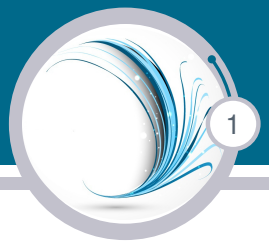
Università degli studi di Firenze
Scuola di Ingegneria

Most Common N-Grams in the English and the Italian Languages

Elena Sesoldi, Giulia Pellegrini
parallel18computing@gmail.com

Febrary 26, 2018

Content



Introduction
Task

- Implementation of an algorithm that computes the number of N-gram occurrences

Introduction
Task

- Implementation of an algorithm that computes the number of N-gram occurrences

- Programming language: Java

Introduction
Task

- Implementation of an algorithm that computes the number of N-gram occurrences

- Programming language: Java
- A sequential implementation

Introduction
Task

- Implementation of an algorithm that computes the number of N-gram occurrences

N = 2 It was all Pride — *word bigrams* → `it-was` `was-all` `all-pride`

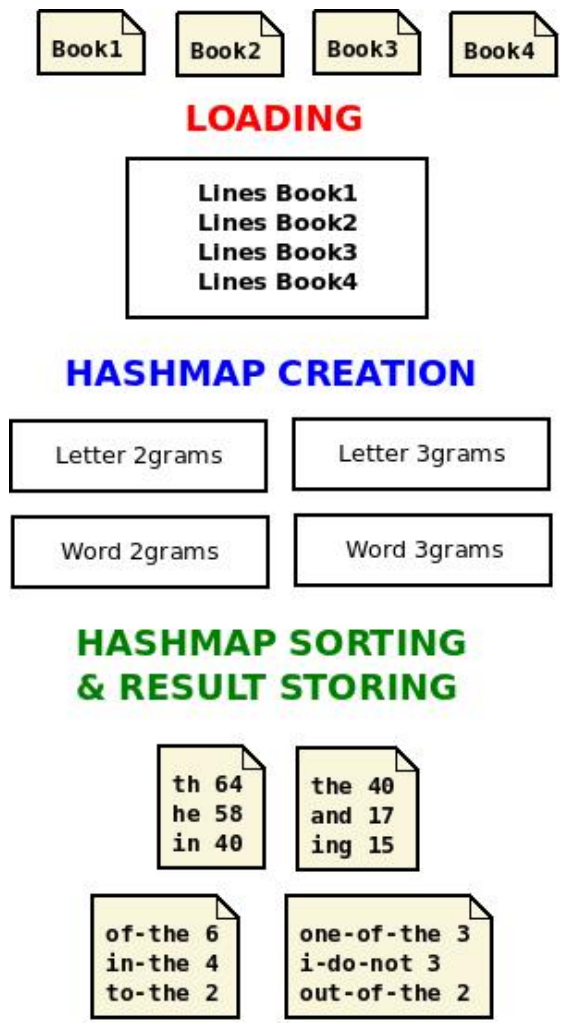
N = 3 It was all Pride — *word trigrams* → `it-was-all` `was-all-pride`

- Programming language: Java
- A sequential implementation
- A parallel implementation: Java Threads

Java Implementation
Sequential

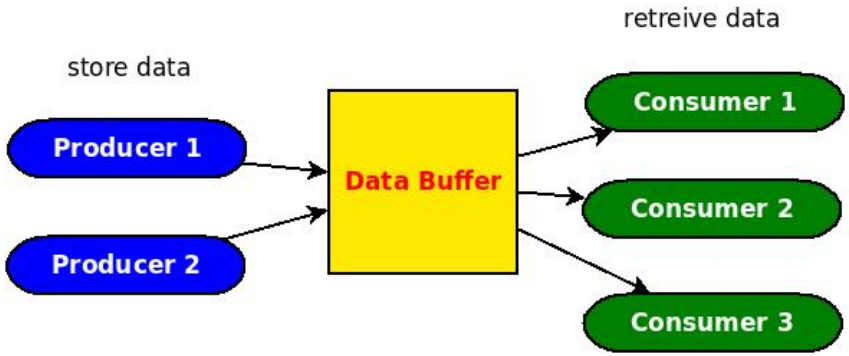
Steps:

