Faculty of Automation, Computers and Electronics

Concurrent and Distributed Systems

Lab Assignment

Stanciu Alin Marian
Speciality: CR 3.3B, 3^{rd} year

October, 2019

Find all prime numbers in [1, n] using k threads
&
Concurrent counting

Abstract

This document introduces the goals and methodology for developing the Concurrent and Distributed Systems homework. The assignment is solved in Java language and project is created with IntelliJ IDEA. The main principle is to implement solution with threads help, every thread solving a specific task one time. The repository of the problem is on GitHub at: https://github.com/StanciuAlin/Concurrent-and-Distributed-Systems-Assignments.

1 *Problem statement*

Find all prime numbers in [1,n] using k threads
Find all prime numbers in [1,n] using k threads. Let n = k * q + r, where $0 \leq$ r $\leq$ k where r is the remainder of n divided to k. You should consider 2 solutions:
$\rightarrow$ Partition the interval [1,n] in k intervals as follows: I_1=[1,q+1], I_2=[q+2, 2*q+2], . . . , I_r=[(r-1)*q+r, r*q+r], I_{r+1}=[r*q+r+1, (r+1)*q+r], ..., I_k=[(k-1)*q+r+1, k*q+r]. Each thread $1 \leq$ j $\leq$ k will determine the prime numbers in the interval I_j.
$\rightarrow$ The multiples of k+1 strictly bigger than k+1 are not prime numbers. These numbers should be eliminated from the interval [1,n] resulting in set M. This set should be partitioned in k subsets as follows: for each $1 \leq$ j $\leq k$ the set M_j contains all those elements from M who by divided with k+1 give remainder j. Also it is considered that k+1 belongs to M_1. Each thread j will determine the prime numbers from set M_j

Task:
Which of the solutions is better and why?

The problem consist of a natural number set [1, n]. The scope is to partition this set in k subsets and every subsets to be executed by one thread from all k and find all prime numbers. Finding all prime numbers in every subset means that at the final of execution we have all prime numbers from entire set [1, n]. The partitioning algorithm is described in explicitly in statement:
$\rightarrow$ Solution 1 series:

$[1, q + 1]$
$[q + 2, 2 * q + 2]$
.
.
.
$[(r - 1) * q + r, r * q + r]$
$[r * q + r + 1, (r + 1) * q + r]$
.
.
.
$[(k - 1) * q + r + 1, k * q + r]$
This relations are implemented in **for** loops until k subsets are created.

Sample: n = 14, k = 5 $\rightarrow$ q = 2, r = 4
I_1 = [1, 3]
I_2 = [4, 6]
I_3 = [7, 9]
I_4 = [10, 12]
I_5 = [13, 14]

$\rightarrow$ Solution 2 relation:
$M_j = M_i\%(k+1) = j, 0 \leq i \leq n$

Sample: n = 14, k = 5 $\rightarrow$ q = 2, r = 4
M_1 = [1, 7, 13, 6] because $M_{1j}\%(k+1) = 1$ for $0 \leq j \leq |M_1|$
M_2 = [2, 8, 14] because $M_{2j}\%(k+1) = 2$ for $0 \leq j \leq |M_2|$
M_3 = [3, 9] because $M_{3j}\%(k+1) = 3$ for $0 \leq j \leq |M_3|$
M_4 = [4, 10] because $M_{4j}\%(k+1) = 4$ for $0 \leq j \leq |M_4|$
M_5 = [5, 11] because $M_{5j}\%(k+1) = 5$ for $0 \leq j \leq |M_5|$

Concurrent counting
What values can n have at the end of the concurrent counting algorithm execution?
Tasks:
$\rightarrow$ Implement this algorithm in Java. What values do you get at the end for n?
$\rightarrow$ Prove the above by describing in the technical reports a few test scenarios and different states as shown in the first course.

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

Figure 1: Concurrent Counting statement

2 Pseudocode of algorithms / Implementation / Solution

2.1 Algorithms used in problem 1 for solution 1 and solution 2

Algorithm 1 Thread.run()

```
1: for i ← leftMargin, rightMargin do
2:     if partition.get(iterator) <> null then
3:         if partition.get(iterator) is prime then
4:             allPrimeNumbers.add(partition.get(iterator)
5: Print(allPrimeNumbers)
6: Save(allPrimeNumbers)
```

Whenever we make a class which extends java.lang.Thread or implements java.lang Runnable interface, it must to implement the **run()** function (what to execute in that thread). The above run() function implementation is an iteration into an array, from one left limit to other one right limit, both are class members. For every object in array, check if it is not null and then test if that Integer object is a prime number. If that conditions are true, then i^{-th} element is added in set with the rest prime numbers found in previous steps.
After for loop is over, print all prime numbers found and save them in a static variable of thread class.

