K-Means con OpenMP

Gregorio Piqué

January 18, 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 segue in parte questa modalità di esecuzione, ripetendo il processo di clustering K-Means per un numero predefinito (ma configurabile) di iterazioni. Ad ogni "macro-iterazione" 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.

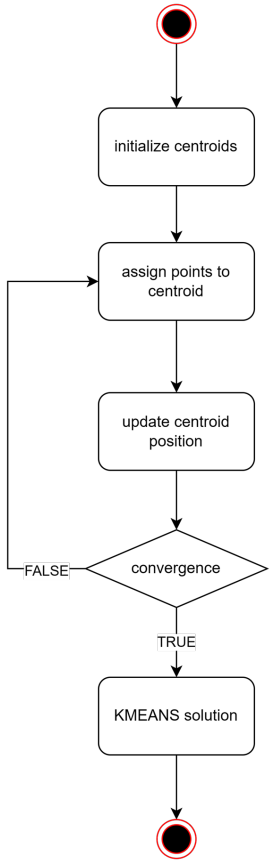
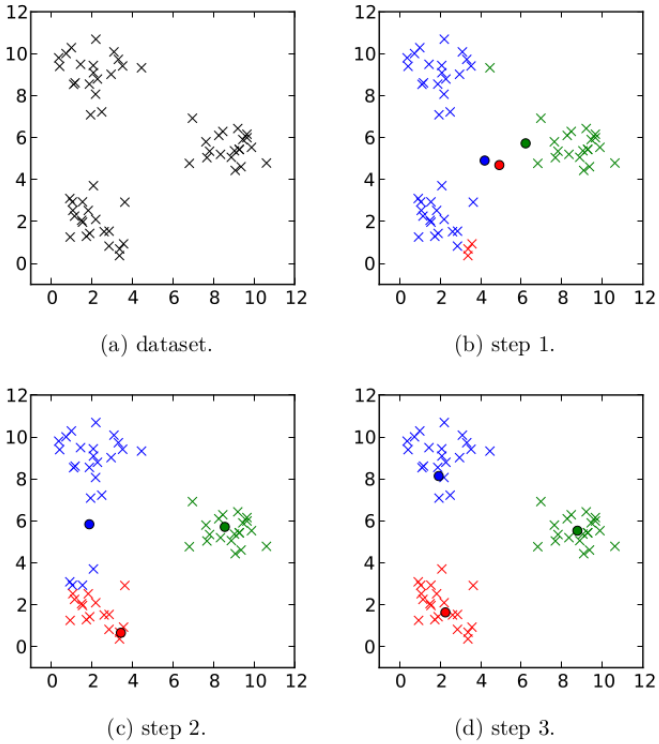


Figure 1: Flusso esecutivo di K-Means.



(a) dataset. (b) step 1.

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

Figure 2: Esempio di clustering con K-Means.

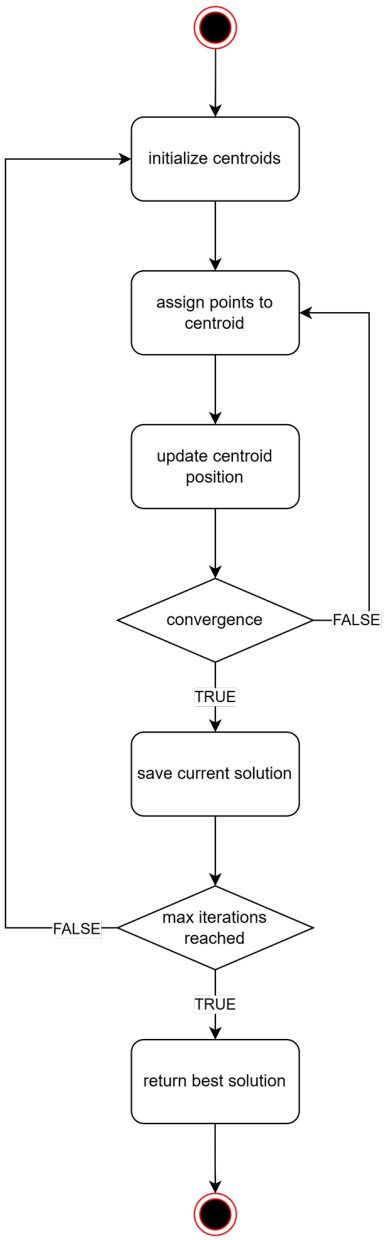


Figure 3: Flusso di K-Means con ripetizione.

