K-means parallelo

Mattia Matucci
E-mail address
mattia.matucci@stud.unifi.it

Abstract

Per cercare di migliorare le prestazioni dell'algoritmo K-means esistono due possibilità lavorare sull'assegnazione iniziale dei centroidi oppure ottimizzare le operazioni di formazione dei cluster. In questa relazione è proposta una possibile implementazione parallela della parte di ottimizzazione. L'idea che ha guidato lo sviluppo del codice è stata quella di ridurre al minimo gli accessi in scrittura in memoria condivisa e l'overhead dovuto alla gestione dei thread.

1. Introduzione

In questa relazione viene presenta una possibile implementazione parallela dell'algoritmo K-means. Al suo interno sarà dedicata un'ampia parte alle scelte implementative fatte nella scrittura del codice e solo alla fine verranno presentati i risultati ottenuti.

L'algoritmo K-means è il più famoso algoritmo di clustering non gerarchico. Tale fama ha portato a realizzarne numerose varianti, in gran parte per cercare di migliorarne le prestazioni. Le strade percorribili per raggiungere tale risultato sono lavorare sull'assegnazione iniziale dei centroidi oppure ottimizzare le operazioni di formazione dei cluster.

Lavorare sull'assegnazione iniziale dei dei centroidi oltre che ha fornire migliori prestazioni, se il tempo richiesto è minore di quello guadagnato con un assegnazione migliore, offre la possibilità di avere risultati migliori alla fine dell'esecuzione. I risultati finali e il tempo d'esecuzione dell'algoritmo infatti dipendono fortemente dai centroidi iniziali. Una variante che si concentra su questa fase è il K-means++.

La seconda strada prevede di ottimizzare le operazioni di formazione dei cluster, tra la possibili ottimizzazioni rientra anche l'esecuzione parallela della parte in questione. Molte delle operazioni di questa fase infatti possono essere facilmente parallelizzate. Ottimizzare questa fase può portate grandi benefici in termini di tempo di esecuzione, a patto di ridurre al minimo il tempo di gestione dei thread. Per questo motivo in questa implementazione la quasi totalità del codice è stata ridotta ad un unica grande regione parallela, ciò significa dover gestire un'unica volta in tutta l'esecuzione dell'algoritmo il tempo per la creazione e la distruzione dei thread. In questo progetto la parallelizzazione è avvenuta attraverso l'utilizzo di direttive al compilatore della famiglia di OpenMP.

La scrittura e l'esecuzione dell'algoritmo proposto è stata realizzata su un sistema Windows 11 con CPU Intel i7-10710U con 6 core fisici e 12 thread e una RAM da 16 GB, come compilatore C++ è stato scelto MinGW.

Dal momento che l'algoritmo proposto opera per cercare di ottimizzare la seconda fase, è stata adottata l'inizializzazione casuale dei centroidi, come proposto nella variante classica. Questo però non impedisce di combinare eventualmente tale algoritmo con una variante più ottimizzata della prima fase.

2. Implementazione

In questa sezione vengono presentate le scelte implementative realizzate, a partire dalla rappresentazione dei dati. La sezione è stata suddivisa in sottosezioni seguendo il filo logico dettato dall'algoritmo K-means nella sua versione classica. Viene qui riportato uno schema riassuntivo dell'algoritmo.

```
inizializzazione centroidi;
do {
 formazione dei cluster;
 aggiornamento centroidi;
   } while(convergenza)
```

Tale schema non è tuttavia stato applicato nell'implementazione dell'algoritmo, poiché è stata preferita una struttura che cercasse di ridurre il numero di cicli all'interno del codice. Questa scelta porta notevoli benefici, in termini di tempo, sia all'esecuzione sequenziale che a quella parallela. Nel proseguo della sezione è importante ricordare che durante la scrittura del codice le decisioni prese avevano come fine quello di ottimizzare quanto più l'esecuzione parallela.

