Relazione del Progetto di Parallel Computing

Filtro di Bloom

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 alla creazione del filtro di Bloom.
Per la loro **implementazione** è stato utilizzato il **linguaggio Python** e, per la parallelizzazione, è stata utilizzata la libreria **Joblib**.

Descrizione del filtro di Bloom:

Un **filtro di Bloom** si definisce come un vettore di n bit che consente di **filtrare** un insieme di dati considerato. Il suo **obiettivo** è quello di filtrare l'insieme di dati considerato selezionando tra questi solo le tuple che soddisfano il criterio di selezione utilizzato.

Essendo un **metodo probabilistico** usato per testare se un elemento appartiene a un insieme o meno è possibile che si ottengano tra i risultati dei **falsi positivi**, ovvero degli elementi che non devono far parte dell'insieme filtrato dei dati ma che il filtro non è riuscito ad escludere.

Il filtro di Bloom si compone di:

1. Un **vettore di n bit**, inizialmente posti tutti a zero;
2. Una collezione di **funzioni hash** $\boldsymbol{h_1}, \boldsymbol{h_2}, \dots, \boldsymbol{h_r}$, ciascuna delle quali trasforma i valori chiave in un intero compreso tra 0 ed (n – 1);
3. Un **insieme S** di m valori dell'attributo chiave;

Dati questi componenti, il filtro cerca di selezionare tutti gli elementi del flusso di dati aventi valore del campo chiave nell'insieme S, e cerca di rifiutare la maggior parte degli elementi del flusso il cui valore di tale campo non è all'interno di S.

L'idea è quella di valutare le r funzioni hash sugli elementi dell'insieme S, andando successivamente a porre ad 1 il valore del bit del vettore prima citato, corrispondente al risultato dell'applicazione della funzione hash attualmente considerata.

Per verificare l'appartenenza di un elemento all'insieme S, si applicano le r funzioni hash a tale elemento e si verifica se i bit del vettore, nelle posizioni indicate dai valori risultanti, sono posti a 0 o ad 1.

- Se tutti i bit nelle posizioni corrispondenti nel vettore hanno valore 1, il dato viene fatto **passare** dal filtro;
- Se invece sono presenti uno o più bit posti a 0, il dato non può essere in S e si può **scartare**.

Filtro di Bloom con implementazione sequenziale:

Nella versione sequenziale del programma è stata definita inizialmente la dimensione **n** del vettore di bit usato per il filtraggio, il quale viene successivamente creato col nome di **"bloomBitVector"** ed inizializzato con tutti i suoi bit a 0 (con l'uso di un ciclo for).

In questo contesto sono state utilizzate come **dati** una serie di stringhe. Più in particolare è stato scelto di considerare l'insieme S come l'insieme delle parole più frequenti della lingua inglese, mentre come insieme da filtrare un insieme di altre parole inglesi, contenente però 14 delle parole presenti anche all'interno di S.

Entrambi gli insiemi di parole sono stati passati alle liste del programma set_S ed extended_set mediante i due rispettivi file **"most_uses_english_words.txt"** e **"parole_da_filtrare.txt"** all'interno dei quali erano contenuti.

Nel programma sono state poi definite 7 funzioni hash (da h_0 ad h_6), che prendono come parametro di input la parola su cui devono operare ed utilizzano logiche diverse nelle loro definizioni. Per tutte queste funzioni però è stato applicato al loro risultato il modulo n per far sì che fosse rispettato il vincolo per cui il loro risultato fosse compreso tra i valori 0 ed (n-1).

Successivamente, per ogni parola presente all'interno dell'insieme S (**set_S**) iniziale, si pongono i rispettivi bit del filtro al valore 1 nelle posizioni calcolate dalle funzioni hash precedentemente definite mediante delle istruzioni con la forma illustrata di seguito:

bloomBitVector[h_i(set_S[i])] = 1

(dove nel codice al posto della "i" c'è il numero della corrispondente funzione hash).

