Concurrent and Distributed Systems

Laboratory Assignment

October 27, 2019

Student: Marcu Andrei Cristian

Computers and Information Technology

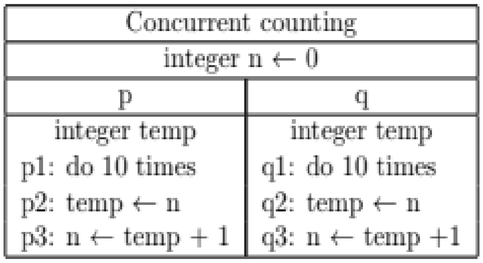
CEN 3.2 A

3rd Year

Problems statements

Problem 1 (50p)
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. You should consider 2 solutions:
- 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 .
- 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 j k the set Mj contains all those elements from M who by divided with k+1 give remainder j. Also it is considered that k+1 belongs to M1. Each thread j will determine the prime numbers from set Mj .
Which of the solutions is better and why?

Problem 2 (50p)
What values can n have at the end of the concurrent counting algorithm execution?
Assignment: Implement this algorithm in Java. What values do you get at the end for n?
Assignment: 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 ← 0	
p	q
integer temp p1: do 10 times p2: temp ← n p3: n ← temp + 1	integer temp q1: do 10 times q2: temp ← n q3: n ← temp +1

Problem1: As we've seen in the statement we need to find all the prime numbers in the [1,n] interval using k threads. Each thread should iterate through a smaller interval which can be defined using the generalised formula Ik = [(k-1)q + r + 1, kq + r]. The formula will skip the numbers from [1, r] so we need to treat that separately.

For example for an input like this: n = 10 k = 2.
The prime numbers in the [1,n] interval are : 2, 3, 5, 7
q = 10/2 = 5 ; r = 10 % 2 = 0 . Because r is 0 we don't need to check the ones from [1,r]
We've got k = 2 so 2 threads.
T1 = [(1 - 1) * 5 + 0 + 1, 1 * 5 + 0] = [1, 5]
T2 = [(2 - 1) * 5 + 0 + 1, 2 * 5 + 0] = [6, 10]
Each of those threads will iterate through the smaller intervals presented before.

For example for an input like this case: n = 8 k = 3.
The prime numbers in the [1,n] interval are : 2, 3, 5, 7
q = 10/2 = 2 ; r = 10 % 2 =2 . Because r is 2 we need to check the ones from [1,r]
We've got k = 3 so 3 threads.
T1 = [(1 - 1) * 2 + 2 + 1, 1 * 2 + 2] = [3, 4]
T2 = [(2 - 1) * 2 + 2 + 1, 2 * 2 + 2] = [5, 6]
T3 = [(3 - 1) * 2 + 2 + 1, 3 * 2 + 2] = [7, 8]
So the numbers from 1 to r (1 to 2) will not be checked. I've treated this case separately. Thread number 1 will check those numbers
Each of those threads will iterate through the smaller intervals presented before.

Problem2: As we've seen in the statement we have 2 threads incrementing the same variable n 10 times. N is a global variable(static in my solution) so it will have the same value for all instances of the ConcurrentThread class.
For example the result for the input presented in the solution (2 threads, 10 iterations each) n will always reach the value 20.
I'll explain more examples later on my presentation.
Now that we've clarified the Problems Statements we can move on to the implementation.

Solution Problem 1

Method1

```
//----------------------------PrimeThreadMethod1.java----------------------------
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class PrimeThreadMethod1 extends Thread{
       private int threadNumber; // thread it
       private int n, k, q, r;

       public static volatile List<Integer> primeNumbers =
           Collections.synchronizedList (new ArrayList<Integer>());
       // In this list we add prime numbers everytime we find one.
       //It's a synchronizedList in order to be thread-safe.
       PrimeThreadMethod1(int n, int k, int threadNumber)
       {
              this.n = n;
              this.k = k;
              this.threadNumber = threadNumber;
              this.q = this.n / this.k;
              this.r = n % k;

              if(k > n)
                     throw new ArrayIndexOutOfBoundsException("More
                         threads than numbers");
              // We can't have more threads than numbers.
       }
       public void run()
       {
              //The general formula doesn't treat the case from [1, r].
                  So i've added the numbers from [1, r] to thread
                  number 1.
              int rest;
              if(threadNumber == 1)
                     rest = 0;
              else
                     rest = r;
              for( int i = (threadNumber - 1) * q + rest + 1; i <=
                  threadNumber * q + r; i++)
                     //We split the interval in multiple invervals
                         using the threadNumber
              {
                     Boolean primeFlag = false;
                     //Boolean value that checks if the number is prime
                         or not
                     int sqrt = (int) (Math.ceil(Math.sqrt((double) i
                            )));
                        // sqrt in order to iterate from 2 to sqrt
                        for( int j = 2 ; j <= sqrt; j++ )
                        {
                                if( i == 2 )// 2 is the first prime number
                                {
                                        //System.out.println("Thread number
                                            " + threadNumber + " added the
                                            number :" + i);
                                        primeFlag = true;
                                        primeNumbers.add(i);
                                        // we add 2 in the ArrayList, we
                                            make the flag true so we don't
                                            add 2 two times, and we break
                                            the loop because the next
                                            values are pointless.
                                        break;
                                }
                                if(i % j == 0)
                                {
                                        primeFlag = true;
                                        //we've found a divider, the number
                                            is not prime. We break the loop
                                            because the next values are
                                            pointless.
                                        break;
                                }
                        }
                        if(primeFlag == false && i != 1)
                        {
                                //System.out.println("Thread number " +
                                    threadNumber + " added the number :" +
                                    i);
                                primeNumbers.add(i);
                                //If the number it's not prime and it is
                                    different than 1 we add it in the
                                    ArrayList.
                        }
                }
        }
}
```

