PC-2019/20 Implementazione parallela in Java del calcolo di bigrammi di parole e lettere

Marco Calamai
6314073
marco.calamai@stud.unifi.it

Elia Mercatanti
6425149
elia.mercatanti@stud.unifi.it

Abstract

Bigrammi sono lettere o parole adiacenti in un testo. Questa relazione descrive l'implementazione parallela in Java dell'algoritmo che calcola le frequenze di bigrammi in un insieme di file di testo.

1. Introduzione

Un n-gramma è una sequenza contigua di n elementi in un testo. Gli elementi possono essere parole, lettere, sillabe etc. Nel progetto discusso in questa relazione sono state prese in considerazione frequenze di bigrammi di lettere e parole adiacenti da un insieme di libri testuali.
Quando si parla di bigrammi, le modalità di selezione dei caratteri o delle parole dal testo variano molto in base all'obiettivo dell'applicazione.
Nel progetto in questione sono state ricercate sequenze di bigrammi di lettere considerando esclusivamente le lettere dell'alfabeto inglese. Come esempio, considerando la frase: "Hello world!", i bigrammi di lettere estratti saranno: "He", "el", "ll", "lo", "ow", "wo", "or", "rl", "ld".
Per quanto riguarda i bigrammi di parole, anche in questo caso sono state considerate come parole esclusivamente sequenze di lettere adiacenti dell'alfabeto inglese.

2. Implementazione sequenziale della ricerca dei bigrammi

L'implementazione prevede che possano essere calcolati bigrammi anche da più testi. È stata quindi predisposta una coda nella quale vengono inseriti tutti i nomi dei file da processare prima dell'esecuzione dell'algoritmo vero e proprio, come mostrato nel codice 1.

```
try (DirectoryStream<Path> stream = Files.newDirectoryStream(Paths.get("Txt"))) {
  for (Path path : stream) {
    if (!Files.isDirectory(path)) {
      txtListMain.add(path.getFileName().toString());
    }
  }
} catch (IOException e) {
  e.printStackTrace();
}
System.out.println("Input txt files: " + txtListMain + "\n");
```

Listing 1. Caricamento nomi files

Una volta caricati i nomi dei file nella coda, inizia l'algoritmo vero e proprio che legge i file una riga alla volta e conta per ogni riga i bigrammi presenti in essa.
Ogni riga ottenuta viene quindi controllata per verificare che contenga almeno un carattere dell'alfabeto, in caso contrario viene scartata. Fatto ciò, vengono conteggiati i bigrammi, ricercando nelle righe lettere o parole che si susseguono. Per fare ciò, sono state usate le classi *Pattern* e *Matcher* utilizzando un'espressione regolare specifica per trovare bigrammi di lettere ("[a-zA-Z]") o parole ("[a-zA-Z]+"). Una volta estratto un bigramma, quest'ultimo viene aggiunto al dizionario dei bigrammi incrementando il numero di occorrenze di uno come mostrato nel codice 2.

```
while ((line = buffer.readLine()) != null) {
  ml = p.matcher(line);
  if(!ml.find()) {
    continue;
  }
  ml = p.matcher(line);
  while(ml.find()) {
    nextToken = ml.group();
    nGram = (prevToken + nextToken);
    dict.merge(nGram, 1, (prev, one) -> prev + one);
    prevToken = nextToken;
  }
}
```

Listing 2. Calcolo bigrammi di lettere

3. Versioni parallele

L'algoritmo prevede una fase di lettura dei file e una fase di calcolo dei bigrammi.
Analizzando mediante un profiler Java l'esecuzione del codice sequenziale si è visto, come riportato in figura 1, che l'algoritmo impiega circa un 10% di tempo sulla lettura del file e il restante 90% circa sul calcolo dei bigrammi. Ci sarà quindi una parte di esecuzione relativa alla lettura che resterà sequenziale e quindi peserà sui tempi di esecuzione finali.

Lo schema implementato è quello del produttore/consumatore, nel quale c'è un solo produttore che legge i files sequenzialmente e in parallelo più consumatori che calcolano i bigrammi sulle stringhe generate dal produttore.

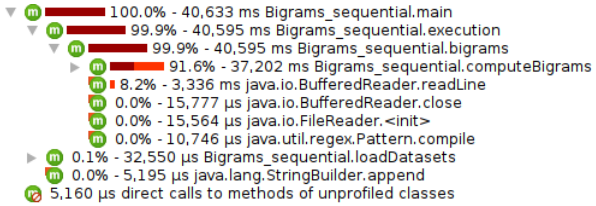

Figura 1. Profiler esecuzione sequenziale

Per gestire la comunicazione tra produttore e consumatori sono state utilizzate:

- Una coda concorrente non bloccante nella quale il produttore inserisce le stringhe da elaborare che i consumatori estraggono.
- Un hash map concorrente, utilizzato come dizionario globale nel quale i consumatori inseriranno il numero delle occorrenze dei bigrammi trovati.