2.1. Rappresentazione dei dati

Un passo fondamentale per ottenere buone prestazioni è la scelta di una buona rappresentazione dei dati. In questo caso dovevano essere rappresentati l'insieme dei punti da clusterizzare e l'insieme dei centroidi. In entrambi i casi è stato utilizzato la metodologia Arrays of Structure per rappresentarli. Tale scelta è stata dettata dal dover utilizzare tutte le coordinate, in questo caso x e y, per calcolare le distanze tra gli elementi dei due insiemi. Con tale rappresentazione è infatti sufficiente un solo accesso alla memoria per leggere le coordinate di un singolo punto, visto che saranno disposte in modo contiguo.

Nella rappresentazione dei punti è inoltre presente l'indicazione del cluster a cui sono associati, mentre nella rappresentazione dei centroidi viene salvato il numero di punti associati ai centroidi. Sapere il numero dei punti di un cluster è importante per aggiornare la posizione dei centroidi.

2.2. Formazione dei cluster

In questa parte della relazione è analizzata l'operazione di formazione dei cluster, cioè l'assegnazione di ogni punto ad un cluster. Per fare questa operazione viene richiesto di calcolare la distanza per ciascun punto dell'insieme con ogni centroide. Se n è il numero dei punti da clusterizzare e k è il numero dei cluster, ad ogni iterazione dell'algoritmo viene richiesto di calcolare kn distanze. Questa serie di operazioni tra loro indipendenti è un'ottima occasione per parallelizzare l'algoritmo.

L'implementazione delle operazione è avvenuta con due cicli for, il ciclo più esterno scorre l'insieme di punti da clusterizzare mentre quello interno scorre sui centroidi. La parallelizzazione è avvenuta solo sul ciclo più esterno, così ogni punto viene processato da un solo thread. Scegliendo questa parallelizzazione non è necessario utilizzare nessuna struttura per memorizzare le distanze, dal momento che lo stesso thread che ha calcolato le distante per un punto ha anche il compito di assegnare il punto al centroide più vicino, prima di passare al punto successivo.

Sempre lo stesso thread ha il compito di registrare l'eventuale cambiamento di cluster del punto rispetto l'iterazione precedente dell'algoritmo. Per registrare il cambiamento è utilizzata la variabile condivisa *change*. In questa fase ogni thread, quando si verifica un cambiamento setta la variabile a *True*. Visto che la scrittura sulla variabile non è protetta, ciò è fonte di race condition, tuttavia per la corretta esecuzione questo non è fonte di problemi, perché questa variabile è utilizzata come una flag per la condizione di convergenza.

Inoltre, ogni thread si costruisce un array privato di centroidi al cui interno memorizza il numero di punti associati a ciascun cluster e la somma delle coordinate, rispetto agli assi x e y, dei punti appartenenti al cluster. Eseguire tale passaggio all'interno dello stesso ciclo utilizzato per la formazione dei cluster è importante perché è possibile ridurre drasticamente il costo dell'aggiornamento dei centroidi. Per come è stata implementata questa operazione non viene richiesta nessuna forma di sincronizzazione dal momento che ogni thread utilizza un array privato.

2.3. Aggiornamento centroidi

In questa fase vengono aggiornate le posizioni di tutti i centroidi, per farlo viene calcolata la media su ciascun asse, dei punti che sono stati associati al centroide nel passo di formazione dei cluster. Viene utilizzata la media dal momento che, in questo caso sia l'asse x che y sono valori continui.

Questa operazione tipicamente viene fatta con un algoritmo che ha complessità O(n): in particolare il vettore dei centroidi viene sovrascritto, con con la somma dei valori assunti lungo ogni asse dai punti appartenenti al cluster. A questo punto i valori ottenuti vengono divisi per il numero dei punti nel cluster, ottenendo così le nuove coordinate. Tale operazione affinché sia parallelizzata correttamente deve gestire l'accesso in scrittura alla memoria condivisa da parte di più thread, ciò risulta essere un limite per le prestazioni. L'obiettivo è quindi ridurre al minimo tali operazioni.