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, 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 Figura 4 presenta la struttura della classe *Points*.

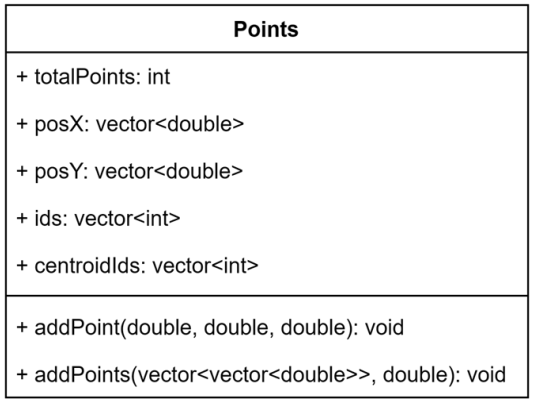


Figure 4: Classe Points.

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 Figura 5 presenta la struttura della classe *Centroids*.

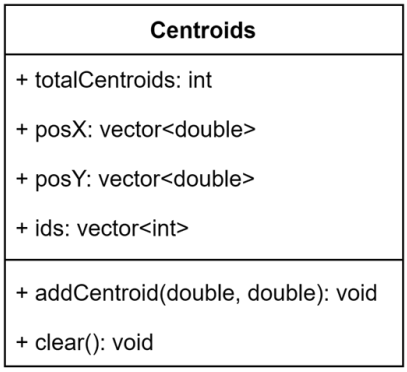


Figure 5: Classe Centroids.

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 *kmeansIterations*, che definisce il numero totale di "macro-iterazioni" da eseguire e quindi quante volte ripetere l'algoritmo riinizializzando i centroidi.

Oltre ai costruttori e distruttori, l'unico metodo pubblico è *run()*, utilizzato per avviare l'esecuzione generale dell'algoritmo. Gli altri metodi (privati) includono *runKmeans()*, che rappresenta una singola esecuzione di K-Means, e *calcSSE()*, il quale calcola la distanza totale tra i punti e i centroidi a cui sono assegnati. Quest'ultimo metodo definisce la bontà della soluzione trovata. La Figura 6 presenta la struttura della classe *Kmeans*.

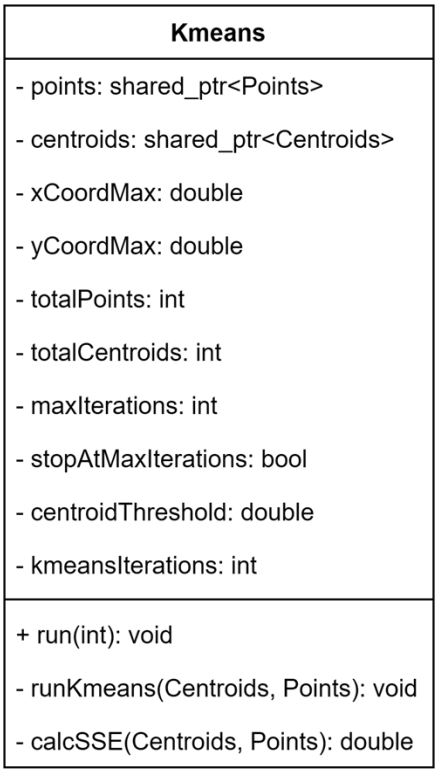


Figure 6: Classe Kmeans.

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 6 presenta la struttura della classe *Solution*.

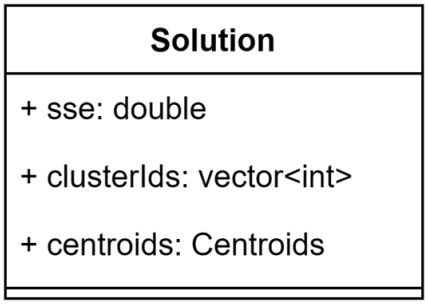


Figure 7: Classe Solution.

