Relazione del Progetto di Parallel Computing

K-means

Nome: Guido
Cognome: Ciardi
Matricola: 7090798
Email: guido.ciardi@stud.unifi.it

Per questo progetto sono state implementate una versione **sequenziale** ed una **parallela** inerenti all'algoritmo di clustering **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 K-means

Il **K-means** è un algoritmo di clustering che consente di dividere un insieme di oggetti (identificabili mediante dei punti) in **k gruppi** distinti, formalmente identificati mediante il nome di **clusters**.
Per effettuare tale suddivisione ci si basa su degli oggetti (o punti) "speciali" denominati come **centroidi** (o punti medi), mediante i quali viene identificato ciascun cluster.

La procedura seguita è di tipo iterativo ed è la seguente:

1) Considerando un insieme di N punti complessivi che devono essere clusterizzati, vengono scelti tra questi (con il criterio desiderato) i **k centroidi**;
   Nota: Esistono diversi criteri inerenti alla scelta dei centroidi, uno di questi è quello per cui questi vengano scelti in **maniera casuale**;

Successivamente vengono iterati i tre step successivi fino al raggiungimento della **condizione di terminazione** dell'algoritmo:

2) Per ciascun punto del dataset considerato viene calcolata la distanza rispetto a ciascuno dei k centroidi;
   Nota: L'algoritmo può funzionare con tipi di distanze differenti, ma tale tipo deve essere prefissato inizialmente;

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) Avviene poi un processo di **aggiornamento delle coordinate dei centroidi**, mediante il quale ciascuna nuova coordinata del centroide a cui ci si sta riferendo sarà definita dalla media di ogni rispettiva coordinata dei punti interni al suo cluster;

CONDIZIONE DI TERMINAZIONE: Il processo sopra descritto viene ripetuto fin quando non viene raggiunta la situazione per cui le coordinate dei k centroidi **non** vengono più alterate, la quale rappresenta la situazione di **convergenza**.

L'algoritmo produce come **output** la lista dei clusters finali, ciascuno contenente la sequenza dei punti ad esso associati.

K-means con implementazione sequenziale:

In quest'implementazione sono stati definiti i due vettori **"points"** e **"clusters"** che rappresentano rispettivamente l'insieme dei **punti** da clusterizzare e l'insieme dei **centroidi** inerenti a ciascuno dei clusters che dovrà essere generato.

Ogni punto considerato in questo programma è stato rappresentato come **bidimensionale** mediante la definizione di un'apposita **struct**. Ad esso viene anche associata una terza caratteristica che è quella inerente all'identificativo del cluster a cui appartiene. Mentre le due dimensioni (**x**, **y**) sono state definite come valori "**float**", l'identificatore (**"clusterId"**) risulta invece essere di tipo "**intero**".

Come preannunciato durante la descrizione dell'algoritmo, i punti da considerare sono di due tipi, ovvero, quelli standard da clusterizzare ed i centroidi. A tal proposito, mentre l'id del cluster dei punti standard viene valorizzato con **-1** alla loro creazione, l'id dei centroidi si inizializza in base al loro **ordine di creazione** (che corrisponderà infine all'id del cluster a cui infine apparterranno). L'assegnazione di tali valori a ciascun punto avviene mediante la funzione **"point_Initialization"**.

La logica applicata alla memorizzazione dei punti nei due array (points e clusters) prima descritti è quella degli **Arrays of Structures**, che è stata preferita rispetto alla **Structures of Arrays**. Questo perché nel nostro caso, per ciascuna computazione, è necessario considerare tutte le coordinate dei punti per il **calcolo della distanza**. Structures of Arrays avrebbe avuto senso qualora avessimo utilizzato singolarmente le varie coordinate per **scopi differenti**.

Il numero di punti ed il numero di clusters da considerare è stato definito mediante l'uso delle due macro "**numPoints**" e "**numClusters**".

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.

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. Questo è ciò che avviene nella funzione **"nearestCentroid"**, che si ricollega alla funzione **"euclideanDistance"** per il calcolo della distanza tra i punti che per convenzione è stata scelta **"Euclidea"**, ovvero per cui:

Definendo $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$, si ha che

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Tornando al funzionamento di “nearestCentroid”, 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, dove richiama la funzione **“newCentroidCoordinate”** per ciascuna coordinata. Quest’ultima funzione prende come parametri di input l’insieme dei punti, l’id del centroide su cui operare, la sua vecchia coordinata che dovrà essere modificata, ed il flag **“wichCoordinate”** che nel caso bidimensionale considererà due sole situazioni:

- wichCoordinate == 0 → Si considera la coordinata x del punto;
- Altrimenti → Si considera la coordinata y del punto;

Per la coordinata considerata si andrà a calcolare la media delle rispettive coordinate per tutti i punti presenti nel cluster considerato.

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.

Esempio di esecuzione:

Nel codice è presente una parte che consente di generare dei punti da clusterizzare e dei centroidi in varie parti del piano. Esempio di esecuzione con numPoints = 6 e numClusters = 3:

Il **Cluster 0** con Centroide = (0.5, 5.5) contiene i punti: (0.0, 6.0), (1.0, 5.0);

Il **Cluster 1** con Centroide = (2.5, 3.5) contiene i punti: (2.0, 4.0), (3.0, 3.0);

Il **Cluster 2** con Centroide = (4.5, 1.5) contiene i punti: (4.0, 2.0), (5.0, 1.0);

K-means con implementazione parallela:

La logica implementativa della versione parallela del K-means è la stessa di quella che è stata precedentemente descritta, con la differenza però dell'utilizzo dell'**API OpenMP** usato per la **parallelizzazione** di alcune sezioni di codice.

