Parallel Computing Final
BigramTrigramGenerator con Java, Java Threads & C++ Threads

Edoardo D'Angelis
Università degli studi di Firenze
edoardo.dangelis@stud.unifi.it

Mirco Ceccarelli
Università degli studi di Firenze
mirco.ceccarelli@stud.unifi.it

Abstract

L'obiettivo di questo progetto è quello di calcolare tutte le occorrenze di bigrammi e trigrammi di lettere presenti all'interno di un testo. In questa relazione verranno presentati e successivamente confrontati tre approcci diversi: uno sequenziale (in Java) e due paralleli (con Java Thread e C++ Thread).

1. Introduzione

Un algoritmo che ricerca bigrammi e trigrammi di lettere o, più in generale, un algoritmo che esegue delle computazioni indipendenti su ciascuna riga di un file testo fa parte della classe di problemi che si prestano meglio alla parallelizzazione: due o più thread possono lavorare nello stesso momento su porzioni del testo differenti, per poi unire tutti i risultati alla fine.

1.1. Bigrammi & Trigrammi

In generale un *n-gramma* è una sottosequenza di n elementi di una data sequenza; gli elementi in questione possono essere sillabe, lettere, parole, ecc [1]. Per questo progetto ci siamo occupati di n-grammi di lettere. In particolare:

- Una sequenza di due lettere si dice *bigramma* (o *digramma*)
- Una sequenza di tre lettere si dice *trigramma*

1.2. Descrizione del dataset

Il dataset utilizzato è stato estratto dal database del *Progetto Gutenberg*, una biblioteca digitale di eBook di pubblico dominio, liberamente riproducibili e scaricabili [2].

Il testo scelto è *"La scienza in cucina e l'arte di mangiar bene"* di Pellegrino Artusi [3]. Per eseguire i vari esperimenti, il testo è stato accorciato ed allungato in modo da ottenere file di dimensioni diverse: 50kb, 100kb, 200kb, 500kb, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB.

1.3. Specifiche Hardware

Per questi esperimenti è stato utilizzato un MacBook Pro (Retina, 13-inch, Mid 2014) con le seguenti specifiche:

- Processore 2,6 GHz Dual-Core Intel Core i5
- Memoria 8 GB 1600 MHz DDR3
- Memoria Flash Storage 128 GB

2. Implementazione

Sono state eseguite tre implementazioni, una in maniera sequenziale in linguaggio Java e due in maniera parallela rispettivamente con Java Thread e C++ Thread.

2.1. Versione Sequenziale in Java

La versione sequenziale dell'algoritmo è piuttosto semplice. Per prima cosa è stata creata una classe, *LinesExtractor*, che prende in input il nome del file *txt* ed espone un metodo, *extractLines*, che legge il file e restituisce una lista di tutte le righe del testo, escludendo per semplicità righe vuote e caratteri speciali. L'estrazione delle righe avviene tramite il metodo *readAllLines* della classe *Files* della libreria standard di Java. La rimozione di caratteri speciali è stata eseguita con un'espressione regolare.

```
public List<String> extractLines() {

  List<String> finalLines = new ArrayList<>();
  List<String> lines = null;
  String regex = "[^A-Za-z0-9' ]";

  FileSystem fs = FileSystems.getDefault();
  lines = Files.readAllLines(fs.getPath("", filename));
  lines.forEach(line -> {
    if (!line.isBlank() && !line.isEmpty()) {
      line = line.toLowerCase().replaceAll(regex, "");
      finalLines.add(line);
    }
  });

  return finalLines;
}
```

1. Esempio di estrazione di righe da un file in Java

La funzione principale prende in input la lista di righe ed estrae una nuova lista contenente tutte le parole al suo interno. L'estrazione di parole è stata implementata con un'altra espressione regolare (\\s+).

```
LinesExtractor extractor = new LinesExtractor(filename);
List<String> lines = extractor.extractLines();

List<String> words = new ArrayList<>();
lines.forEach(line -> {
  words.addAll(Arrays.asList(line.split("\\s+")));
});

words.removeAll(Collections.singleton(""));
```

2. Esempio di estrazione di parole in Java

A questo punto vengono calcolati i bigrammi o i trigrammi su ognuna delle parole estratte, utilizzando il metodo *substring* della classe *String* della libreria standard di Java. Il tempo di esecuzione viene misurato con la funzione *currentTimeMillis()* della libreria standard *System*.

```
long start = System.currentTimeMillis();

int n = 2; // 2: bigrammi, 3: trigrammi

List<String> ngrams = new ArrayList<>();
for (String word: words) {
  if (word.length() >= n) {
    for (int i = 0; i < word.length() - (n - 1); i++) {
      ngrams.add(word.substring(i, i + n));
    }
  }
}

long end = System.currentTimeMillis();
```

3. Esempio di ricerca di bigrammi/trigrammi in Java

2.2. Versione Parallela con Java Thread

Per la versione parallela con Java Thread è stata utilizzata la stessa classe *LinesExtractor* (come per la versione sequenziale) per estrarre le righe dal testo, ed è uguale anche la procedura con cui vengono estratte le parole.

