



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Implementazione sequenziale e parallela del Filtro di Bloom

Guido Ciardi 7090798
guido.ciardi@stud.unifi.it

Esame orale di Parallel Computing

Università degli Studi di Firenze

Firenze, 12/01/2023

Filtro di Bloom

Un **filtro di Bloom** si definisce come un vettore di n bit che consente di filtrare un insieme di dati considerato.

Composizione strutturale:

- ***Vettore di n bit***, inizialmente posti tutti a zero;
- Collezione di ***funzioni hash*** h_1, h_2, \dots, h_r , ciascuna delle quali trasforma i valori chiave in un intero compreso tra 0 ed $(n - 1)$;
- ***Insieme S*** di m valori dell'attributo chiave;

Dati utilizzati

Nel codice sono stati considerati 2 file contenenti ciascuno un insieme di parole in lingua Inglese:

- File: **"*most_used_english_words.txt*"** → Insieme S da considerare, contenente una lista tra le parole inglesi più usate;
- File: **"*parole_da_filtrare.txt*"** → Insieme di parole da filtrare mediante l'ausilio di S e del filtro di Bloom;

Struttura del codice

- **Gestione** dei due insiemi di parole considerati e del vettore di bit;
- **Funzioni Hash**;
- Funzione di **matching** del vettore;

Fase di inizializzazione

- **Riempimento liste associate agli insiemi:**

```
with open('most_used_english_words.txt') as file:
    for line in file:
        line = line.replace('\n', '')
        set_S.append(line)
```

```
with open('words_to_filter.txt') as file:
    for line in file:
        line = line.replace('\n', '')
        extended_set.append(line)
```

- **Inizializzazione del filtro:**

```
for i in range(n):
    bloomBitVector.append(0)
```

```
for i in range(0, len(set_S)):
    bloomBitVector[h0(set_S[i])] = 1
    bloomBitVector[h1(set_S[i])] = 1
    bloomBitVector[h2(set_S[i])] = 1
    bloomBitVector[h3(set_S[i])] = 1
    bloomBitVector[h4(set_S[i])] = 1
    bloomBitVector[h5(set_S[i])] = 1
    bloomBitVector[h6(set_S[i])] = 1
```

Funzioni hash implementate

```
def h0(string):  
    return (len(string) * 48765276) % n
```

```
def h1(string):  
    sumUnicode = 0  
  
    for i in string:  
        sumUnicode = sumUnicode + ord(i)  
  
    return sumUnicode % n
```

```
def h2(string):  
    productUnicode = 1  
  
    for i in string:  
        productUnicode = productUnicode * ord(i)  
  
    return productUnicode % n
```

```
def h3(string):  
    mixUnicode = 1  
  
    for i in string:  
        mixUnicode = (mixUnicode * ord(i)) + (22543 * len(i))  
  
    return mixUnicode % n
```

Funzioni hash implementate

```
def h4(string):  
    return ((len(string) + ord(string[0])) * 9876367) % n
```

```
def h5(string):  
    result = []  
  
    for i in range(0, len(string)):  
        result.append(0)  
  
    for j in range(0, len(string)): # Conversione di ciascun caractere  
        result[j] = ord(string[j])  
  
    vanderMatrix = np.vander(result, n) # Creazione di una matrice  
    sumElements = 0  
    for j in range(0, len(vanderMatrix)): # Tutti gli elementi della matrice  
        for i in range(0, len(vanderMatrix[0])):  
            sumElements = sumElements + (vanderMatrix[j][i]) % 10  
    return sumElements % n
```

```
def h6(string):  
    return pow(ord(string[0]), 5) % n
```

Funzione di filtraggio

```
def matchingCheck(actualString, bloomBitVector):  
  
    if bloomBitVector[h0(actualString)] == 0: #  
        pass  
    elif bloomBitVector[h1(actualString)] == 0:  
        pass  
    elif bloomBitVector[h2(actualString)] == 0:  
        pass  
    elif bloomBitVector[h3(actualString)] == 0:  
        pass  
    elif bloomBitVector[h4(actualString)] == 0:  
        pass  
    elif bloomBitVector[h5(actualString)] == 0:  
        pass  
    elif bloomBitVector[h6(actualString)] == 0:  
        pass  
    else: # Caso di match trovato per la parola  
        print(actualString)
```

