PC-2019/20 Q-grams in Java threads

Matteo Marulli
E-mail address
matteo.marulli@stud.unifi.it

Matteo Gemignani
E-mail address
matteo.gemignani@stud.unifi.it

Abstract

In questo report, si descrive un programma java che genera bigrammi e trigrammi in due modi differenti: sequenziale e parallelo. Sarà descritta la logica dei due approcci e saranno confrontati i risultati ottenuti.

1. Introduzione

Un q-gramma è una sequenza di q *items* dato un testo.

Gli *items* possono essere lettere, parole, sillabe o fonemi.

In informatica i q-grammi sono regolarmente impiegati in criptografia per condurre attacchi a cifrari monoalfabetici come il *cifrario di Cesare*, nella NLP per la word prediction o anche per gli errori di spelling.

In questo progetto ci si focalizzerà solo sui bigrammi, tuttavia il programma scritto può analizzare trigrammi, quadrigrammi, ecc... impostando il valore del parametro q in modo che sia $q \leq 2$.

Nella seconda sezione saranno descritti i dataset di testing usati per testare le due versioni, nella terza sezione saranno analizzate l' implementazioni del programma sequenziale e parallelo, in fine nella quarta sezione saranno commentati i risultati ottenuti.

2. Dataset di testing

Per testare i programmi si è costruito un dataset composto da 9 libri presi dal sito "Progetto Gutenberg".

Ogni libro pesa all'incirca 360Kb e da questi 9 libri si sono costruiti i seguenti dataset ripetendoli:

- 101 (35.5Mb)
- 51 (17.9Mb)
- 41 (14.4 Mb)
- 31 (10.9Mb)
- 21 (7.38Mb)
- 11 (3.87Mb)

3. Implementazioni

In questa sezione vengono spiegate le versioni sequenziale e parallela per l'analisi dei q-grammi da testo nel linguaggio Java.

Partiremo dalla versione sequenziale perché più semplice da spiegare.

3.1. Implementazione sequenziale

La versione sequenziale è un programma Java che si compone di due classi:

- `BookDataset`
- `Qgramma`

`BookDataset` è una classe che ha il compito di recuperare i libri salvati in una cartella interna al progetto.

Una volta che questi libri sono stati recuperati, vengono resi disponibili come una lista di stringhe tramite il metodo `getDataset()`.

```
public List<String> getDataset() throws
    IOException {
   String path = "/home/matteo/
       JavaWorkspace/ParallelComputing/
       Qgrammi/datasetBook";
   for (int i = 0; i <= numBook; i++) {
    FileReader fileBook;
    fileBook = new FileReader(path+"/
        book_" + i + ".txt");

    BufferedReader readerBook;
    readerBook = new BufferedReader(
        fileBook);

    StringBuilder book = new
        StringBuilder();
    String line;
    while (true) {
     line = readerBook.readLine();
     if (line == null)
      break;
     book.append(line+"\n");
    }
    dataset.add(book.toString());
   }

   return dataset;
  }
```

La classe `Qgram` ha il compito di analizzare i q-grammi di un testo.

`Qgram` ha un metodo chiamato `qgrams()` che accetta in input una stringa con lettere in minuscolo di lunghezza n e itera su di essa usando una finestra scorrevole di dimensione fissa q.

Se la finestra trova un q-gramma allora, si procede a verificare la sua presenza nella hashmap, nel quale il q-gramma è salvato come chiave.

Se è già presente il suo campo valore viene aggiornato aumentandolo di 1, altrimenti lo si inserisce e il suo valore viene impostato ad 1.

Se la finestra scorrevole trova un sequenza che contiene spazi o caratteri speciali, allora avanza verso destra.