Method2

```
//----------------------------PrimeThreadMethod2.java----------------------------
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map.Entry;
import java.util.concurrent.ConcurrentHashMap;

public class PrimeThreadMethod2 extends Thread{
        private int threadNumber; // Thread identifier

        public static volatile ConcurrentHashMap<Integer,
            ArrayList<Integer> > splitedInterval = new
            ConcurrentHashMap<Integer, ArrayList<Integer>>();
        // I've used a map which keeps the numbers each thread should
            check
        // So each splitedInterval[threadId] will have an arrayList
            which contains the numbers it should check.
        // This map is created in main. If we create it inside the class
            the code will be executed multiple times.
        // I've used ConcurrentHashMap in order to be thread safe.
        public static volatile List<Integer> primeNumbers =
            Collections.synchronizedList (new ArrayList<Integer>());
        // The list contains the prime numbers we find. It's a
            synchronizedList which is thread-safe.
        PrimeThreadMethod2(int n, int k, int threadNumber)
        {
                this.threadNumber = threadNumber;
                if(k > n)
                        throw new ArrayIndexOutOfBoundsException("More
                            threads than numbers");
                // We can't have more threads than numbers
        }
        public void run()
        {
                for( Entry<Integer, ArrayList<Integer>> entry:
                    PrimeThreadMethod2.splitedInterval.entrySet())
                        // We iterate through all the elements in map
            {
                if(threadNumber - 1 == entry.getKey())
                        // We check if we found the key which is equal to
                            threadId
                {
                        for(int i: entry.getValue())
                                // We get the arrayList which has the key
                                    equal to threadId
                        {
                                        Boolean primeFlag = false;
                                        //Boolean value that checks if the
                                            number is prime or not
                                        int sqrt = (int)
                                            (Math.ceil(Math.sqrt((double) i
                                            )));
                                        // sqrt in order to iterate from 2
```

```
    to sqrt
for( int j = 2 ; j <= sqrt; j++ )
{
       if( i == 2 )
              // 2 is the first
                  prime number
       {
              //System.out.println("Thread
                  number " +
                  threadNumber + "
                  added the number
                  :" + i);
              primeFlag = true;
              primeNumbers.add(i);
              // we add 2 in the
                  ArrayList, we
                  make the flag
                  true so we don’t
                  add 2 two times,
                  and we break the
                  loop because the
                  next values are
                  pointless.
              break;
       }
       if(i % j == 0)
       {
              primeFlag = true;
              //we’ve found a
                  divider, the
                  number is not
                  prime. We break
                  the loop because
                  the next values
                  are pointless.
              break;
       }
}
if(primeFlag == false && i != 1)
{
       //System.out.println("Thread
           number " + threadNumber
           + " added the number :"
           + i);
       primeNumbers.add(i);
       //If the number it’s not
           prime and it is
           different than 1 we add
           it in the ArrayList.
}
```

```
                     }
              }
              }
       }
}
```

Main

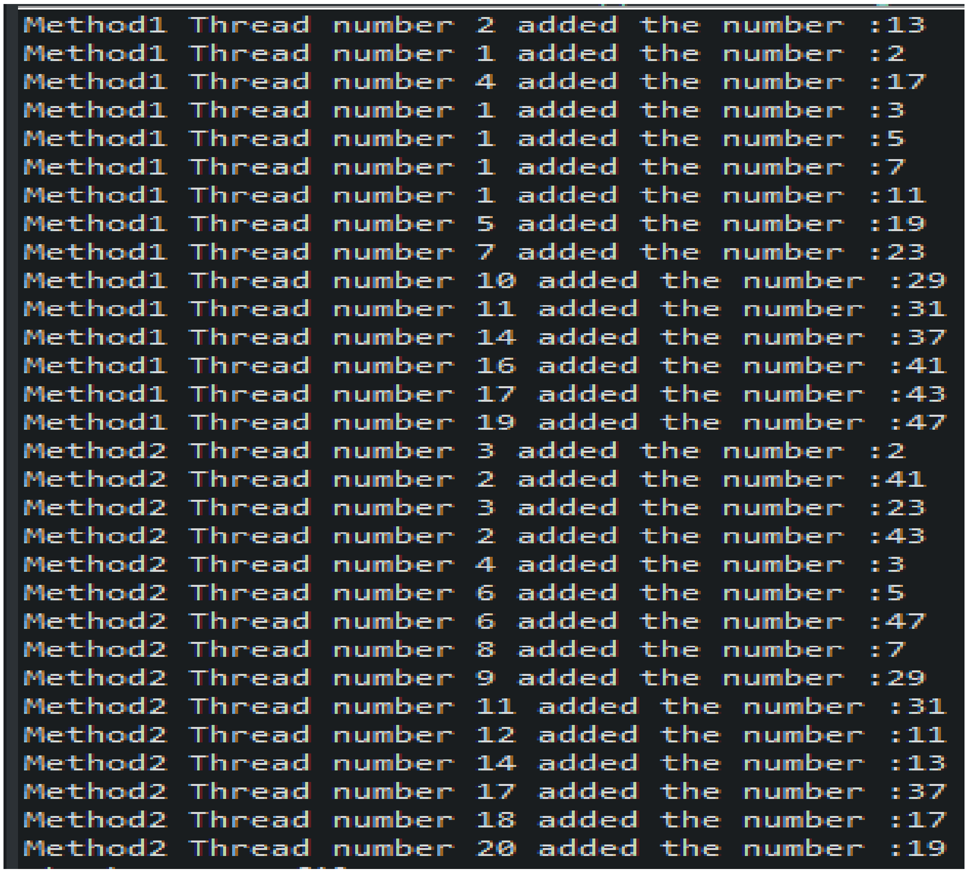
```
//----------------------------MainThread.java----------------------------
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.Random;

public class MainThread {

       public static void main(String[] args) throws
           FileNotFoundException, UnsupportedEncodingException {
              PrintWriter writer = new
                  PrintWriter("execution-output.txt", "UTF-8"); //
                  writer used to print in file
              Random rand = new Random();
              // random instance
              int n = rand.nextInt(2000000);
              // random int from 1 to 2000000
           //int k = rand.nextInt(2000000);
           //int n = 37028;
           int k = 2000;

           while(k > n)
           {
              k = rand.nextInt(2000);
              // we make sure that we don't have more threads than
                  numbers
           }

           long startTime = System.currentTimeMillis();
           // the time when we start the threads

           PrimeThreadMethod1[] t = new PrimeThreadMethod1[k];
           for(int i = 1; i <= k; i++ ){
             t[i-1] = new PrimeThreadMethod1(n, k, i);
             //We create k threads and start them
             t[i-1].start();
           }
           //We wait for the threads to finish
```

```
for(int i = 0; i < k; i++ ){
   try {
      t[i].join();
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
//the time when we finish
long stopTime = System.currentTimeMillis();
//the time of execution
long elapsedTime = stopTime - startTime;

//We print the result in file
writer.println("#####################Method1#####################");
writer.println("Numbers from 1 to " + n);
writer.println("Number of threads is " + k);
writer.println("Execution time : " + elapsedTime/1000 + "," +
    elapsedTime%1000 + "s");
writer.println("All the prime numbers:" +
    PrimeThreadMethod1.primeNumbers.toString());
//Random lines to separate the methods
writer.println("##########################################################################
writer.println("##########################################################################
writer.println("##########################################################################
writer.println("##########################################################################
writer.println("##########################################################################

for(int i = 1; i <= n; i++)
    // We create the splittedInterval map
{
    if(i % (k+1) == 0 && i > (k + 1))
    {
           continue;
           // If i is divided by (k+1) without rest and it is
               bigger than (k+1) it can't be prime number
    }
    if(!PrimeThreadMethod2.splitedInterval.containsKey(i %
        (k+1)))
           // The key will be the threadId
    {
           PrimeThreadMethod2.splitedInterval.put(i % (k+1),
               new ArrayList<Integer>());
           //We check if we already created the key in the
               map. If not we create it
    }
    if(i % (k + 1) == k) {
           PrimeThreadMethod2.splitedInterval.get(1).add(i);
           // My threads are from 0 to k - 1. The operation %
               (k + 1) will generate results from 0 to k.
           // So if the result is k we move that value to
```