Per il conteggio dei bigrammi o trigrammi è stata utilizzata una variabile *globalCounter* di tipo *AtomicInteger*, condivisa tra tutti i thread, in modo che ogni thread possa aggiornarla evitando inconsistenze.

È stata quindi creata una classe, *NGramThread*, che estende la classe *Thread* e il suo costruttore riceve in ingresso:

- *globalCounter*: riferimento alla variabile di tipo *AtomicInteger*, rappresenta il contatore globale del numero di *n-grammi* trovati
- *words*: l'intera lista di parole
- *n*: intero che specifica se vengono trattati bigrammi (*n=2*) oppure trigrammi (*n*=3)
- *start* e *stop*: due interi che indicano gli indici di partenza e arrivo del range di parole su cui ciascun thread lavora

Appena il thread viene lanciato, viene eseguita la funzione *run()* che si occupa del calcolo degli *n-grammi*.

La classe presenta anche un intero privato, *nGramCounter*, inizializzato a zero e che viene incrementato ad ogni *n-gramma* trovato. Una volta calcolati tutti gli *n-grammi* sul range di parole, si aggiorna il valore dell'intero atomico *globalCounter* tramite la funzione *addAndGet* della classe *AtomicInteger* nel package *java.util.concurrent.Atomic*, aggiungendo in maniera atomica il valore di *nGramCounter*. Questo viene fatto per ridurre al minimo il numero di operazioni atomiche da eseguire in ogni thread.

```
public class NGramThread extends Thread {

  private int n, start, stop;
  private List<String> words;
  private AtomicInteger globalCounter;
  private int nGramCounter = 0;

  public void run() {
    if (stop > words.size()) stop = words.size();
    List<String> ngrams = new ArrayList<>();
    for (int i=start; i<stop; i++) {
      String word = words.get(i);
      for (int j = 0; j < word.length() - (n-1); j++) {
        ngrams.add(word.substring(j, j+n));
        nGramCounter++;
      }
    }
    globalCounter.addAndGet(nGramCounter);
  }
}
```

4. Esempio di ricerca di bigrammi/trigrammi con Java Thread

Nel main vengono quindi lanciati un numero variabile di thread, e ne viene fatto il *join* per aspettare che terminino l'esecuzione. Anche in questo caso il tempo di esecuzione viene misurato con la funzione *currentTimeMillis()* della libreria standard *System*.

```
AtomicInteger gCounter = new AtomicInteger(0);
List<NGramThread> threads = new ArrayList<>();
int blockSize = words.size()/numThreads+1;
int n = 2; // 2: bigrammi, 3: trigrammi
int i = 0;

long start = System.currentTimeMillis();

while (i < numThreads) {
  int s = i*blockSize; // indice partenza
  int e = (i+1)*blockSize; // indice arrivo
  threads.add(new NGramThread(gCounter,words,n,s,e));
  threads.get(i).start();
  i++;
}

for (NGramThread worker: threads) {
  worker.join();
}

long end = System.currentTimeMillis();
```

5. Lancio di Java Thread per ricerca di bigrammi/trigrammi

2.3. Versione Parallela con C++ Thread

Per la versione parallela con C++ Thread è stata creata una classe *LinesExtractor*, equivalente della versione in Java, che prende in input il nome del file *txt* ed espone un metodo, *extractLines*, che legge il file e restituisce un vettore di stringhe, una per ogni riga estratta dal testo (escludendo per semplicità righe vuote e caratteri speciali). L'estrazione delle righe avviene tramite la funzione *getline* della classe *ifstream* della libreria standard del C++.