La struttura dati Hasmap ci consente di salvare e aggiornare i g-grammi trovati in modo efficiente in quanto tali operazioni hanno un costo costante $\mathbf{O}(1)$; inoltre si presta bene a riportare a rappresentare i q-grammi e le loro frequenze assolute trovate nel testo.

```
public HashMap<String, Integer> qGrams(
    String text) {
   String tupla; //tupla == window
   for (int i = 0; i < text.length() - q
       + 1; i++) {
    tupla = text.substring(i, i + q);
    if (onlyChar(tupla)) {
     if (grams.containsKey(tupla)) {
      grams.replace(tupla, grams.get(
          tupla) + 1);
     } else {
      grams.put(tupla, 1);
     }
    }
   }
   return grams;
  }
```

La finestra per decidere se scorrere a destra usa un metodo chiamato un `onlyChar`.

Il metodo prende in input la sotto stringa analizzata dalla finestra scorrevole e restituisce un valore booleano.

Se la stringa è formata solo da lettere dell'alfabeto in minuscolo allora il metodo restituisce true e quindi il metodo qgrams procede a controllare la presenza/aggiornamento del q-gramma nella hashmap, altrimenti ritorna false quindi il metodo `qgrams` fa scorre la finestra a destra di una posizione.

```
private boolean onlyChar(String tupla) {
   int i=0;
   while(i<q){
    int symNum = (int) tupla.charAt(i);
    if (symNum == 224 || symNum == 232
        || symNum == 233 ||symNum ==
        236|| symNum == 242)
     i++;
    else if (symNum >= 97 && symNum <=
        122)
     i++;
    else
     return false;
   }
```

```
    return true;
  }
```

3.2. Implementazione parallela

Nella programmazione parallela e concorrente in java bisogna predisporre delle classi che vogliamo che eseguano un task in modo parallelo, prima di java 8 si era soliti scrivere le classi thread implementando l'interfaccia Runnanble. Da java 8 in poi sono state introdotte molti novità per scrivere i thread come i Callable che descrivono task che restituiscono un risultato a differenza di Runnable. Usando questi nuovi strumenti, possiamo riscrivere la classe che analizza i q-grammi nel seguente modo:

```
public HashMap<String, Integer> call()
    throws Exception {
      String tuple;
      for (int i = 0; i < text.length() -
           q + 1; i++) {
         tuple = text.substring(i, i + q)
             ;
         if (onlyLetters(tuple)) {
            if (qGrams.containsKey(tuple)
                ) {
               qGrams.replace(tuple,
                   qGrams.get(tuple) + 1)
                   ;
            } else {
               qGrams.put(tuple, 1);
            }
         }
      }
      return qGrams;
   }
```

La nuova classe si chiama ora ParallelQGrams e ha tutti gli stessi metodi della classe Qgrams, va osservato che il metodo che si chiamava nella versione sequenziale qGrams, adesso si chiama call() e non prende in input nessuna stringa, questo perché si deve rispettare l'interfaccia Callable. Per la versione parallela è stato deciso di applicare la tecnica della *data decomposition* per distribuire il lavoro ai threads, cioè ogni libro è stato diviso equamente per ogni thread. Un libro viene visto come una stringa e viene usata la strategia blockwise per dividere in modo equo il carico di lavoro tra i thread come viene riportato in Figura 1.

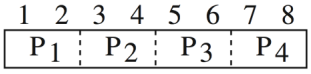

Figure 1. Data distributions: blockwise

Tale strategia viene impiegata nel metodo `divideText` della classe `MasterQgrams` che ha 3 importanti campi:

- un oggetto di tipo executor per eseguire i task;
- una lista di oggetti di tipo `parallelQGrams`;
- un intero che rappresenta il numero di thread richiesto chiamato processors.

L'oggetto esecutore dovrà eseguire i `parallelQgrams` che sono oggetti che implementano l'interfacci `Callable<HashMap<String, Integer>>` ma prima di eseguire tali oggetti, il Master deve prepara il lavoro a questi thread usando la strategia blockwise.

```
private void divideText(String text) {
   double textLen = text.length();
   int i, j;
   double work = Math.ceil(textLen /
       processors);
   i = 0;
   j = (int) work;
   String subText = null;
   while (j < textLen) {
     while (text.charAt(j) != ' ') {
       j++;
     }
     subText = text.substring(i, j);
     i = j;
     j = (int) (i + work);
     taskQGrams.add(new ParallelQgrams(
         subText, q));
   }
   subText = text.substring(i, (int)
       textLen);
   taskQGrams.add(new ParallelQgrams(
       subText, q));
```

```
}
```

Dopo aver fatto la suddivisione del lavoro, per analizzare i q-grammi di un testo, bisogna invocare il metodo `qGrams` di `MasterQgram`.

