Parallel KMeans
Parallel Programming for Machine Learning

Niccolò Arati
niccolo.arati@edu.unifi.it

Abstract

*Questo progetto ha come scopo quello di ad analizzare i vantaggi di un'implementazione parallela dell'algoritmo di clustering KMeans. In particolare si vuole andare ad osservare lo **speedup**, calcolato come rapporto tra il tempo di esecuzione sequenziale e tempo di esecuzione parallelo, al variare di diverse configurazioni dei parametri principali dell'algoritmo. Oltre ai parametri dell'algoritmo, verranno analizzati i risultati anche al variare del numero di threads relativi alla configurazione parallelizzata.*

1. Introduzione

KMeans è un algoritmo di clustering basato su apprendimento non supervisionato. Dato un dataset di punti e un numero K di cluster, KMeans assegna ogni punto ad uno dei K cluster, andando a minimizzare la seguente funzione obiettivo per ogni punto x:

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} ||x - m_i||^2 \qquad (1)$$

Nella Eq. (1), C_i rappresenta l'i-esimo cluster e m_i il corrispondente centroide, mentre SSE indica Sum of Squared Error. La misura di distanza è relativa ad ogni dimensione dei punti.
Il processo di clustering specificato dall'algoritmo può essere condensato in 4 step principali:

- **Inizializzazione** dei cluster: vengono presi K punti del dataset in modo random, e quei punti rappresenteranno i centroidi dei cluster iniziali;

- **Assegnazione** dei punti: ogni punto viene assegnato al cluster più vicino ad esso. Per fare questo si va a calcolare la distanza del punto da ogni centroide, e si assegna il punto al cluster corrispondente al centroide con distanza minore;

- Calcolo dei **centroidi**: i centroidi a questo punto vengono aggiornati, calcolando la media delle coordinate dei punti presenti nel cluster che rappresentano;

- Il processo di assegnazione dei punti e ricalcolo dei centroidi viene ripetuto fino a quando non si arriva a convergenza.

In questo lavoro la **convergenza** è stata impostata tramite un numero massimo di iterazioni, 35, e tramite la percentuale di punti che variano di cluster dopo ogni iterazione. Se questa percentuale scende sotto lo 0.1%, l'algoritmo termina.
Lo scopo del progetto e quello di misurare 'lo speedup ottenuto parallelizzando l'algoritmo rispetto alla sua versione sequenziale. Per parallelizzare è stato utilizzato OpenMP, il lunguaggio di programmazione utilizzato è quindi il C++.
Tutti gli esperimenti sono stati svolti sul server ssh "papavero.dinfo.unifi.it".
Inoltre, si è scelto di gestire eventuali punti con dimensione maggiore di 2.

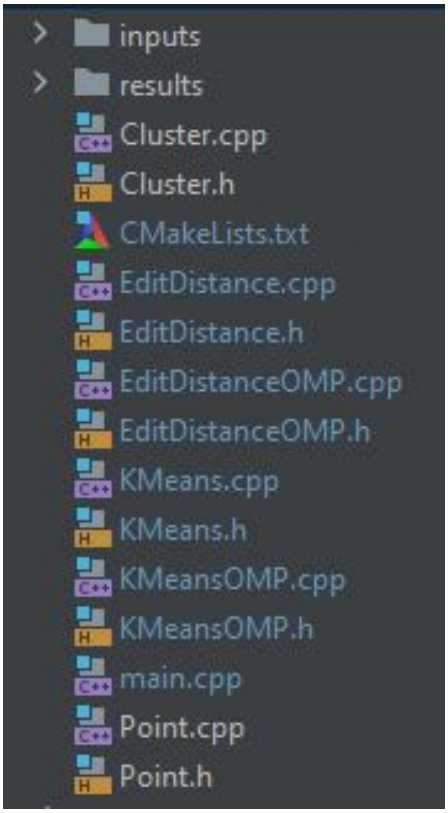

Figure 1. Organizzazione del codice.