Anche in questo caso la rimozione di caratteri speciali è stata eseguita con un'espressione regolare.

```
vector<string> extractLines() {

    vector<string> lines;
    string line;

    ifstream file(filename);

    if (!file) {
        throw runtime_error("Could not open file!");
    }

    regex reg("[^A-Za-z0-9' ]");

    while (getline(file, line)) {
        transform(line.begin(), line.end(), ::tolower);
        string cleaned = regex_replace(line, reg, "");
        if (!cleaned.empty()) {
            lines.push_back(cleaned);
        }
    }

    file.close();
    return lines;
}
```

6. Esempio di estrazione di righe da un file in C++

La funzione principale prende in input la lista di righe ed estrae una nuova lista contenente tutte le parole al suo interno. L'estrazione di parole è stata implementata utilizzando la classe *stringstream* e l'iteratore *istream_iterator*.

```
LinesExtractor extractor(filename);
vector<string> lines = extractor.extractLines();
vector<string> words;

for (auto &line: lines) {
  stringstream ss(line);
  istream_iterator<string> begin(ss), end;
  vector<string> lineWords(begin, end);

  for (auto &word: lineWords) {
    if (!word.empty()) {
      words.push_back(word);
    }
  }

}
```

7. Esempio di estrazione di parole in C++

Per il conteggio dei bigrammi o trigrammi è stato utilizzata una variabile *globalCounter* di tipo *atomic_int*, condivisa tra tutti i thread, in modo che ogni thread possa aggiornarla evitando inconsistenze.

La classe *thread* della libreria standard del C++ (*std::thread*), introdotta con C++11, richiede nel costruttore un oggetto *callable* (riferimento a funzione o laMBda expression), porzione di codice che viene eseguita non appena il thread viene lanciato. È stata quindi creata una funzione *run()* per il calcolo degli *n-grammi* che riceve in ingresso:

- *globalCounter*: riferimento alla variabile di tipo *atomic_int*, rappresenta il contatore globale del numero di *n-grammi* trovati
- *words*: l'intero vettore di parole
- *n*: intero che specifica se vengono trattati bigrammi (*n*=2) oppure trigrammi (*n*=3)
- *start* e *stop*: due interi che indicano gli indici di partenza e di arrivo del range di parole su cui ciascun thread lavora

La funzione *run()* al suo interno instanzia un intero, *nGramCounter*, inizializzato a zero e che viene incrementato ad ogni *n-gramma* trovato. Una volta calcolati tutti gli *n-grammi* sul range di parole, si aggiorna il valore dell'intero atomico *globalCounter* tramite la funzione *fetch_add* della libreria *std::atomic*, aggiungendo in maniera atomica il valore di *nGramCounter*. Come per la versione parallela con Java Thread, questo viene fatto per ridurre al minimo il numero di operazioni atomiche da eseguire in ogni thread.

```
void run(atomic_int& counter, int id, int n,
         vector<string> words, int start, int stop) {

  int nGramCounter = 0;

  if (stop > words.size()) stop = words.size();

  vector<string> ngrams;
  for (int i=start; i<stop; i++) {
    string word = words[i];
    if (word.length() >= n) {
      for (int j=0; j<word.length()-(n-1); j++) {
        ngrams.push_back(word.substr(j, n));
        nGramCounter++;
      }
    }
  }

  counter.fetch_add(nGramCounter, memory_order_relaxed);
}
```

8. Esempio di funzione lanciata da un thread in C++

Nel main vengono quindi lanciati un numero variabile di thread, e ne viene fatto il join per aspettare che terminino l'esecuzione.

Il tempo di esecuzione è stato misurato utilizzando il metodo *now()* della classe *steady_clock* della libreria *chrono*. La classe *steady_clock* rappresenta un *clock* di tipo *monotonic* e il suo funzionamento è simile a quello di un *wall-clock*, ideale per misurare intervalli di tempo in un thread (a differenza del tempo di CPU, che misurerebbe il tempo totale speso in tutti i thread).

```cpp
int blockSize = words.size() / numThreads + 1;
atomic_int globalCounter(0);

int n = 2; // 2: bigrammi, 3: trigrammi

auto start = chrono::steady_clock::now();

std::vector<thread> threads(numThreads);
for (int i = 0; i < numThreads; i++) {
  int s = i*blockSize; // indice partenza
  int e = (i+1)*blockSize; // indice arrivo
  threads[i] = thread(run, ref(globalCounter),
                      i, n, words, s, e);
}
for (auto &th: threads) {
  th.join();
}

auto end = chrono::steady_clock::now();
chrono::duration<double> elapsedSeconds = end - start;
```

9. Lancio di C++ Thread per la ricerca di bigrammi/trigrammi

3. Esperimenti & Risultati

Per ogni esperimento sono state eseguite cinque iterazioni in modo da ottenere un misura più accurata dei risultati. Tutti i grafici presentati di seguito sono stati generati con uno script Python utilizzando la libreria *matplotlib*.