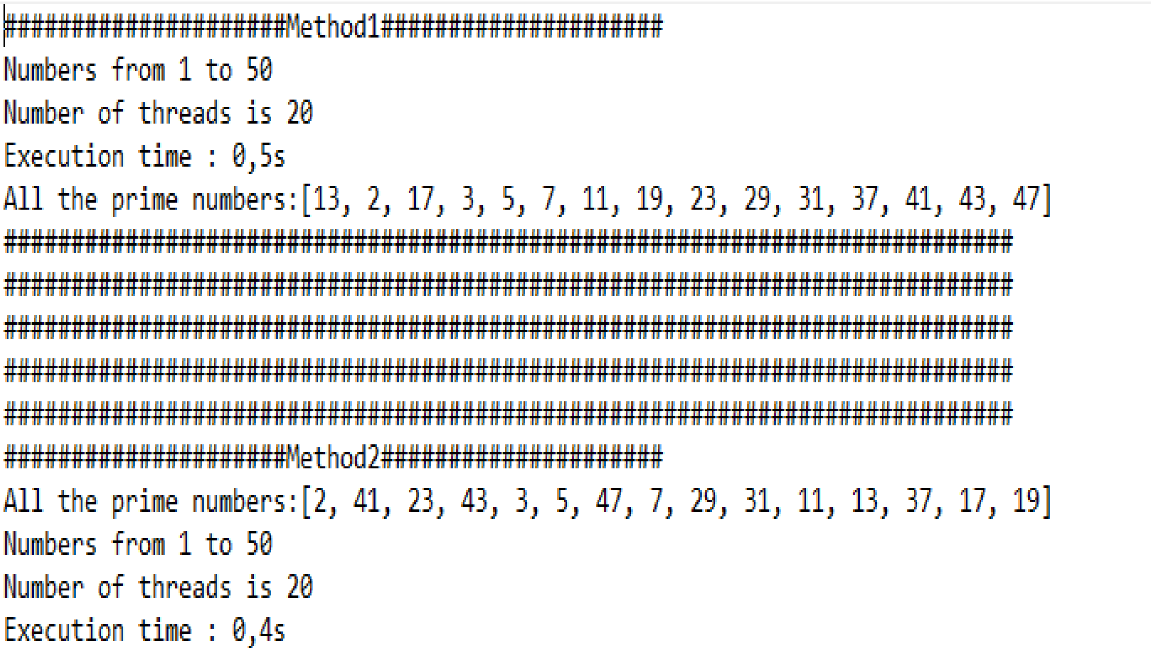
```
                          thread 1.
                      continue;
               }
               //We add the value in the arrayList
               PrimeThreadMethod2.splitedInterval.get(i % (k+1)).add(i);
           }


           startTime = System.currentTimeMillis();
           // the time when we start the threads

           PrimeThreadMethod2[] t2 = new PrimeThreadMethod2[k];
           for(int i = 1; i <= k; i++ ){
              t2[i-1] = new PrimeThreadMethod2(n, k, i);
              //We create k threads and start them
              t2[i-1].start();
           }
           //We wait for the threads to finish
           for(int i = 0; i < k; i++ ){
              try {
                  t2[i].join();
              } catch (InterruptedException e) {
                  e.printStackTrace();
              }
           }
           //the time when we finish
           stopTime = System.currentTimeMillis();
           elapsedTime = stopTime - startTime;
           //the time of execution

           //We print the result in file
           writer.println("#####################Method2#####################");
           writer.println("All the prime numbers:" +
               PrimeThreadMethod2.primeNumbers.toString());
           writer.println("Numbers from 1 to " + n);
           writer.println("Number of threads is " + k);
           writer.println("Execution time : " + elapsedTime/1000 + "," +
               elapsedTime%1000 + "s");

           System.out.println("Check output file");
           // Console warning that tells us the results are in folder.
           writer.close();
           // We close the file writer.
       }
}
```

