Progetto di Parallel Computing 2021-2022

Gabriele Boddi
gabriele.boddi@stud.unifi.it

Abstract

In questo elaborato verrà mostrata l'implementazione dell'algoritmo di "Pattern Recognition", una metodologia che consente di ricercare i valori di similarità tra una query di ricerca ed un insieme di dati.
Il suo sviluppo verrà mostrato in versione sequenziale e versione parallela tramite l'utilizzo di C++ e OpenMP.

1. Introduzione

Un algoritmo di **Pattern Recognition** consente il riconoscimento di un certo insieme di dati grezzi rispetto a degli specifici parametri di ricerca (o detti anche *query di ricerca*).
Questo tipo di algoritmo viene utilizzato in numerose branche dell'informatica come *analisi dei dati*, *bioinformatica* e *machine learning* [1].
La sua implementazione richiede l'utilizzo di una misura matematica che prende il nome di **SAD** (*Sum of Absolute Differences*).
Con essa è possibile dividere un insieme di dati (o *dataset*) in *Regioni* e controllare quale di esse è più *simile* alla query di ricerca.
Maggiori dettagli su tale misura saranno descritti di seguito [2].

2. Descrizione della misura SAD

Supponiamo di voler confrontare un *template* con un'*immagine*, definiti rispettivamente dai seguenti valori numerici:

```
Template    Search image
 2 5 5       2 7 5 8 6
 4 0 7       1 7 4 2 7
 7 5 9       8 4 6 8 5
```

Il numero di regioni in cui andremo a suddividere i controlli sarà data dalla seguente formula:

$$regions = cols - querycols + 1 \qquad (1)$$

dove *cols* sono le colonne della matrice che rappresenta l'immagine (ossia 5), mentre *querycols* sono le colonne della matrice che rappresenta il template (ossia 3).
Il controllo sarà dunque diviso in 3 regioni, che rappresentano rispettivamente il lato sinistro dell'immagine, il suo centro ed il suo lato destro.

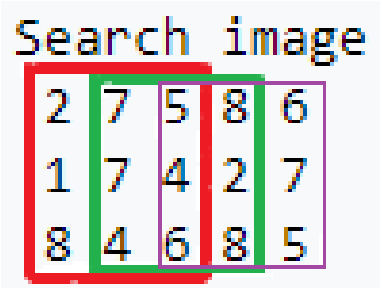

Figura 1. Suddivisione dell'immagine in regioni.

A questo punto, per ogni regione si comincia ad effettuare delle differenze, in valore assoluto, di ogni loro elemento con quelli del template.
Si otterranno così i seguenti risultati:

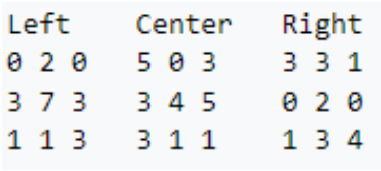

Figura 2. Risultati delle differenze in valore assoluto.

Infine, effettuando la somma tra gli elementi di ogni regione, si ottiene che i valori *SAD* sono rispettivamente 20, 25 e 17.
Si può quindi asserire che la sezione destra dell'immagine è quella più simile al template, poiché corrisponde al valore più piccolo individuato.

3. Descrizione dell'Algoritmo

Il codice implementato si divide in 2 classi principali: la classe *Pr* (*Pr_parallel* per la versione parallela) dove risiedono le funzioni principali dell'algoritmo e la classe *GetElements* che permette di convertire i valori numerici dei Dataset presi da file in *matrici*, così come per la query di ricerca.
Le funzioni di quest'ultima vengono richiamate direttamente nel costruttore della classe di controllo.

```
GetElements g;

cout << "Loading Dataset..." << endl;
elements = g.get_elements_from_file(rows,cols,"blogData_train.csv");
cout << "Dataset Loaded." << endl << endl;


cout << "Loading Query..." << endl;
query = g.get_random_query(queryrows,querycols,rows,cols,regions);
cout << "Query Loaded." << endl << endl;

if(querycols>=cols || queryrows!=rows) throw std::runtime_error("Error: Query parameters are incorrect!");

if(regions == -1) regions = cols-querycols+1;
if(regions<2 || regions>cols) throw std::runtime_error("Error: the value of the Regions is incorrect!");
```

Figura 3. Costruttore della classe di controllo.

