Concurrent and Distributed Systems

Mitre Flavia Antonia

November 1, 2019

Third Year
Group: C.E.N. 3.1
Lecturer: Costin Bădică
Teaching assistant: Cristinel Ungureanu

1 Problem statement

1.1 First problem

Find all the prime numbers on the interval [1, n] using k threads. Let n = kq + r where $0 \leq r \leq k$ where r is the rest of the division of n to k. We will consider 2 solutions:

- Partition the interval [1, n] in k intervals so: I1 = [1, q+1], I2 = [q+2, 2q+2], ..., Ir = [(r-1)q+r, rq+r], Ir + 1 = [rq+r+1, (r+1)q + r], ..., Ik = [(k1)q+r+1, kq+r]. Each thread $1 \leq j \leq k$ will determine the prime numbers from the Ij interval.

- k+1 multiples that are strictly greather than k+1 are not prime numbers. Eliminate these numbers from the interval [1, n] resulting the set M. This set will be partitioned in k subsets so: for each $1 \leq j \leq k$ the set Mj contains those elements from M that give the rest j to the division through k+1. Also consider that k+1 $\in$ M1. Each thread j will determine the prime numbers from the Mj set.

1.2 Seconds problem

Concurrent counting algorithm	
integer n ← 0	
p	q
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

- What values does n has in the end of this algorithm?
- Implement it in JAVA.

2 Implementation

2.1 First problem

2.1.1 First solution

For solving the first problem, for the first solution I've created four java classes: MainThread, MyThread, DivisionTerms and RandomGenerator.

Class **MainThread**

1. Static variables created for division: **Quotient**, **Remainder**
2. Setters and getters for quotient and reminder
3. Contains the main of the program
4. Beginning of main.
    (a) Creating two objects: one for **DivisionTerms** and one for **RandomGenerator**, used to ease the access to the methods in this classes
    (b) Getting a random value for n (the number that is the right margin of our interval) using a method from the **RandomGenerator** class
    (c) Getting a random value for the number of threads using a method from the **RandomGenerator** class. This number must be lower than n and different from 0, to avoid division to 0
    (d) Calculating and storing the values for the quotient and the remainder using methods from class **DivisionTerms** (**CalculateRemainder**, **CalculateQuotient**) and storing them using setters.
    (e) Creating a vector of threads using the generated length
    (f) Creating as many threads as the generated number of threads, using the constructor from **MyThread** class, and starting them
    (g) Using the join method for all the existing threads, also using try-catch because the join method can throw exceptions
    (h) Printing the n and the number of threads
5. End of main

Class **MyThread**

1. Variable used to store the index for every thread: **ThreadIndex**
2. Creating an object of class **DivisionTerms**
3. **Constructor** for this class, that has a long number as parameter
4. Overriding the method **run**, from the imported java class **Thread**
    (a) Declaring variables for **position**, **left interval margin** and **right interval margin**
    (b) **Calculating the left interval margin** using the method from the class **DivisionTerms** for the actual thread index
    (c) Initialising the **position** with the value of **left margin interval**
    (d) **Calculating the right interval margin** using the method from the class **DivisionTerms** for the actual thread index
    (e) For every number in the interval:
        i. Checking if the number is prime, using the method **CheckPrime** from the class **DivisionTerms**, and if it is, printing it
        ii. Incrementing the position (number)

Class **DivisionTerms**

1. Creating an object of class **MainThread**
2. Method **CalculateQuotient**
    (a) Receives the number n and the number of threads randomly generated
    (b) Returns the result of the division between n and the number of threads
3. Method **CalculateRemainder**
    (a) Receives the number n and the number of threads randomly generated
    (b) Returns the remainder of the division between n and the number of threads

4. Method **CheckPrime**
    (a) Receives a number
    (b) If the number equals 1, the method returns 0 (as 1 is not a prime number)
    (c) If the number equals 2, the method returns the number
    (d) If none of the above happens (obviously) if the number has any divisor, the method returns 0
    (e) If neither this happened, the method returns the number, it means it is prime
5. Method **IntervalLeftMargin**
    (a) Receives a **position**
    (b) The variable **ILeft** is used to calculate the left margin of an interval
    (c) If the **position** equals 1, the method returns 1 (the left margin equals 1)
    (d) If the **position** equals 2, **ILeft** becomes the **quotient** (using the getter from the class **MainThread** for this) plus 2
    (e) If the position equals 3, **ILeft** becomes **position** minus 1, all multiplied by the **quotient**, plus the **remainder**, plus 1
    (f) In any other case, ILeft becomes position minus 1, all multiplied by the quotient, plus 3
    (g) The method returns the calculated value for the left interval margin
6. Method **IntervalRightMargin**
    (a) Receives a number (position)
    (b) The variable **IRight** is used to calculate the right margin of an interval
    (c) If the **position** equals 1, **IRight** becomes the **quotient** (using the getter from the class **MainThread** for this) plus 1
    (d) If the **position** equals 2, **IRight** becomes the **quotient** multiplied by 2 plus 2
    (e) In any other case, **IRight** becomes the **position** multiplied by the **quotient**, plus the **remainder**
    (f) The method returns the calculated value for the right interval margin

