Elaborato Programmazione Parallela

Anno Accademico 2022/23
Alberto Biliotti
Matricola: 7109894
alberto.biliotti@stud.unifi.it

Abstract

Per svolgere la prima parte del progetto del corso di Programmazione Parallela è stato deciso di sviluppare una versione parallela dell'algoritmo K-Means bidimensionale sfruttando OpenMP con il linguaggio C++.

Descrizione dell'ambiente di lavoro utilizzato

Per sviluppare questo progetto è stato utilizzato come ambiente di lavoro interno (IDE) CLion aggiornato alla versione 2022.2 con sistema operativo Windows 11 e con una CPU Intel di undicesima generazione i7-11800H con frequenza di 2.30GHz, 8 core fisici e 16 virtuali e con 16 GB di memoria RAM disponibile.

1. Caratteristiche implementative sequenziali

Come accennato in precedenza, nel progetto è stato implementato l'algoritmo di clustering K-means, che in maniera iterativa associa ciascuno dei punti in input al gruppo che possiede il centroide più prossimo al punto preso in analisi; i centroidi inizialmente possono essere scelti in maniera casuale o passati anche essi in input. Una volta assegnato ciascun punto ad un gruppo, l'algoritmo ricalcola il centroide di tale gruppo come media aritmetica delle coordinate di tutti i punti appartenenti al grupoi. Nella versione "classica" questo procedimento è ripetuto fino a che i centroidi non rimangono costanti, ma la versione da me implementata prevede che l'algoritmo termini una volta che nessun punto ha variato il proprio gruppo di appartenenza dopo un'iterazione. Inoltre, una volta terminata l'esecuzione dell'algoritmo, calcolo anche la somma degli errori al quadrato (SSE) per poter misurare la "bontà" della mia esecuzione, poiché l'algoritmo K-means presenta un comportamento non deterministico in quanto la scelta dei centroidi iniziali avviene in maniera casuale.

1.1. Formato dei dati

Per poter memorizzare le coordinate dei miei punti e le informazione sul gruppo di appartenenza e sulla distanza del punto dal centroide di tale gruppo è stata scelta una "Structure of Arrays" ossia una Structure composta da quattro vettori che hanno lunghezza pari al numero di punti desiderati e che ad un dato indice contengono le informazioni su un singolo punto. Lo stesso sistema è stato utilizzato per memorizzare le coordinate dei miei centroidi insieme al numero di punti appartenenti al gruppo rappresentato dal centroide.

1.2. Calcolo della distanza

Per calcolare la distanza tra un punto ed un centroide è stato scelto di utilizzare la distanza euclidea sia per il fatto che essa è quella usata più di frequente per effettuare questo tipo di analisi sia per il fatto che, insieme alla scelta del centroide come media aritmetica dei punti, essa permette di minimizzare lo SSE fissando i centroidi iniziali. Volendo pero è anche possibile utilizzare una qualsiasi altra distanza di Minkowski passando come parametro al metodo per calcolare la distanza un valore diverso da 0.

1.3. Implementazione sequenziale

Per implementare una versione sequenziale dell'algoritmo descritto in precedenza è stato deciso di dividere il programma in due file, uno header contenente: gli import delle varie librerie utilizzate nell'implementazione, la dichiarazione delle variabili d'ambiente utili per l'esecuzione del nostro programma, le due funzioni di calcolo della distanza e di generazione di numeri casuali nell'intervallo desiderato ed infine anche la definizione delle due "Structure of Arrays" per contenere i punti ed i centroidi. L'altro file si limita invece ad ospitare la funzione main. Dentro al main, per assegnare i valori ai punti ed ai centroidi, ho scelto di implementare una variabile di ambiente che, se definita, permette di assegnare manualmente le ascisse e le ordinate dei punti e dei centroidi, mentre se la variabile di ambiente non è definita l'assegnazione