4.2 Codice Sorgente

Questa sezione presentata il codice sorgente delle parti principali dell'algoritmo. In particolare, vengono riportate e commentate le funzioni della classe *Kmeans*, quali *run()*, *runKmeans()* e *calcSSE()*. Questa sezione non comprende l'intero codice sviluppato; questo è però pubblicamente disponibile al repository https://github.com/Pikerozzo/openmp-kmeans.

run()

Il metodo *run()* consente di avviare l'esecuzione 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 thread che saranno successivamente creati: ogni thread accederà successivamente in modo autonomo e indipendente all'elemento situato alla posizione definita dal suo ID, ottenuto mediante *omp_get_thread_num()*. Altri oggetti istanziati sono *currCentroids* e *currPoints*, oggetti manipolati per trovare le soluzioni alle singole macro iterazioni, e *indices*, successivamente utilizzato per la selezione dei centroidi iniziali.

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, mentre per ciascuno di essi viene creata una copia privata di *currPoints*, *currCentroids*, e *indices*. Successivamente, inizia il ciclo *for* dedicato all'assegnazione del lavoro ai thread, ognuno dei quali esegue, con una iterazione del ciclo, l'algoritmo K-Means completo. All'inizio di ciascuna iterazione, i thread inizializzano i centroidi scegliendo in modo randomico dei punti del dataset. Una volta individuata la soluzione corrente, viene calcolato il relativo SSE; se quest'ultimo è minore del SSE corrispondente alla migliore soluzione trovata dal thread attuale, questa viene aggiornata con quella nuova. Fuori dalla sezione parallela viene successivamente ricercata la migliore soluzione tra quelle presenti nel vettore *solutions*.

```
1  void Kmeans::run(int numThreads) void Kmeans::run(int numThreads) {
2
3    // vector of solutions, one for each thread
4    std::vector<Solution> solutions(numThreads);
5
6    // initialize current centroids and points
7    Centroids currCentroids{};
8    Points currPoints{ *p };
9
10   // vector of point ids
11   std::vector<int> indices(totalPoints);
12   std::iota(indices.begin(), indices.end(), 0);
13
14   // start of parallel section
15 #pragma omp parallel firstprivate(currPoints, currCentroids, indices) shared(solutions
       ) default(none) num_threads(numThreads)
16   {
17     // create thread-custom random number generator
18     int threadId = omp_get_thread_num();
```

```
    std::mt19937_64 gen(threadId + seed);

    // initialize thread best solution
    solutions[threadId] = Solution{};

    // parallel for loop, each thread runs a "macro k-means iteration"
#pragma omp for
    for (int run = 0; run < kmeansIterations; ++run) {

      // initialize iteration centroids with random points
      currCentroids.clear();
      std::shuffle(indices.begin(), indices.end(), gen);
      for (int i = 0; i < totalCentroids; i++) {
        currCentroids.addCentroid(currPoints.posX[indices[i]], currPoints.posY[indices
            [i]]);
      }

      // run kmeans on the current initial centroids
      runKmeans(currCentroids, currPoints);

      // get SSE of current solution
      double currentSSE = calcSSE(currCentroids, currPoints);

      // update best solution found by the current thread
      Solution threadBestSolution = solutions[threadId];
      if (currentSSE < threadBestSolution.SSE) {
        solutions[threadId].SSE = currentSSE;
        solutions[threadId].centroids = currCentroids;
        solutions[threadId].centroidIds = currPoints.centroidIds;
      }
    }
  }
  // end of parallel section


  // find best solution
  auto bestSolution = std::min_element(solutions.begin(), solutions.end(), [](const
      Solution& sol1, const Solution& sol2) {
    return sol1.SSE < sol2.SSE;
    });

  *c = bestSolution->centroids;
  p->centroidIds = bestSolution->centroidIds;
}
```

runKmeans()

