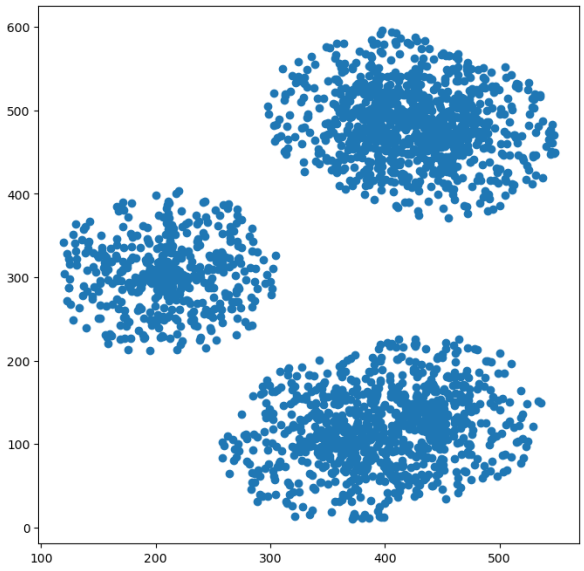


Figure 8: Esempio di risultato della funzione *makeBlobs()*.

```cpp
vector<vector<double>> Utils::makeBlobs(int totalPoints, int totalCenters, double
    clusterStd, double xCoordMax, double yCoordMax, int seed) {
  vector<vector<double>> points;
  vector<vector<double>> centers;

  // generate random cluster centers
  mt19937 gen(seed);
  uniform_real_distribution<> xDis(0, xCoordMax);
  uniform_real_distribution<> yDis(0, yCoordMax);
  for (int i = 0; i < totalCenters; i++) {
    double x = xDis(gen);
    double y = yDis(gen);
    centers.push_back({ x, y });
  }

  // generate points around cluster centers
  uniform_int_distribution<> centersDis(0, totalCenters - 1);
  uniform_real_distribution<> clusterStdDis(0, clusterStd);
  uniform_real_distribution<> angleDis(0, 2 * _Pi);
  for (int i = 0; i < totalPoints; i++) {
    int center = centersDis(gen);
    double radius = clusterStdDis(gen);
    double angle = angleDis(gen);

    double x = centers[center][0] + radius * cos(angle);
    double y = centers[center][1] + radius * sin(angle);
    points.push_back({ x, y });
  }

  return points;
}
```

5 Risultati

Questa sezione illustra e commenta i risultati ottenuti, concentrandosi in particolare sulle prestazioni e sui potenziali miglioramenti ottenuti passando da un'esecuzione sequenziale a una parallela. L'algoritmo produce come soluzione un insieme di coordinate che rappresentano la posizione finale dei centroidi, minimizzando la somma delle distanze tra ogni punto e il centroide a cui è assegnato.

5.1 Soluzione K-Means

Per agevolare la visualizzazione e la verifica della correttezza e dell'ammissibilità del risultato, il progetto implementato salva su file *.csv* ("comma-separated values") tutti i dati necessari per ricostruire la soluzione finale identificata. In particolare, i file prodotti sono *centroids.csv*, che contiene gli identificativi dei centroidi e le coordinate delle loro posizioni finali, e *points.csv*, che contiene le coordinate dei punti e l'*ID* del centroide a cui ciascun punto è associato. La correttezza dei risultati ottenuti è stata confrontata anche con la soluzione prodotta da implementazioni ampiamente riconosciute e utilizzate dell'algoritmo in questione, come quella presente nella libreria *scikit-learn*[4]. La Figura 9 mostra il confronto tra i risultati ottenuti utilizzando la funzione *KMeans* di *scikit-learn* (a sinistra) e quelli dell'implementazione sviluppata per il progetto in esame (a destra); si noti che i colori sono puramente indicativi e servono per rappresentare cluster diversi.