delle ascisse e delle ordinate avviene in modalità casuale; il numero dei punti e dei centroidi generati deve essere specificato tramite le due variabili di ambiente "NUM_PUNTI" e "NUM_CENTROIDI". Per fare iterare l'algoritmo è stato scelto di utilizzare un ciclo while che si interrompe quando l'esecuzione non cambia il gruppo di appartenenza di nessun punto, ossia quando la posizione dei centroidi non varia. All'interno di questo ciclo while calcolo per prima cosa, dentro ad un doopio ciclo for innestato, la distanza di ogni punto da ogni centroide e memorizzo quella minore insieme al numero del gruppo del centroide così individuato nella mia Structure of Arrays. Successivamente, dopo aver azzerato le coordinate dei centroidi ed il numero di punti appartenente ad un gruppo, vado a calcolare le coordinate dei nuovi centroidi dentro un altro ciclo for. Nel caso però ad un gruppo non venga assegnato nessun punto, al centroide di tale gruppo verranno assegnate le coordinate di uno qualsiasi dei punti del nostro insieme per evitare che rimangano gruppi vuoti. Infine, dopo la terminazione del ciclo while, calcolo l'errore quadratico medio sommando il quadrato di tutte le distanze dei punti dai centroidi dei loro gruppi e, se la variabile d'ambiente "output" è definita, stampo su un documento CSV le coordinate dei punti e dei centroidi in modo da poter analizzare il risultato della mia esecuzione.

2. Sviluppo del programma parallelo

2.1. Trovare gli hotspot dell'algoritmo

Per poter parallelizzare il codice visto in precedenza è necessario individuare le parti dell'algoritmo che richiedono più tempo e risorse per poter essere eseguite in modo tale da poter parallelizzare tali parti in maniera che il nostro programma parallelo ottenga effettivamente uno speed up rispetto alla versione sequenziale. Per quanto riguarda l'algoritmo K-Means, le parti più esose dal punto di vista computazionale sono state individuate: nel calcolo della distanza dei punti dai centroidi e nella successiva scelta della distanza minima, in quanto è indipendente per ogni punto, potendo quindi essere considerato un problema imbarazzantemente parallelo, nel ricalcolo della posizione dei centroidi ed infine nel calcolo finale della somma degli errori al quadrato. Queste ultime due operazioni sono comunque parallelizzabili ma richiedono necessariamente un minimo di sincronizzazione per poter essere eseguite correttamente, in quanto prevedono la scrittura su variabili condivise.

2.2. Implementazione parallela

Per parallelizzareil programma è stato sufficiente aggiornare il codice sequenziale illustrato in precedenza utilizzando le direttive offerte da OpenMP. È stato quindi scelto di implementare un'unica regione parallela, posta dentro il ciclo while, per poter parallelizzare il calcolo dell'assegnazione dei punti ai gruppi,il calcolo della posizione dei nuovi centroidi oltre al calcolo della somma degli errori al quadrato. Per realizzare ciò è stato necessario usare la direttiva "pragma omp for" per parallelizzare i singoli cicli for definiti in precedenza oltre ad includere tutti questi for dentro una regione parallela definita con la direttiva "pragma omp parallel" È stato inoltre modificato il sistema di calcolo dei nuovi centroidi, inserendo due vettori di appoggio per poter contenere le somme delle coordinate dei centroidi che vado ad aggiornare dopo aver associato il punto ad un cluster. I valori contenuti in tali vettori verranno semplicemente divisi per il numero dei punti assegnati al cluster (che aggiorno anch'esso dopo l'assegnamento di un nuovo punto al cluster) ottenendo cosi le nuove coordinate dei centroidi (di fatti calcolando la media aritmetica delle coordinate dei punti appartenenti al cluster). Infine per poter parallelizzare in maniera più efficiente il programma è stato deciso di includere il ciclo for per il calcolo del SSE dentro quest'unica sezione parallela, includendolo quindi dentro il ciclo while, ponendo però che non siano stati effettuati cambi affinché quest'ultimo for venga effettivamente eseguito.