Una volta caricato il Dataset di interesse, se l'utente ha deciso di utilizzare una query casuale, gli verrà chiesto di inserire in Input il numero delle *Regioni* di suddivisione del Dataset, per poi procedere con il caricamento della query.
Caricata anche quest'ultima, il programma inizia a calcolare la metrica di *SAD* su tutte le regioni, salvando i risultati all'interno di un *array*, per poi restituire l'indice del valore più piccolo trovato che indicherà la regione più simile alla query inserita.

4. Dettagli di implementazione

Di seguito verranno mostrati i dettagli di implementazione di entrambe le classi.
Nota Importante: *dalla versione sequenziale a quella parallela cambia soltanto la classe principale dell'algoritmo, mentre quella per ottenere in input il Dataset e la query resta invariata.*
Nel calcolo delle prestazioni e dello Speedup verrà tenuto conto solo dei metodi principali della classe Pr (e Pr_parallel).

4.1. Caricamento del Dataset e della query

All'interno della classe *GetElements* è possibile trovare le due funzioni **get_elements_from_file** e **get_random_query**: con la prima è possibile caricare un dataset, così come la query stessa, direttamente da dei file.
Il contenuto verrà analizzato e ripulito da eventuali caratteri non numerici attraverso l'uso delle *regex expressions*, per poi costruire le matrici di double che serviranno durante lo svolgimento del programma.
La seconda funzione, invece, consente di caricare una query casuale direttamente dal programma.
Nota Importante: *L'utente è libero di scegliere come effettuare il caricamento dei dati, così come decidere se utilizzare una query casuale o no: insieme ai programmi sono presenti i file* ***example_query.data*** *ed* ***example_dataset.data*** *che mostrano degli esempi di strutture dati che possono essere caricate correttamente nel programma qualora si scelga la strada del caricamento esterno.*
I numeri devono essere separati da spazi, che indicheranno il numero di colonne delle matrici, e ad ogni nuova riga (anche dopo l'ultima) dovrà essere effettuato un "ritorno a capo" così che il programma possa conteggiare correttamente le righe delle matrici.
Se l'utente carica una query da file essa dovrà avere il numero di colonne minore di quelle del dataset, mentre il numero di righe dovrà essere

invece identico: in caso contrario, il programma solleverà un'eccezione a runtime.
Nel caso in cui all'utente venga chiesto di inserire il numero delle regioni in input, esso dovrà essere maggiore di 2 (altrimenti non si effettua una vera suddivisione) e minore o uguale al numero di colonne del dataset.
Anche in questo caso, se non vengono rispettate tali proprietà viene generata un'eccezione a runtime.
Infine, se l'utente utilizza delle strutture su file con caratteri non numerici, dovrà fare attenzione a quali di essi sono presenti ed eventualmente a modificare le regex expression per una pulizia corretta (attualmente esse consentono di rimuovere i caratteri alfabetici preceduti o seguiti da uno spazio).

4.2. Classe di controllo (Versione Sequenziale)

Il programma utilizza 4 funzioni principali:

1. **calculate_sad()**: funzione che fa partire il controllo sul dataset.
   Essa richiama la funzione che sarà descritta di seguito, per inizializzare l'array delle regioni, per poi andare a chiamare la funzione di somma per gestire tutte le metriche.

2. **initialize_sad()**: setta ognuno dei valori di similarità delle regioni a 0.

3. **sum(*int k*)**: calcola la similarità della *k*-esima regione inserita in input.

4. **find_min_sad()**: restituisce la regione più simile alla query di ricerca.

```
void Pr::calculate_sad()
{
    initialize_sad();

    for(int k=0; k<regions; k++)
    {
        sad[k] = sum(k);
    }
}

void Pr::initialize_sad(){

    sad = new double[regions];

    for(int i = 0; i<regions; i++)
    {
        sad[i] = 0;
    }
}
```

```
double Pr::sum(int k)
{
    double sum = 0;

    for(int i=0; i<rows; i++)
    {
        for(int j=0; j<querycols ; j++)
        {
            sum += abs(elements[i][j+k]-query[i][j]);
        }
    }

    return sum;
}
```

```
int Pr::find_min_sad()
{
    double min_value = sad[0];
    int index = 0;

    for(int i=1; i<regions; i++)
    {
        if(sad[i]<min_value)
        {
            min_value = sad[i];
            index = i;
        }
    }

    return index;
}
```