Nel codice questa fase è stata implementata in modo diverso: nella sezione 2.2 viene costruito per ogni thread un vettore privato, che contiene il numero dei punti associati ad ogni cluster e le loro somme, rispettivamente ai punti visti dal thread. Per implementare questa operazione è sfruttato il ciclo di formazione dei cluster, pertanto non è necessario applicare l'algoritmo classico, che ha complessità O(n).

Adesso è tuttavia necessario riportate le informazioni contenute nei vettori privati nel vettore condiviso, per farlo ogni thread accede in modo mutuamente esclusivo al vettore condiviso per scrivere le informazioni raccolte privatamente. La figura 1 mostra l'implementazione di questo passo nel codice, da ciò si vede l'utilizzo massicci della direttiva *atomic*. In questo modo però gli accessi alla memoria condivisa sono comunque sensibilmente ridotti, tale implementazione infatti richiede kt con k numero di cluster e t numero di thread, mentre l'implementazione base richiede n accessi in memoria, uno per ogni punto da clustrerizzare.

Una volta riportati tutti i dati nel vettore condiviso basta eseguire le medie, in questo caso delle divisioni dal momento che sono disponibili le somme dei valori e il numero di elementi nei cluster.

2.4. Criterio di convergenze

La teoria garantisce la convergenza del K-means, per questo motivo è stata utilizzata la classica condizione di uscita, cioè *nessun cambiamento dei cluster*. L'implementazione della condizione d'arresto è stata fatta attraverso la variabile condivisa *change*, già trattata nella sezione 2.2.

```
  for (int j = 0; j < numCentroids; j++) {
#pragma omp atomic update
        centroids[j].numPointsAssigned += privateCentroids[j].numPointsAssigned;
#pragma omp atomic update
        centroids[j].x += privateCentroids[j].x;
#pragma omp atomic update
        centroids[j].y += privateCentroids[j].y;
  }
```

Figura 1. Riduzione dei vettori privati nel vettore condiviso

La condizione d'arresto viene verificata da ogni singolo thread ad ogni iterazione dell'algoritmo k-means, in questo caso implementato con un *do-while*. Questa implementazione permette di restare sempre all'interno della sezione parallela dovendo così pagare un'unica volta in tutta l'esecuzione il tempo della creazione e distruzione dei thread. Per garantire la correttezza dell'algoritmo ad inizio di ogni iterazione la variabile *change* viene impostata a False. Dal momento che la scrittura della variabile avviene immediatamente dopo la verifica della condizione d'uscita è necessario introdurre una barriera prima di procedere, in questo modo prima di resettarla tutti i thread avranno verificato tale condizione.

2.5. Considerazioni finali

Grazie a questa implementazione dell'algoritmo K-means è possibile eseguire quasi tutto l'algoritmo in una regione parallela, pagando solo una volta l'inizializzazione dei thread. Per semplicità l'inizializzazione dei vettore dei centroidi viene fatta sequenzialmente, dal momento che l'impatto è minimo sull'esecuzione dell'algoritmo.

3. Esecuzione

La sezione si occupa di analizzare i risultati, in termini di tempo, per differenti esecuzioni al variare del numero di thread utilizzati. Nei grafici proposti in seguito sono stati rappresentati i dati fino a 13 thread, ovvero uno in più rispetto a quelli disponibili sulla macchina. Per poter confrontare i risultati tra loro sono stati fissati dei semi, dal momento che sia i punti che i centroidi sono inizializzati in modo casuale. I semi utilizzati per questi test sono stati {4, 19, 27, 47, 52}. Nel grafico 2 è riportata la media dei tempi di esecuzione, in secondi, dell'algoritmo. I tempi sono stati misurati eseguendo il codice in modalità release.