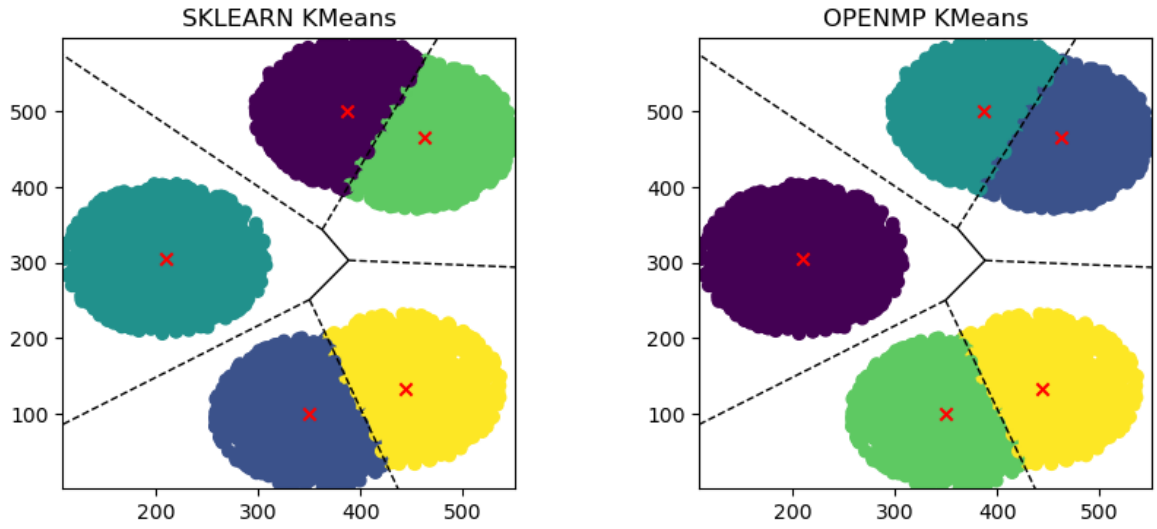


Figure 9: Confronto dei risultati (a sinistra *sklearn*, a destra versione parallelizzata con *OpenMP*).

5.2 Performance

Questa sottosezione presenta e analizza i risultati in termini di performance della soluzione sviluppata. Vengono mostrate e commentate diverse metriche di misurazione e confronto delle prestazioni tra un programma parallelo e uno sequenziale, tutte basate (più o meno direttamente) sul tempo di esecuzione del programma.

Le Figure 10 e 11 di seguito illustrano il tempo di esecuzione in relazione alla variazione del numero di punti (Figura 10) e del numero di centroidi (Figura 11), considerando diverse quantità di thread utilizzati. In entrambi i casi, è stato mantenuto un valore fisso per l'altro parametro, ovvero il numero totale di centroidi e di punti rispettivamente (nello specifico, 25 centroidi per il primo caso e 100′000 punti per il secondo). È evidente che l'aumento della quantità di dati comporta un incremento del numero di operazioni da eseguire, e quindi una maggiore complessità del problema: sia con un maggior numero di punti che di centroidi, ciò si traduce in un incremento generale del tempo totale di esecuzione. Nel caso di esecuzione sequenziale, il tempo di esecuzione cresce in modo quasi lineare all'aumentare dei punti o centroidi, portando rapidamente a una netta differenza rispetto ai tempi ottenuti con esecuzioni parallelizzate.

