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

Abstract

L'obiettivo di questo progetto è quello di calcolare le occorrenze dei bigrammi e trigrammi di lettere con tre approcci diversi: uno sequenziale (in Java) e due paralleli (rispettivamente con Java Thread e C++ Thread). I risultati vengono espressi in termini di tempo computazionale complessivo e Speedup.

1. Introduzione

L'algoritmo, che viene utilizzato per la ricerca di bigrammi e trigrammi di lettere all'interno di un testo di lunghezza variabile, si presta bene alla parallelizzazione in quanto due o più thread possono lavorare nello stesso momento su porzioni di testo differenti.

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 gli esperimenti il testo è stato accorciato e 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. È stata utilizzata una classe *"LinesExtractor"* che prende in input il nome del file txt ed espone un metodo *extractLines* che restituisce una lista di tutte le righe del testo escludendo per semplicità righe vuote e caratteri speciali. L'estrazione delle righe avviene tramite la funzione *readAllLines* della classe *"Files"* standard del Java.

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

  try {
    FileSystem fs = FileSystems.getDefault();
    lines = Files.readAllLines(fs.getPath("", filename));
  } catch (IOException e) {
    System.out.println("Failed to read " + 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 questa lista di righe ed estrae la lista di tutte parole al suo interno tramite un'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

A questo punto vengono calcolati i bigrammi o i trigrammi per ognuna delle parole estratte.
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

2.2. Versione Parallela con Java Thread

Per la versione parallela con Java Thread, la classe *"LinesExtractor"* è la stessa della versione sequenziale ed è uguale anche la metodologia con cui vengono estratte le parole.
Per il conteggio dei bigrammi o trigrammi è stato utilizzato un *AtomicInteger* in modo che ogni thread possa aggiornarlo evitando inconsistenze.
È stata creata una classe *NGramThread* che estende la classe *Thread* e il suo costruttore prende in ingresso:

- *globalCounter*: riferimento alla variabile *AtomicInteger*
- *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 il thread lavora

La classe presenta anche un intero *nGramCounter* inizializzato a zero e che viene incrementato ad ogni n-gramma trovato. Una volta calcolato tutti gli n-grammi 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*.

```
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);
      if (word.length() >= n) {
        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 lanciati un numero variabile di thread e ne viene fatto il *join* per aspettare che terminino.
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 i = 0;
int n = 2; // 2: bigrammi, 3: trigrammi

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) {
  try {
    worker.join();
  } catch (InterruptedException ignored) {}
}

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 utilizzata la classe *"LinesExtractor"* che prende in input il nome del file txt ed espone un metodo *extractLines* che restituisce un vettore di stringhe estratte 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++.

```
vector<string> extractLines(const string& filename) {

    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 questa lista di righe ed estrae la lista di tutte parole al suo interno tramite la classe *stringstream* e l'iteratore *istream_iterator*.

```
vector<string> lines = extractLines(filename);
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) {
    words.push_back(word);
  }
}
```

7. Esempio di estrazione di parole in C++

Per il conteggio dei bigrammi o trigrammi è stato utilizzato un *atomic_int globalCounter* in modo che ogni thread possa aggiornarlo evitando inconsistenze.
I thread della libreria standard del C++ (*std::thread*) non vengono definiti come classe, ma vengono lanciati con una funzione *run()* che prende in ingresso:

- *globalCounter*: riferimento alla variabile *atomic_int*
- *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 il thread lavora

Viene usato anche un intero *nGramCounter* inizializzato a zero e che viene incrementato ad ogni n-gramma trovato. Una volta calcolato tutti gli n-grammi 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*.

```
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 lanciati un numero variabile di thread e ne viene fatto il *join* per aspettare che terminino.
Il tempo di esecuzione è stato misurato utilizzando il metodo *now()* della classe *steady_clock* della libreria *chrono*.

