Progetto di Parallel Computing 2021-2022

Gabriele Boddi
gabriele.boddi@stud.unifi.it

Abstract

In questo elaborato verrà mostrata l'implementazione del "Bloom Filter", una struttura utilizzata negli algoritmi di ricerca.
Il suo sviluppo verrà mostrato in versione sequenziale e parallela tramite l'utilizzo di C++ e OpenMP.

1. Introduzione

Un **Bloom Filter** è una struttura dati *probabilistica*, concepita da Burton Howard Bloom nel 1970, che consente di calcolare la probabilità della presenza di un elemento all'interno di un insieme di dati.
Il suo utilizzo può comportare la presenza di *Falsi positivi* (elementi non presenti che invece vengono segnati come tali), ma non di *Falsi negativi*.
Il suo algoritmo sfrutta la logica delle *Funzioni Hash* [1].

2. Descrizione dell'Algoritmo

La struttura si presenta come un *array* vuoto di *m* bit, tutti a 0.
Devono esserci *k* funzioni hash differenti per poter effettuare una mappatura efficiente dei vari elementi, presenti all'interno dell'insieme che vogliamo controllare, rispetto ai bit del Bloom Filter.
Il funzionamento dell'algoritmo si basa essenzialmente sulle funzioni di *Aggiunta* e di *Controllo della presenza di un elemento nell'insieme*: nel primo caso si passa il valore che si vuole aggiungere su ognuna delle funzioni hash per ottenere *k* posizioni dell'array che saranno settate ad 1.
Nel secondo caso, oltre a passare il valore su tutte le funzioni, si verifica se almeno una delle posizioni dell'array è 0: in quel caso l'elemento **sicuramente non è presente nell'insieme**, altrimenti è **probabile la sua presenza**.

2.1. Ottimizzazione per evitare i falsi positivi

Come detto in precedenza è possibile riscontrare dei falsi positivi, ossia degli elementi che pur non essendo presenti nell'insieme di studio vengono considerati come tali.
La probabilità di ottenere questi valori aumenta all'aumentare del numero dei bit ad 1 presenti nella struttura dati: tanti più bit sono uguali a 1, tanto più facile è incontrare qualche valore assente che viene mappato delle funzioni hash in posizioni dell'array diverse da 0.
Si possono effettuare dei miglioramenti considerevoli prestando attenzione al numero di funzioni hash da utilizzare ed alla dimensione del Bloom Filter stesso.
Di seguito verranno riportati maggiori dettagli.

2.2. Numero ottimale di bit e funzioni hash

Per garantire una grande efficienza dell'algoritmo la tolleranza dei falsi positivi deve essere mantenuta dell'1% [1]: per questo motivo il numero migliore di bit che si possono usare è dato dalla seguente equazione:

$$m\_ottimo = \frac{|n \cdot ln(0.01)|}{2 \cdot ln(2)} \qquad (1)$$

conoscendo tale valore, si può anche definire il numero *ottimale* delle funzioni hash:

$$k\_ottimo = \frac{m\_ottimo}{n} \cdot ln(2) \qquad (2)$$

in questo modo, unendo una dimensione ragionevole della struttura ad un numero ottimale di funzioni hash da utilizzare, le operazioni di aggiunta e di controllo richiederanno poche istruzioni su ogni elemento che vogliamo testare [2].

3. Implementazione

I programmi presenti con questa documentazione sono 2: un programma *sequenziale* ed uno *parallelo* dove è stato utilizzato **OpenMP**.
Entrambi sono stati implementati in **C++** con la possibilità di studiare un insieme di dati di qualsiasi tipo.
Il *Costruttore* di entrambi i programmi, che sarà visibile di seguito, inizializzerà il bloom filter con la sua dimensione ed il numero di funzioni di hash.

```
this->my_set = my_set;
this->n = this->my_set.size();
this->m = abs((n*log(0.01))) / (log(2)*2);
this->k = (m/n)*log(2);
initializeBloomFilter();
add(my_set);
```

L'insieme di dati studiato viene rappresentato da una *deque*, ossia una struttura dinamica che utilizza le funzionalità di una coda e può essere usata anche come array.
Di seguito verranno mostrati i dettagli di entrambi i codici, seguiti da un test effettuato su un testo di grandi dimensioni (*1235195* caratteri) per calcolare lo **Speedup**.

3.1. Versione Sequenziale

Il programma lavora con 4 funzioni principali: **initializeBloomFilter** che inizializza la struttura dati, **add** che permette di aggiungere un valore, **probably_contains** che controlla la presenza di uno o più elementi all'interno dell'insieme principale e **check** che restituisce un valore booleano indicante la probabilità di presenza di un elemento nell'insieme.
add e **probably_contains** utilizzano le funzioni di hashing (anche queste tipizzate per avere un calcolo corretto dell'elemento di interesse), per calcolare una posizione del bloom filter nel seguente modo:

