Relazione progetto Parallel Computing
K-Means

Studente: Massimiliano Sirgiovanni
Matricola: 7077251
Email: massimiliano.sirgiovanni@stud.unifi.it

Tra le tracce proposte, si è optato per la realizzazione di programma che implementi l'algoritmo **K-Means**, *sequenzialmente ed in parallelo*. Il linguaggio scelto per l'implementazione è **C**, utilizzando l'API **OpenMP**, per la fase parallelizzazione.
In primis, è opportuno definire gli scopi ed il funzionamento del K-Means.

L'algoritmo K-Means

Quando si parla di K-Means, si fa riferimento ad un **algoritmo di clustering partizionato**, il cui **scopo è quello di trovare k raggruppamenti distinti**, *detti **cluster***, a partire **da un insieme di punti**.
Ad ognuno dei k cluster, è associato un punto, detto **centroide**, che indica il punto centrale all'interno del cluster. L'algoritmo K-Means sfrutta il concetto di centroide per poter creare i raggruppamenti richiesti.
In generale, prima dell'esecuzione dell'algoritmo **bisogna specificare il valore di k**, che corrisponderà al **numero di centroidi** considerati *(ed al numero di cluster che si vogliono ottenere)*.
Una volta scelti i centroidi, **si calcolano le distanze tra i punti in input ed i centroidi scelti**. Per ogni punto, prima si seleziona il centroide a distanza minima. Una volta trovato, si inserisce il punto nel cluster corrispondente.
Conclusa la fase di assegnazione, **si ricalcolano i centroidi calcolando la media** di tutti gli elementi appartenenti ai rispettivi cluster, *trovando così l'effettivo punto centrale*.
L'algoritmo termina quando tutti i k centroidi non cambiano le loro coordinate.
L'output dell'algoritmo sarà, appunto, composto da **k cluster**, contenenti i punti dati in input all'algoritmo.
Per quanto concerne **la scelta dei centroidi**, solitamente, **viene effettuata casualmente**, ma vi sono anche altre possibilità quali l'utilizzo di un algoritmo di clustering gerarchico o di un bisecting K-Means *(i quali non verranno approfonditi in questa sede)*.
Un algoritmo di clustering può, naturalmente, anche produrre **cluster vuoti**. Per gli obiettivi prefissati tali cluster non creano alcun tipo di problema, se non per l'aggiunta di alcuni controlli, ma sono di scarsa rilevanza come risultato. *In effetti, solitamente, questi vengono eliminati tramite tecniche di post processing*.

Implementazione sequenziale

La prima sfida per implementare l'algoritmo riguarda la **rappresentazione dei punti e dei cluster** nel programma. Innanzitutto, è da sottolineare che **sono stati utilizzati punti in due sole dimensioni**.
Sono state prese in considerazione diverse opzioni, specie per i cluster. Infine, si è optato per la **costruzione di una struttura che rappresentasse i punti**, contenente **due attributi double ed un attributo int**. I primi due rappresentano **le coordinate del punto**, mentre l'ultimo conterrà l'**id del cluster**, una volta assegnato. Era stata vagliata l'ipotesi di costruire una struttura anche per i cluster, ma, dopo alcuni tentativi, è stato evidente che questa tecnica comportasse un inutile spreco di memoria.
Per quanto concerne i **centroidi**, anch'essi sono punti dunque non è stato necessario creare altre strutture. Per comodità, si è poi creato un metodo, ***initializePoint(double a, double b)***, il cui compito è creare un punto con coordinate *(a, b)* e con valore dell'attributo cluster inizializzato a *-1*.
Una volta creati gli strumenti per realizzare gli elementi necessari all'applicazione del K-Means, si può procedere all'implementazione dell'algoritmo vero e proprio.
In rappresentanza dell'algoritmo, è stata creata la funzione ***KMeans(point pts[], point centroids[])***. Gli argomenti passati sono l'**insieme dei punti** su cui applicare l'algoritmo e l'**insieme dei centroidi** scelti per l'esecuzione.
Le dimensioni dei due insiemi, sono state inserite come **Macro**, fuori dal main *(n per i punti e k per i centroidi)*:

```
#define n 1000 //Number of points
#define k 300 //Number of centroids
```

