**Relazione del progetto di Parallel Computing** (**K-Means)**

Giuseppe Martinelli Matr: 7093926

La seguente è una descrizione delle scelte implementative che sono state prese per la costruzione di una versione **sequenziale** e **parallela** dell’algoritmo **K-Means**. Per la loro implementazione è stato utilizzato il linguaggio C e per la parallelizzazione è stata utilizzata l’API OpenMP.

**Descrizione dell’algoritmo**

L’algoritmo **K-Means** è un algoritmo di clustering il cui scopo è quello di partizionare N osservazioni in K insiemi in cui ogni osservazione appartiene al cluster con la media più vicina al punto centrale di tale insieme (il così detto centroide). Per questa motivazione, questa tipologia di clustering prende il nome di “*centroid-base clustering”* .

Gli step che l’algoritmo esegue per ottenere tali insiemi sono i seguenti:

1. Considerando un insieme di N punti complessivi che devono essere clusterizzati, vengono scelti tra questi i **k centroidi;**
2. Per ciascun punto del dataset considerato viene calcolata la distanza rispetto a ciascuno dei k centroidi
3. Tra le varie distanze calcolate per ogni punto, viene mantenuta la distanza minima che è quella che identifica il cluster di appartenenza del punto considerato
4. Per ogni cluster, vengono ricalcolati i centroidi facendo la media delle coordinate di tutti punti
5. Se le coordinate dei centroidi non cambiano, l’algoritmo termina. Altrimenti, si ritorna al punto 2

**K-Means con implementazione sequenziale**

Per poter modellare ciascun punto nello spazio è stata definita una **struct** chiamata *point*, la quale contiene le coordinate spaziali di ciascun punto (tali coordinate rappresentate attraverso due float) e l’id di appartenenza di un punto ad un determinato cluster (rappresentato da un intero e di default inizializzato con il valore -1). Il codice procede alla inizializzazione casuale dei punti e dei centroidi tramite la funzione **point\_init;** si noti come l’id dei centroidi in questo passo sia inizializzato rispetto al loro ordine di creazione.

I punti ed i centroidi vengono memorizzati attraverso due liste di structs, la motivazione per cui si è scelta questa tipologia di logica di memorizzazione (cioè **AoS** rispetto a **SoA**) è dettata dalle seguenti motivazioni:

1. **AoS** tende ad essere più leggibile da parte di un programmatore per ogni ““oggetto”” ( questa so che è una bestemmia, ma è giusto per trasmettere il concetto di insieme di variabili dichiarate con una idea comune)
2. In questo caso, l’utilizzo di **AoS** tende a sfruttare meglio la località di cache dal momento che quelle variabili vengono quasi tutte accesse ad ogni singola invocazione
3. Dal momento che due di queste variabili sono intere, vi è la concreta possibilità di vettorizzazione

Tralasciando motivazione 1. ,le motivazioni 2. e 3. sono a mio avviso **più che sufficienti** nel giustificare la seguente logica di memorizzazione. Ciò non toglie che (nel caso in cui ho tempo) proverò a verificare eventuali speed-up ottenuti con memorizzazioni differenti.

Una volta definiti i punti ed i centroidi da utilizzare viene fatta partire l’esecuzione dell’algoritmo vero e proprio che è stato implementato all’interno della funzione **k\_means** alla quale si passano: i due insiemi points e clusters, il numero di punti ed infine il numero di clusters. In questa funzione vengono scanditi tutti i punti del dataset e per ciascuno di loro viene identificato il centroide più vicino, ovvero a distanza minore.

In questa funzione vengono scanditi tutti i punti del dataset e per ciascuno di loro viene identificato il centroide più vicino, ovvero a distanza minore. Il centroide avente la distanza minore viene trovato dalla funzione **nearestCenter** con l’ausilio della funzione **euclideanDistance**. via via che si calcolano le predette distanze, si mantiene all’interno di una variabile *minDistance* il valore della distanza minima finora trovata, e nella variabile *centroidIndex* l’indice del centroide a minima distanza dal punto attuale. Mediante tale funzione si ottiene dunque l’id del centroide relativo al cluster a cui dovrà appartenere il punto attualmente considerato.