In questo modo viene così conclusa quella che può essere definita come **fase di inizializzazione (e/o gestione) del filtro**.

Dopo questa prima fase si passa poi alla **fase di filtraggio** vero e proprio.
Questa è stata definita nella funzione **"matchingCheck"**, che prende come parametri di input la stringa (parola) da considerare ed il vettore di bit (**bloomBitVector**) da usare per il filtraggio. Nel caso in cui il bit del vettore che si trova alla posizione calcolata applicando la funzione hash attualmente considerata sulla stringa abbia valore 0, si passa all'iterazione successiva del ciclo (e si considera dunque la parola successiva dell'insieme). Se invece tutte le funzioni hash calcolate sulla parola attuale portano in posizioni del vettore di bit che hanno valore pari ad 1, verrà stampata tale stringa come risultato attuale.

Questo procedimento consente dunque di trovare tutte le **parole comuni** sia al primo che al secondo insieme e le riproduce in output come illustrato nell'esempio di esecuzione sotto riportato.

Esempio di esecuzione:

```
Corrispondenze trovate:
hot
come
did
my
sound
no
most
number
who
over
know
```

Filtro di Bloom con implementazione parallela:

L'implementazione parallela risulta dal punto di vista strutturale la stessa di quella sequenziale, con l'aggiunta però dell'utilizzo della libreria **Joblib di Python**.

Inizialmente è stata aggiunta una variabile **"numThread"**, nella quale viene definito il numero di threads che si vogliono usare per parallelizzare il codice, mentre la libreria Joblib è stata utilizzata in **3 parti** del programma:

1. Il **primo punto** in cui questa si incontra nel codice è all'interno della **funzione hash h5**. La logica di questa funzione sfrutta la codifica Unicode dei caratteri della stringa passata come parametro in input e memorizza in una lista **"result"** i loro valori. Successivamente si calcola la corrispondente **matrice di Vandermonde di ordine n** usando la funzione **"vander"** del modulo **numpy** di Python ed effettua la somma dei valori interni a tale matrice, restituendola come risultato.
   Questa funzione può essere parallelizzata usando la libreria Joblib per la costruzione della lista result. In questo caso la funzione su cui si parallelizza è la funzione **"ord()"** che, usando il for parallelizzato mediante **"Parallel"**, viene applicata a tutti i caratteri della parola attualmente considerata.

2. Il **secondo punto** riguarda la **gestione parallela dell'array di bit** mediante l'uso delle funzioni hash precedentemente definite. Mentre nel codice sequenziale ciascuna funzione hash era applicata internamente ad un ciclo e veniva eseguita in maniera sequenziale, è stato pensato in questo caso di parallelizzare creando un **ciclo parallelo per ognuna** delle 7 funzioni hash a disposizione. Ciascun ciclo parallelo applica una funzione hash ad ogni parola dell'insieme considerato, e restituisce come output un vettore (lungo come quello di input) contenente i risultati delle varie funzioni hash. Questa strategia consente di parallelizzare l'inizializzazione dell'array di bit sfruttando il **multiprocessing**.

3. La **terza parte** del programma su cui si può agire per parallelizzare è quella relativa al **filtraggio** vero e proprio dei dati. In questo caso si definisce un ciclo parallelo che consente di considerare ogni parola del set da filtrare (**"extended_set"**), applicando a ciascuna la funzione di filtraggio **"matchingCheck"**. Siccome in questa avviene il controllo del valore dei bit di controllo per la parola considerata, parallelizzando preventivamente sul ciclo che considererà le varie parole da filtrare si riesce a velocizzare l'esecuzione del programma.

Confronto delle prestazioni:

Il confronto delle prestazioni tra la versione sequenziale e quella parallela è avvenuto basandosi sui **tempi di esecuzione** dei due programmi e/o di alcune parti al loro interno.