Class **RandomGenerator**

1. Contains the method **GetRandomNr**
    (a) Creating an instance of the imported class **Random**
    (b) This instance is used to get a random number between 0 and a given number

2.2 First problem

2.2.1 Second solution

For solving the first problem, for the second solution I've created four java classes: MainThread, MyThread, DivisionTerms and RandomGenerator.

Class **MainThread**

1. Static variables created for n: **Number**, and for the number of threads: **ThreadsNr**
2. Setters and getters for n and for the threads number
3. Contains the main of the program
4. Beginning of main.
   (a) Creating an object for **RandomGenerator**, used to ease the access to the methods in this class
   (b) Getting a random value for n (the number that is the right margin of our interval) using a method from the **RandomGenerator** class
   (c) Getting a random value for the number of threads using a method from the **RandomGenerator** class. This number must be lower than n and different from 0, to avoid division to 0
   (d) Storing the values for n and for the number of threads using setters (**setNumber**, **setThreadsNr**)
   (e) Creating a vector of threads using the generated length
   (f) Creating as many threads as the generated number of threads, using the constructor from **MyThread** class, and starting them
   (g) Using the join method for all the existing threads, also using try-catch because the join method can throw exceptions
   (h) Printing the n and the number of threads
5. End of main

Class **MyThread**

1. Variable used to store the index for every thread: **ThreadIndex**
2. Creating two objects: one for the class **DivisionTerms** and one for the class **MainThread**
3. **Constructor** for this class, that has a long number as parameter
4. Overriding the method **run**, from the imported java class **Thread**
    (a) Declaring variables for **position**, **right interval margin** and **CheckPrime**
    (b) Initialising the **position** with the value of **ThreadIndex**
    (c) **Calculating the right interval margin** using the method from the class **DivisionTerms** for the actual thread index
    (d) For every number in the interval:
        i. If the **number** isn't divisible with the **number of threads plus one**, and isn't bigger than the **number of threads plus one**, we check if it is prime, using the method **CheckPrime** from the class
        ii. If the number equals the **number of threads plus one**, we check if it is prime, using the method **CheckPrime** from the class **DivisionTerms**
        iii. If none of the above cases happens, it means the number is not prime (it is a multiple of **number of threads plus one**)
        iv. If the number is prime, it is printed
        v. Incrementing the **position** (number) with the **number of threads** (using a getter from class **MainThread**) plus 1. The idea is to make the partitions as in the problem statement

Class **DivisionTerms**

1. Creating an object of class **MainThread**
2. Method **CheckPrime**
    (a) Receives a number
    (b) If the number equals 1, the method returns 0 (as 1 is not a prime number)
    (c) If the number equals 2, the method returns the number
    (d) If none of the above happens (obviously) if the number has any divisor, the method returns 0
    (e) If neither this happened, the method returns the number, it means it is prime

3. Method **IntervalRightMargin**
   (a) Receives a number (position)
   (b) The variable **IRight** is initialised with the value of the number
   (c) While **IRight** is smaller than the number **n** (found using a getter from class **MainThread**)
      i. **IRight** is incremented with **number of threads** (found using a getter from class **MainThread**) plus 1
   (d) If **IRight** is bigger than the number **n** (found using a getter from class **MainThread**), **IRight** is decremented by the **number of threads** plus 1
   (e) The method returns the calculated value for the **right interval margin**

Class **RandomGenerator**

1. Contains the method **GetRandomNr**
   (a) Creating an instance of the imported class **Random**
   (b) This instance is used to get a random number between 0 and a given number

2.3 Second problem

For solving the second problem I've created two java classes: MainThread and MyThread.