2.3. Sincronia

Per evitare la presenza di comportamenti inattesi dovuti alla parallelizzazione del codice (ossia di "race condition") è stato necessario inserire alcune forme di sincronizzazione in modo tale da garantire che le parti più "delicate" del codice (ossia quando devo scrivere su una variabile condivisa) non vengano eseguite in parallelo da più thread. Tale rischio è presente in alcuni punti del programma, in particolare nell'aggiornamento del numero di cambi, nel calcolo della somma degli errori al quadrato e nel calcolo dei nuovi centroidi. Le prime due criticità possono essere risolte semplicemente sfruttando la funzionalità delle reduction offerta da OpenMP, che mi garantisce l'assenza di criticità andando a generare una variabile privata per thread e andando ad applicare l'operazione richiesta (nel mio caso semplicemente la somma) su tutte queste variabili private alla fine della sezione (nel mio caso dei for). Per quanto riguarda il calcolo dei nuovi centroidi, in questo caso non posso più affidarmi al meccanismo della reduction in quanto ho necessità di scrivere su intero vettore di coordinate, per questo motivo è stato necessario ricorrere alla direttiva atomic che permette di eseguire in sicurezza operazioni di base come, nel nostro caso, l'incremento di una variabile condivisa.

3. Analisi delle prestazioni e speed up

Per confrontare le prestazioni dell'implementazione parallela rispetto a quella sequenziale sono stati forniti gli stessi punti e centroidi di partenza ad entrambi i programmi (per

fare ciò è bastato non fornire alcun seme di generazione all'algoritmo di generazione dei dati casuali) e sono stati registrati i tempi di esecuzione della parte di programma di nostro interesse (escludendo quindi il tempo di generazione dei punti casuali e quello dell'eventuale stampa dei risultati finali nel file CSV) in modalità release. L'implementazione parallela è stata eseguita inoltre con un numero diverso di thread massimi (2,4,8,16) per osservare il comportamento del codice parallelo al variare del numero di thread massimi utilizzabili. Infine, per registrare correttamente il tempo di esecuzione di entrambe le implementazioni, sono stati eseguiti più volte entrambi i programmi ed è stato scelto come tempo da registrare la media di tutti i tempi di esecuzione.

3.1. Esecuzione con 100000 punti e 30 centroidi

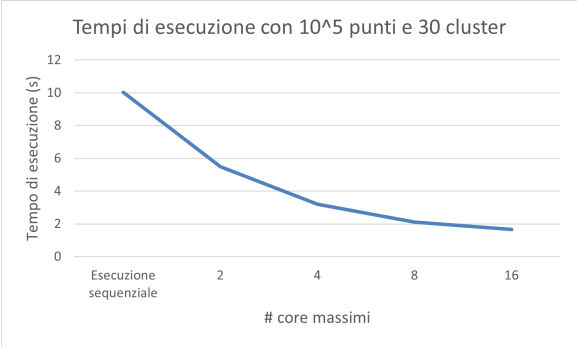

Figure 1. Tempi di esecuzione per 10^5 centroidi e 30 cluster al variare del numero di thread massimi

# thread massimi	Tempo esecuzione (s)
Implementazione sequenziale	10,03
2	5,5
4	3,21
8	2,11
16	1,67

Table 1. Tempo di esecuzione per 10^5 punti e 30 cluster.

Abbiamo deciso di iniziare testando la nostra implementazione parallela su 100.000 punti in ingresso e 30 cluster generati casualmente (ma identici tra le due implementazioni) poiché intendevo dimostrare l'efficienza del programma parallelo anche su un numero di punti ridotto. Il numero abbastanza elevato di cluster è stato scelto in quanto, ricordando che la complessità dell'algoritmo di clustering K-Means sequenziale è dell'ordine di $O(2 * n * k * i)$, ha permesso di avere un tempo di esecuzione consistente in quanto un tempo di esecuzione troppo basso avrebbe risentito maggiormente delle (piccole) differenze dovute alle condizioni in cui si trova il sistema operativo e la macchina al momento del lancio dell'esecuzione. Possiamo quindi osservare come, nonostante tutto, si osservi comunque una drastica riduzione del tempo di esecuzione rispetto alla versione sequenziale tanto che, con tutti i thread utilizzati, il tempo di esecuzione parallela è un sesto di quello dell'esecuzione sequenziale.