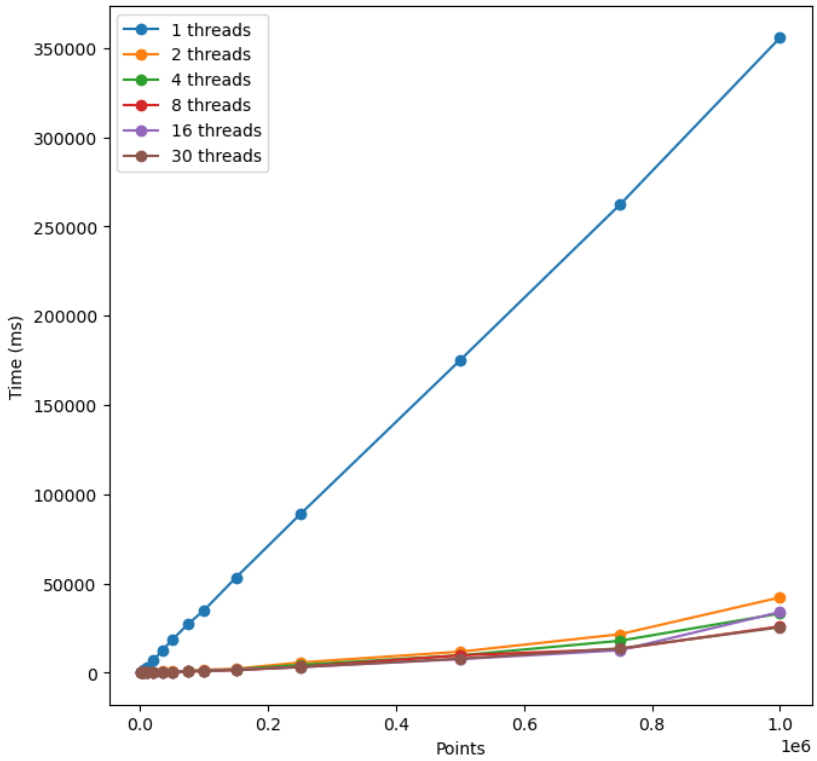


Figure 10: Tempo di esecuzione per numero di punti (distinzione per numero di thread usati).

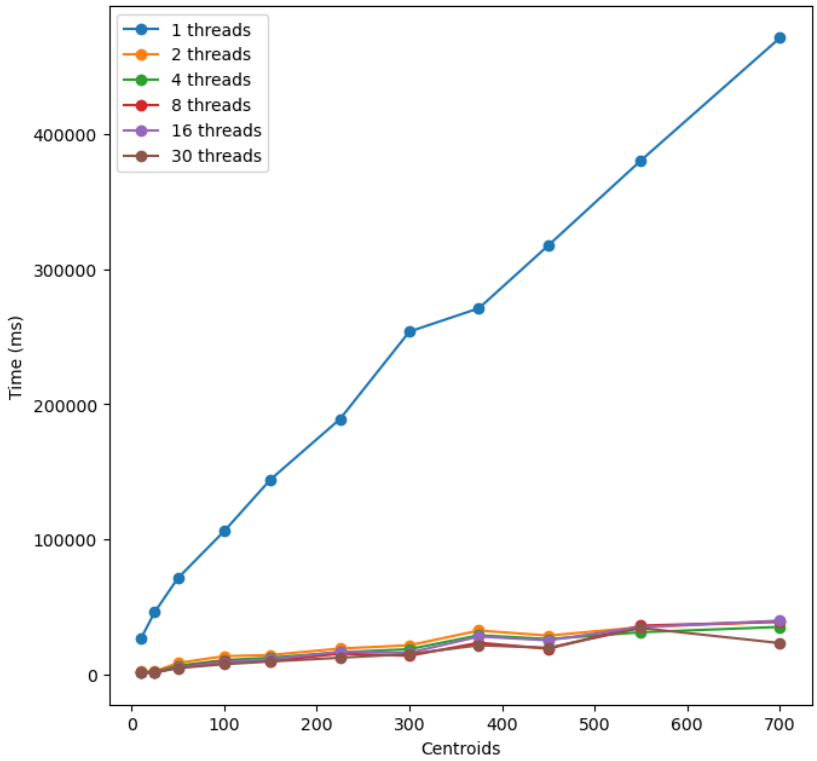


Figure 11: Tempo di esecuzione per numero di centroidi (distinzione per numero di thread usati).

Il grafico in Figura 12 illustra lo *speedup* medio, un parametro che misura le prestazioni relative

nell'elaborazione di un problema. Tecnicamente, lo speedup rappresenta il miglioramento della velocità di esecuzione di un compito su un sistema che utilizza una quantità variabile di risorse ad ogni esecuzione [5], che in questo caso sono il numero di thread. In pratica, lo speedup con P thread si calcola dividendo il tempo di esecuzione sequenziale per il tempo di esecuzione parallelo, quindi $S_P = \frac{t_s}{t_P}$, dove t_s rappresenta il tempo di esecuzione sequenziale e t_P rappresenta il tempo di esecuzione con P thread [6].

Si osservi che con 2, 4, 6 e 8 thread, lo speedup ottenuto (rispettivamente pari a 8.35, 9.95, 11.95 e 13.17) supera il numero di thread impiegati, risultando così in uno speedup, in questi casi, *super-lineare*. Nel complesso, lo speedup è *sub-lineare*, e a partire dai 10 thread la curva oscilla per poi stabilizzarsi poco sopra il valore 13. In una situazione ideale, lo speedup dovrebbe crescere linearmente all'aumentare del numero di thread impiegati.