The function **isPrime(value)** implement an algorithm to check if **value** parameter is a prime number. A prime number is a number with only 2 divisors, 1 and itself. The algorithm trying to find a number from 2 to half of value which divided the **value** exactly (remainder to be 0). If one number is found, a prime flag is set to false otherwise flag remain with intial value, true, so the value parameter represent a prime number.

Algorithm 2 isPrime(value)

```
1: divisor ← 2
2: flagPrime ← true
3: while divisor ≤ value / 2 and flagPrime is true do
4:     if value % divisor = 0 then
5:         flagPrime ← false
6: if flagPrime is true and value <> 1 then
7:     return true
8: return false
```

2.2 Find all prime numbers in [1,n] using k threads

Solution 1 - Algorithm 3

Solution 1 implement mathematical relations described above. Firstly, find quotient and reminder, and for first r threads instantiate a thread which find all prime numbers in partition (created with Algorithm 4). After that, continue until k threads are created in the same way, but with other dimension for partition.

At the beginning of the function, we start a timer, and the final, we stop it and print total execution time recorded. The output is written in a CSV file as n = ..., k = ..., executionTime = ... ms, requirement = 1.

Algorithm 3 Problem1Requirement1()

```
1: SetValues() for n and k
2: StartTimerExecution()
3: q ← n / k
4: r ← n % k
5: for i ← 1, r do
6:     Thread thread ← new PrimeNumThread(0, q, ConstructPartition((i - 1)*q+i, i*q+i)
7:     thread.start()
8:     thread.join()
9: i ← i - 1
10: while iterator < k do
11:     Thread thread ← new PrimeNumThread(0, q - 1, ConstructPartition(i*q+r+1, (i+1)*q+r))
12:     thread.start()
13:     thread.join()
14:     i ← i + 1
15: StopTimerExecution()
16: PrintExecutionTime()
17: StoreResultCSV.WriteLine(n, k, executionTime, requirement)
```

Algorithm 4 ConstructPartitionRequirement1(left, right)

```
1: for i ← left, right do
2:     partition.add(i)
3: return partition
```

Solution 2 - Algorithm 5

Solution 2 implement mathematical relations described above. Firstly, find quotient and reminder. Add in array all consecutive numbers from 0 to n, and then remove values which are multiples of k + 1 value. Construct partition (Algorithm 6) where firstly check if exist k + 1 value (if yes, remove it). After that, add to an array of array all values less or equal than n. In problem statement it's specified to insert k + 1 value into set for first thread and return the k array, every array with at most k values. Create

k threads, start and join them.
At the beginning of the function, we start a timer, and the final, we stop it and print total execution time recorded. The output is written in a CSV file as n = ..., k = ..., executionTime = ... ms, requirement = 2.

Algorithm 5 Problem1Requirement2()

```
SetValues() for n and k
StartTimerExecution()
q ← n / k
r ← n % k
for i ← 0, n do
    m.add(i)
for i ← 2, n do
    m.remove(i * (k + 1))
setPartitioned = ConstructPartition(m)
for i ← 0, k + 1 do
    c ← setPartitioned[i]
    Thread thread ← new PrimeNumThread(0, c.length - 1, ConvertArray-
ToList(c))
    thread.start()
    thread.join()
StopTimerExecution()
PrintExecutionTime()
StoreResultCSV.WriteLine(n, k, executionTime, requirement)
```

Algorithm 6 ConstructPartitionRequirement2(m)

```
if m contains k + 1 then
    m.remove(k + 1)
for i ← 0, k - 1 do
    for j ← 0, k - 1 do
        currentValue = (k+1)*j+i+1
        if currentValue ≤ n then
            M[i+1][j] ← currentValue
M[1][k-1] ← k + 1
return M
```

2.3 Concurrent counting

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node has predecessor is the start node.