Il metodo *runKmeans()* esegue l'algoritmo K-Means trovando una possibile soluzione, partendo dai centroidi inizializzati nella funzione precedente *run()*. Inizializzate alcune variabili per i controlli sulla convergenza e raggiungimento del massimo numero di iterazioni, iniziano i passaggi 2 e 3 dell'algoritmo (si veda la Sezione 2), rispettivamente di assengamento dei punti ai cluster più vicini e successivo aggiornamento delle coordinate dei centroidi. Le due operazioni principali vengono ripetute fino a quando viene raggiunto il numero massimo di iterazioni o se, tra un'iterazione e l'altra, nessuno dei centroidi si è spostato al di fuori di un margine definito da *centroidThreshold*.

```
void Kmeans::runKmeans(Centroids& centroids, Points& points) const {

  // variables for convergence check
  int it = 0;
  bool centroidsMoved = false;
  double centroidThresholdSquared = centroidThreshold * centroidThreshold;
  bool notConverged = true;

  // run kmeans until convergence or max iterations reached
  while (true) {
    centroidsMoved = false;

    // assign points to closest centroid
    for (int i = 0; i < points.totalPoints; i++) {
```

```
      double p_x = points.posX[i];
      double p_y = points.posY[i];

      int closest_c_id = points.centroidIds[i];
      double closest_c_dist = -1;
      if (closest_c_id != -1)
      {
        closest_c_dist = sqrt(pow(centroids.posX[closest_c_id] - p_x, 2) + pow(
            centroids.posY[closest_c_id] - p_y, 2));
      }

      // find closest centroid to point
      double distance = 0;
      for (int j = 0; j < centroids.totalCentroids; j++) {

        distance = sqrt(pow(centroids.posX[j] - p_x, 2) + pow(centroids.posY[j] - p_y,
             2));
        if (distance < closest_c_dist || closest_c_id == -1) {
          closest_c_dist = distance;
          closest_c_id = centroids.ids[j];

          points.centroidIds[i] = closest_c_id;
        }
      }
    }

    // update centroid positions
    for (int i = 0; i < centroids.totalCentroids; i++) {
      double tempX = 0;
      double tempY = 0;

      int currCentroidId = centroids.ids[i];
      double currCentroidPoints = 0;
      double newX = 0;
      double newY = 0;
      for (int j = 0; j < points.totalPoints; ++j) {

        // skip points not assigned to current centroid
        if (currCentroidId != points.centroidIds[j])
          continue;

        tempX += points.posX[j];
        tempY += points.posY[j];
        currCentroidPoints++;
      }

      if (currCentroidPoints > 0) {
        newX = tempX / currCentroidPoints;
        newY = tempY / currCentroidPoints;

        // update coordinates if centroid moved more than threshold
        if (pow(newX - centroids.posX[i], 2) > centroidThresholdSquared || pow(newY -
            centroids.posY[i], 2) > centroidThresholdSquared) {
          centroids.posX[i] = newX;
          centroids.posY[i] = newY;

          centroidsMoved = true;
        }
      }
    }

    // check for convergence: no centroids moved or max iterations reached
    if (!centroidsMoved || (stopAtMaxIterations && it == maxIterations)) {
      break;
    }

    // increment iteration counter
    it++;
  }
}
```

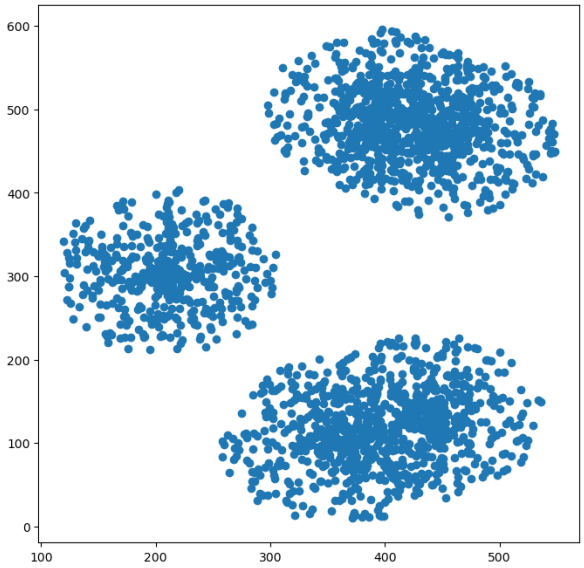


Figure 8: Esempio di risultato della funzione *makeBlobs()*.