L'esecuzione dell'algoritmo viene, quasi totalmente, **svolta all'interno del seguente ciclo while**:

```
while(end == 0){
        end = 1;
        //Applicazione del K-Means
}
```

La variabile *end* è usata per indicare se l'algoritmo ha terminato le sue iterazioni. **Le viene assegnato valore uno** subito dopo l'accesso al ciclo, però potrà essergli assegnato nuovamente valore zero durante l'esecuzione.
Entrati nel ciclo, **è necessario assegnare ad ogni punto un cluster di appartenenza**, modificando l'attributo cluster della struttura.
Col fine di **applicare questa procedura ad ogni punto nell'insieme**, è stato creato un ciclo for, con n iterazioni. Ad ogni iterazione, viene ricercato il centroide *j* più vicino al punto *i*, in modo tale da assegnargli il cluster di indice j.
Per ricercare il centroide più vicino è stato realizzato il metodo ***MinCentroid(point Point, point centroids[])***. Questo **confronta ogni centroide col punto passatogli**,

calcolandone la distanza e **restituendo l'id del centroide più vicino** *(l'id corrisponde alla posizione all'interno del vettore dei centroidi)*.
Per quanto concerne la ricerca del minimo, il codice è piuttosto elementare.
Più interessante è la **definizione di distanza tra i punti** e la sua realizzazione nel codice. Infatti, si è fatto riferimento alla **distanza Euclidea**, la cui formula è la seguente:

$$d(p,\ q)\ =\ \sqrt{\sum_{i=1}^{n}(p(i)\ -\ q(i))^2}$$

Dove p e q sono due punti e n è il numero totale di dimensioni (richiamando $p(i)$ si fa riferimento alla coordinata i-esima di p).
Nel codice, il calcolo della distanza Euclidea viene effettuato da un altro metodo, considerando punti a due dimensioni.
Al metodo *EuclideanDistance(point a, point b)* vengono passati come argomenti due punti, e restituire la distanza calcolata in questo modo:

sqrt(pow((a.x - b.x),2) + pow((a.y - b.y), 2))

Tornando al metodo precedente, una volta calcolate le distanze Euclidee tra il punto ed i centroidi, viene restituito*, come anticipato,* l'indice del centroide quella minima.
Questo indice verrà poi assegnato al punto, all'interno dell'attributo cluster.
Una volta **assegnato un cluster a tutti i punti**, bisogna **verificare se è necessario effettuare una nuova iterazione**. Come visto in precedenza, per farlo bisogna **calcolare la media dei punti di un cluster**, per ogni dimensione. Se questi valori corrispondono a quelli dei centroidi, della precedente iterazione, allora l'algoritmo può terminare.
D'altra parte, a*nche se la condizione non fosse soddisfatta per un solo cluster*, vengono sostituiti i centroidi con i valori delle medie e si dovrà eseguire una nuova iterazione dell'algoritmo.
Questo viene fatto nel metodo KMeans, tramite l'utilizzo di un secondo ciclo for, con k iterazioni differenti.

```
for(j = 0; j < k; j++){
        x = recomputesCentroidX(j, pts, centroids[j].x);
        y = recomputesCentroidY(j, pts, centroids[j].y);

        if(x != centroids[j].x | y != centroids[j].y){
           end = 0;
           centroids[j].x = x;
           centroids[j].y = y;
        }

    }
```