2. Organizzazione del codice

Si è utilizzato CLion come IDE e MinGW come compilatore.
Il codice presenta dei file appartenenti ad un altro progetto con directory comune, i file che interessano questo progetto sono:

- **Cluster.h/cpp**: contengono la dichiarazione della classe Cluster e le definizioni dei suoi metodi;
- **KMeans.h/cpp**: contengono la dichiarazione della classe KMeans e le definizioni dei suoi metodi;
- **KMeansOMP.h/cpp**: contengono la dichiarazione della classe KMeansOMP e le definizioni dei suoi metodi con implementazione parallelizzata;
- **Point.h/cpp**: contengono la dichiarazione della classe Point e le definizioni dei suoi metodi;
- **main.cpp**: contiene le funzioni relative ai test svolti ed altre funzioni ausiliarie.

Viene considerata anche la cartella inputs, nella quale sono presenti 4 file csv. L'informazione in essi contenuta verrà utilizzata per istanziare i punti da clusterizzare con KMeans utilizzati per i test.

2.1. Classe Point

La classe Point rappresenta i punti su cui verrà eseguito l'algoritmo.
Ha come attributi pointId, che specifica un identificativo univoco per il punto, clusterId, che specifica l'identificativo del cluster al quale è assegnato il punto, dimensions, che indica il numero di coordinate del punto, e values, che è un vettore in cui vengono salvate le coordinate.
Per quanto riguarda i metodi sono presenti tutti i getter e i setter per i vari attributi, e inoltre vengono definiti due metodi privati, linetoVecTXT e linetoVecCSV, i quali ricevono una linea rispettivamente da un file txt o csv, e convertono l'informazione nelle coordinate del punto.
Infine sono definiti due costruttori, il secondo è quello di copia.

2.2. Classe Cluster

La classe Cluster rappresenta i cluster su cui verranno raggruppati i punti tramite l'esecuzione dell'algoritmo.
Ha come attributi clusterId, che indica un identificativo univoco per il cluster, centroid, che è un vettore in cui sono memorizzate le coordinate del centroide del cluster, e points, un vettore in cui sono contenuti i punti assegnati al cluster.
I metodi comprendono i getter e i setter per il clusterId, due metodi per gestire le coordinate del centroide (per ottenerle e per reimpostarle), tre metodi per gestire i punti assegnati al cluster (tra cui un metodo per resettare il vettore points), ed infine un metodo che restituisce la dimensone del cluster.
Infine, è ovviamente presente un costruttore per la classe.

2.3. Classe KMeans

```
class KMeans {
public:
    KMeans(int K, int epochs);

    void run(std::vector<Point> allPoints, int seed);

private:
    int K, epochs, dimensions, nPoints;
    std::vector<Cluster> clusters;

    void clearClusters();
    int getNearestClusterId(const Point& p);
};
```

Figure 2. Dichiarazione della classe KMeans

La classe KMeans racchiude i parametri e il metodo per eseguire l'algoritmo.
Ha come attributi K, che indica il numero di cluster a cui assegnare i punti, epochs, che indica il numero massimo di iterazioni prima di interrompere l'esecuzione di KMeans, due attributi per memorizzare il numero di punti e le loro dimensioni associate (dimensions e nPoints, che vengono assegnati tramite il metodo run()), e infine clusters, un std::vector in cui vengono salvati i K cluster.
I metodi presenti sono run, che inizializza ed esegue l'algoritmo richiedendo in input una lista di punti da clusterizzare e un seed (per replicare eventualmente i risultati, dato che i centroidi inizialmente vengono scelti casualmente), e due metodi privati, clearClusters per resettare i cluster (utilizzato quando si devono riassegnare i punti ai cluster più vicini) e getNearestClusterId per individuare il cluster più vicino ad un determinato punto richiesto in input.
Il costruttore richiede solo i parametri K e epochs, assegnando inizialmente a nPoints e dimensions valore 0.

2.4. Classe KMeansOMP

La classe KMeansOMP è identica alla classe KMeans. L'unica differenza è l'implementazione del metodo run, che è parallelizzata.
Quest'ultima verrà discussa in Sec. 3

2.5. main.cpp

Nel file sono presenti i test relativi alla parallelizzazione dell'algoritmo di KMeans (presentati in Sec. 4), insieme alla funzione di **main** che esegue i test, e alla funzione **readPointsCSV**.
Quest'ultima è la funzione che si occupa di istanziare gli oggetti di classe Point su cui eseguire poi l'algoritmo. Come si intuisce dal nome, è pensata per leggere i valori delle dimensioni dei punti da dei file csv, richiede in input il nome del file e un intero chiamato stop, che indica quante colonne si andranno a considerare per le dimensioni dell'oggetto di classe Point.
Per ogni riga del file letta, istanzia il corrispondente Point e lo aggiunge ad una lista (std::vector) che poi restituisce in output.