Inizialmente consideriamo una normale esecuzione sia nel caso sequenziale che in quello parallelo agendo al massimo delle prestazioni, ovvero utilizzando **8 threads** durante l'esecuzione (del caso parallelo) ed utilizzando in entrambi i casi una dimensione del vettore di bit pari ad **n = 30000**.

In questa situazione il **programma sequenziale** mostra in output i seguenti risultati:

```
Tempo di inizializzazione:  8.770047187805176
Corrispondenze trovate:
ground
mind
wonder
hot
come
did
my
sound
no
most
number
who
over
know
Tempo complessivo di esecuzione:  9.266271829605103
```

Nell'output, come si può notare dall'immagine precedente, vengono illustrati:

- Il tempo relativo alla fase di inizializzazione (e dunque anche gestione) del filtro;
- Le corrispondenze (parole) trovate del secondo insieme (fatte passare, dunque, dal filtro);
- Il tempo complessivo di esecuzione del programma.

Analizzando invece **l'output del programma parallelo**, si osservano i seguenti risultati:

```
Tempo di inizializzazione:  4.435826778411865
Corrispondenze trovate:
ground
mind
hot
my
no
did
wonder
come
most
sound
who
number
over
know
Tempo complessivo di esecuzione:  4.717247009277344
```

Guardando i tempi di inizializzazione (e gestione) del filtro notiamo già un'ottima riduzione dei tempi di esecuzione, passando da circa 8.77 secondi a circa 4.43 secondi. Analizzando invece i tempi complessivi di esecuzione dei due programmi, i risultati osservati sono i seguenti:

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	**CASO PARALLELO**
9.266271829605103	4.503967571258545

Calcolando dunque lo **speedup** risultante si ottiene:

$$Speedup = \frac{T_S}{T_P} = \frac{9.266271829605103}{4.503967571258545} = 2.057357581510256$$

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

Un **altro caso interessante** da valutare può essere quello in cui si dimezzano ad esempio il numero di threads utilizzati mantenendo costante il numero di dati. Settiamo dunque **numThreads = 4**, e notiamo che i tempi di esecuzione cambiano come riportato nella seguente tabella:

TEMPI DI ESECUZIONE		
	CASO SEQUENZIALE	**CASO PARALLELO**
TEMPI DI GESTIONE FILTRO	8.770047187805176	4.510117292404175
TEMPI DI ESECUZIONE TOTALE	9.266271829605103	4.78024959564209

Calcoliamo allora lo **speedup** sul tempo complessivo di esecuzione:

$$Speedup = \frac{T_S}{T_P} = \frac{9.266271829605103}{4.78024959564209} = 1.938449372612821$$

Si nota infatti che tale speedup risulta ridotto in questa situazione, come ci si aspettava.

Adesso, una prova interessante da fare, è invece quella in cui si analizzano i tempi di esecuzione partendo dal caso sequenziale ed aumentando via via il numero di threads utilizzati nella versione parallela:

Tempi di esecuzione con n = 30000	
Sequenziale	9.266271829605103
nThreads = 1	9.359665632247925
nThreads = 2	6.408173561096191
nThreads = 3	5.130772590637207
nThreads = 4	4.78024959564209
nThreads = 5	4.687883138656616
nThreads = 6	4.541127681732178
nThreads = 7	4.677275896072388
nThreads = 8	**4.503967571258545**
nThreads = 9	4.85349702835083
nThreads = 10	5.1233971118927
nThreads = 11	5.891347408294678

Da questa tabella si nota che i tempi di esecuzione diventano sempre più piccoli fino al raggiungimento del **valore minimo** nel caso in cui il loro numero sia pari al numero di **processori logici** della macchina utilizzata (che nel caso in questione è, infatti, 8). Continuando ad aumentare il numero di threads è visibile invece un costante **peggioramento** dei tempi di esecuzione, i quali tendono a diventare poi via via più lunghi.