- book loading
- Hashmap creation
- Hashmap sorting & result sorting

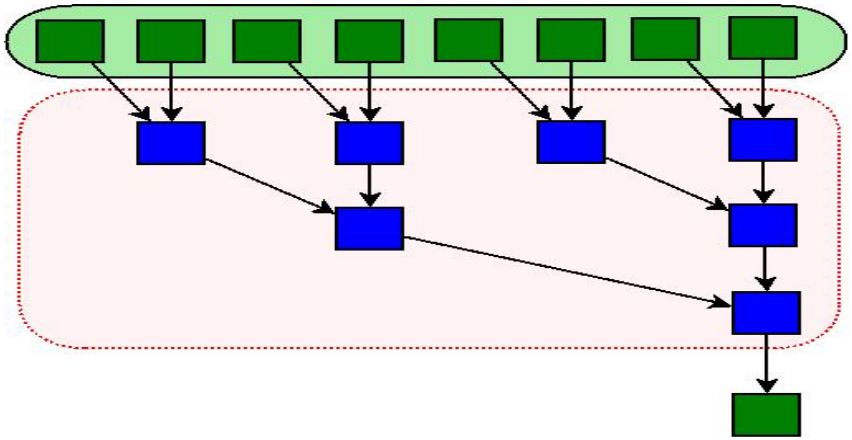


Pattern involved

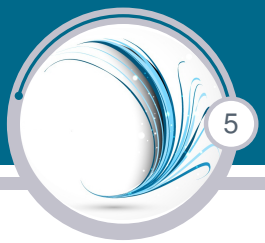
- Producer Consumer Pattern



- Reduction Pattern

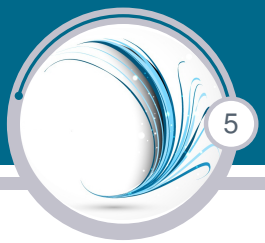


Java Implementation
Steps



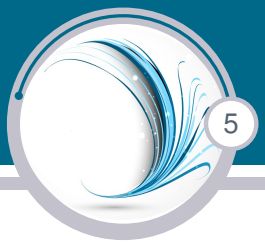
- We use two different classes that implements **Runnable** interface, one named **Producer** and the other **Consumer**

Java Implementation
Steps



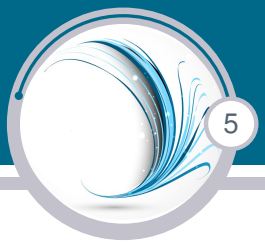
- We use two different classes that implements **Runnable** interface, one named **Producer** and the other **Consumer**
- The ***Producer Thread*** loads each book's line from the directory in a **BlockingQueue**.

Java Implementation
Steps



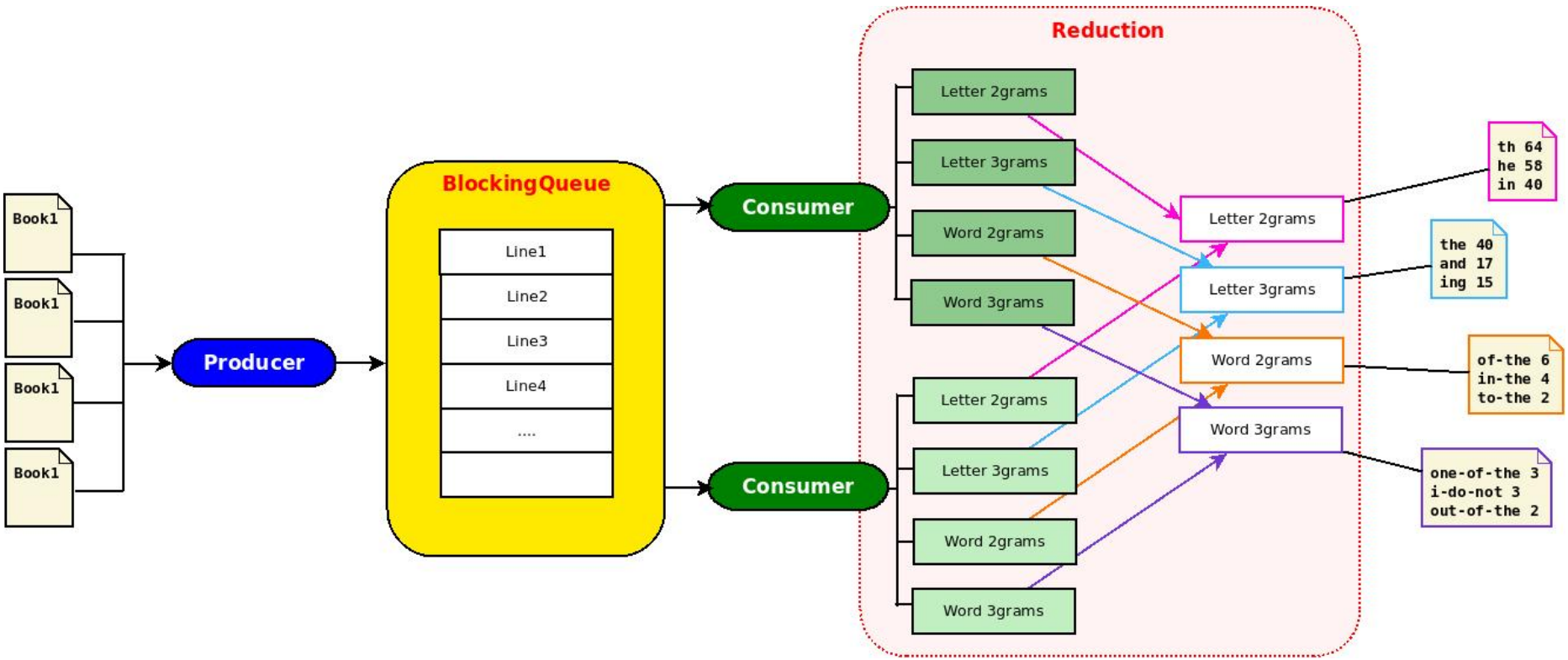
- We use two different classes that implements **Runnable** interface, one named **Producer** and the other **Consumer**
- The ***Producer Thread*** loads each book's line from the directory in a **BlockingQueue**.
- The ***Consumer Thread*** takes the lines from the same **BlockingQueue** one by one and compute the partial results of the current line, which are four **HashMap** for the N-grams of every type.