3. Parallelizzazione con OpenMP

Nel codice della classe KMeans sono presenti vari cicli for che potenzialmente sarebbero parallelizzabili, tuttavia molti di essi presentano relativamente poche iterazioni, e quindi si è valutato di lasciare l'esecuzione sequenziale in corrispondenza di essi.

```
//reassign points to their new clusters
for (int i = 0; i < nPoints; i++) {
    //cluster index is ID-1
    clusters[algPoints[i].getClusterId() - 1].addPoint( p: algPoints[i]);
}
```

Figure 3. Ciclo for per memorizzare nei cluster i punti assegnati

L'unico caso che non si colloca in quest'ottica è il ciclo for utilizzato per assegnare i punti ai nuovi cluster, che itera su tutti i punti da clusterizzare (Fig. 3).
Il motivo per cui si è scelto di non parallelizzare è che l'unica istruzione presente nel ciclo accede ad un std::vector che, con la parallelizzazione, sarebbe condiviso tra tutti i threads. Così facendo, si è verificato sperimentalmente che l'accesso concorrente a questa risorsa condivisa risultava in un errore per lo heap. Una possibile soluzione potrebbe essere l'utilizzo di una direttiva critical, ma ciò renderebbe inutile la parallelizzazione in quanto si imporrebbe che un thread alla volta esegua l'unica istruzione presente nel ciclo.
Il codice è stato quindi parallelizzato in due punti, entrambi relativi al metodo run:

- Nella parte che assegna i punti al cluster più vicino;

- Nella parte che ricalcola il centroide di ogni cluster.

3.1. Parallelizzazione dell'assegnazione dei punti ai cluster più vicini

```
//add all points to their nearest cluster
#pragma omp parallel for default(none) shared(algPoints, changed) num_threads(threads)
for (int i = 0; i < nPoints; i ++) {
    int currentClusterId = algPoints[i].getClusterId();
    int nearestClusterId = getNearestClusterId( p: algPoints[i]);
    if (currentClusterId != nearestClusterId) {
        #pragma omp atomic
        changed ++;
        algPoints[i].setClusterId( c: nearestClusterId);
    }
}
```

Figure 4. Parallelizzazione in corrispondenza dell'assegnazione dei punti ai cluster più vicin

In Fig. 4 si osserva come venga creata una sezione **omp parallel for** specificando come shared algPoints, che è la lista contenente i punti sui quali viene eseguito l'algoritmo, e changes, che è un contatore relativo al numero di punti che vengono assegnati ad un nuovo cluster.
Per gestire l'accesso concorrente a queste due variabili, è stata utilizzata una direttiva **omp atomic** in corrispondenza dell'incremento di changes. Così facendo, si assicura che l'accesso alla variabile sia atomico, impedendo eventuali modifiche simultanee dei vari threads.
Invece per algPoints non è stato necessario specificare una direttiva **omp critical**, dato che l'accesso concorrente alla lista serve per modificare l'attributo clusterId di ogni elemento della lista. Quindi non avverrà mai un tentativo di accesso contemporaneo da parte di due o più threads ad uno stesso oggetto di classe Point, come invece sarebbe avvenuto in Fig. 3.
In quel caso infatti, si accede alla lista di cluster per andare ad aggiungere gli oggetti di classe Point alle liste private di punti degli oggetti di classe Cluster, e così facendo c'è la possibilità di chiamare in modo concorrente il metodo addPoint su uno stesso oggetto Cluster.