```
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 5 iterazioni per avere un misura più accurata dei vari risultati.
Tutti i grafici presentati di seguito sono stati generati con uno script Python utilizzando la libreria *"matplotlib"*.
La bontà dei tempi di esecuzione ottenuti è stata valutata tramite la 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

È stato scelto di utilizzare 2 o 4 thread. Dunque date le varie dimensioni del file (50 kb, 100 kb, 200 kb, 500 kb, 1 mb, 2 mb, 4 mb, 8 mb, 16 mb, 32 mb) de *"la scienza in cucina e l'arte di mangiar bene"* di Pellegrino Artusi, è

stato misurato lo *Speedup* al variare delle dimensioni.
Come si può notare dalle immagini lo Speedup risulta essere maggiore di 1 oltre una certa soglia ($\approx$ 100 kb per i bigrammi e $\approx$ 500 kb per i trigrammi).

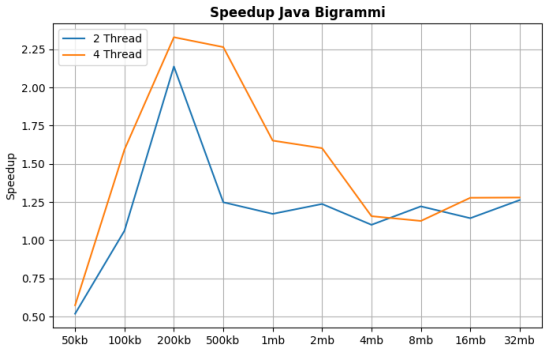

Figura 1. Speedup Bigrammi con Java Thread

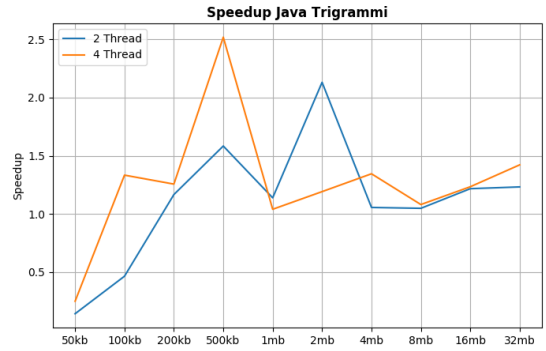

Figura 2. Speedup Trigrammi con Java Thread

3.2. Valutazione dello Speedup con C++ Thread

Anche in questo caso è stato scelto di utilizzare 2 o 4 thread. Dunque date le varie dimensioni del file è stato misurato lo *Speedup* al variare delle dimensioni.
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.

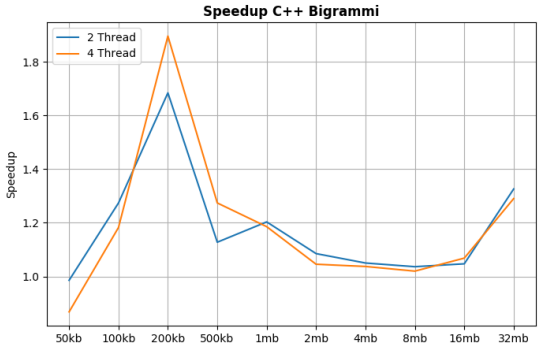

Figura 3. Speedup Bigrammi con C++ Thread

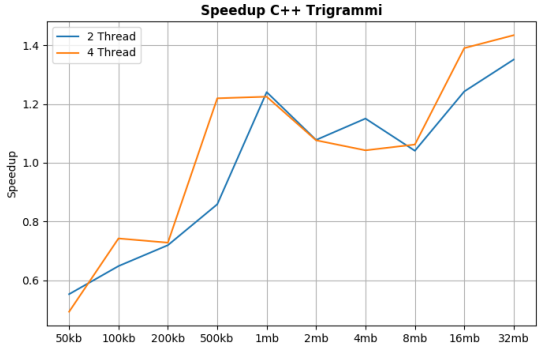

Figura 4. Speedup Trigrammi con C++ Thread

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.
Come si può chiaramente notare dai risultati ottenuti, da una certa dimensione in poi ($\approx$ 2 mb) 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.