I consumatori sono stati gestiti con un thread pools.

3.1. Implementazione parallela della ricerca dei bigrammi

- Il produttore legge un file alla volta e per ogni file concatena un'insieme di righe in una stringa, controllando che ogni riga contenga almeno una lettera dell'alfabeto inglese, come visto nella versione sequenziale, che passa ai consumatori nella coda concorrente. Per gestire l'a capo tra due stringhe generate dal produttore viene salvato l'ultimo token (lettera o parola) e concatenato con la successiva stringa che il produttore genererà.
  Questa fase è mostrata nel codice 3.

```
while ((this.line = buffer.readLine()) != null) {
  m1 = p2.matcher(line);
  if(!m1.find()) {
    continue;
  }
  this.lastLine = this.line;
  this.mergedLine = this.line;
  for (int i = 0; i < 20; i++) {
    if ((this.line = buffer.readLine()) == null) {
      break;
    }
  m1 = p2.matcher(this.line);
  if(!m1.find()) {
    continue;
  }
  this.lastLine = this.line;
  this.mergedLine = this.mergedLine + " " + this.line;
  }
  notBlockingqueue.add(this.lastToken + " " + this.mergedLine);
  this.m1 = this.p.matcher(this.lastLine);

  if(this.m1.find()) {
    this.lastToken = this.m1.group();
  }
  else {
    this.lastToken = "";
  }
}
```

Listing 3. Produttore bigrammi

- I consumatori estraggono stringhe dalla coda concorrente e per ogni stringa estratta calcolano le occorrenze di bigrammi, allo stesso modo visto nella versione sequenziale, e aggiornano un dizionario HashMap privato. Questi threads continuano ad estrarre elementi dalla coda fino a quando viene estratta la stringa NO_MORE_MESSAGES, la quale contiene un identificatore unico universale che segnala che non ci sono più stringhe da elaborare. Il dettaglio del codice è mostrato in 4.

```
while (true) {
  this.textLine = notBlockingqueue.poll();
  if (this.textLine == null) {
    continue;
  }
  if (this.textLine == NO_MORE_MESSAGES) {
    break;
  }
  this.m1 = this.p.matcher(this.textLine);
  if(this.m1.find()) {
    this.prevToken = this.m1.group();
  }
  while(this.m1.find()) {
    this.nextToken = this.m1.group();
    this.nGram = (this.prevToken + this.nextToken);
    dict.merge(this.nGram, 1, (prev, one) -> prev + one);
    this.prevToken = this.nextToken;
  }
}

```

Listing 4. Consumatore bigrammi

Prima di terminare, il thread consumatore procede con un merge del proprio HashMap privato nell'HashMap globale in modo tale da aggiornare tutte le occorrenze di bigrammi trovate da tutti i threads consumatori. Il codice relativo a quest'ultima fase è mostrato in 5.

```
  for (String name : this.dict.keySet()) {
    String key = name;
    Integer value = this.dict.get(name);
    globalDict.merge(key, value, (prev, update) -> prev + update);
}

```

Listing 5. Fase di merge consumatore

4. Considerazioni

Prima di procedere con i test e relativi risultati ci sono da fare alcune considerazioni.

- Come premessa principale c'è da notare che la fase di lettura del file, affidata al thread produttore risulta sequenziale in quanto lettura da memoria di massa e quindi non parallelizzabile.
- È stato scelto di utilizzare 1 thread produttore e 4 threads consumatori. Sono stati tuttavia eseguiti diversi test variando il numero di threads consumatori e produttori. Il risultato è stato che, come ci si potrebbe aspettare, aumentare il numero di threads produttori non porta ad alcun beneficio ma anzi peggiora il tempo di esecuzione complessivo.
  Per quanto riguarda i consumatori è stato scelto di generarne 4 in quanto la macchina sulla quale sono stati fatti i test presenta una CPU a 4 core e i risultati in termini di tempo di esecuzione anche con più threads consumatori non ha portato rilevanti benefici.

- Per quando riguarda il passaggio del lavoro tra produttore e consumatori è stato scelto di utilizzare una coda non bloccante, in quanto ci si aspetta una contesa media, data anche dal basso numero di threads totali. Inoltre un CAS risulta meno costoso di un'acquisizione del lock.
  Sono stati fatti anche test utilizzando una coda bloccante, ottenendo risultati paragonabili nelle stesse condizioni.