Nel ciclo riportato sopra, per ogni centroide, **vengono ricalcolate le due dimensioni**, **qualora dovessero differire** dalle due precedenti, verrebbe impostata la variabile end a zero e **si assegnerebbero i nuovi valori ai centroidi**.
Per ricalcolare i centroidi sull'insieme di punti, si utilizza la funzione ***recomputesCentroidX(j, pts, centroids[j].x)****(la funzione recomputesCentroidY(j, pts, centroids[j].y) ha il medesimo funzionamento, ma per la coordinata y)*.
La funzione c**onsidera tutti i punti appartenenti a quel cluster, somma i valori delle loro ascisse e divide questo valore per il numero di elementi**. Viene, dunque, restituito questo valore, **purché il cluster non sia vuoto**.
In questo secondo caso viene restituito un **valore di default**, passato come argomento ed, *in questo caso*, **corrispondente al vecchio valore del centroide**.
A questo punto, se durante l'ultima iterazione del ciclo while sono stati modificati i centroidi, allora è necessario effettuare un'altra iterazione, in caso contrario è possibile uscire dal ciclo e l'algoritmo è terminato.
L'ultimo passaggio che rimane da svolgere è la **stampa a video dei cluster**.
Viene eseguita nel metodo *KMeans*, dopo l'esecuzione il ciclo while, applicando, per ogni cluster, un'altro metodo per la stampa *(tramite un altro ciclo for di lunghezza k)*.
Un esempio dell'output del programma è visibile nella tabella sottostante *(Tabella 1)*.

Cluster ID	Centroide	Punti			
0	(1.5, 2.75)	(0, 7)	(1, 6)	(2, 5)	(3, 4)
1	(4.5, 1.25)	(4, 3)	(5, 2)		
2	(6, 0.5)	(6, 1)			

Tabella 1: Output esecuzione sequenziale del K-Means (n:7, k:3)

Nel caso specifico, è stato realizzato un sistema di **generazione automatica dei punti e dei centroidi**, che, *a parità di dimensioni*, **restituisce gli stessi punti per ogni esecuzione**. Per fare ciò sono stati realizzati due cicli for, uno di dimensione n e l'altro k, che assegnano valori dipendenti da n e k alle coordinate dei punti.

Implementazione parallela

Il programma sopra implementato, *in caso di valori per n e k grandi*, **impiega parecchio tempo per terminare la sua esecuzione**. Di conseguenza, potrebbe essere ottimale l'utilizzo di **tecniche di parallelizzazione** che possano accelerare i tempi dell'algoritmo di clustering.
Per la realizzazione di un programma parallelo, come accennato in precedenza, è stato usato l'API **OpenMP**.

Ottenuti gli strumenti necessari, si deve **analizzare il codice per identificare le parti che è possibile parallelizzare** e per modificarle di conseguenza.
Il punto di partenza è dunque il metodo ***KMeans(point pts[], point centroids[])***, nel quale sono presenti un **ciclo while(end==0)** e **tre cicli for**. Il ciclo while, nella sua interezza, non è parallelizzabile. All'interno del corpo del ciclo while viene effettuata l'assegnazione del valore uno alla variabile end e vengono eseguiti due cicli for.
Il **primo ciclo for**, *come visto in precedenza*, **permette l'assegnazione di un cluster ad ognuno dei punti**. Ogni iterazione è separata rispetto alle altre, dato che considera un diverso punto della lista. Dunque, **ogni iterazione del ciclo potrebbe essere effettuata in un qualsiasi ordine ed in qualunque momento**, purché venga assegnato a tutti i punti un cluster, prima di passare al ciclo successivo.
Queste caratteristiche rendono questa porzione di codice perfetta per la parallelizzazione. Si utilizzerà dunque una **direttiva di OpenMP**:

#pragma omp parallel for

Il funzionamento di questa direttiva è quello di **gestire le iterazioni di un ciclo for tramite l'utilizzo di diversi thread**, in modo da velocizzare l'esecuzione del ciclo.
Alla direttiva, sono state specificate **due clausole**: **schedule** e **private**.
La prima consente di **specificare l'ordine di assegnazione delle iterazioni ai thread**. Dato che non vi sono particolari esigenze per la fase di scheduling, la gestione di questa è stata delegata al compiler, assegnando alla clausola schedule l'opzione ***auto***.
Per quanto riguarda la **clausola private**, d'altra parte, questa permette di **costruire una copia in locale**, *per ogni thread*, **di una variabile**. Dato che la variabile *nearestCentroid* deve assumere valore diverso ad ogni iterazione, e tale valore viene utilizzato solo all'interno dell'iterazione stessa, porre la variabile come privata è la soluzione migliore per **evitare problemi di race condition**.
La variabile nearestCentroid, nel ciclo, assumerà un valore datogli da una funzione esterna, *MinCentroid(point Point, point centroids[])*. Anche all'interno di questa funzione è presente un ciclo che restituisce un valore minimo da un insieme di centroidi, ma non può essere parallelizzato. Questo perché ogni iterazione contribuisce alla ricerca di un unico valore utilizzando variabili in comune. Questo genererebbe problemi di race condition nel codice, facendo sì che la funzione restituisca un valore errato.
Come soluzione si potrebbe inserire una sezione critica attorno all'intero corpo del ciclo for, ma in questo modo non si otterrebbe alcun vantaggio rispetto ad un'esecuzione sequenziale.
Tornando al ciclo precedente, nel metodo KMeans, l'ultima operazione presente nel ciclo consiste in un **accesso in scrittura** alla variabile cluster dei punti.
Dato, però, che per ogni iterazione, viene modificato l'attributo cluster di un solo punto, che non verrà mai selezionato nuovamente, non vi è problema di imbattersi in race condition.

