Relazione Progetto Parallel Programming
Filtro Bloom

Ruci Sindi
7090797 sindi.ruci@stud.unifi.it

September 13, 2022

1 Introduzione

I ***filtiri Bloom*** sono una serie di funzioni di analisi delle informazioni che permettono di determinare se un dato (oppure un insieme di dati) è contenuto all'interno di un data Base. Elemento fondamentale dei filti Bloom è un array di bit. Il filtro funziona tramite delle funzioni hash, ogni informazione inserita passa attraverso alcune funzioni hash, ognuna di queste funzioni produce un numero compreso tra 0 e la lunghezza dell'array di bit. Questo numero indica il bit che deve essere posto a 1 all'interno dell'array. Per ogni informazione quindi vengono messi a 1 un numero di bit al massimo pari al numero di funzioni hash che costituiscono il filtro.

1.1 Struttura del codice

Il codice è composto da tre file, due sono i codici in sequenziale e in parallelo, mentre uno è il file con le funzioni hash da me inventate per implementare i filtri Bloom. Il file di utilità con le funzioni è però utilizzato solo nell'implementazione sequenziale, in quanto in quella parallela le funzioni sono direttamente nel file contenente il main. Questa scelta viene dal fatto che il codice mi è parso più veloce con le funzioni nello stesso file e con qualche variabile globale. Sono implementate per ogni file due funzioni, una che inizializza l'array di bit e una che fa la ricerca di alcune parole. Per inizializzare i l'array è stata scelto un file con 199 parole (una per riga), per la ricerca ci sono invece due file, uno con 68 parole e uno con 90. Il file per inizializzare l'array ha solo parole in inglese, mente il file con 90 parole ha 82 parole in italiano e 8 in inglese, questo per fare la verifica dei falsi positivi che il filtro può restituire.

2 Implementazione sequenziale

Come descritto nella sezione precedente l'implementazione sequenziale presenta due funzioni, la prima è:

Listing 1: Funzione di inizializzazione dell'array di bit

```
def bool_hash_functions(string, array_of_bits,n):
```

La funzione prende come argomento una stringa (nel mio caso si tratta di una parola), l'array di bit e la grandezza di questo array e calcola tutte le funzioni hash per quella parola e pone i corrispondenti bit a uno nel array di bit passato come argomento. Nel main questa funzione è inserita all'inerno di un for che esegue la funzione per tutte le parole presenti nel file di inizializzazione. Le parole presenti nel file di inizializzazione sono immagazzinate in una lista, che è l'effettivo elemento che viene usato nel ciclo for precedete e non il file di testo.

```
def bool_hash_functions(string, array_of_bits,n):
    i = h0(string,n)
    array_of_bits[i] = 1
    i = h1(string,n)
    array_of_bits[i] = 1
    i = h2(string,n)
    array_of_bits[i] = 1
    i = h3(string,n)
    array_of_bits[i] = 1
    i = h4(string,n)
    array_of_bits[i] = 1
    i = h5(string,n)
    array_of_bits[i] = 1
    i = h6(string,n)
    array_of_bits[i] = 1
    i = h7(string,n)
    array_of_bits[i] = 1
```

Figure 1: Codice per l'inizializzazione dell'array di bit

Per la ricerca invece i file sono 2, uno con parole in inglese e poche in italiano, e uno con molte parole in italiano e poche in inglese. Il file di ricerca viene aperto, letto e ogni parola al suo interno viene immagazzinata in una lista. Questa lista viene poi scorsa all'inerno di un for che esegue la funzione di ricerca.

Listing 2: funzione di ricerca

```
def is_the_string_present(string_to_find ,n, array_of_bits):
```

La funzione prende come argomento la parola da cercare, l'array di bit e la sua dimensione. All'inerno della funzione vengono calcolate tutte le funzioni hash di quella parola e viene verificato che ogni bit risultante sia uguale a 1 nell'array di bit, se anche solo uno di questi bit è uguale a 0, la parola viene considerata non trovata.

```python
def is_the_string_present(string_to_find,n,array_of_bits):
    if array_of_bits[h0(string_to_find, n)] == 0:
        pass
    if array_of_bits[h1(string_to_find, n)] == 0:
        pass
    if array_of_bits[h2(string_to_find, n)] == 0:
        pass

    if array_of_bits[h3(string_to_find, n)] == 0:
        pass

    if array_of_bits[h4(string_to_find, n)] == 0:
        pass

    if array_of_bits[h5(string_to_find, n)] == 0:
        pass

    if array_of_bits[h6(string_to_find, n)] == 0:
        pass

    if array_of_bits[h7(string_to_find, n)] == 0:
        pass

    else:
        print('the string: ' + string_to_find +  ' is in the array')
```