Java Implementation
Steps



- We use two different classes that implements **Runnable** interface, one named **Producer** and the other **Consumer**
- The ***Producer Thread*** loads each book's line from the directory in a **BlockingQueue**.
- The ***Consumer Thread*** takes the lines from the same **BlockingQueue** one by one and compute the partial results of the current line, which are four **HashMap** for the N-grams of every type.
- The **Consumer** share four **ConcurrenHashMap** and as soon as the **BlockingQueue** is empty and the **Producer** has completed its task, the partial results are completed and the consumers can add them in the corrisponding shared **ConcurrentHashMap**.

Java Implementation
Steps



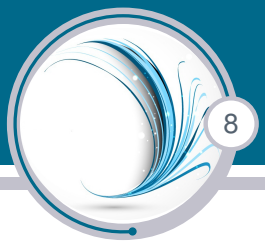
Experiments
Java Thread

We test our program with two datasets of books, downloaded from the Gutenberg Project [1], repeating them several times. I For the parallel versions the number of producer is always one for English datasets and one for the Italian ones. The number of consumers instead varies from one to four for each dataset to see the benefit of parallelism..

Size	Seq	Par1	Par2	Par3	Par4
2.7	2902	3168	2661	2483	2573
5.4	5564	5518	4249	4203	4185
10.8	10563	10430	7479	7334	6177
21.6	22269	21109	13539	11326	11121
43.2	54458	41346	27533	22001	21988
86.4	156008	83835	55353	43905	42066

[1] https://www.gutenberg.org/

Experiments
OpenMP



Once the execution times have been detected we compute the relative **Speedup**

Rep	Sp1	Sp2	Sp3	Sp4
2.7	0.9160	1.0906	1.1687	1.1279
5.4	1.0083	1.3095	1.3238	1.3295
10.8	1.0128	1.4124	1.4403	1.7101
21.6	1.0055	1.6448	1.9662	2.0024
43.2	1.3171	1.9779	2.4753	2.4767
86.4	1.8609	2.8184	3.5533	3.7086

Experiments
OpenMP

Once the execution times have been detected we compute the relative **Speedup**

Rep	Sp1	Sp2	Sp3	Sp4
2.7	0.9160	1.0906	1.1687	1.1279
5.4	1.0083	1.3095	1.3238	1.3295
10.8	1.0128	1.4124	1.4403	1.7101
21.6	1.0055	1.6448	1.9662	2.0024
43.2	1.3171	1.9779	2.4753	2.4767
86.4	1.8609	2.8184	3.5533	3.7086

- No speedup
- Linear speedup
- Maximum speedup

Results

Letter2grams	Letter3grams	Word2grams	Word3grams
th 2062144	**the** 1284320	**of-the** 99776	**one-of-the** 3392
he 1866208	**and** 566816	**in-the** 71328	**i-do-not** 3392
in 1300448	**ing** 510048	**to-the** 41952	**out-of-the** 3232

Letter2grams	Letter3grams	Word2grams	Word3grams
on 1063040	**ent** 435072	**e-di** 15328	**per-lo-piu** 1472
er 963808	**ell** 415040	**per-la** 12000	**poco-a-poco** 1472
en 914400	**ion** 332768	**che-si** 11840	**a-poco-a** 1472

Conclusions & Application

Conclusions

- two different implemetations of N-grams occurence calculus: bigrams and trigrams of letters and words.
- the sequential version takes one line of a book at a time and processes it.
- the parallel version with explicit *Java Threads*, *Reduction Pattern* and the *Producer and Consumer Pattern* computes the same work but the execution time is quicker.
- with the increasing of the dimension of the dataset the execution time of the parallel version is three times less of sequential one, indeed the speedup has a value of 3.7086.

Conclusions & Application

Aplication

- Text Prediction
- Spelling Correction
- Language Identification
- Plagiarism Detection

Thank you for the attention