Class **MainThread**

1. Contains the main of the program
2. Beginning of main.
   (a) Creating the first thread, using the imported class **Thread** and starting it
   (b) Creating the second thread, using the imported class **Thread** and starting it
   (c) Using the join method for the existing threads, also using try-catch because the join method can throw exceptions
3. End of main

Class **MyThread** extends **Thread**

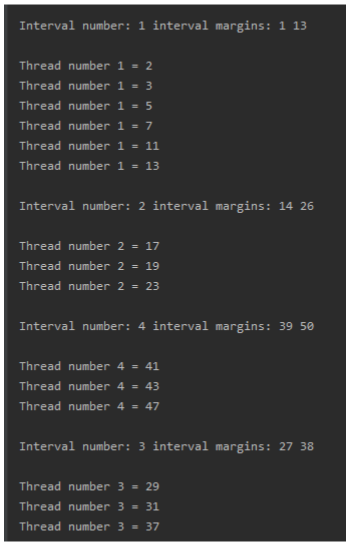
1. Static variable used to store the sum for every thread: **n**, initialised with zero
2. Variable used to store the index for every thread: **ThreadIndex**
3. Creating an object of class **DivisionTerms**
4. **Constructor** for this class, that has a long number as parameter (here, the index of the thread)
5. Overriding the method **run**, from the imported java class **Thread**
   (a) Declaring variables **i**, initialised with zero, and **temp**
   (b) While i is smaller than 10
      i. Print current thread index and n
      ii. The variable **temp** receives the value of **n**
      iii. The variable **n** receives the value of **temp** plus 1
      iv. The counter **i** is incremented by one
   (c) The variable **n** is printed

3 Experimental Data

3.1 First problem

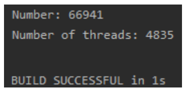
3.1.1 First solution

In this section are a few non-trivial input tests and their outputs, and, also the execution time of every one of them.



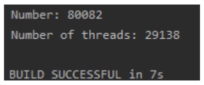
```
Interval number: 1 interval margins: 1 13

Thread number 1 = 2
Thread number 1 = 3
Thread number 1 = 5
Thread number 1 = 7
Thread number 1 = 11
Thread number 1 = 13

Interval number: 2 interval margins: 14 26

Thread number 2 = 17
Thread number 2 = 19
Thread number 2 = 23

Interval number: 4 interval margins: 39 50

Thread number 4 = 41
Thread number 4 = 43
Thread number 4 = 47

Interval number: 3 interval margins: 27 38

Thread number 3 = 29
Thread number 3 = 31
Thread number 3 = 37
```

Figure 1: Output for the first set of data.
In this case everything worked fine, the interval is partitioned as the formula says, and the execution time was approximate 0 seconds.



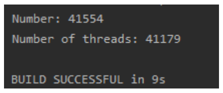
```
Number: 66941
Number of threads: 4835

BUILD SUCCESSFUL in 1s
```

Figure 2: Output for the second set of data.



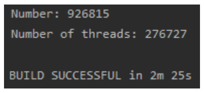
```
Number: 80082
Number of threads: 29138

BUILD SUCCESSFUL in 7s
```

Figure 3: Output for the third set of data.



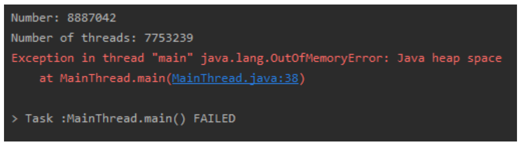
```
Number: 41554
Number of threads: 41179

BUILD SUCCESSFUL in 9s
```

Figure 4: Output for the fourth set of data.



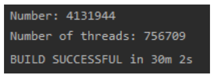
```
Number: 926815
Number of threads: 276727

BUILD SUCCESSFUL in 2m 25s
```

Figure 5: Output for the fifth set of data.
In this case we can see that the execution time grows when the number and the number of threads get greater.



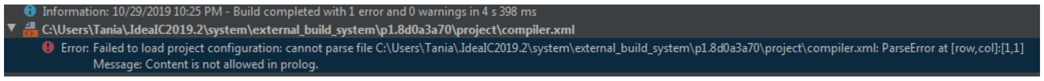
```
Number: 8887042
Number of threads: 7753239
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at MainThread.main(MainThread.java:38)

> Task :MainThread.main() FAILED
```

Figure 6: Output for the sixth set of data.
Here we can see that a big number of threads can produce a Java Heap space exception. The execution stopped after 30 seconds, maybe one minute.



```
Number: 4131944
Number of threads: 756709
BUILD SUCCESSFUL in 30m 2s
```