Figure 2: Codice per la ricerca delle parole

3 Implementazione Parallela

Il codice parallelo esegue entrambe le mansioni in parallelo, sia l'inizializzazione che la ricerca. Per l'inzializzazione esegue ognuna delle funzioni hash in parallelo e ritorna per ogni funzione una lista con all'interno gli indici da porre a uno nell'array di bit. Una volta finita l'esecuzione di tutte le funzioni hash i risultati vengono utilizzati per inizializzare l'array in maniera sequenziale.

```
array_0 = Parallel(n_jobs=cores)(delayed(h0) (string) for string in text_array)
array_1 = Parallel(n_jobs=cores)(delayed(h1) (string) for string in text_array)
array_2 = Parallel(n_jobs=cores)(delayed(h2) (string) for string in text_array)
array_3 = Parallel(n_jobs=cores)(delayed(h3) (string) for string in text_array)
array_4 = Parallel(n_jobs=cores)(delayed(h4) (string) for string in text_array)
array_5 = Parallel(n_jobs=cores)(delayed(h5) (string) for string in text_array)
array_6 = Parallel(n_jobs=cores)(delayed(h6) (string) for string in text_array)
array_7 = Parallel(n_jobs=cores)(delayed(h7) (string) for string in text_array)

for i in range(len(array_0)):
    array_of_bits[array_0[i]] = 1
    array_of_bits[array_1[i]] = 1
    array_of_bits[array_2[i]] = 1
    array_of_bits[array_3[i]] = 1
    array_of_bits[array_4[i]] = 1
    array_of_bits[array_5[i]] = 1
    array_of_bits[array_6[i]] = 1
    array_of_bits[array_7[i]] = 1
```

Figure 3: Codice per l'inzializzazione dell'aray in parallelo

Per la ricerca invece la funzione è esattamente la stessa che era in sequenziale solo che questa viene avviata all'interno del comando Parallel()(delayed) in modo che venga eseguita in parallelo sulle parole del file di ricerca.

```
Parallel(n_jobs=cores)(delayed(is_the_string_present)(string, array_of_bits) for string in string_to_find)
```

Figure 4: Comando che parallelizza il codice di ricerca

Entrambi i codici (parallelo e sequenziale) presentano lo stesso numero di falsi positivi, questo si aggira a circa il 7.04%.

4 Confronto delle prestazioni

I confronti che si prenderanno in considerazione sono due:

1. confronto tra codice sequenziale e parallelo nella ricerca e nell'inizializzazione ponendo il codice parallelo al massimo dei thread
2. prestazioni del codice parallelo al cambiare del numero di thread

4.1 Sequenziale vs Parallelo

L'inizializzazione per quanto riguarda il file in sequenziale impiega poco più di 8 secondi, mentre per quanto riguarda l'inizializzazione in parallelo l'esecuzione dura 3.66 secondi.

```
"C:\Users\rucis\OneDrive\Desktop\UNI\MAGISTRALE\PRIMO ANNO\SECONDO SEMESTRE\PARALLEL PROGRAMING\Bloom filter\Scripts\python.exe"
 "C:\Users\rucis\OneDrive\Desktop\UNI\MAGISTRALE\PRIMO ANNO\SECONDO SEMESTRE\PARALLEL PROGRAMING\Bloom filter\sequential_bloom_filter.py"
8.007917642593384
```

Figure 5: Inizializzazione sequenziale

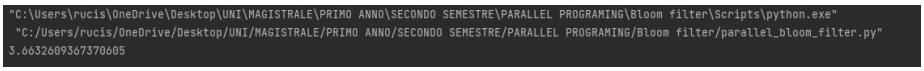

Figure 6: Inizializzazione parallela

Come si vede anche dagli screen la versione parallela ha una velocità maggiore di oltre due volte quella sequenziale, per la precisione lo speedup è pari a 2,1860

Per la ricerca invece i tempi di esecuzione in sequenziale sono circa 3 secondi, per la precisione sono 3.4530439376831055 secondi. Sotto troviamo lo screen delle parole trovate e del tempo di esecuzione. C'è da precisare che il file utilizzato per la ricerca è di 90 parole e tra queste 8 sono in inglese ed è giusto che siano state trovate, mentre tutte le altre sono dei falsi positivi.
Le parole corrette sono: survey,geez,dishonour,appliance,powerless,level,rightfully.