- Il produttore cerca di concatenare in una stringa più righe del file, in modo tale da creare più lavoro per i consumatori quando estraggono dalla coda.
  Il numero di stringhe che il produttore cerca di concatenare è stato scelto pari a 20. Questo rappresenta un compromesso. Un numero troppo basso rallenterebbe il tempo di esecuzione totale del produttore, in quanto dovrebbe ricercare per più stringhe il "lastToken" da concatenare alla stringa successiva e fornirebbe inoltre poco lavoro alla volta ai consumatori, i quali intaserebbero di richieste la coda. Un numero troppo alto invece, porterebbe a troppe operazioni di concatenazione da parte del produttore per generare una stringa, le quali renderebbero il produttore lento per fornire a tutti i consumatori stringhe da consumare. Anche questo valore quindi è stato il risultato di diversi test eseguiti.

5. Test e risultati

I test sono stati eseguiti su CPU quad core i7 3770K.
I testi utilizzati per i test sono stati presi dal sito web https://www.gutenberg.org/. Sono stati scelti 15 libri e per fare in modo da avere un dataset di più grandi dimensioni sono stati copiati questi testi più volte creando datasets di dimensioni complessive pari a 105, 315, 630 e 1260 testi.
Per calcolare i tempi di esecuzione è stato eseguito l'algoritmo 5 volte e fatta una media dei tempi risultanti.
Dai risultati mostrati in figura 2, si può notare come lo speedup massimo raggiunto è di circa 2.7 per i bigrammi di lettere e di poco superiore a 2 per quello di parole. La differenza tra i due è imputabile a due fatti principali:

1. Il produttore risulta più lento nei bigrammi di parole, in quanto impiega più tempo a ricercare l'ultima parola della stringa precedente da concatenare alla stringa successiva, rispetto alla ricerca dell'ultima lettera nei bigrammi di lettere.

2. I consumatori hanno più lavoro da svolgere nel calcolo dei bigrammi di lettere rispetto ai bigrammi di parole, che risulta la parte di codice parallelizzata.

I tempi di esecuzione delle versioni parallele sono penalizzati principalmente dalla parte svolta dal produttore che non è parallelizzata e dall'overhead causato dagli accessi alla coda condivisa.
Si può notare inoltre come lo speedup aumenti quando il dataset diventa più grande, questo è dovuto al fatto che incrementa il carico di lavoro parallelizzabile.
Si è voluto infine fare un altro test, nell'obiettivo di ridurre il tempo di esecuzione del produttore. È stata quindi modificata la parte di codice del produttore facendo in modo che inserisca ogni libro di testo in un'unica stringa mediante la funzione *readString*, aggiungendo quindi nella coda condivisa un testo completo alla volta. Questo ha portato come mostrato in figura 3 a dei tempi di esecuzione generalmente più bassi e uno speedup leggermente migliore per i datasets testati.
Il problema di questa versione è il fatto che con datasets composti da meno di 4 files non vengono sfruttati tutti i threads consumatori e un possibile carico di lavoro sbilanciato tra i vari threads consumatori se sono presenti files di dimensioni molto diverse tra loro.

BIGRAMS

# FILES		Letters	Words
15	**Sequenziale**	1.86	1.55
	Parallelo	0.83	1.15
	Speedup	2.24	1.35
105	**Sequenziale**	12.64	9.5
	Parallelo	5.1	5.42
	Speedup	2.48	1.75
315	**Sequenziale**	39.17	28.5
	Parallelo	15.09	14.8
	Speedup	2.60	1.93
630	**Sequenziale**	77.15	53
	Parallelo	31.11	24.74
	Speedup	2.48	2.14
1260	**Sequenziale**	150.29	117.11
	Parallelo	55.48	52.35
	Speedup	2.71	2.24

Figura 2. Risultati in secondi

BIGRAMS

# FILES		Letters	Words
15	**Sequenziale**	1.61	1.34
	Parallelo	0.76	0.963
	Speedup	2.12	1.39
105	**Sequenziale**	11.4	9.12
	Parallelo	4.19	4.74
	Speedup	2.72	1.92
315	**Sequenziale**	34.19	27.17
	Parallelo	11.76	12.56
	Speedup	2.91	2.16
630	**Sequenziale**	67.41	49.94
	Parallelo	24.33	21.36
	Speedup	2.77	2.34

Figura 3. Risultati in secondi con una stringa per libro di testo

6. GitHub

https://github.com/elia-mercatanti/parallel-bigrams

Nella cartella src sono presenti 4 file. Bigrams_sequential e Bigrams_parallel sono le implementazioni sequenziale e parallela del calcolo dei bigrammi. I file Alternative_bigrams_sequential e Alternative_bigrams_parallel sono l'implementazione dello stesso algoritmo con la modifica della lettura dei files discussa precedentemente.
Per eseguire ogni versione è necessario passare al programma una stringa, "letters" per scegliere di calcolare bigrammi di lettere e "words" per calcolare bigrammi di parole.