Algoritmo K-Means: versione sequenziale e versione parallela

Alessandro Sestini
Matricola - 6226094
alessandro.sestini@stud.unifi.it

Abstract

Questo elaborato mid-term si concentra sullo studio e sull'implementazione dell'algoritmo k-means sia in versione sequenziale che in versione parallela. L'implementazione é stata fatta in C++ e OpenMP per la parte parallela. Al termine sono state eseguite delle analisi e dei confronti delle performance delle due versioni su una macchina con un processore Intel i5 a 2 core. Per lo studio e sviluppo del programma sono state inoltre adottate delle tecniche di profilazione e thread sanitizing, e per valutare le performance é stato misurato lo speedup ottenuto passando dalla versione sequenziale a quella parallela: i risultati dicono che utilizzando la versione parallela otteniamo uno speedup medio di 2, dunque uno speedup lineare, che é quello che ci aspetteremmo con una macchina a 2 core.

1. Introduzione

1.1. L'algoritmo

Il K-Means é un algoritmo di clustering che ha il compito di suddividere i punti dello spazio in K gruppi sulla base delle caratteristiche dei punti stessi. L'obbiettivo finale é avere i punti che appartengono alla stessa classe sullo stesso cluster (devono quindi essere simili e avere una distanza piccola tra di loro), mentre quelli che stanno su cluster diversi devono essere di classi diverse (dunque non devono essere simili e avere una distanza relativamente grande). Per semplicitá, in questo elaborato vengono generati dei punti casuali nello spazio 2D, ma é possibile generalizzare usando piú dimensioni ma adoperando sempre lo stesso ragionamento. L'algoritmo é molto semplice ed é composto sostanzialmente da pochi passi:

1. Si generano N punti casuali nello spazio;
2. Si generano K centroidi che rappresentano i K cluster. Nel passo di inizializzazione, il cluster avrá al suo interno solamente il centroide;
3. Per ogni punto, si calcola la distanza con tutti i K cluster e si assegna il punto al cluster piú vicino;
4. Si aggiornano le caratteristiche del centroide, in particolare la sua posizione all'interno del cluster;
5. Si ripete dal punto 3 fino a che i centroidi non si muovono piú o fino a che non si raggiunge un massimo numero di iterazioni.

In Fig. 1 possibile vedere il funzionamento dell'algoritmo.

1.2. Parametri

L'algoritmo ha bisogno di specificare alcuni parametri. Il numero K dei cluster deve essere noto e scelto a priori; alternativamente é possibile adottare delle soluzioni piú complesse per scegliere il K migliore dato i punti, ad esempio ripetere l'algoritmo per diversi K e scegliere quello che minimizza la distanza tra i centroidi. Non é peró compito di questo elaborato valutare la situazione migliore, dunque verrá preso un numero di cluster fisso. Deve essere scelto come vengono definiti i centroidi e come vengono generati nel passo di inizializzazione: nel nostro caso in cui i punti stanno in uno spazio euclideo a 2 dimensioni, il centroide puó essere a sua volta un punto dello spazio definito dalla media dei punti appartenenti a quel cluster; se si usasse uno spazio non-euclideo questo non sarebbe possibile. Inoltre, é necessario scegliere i centroidi iniziali: esistono varie tecniche, quella utilizzata in questo programma é quella di prendere K punti casuali dello spazio. Un altro dettaglio importante é la scelta della distanza: sono possibili diverse soluzioni, in questo progetto viene usata una distanza euclidea. L'ultimo parametro é la scelta del numero di iterazioni massimo, che potrebbe variare dalla natura e dal numero dei punti e dei cluster.

1.3. Considerazioni

É un algoritmo che converge molto in fretta, sebbene non garantisca di trovare l'ottimo globale. La qualitá del risultato finale dipende soprattutto dal numero e dalla posizione iniziale dei cluster. Inoltre, avendo inizializzato i cluster

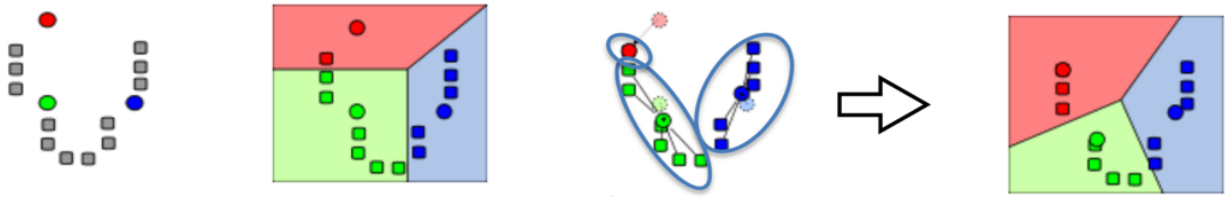

Fig. 1: Esempio di K-Means

in maniera randomica, non é detto che ripetendo 2 volte l'algoritmo sugli stessi punti, il risultato sia lo stesso.