Experiments and results for problem 1

1. Example with n = 50, k = 20.

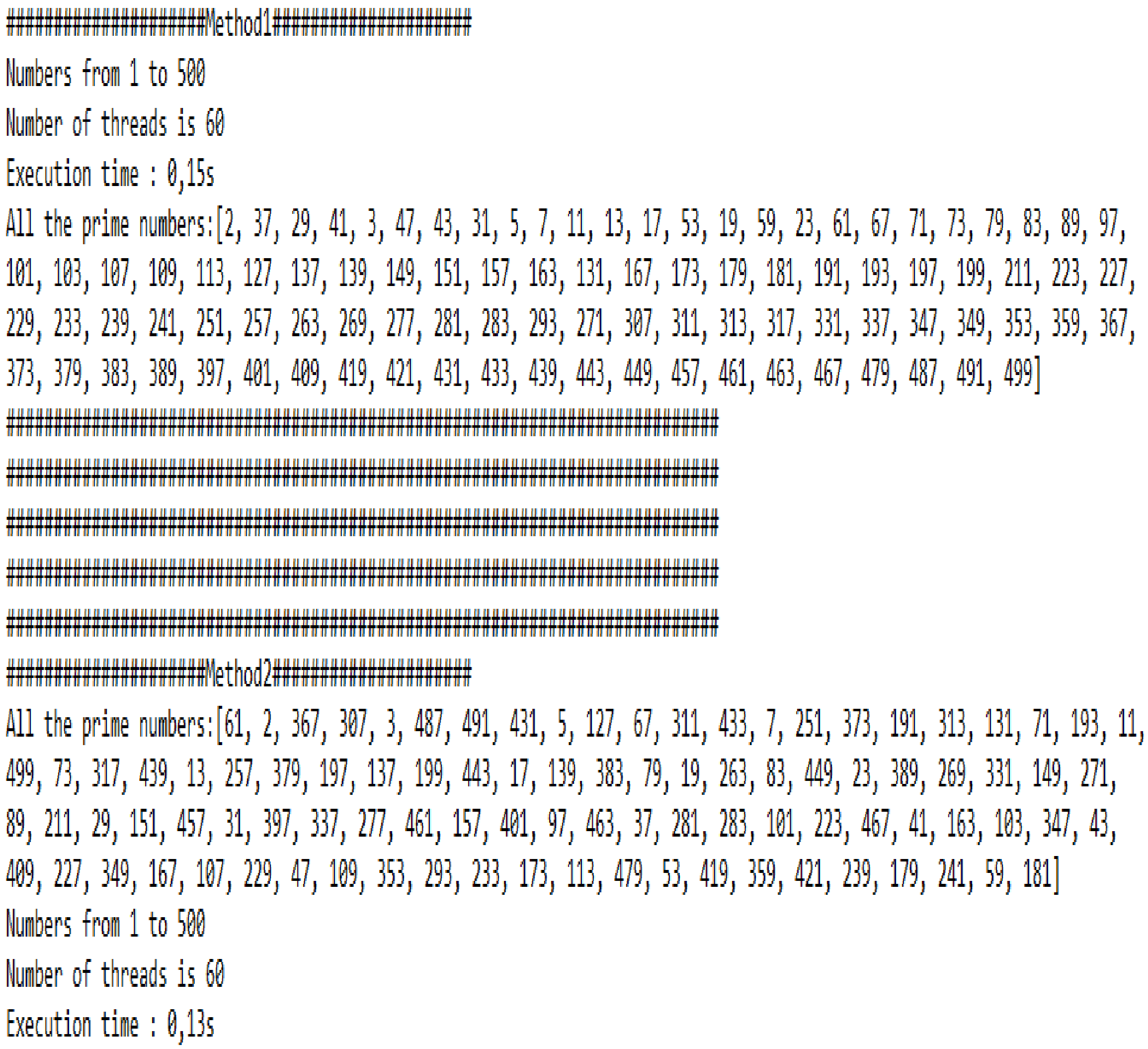


```
Method1 Thread number 2 added the number :13
Method1 Thread number 1 added the number :2
Method1 Thread number 4 added the number :17
Method1 Thread number 1 added the number :3
Method1 Thread number 1 added the number :5
Method1 Thread number 1 added the number :7
Method1 Thread number 1 added the number :11
Method1 Thread number 5 added the number :19
Method1 Thread number 7 added the number :23
Method1 Thread number 10 added the number :29
Method1 Thread number 11 added the number :31
Method1 Thread number 14 added the number :37
Method1 Thread number 16 added the number :41
Method1 Thread number 17 added the number :43
Method1 Thread number 19 added the number :47
Method2 Thread number 3 added the number :2
Method2 Thread number 2 added the number :41
Method2 Thread number 3 added the number :23
Method2 Thread number 2 added the number :43
Method2 Thread number 4 added the number :3
Method2 Thread number 6 added the number :5
Method2 Thread number 6 added the number :47
Method2 Thread number 8 added the number :7
Method2 Thread number 9 added the number :29
Method2 Thread number 11 added the number :31
Method2 Thread number 12 added the number :11
Method2 Thread number 14 added the number :13
Method2 Thread number 17 added the number :37
Method2 Thread number 18 added the number :17
Method2 Thread number 20 added the number :19
```



```
#####################Method1#####################
Numbers from 1 to 50
Number of threads is 20
Execution time : 0,5s
All the prime numbers:[13, 2, 17, 3, 5, 7, 11, 19, 23, 29, 31, 37, 41, 43, 47]
###########################################################################
###########################################################################
###########################################################################
###########################################################################
###########################################################################
#####################Method2#####################
All the prime numbers:[2, 41, 23, 43, 3, 5, 47, 7, 29, 31, 11, 13, 37, 17, 19]
Numbers from 1 to 50
Number of threads is 20
Execution time : 0,4s
```