Esempio di output in un'esecuzione sequenziale

```
Tempo di esecuzione h5: 157.98341393470764
Tempo di inizializzazione: 158.0486147403717
Corrispondenze trovate:
hot
come
did
my
sound
no
most
number
who
over
know
Tempo di matching: 5.371013402938843
Tempo complessivo di esecuzione: 163.42062401771545
```


Codice parallelo

Parti di codice parallelizzate:

- Inizializzazione del filtro;
- Funzione hash "h5";
- Fase di filtraggio;

Inizializzazione del fitro

```
parallel_set0 = Parallel(n_jobs=numThreads)(delayed(h0)(set_S[i]) for i in range(0, len(set_S)))  
parallel_set1 = Parallel(n_jobs=numThreads)(delayed(h1)(set_S[i]) for i in range(0, len(set_S)))  
parallel_set2 = Parallel(n_jobs=numThreads)(delayed(h2)(set_S[i]) for i in range(0, len(set_S)))  
parallel_set3 = Parallel(n_jobs=numThreads)(delayed(h3)(set_S[i]) for i in range(0, len(set_S)))  
parallel_set4 = Parallel(n_jobs=numThreads)(delayed(h4)(set_S[i]) for i in range(0, len(set_S)))  
parallel_set5 = Parallel(n_jobs=numThreads)(delayed(h5)(set_S[i]) for i in range(0, len(set_S)))  
parallel_set6 = Parallel(n_jobs=numThreads)(delayed(h6)(set_S[i]) for i in range(0, len(set_S)))
```

```
for i in range(0, len(set_S)):  
    bloomBitVector[parallel_set0[i]] = 1  
    bloomBitVector[parallel_set1[i]] = 1  
    bloomBitVector[parallel_set2[i]] = 1  
    bloomBitVector[parallel_set3[i]] = 1  
    bloomBitVector[parallel_set4[i]] = 1  
    bloomBitVector[parallel_set5[i]] = 1  
    bloomBitVector[parallel_set6[i]] = 1
```



Funzione hash "h5"

```
def h5(string):  
    # Definiamo parallelamente la lista result che contiene i valori unicode dei caratteri di stringa  
    result = Parallel(n_jobs=numThreads)(delayed(ord)(string[i]) for i in range(0, len(string)))  
  
    vanderMatrix = np.vander(result, n) # Creazione di una matrice di Vandermonde basata sugli elementi  
    sumElements = 0  
    for j in range(0, len(vanderMatrix)): # Tutti gli elementi della matrice vengono sommati  
        for i in range(0, len(vanderMatrix[0])):  
            sumElements = sumElements + (vanderMatrix[j][i]) % 10  
    return sumElements % n
```

Fase di filtraggio

```
Parallel(n_jobs=numThreads)(delayed(matchingCheck)(extended_set[i], bloomBitVector)  
                             for i in range(0, len(extended_set)))
```



Confronto delle prestazioni

n = 500000;
numThreads = 8;

Fase di inizializzazione:

$$\text{Speedup} = \frac{T_S}{T_P} = \frac{158.049}{62.041} = 2.5475$$

Funzione h5:

$$\text{Speedup} = \frac{T_S}{T_P} = \frac{157.983}{60.871} = 2.5954$$

Tempo di matching:

$$\text{Speedup} = \frac{T_S}{T_P} = \frac{5.371}{4.907} = 1.0946$$

Esecuzione complessiva:

$$\text{Speedup} = \frac{T_S}{T_P} = \frac{163.421}{66.948} = 2.4410$$

Confronto delle prestazioni

Tempi di esecuzione con n = 500000	
Sequenziale	163.420624
nThreads = 2	96.676440
nThreads = 4	74.713442
nThreads = 6	69.186874
nThreads = 8	66.948126
nThreads = 10	74.268388