Ottenuto un primo ciclo parallelizzato, si può procedere ad **esaminare il secondo ciclo for**, *contenuto nel while(end==0)*.
In questo ciclo, **si ricalcolano i valori delle coordinate dei centroidi e si verifica se è necessario effettuare un'altra iterazione** del ciclo while.
Anche questo ciclo, di lunghezza k, è parallelizzabile. Si utilizza la stessa direttiva vista in precedenza, **impostando come private due variabili x ed y**, *di tipo double*.
A queste due variabili verrà assegnato il valore di una funzione, il cui corpo non è parallelizzabile a causa di problemi di race condition ed il cui risultato sarà utile solo nell'iterazione corrente.
Verrà poi effettuato un controllo, tramite istruzione if, per verificare se il centroide considerato è cambiato rispetto alla precedente iterazione:

```
if (x != centroids[j].x | y != centroids[j].y)
```

Se così fosse, verrebbero eseguite le seguenti operazioni di scrittura:

```
end = 0;
centroids[j].x = x;
centroids[j].y = y;
```

Le ultime due modificano un elemento dell'array centroids al quale non verranno fatti altri accessi in altre iterazioni del ciclo, dunque non si hanno problemi di race codition. La **prima istruzione** *(end = 0;)* è più particolare. Infatti, sebbene un thread potrebbe modificare la variabile in ordine diverso rispetto al caso sequenziale, non è rilevante per gli scopi del programma. L'importante è che la variabile venga modificata almeno una volta all'interno del ciclo, se necessario, non è importante quale delle iterazioni la modifica per prima. Se tutte le iterazioni del ciclo superassero il controllo dell'if o, d'altra parte, fosse una sola a farlo, il valore assunto dalla variabile end sarebbe sempre e comunque zero.
Dato che i due cicli for parallelizzati sono consecutivi, per **migliorare le prestazioni**, si può **circondare questa porzione di codice con un'altra direttiva**, modificando le direttive dei for paralleli, in questo modo:

```
#pragma omp parallel
    {
        //Iteration on all the points considered
#pragma omp for schedule(auto) private(nearestCentroid)
        for (i = 0; i < n; i++) {
                // For cicle body
        }
        //Verify if centroid don't change
#pragma omp for schedule(auto) private(x, y)
        for (j = 0; j < k; j++) {
                // For cicle body
    }
  };    //end omp parallel
```

L’effetto del codice non cambia, ma in questo modo si evita di generare e chiudere i thread al termine di ogni ciclo.
Superato il ciclo while(end==0), **l’unica operazione rimasta è la stampa**.
Il ciclo, nel metodo *KMeans,* adibito alla stampa è preferibile **non parallelizzarlo**, poiché restituirebbe un output disordinato e di difficile comprensione. Tuttavia, all’interno del metodo ***printClusters()***, *invocato dal ciclo for sopracitato*, viene utilizzato un altro ciclo for, *di lunghezza n*, per la stampa dei punti nel cluster.
Dato che, per la comprensione dell’output, non è necessario mantenere un ordine particolare per la stampa dei punti in un cluster, questo ciclo può essere facilmente parallelizzabile. Gli unici effetti sulla stampa, sono osservabili confrontando la *Tabella 1*, vista sopra, con la *Tabella 2*, che mostra l’output di un esecuzione parallela.