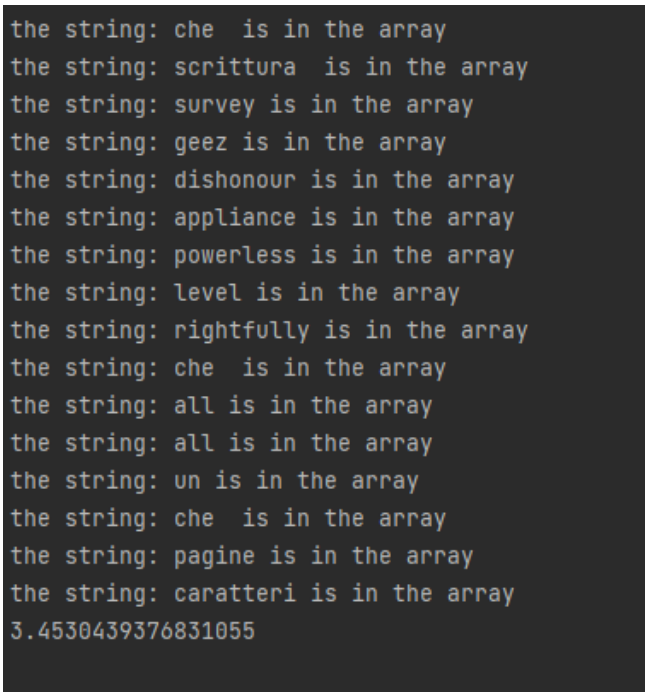

Figure 7: Risultati ricerca sequenziale

I tempi di esecuzione della ricerca in parallelo invece sono sotto al secondo, in particolare 0.7656118869781494 e l'output che otteniamo dal codice è il seguente

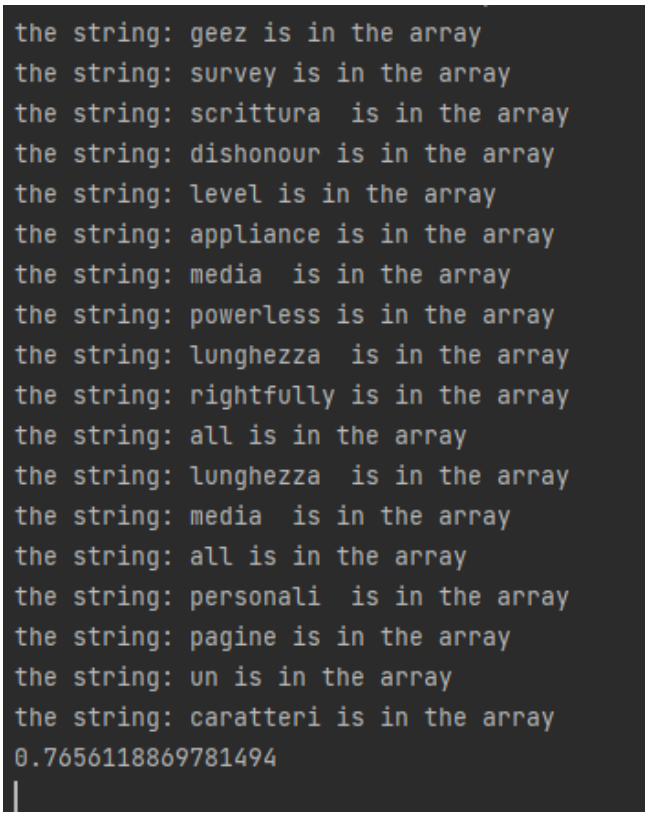

Figure 8: Risultati ricerca parallela

È evidente la maggior velocità della ricerca in parallelo, abbiamo uno speedup pari a 4,5101754510528590313159226086437, ben oltre quattro volte.

4.2 Prestazioni parallele

Per le prestazioni in paralleo si considera un numero massimo di thread pari a 8 e un numero minimo pari a 1, in modo da vedere se il codice parallelo con un solo thread ha le stesse prestazioni del sequanziale. In più si visionerà il caso in cui si vogliano utilizzare più thread rispetto a quelli effettivamente disponibili, per vedere se i tempi di esecuzione deteriorano.

Numero di thread	Tempi di inizializzazione	Tempi di ricerca
1	8.59s	3.67s
2	5.50s	2.13s
3	4.50s	1.35s
4	4.01s	1.10s
5	3.80s	1.06s
6	3.49s	0.92s
7	3.33s	0.81s
8	3.56	0.78s
10	4.14s	0.92s

Da notare come il codice parallelo, utilizzando un solo core abbia dei tempi di esecuzione più lunghi rispetto al codice sequenziale, infatti abbiamo che per l'inizializzazione il codice sequenziale impiega 8.00 secondi mentre il codice parallelo con un thread ne impiega quasi 9; per la ricerca il codice sequenziale impiega 3.45 secondi mentre il codice parallelo con un thread ne impiega 3.67. Questo mostra come in alcuni casi se si ha un codice parallelo e serve sequenziale sia meglio riscriverlo piuttosto che abbassare il numero di thread messi a disposizione.