4.3. Classe di controllo (Versione Parallela)

Effettuando la parallelizzazione del programma si va a toccare 3 delle precedenti funzioni, escludendo quella per il calcolo del valore minimo: questo perché, come vedremo nei test effettuati, per la suddivisione in regioni di studio può essere molto utile mantenere quel parametro anche a meno di 10 unità, il che non influisce

molto sulle prestazioni finali.
Per quanto riguarda **calculate_sad()** e **initialize_sad()** è stata effettuata una parallelizzazione dei cicli for, attraverso le direttive *pragma omp*, in modo da utilizzare il numero massimo di threads disponibili sul dispositivo su cui si stà lavorando.
Riguardo a **sum(*int k*)**, invece, è stata effettuata una *riduzione*, rispetto alla somma presente nel ciclo for di iterazione delle colonne delle matrici, in modo da eseguire in parallelo più somme contemporaneamente.

```
void Pr_parallel::calculate_sad()
{
    initialize_sad();

    #pragma omp parallel for num_threads(omp_get_max_threads())
    for(int k=0; k<regions; k++)
    {
        sad[k] = sum(k);
    }
}
```

```
void Pr_parallel::initialize_sad(){

    sad = new double[regions];

    #pragma omp parallel for num_threads(omp_get_max_threads())
    for(int i = 0; i<regions; i++)
    {
        sad[i] = 0;
    }
}
```

```
double Pr_parallel::sum(int k)
{
    double sum = 0;

    for(int i=0; i<rows; i++)
    {
        #pragma omp simd reduction(+:sum)
        for(int j=0; j<querycols ; j++)
        {
            sum += abs(elements[i][j+k]-query[i][j]);
        }
    }

    return sum;
}
```

5. Calcolo delle prestazioni

Per effettuare dei test è stato utilizzato un Dataset basato su valori reali, che prende il nome di **BlogFeedback**, disponibile all'*UCI Machine Learning Repository* [3] (il file corrispondente è *blogData_train.csv*).
La matrice che corrisponde ai suoi valori numerici è data da 281 righe per 60021 colonne.
Poiché si tratta di un'enorme quantità di dati, sono state utilizzate delle query definite casualmente dal programma stesso per poter procedere subito alle analisi.
Nella seguente tabella vengono riportati i risultati ottenuti, suddividendo il Dataset dalle 2 alle 10 regioni, con un dispositivo che utilizza la seguente architettura: *CPU Intel(R) Core(TM) i3-3217U, RAM 4 GB ddr3, Processori Logici: 4, Processori Fisici: 2*.
I valori di tempo sequenziale e parallelo vengono rappresentati in millisecondi.

Regioni	Tempo Seq.	Tempo Par.	Speedup
2	0.588862	0.298821	1.971
3	0.977397	0.335793	2.911
4	1.19542	0.391792	3.051
5	1.4571	0.653596	2.23
6	1.92481	0.654602	2.94
7	1.91282	0.71156	2.688
8	2.18664	1.01016	2.164
9	2.62541	1.14934	2.284
10	2.83424	1.59339	1.778

Come è possibile vedere nei risultati ottenuti con i valori delle regioni tra 3 e 9, si ottiene quello che viene definito **Speedup Superlineare**, ossia uno di quei rari casi in cui lo Speedup supera il numero dei processori fisici.
Pertanto, conviene effettuare una suddivisione di regioni scegliendo uno di questi valori per mantenere le prestazioni dell'algoritmo parallelo più che ottimali.

6. Conclusioni

Con la versione definita in OpenMP, questo programma raggiunge uno Speedup superlineare quando si tratta di suddividere la ricognizione del dataset in poche regioni.
Le prestazioni restano comunque ottime anche per suddivisioni differenti, il che rende questo al-

goritmo ottimo per lo studio di pattern, riguardanti immagini o altri tipi di dati, mediante la metrica di controllo *SAD*.

Riferimenti bibliografici

[1] Pattern Recognition. URL: https://en.wikipedia.org/wiki/Pattern_recognition

[2] Sum of absolute differences. URL: https://en.wikipedia.org/wiki/Sum_of_absolute_differences

[3] UCI Machine Learning Repository. URL: https://archive.ics.uci.edu/ml/index.php