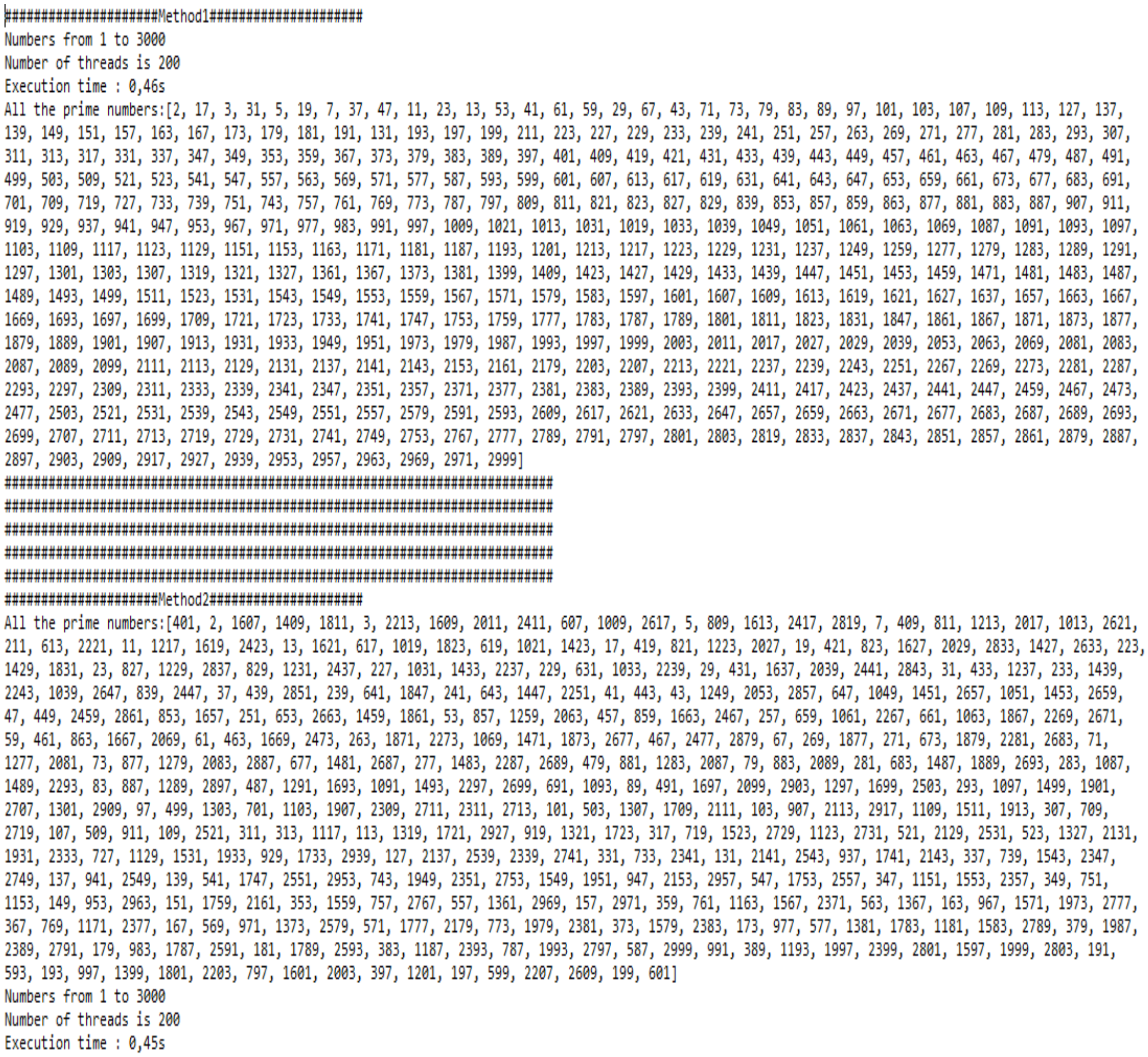
2. Example with n = 500, k = 60.



```
#####################Method1#####################
Numbers from 1 to 500
Number of threads is 60
Execution time : 0,15s
All the prime numbers:[2, 37, 29, 41, 3, 47, 43, 31, 5, 7, 11, 13, 17, 53, 19, 59, 23, 61, 67, 71, 73, 79, 83, 89, 97,
101, 103, 107, 109, 113, 127, 137, 139, 149, 151, 157, 163, 131, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227,
229, 233, 239, 241, 251, 257, 263, 269, 277, 281, 283, 293, 271, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,
373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499]
###########################################################################
###########################################################################
###########################################################################
###########################################################################
###########################################################################
#####################Method2#####################
All the prime numbers:[61, 2, 367, 307, 3, 487, 491, 431, 5, 127, 67, 311, 433, 7, 251, 373, 191, 313, 131, 71, 193, 11,
499, 73, 317, 439, 13, 257, 379, 197, 137, 199, 443, 17, 139, 383, 79, 19, 263, 83, 449, 23, 389, 269, 331, 149, 271,
89, 211, 29, 151, 457, 31, 397, 337, 277, 461, 157, 401, 97, 463, 37, 281, 283, 101, 223, 467, 41, 163, 103, 347, 43,
409, 227, 349, 167, 107, 229, 47, 109, 353, 293, 233, 173, 113, 479, 53, 419, 359, 421, 239, 179, 241, 59, 181]
Numbers from 1 to 500
Number of threads is 60
Execution time : 0,13s
```

For bigger examples I can't print which thread adds the number because the result is too big.

