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: ***"words_to_filter.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

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

```python
def h1(string):
    sumUnicode = 0

    for i in string:
        sumUnicode = sumUnicode + ord(i)

    return sumUnicode % n
```

```python
def h2(string):
    productUnicode = 1

    for i in string:
        productUnicode = productUnicode * ord(i)

    return productUnicode % n
```

```python
def h3(string):
    mixUnicode = 1

    for i in string:
        mixUnicode = (mixUnicode * ord(i)) + (22543 * len(i))

    return mixUnicode % n
```

Funzioni hash implementate

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

```python
def h5(string):
    result = []

    for i in range(0, len(string)):
        result.append(0)

    for j in range(0, len(string)):  # Conversione di ciascun cara
        result[j] = ord(string[j])

    vanderMatrix = np.vander(result, n)  # Creazione di una matric
    sumElements = 0
    for j in range(0, len(vanderMatrix)):  # Tutti gli elementi de
        for i in range(0, len(vanderMatrix[0])):
            sumElements = sumElements + (vanderMatrix[j][i]) % 10
    return sumElements % n
```

```python
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 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
```

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 str
    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 u
    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 = 30000;

numThreads = 8;

TEMPI DI ESECUZIONE	
CASO SEQUENZIALE	CASO PARALLELO
9.266271829605103	4.503967571258545

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

Confronto delle prestazioni

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