Nella parte finale della funzione “k\_means” avviene invece l’aggiornamento delle coordinate dei centroidi in cui, tale update, considera le coordinate dei punti interni al cluster attualmente considerato facendone una media tra i rispettivi valori.

Qualora non ci dovessero essere punti all’interno del cluster che si sta considerando, verrà ritornato il valore della vecchia coordinata. Questa è una situazione che può capitare in quanto un centroide potrebbe essere molto più lontano dai vari punti rispetto agli altri centroidi, andando così a generare un cluster vuoto alla fine dell’algoritmo. Una volta ottenute le nuove coordinate viene valutata una condizione finale inerente allo stato di convergenza dell’algoritmo. In particolare, la logica implementativa della funzione k\_means è stata definita all’interno di un ciclo while che verifica se il flag “convergence” abbia valore 0.

* In tal caso vuol dire che almeno una delle due coordinate di un centroide è cambiata e dunque si dovrà effettuare una nuova iterazione dell’algoritmo;
* In caso contrario, se il valore di tale flag risulta alterato e portato ad 1 significa che nessuno dei centroidi ha più cambiato i valori delle coordinate e dunque l’algoritmo termina con la situazione di convergenza.

**K-Means con implementazione parallela**

La logica esecutiva dell’algoritmo rimane uguale a quella della sezione sequenziale, ma ovviamente ci sono le seguenti modifiche:

* Riga 35 → #pragma omp parallel num\_threads(nThreads) { … }
* Riga 37 → #pragma omp for schedule(auto)

Il costrutto “for” indica ad OpenMP che l’insieme delle iterazioni del ciclo for seguente alla direttiva dovrà essere distribuito tra i thread considerati, per velocizzare l’esecuzione del ciclo. La seconda clausola visibile “schedule(auto)”, la quale dice ad OpenMP in quale modo le iterazioni del ciclo dovranno essere distribuite tra i threads, e non trovando particolari miglioramenti nella modifica del suo argomento è stato scelto di delegarla al compilatore.

* Riga 45 → #pragma omp for schedule(auto) nowait

Si nota dunque l’aggiunta della clausola “nowait”, che viene applicata per rimuovere la barriera implicita che si genererebbe al termine del primo for. Tale barriera è infatti inutile dato che i punti generati sono indipendenti sia tra loro che dai successivi centroidi da generare, quindi non è necessario l’uso di una barriera per sincronizzare i threads al termine del primo loop.

Nel file utils.c, dove riesiede l’implementazione del K-Means sono state attuate le seguenti modifiche:

* Riga 31 → #pragma omp parallel num\_threads(nThreads)
* Riga 33 → #pragma omp for schedule(auto) private(centroidIndex)

Siccome si sta effettuando l’esecuzione del ciclo in parallelo, dobbiamo stare attenti al fatto per cui non si verifichino race conditions, ovvero situazioni per cui un certo elemento venga aggiornato simultaneamente da più threads. Una variabile che nel ciclo potrebbe dare questo tipo di problema è “centroidIndex” in quanto ogni punto deve essere associato ad un cluster specifico ed accessi contemporanei ad essa potrebbero modificarne erroneamente il valore da associare ad un punto in particolare. Per questo motivo, alle clausole già esposte nel processo di generazione dei dati c’è l’aggiunta della clausola “private(centroidIndex)”, con la quale si fa in modo che ogni thread abbia una copia locale della variabile “centroidIndex” evitando dunque il problema descritto. Questo evita problemi anche nel successivo assegnamento, dove il valore del cluster (contenuto in centroidIndex) essendo a questo punto locale per i threads potrà essere assegnato in sicurezza al punto considerato.Il ciclo interno alla funzione “nearestCentroid(point actualPoint, point clusters[])” invece non può essere parallelizzato perché è molto facile che si verifichino situazioni di race condition. Un ragionamento molto simile è stato fatto per il loop successivo (sempre all’interno della funzione K-means) che è quello in cui avviene l’aggiornamento delle coordinate dei centroidi. Come si può osservare, tale for è stato parallelizzato nella stessa sezione del for prima discusso senza creare una nuova zona parallela. Questo dettaglio consente di incrementare le prestazioni evitando di chiudere tale sezione ed aprirne inutilmente una nuova,