3. Example with n = 3000, k = 200.



```
####################Method1####################
Numbers from 1 to 3000
Number of threads is 200
Execution time : 0,46s
All the prime numbers:[2, 17, 3, 31, 5, 19, 7, 37, 47, 11, 23, 13, 53, 41, 61, 59, 29, 67, 43, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 137,
139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 131, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307,
311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691,
701, 709, 719, 727, 733, 739, 751, 743, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1021, 1013, 1031, 1019, 1033, 1039, 1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097,
1103, 1109, 1117, 1123, 1129, 1151, 1153, 1163, 1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237, 1249, 1259, 1277, 1279, 1283, 1289, 1291,
1297, 1301, 1303, 1307, 1319, 1321, 1327, 1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433, 1439, 1447, 1451, 1453, 1459, 1471, 1481, 1483, 1487,
1489, 1493, 1499, 1511, 1523, 1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579, 1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637, 1657, 1663, 1667,
1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741, 1747, 1753, 1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873, 1877,
1879, 1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993, 1997, 1999, 2003, 2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083,
2087, 2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161, 2179, 2203, 2207, 2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269, 2273, 2281, 2287,
2293, 2297, 2309, 2311, 2333, 2339, 2341, 2347, 2351, 2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399, 2411, 2417, 2423, 2437, 2441, 2447, 2459, 2467, 2473,
2477, 2503, 2521, 2531, 2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609, 2617, 2621, 2633, 2647, 2657, 2659, 2663, 2671, 2677, 2683, 2687, 2689, 2693,
2699, 2707, 2711, 2713, 2719, 2729, 2731, 2741, 2749, 2753, 2767, 2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843, 2851, 2857, 2861, 2879, 2887,
2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963, 2969, 2971, 2999]
########################################################################
########################################################################
########################################################################
########################################################################
########################################################################
####################Method2####################
All the prime numbers:[401, 2, 1607, 1409, 1811, 3, 2213, 1609, 2011, 2411, 607, 1009, 2617, 5, 809, 1613, 2417, 2819, 7, 409, 811, 1213, 2017, 1013, 2621,
211, 613, 2221, 11, 1217, 1619, 2423, 13, 1621, 617, 1019, 1823, 619, 1021, 1423, 17, 419, 821, 1223, 2027, 19, 421, 823, 1627, 2029, 2833, 1427, 2633, 223,
1429, 1831, 23, 827, 1229, 2837, 829, 1231, 2437, 227, 1031, 1433, 2237, 229, 631, 1033, 2239, 29, 431, 1637, 2039, 2441, 2843, 31, 433, 1237, 233, 1439,
2243, 1039, 2647, 839, 2447, 37, 439, 2851, 239, 641, 1847, 241, 643, 1447, 2251, 41, 443, 43, 1249, 2053, 2857, 647, 1049, 1451, 2657, 1051, 1453, 2659,
47, 449, 2459, 2861, 853, 1657, 251, 653, 2663, 1459, 1861, 53, 857, 1259, 2063, 457, 859, 1663, 2467, 257, 659, 1061, 2267, 661, 1063, 1867, 2269, 2671,
59, 461, 863, 1667, 2069, 61, 463, 1669, 2473, 263, 1871, 2273, 1069, 1471, 1873, 2677, 467, 2477, 2879, 67, 269, 1877, 271, 673, 1879, 2281, 2683, 71,
1277, 2081, 73, 877, 1279, 2083, 2887, 677, 1481, 2687, 277, 1483, 2287, 2689, 479, 881, 1283, 2087, 79, 883, 2089, 281, 683, 1487, 1889, 2693, 283, 1087,
1489, 2293, 83, 887, 1289, 2897, 487, 1291, 1693, 1091, 1493, 2297, 2699, 691, 1093, 89, 491, 1697, 2099, 2903, 1297, 1699, 2503, 293, 1097, 1499, 1901,
2707, 1301, 2909, 97, 499, 1303, 701, 1103, 1907, 2309, 2711, 2311, 2713, 101, 503, 1307, 1709, 2111, 103, 907, 2113, 2917, 1109, 1511, 1913, 307, 709,
2719, 107, 509, 911, 109, 2521, 311, 313, 1117, 113, 1319, 1721, 2927, 919, 1321, 1723, 317, 719, 1523, 2729, 1123, 2731, 521, 2129, 2531, 523, 1327, 2131,
1931, 2333, 727, 1129, 1531, 1933, 929, 1733, 2939, 127, 2137, 2539, 2339, 2741, 331, 733, 2341, 131, 2141, 2543, 937, 1741, 2143, 337, 739, 1543, 2347,
2749, 137, 941, 2549, 139, 541, 1747, 2551, 2953, 743, 1949, 2351, 2753, 1549, 1951, 947, 2153, 2957, 547, 1753, 2557, 347, 1151, 1553, 2357, 349, 751,
1153, 149, 953, 2963, 151, 1759, 2161, 353, 1559, 757, 2767, 557, 1361, 2969, 157, 2971, 359, 761, 1163, 1567, 2371, 563, 1367, 163, 967, 1571, 1973, 2777,
367, 769, 1171, 2377, 167, 569, 971, 1373, 2579, 571, 1777, 2179, 773, 1979, 2381, 373, 1579, 2383, 173, 977, 577, 1381, 1783, 1181, 1583, 2789, 379, 1987,
2389, 2791, 179, 983, 1787, 2591, 181, 1789, 2593, 383, 1187, 2393, 787, 1993, 2797, 587, 2999, 991, 389, 1193, 1997, 2399, 2801, 1597, 1999, 2803, 191,
593, 193, 997, 1399, 1801, 2203, 797, 1601, 2003, 397, 1201, 197, 599, 2207, 2609, 199, 601]
Numbers from 1 to 3000
Number of threads is 200
Execution time : 0,45s
```

For bigger examples I can't print which thread adds the number because the result is too big.

Random generated bigger examples will be found in the txt files from the Experimental data and results directory.

Conclusions Problem 1

Working on this problem, I have acquired lots of knowledge about thread programming, data containers which are thread-safe and the problems normal data containers can cause. Both methods are equal if we talk about the execution time, sometimes the first method finished faster, sometimes the second one.