calcSSE()

L'ultimo metodo della classe *Kmeans* è *calcSSE()*, che permette di calcolare la somma delle distanze euclidee tra i centroidi e i punti a loro assegnati.

```
double Kmeans::calcSSE(Centroids& centroids, Points& points) const {
  double sse = 0;
  for (int i = 0; i < points.totalPoints; i++) {
    sse += sqrt(pow(centroids.posX[points.centroidIds[i]] - points.posX[i], 2) +
      pow(centroids.posY[points.centroidIds[i]] - points.posY[i], 2));
  }
  return sse;
}
```

makeBlobs()

Tra le restanti funzioni della soluzione implementata, 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.

```
vector<vector<double>> Utils::makeBlobs(int totalPoints, int totalCenters, double
    clusterStd, int xCoordMax, int 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.

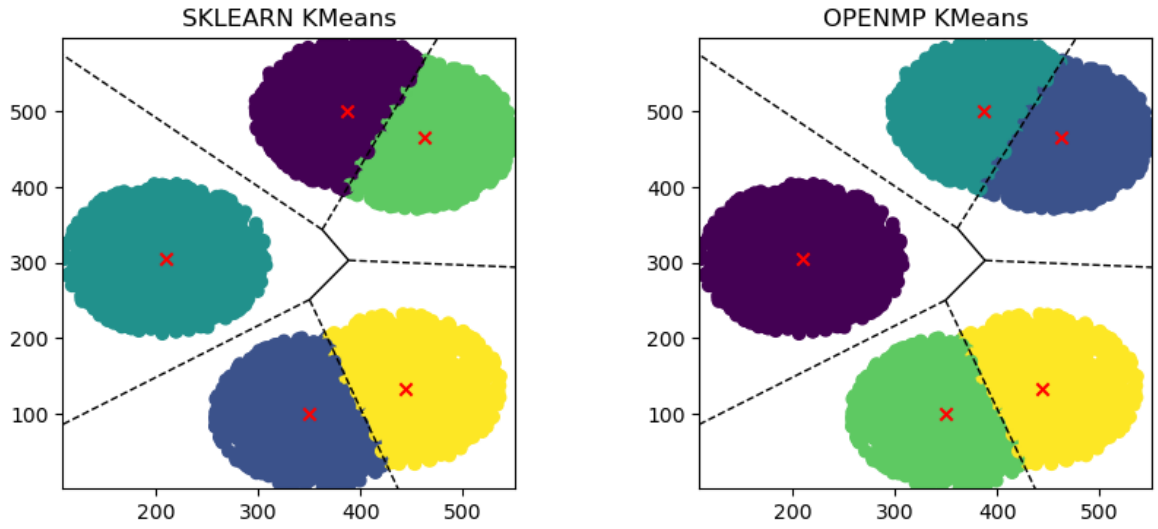