There are scenarios which can depreciated the final result like:
p1 $\rightarrow$ p2 $\rightarrow$ p3 $\rightarrow$ p1 $\rightarrow$ p2 ... without entering in q process
q1 $\rightarrow$ q2 $\rightarrow$ q3 $\rightarrow$ q1 $\rightarrow$ q2 ... without entering in p process
p1 $\rightarrow$ q1 $\rightarrow$ p2 $\rightarrow$ p3 $\rightarrow$ p1 ...
q1 $\rightarrow$ p1 $\rightarrow$ q2 $\rightarrow$ q3 $\rightarrow$ q1 ...
p1 $\rightarrow$ p2 $\rightarrow$ q2 $\rightarrow$ p3 $\rightarrow$ p1 ...

These are bad cases when after the thread is finished, the n value may not be 20 (make these processes sequentially, result n = 20). For avoid these situation must be

2.4 Input data format

The input for first problem is:
n, $10^3 \leq$ n $\leq 10^8$
k, $10^3 \leq$ k $\leq 10^8$
For example:
n = 90000, k = 5000 At first execution, a context and leave the user to choose an option: 1, 2 or 3. In this example is choose to solve problem 1 with solution 1
The n and k values are generated with Java Random class with maximum dimension required.

2.5 Output data format

The output for first problem is represented by the statement, and every thread with his name, his state and the prime numbers which are in the subset corresponding the thread. After finish running, if shows the total execution time for solution 1 (a) or solution 2 (b) and a new menu for another choice.

```
    Please select an option:
For problem 1 part 1 insert value "1"
For problem 1 part 2 insert value "2"
For exit insert "3"

    The option is: 1
```

Figure 2: Menu with all alternatives

The second problem has not input.

```
******************************************************** Statement 1, part 1 ************************************************************

        Find all prime numbers in [1,n] using k threads. Let n =kq + r, where 0 ≤ r <k where r is the remainder of n divided to k.
        Partition the interval [1,n] in k intervals as follows: 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=[(k-1)q+r+1,kq+r]. Each thread 1 ≤ j ≤ k will determine the prime numbers in the interval Ij

                [1, n] interval partitioned in k intervals, one for every thread

        ------------------> Thread name: Thread-0, State:RUNNABLE <--------------------------
                            Thread found next prime numbers: 2

        ------------------> Thread name: Thread-1, State:RUNNABLE <--------------------------
                            Thread found next prime numbers: 3
```

Figure 3: Header output problem 1

```
                            ************************************************************
                            * Total time execution for problem 1, a is 5464 ms *
                            ************************************************************
    Please select an option:
For problem 1 part 1 insert value "1"
For problem 1 part 2 insert value "2"
For exit insert "3"

    The option is:
```

Figure 4: Footer output problem 1

3 *Experimental data*

42	n = 84	k = 9	Exe time = 52 ms	Requirement = 2	27-10-2019 18:50:47
43	n = 95052	k = 80	Exe time = 1497 ms	Requirement = 1	27-10-2019 18:52:43
44	n = 85952	k = 9	Exe time = 914 ms	Requirement = 2	27-10-2019 18:52:58
45	n = 799335	k = 9	Exe time = 73403 ms	Requirement = 2	27-10-2019 18:56:00
46	n = 472870	k = 3	Exe time = 23529 ms	Requirement = 2	27-10-2019 18:57:58
47	n = 53985	k = 13	Exe time = 536 ms	Requirement = 2	27-10-2019 18:59:56
48	n = 76351	k = 4	Exe time = 961 ms	Requirement = 2	27-10-2019 19:00:23
49	n = 118406	k = 20	Exe time = 1747 ms	Requirement = 2	27-10-2019 19:05:51
50	n = 831528	k = 18	Exe time = 70919 ms	Requirement = 2	27-10-2019 19:08:48
51	n = 732417	k = 634	Exe time = 69375 ms	Requirement = 2	27-10-2019 19:12:17
52	n = 53059	k = 517	Exe time = 1857 ms	Requirement = 2	27-10-2019 19:15:02
53	n = 19016	k = 631	Exe time = 1031 ms	Requirement = 2	27-10-2019 19:17:32
54	n = 94899	k = 837	Exe time = 3295 ms	Requirement = 2	27-10-2019 19:19:41
55	n = 59210	k = 33	Exe time = 711 ms	Requirement = 1	27-10-2019 19:25:20
56	n = 114427	k = 40	Exe time = 1658 ms	Requirement = 1	27-10-2019 19:27:54
57	n = 426323	k = 49	Exe time = 17422 ms	Requirement = 1	27-10-2019 19:29:18
58	n = 562842	k = 5	Exe time = 29050 ms	Requirement = 1	27-10-2019 19:31:40
59	n = 718332	k = 10	Exe time = 51292 ms	Requirement = 1	27-10-2019 19:34:49
60	n = 995475	k = 4	Exe time = 74149 ms	Requirement = 1	27-10-2019 19:37:04
61	n = 51124	k = 11	Exe time = 227 ms	Requirement = 1	27-10-2019 19:39:10
62	n = 49327	k = 3	Exe time = 204 ms	Requirement = 1	27-10-2019 19:40:59

