Programmazione di Sistemi Multicore : K-Mean Clustering

Andras. B. Arancio, Eugenio Schiera

January 2023

1 Introduzione

Gli obiettivi fondamentali che ci siamo posti nella realizzazione del nostro programma sono stati quelli di scrivere un programma altamente efficiente, anche nella sua versione sequenziale, e quello di effettuare una parallellizzazione più accurata e minuziosa possibile, che renda il programma scalabile all'aumentare dei thread. I risultati ottenuti sono stati soddisfacenti, e verranno illustrati nel corso di questa relazione.

Figure 1: Una immagine clusterizzata con il nostro programma: a sx. l'originale, al centro clusterizzata con 5 cluster, a dx. con 10 cluster

2 L'architettura del programma

Al fine di adempiere agli obbiettivi di efficienza che ci eravamo preposti, abbiamo deciso di utilizzare strutture dati efficienti; prevale infatti l'utilizzo degli array, a dispetto di soluzioni come le linked list, più lente di per sè e dalle operazioni difficilmente scalabili nella parallelizzazione. Il risultato di questa scelta architetturale è stato quello di ottenere un programma in cui la maggior parte dei loop è vettorizzabile.
Il programma è composto da quattro funzioni principali: una deputata al recupero delle coordinate dell'input da un file di testo, due responsabili dell'allocazione e del liberamento della memoria utilizzata e un'altra che esegue l'algoritmo vero e proprio.

2.1 Argomenti necessari all'esecuzione

Al lancio del programma sono richiesti i seguenti argomenti: il nome del file di testo contenente le coordinate, il numero M di punti in input, il numero N di dimensioni dei punti, il numero K di cluster in cui si vuole suddividere i punti e, nel caso in cui si stia eseguendo una delle versioni parallele, il numero di thread con cui si desidera eseguire.

```
./main miei_punti.txt 1000000 500 12 8
```

2.2 Strutture dati e funzionamento del programma

Operiamo su un insieme di M punti, che verranno suddivisi in K cluster; il punto medio di ogni cluster è detto centroide. Le strutture che utilizza il nostro algoritmo sono le seguenti:

```
struct point {
    int dimensions;
    double* coordinates;
    int cluster;
    double distance_from_centroid;
};
```

usato per immagazzinare le informazioni riguardanti un punto. In particolare, l'appartenenza al cluster è identificata tramite un dato primitivo `int`, in questo modo i cluster sono definiti a partire dai punti, facendo a meno di strutture come linked list per le operazioni riguardanti i cluster. L'algoritmo converge quando ogni punto appartiene allo stesso cluster a cui apparteneva nell'iterazione precedente.

```
struct centroid {
    struct point centroid;
    struct point accumulator;
    int points_in_cluster;
};
```

Rappresenta il centroide di un cluster. Ad ogni iterazione, la somma delle coordinate dei punti appartenenti al cluster al momento dell'attuale iterazione viene accumulata in un accumulatore e ne viene successivamente fatta la media, al fine di calcolare la posizione del nuovo centroide per l'iterazione successiva.

```
struct space {
    struct point* points;
    struct centroid* centroids;
    int number_of_points;
    int dimensions;
    int clusters;
};
```

Rappresenta l'insieme composto dai punti su cui è richiesta l'operazione di clustering. Contiene i punti dell'insieme, i centroidi dei cluster, e i dati che specificano le loro dimensioni.

3 Tecniche di parallelizzazione

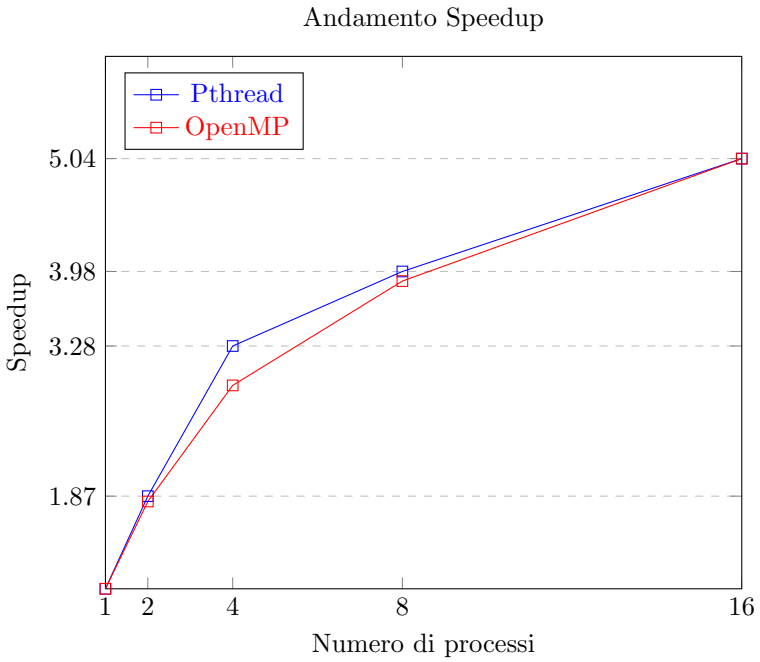
Le due API che abbiamo deciso di utilizzare per la parallelizzazione del nostro programma sono state *Pthread* e *OpenMP*. Ciascun thread lavora su una propria porzione dell'insieme, si può dunque affermare che, secondo la tassonomia di Flynn, il nostro è un programma *Single Instruction Multiple Data*. I controlli di convergenza vengono ovviamente effettuati sull'insieme per intero.
Ci siamo concentrati sulla parallelizzazione dell'algoritmo principale, quello di *K-Mean Clustering*, all'interno della omonima funzione. Sono stati diversi i punti in cui ci siamo soffermati ad eseguire dei test che ci rendessero chiara quale fosse la strategia migliore da applicare, ogni scelta implementativa è stata presa a seguito di una attenta analisi.

