**Relazione del progetto di Parallel Computing** **(Filtro di Bloom)**

Giuseppe Martinelli Matr: 7093926

La seguente è una descrizione delle scelte implementative che sono state prese per la costruzione di una versione **sequenziale** e **parallela** 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 dell’algoritmo**

Il filtro di Bloom è una struttura dati probabilistica altamente efficiente che viene utilizzata per testare ogniqualvolta un elemento è un membro di un set. In questo scenario, vi è la possibilità di falsi positivi mentre falsi negativi sono impossibili. In altre parole, una eventuale query può ritornare il valore “forse nel set” oppure “sicuramente non nel set”. Più elementi vengono aggiunti, maggiore è la possibilità di falsi positivi.

Il filtro di Bloom si compone di:

1. Un vettore di *N* bit, inizialmente posti tutti a zero
2. Una collezione di funzioni hash, 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. Altrimenti,se sono presenti uno o più bit posti a 0, il dato non può essere in S e si può scartare.

**Implementazione Sequenziale**

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. Altrimenti,se sono presenti uno o più bit posti a 0, il dato non può essere in S e si può scartare.

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 ℎ0 ad ℎ6), 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.

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.

**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**

In questa sezione, ci concentriamo sui tempi di esecuzione ottenuti con configurazioni di parametri differenti

**Nota:** Per ogni configurazione sono state eseguite 10 esecuzioni e poi fatta la media

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Configurazione | Seq Init | Seq Esec | Parallel Init | Parallel Esec | Speedup |
| 30000 - 4 | 17.822 | 19.076 | 8.890 | 9.522 | 2.003 |
| 80000 - 4 | 42.428 | 43.781 | 24.544 | 25.586 | 1.7111 |
| 80000 - 2 | 42.428 | 43.781 | 39.085 | 40.509 | 1.08 |
| 80000 - 6 | 42.428 | 43.781 | 27.038 | 28.163 | 1.554 |
| 80000 - 8 | 42.428 | 43.781 | 23.773 | 24.852 | 1.761 |
| 80000 - 10 | 42.428 | 43.781 | 23.169 | 24.382 | 1.795 |
| 80000 - 16 | 42.428 | 43.781 | 23.567 | 24.765 | 1.767 |

Per ogni configurazione sono state calcolate le probabilità di falsi positivi, il numero di funzioni di hash ottimale così come la grandezza ottimale di filtro