Solution Problem 2

Thread Class

```
//----------------------------ConcurrentThreadMethod1.java----------------------------

public class ConcurrentThread extends Thread {
       static volatile int n = 0; // I've used static in order to keep
           it's value in all the instances of ConcurrentThread class.
       //Also it is a volatile variable because the cache memory of
           each thread can alter the value, so using volatile will make
           it thread-safe, the value being saved in memory instead of
           CPU cache.
       String message;// To keep an identification for threads.
       private int temp;

       public ConcurrentThread(String message)
       {
              this.message = message; // thread id
              this.temp = 0; // temp value is initially 0
       }

       public void run() {
              for(int i = 1 ; i <= 10; i++)
              {
                     this.temp = n;
                     n = this.temp + 1;
                     System.out.println("The value of n in " +
                         this.message + " iteration " + i + " is " + n);
              }
       }
}
```

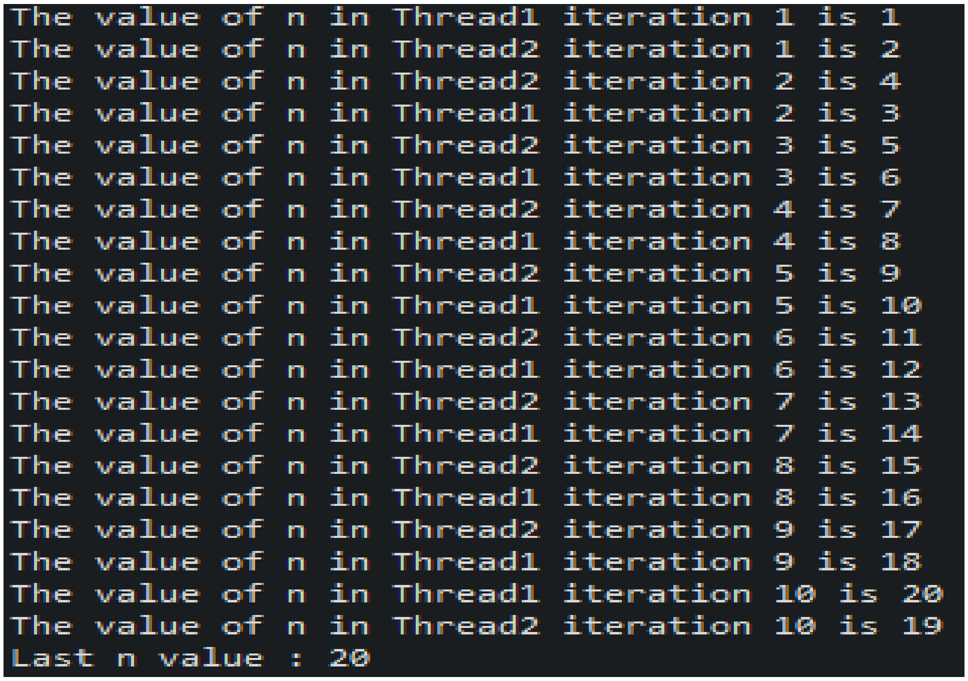
Main

```
//----------------------------MainThread.java----------------------------

public class MainThread {

       public static void main(String[] args) {
              //We create 2 threads
              ConcurrentThread Thread1 = new
                  ConcurrentThread("Thread1");
       ConcurrentThread Thread2 = new ConcurrentThread("Thread2");
       //We start both threads
       Thread1.start();
```

```
        Thread2.start();
        //We wait until threads finish
        try {
                Thread1.join();
                Thread2.join();
        }
        catch(InterruptedException e){
                e.printStackTrace();
        }
        //We print the value of n

        System.out.println("Last n value : " + ConcurrentThread.n);
        }

}
```