Figure 5: Some tests results

Problem 1

I consider solution 2 a better way to solve the problem. For all values which are multiples of k + 1, in first solution is tested for prime but in second solution is not because are removed. The cost to remove the i^{-th} value from an array is less than to check if it is a prime number. In that way, the difference between the partitioning algorithms is ignored because O(n) complexity is for both.

The results of test are in "Experimental data and results-problem1-solution1.txt" for problem 1, requirement 1 and "Experimental data and results-problem1-solution2.txt" for Problem 1, requirement 2.

Problem 2

The implementation of the problem in Java language is in source folder. Running the application I get value 20 for n. The value is saved in a CSV file with the date and time for current execution. Execute every instruction from p and q process in order to obtain value 20.

Tests:

2	n = 20	27-10-2019 01:45:02
3	n = 20	27-10-2019 01:45:02
4	n = 20	27-10-2019 01:45:02
5	n = 20	27-10-2019 01:45:02
6	n = 20	27-10-2019 01:45:02
7	n = 20	27-10-2019 01:45:02
8	n = 20	27-10-2019 01:45:02
9	n = 20	27-10-2019 01:45:02
10	n = 20	27-10-2019 01:45:02
11	n = 20	27-10-2019 01:45:02
12	n = 20	27-10-2019 01:47:37
13	n = 20	27-10-2019 01:47:37
14	n = 20	27-10-2019 01:47:37
15	n = 20	27-10-2019 01:47:37
16	n = 20	27-10-2019 01:47:37
17	n = 20	27-10-2019 01:47:37
18	n = 20	27-10-2019 01:47:37
19	n = 20	27-10-2019 01:47:37
20	n = 20	27-10-2019 01:47:37
21	n = 20	27-10-2019 01:47:37
22	n = 20	27-10-2019 17:03:02
23	n = 20	27-10-2019 17:03:02
24	n = 20	27-10-2019 17:03:02
25	n = 20	27-10-2019 17:03:02
26	n = 20	27-10-2019 17:03:02
27	n = 20	27-10-2019 17:03:02
28	n = 20	27-10-2019 17:03:02
29	n = 20	27-10-2019 17:03:02
30	n = 20	27-10-2019 17:03:02
31	n = 20	27-10-2019 17:03:02

Figure 6: Output for problem 2

4 Results and conclusions

Lab assignment was a great task to practice course knowledge about concurrent and distributed systems. It was created an object-oriented application, with specific thread class which extends or implements Thread class or Runnable interface, overriding run() method, use static and volatile variables (in Counter class, n is a static, volatile variable to access variable from the same memory address/location). The challenging achievement was to debug the code because a multithread application execute every thread how is programmed by Scheduler of operating system and at every running may appear a different scenario.

References

[1] M. Ben-Ari *Principles of Concurrent and Distributed Programming*. Pearson Education, Second Edition, 2006.

[2] Scott Oaks, Henry Wong *Java Threads*. O'Reilly Media, Inc, USA, Third Edition.

[3] Maurice Herlihy, Nir Shavit *The art of multiprocessor programming*. the art of multiprocessor programming, Revised First Edition, 2012.

[4] GeeksforGeeks, https://www.geeksforgeeks.org/, accessed in October 2019.

[5] Introduction to Javadoc, https://www.baeldung.com/javadoc, accessed in October 2019.

[6] Generate a Javadoc reference, https://www.jetbrains.com/help/idea/working-with-code-documentation.html, accessed in October 2019.

[7] Overleaf, https://www.overleaf.com/learn, accessed in October 2019.