Figure 9: Confronto dei risultati (a sinistra *sklearn*, a destra versione parallelizzata con *OpenMP*).

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, per ciascun thread, il tempo di esecuzione in relazione alla variazione del numero di punti (Figura 10) e del numero di centroidi (Figura 11). Emerge chiaramente che l'aumento della quantità di dati e quindi di operazioni da svolgere (sia con il maggior numero di punti che di centroidi), comporti un incremento generale del tempo totale di esecuzione.

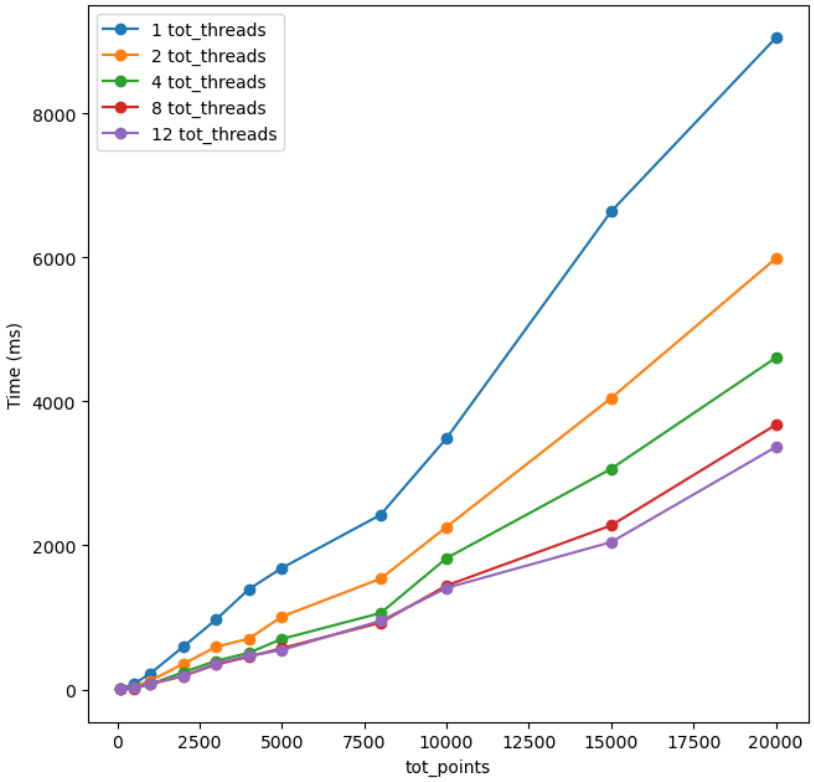


Figure 10: Tempo di esecuzione per numero di punti (distinzione per numero di thread usati).

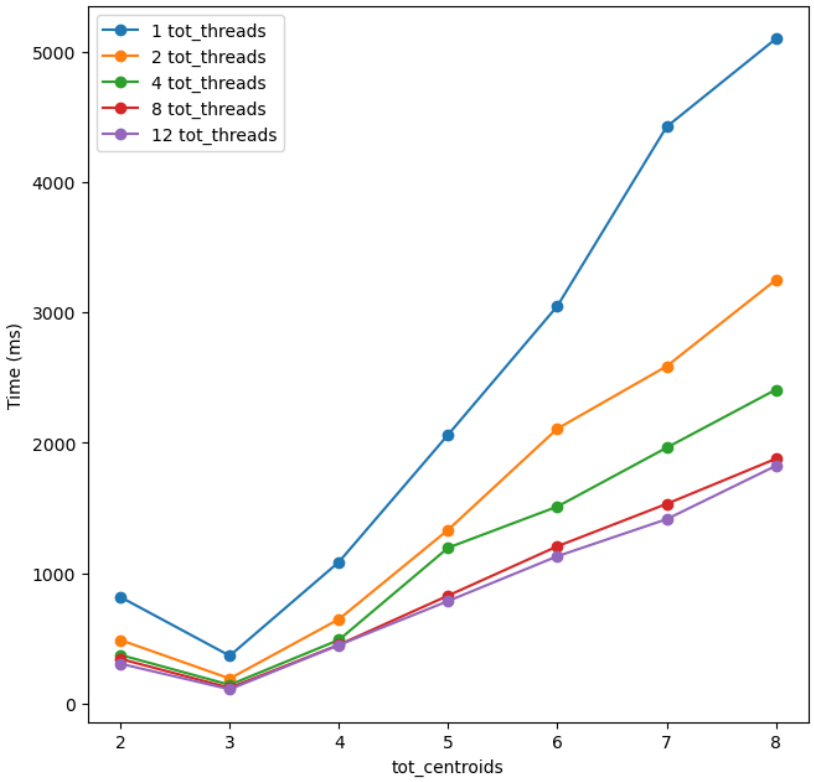


Figure 11: Tempo di esecuzione per numero di centroidi (distinzione per numero di thread usati).