Cluster ID	Centroide	Punti			
0	(1.5, 2.75)	(2, 5)	(3, 4)	(0, 7)	(1, 6)
1	(4.5, 1.25)	(4, 3)	(5, 2)		
2	(6, 0.5)	(6, 1)			

Tabella 2: Output esecuzione parallela del K-Means (n:7, k:3)

Come anticipato, l’unica differenza con l’esecuzione sequenziale *(Tabella 1)* è l’ordine di stampa dei punti nel cluster *(in questo caso per il cluster 0)*. Ovviamente, solo l’ordine dei punti è differente, non i punti stessi.
Per il caso specifico, *data la natura della creazione dei punti e dei centroidi*, si è pensato di **parallelizzare anche la procedura di inizializzazione dei dati**.
Dunque, i cicli for adibiti all’inizializzazione dei punti, *accennati in precedenza*, sono stati parallelizzati, con le direttive di OpenMP, in questo modo:

```
#pragma omp parallel
   {
#pragma omp for schedule(auto) nowait
      for (p = 0; p < n; p++) {
         // Points Creation
      }

#pragma omp for schedule(auto)
      for (q = 0; q < k; q++) {
         // Centroids creation
      }
   };
```

Le procedure usate ed i ragionamenti fatti sono molto simili a quelli visti per gli altri cicli, tuttavia, vi è l'introduzione di una nuova clausola per il primo ciclo for, **nowait**.
Aggiungere la clausola nowait ad un ciclo for, significa **rimuovere la barriera implicita** presente alla fine di ogni ciclo for parallelo. Ciò significa che **se uno o più thread terminano l'esecuzione del ciclo**, nonostante non tutti i thread abbiano terminato, **possono procedere ad eseguire le successive linee di codice**.
In questo caso, dato che il secondo ciclo for non utilizza alcuna informazione che venga modificata nel primo, si può procedere senza problemi ad eseguire le istruzioni per la creazione dei centroidi.

Testing e valutazione delle performance

Viste le scelte implementative e le differenze tra la versione sequenziale e parallela dell'algoritmo, la sintesi è rappresentata dal **confronto tra le due versioni**, principalmente in termini di performance.
Si è già accennato come il modello di creazione automatica dei punti permetta generare gli stessi punti, a parità di n e k. Dunque, i test metteranno a confronto i due programmi con il medesimo input, in modo che i risultati possano essere i più accurati possibile.
Un primo test presentato considera valori di **n e k molto bassi**. Avremo:

- *n = 15*
- *k = 5*

Una volta eseguiti i due programmi si possono mettere gli output a confronto.
Nel caso sequenziale, **il tempo impiegato per l'esecuzione è ancora estremamente breve**, essendo i dati in input pochi, ma si potrà immediatamente osservare un **miglioramento**, in performance temporali, **nell'esecuzione effettuata in parallelo** *(Tabella 3)*.

	Sequenziale	Parallela
Tempo di esecuzione	0.007907	0.002472

Tabella 3: Confronto tempi di esecuzione (n:15, k:5)

Nonostante i valori estremamente piccoli, si può utilizzare la formula dello **speedup** per verificare quanto efficace è stata la procedura di parallelizzazione, in questo caso.

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{0.0079}{0.0024} \simeq 3,3$$

In questo caso, **l'esecuzione in parallelo è tre volte più veloce rispetto a quella sequenziale**. Tuttavia, a questo livello i tempi di esecuzione sono irrisori.
Più interessante è affrontare un **caso di medie dimensioni**. Per questo nuovo test, verranno assegnati i seguenti valori:

- *n = 1000*
- *k = 100*

Osservando l'output, *non completo per mancanza di spazio*, si osserverà come anche in questo caso, si ottenga uno **speedup circa pari a tre**. Dunque, **il codice in parallelo eseguirà tre volte più velocemente le stesse operazioni** rispetto a quello sequenziale *(Tabella 4)*.