Come spiegato durante il corso, le varie sezioni parallele sono definite internamente ad una sezione del tipo:

#pragma omp parallel num_threads(nThreads) { … }

Dove la clausola **"num_threads(nThreads)"** aggiunta consente di specificare il numero di threads che dovranno essere usati quando si creano regioni parallele.

Siccome sono state specificate più di una sezione parallela in cui si fa riferimento alla variabile **"nThreads"**, questa è stata definita come una macro per poter modificare con più facilità il numero di threads applicati alle varie sezioni.

Una prima sezione di codice che è stata parallelizzata è quella relativa al processo di generazione dei punti e dei centroidi. Questo è caratterizzato da due cicli for consecutivi che vengono parallelizzati entrambi con l'ausilio della seguente direttiva:

#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.

A differenza del secondo for però (relativo alla generazione dei centroidi), la direttiva associata al primo for (inerente alla generazione dei punti) è della forma:

#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.

Osservando ciò che invece avviene internamente alla funzione K-means si nota che, la prima sezione di codice possibile da parallelizzare si trova all'interno del ciclo while che controlla la **convergenza**, ed è quella del primo ciclo for incontrato al suo interno.

Tale ciclo, come detto in precedenza, consente di stabilire per ciascun punto che viene scorso quale sia il rispettivo cluster di appartenenza. Siccome tutti i punti vengono considerati in maniera indipendente gli uni dagli altri e non essendoci dunque dipendenze particolari, questa sezione di codice risulta facilmente parallelizzabile mediante la direttiva:

#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. Tale dettaglio era stato attuato anche nella precedente fase di generazione dei dati.

L'esecuzione di questo secondo loop può essere velocizzata specificando nuovamente la direttiva:

#pragma omp for schedule(auto) private(x,y)

dove ora si nota che le variabili poste come private sono le due coordinate del centroide: x ed y.

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”).

L’ultima sezione di codice che è stata parallelizzata è quella relativa alla **stampa finale** dei punti associati a ciascun centroide. Mentre la lista dei centroidi viene stampata in modo ordinato, i punti al suo interno non è fondamentale che compaiano con un ordine preciso, dunque per velocizzare ulteriormente l’esecuzione del codice è stata applicata la direttiva:

#pragma omp parallel for schedule(auto) num_threads(nThreads)

Confronto delle prestazioni:

Vediamo cosa accade quando si effettuano esecuzioni multiple che differiscono per il numero dei dati considerati e/o per il numero di threads utilizzati.

Come primo caso si può provare a considerare il caso in cui, utilizzando **4 threads** si esegue l’algoritmo su un dataset formato da **20 punti** e **5 centroidi**. In tale situazione, confrontando l’esecuzione del codice sequenziale con quello parallelo si ottengono i seguenti risultati:

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	**CASO PARALLELO**
0.033	0.027

In questo caso, calcolando lo **speedup** si ottiene che:

$$Speedup = \frac{T_S}{T_P} = \frac{0.033}{0.027} = 1.22222$$

Quindi questo risultato ci dice che con i parametri specificati la versione parallela è 1.22222 volte più veloce rispetto a quella sequenziale.

Se si prova ad aumentare il numero di **punti** e **centroidi** rispettivamente ai valori di **3000** e **100**, mantenendo costante il **numero di threads a 4** si ottiene:

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	**CASO PARALLELO**
5.44200	2.8340

Adesso, calcolando lo **speedup** si ottiene che:

$$Speedup = \frac{T_S}{T_P} = \frac{5.44200}{2.8340} = 1.92025$$

Lo speedup risulta dunque **aumentato** rispetto a quello del primo caso ed il programma parallelo risulta essere in questo quasi il doppio più veloce de quello sequenziale.

Proviamo ad effettuare un caso ancora più particolare, quindi usiamo ora ad esempio

numPoints = 10000; numClusters = 150; sempre con **nThreads = 4;**

Con queste condizioni si ottengono i seguenti risultati:

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	**CASO PARALLELO**
14.58200	9.523

Lo speedup ottenuto ora risulta pari a:

$$Speedup = \frac{T_S}{T_P} = \frac{14.58200}{9.523} = 1.53124$$

Tutte queste esecuzioni sono state effettuate mantenendo **costante** il numero di thread utilizzati.

Una prova interessante da fare è adesso quella in cui si effettuano esecuzioni con un numero variabile di threads utilizzati durante l'esecuzione del programma. Per questa situazione si considera un numero fisso di dati e si riassumono i risultati mediante la seguente tabella:

Tempi di esecuzione con numPoints = 10000 e numClusters = 150	
Sequenziale	14.58200
nThreads = 1	14.53600
nThreads = 2	11.17300
nThreads = 3	10.13700
nThreads = 4	9.52300
nThreads = 5	9.47200
nThreads = 6	9.15100
nThreads = 7	9.03700
nThreads = 8	**8.85500**
nThreads = 9	9.36600
nThreads = 10	9.37300
nThreads = 11	9.38100

Da questa tabella è possibile vedere che il tempo di esecuzione del codice, a parità del numero di dati, **diminuisce all'aumentare del numero di threads**. Tuttavia, quando si eccede con tale numero si osserva che i tempi di esecuzione tendono nuovamente ad **allungarsi**.