È inoltre possibile notare un risultato apparentemente controintuitivo nel grafico della Figura 11, dove il passaggio da due centroidi a tre centroidi si traduce in un tempo di esecuzione minore. Questo fenomeno potrebbe essere spiegato dal grafico della Figura 12, che mostra il numero medio di iterazioni necessarie per ottenere una soluzione K-Means (di una delle "macro-iterazioni") in funzione del numero di centroidi: questo evidenzia come il numero di iterazioni diminuisca notevolmente passando da 2 a 3 centroidi, per poi riprendere ad aumentare a partire da 4 cluster. Il tempo di esecuzione maggiore è quindi probabilmente dato dal più alto numero di iterazioni necessarie al completamento dell'algoritmo.

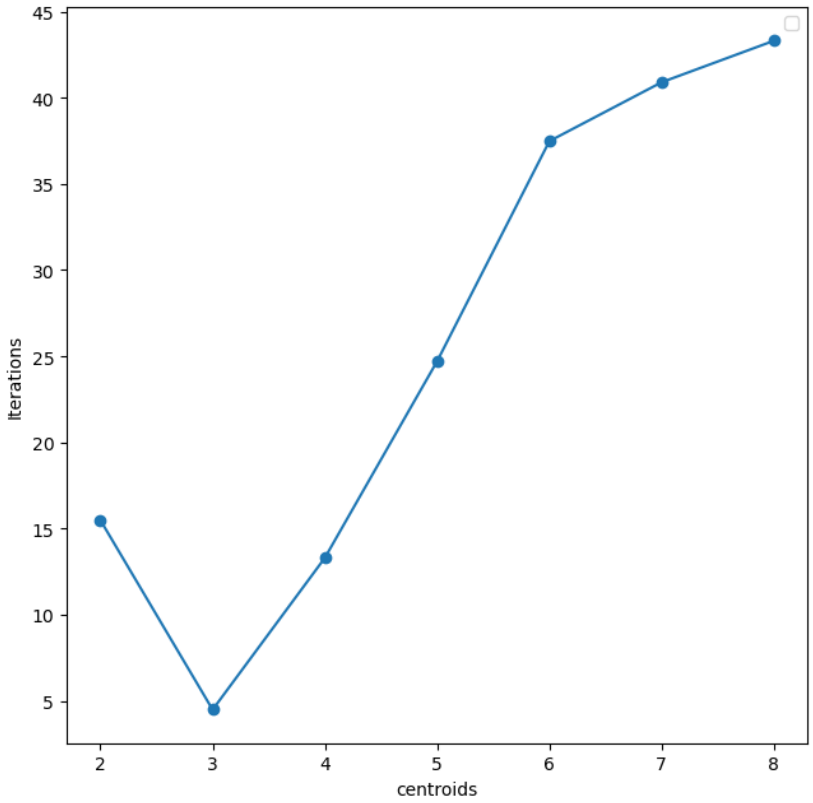


Figure 12: Numero medio di iterazioni per numero di centroidi.