Per comprendere meglio il valore dello speedup, è essenziale considerare la proporzione del programma parallelizzato. Secondo la legge di Amdahl, che riguarda il teorico valore massimo di speedup ottenibile, tale valore è limitato e dipende dalla parte sequenziale (non parallelizzata) del programma [6]. Nel caso del progetto sviluppato, la percentuale di codice parallelizzato si attesta intorno al $85 - 95\%$ del totale della parte considerata nelle misurazioni. Nel caso in cui la porzione parallelizzata di programma sia circa del 95%, lo speedup massimo teorico secondo la legge di Amdahl è di circa 20 [7].

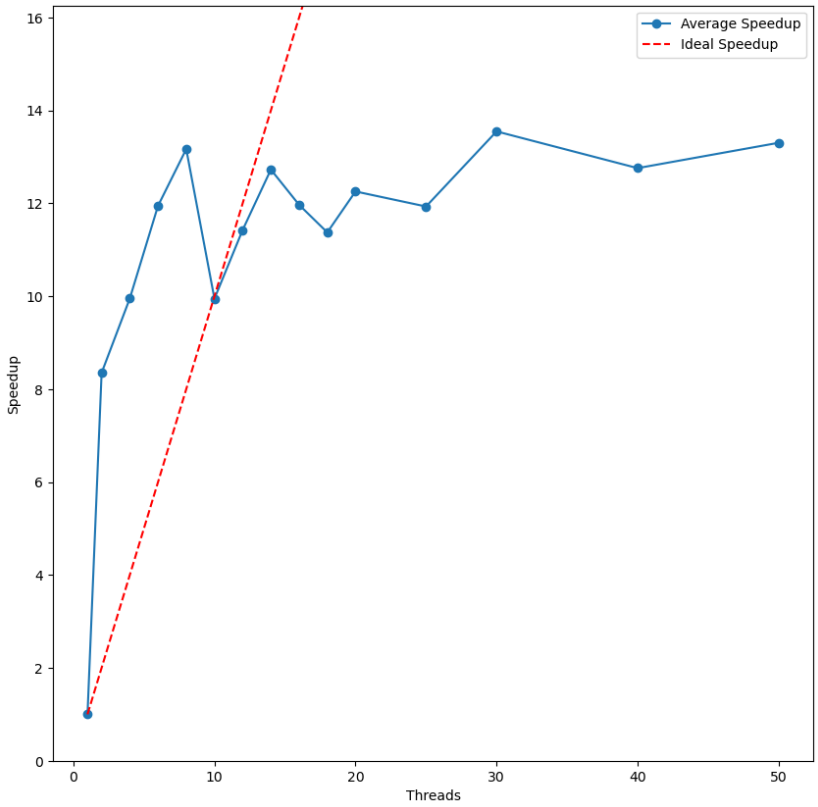


Figure 12: Speedup medio per il numero di thread contro lo speedup ideale.

La Figura 13 mostra invece la *efficiency* media, un parametro che stima quanto bene i processori siano impiegati nella risoluzione del problema affrontato, considerando il tempo trascorso in *idle*, per l'attesa di ricevere del lavoro da svolgere o per questioni di comunicazione e sincronizzazione. L'efficienza si calcola direttamente dallo speedup (mostrato sopra), dividendolo per il numero di thread impiegati, quindi: $E_P = \frac{S_P}{P}$ [6].

Poiché per P pari a 2, 4, 6 e 8 lo speedup è maggiore al numero di thread usati, l'efficienza risultante è superiore a 1; questa poi diminuisce in modo abbastanza regolare a partire dai $P = 10$, raggiungendo un limite di circa 0.15 con 50 thread.