* Riga 44 → #pragma omp for schedule(auto) private(newX,newY)

Dove ora si nota che le variabili poste come private sono le due coordinate del centroide, che vengono via via aggiornate: “newX” e “newY”. C’è da notare anche qui che per gli stessi motivi di prima non è possibile parallelizzare i cicli for interni alla funzione di aggiornamento delle coordinate. Un’altra situazione da discutere è poi quella del ciclo while principale della funzione K-means, che gestisce la convergenza dell’algoritmo. Tale ciclo non risulta possibile da parallelizzare perché potrebbe verificarsi una race condition che porti poi alla modifica del valore del flag “convergence”, la quale potrebbe non far terminare il loop nel caso in cui il valore sia continuamente posto a 0, o magari potrebbe farlo terminare troppo presto (in caso di un altro valore di “convergence”).

**Confronto prestazionale**

In questa sezione, ci concentriamo sui tempi di esecuzione ottenuti con configurazioni di parametri differenti come ad esempio il numero di punti ed il numero di threads. **Si noti che sulla mia macchina il numero di cores disponibili sono 4.**

La seguente tabella riassume i tempi di esecuzione ottenuti con determinate configurazioni, si noti come il primo intero rappresenti il numero di punti generati, il secondo il numero di centroidi ed infine il terzo il numero di threads creati.

**Nota:** Per ogni configurazione sono state eseguite 10 esecuzioni e poi fatta la media

|  |  |  |  |
| --- | --- | --- | --- |
| Configurazione | Tempo sequenziale | Tempo parallelo | Speedup |
| 3000 – 50 – 4 | 0.836400 | 0.428400 | 1.952 |
| 10000 – 150 – 4 | 10.263400 | 3.3117 | 3.099 |
| 10000 – 200 – 4 | 59.706900 | 11.47 | 5.205 |

Come si può notare dalla tabella precedente, il numero di thread è costante. Vediamo adesso cosa succede alla variazione di esso.

**Nota:** Per ogni configurazione sono state eseguite 10 esecuzioni e poi fatta la media

|  |  |  |
| --- | --- | --- |
| Configurazione | Tempo parallelo | Speedup |
| 10000 – 200 – 2 | 11.851000 | 4,9310 |
| 10000 – 200 – 6 | 11.847000 | 4,9327 |
| 10000 – 200 – 8 | 12.0717 | 4.94 |
| 10000 – 200 – 10 | 12.331000 | 4.7391 |
| 10000 – 200 – 12 | 14.162000 | 4.1263 |
| 10000 – 200 – 16 | 15.344000 | 3.811 |
| 10000 – 200 – 18 | 9.13 | 6.539 |
| 10000 – 200 – 400 | 200.745 | // |

Si noti che:

1. Dal momento in cui vi è una parallelizzazione dei centroidi, all’aumentare di essi si ottiene uno speedup maggiore nell’esecuzione del programma parallelo
2. Il miglior speedup a pari di larghezza di input e centroidi ma al differire del numero di thread, si ottiene con esattamente **18 threads**. Questo risultato,che a primo impatto può sembrare anomalo, può essere tranquillamente fattibile; in quanto l’istruzione di *schedule(auto)* può permettere di dividere in maniera efficace l’input. Permettendoci quindi una esecuzione migliore.