3.2. 1000000 punti e 30 centroidi

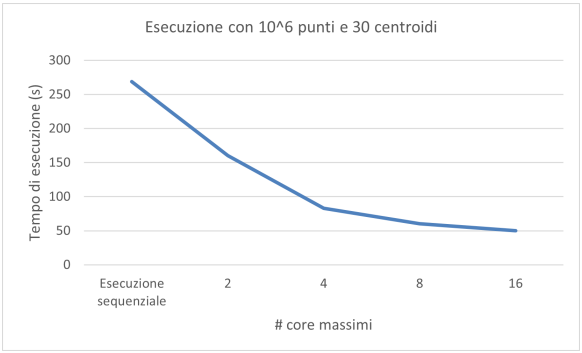

Figure 2. Tempi di esecuzione per 10^6 punti e 30 cluster al variare del numero di thread massimi

# thread massimi	Tempo esecuzione (s)
Implementazione sequenziale	268,8
2	159,96
4	82,98
8	60,34
16	49,86

Table 2. Tempo di esecuzione per 10^6 punti e 30 cluster.

Come secondo test abbiamo deciso di aumentare di un ordine di grandezza il numero di punti mantenendo costante il numero di cluster. Possiamo osservare come, anche in questo caso, i risultati evidenzino la validità della nostra implementazione, se pur in maniera leggermente inferiore rispetto al caso precedente, con un tempo di esecuzione sequenziale che è più di cinque volte quello parallelo con tutti i core fisici e logici a disposizione.

3.3. 10000000 punti e 10 centroidi

Come ultimo test è stato deciso di testare le implementazioni su dieci milioni di punti da raggruppare in dieci cluster. Il numero dei cluster è stato ridotto in quanto altrimenti, con troppi centroidi, il tempo di esecuzione dell'implementazione sequenziale rischiava di diventare veramente troppo alto (quasi un quarto d'ora) rischiando così di dover perdere più di un ora per testare ogni esecuzione con i vari numeri di thread massimi (senza poter utilizzare il computer nel frattempo).

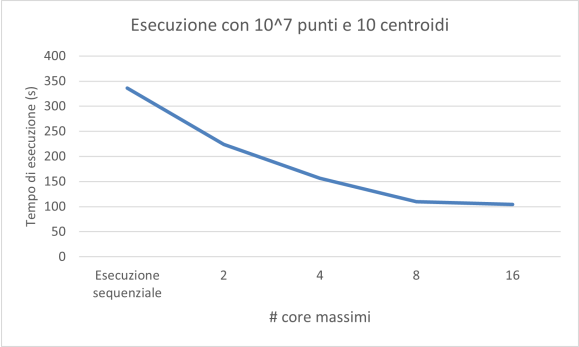

Figure 3. Tempi di esecuzione parallela per 10^7 centroidi e 10 cluster al variare del numero di thread massimi

# thread massimi	Tempo esecuzione (s)
Implementazione sequenziale	335,859
2	223,85
4	156,3
8	109,768
16	88,97

Table 3. Tempo di esecuzione per 10^7 punti e 10 cluster.

Notiamo come all'aumentare del numero di centroidi si abbia sempre un incremento delle prestazioni, anche se in proporzione minore rispetto agli altri due test, ciò è dovuto probabilmente dal numero minore di centroidi che inficia sullo speed up della versione parallela.

3.4. Confronto dello speed up

Per concludere riportiamo il grafico, che rappresenta il valore dello speed up al variare del numero di thread massimi disponibili per le varie combinazione di punti e cluster, ottenuto dividendo per il tempo di esecuzione sequenziale i tempi di esecuzione parallela.