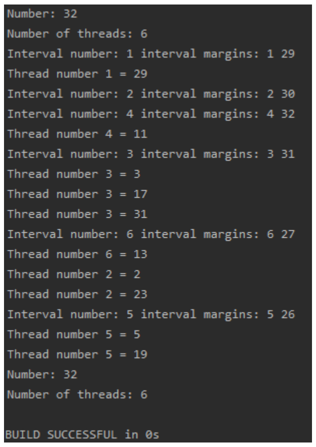
Figure 7: Output for the seventh set of data.
For a number greater than four million and approximate eight hundred thousand threads the program finished the execution, but it lasted an eternity: 30 minutes and two seconds.



```
Information: 10/29/2019 10:25 PM - Build completed with 1 error and 0 warnings in 4 s 398 ms
C:\Users\Tania\.IdeaIC2019.2\system\external_build_system\p1.8d0a3a70\project\compiler.xml
    Error: Failed to load project configuration: cannot parse file C:\Users\Tania\.IdeaIC2019.2\system\external_build_system\p1.8d0a3a70\project\compiler.xml: ParseError at [row,col]:[1,1]
        Message: Content is not allowed in prolog.
```

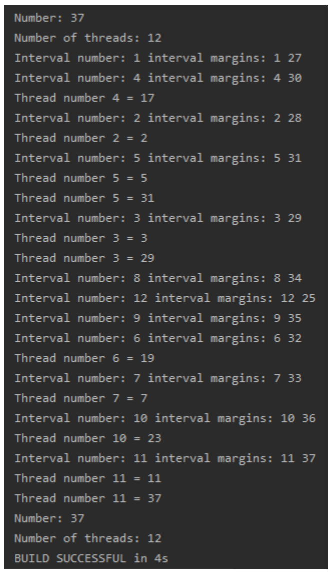
Figure 8: Output for the eighth set of data.
I've also tried to run the program with some values around nine hundred million, and after two or three minutes the laptop shut down and the content of a file from Intellij IDEA was deleted. Had to remake the project.

3.1.2 Second solution



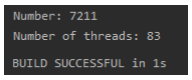
```
Number: 32
Number of threads: 6
Interval number: 1 interval margins: 1 29
Thread number 1 = 29
Interval number: 2 interval margins: 2 30
Interval number: 4 interval margins: 4 32
Thread number 4 = 11
Interval number: 3 interval margins: 3 31
Thread number 3 = 3
Thread number 3 = 17
Thread number 3 = 31
Interval number: 6 interval margins: 6 27
Thread number 6 = 13
Thread number 2 = 2
Thread number 2 = 23
Interval number: 5 interval margins: 5 26
Thread number 5 = 5
Thread number 5 = 19
Number: 32
Number of threads: 6

BUILD SUCCESSFUL in 0s
```

Figure 9: Output for the first set of data.
In this case everything worked fine, the interval is partitioned as the formula says, and the execution time was approximate 0 seconds.



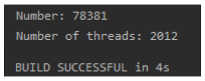
```
Number: 37
Number of threads: 12
Interval number: 1 interval margins: 1 27
Interval number: 4 interval margins: 4 30
Thread number 4 = 17
Interval number: 2 interval margins: 2 28
Thread number 2 = 2
Interval number: 5 interval margins: 5 31
Thread number 5 = 5
Thread number 5 = 31
Interval number: 3 interval margins: 3 29
Thread number 3 = 3
Thread number 3 = 29
Interval number: 8 interval margins: 8 34
Interval number: 12 interval margins: 12 25
Interval number: 9 interval margins: 9 35
Interval number: 6 interval margins: 6 32
Thread number 6 = 19
Interval number: 7 interval margins: 7 33
Thread number 7 = 7
Interval number: 10 interval margins: 10 36
Thread number 10 = 23
Interval number: 11 interval margins: 11 37
Thread number 11 = 11
Thread number 11 = 37
Number: 37
Number of threads: 12
BUILD SUCCESSFUL in 4s
```

Figure 10: Output for the second set of data.
For number 37 and 12 threads, the execution time was approximate 4 seconds.