Il metodo prende in input una stringa con lettere in minuscolo e ritorna una `Map<String, Integer>`, che rappresenta i q-grammi esattamente come le hashmap nella versione sequenziale.

Tuttavia la `Map` che viene restituita è di tipo `ConcurrentHashMap<String, Integer>()`; tale struttura è thread-safe pertanto qualunque sia il numero di thread coinvolti che usano le operazioni di questa map, gli elementi memorizzati in essa non vengono corrotti.

Infatti, se si osserva il codice, ogni thread è invitato a fondere la propria HashMap locale con la `ConcurrentHashMap<String, Integer>()`. Pertanto la Map in caso 2 o più thread cercassero di fondersi con lei, verranno bloccati per garantire un accesso mutuale ad essa.

Per riassumere il metodo `qGrams` di `MasterQgram` esegue questi passi per trovare i q-grammi di un testo:

1. suddivide la stringa in modo equo tra i vari
2. avvia tutti i thread per l'analisi dei q-grammi
3. fonde le hashmap locali di ogni thread nella map concorrente globale
4. restituisce la map globale

```
1  public Map<String, Integer> qGrams(String
       text)
           throws InterruptedException,
               ExecutionException {
3      divideText(text);
       List<Future<HashMap<String, Integer
           >>> resultsQgrams = exec.
           invokeAll(taskQGrams);
5
       Map<String, Integer> mergeAll = new
            ConcurrentHashMap<String,
           Integer>();
7      for (Future<HashMap<String, Integer
           >> result : resultsQgrams) {
           result.get().forEach((k, v) ->
               mergeAll.merge(k, v, Integer
               ::sum));
9      }
       taskQGrams.clear();
11     return mergeAll;
   }
```

4. Analisi delle performance

4.1. Hardware

Per condurre le analisi delle prestazioni è stata usata una macchina MSI con le seguenti specifiche:

- **Processore**: Intel® Core™ i7-10750H CPU @ 2.60GHz 6 core
- **Ram**: 16GB DDR4 3200Mhz
- **Hardisk**: SSD 1TB

4.2. Parallelo vs Sequenziale

L'analisi è stata condotta eseguendo sulle diverse versioni i dataset discussi nella sezione 2.

E' stato fissato $q = 2$ per ricavare dei bigrammi dai testi. Come possiamo notare dallo speedup mostrato in Figura 2, con 12 thread il programma è 3 volte più veloce della versione sequenziale.

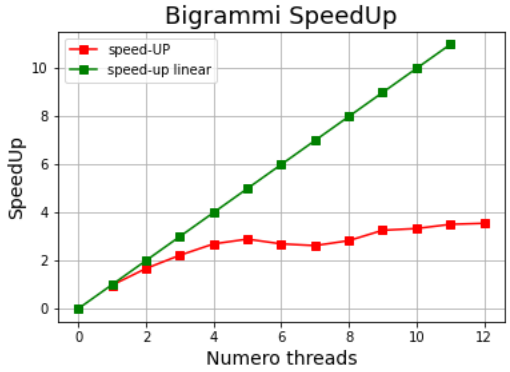

Figure 2. SpeedUp della versione parallela del programma in java

Si nota però che lo speedup inizia a divergere dallo speedup lineare già a partite da 3

thread e, man mano che si aumentano i thread, tale divergenza diventa sempre più netta.

Osservando lo speedup ottenuto, si ritiene che il programma parallelo soffra del fenomeno del false sharing, problema che in Java non è possibile risolvere facilmente come in C++ o in un altro linguaggio a basso livello.