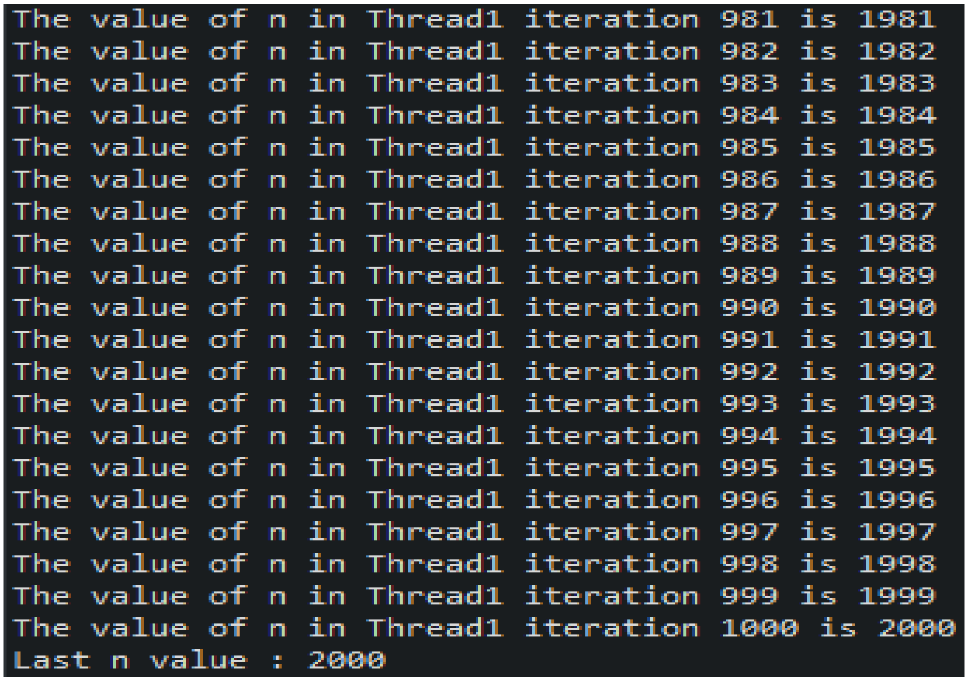
Experiments and results for problem 2

1. Example with 10 iterations.



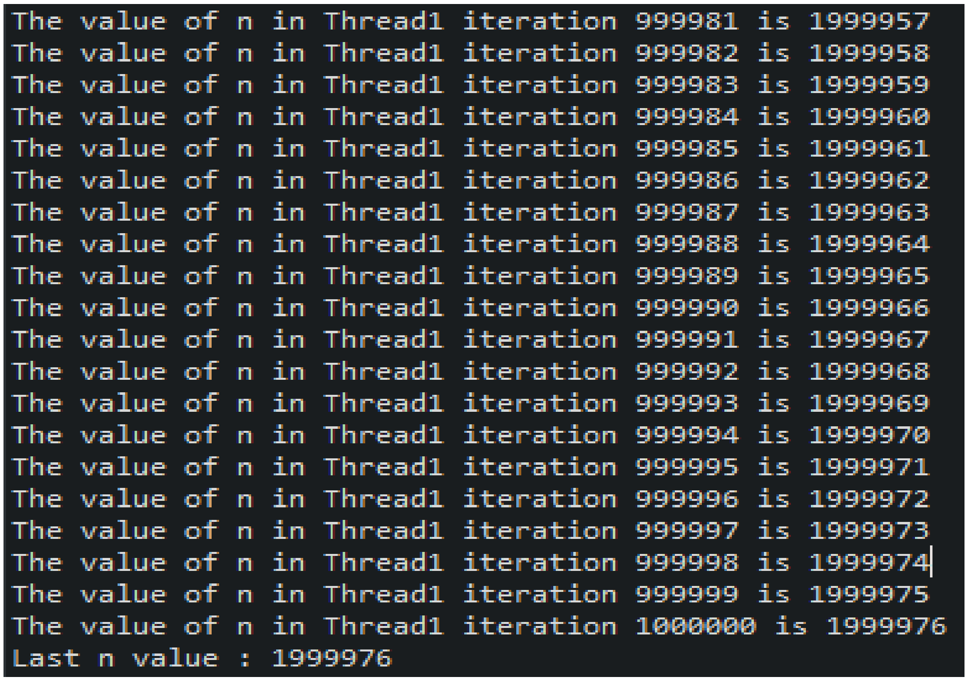
```
The value of n in Thread1 iteration 1 is 1
The value of n in Thread2 iteration 1 is 2
The value of n in Thread2 iteration 2 is 4
The value of n in Thread1 iteration 2 is 3
The value of n in Thread2 iteration 3 is 5
The value of n in Thread1 iteration 3 is 6
The value of n in Thread2 iteration 4 is 7
The value of n in Thread1 iteration 4 is 8
The value of n in Thread2 iteration 5 is 9
The value of n in Thread1 iteration 5 is 10
The value of n in Thread2 iteration 6 is 11
The value of n in Thread1 iteration 6 is 12
The value of n in Thread2 iteration 7 is 13
The value of n in Thread1 iteration 7 is 14
The value of n in Thread2 iteration 8 is 15
The value of n in Thread1 iteration 8 is 16
The value of n in Thread2 iteration 9 is 17
The value of n in Thread1 iteration 9 is 18
The value of n in Thread1 iteration 10 is 20
The value of n in Thread2 iteration 10 is 19
Last n value : 20
```

2. Example with 1000 iterations.



```
The value of n in Thread1 iteration 981 is 1981
The value of n in Thread1 iteration 982 is 1982
The value of n in Thread1 iteration 983 is 1983
The value of n in Thread1 iteration 984 is 1984
The value of n in Thread1 iteration 985 is 1985
The value of n in Thread1 iteration 986 is 1986
The value of n in Thread1 iteration 987 is 1987
The value of n in Thread1 iteration 988 is 1988
The value of n in Thread1 iteration 989 is 1989
The value of n in Thread1 iteration 990 is 1990
The value of n in Thread1 iteration 991 is 1991
The value of n in Thread1 iteration 992 is 1992
The value of n in Thread1 iteration 993 is 1993
The value of n in Thread1 iteration 994 is 1994
The value of n in Thread1 iteration 995 is 1995
The value of n in Thread1 iteration 996 is 1996
The value of n in Thread1 iteration 997 is 1997
The value of n in Thread1 iteration 998 is 1998
The value of n in Thread1 iteration 999 is 1999
The value of n in Thread1 iteration 1000 is 2000
Last n value : 2000
```

3. Example with 1000000 iterations.



```
The value of n in Thread1 iteration 999981 is 1999957
The value of n in Thread1 iteration 999982 is 1999958
The value of n in Thread1 iteration 999983 is 1999959
The value of n in Thread1 iteration 999984 is 1999960
The value of n in Thread1 iteration 999985 is 1999961
The value of n in Thread1 iteration 999986 is 1999962
The value of n in Thread1 iteration 999987 is 1999963
The value of n in Thread1 iteration 999988 is 1999964
The value of n in Thread1 iteration 999989 is 1999965
The value of n in Thread1 iteration 999990 is 1999966
The value of n in Thread1 iteration 999991 is 1999967
The value of n in Thread1 iteration 999992 is 1999968
The value of n in Thread1 iteration 999993 is 1999969
The value of n in Thread1 iteration 999994 is 1999970
The value of n in Thread1 iteration 999995 is 1999971
The value of n in Thread1 iteration 999996 is 1999972
The value of n in Thread1 iteration 999997 is 1999973
The value of n in Thread1 iteration 999998 is 1999974
The value of n in Thread1 iteration 999999 is 1999975
The value of n in Thread1 iteration 1000000 is 1999976
Last n value : 1999976
```

Bigger examples will be found in the txt files from the Experimental data and results directory.

Conclusions Problem 2

The conclusion of the examples presented before is that n should be 2 * number of iterations. Even though when we've got a big number of iterations n won't finish with the value we expect. This happens because of the memory synchronization. Even if we use volatile variable it will still have the same value. Thread race conditions can make the increment operation to be missed. In order to prevent this problem we should use an AtomicInteger which is thread-safe. If we use "synchronize" only one thread can modify the variable at a time and the problem would be solved.

References

https://en.wikipedia.org/wiki/Race$_c$*onditionSoftware*

*https : //stackoverflow.com/questions/*14983847*/make−multiple−threads − use − and − change − the − same − variable*

*https : //docs.oracle.com/javase/*8*/docs/api/java/util/concurrent/ConcurrentHashMap.h*

https : //www.geeksforgeeks.org/collections−synchronizedlist−method − in − java − with − examples/

https : //www.geeksforgeeks.org/synchronized − in − java/