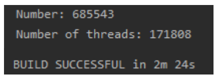
```
Number: 7211
Number of threads: 83

BUILD SUCCESSFUL in 1s
```

Figure 11: Output for the third set of data.
For number 7211 and 83 threads, the execution time was 1 second.



```
Number: 78381
Number of threads: 2012

BUILD SUCCESSFUL in 4s
```

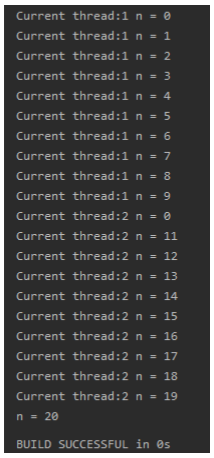
Figure 12: Output for the fourth set of data.
For number 78381 and 2012 threads, the execution time was approximate 4 seconds.



```
Number: 685543
Number of threads: 171808

BUILD SUCCESSFUL in 2m 24s
```

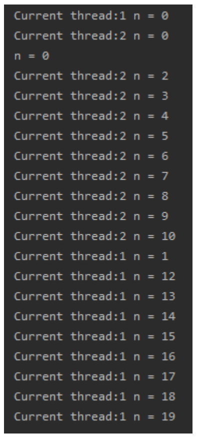
Figure 13: Output for the fifth set of data.
In this case we can see that the execution time grows when the number and the number of threads get greater. The execution time was 2 minutes and 24 seconds.

3.2 Second problem



```
Current thread:1 n = 0
Current thread:1 n = 1
Current thread:1 n = 2
Current thread:1 n = 3
Current thread:1 n = 4
Current thread:1 n = 5
Current thread:1 n = 6
Current thread:1 n = 7
Current thread:1 n = 8
Current thread:1 n = 9
Current thread:2 n = 0
Current thread:2 n = 11
Current thread:2 n = 12
Current thread:2 n = 13
Current thread:2 n = 14
Current thread:2 n = 15
Current thread:2 n = 16
Current thread:2 n = 17
Current thread:2 n = 18
Current thread:2 n = 19
n = 20

BUILD SUCCESSFUL in 0s
```

Figure 14: Output for the first set of data.
In this case the value for n is 20, I used the join method. The running time was approximate 0 seconds.

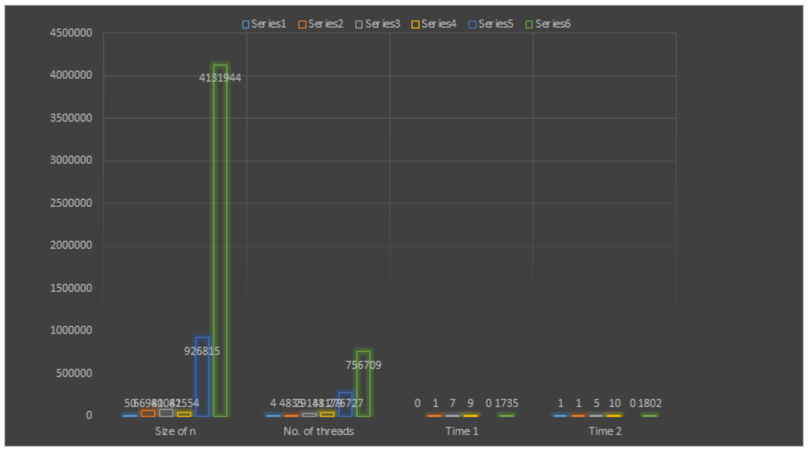


```
Current thread:1 n = 0
Current thread:2 n = 0
n = 0
Current thread:2 n = 2
Current thread:2 n = 3
Current thread:2 n = 4
Current thread:2 n = 5
Current thread:2 n = 6
Current thread:2 n = 7
Current thread:2 n = 8
Current thread:2 n = 9
Current thread:2 n = 10
Current thread:1 n = 1
Current thread:1 n = 12
Current thread:1 n = 13
Current thread:1 n = 14
Current thread:1 n = 15
Current thread:1 n = 16
Current thread:1 n = 17
Current thread:1 n = 18
Current thread:1 n = 19
```

Figure 15: Output for the second set of data.
In this case the value for n is 0, I didn't use the join method. The running time was approximate 0 seconds.

4 Conclusions

Diagram of results for the first problem



The first problem has two solutions:

For the first solution, one thread must check every number in a closed interval (1, 2, 3, ...end of interval).

For the second solution, the interval checked at every step is composed of numbers which subtracted give the result number of threads plus one (number of threads equals 7 $\rightarrow$ number of threads plus 1 equals 8, example of interval for a thread: 1, 9, 17, ...end of interval).

In my opinion the second one is the best, because the run time is faster and it also eliminates as many numbers as the quotient's value from the interval. The time for checking if this are prime is saved.

The second problem is a demonstration of the concurrent counting. It is a good example to illustrate the multithreading and how it works. It also illustrates the utility of the join method, as if it is not used, the counting algorithm is useless, because the result is zero.