M = 600'000, N = 50, K = 12					
Programma	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
Sequenziale	1489s				
Pthread		795s	453s	374s	295s
OMP		815s	510s	382s	295s

Table 1: Test effettuati su processore AMD Ryzen 3800X, 8 core, 16CPU

M = 600'000, N = 50, K = 12				
Programma	2 Thread	4 Thread	8 Thread	16 Thread
Speedup Pthread	1,87	3,28	3,98	5,04
Speedup OMP	1,82	2,91	3,89	5,04
Efficienza Pthread	0,93	0,82	0,49	0,31
Efficienza OMP	0,91	0,72	0,48	0,31

Table 2: Test effettuati su processore AMD Ryzen 3800X, 8 core, 16CPU



All'aumentare dei processi diminuisce lo speedup, come si evince dalla curva logaritmica. Con dimensioni del problema maggiori ed una conseguente decrescita, in proporzione, dell'overhead è prevedibile un miglioramento ulteriore dello speedup ed una maggiore efficienza al crescere del numero dei thread.

3.1 Parallelizzazione

Nella realizzazione della versione *Pthread* del nostro programma siamo stati abbastanza minuziosi da riuscire a replicare i modelli di parallelizzazione applicati dalle direttive di *OpenMP*, anche nei casi più ostici. La Figura 2 ne descrive un esempio: il `for` annidato in `Sequenziale::kmean::74`, risolto nella versione *OMP* con una direttiva `omp for collapse(2)`, viene

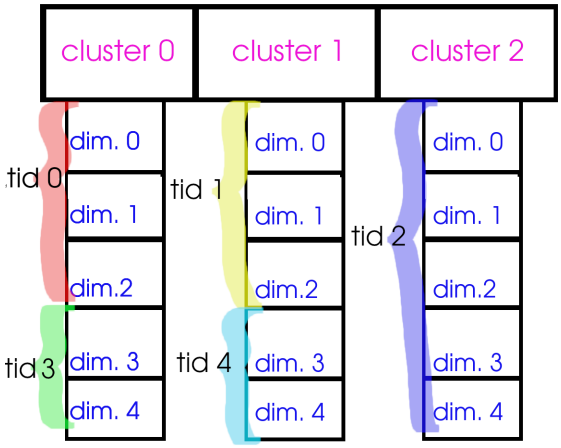


Figure 2: Abbiamo assegnato a ciascun processo un sottoinsieme di dimensioni

suddiviso anche nella versine *Pthread* in una maniera tale da non lasciare mai i thread "a riposo". Il risultato è una versione

Pthread più veloce di quella *OMP*, essendo libera dall'overhead dovuto alle direttive `omp for`.

3.2 Limitazioni riscontrate

Un caso particolare sul quale ci siamo soffermati particolarmente a lungo è stata la seguente porzione di codice sequenziale:

```
1.     for (int i = 0; i < M; i++) {
2.         centroids[points[i].cluster].points_in_cluster++;
3.         for(int coord = 0; coord < N, coord++) {
4.             centroids[points[i].cluster].accumulator.coordinates[coord]
5.             += points[i].coordinates[coord];
6.    }
```

in cui viene calcolato il numero di punti di ciascun cluster durante l'iterazione attuale, e sommate nell'accumulatore del centroide le coordinate dei punti appartenenti al suo cluster.
Ci sono in questo codice due *critical section*: riga `2` e `4`; al fine di raggirare la somma sull'accumulatore(riga `4`) abbiamo separato le due istruzioni, e parallelizzato la seconda vettorizzando solo il for interno:

```
      for (int i = 0; i < M; i++) {
  #         pragma omp for nowait
            for (int coord = 0; coord < N; coord++) {
            centroids[points[i].cluster].accumulator.coordinates[coord]
            += points[i].coordinates[coord];
            }
      }
```

La prima *critical section*(riga `2`), sull'aumento del contatore dei punti in cluster, è inevitabile. La prima idea che ci è venuta in mente è stata quella di utilizzare una `omp atomic`:

```
  #   pragma omp for
      for (int i = 0; i < M; i++) {
  #   pragma omp atomic
          centroids[points[i].cluster].points_in_cluster++;
      }
```

Abbiamo però notato che adottare questa soluzione risultava in un programma più lento rispetto ad una versione in cui questa porzione fosse sequenziale: vista la bassa mole computazionale all'interno del ciclo, l'overhead delle direttive `omp for` e `atomic` era maggiore rispetto al tempo di computazione in sequenziale.
Alla fine abbiamo trovato una soluzione nell'utilizzo della direttiva `reduction`: abbiamo creato un array che conservasse i contatori dei punti nei cluster, rifacendoci al modello *struct of arrays*, e applicato a questo la direttiva di riduzione.

```
   #   pragma omp for reduction(+ : points_in_cluster[:K])
       for (int i = 0; i < M; i++){
           points_in_cluster[points[i].cluster]++;
       }
```

Nonostante gli incrementi nelle prestazioni non siano significativi, abbiamo comunque optato per la scelta della reduction per conseguire comunque una parallelizzazione quanto più completa possibile.