L'accuratezza dei risultati ottenuti è stata valutata tramite metrica dello *Speedup* (S), definito come il rapporto tra il tempo di esecuzione sequenziale (t_s) e il tempo di esecuzione parallelo (t_p): $S = \frac{t_s}{t_p}$.

3.1. Valutazione dello speedup con Java Thread

Viste le risorse hardware, è stato scelto di eseguire gli esperimenti con 2 o 4 thread. Una volta scelto il testo *"La scienza in cucina e l'arte di mangiar bene"* di Pellegrino Artusi, è stato quindi misurato lo *speedup* al variare delle dimensioni del file di testo (50kb, 100kb, 200kb, 500 kb, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB).

Come si può notare dalle immagini lo speedup risulta essere maggiore di 1 oltre una certa soglia ($\approx$ 100kb per i bigrammi e $\approx$ 500kb per i trigrammi).

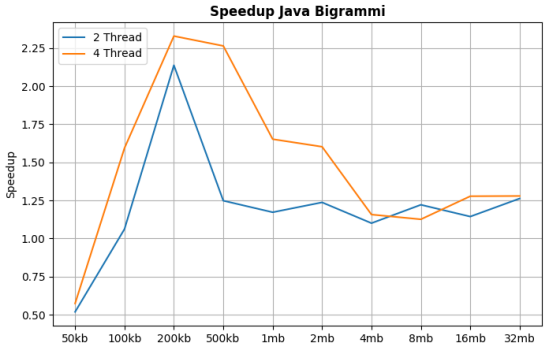

Figura 1. Speedup bigrammi con Java Thread

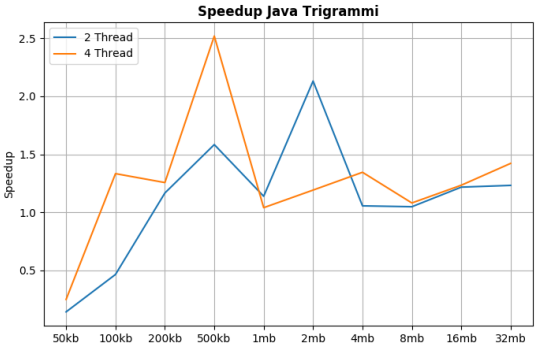

Figura 2. Speedup trigrammi con Java Thread

3.2. Valutazione dello speedup con C++ Thread

Anche in questo caso è stato scelto di utilizzare 2 o 4 thread. Come si può notare dalle immagini, dopo una certa soglia ($\approx$ 16 MB) c'è un aumento significativo dello *speedup*. È ragionevole pensare che aumentando ulteriormente la dimensione del file, si possa ottenere uno *speedup* ancora maggiore.

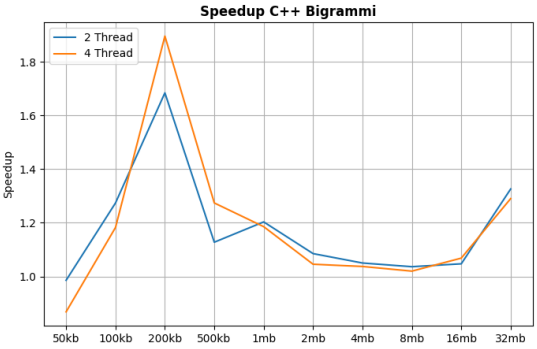

Figura 3. Speedup Bigrammi con C++ Thread

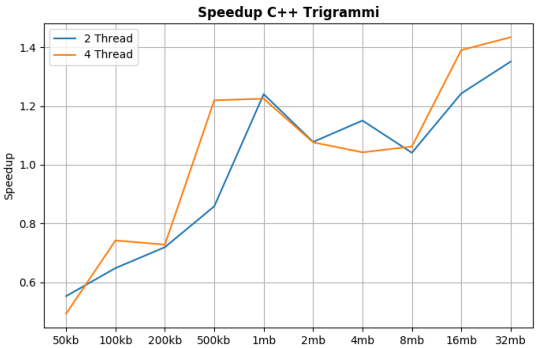

Figura 4. Speedup Trigrammi con C++ Thread

3.3. Confronto sui tempi di esecuzione con Java Thread

Per avere dei risultati più accurati, è stato eseguito un confronto anche sui tempi di esecuzione con Java Thread rispetto alla versione sequenziale.