2. Implementazione

Passiamo ora all'implementazione dell'algoritmo: analizzeremo prima la versione sequenziale e poi passeremo alla versione parallela. Come linguaggio di programmazione é stato usato C++ e poi esteso con OpenMP.

2.1. Classi

Sono state create 2 classi principali:

1. **Point**: gli oggetti di tipo Point sono composti da 3 attrbuti: double *x_coord*, il valore della coordinata x del punto nello spazio; double *y_coord*, il valore della coordinata y; int *cluster_id*, l'id del cluster a cui quel punto appartiene. Nel passo di inizializzazione, tutti i punti saranno assegnati al cluster 0. Le funzioni presenti in questa classe sono semplicemente i metodi *getter()* e *setter()* degli attributi.

2. **Cluster**: gli oggetti di tipo Cluster sono composti dagli stessi parametri *x_coord* e *y_coord*, piú un intero *size* che indica il numero di punti appartenenti a quel cluster, necessario ad aggiornare le caratteristiche di quest'ultimo. I metodi in questa classe sono al solito le funzioni *getter()* e *setter()* degli attributi, piú dei metodi *add_point()*, *free_point()* e *update_coords()* che servono per il passo 4 ad aggiornare le coordinate del centroide e verificare se quel particolare punto si é mosso o meno rispetto alla posizione precedente.

2.2. main_sequential.cpp

Passiamo ora all'implementazione dell'algoritmo sequenziale, tralasciando le parti banali di inizializzazione (come detto precedentemente, i punti e i cluster vengono scelti in maniera randomica) e soffermandoci soprattutto sulle 2 funzioni fondamentali dell'algoritmo: *compute_distance()* e *update_clusters()*. I punti e i cluster vengono salvati all'interno di un oggetto vector della libreria stl per poter trattare vettori di punti e di cluster.

2.3. compute_distance

```
void compute_distance(vector<Point> &points,
                      vector<Cluster> &clusters){

unsigned long points_size = points.size();
unsigned long clusters_size = clusters.size();
double min_distance;
int min_index;

for(int i = 0; i < points_size; i++){
  Point &point = points[i];
  min_distance = euclidean_dist(point, clusters[0]);
  min_index = 0;

    for(int j = 0; j < clusters_size; j++){
     Cluster &cluster = clusters[j];
     double distance = euclidean_dist(point, cluster);

     if(distance < min_distance){
       min_distance = distance;
       min_index = j;
     }
   }
   points[i].set_cluster_id(min_index);

   //In quest'ultima operazione si aggiorna
   // il size del cluster e si accumulano
   // le coordinate dei punti che sono finiti all'interno
   clusters[i].add_point(points[i]);
 }
}
```

Come é possibile vedere, in questa funzione é racchiusa la maggior parte della complessitá dell'algoritmo, in cui per ogni punto si calcola la distanza euclidea con ogni cluster e alla fine si assegna al punto il cluster che é risultato piú vicino. Ovviamente, nel nostro caso, si parla di moltissimi punti (sulle centinaia di migliaia) e con relativamente tanti cluster (20-30). Per ogni punto, si aggiornano delle variabili all'interno di ogni oggetto di tipo Cluster che serviranno poi per aggiornare le coordinate dei centroidi.

2.4. update_cluster

```
bool update_clusters(vector<Cluster> &clusters){
  bool conv = false;
  for(int i = 0; i < clusters.size(); i++){
    //Si dividono le coordinate accumulate
    // per la variabile size del cluster
    conv = clusters[i].update_coords();
    clusters[i].free_point();
  }
  return conv;
}
```

Si aggiornano le coordinate dei centroidi a seconda dei punti che sono finiti dentro al relativo cluster in questa iterazione. Se anche un solo cluster si sará spostato dalla sua posizione precedente, *conv* sará uguale a true, altrimenti sará uguale a false e usciremo dal ciclo principale del main, terminando l'algoritmo.

3. Versione parallela

La versione sequenziale del problema é molto parallelizzabile nella prima funzione *compute_distance()*. Questo perché é sia il centro dell'algoritmo, sia perché le operazioni che vengono fatte per ogni punto sono indipendenti le une dalle altre. É quindi possibile prendere ogni punto, calcolare le distanze tra tutti i cluster e assegnargli quello piú vicino in modo parallelo; in pratica, basta parallelizzare il ciclo for piú esterno. La Tab. 1 mostra il tempo medio trascorso all'interno di ogni funzione per 500000 punti e 20 cluster, a dimostrazione del fatto che il ciclo in cui la CPU passa piú tempo é sostanzialmente il primo:

Funzione	Tempo
compute_distance()	1.075 s
update_cluster()	0.000001 s

Table 1: Profilazione dell'algoritmo sequenziale.

Il problema é dunque racchiuso in un ciclo for, e quindi la soluzione piú intuitiva é usare OpenMP che, per le sue caratteristiche, é ottimo per parallelizzare questo tipo di ciclo.