3.2. Parallelizzazione del calcolo delle coordinate dei nuovi centroidi

In Fig. 5 osserviamo che sono presenti tre cicli for annidati. Si è deciso di inserire un blocco

```
//recalculating the center of each cluster
for (int i = 0; i < K; i++) {
    int clusterSize = clusters[i].getClusterSize();
    for (int j = 0; j < dimensions; j++) {
        double sum = 0.0;
        #pragma omp parallel for default(none) firstprivate(i, j) shared(clusterSize) \
        reduction(+: sum) num_threads(threads)
        for (int p = 0; p < clusterSize; p++) {
            sum += clusters[i].getPoint( pos: p).getVal( pos: j);
        }
        if (clusterSize > 0) {
            clusters[i].setCentroidPos( pos: j,  value: sum / clusterSize);
        }
    }
}
```

Figure 5. Parallelizzazione in corrispondenza del calcolo dei nuovi centroidi

omp parallel for in corrispondenza del ciclo più interno, dato che vi è una dipendenza tra i tre cicli che impedirebbe un'eventuale sezione omp parallel for al ciclo più esterno specificando una clausola di collapse.
Vengono specificati come firstprivate gli indici dei cicli for esterni, mentre come shared abbiamo la dimensione del cluster (intesa come numero di punti appartenenti al cluster, che è anche il numero di iterazioni del ciclo for parallelizzato).
Lo scopo di questa sezione di codice è calcolare i nuovi centroidi dei K cluster, il primo ciclo for scorre tra i cluster, il secondo tra le dimensioni (intesa come numero di coordinate) dei centroidi, e il terzo calcola la j-esima coordinata del cluster. Per farlo, somma tutte le j-esime coordinate dei punti del cluster, memorizzandola in una variabile chiamata sum, e poi divide per il numero di punti del cluster.
Per questo è stata specificata una clausola di **reduction** per la variabile sum.

4. Esperimenti svolti

Tornando al file main.cpp, le funzioni ausiliarie scritte per i test sono le seguenti:

- testKMeansTime: richiede in input i parametri per eseguire KMeans, quindi K e epochs, e una lista di punti su cui andare ad eseguire l'algoritmo. La funzione quindi misura il tempo di esecuzione dell'algoritmo e lo restituisce in output.

- testKMeansOMPTime: richiede in input i parametri per KMeans, una lista di punti da

clusterizzare e il numero di threads per la parallelizzazione. Viene misurato il tempo computazionale della versione parallelizzata di KMeans, che è anche l'output della funzione.

- **testKMeans**: funzione in cui sono esplicati i test veri e propri, viene eseguita dal main. Gli speedup vengono calcolati sfruttando le due funzioni descritte in precedenza, senza però definire una funzione dedita a questo.

In generale il principale parametro tenuto in considerazione per gli esperimenti è ovviamente il numero di threads usati per la parallelizzazione, mentre per gli altri ci basiamo sulla **complessità** dell'algoritmo:

$$\mathcal{O}(n * K * I * d) \tag{2}$$

Nella Eq. (2) viene indicato che la complessità dipende dal numero di punti (n), dal numero di cluster specificati (K), dal numero di iterazioni (I), e dal numero di dimensioni associate ai singoli punti (d).
Terremo quindi conto di tutti questi parametri nei test, tranne il numero di iterazioni per il quale sono state fatte delle scelte in precedenza che verranno mantenute tali (Sec. 1, nella parte relativa alla convergenza).
I risultati vengono salvati in appositi file csv contenuti nella cartella results, ed eventuali grafici vengono generati tramite del codice python (non reso disponibile).
I risultati dei test vengono salvati in appositi file csv contenuti nella cartella **results**.
Di seguito vengono presentati i test svolti per l'algoritmo KMeans.

4.1. Test1: numero di punti minimo

Con questo primo breve test si voleva intuire quale potesse essere il numero minimo di punti per il quale si rivelasse effettivamente utile la parallelizzazione. Si è scelto il numero di punti come parametro anche perchè la parte parallela del codice si riferisce a cicli for che iterano appunto sui punti da clusterizzare.
Ci aspetteremmo che, all'aumentare del numero di punti (e delle loro dimensioni) lo speedup tra versione sequenziale e parallela risulti maggiore a parità di numero di threads utilizzati nella parallelizzazione.
Scrivere risultati attesi, e che aumentando il numero di punti ci si aspetta che lo speedup sia maggiore a parità di nThreads.

	3000 points 2D	19000 points 8D
4 threads	0.946291	2.23869
8 threads	0.948598	3.27011
4 threads	0.989534	2.64693
8 threads	0.964235	3.29542

Table 1. Tabella con i valori di speedup al variare di numero dei punti da clusterizzare e numero di threads usati per la versione parallela. La differenza tra le prime due righe e le ultime due (indicate sempre con 4 e 8 threads) è che nelle ultime due si è provato ad aumentare K per aumentare la complessità dell'algoritmo.