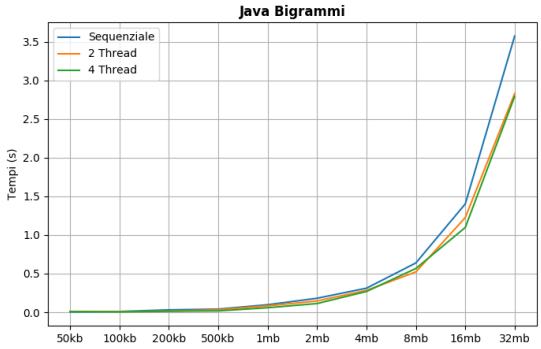

Figura 5. Confronto tempi di esecuzione Bigrammi Java

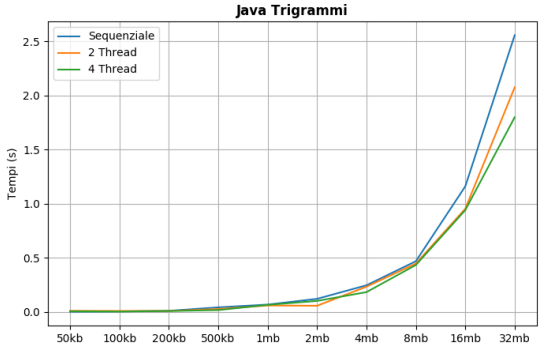

Figura 6. Confronto tempi di esecuzione Trigrammi Java

3.4. Confronto sui Tempi di Esecuzione con C++ Thread

Per avere dei risultati più accurati, è stato eseguito un confronto anche sui tempi di esecuzione.
Come si può chiaramente notare dai risultati ottenuti, da una certa dimensione in poi ($\approx$ 8 mb) 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.

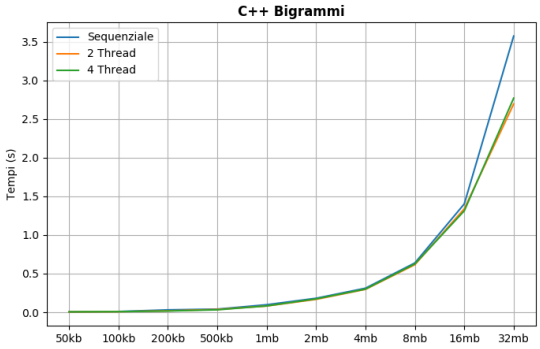

Figura 7. Confronto tempi di esecuzione Bigrammi C++

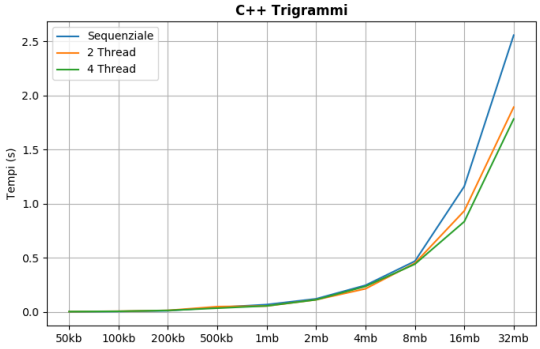

Figura 8. Confronto tempi di esecuzione Trigrammi C++

4. Conclusione

Come era possibile aspettarsi per file di grosse dimensioni l'approccio parallelo è sicuramente migliore di quello sequenziale.
In base ai risultati ottenuti si può anche notare che il tempo di esecuzione per la ricerca di bigrammi e trigrammi è leggermente inferiore per i Java Thread rispetto ai C++ Thread. Considerando anche il fatto che non è stato calcolato 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`.