UNIVERSITÀ
DEGLI STUDI
FIRENZE

Sequential and parallel implementation of bigrams and trigrams with Java

A.A. 2019-2020

Lorenzo Pratesi Mariti

UNIVERSITÀ DEGLI STUDI FIRENZE

Table of contents

Introduction

In the fields of computational linguistics ***n*-gram** with $n \in [1,\infty)$ is a contiguous sequence of n items from a given sample of text or speech.

- $n = 1$ is referred to as a *unigram*;
- $n = 2$ is a *bigram* (or, *digram*);
- $n = 3$ is a *trigram*;
- and so on …

Introduction

The main goal is to analyze the frequency of letters related to Bigrams and Trigrams using both *sequential* and *parallel* approaches.

Sequential version:

- **Java**

Parallel version:

- **Java Thread**

Computation - Sequential

The sequential version has 2 types of data:

- `fileString`: contains a list of character from the txt file
- `n`: number of grams, 2 for bigram and 3 for trigrams

```
algorithm computeNGrams(fileString, n) is

   for i = 0 to fileString.length - n do
      ngram = “”
      for j = 0 to n - 1 do
         ngram += fileString[i + j]
      end
      \\ insert the ngram in a data structure
   end
```

Computation - Parallel

The parallel version has the same structure but uses more information, depending on how many threads we use:

- `id`: contains the thread ID
- k: dimension of txt for each thread, calculated as `floor(text.length / nThread)`
- `start`: start character, calculated as `(k * i)`
- `stop`: end character, calculated as `((i + 1) * k) + (n - 1) - 1`

```
algorithm computeNGrams(id, k, n, start, stop, n,  fileString) is

   for i = this.start to this.stop - this.n do
      ngram = “”
      for j = 0 to this.n - 1 do
         ngram += fileString[i + j]
      end
      \\ insert the ngram in a data structure
   end
```

Sequential Implementation

Data preprocessing

- **`readTextFromFile`**`()`: the input text is been analyzed and cleaned up by removing those character defined invalid.

Data structure

- **`HashMap`**: selected data structure to store bigrams and trigrams. Constant time for basic operation for large sets.

Computation

- **`computeNgrams`**`()`: The function behavior follows the computation for sequential version explained in previous slide, and returns the HashMap with bigrams or trigrams calculated.
  - `List<Characters> fileString`: the text to analyze
  - `int n`: identifies bigram or trigram (2 or 3)