	Sequenziale	Parallela
Tempo di esecuzione	3.070847	0.981001

Tabella 3: Confronto tempi di esecuzione (n:1000, k:100)

Di default, l'elaboratore usato istanzia **quattro thread** per l'esecuzione del codice. Per osservare in maggior dettaglio gli effetti della programmazione parallela si può **impostare manualmente il numero di thread** che si desidera utilizzare, tramite la semplice istruzione:

omp_set_num_threads(T);

Dove **T rappresente il numero di thread** da utilizzare. Quest'istruzione va posta prima di una o più regioni parallele consecutive.
Si può quindi, riprendendo il test appena svolto, provare a diminuire il numero di thread, verificando come il tempo di esecuzione cambi al diminuire di tale numero. Effettuando un'esecuzione con T pari a quattro, si noterà come i valori ottenuti siano pressoché identici a quelli visti nella *Tabella 3*.
Provando ad assegnare valore **tre** alla variabile T, i risultati ottenuti sono i seguenti:

	Sequenziale	Parallela
Tempo di esecuzione	3.070847	1.173633

Tabella 4: Confronto tempi di esecuzione con tre Thread (n:1000, k:100)

Nonostante il guadagno, in termini di tempo, sia ancora parecchio, è evidente come **usando un numero di thread inferiori lo speedup diminuisca**.

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{3.07}{1.17} \simeq 2.62$$

Abbassando ulteriormente il numero di thread usati, le prestazioni si avvicinano sempre più al caso sequenziale.
Impostando T pari a **due**, si ottengono i risultati osservabili nella *Tabella 5*.

	Sequenziale	Parallela
Tempo di esecuzione	3.070847	1.631923

Tabella 5: Confronto tempi di esecuzione con due Thread (n:1000, k:100)

Proseguendo col calcolo dello speedup:

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{3.07}{1.63} \simeq 1,88$$

Abbassando ulteriormente il numero di thread, ci si aspetterebbe un risultato che restituisca uno **speedup molto vicino al valore uno**.
In effetti, come si vede dalla tabella sottostante, i risultati confermano l'ipotesi:

	Sequenziale	Parallela
Tempo di esecuzione	3.070847	2.985938

Tabella 6: Confronto tempi di esecuzione con un Thread (n:1000, k:100)

Che in termini di speedup si ottiene un valore pari a ***1.03***, circa.

Le prove che si potrebbero fare sono innumerevoli, basta utilizzare le diverse combinazioni dei valori di n e k. In generale, tuttavia, **ha poco senso usare un valore di k che sia maggiore del numero di punti**, dato che l'obiettivo è quello di trovare k raggruppamenti di punti. Tuttavia, ipoteticamente, anche questa eventualità è gestita dal programma *(sono da aspettarsi diversi cluster vuoti)*.
Per concludere con la fase di testing, avendo mostrato un caso di dimensioni ridotte ed uno di medie dimensioni, è opportuno provare ad utilizzare **valori di n e k molto grandi**:

- *n = 5000*
- *k = 300*

Osservando i risultati si vede facilmente come, anche in questo caso, l'esecuzione nel caso parallelo risulta essere, circa, tre volte più veloce rispetto al caso sequenziale *(Tabella 7)*.

	Sequenziale	Parallela
Tempo di esecuzione	132.842004	37.332709

Tabella 7: Confronto tempi di esecuzione con quattro Thread (n:5000, k:300)

Come anticipato, usando la formula dello speedup si ottiene un valore circa pari a tre.

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{132.84}{37.33} \simeq 3,55$$

Volendo provare ad abbassare il numero di thread, portandolo da quattro a due, i risultati ottenuti sono i seguenti.

	Sequenziale	Parallela
Tempo di esecuzione	132.842004	54.826176

Tabella 8: Confronto tempi di esecuzione con due Thread (n:5000, k:300)

In questo caso si avrà il seguente speedup:

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{132.84}{54.83} \simeq 2.42$$

Dai risultati osservati, in generale, si può affermare che all'aumentare del tempo di esecuzione dell'algoritmo nel caso sequenziale, *ed all'aumentare del numero di thread*, risulta sempre più evidente il **vantaggio ottenibile tramite l'utilizzo della programmazione parallela**.