Osservando la Tab. 1 si vede chiaramente come con 3000 punti a 2 dimensioni l'algoritmo parallelo ha prestazioni leggermente inferiori alla versione sequenziale, anche se aumentando il numero di cluster da ricercare (da 3 a 6) il valore di speedup tende maggiormente ad 1.
Questo potrebbe indicare che già con pochi punti in più riusciremmo ad avere uno speedup positivo, ed infatti col secondo dataset più piccolo disponibile, 19000 punti con 8 dimensioni, già abbiamo dei risultati molto significativi, nonostante il numero relativamente ridotto di threads utilizzati.
Si è deciso di non usare un numero elevato di threads proprio per concentrarsi sul numero di punti e per valutare quindi il loro effettivo impatto.
Abbiamo quindi dei risultati in linea con le aspettative, e questo mostra l'effettiva utilità dell'approccio parallelo. Infatti, per la versione sequenziale l'aumento della complessità dovuto al crescente numero di punti porta ad un aumento dei tempi computazionali, mentre questo aumento è molto più contenuto nella versione parallela e questo porta inevitabilmente ad una crescita del valore di speedup.

4.2. Test2: variare nThreads e K

Per il secondo test sono stati fatti variare il numero di threads e il numero di cluster ricercati dall'algoritmo, come indicato in Fig. 6.
Ciò che ci aspettiamo è che, all'aumentare del numero di threads ovviamente l'algoritmo parallelo risulti sempre più veloce di quello sequenziale, ottenendo quindi un aumento dello speedup. Il numero di threads viene fatto variare tra 2 e 16 considerando i multipli di 2.
Per quanto riguarda il numero di cluster, dato che influiscono sulla complessità, all'aumentare di K crescerà anche il tempo computazionale dell'algoritmo sequenziale, e quindi ci aspetteremmo che, a parità di numero di threads, con più cluster la parallelizzazione sia più impattante.

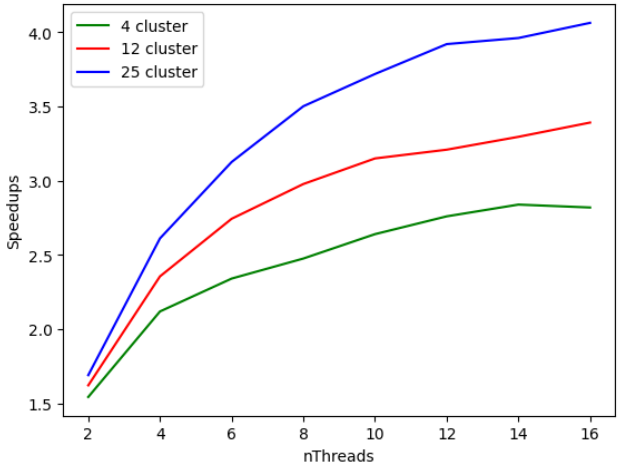

Figure 6. Grafico con i valori di speedup al variare di nThreads. Le 3 linee si riferiscono a diversi valori di K, in verde abbiamo 4 cluster, in rosso 12 ed in blu 25. L'algoritmo è stato eseguito su 500000 punti con 5 dimensioni.

Si osserva come per tutti e 3 i valori di K, aumentando il numero di thread utilizzati per la parallelizzazione aumenta anche lo speedup, come ci aspettavamo, anche se la crescita non è uguale per le 3 linee.
Si nota anche che con un numero maggiore di cluster l'aumento del numero di threads porta ad un valore di speedup maggiore, soprattutto se comparato con l'esecuzione associata ad un valore inferiore di K (ad esempio, con 16 threads e 4 cluster ho uno speedup di 2.81982, mentre con 16 threads e 25 cluster lo speedup è di 4.06303).
Nel testing ci siamo fermati ad un numero massimo di threads pari a 16, ma questo può essere ovviamente aumentato, anche se la crescità dello speedup non sarà sempre costante, ci aspettiamo che dopo un certo numero di threads la CPU risulti satura e quindi avremo un'attenuazione del trend osservato nei grafici (cioè lo speedup rimarrà invariato aumentando il numero di threads, questo perchè il tempo impiegato a gestire i threads diverrà maggiore del tempo guadagnato dalla divisone del lavoro tra i threads stessi).