Parallel implementation - idea

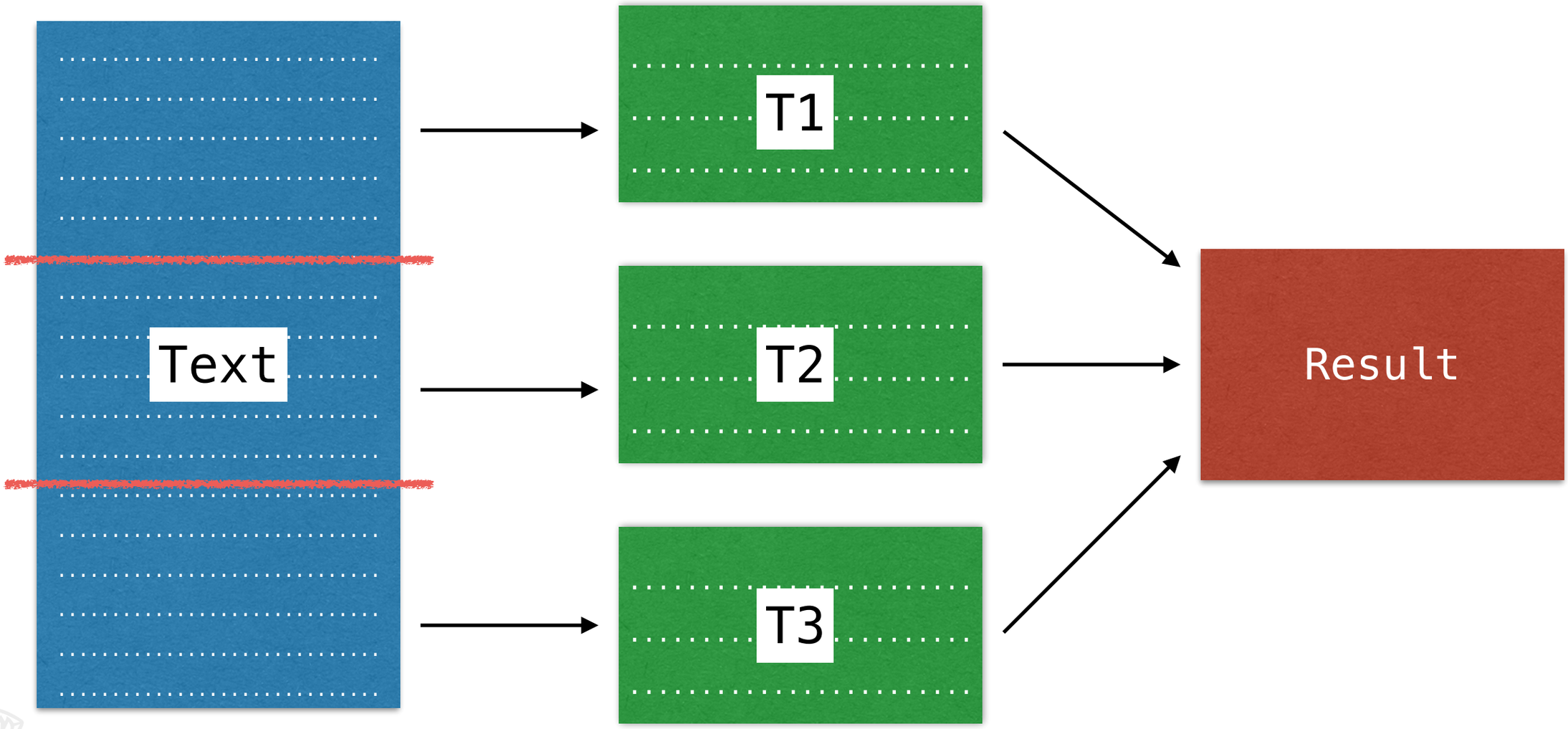
- The main idea is to parallelize the sequential behavior.

- The text it's been divided in as many parts as are the thread instances and leave the search of bigrams and trigrams on a single part to a single thread.

- In this way we'll see in results how the computational times are significantly reduced compared to sequential times.

Parallel implementation - Java Thread

Example of parallel implementation idea with 3 threads



Parallel implementation - Java Thread

Data preprocessing

- **`readTextFromFile()`:** equal to the sequential version.

Data structure - `ConcurrentHashMap`

- Allows concurrent modifications of the Map from several threads without the need to block them.
- Advantages: instead of needing to lock the whole structure when making a change, it's only necessary to lock the bucket that's being altered.
- It is basically lock-free on reads.

Parallel implementation - Java Thread

Computation

The computation is divided by 4 main tasks.

1. Declare a `ParallelThread` class which implements `Callable`

2. Implement the `call()` method as described in the “Parallel version” that allows to compute bigrams or trigrams and create the `HashMap`.

3. Implement a `HashMerge` function that merges the `ConcurrentHashMaps` returned from different threads.

4. Instantiate a `Future` array and an `ExecutorService` specifying the thread pool size.

Finally use the **`ExecutorService`** object to submit the compute method and get the results through **`.get()`** Future method.