Alla fine sarà dedicata una breve sezione all'analisi di situazioni in cui è stato variato il numero di punti da clusterizzare o il numero di cluster

I tempi dell'esecuzione sequenziale sono stati misurati eseguendo lo stesso algoritmo descritto nella relazione, tuttavia per evitare l'overhead dovuto alla gestione del multithreading, sono state ignorate tutte le direttive di OpenMP. Questa scelta implica che non è stata realizzata nessuna ottimizzazione per l'esecuzione sequenziale. Dal grafico è possibile vedere un rallentamento delle prestazioni passando da 6 a 7 thread, questo presumibilmente è spiegabile grazie al superamento del numero di core fisici della macchina, quindi l'introduzione di overhead per la gestione dei thread.

3.1. Speedup

Lo speedup è stato misurato dividendo la media dei tempi d'esecuzione sequenziale con quella parallela sullo stesso carico di lavoro. La media dei tempi è stata fatta sui tempi d'esecuzione misurati su ogni seme. Il grafico 3 riporta lo speedup al variare del numero dei core usati.

Dalla figura emerge, in modo prevedibile, lo stesso comportamento per il passaggio da 6 a 7 thread. Questo passaggio inoltre suddivide graficamente lo speedup in due parti pressapoco uguali. In entrambi i casi lo speedup cresce molto lentamente evidenziando fin da subito un distaccamento dallo speedup lineare. Dai test fatti si evidenzia come il massimo valore raggiunto sia stato ottenuto utilizzando il numero massimo di thread disponibili. In seguito il codice è stato eseguito con un numero maggiore di thread rispetto a quelli disponibili, ma sono stati ottenuti sempre risultati peggiori in termini di tempi medi di esecuzione. Nei grafici sono riportati i dati ottenuti eseguendo da 13 a 15 thread, da cui e possibile evidenziare un comportamento che rimane pressapoco lo stesso anche aumentando il numero di thread.

3.2. Test aggiuntivi

In questa sezione analizziamo i risultati ottenuti dall'algoritmo con un numero diverso di punti di clusterizzare o con un numero diverso di cluster. I grafici relativi sono riportati in figura 4 e 5. Dai grafici possiamo vedere che il tempo d'esecuzione dipende linearmente dal numero di punti, infatti dividendo il tempo registrato per il numero di punti possiamo osservare che il tempo per punto è pressappoco lo stesso. Questo discorso invece non vale per il numero di cluster, in questo caso infatti non sembra sussistere nessuna relazione tra il tempo d'esecuzione e il numero di cluster. Questa è una buona notizia dal momento che il codice è stato pensato per non dipendere da K.

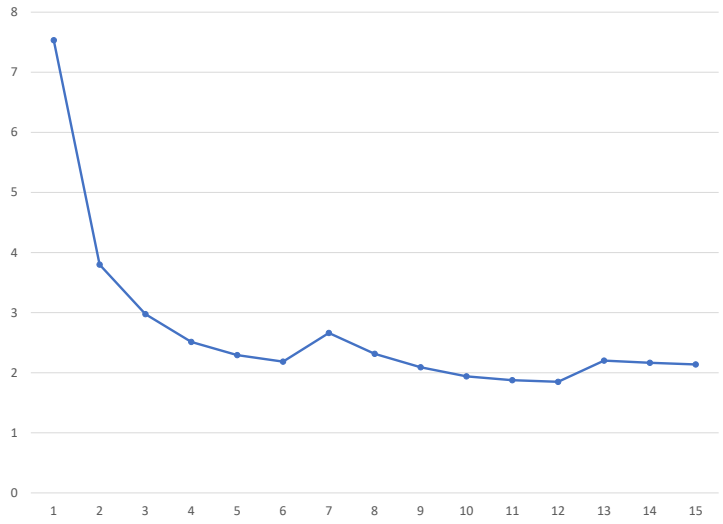

Figura 2. Media dei tempi misurati in secondi al variare del numero dei thread, per un carico di lavoro di un milione di punti e 20 centroidi