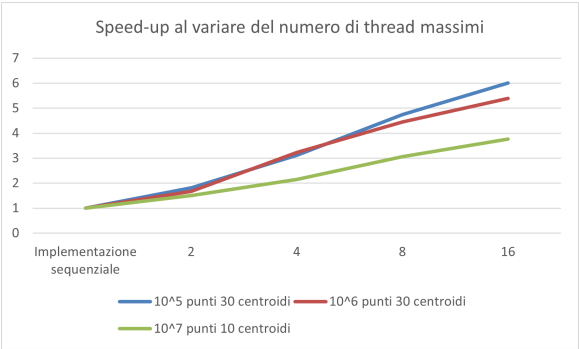

Figure 4. Speed up delle varie combinazioni di numero di punti e centroidi

Come accennato nella sezione precedente, lo speed up dell'esecuzione con dieci cluster risulta essere inferiore rispetto a quello delle esecuzioni con trenta. Ciò è dovuto probabilmente al fatto che il mio codice parallelizzi l'esecuzione anche rispetto ai centroidi, motivo per il quale all'aumentare del numero di cluster la parallelizzazione del mio programma tenderà ad aumentare. Ciò avverrà in maniera più marcata rispetto alla semplice aggiunta di punti, poichè la complessità temporale dell'algoritmo di clustering K-Means è dell'ordine di $O(2*n*k*i)$, quindi aumentando il numero di cluster aumento maggiormente la complessità rispetto al semplice aumento del numero di punti e di conseguenza anche lo speed up dovuto alla parallelizzazione.

4. Conclusione

Per quanto riportato nella sezione precedente, possiamo affermare che la mia implementazione parallela, a prescindere dalla grandezza del problema, abbia un evidente convenienza in termini di tempo di esecuzione rispetto alla versione sequenziale, andando a sfruttare a pieno le potenzialità del processore, come evidenziato anche dalle immagini sotto riportate.

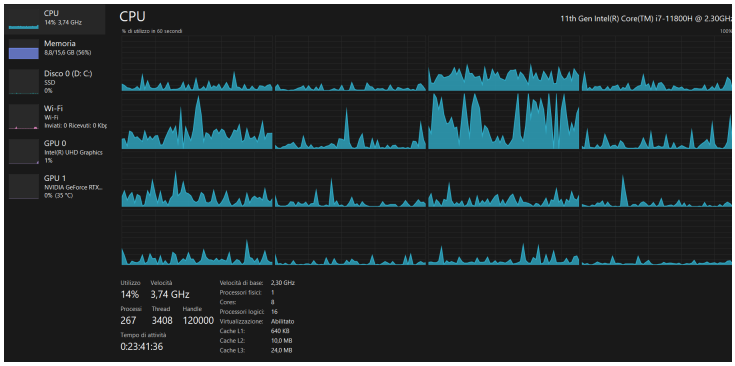

Figure 5. Utilizzo core logici CPU programma sequenziale

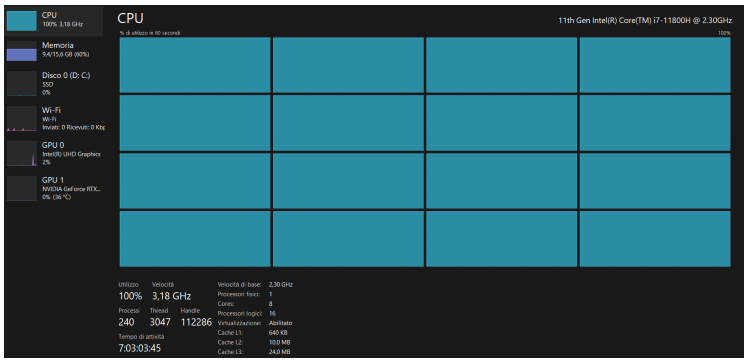

Figure 6. Utilizzo core logici CPU programma parallelo