Experimental Result

- Execute both sequential and parallel version with different text sizes
- Analyze the behavior of SpeedUp between the two version. $S_p = t_s / t_p$
- Plot some interesting results

Experimental Result

Dataset for testing

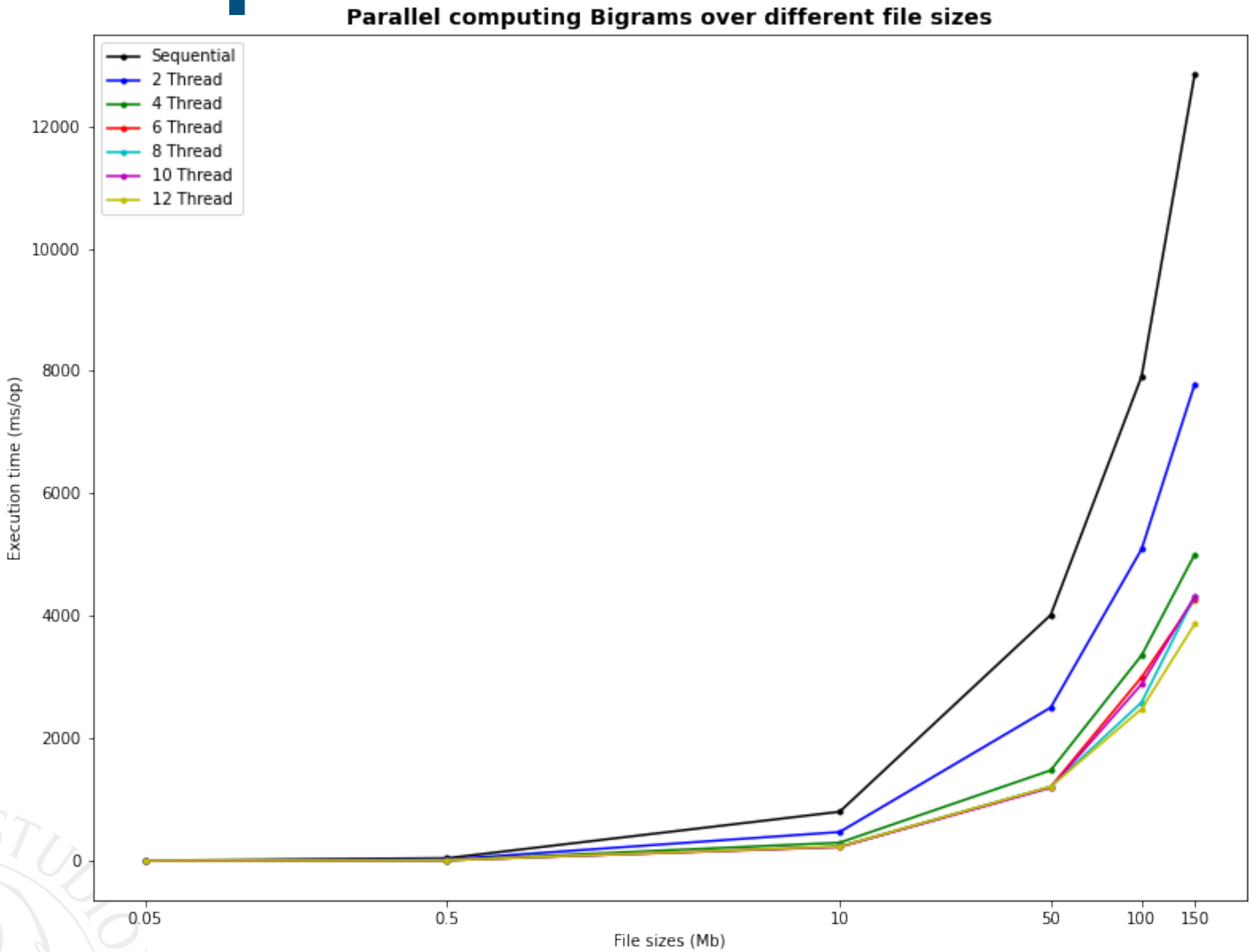
- 6 text file with dimension 50KB, 500KB, 10MB, 50MB, 100MB, 150MB

- For the parallel version each test has been repeated six time regarding the number of threads (from 2 to 12).