Il grafico in Figura 13 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 thread, lo speedup ottenuto (per la precisione pari a 2.285) supera il numero di thread impiegati, risultando così in uno speedup, in questo caso, *super-lineare*. Nel complesso, lo speedup è *sub-lineare*, e smette di crescere a partire dall'uso di 6 thread, oltre i quali si osserva un appiattimento della curva poco sopra il valore 4.

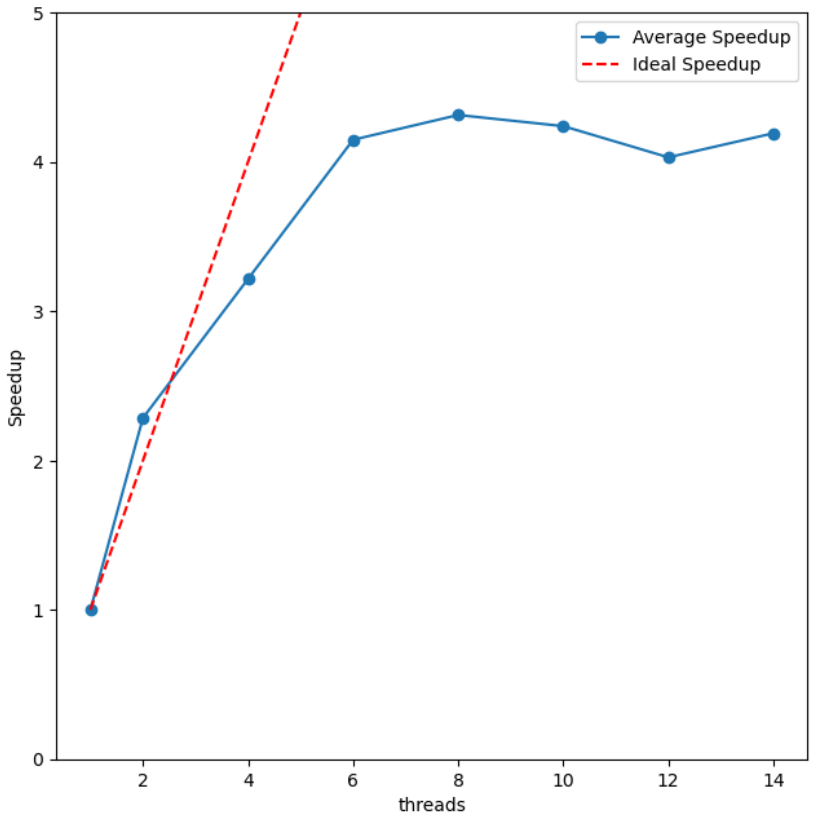


Figure 13: Speedup medio per il numero di thread contro lo speedup ideale.

La Figura 14 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 $P = 2$ lo speedup è maggiore del numero di thread usati, l'efficienza risultante è superiore a 1; successivamente diminuisce, raggiungendo un limite di circa 0.258 con 12 thread.

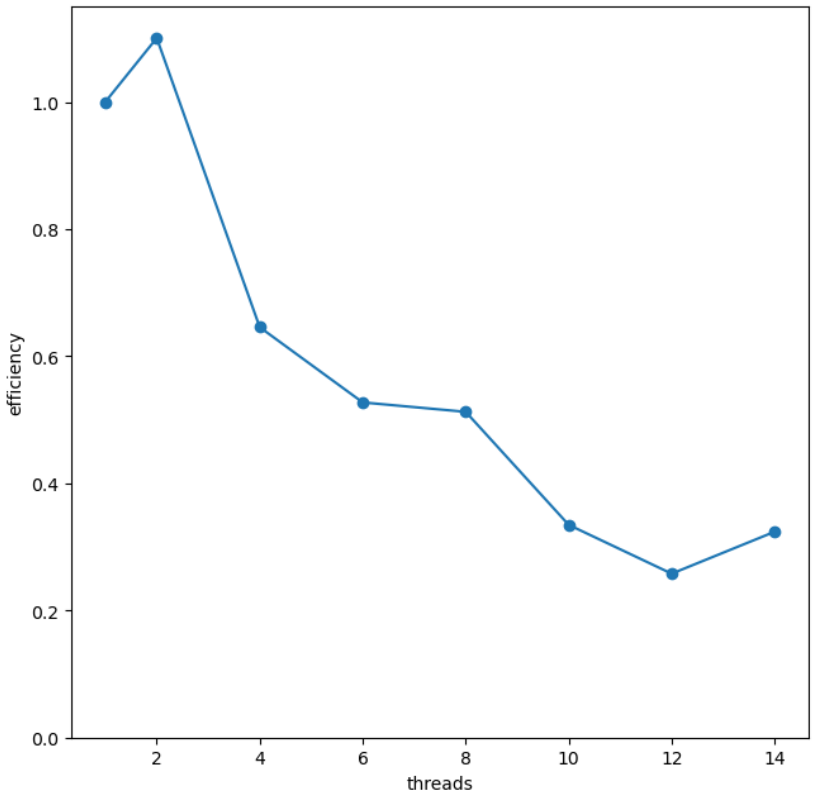


Figure 14: *Efficiency* per il numero di thread.

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. 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.