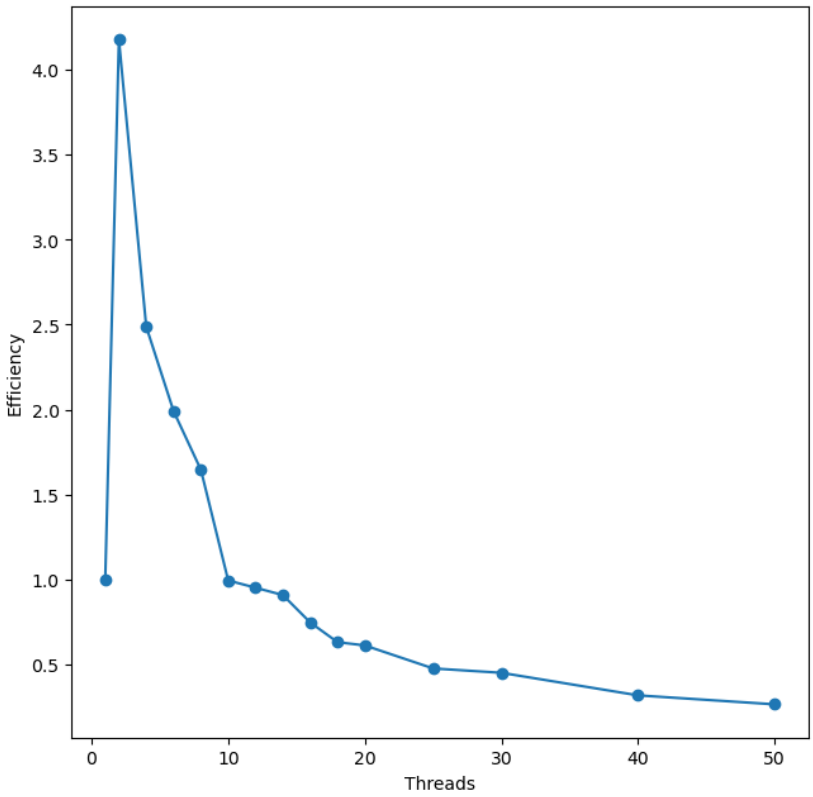


Figure 13: *Efficiency* per il numero di thread.

6 Conclusioni

Il progetto affrontato ha richiesto l'implementazione dell'algoritmo K-Means utilizzando l'API OpenMP per parallellizzarne l'esecuzione, consentendo così di sfruttare al massimo le risorse disponibili su architetture multicore. La soluzione risultante è in grado di applicare l'algoritmo di clustering su dati generati in modo flessibile, restituendo una lista di centroidi ottenuti attraverso la minimizzazione della distanza totale tra i punti e i cluster a cui sono assegnati. Le prestazioni complessive dell'applicativo mostrano uno speedup medio generalmente sub-lineare, indicando un buon grado di parallelizzazione del processo, specialmente usando un numero definito di thread (tra 2 e 14). Questi risultati costituiscono un punto di partenza solido per potenziali sviluppi futuri e ottimizzazioni. Ad esempio, si potrebbe esplorare la possibilità di ottimizzare ulteriormente i passaggi di assegnamento dei punti ai centroidi e il calcolo delle nuove posizioni dei cluster (step 2 e 3 della Sezione 2). Inoltre, potrebbe essere interessante considerare l'utilizzo di metodi avanzati per analizzare la distribuzione spaziale dei punti, al fine di rendere più efficiente la selezione iniziale dei centroidi. Queste prospettive offrono opportunità per migliorare ulteriormente le prestazioni dell'algoritmo K-Means in contesti sia sequenziali che paralleli.

References

[1] *K-means clustering - Wikipedia*. URL: https://en.wikipedia.org/wiki/K-means_clustering. (accessed: 10.01.2024).

[2] *OpenMP*. URL: https://www.openmp.org/. (accessed: 11.01.2024).

[3] G. Piqué. *Pikerozzo / openmp-kmeans*. URL: https://github.com/Pikerozzo/openmp-kmeans. (accessed: 13.01.2024).

[4] *sklearn.cluster.KMeans*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html. (accessed: 13.01.2024).

[5] *Speedup - Wikipedia*. URL: https://en.wikipedia.org/wiki/Speedup. (accessed: 13.01.2024).

[6] M. Bertini. *Introduction to Parallelism and Concurrency*. Dipartimento di Ingegneria dell'Informazione (DINFO). University of Florence, IT.

[7] *Amdahl's law - Wikipedia*. URL: https://en.wikipedia.org/wiki/Amdahl%27s_law. (accessed: 13.01.2024).