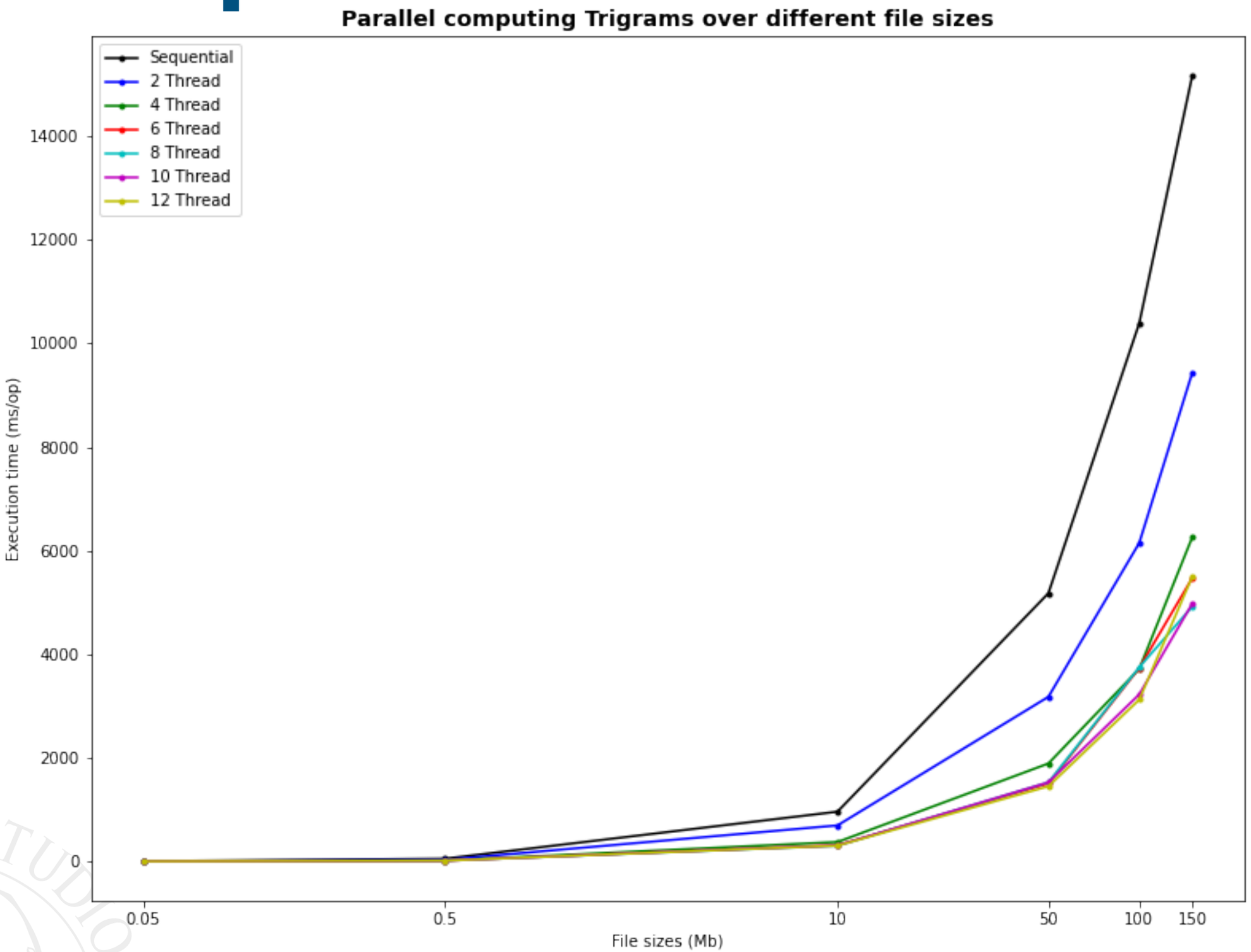
For every thread number the computational time has been collected 100 times and averaged.

The execution of all the test has been managed by JMH
(Java Microbenchmark Harness)

Experimental Result



Experimental Result



UNIVERSITÀ DEGLI STUDI FIRENZE

Experimental Result