$$j = ((h(e) \cdot (i+1)) + n) \mod m \qquad (3)$$

dove *h(e)* è la funzione di hash sull'elemento *e* ed *i* indica l'*i*-esima funzione che si sta utilizzando.
Di seguito si riporta i 4 codici delle funzioni principali.

```
template <typename T>
void BloomFilter<T>::initializeBloomFilter(){
    bloom_filter = new bool[m];
    for(size_t i=0; i<m; i++){
        bloom_filter[i] = false;
    }

}
```

```
template <typename T>
void BloomFilter<T>::add(std::deque<T> my_set){
    std::hash<T> h;
    while(!my_set.empty()){
        for(size_t j=0; j<k; j++)
        {
            size_t index =
            ((h(my_set.front())*(j+1))+n)%m;
            if(!bloom_filter[index]) {
                bloom_filter[index] = true;
            }
        }
        my_set.pop_front();
    }

}
```

```
template <typename T>
bool*
BloomFilter<T>::probably_contains(std::deque<T>
elements)
{

    bool *results = new bool[elements.size()];

    for(size_t i=0; i<elements.size(); i++){
        results[i] = check(elements[i]);
```

```
    }

    return results;
}
```

```
template <typename T>
bool* BloomFilter<T>::check(T e){
    std::hash<T> h;
    bool probably_in = true;

    for(size_t i=0; i<k && probably_in; i++){
        size_t j = ((h(e)*(i+1))+n)%m;
        if(!bloom_filter[j]) probably_in = false;
    }

    return probably_in;
}
```

3.2. Versione Parallela

L'algoritmo parallelo, sviluppato tramite l'utilizzo di **OpenMP**, condivide il codice sequenziale con la possibilità di definire il numero di thread che si vogliono usare per decidere se eseguire l'algoritmo in modalità parallela.
In questo codice sono stati parallelizzati i cicli for delle funzioni sopra descritte tramite le direttive **pragma**, dove è stato specificato il numero dei thread da utilizzare come il *massimo* disponibile sul dispositivo in cui si stà lavorando.
Inoltre, la funzione **add** è stata scomposta nella sottofunzione **insert_element** per eseguire una parallelizzazione più efficiente evitando sezioni critiche.
Di seguito è possibile vedere come sono definite le funzioni con tali modifiche.

```
template <typename T>
void
BloomFilter_parallel<T>::initializeBloomFilter(){
    bloom_filter = new bool[m];

    #pragma omp parallel for
    num_threads(omp_get_max_threads())
    for(size_t i=0; i<m; i++){
        bloom_filter[i] = false;
    }

}
```

```
template <typename T>
void BloomFilter_parallel<T>::add(std::deque<T>
my_set){

    #pragma omp parallel for
    num_threads(omp_get_max_threads())
    for(size_t i=0; i<my_set.size(); i++)
    {
        insert_element(my_set[i]);
    }
}
```

```
template <typename T>
bool* BloomFilter_parallel<T>::probably_contains
(std::deque<T> elements){

    bool *results = new bool[elements.size()];

    #pragma omp parallel for
    num_threads(omp_get_max_threads())
    for(size_t i=0; i<elements.size(); i++){
        results[i] = check(elements[i]);
    }

    return results;
}
```

I cicli che iterano sul numero di funzioni hash disponibili non sono stati parallelizzati: questo perché tramite l'utilizzo di un *flag* booleano per indicare la fine dei controlli bastano pochissime istruzioni ogni volta che vengono chiamati.
Nel caso ottimo, un elemento che non fa parte dell'insieme richiede al più una sola iterazione.

4. Esecuzione

Nel corpo principale del *main* il contenuto dell'insieme di interesse viene copiato all'interno di una *deque*, che verrà poi passata dentro al costruttore del programma principale.
Una volta completata l'inizializzazione dei parametri, verrà richiamato il metodo **probably_contains** per controllare se uno o più elementi sono presenti nell'insieme principale.
Poiché il *caso pessimo* in termini di tempo si ha quando si confronta solo elementi che sono presenti, è stata utilizzata la stessa *deque* utilizzata in precedenza per vedere dove si possono spingere i costi.
Il dispositivo su cui sono stati effettuati i test ha le

seguenti caratteristiche: *CPU Intel(R) Core(TM) i3-3217U, RAM 4 GB ddr3, Processori Logici: 4, Processori Fisici: 2*.

4.1. Lavoro su testo

Entrambi i programmi sono stati eseguiti sul file di testo *2701.txt*, presente all'interno della distribuzione *Moby-Dick-or-the-Whale*, del progetto Gutenberg, disponibile su Github [3].
Nella tabella sottostante si può vedere il risultato di tale test.

Tempo Sequenziale	Tempo Parallelo	Speedup
0.421866	0.249991	1.687

4.2. Lavoro su interi

I seguenti risultati, invece, riguardano l'esecuzione dei due programmi su un insieme di chiavi *intere* che va da 10^3 a 10^7.
Per i valori piccoli lo *Speedup* tende ad 1, mentre cresce vistosamente via via che aumenta la cardinalità dei dati stessi.

Valori	Tempo Seq.	Tempo Par.	Speedup
10^3	0.000999	0.000998	1.001
10^4	0.004001	0.003996	1.001
10^5	0.038977	0.028984	1.345
10^6	0.042374	0.026642	1.345
10^7	4.27939	2.55342	1.676

5. Conclusioni

Con la versione definita in OpenMP si può notare un considerevole miglioramento dello Speedup quando si tratta di effettuare test su enormi quantità di dati, il che rende tale programma estremamente utile e vantaggioso per ricercare elementi su insiemi di vari tipi.
Prestando attenzione alla dimensione del Bloom Filter, inoltre, si arriva a sacrificare poco spazio di memoria ottenendo così un grosso bilanciamento in termini di prestazioni.

Riferimenti bibliografici

[1] Bloom Filter (Wikipedia). URL: https://en.wikipedia.org/wiki/Bloom_filter

[2] Bloom Filter | Brilliant Math & Science Wiki. URL: https://brilliant.org/wiki/bloom-filter/

[3] Moby Dick or The Whale. URL: https://github.com/GITenberg/Moby-Dick--Or-The-Whale_2701