Come si può chiaramente notare dai risultati ottenuti, da una certa dimensione in poi ($\approx$ 4MB) gli approcci paralleli risultano sempre più efficienti rispetto all'approccio sequenziale. Non sono state riscontrare differenze sostanziali tra l'utilizzo di 2 o 4 thread.

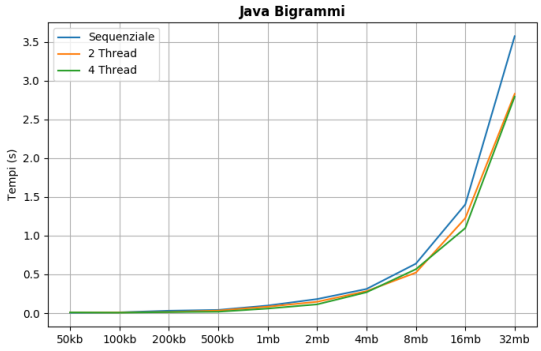

Figura 5. Confronto tempi di esecuzione bigrammi Java

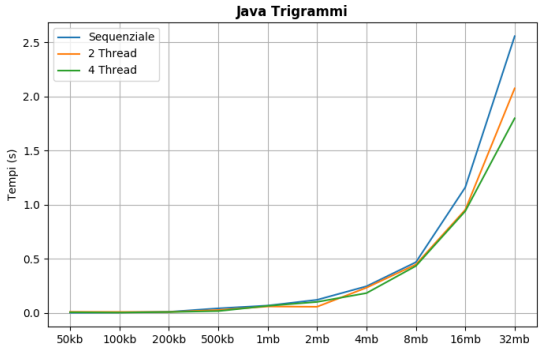

Figura 6. Confronto tempi di esecuzione trigrammi Java

3.4. Confronto sui tempi di esecuzione con C++ Thread

Allo stesso modo è stato eseguito un confronto anche sui tempi di esecuzione con C++ Thread rispetto alla versione sequenziale. Come si può notare dai risultati ottenuti, molto simili ai precedenti, anche in questo caso da una certa dimensione in poi ($\approx$ 8 MB) gli approcci paralleli risultano sempre più efficienti rispetto all'approccio sequenziale.

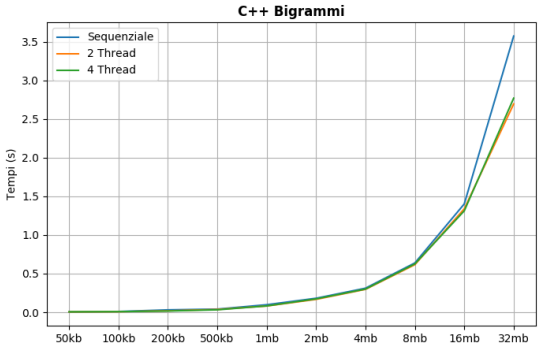

Figura 7. Confronto tempi di esecuzione Bigrammi C++

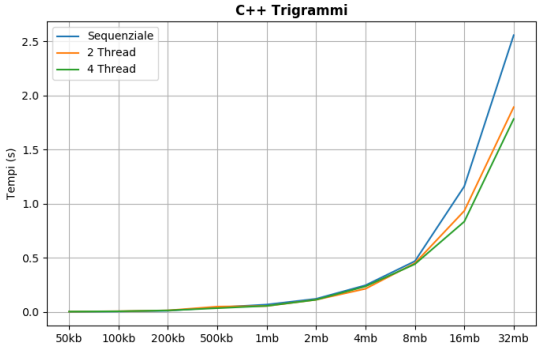

Figura 8. Confronto tempi di esecuzione Trigrammi C++

4. Conclusione

Come era possibile aspettarsi per file di grosse dimensioni l'approccio parallelo è sempre migliore di quello sequenziale: più lungo è il testo è maggiore sarà lo *speedup* ottenuto.

In generale l'implementazione in Java è risultata più veloce, considerando anche il fatto che non negli esperimenti non è stato incluso il tempo di estrazione di righe e parole, che risulta essere significativamente maggiore (circa 3 volte) con C++ rispetto al Java.

Bibliografia

[1] Wikipedia. N gramma. `https://it.wikipedia.org/wiki/N-gramma`.

[2] Progetto Gutenberg. `https://www.gutenberg.org`.

[3] Pellegrino Artusi. La scienza in cucina e l'arte di mangiar bene. `http://www.gutenberg.org/ebooks/59047`.