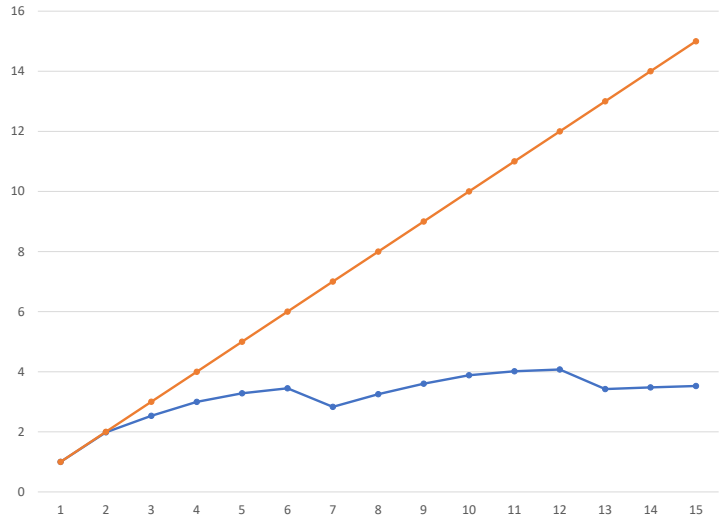

Figura 3. In arancione è riportato lo speedup lineare mentre in blu è riportato lo speedup ottenuto con l'algoritmo implementato. Il carico di lavoro prevedeva un milione di punti e 20 centroidi

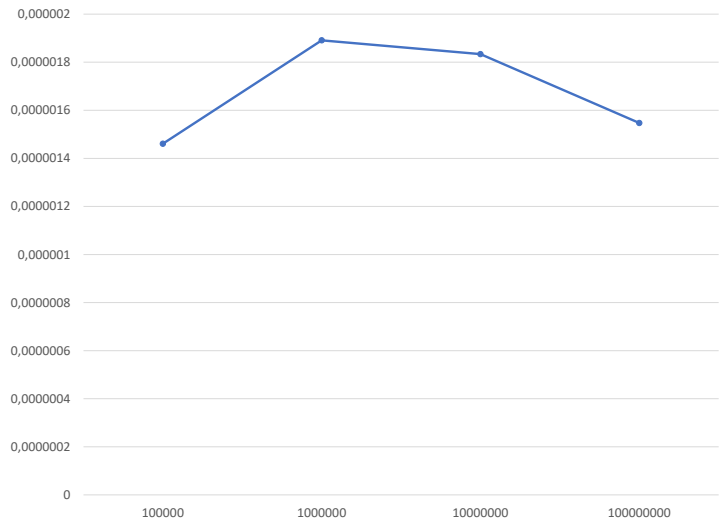

Figura 4. La linea rappresenta il tempo d'esecuzione dell'algoritmo diviso il numero di punti da clusterizzare

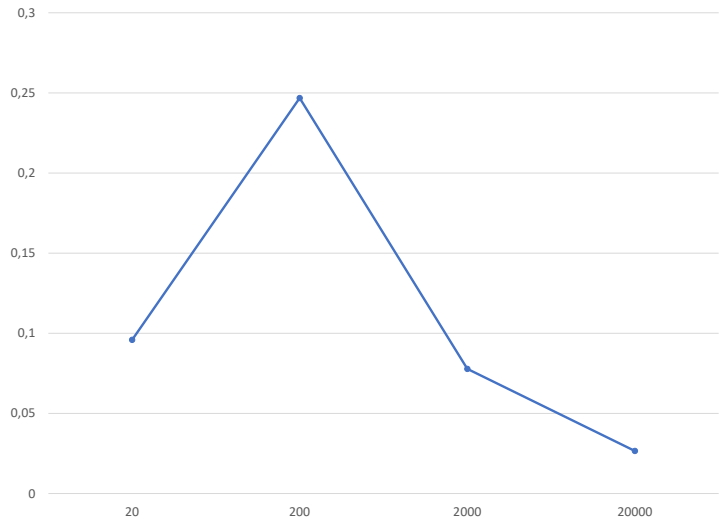

Figura 5. La linea rappresenta il tempo d'esecuzione dell'algoritmo diviso il numero di cluster