4.3. Test3: variare K

In questo ultimo test viene fatto variare il numero di cluster K generato dall'algoritmo di KMeans, con un numero fisso di threads per la versione parallela.
I valori considerati di K sono 4, 10, 20 e 30.
Ci aspetteremmo che, dato che K influisce positivamente sulla complessità dell'algoritmo, aumentando K aumenti anche lo speedup ottenuto con la versione parallelizzata di KMeans.

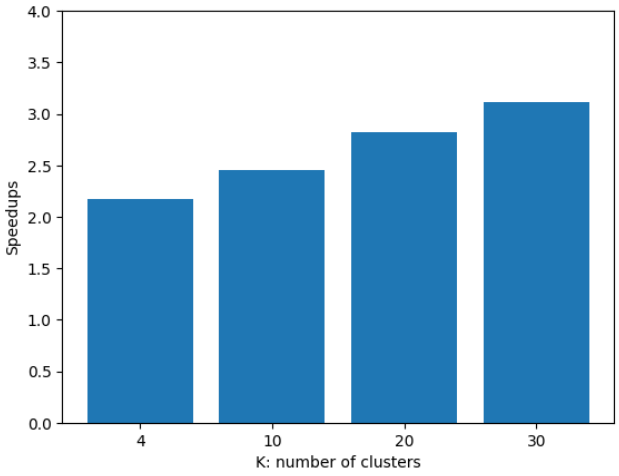

Figure 7. SpeedUp al variare di K con numero di threads fisso pari a 8. L'algoritmo è stato eseguito su 4 milioni di punti con 3 dimensioni.

Nuovamente, il numero di threads è stato tenuto più basso per isolare l'impatto effettivo del numero di cluster sulle prestazioni.
I risultati mostrati in Fig. 7 sono in linea con le attese, notiamo infatti che aumentando K aumenta anche lo speedup, mostrando quindi che la versione parallelizzata di KMeans riesce

a gestire meglio l'aumento della complessità dell'algoritmo (a parità di parametri con la versione sequenziale).

4.4. Considerazioni finali sul numero di punti

Oltre al primo test (Sec. 4.1), possiamo fare delle considerazioni generali sul parametro indicante il numero di punti su cui eseguiamo l'algoritmo semplicemente confrontando i risultati degli altri due test, in quanto vengono effettuati su un numero sempre maggiore di punti (e relative dimensioni associate).

	Speedup with 8 threads
3000 points 2D	0.948598
19000 points 8D	3.27011
500000 points 5D	2.47619
4mln points 3D	2.17211

Table 2. Tabella con i valori di speedup al variare di numero dei punti da clusterizzare con numero di threads pari a 8.

Sono state scelte e mostrate in Tab. 2 le configurazioni più simili tra i vari test, oltre al numero di punti e di dimensioni varia lievemente solo il numero di cluster K (3 per la prima riga, 5 per la seconda, 4 per le ultime due).
Osserviamo che all'aumentare della complessità dell'algoritmo dovuta al numero di punti e di dimensioni, inizialmente a parità di configurazione relativa agli altri parametri osserviamo un aumento dello speedup, mentre poi lo speedup rimane sempre positivo ma è in diminuzione.
Probabilmente questo è dovuto ad un rallentamento dello heap, che non riesce a gestire quel determinato numero di valori (terza e quarta riga).
Guardando i risultati del terzo test (Sec. 4.3) si può intuire un altro fattore che porta a questa conclusione, ovvero il valore di K. Smistando i punti su più cluster (25), osserviamo uno speedup pari a 3.11685, che è un risultato comparabile con la miglior configurazione presentata nella tabella analizzata in questa sezione.
Un altro fattore che influisce è sicuramente il numero di threads utilizzati, infatti osservando i risultati del secondo test (Sec. 4.2) vediamo come con 16 threads i risultati siano migliori (3.50519) di quelli analizzati in Tab. 2.

I risultati proposti servono come intuizione per gestire la complessità derivante da un numero crescente di punti e di dimensioni.