3.1. compute_distance parallelo

É stato applicato un *parallel for* al ciclo piú esterno, mettendo come variabili private per ogni thread *min_distance* e *min_index*, che verranno inizializzate per ogni punto. Inoltre, sono state definite firstprivate le variabili *points_size* e *clusters_size* di modo che ogni thread abbia la sua copia privata delle variabili inizializzate al valore dato dal main thread prima di entrare nella regione parallela. Rimagono peró shared l'array dei punti e dei cluster: alla fine di ogni iterazione, si assegna al punto relativo il cluster piú vicino e si modificano le variabili del centroide per poter effettuare l'aggiornamento nell'operazione successiva. Dato che per quest'ultimo passaggio piú thread possono accedere allo stesso cluster, é necessario che questa sezione sia definita *critical*, mentre le operazioni fatte sui punti sono totalmente indipendenti le une dalle altre e dunque non é necessario nessun meccanismo di sincronizzazione.

```
void compute_distance(vector<Point> &points,
                      vector<Cluster> &clusters){
  unsigned long points_size = points.size();
  unsigned long clusters_size = clusters.size();
  double min_distance;
  int min_index;

#pragma omp parallel default(shared) \
private(min_distance, min_index) \
firstprivate(points_size, clusters_size, clusters)
  {
#pragma omp for schedule(static, 1000)

    for (int i = 0; i < points_size; i++) {
      Point &point = points[i];
      min_distance = euclidean_dist(point, clusters[0]);
      min_index = 0;

      for (int j = 1; j < clusters_size; j++) {
        Cluster &cluster = clusters[j];
        double distance = euclidean_dist(point, cluster);

        if (distance < min_distance) {
          min_distance = distance;
          min_index = j;
        }
      }
      points[i].set_cluster_id(min_index);
#pragma omp critical
      clusters[min_index].add_point(points[i]);
    }
  }
}
```

3.2. Altre considerazioni

La versione parallela differisce da quella sequenziale anche per altri passaggi:

1. In fase di inizializzazione, sono state usate due parallel section per eseguire in parallelo l'inizializzazione dei cluster e dei punti, solo per far notare che sono sostanzialmente delle operazioni molto semplici e il guadagno di effettuarle in parallelo é praticamente nullo;

2. Dalla profilazione effettuata sul programma e dai tempi visti nella tabella precedente, é possibile notare che l'unico ciclo utile da parallelizzare sia quello definito in *compute_distance()*; l'altra funzione *update_cluster()* é lasciata invariata in quanto é sostanzialmente un ciclo su decine di elementi: anche se é effettivamente parallelizzabile dato che le oper-

Points	Clusters	Total sequantial time	Iteration sequential time	Total parallel time	Iteration parallel time
100000	10	3.57 s	0.18 s	2.74 s	0.13 s
100000	20	5.68 s	0.28 s	4.04 s	0.20 s
500000	10	12.75 s	0.64 s	6.40 s	0.37 s
500000	20	22.40 s	1.12 s	12.13 s	0.61 s
1000000	10	22.74 s	1.13 s	12.13 s	0.61 s
1000000	20	40.66 s	2.03 s	21.40 s	1.20 s

Table 2: Tabella degli esperimenti.

azioni fatte per ogni cluster sono indipendenti, essendo un ciclo su pochi oggetti é inutile renderlo parallelo in quanto si perderebbe piú tempo per la creazione e gestione dei thread piuttosto che per fare le operazioni effettivamente utili.

4. Esperimenti

Gli esperimenti sono stati effettuati comparando i tempi di completamento delle due versioni variando il numero di cluster e il numero dei punti, con un numero massimo di iterazioni fisso a 20; nella Tab. 2 sono riassunti i risultati. Ricordiamo che la macchina con la quale si é testato il programma é una macchina a 2 core. Dalla tabella si nota che lo speedup incrementa all'aumentare sia del numero dei punti, ma soprattutto all'aumentare del numero di cluster, in quanto si aumentano le operazioni che ogni thread deve eseguire. Lo speedup raggiunto é comunque lineare in quanto abbiamo un tempo di completamento dimezzato rispetto alla versione sequenziale, mentre per pochi punti non é visibile in quanto dobbiamo considerare il tempo di creazione dei thread CPU. In Fig. 2 é possibile vedere un risultato dell'algoritmo plottato tramite gnuplot.

5. Note

Per visualizzare il risultato dell'algoritmo é necessario avere installato la libreria gnuplot; il programma richiamerá poi tramite una chiamata a system il plot dei punti che vengono prima salvati in un file data.txt che una volta visualizzato verrá eliminato.

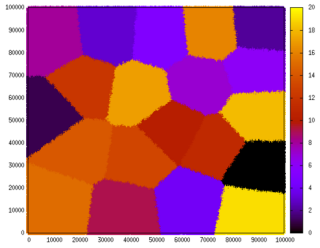

Fig. 2: Risultato dell'algoritmo visualizzato tramite gnuplot.