cphbusiness
COPENHAGEN BUSINESS ACADEMY

Exploration and Presentation Assignment 3

21. april 2021

Forfattere
Rúni Vedel Niclasen - cph-rn118
Camilla Jenny Valerius Staunstrup - cph-cs340
Asger Thorsboe Lundblad - cph-al217

Indhold

1 Opgavebeskrivelse

Work either on the following project (https://github.com/CPHBusinessSoftUFO/ letterfrequencies), or on a an earlier project of yours (something you can optimize).

1.1 Task 1

- Find a point in your program that can be optimized (for speed), for example by using a profiler.
- Make a measurement of the point to optimize, for example by running a number of times, and calculating the mean and standard deviation (see the paper from Sestoft).
- If you work on the letterfrequencies program, make it at least 50% faster.

1.2 Requirements

- Short introduction of the program and the part to be optimized
- Documentation of the current performance
- Explanation of bottleneck(s)
- A hypothesis of what causes the problem
- A changed program with better performance
- Documentation of the new performance
- Written in LaTeX

2 Besvarelse

All test have been run on the same machine with the following specifications:

- **CPU** - Intel Core i7-4770k 4-Core (8 threads) 3.50 GHz
- **RAM** - Corsair 16 GB DDR3 1600 MHz
- **OS** - Windows 10 Pro 64-bit
- **JVM** - Oracle Corporation 16.0

2.1 Original Program Introduction

The Original Program to be optimized supplied by the school (the link in the assignment description) have the following description:

Simple program used to illustrate performance problems. You should be able to optimize this program to run about twice as fast. [1].

The program source consists of the foundation trilogy by Isaac Asimov (roughly 25.000 lines of text) and a single Java file. The file reads the text and returns the frequencies of the letters within the text.

The Java program is purposefully written in an inefficient way. It has two helper methods and at its core it presents a list of the letter frequencies between a-zA-Z.

Below you can find a sample of the output of the program:

Letter	Frequencies
E	118313
T	87190
A	76011
O	74907
N	67908
...	...
K	6630
X	1457
Q	1017
J	806
Z	674

Tabel 1: The 5 most and least frequent letters in the dataset

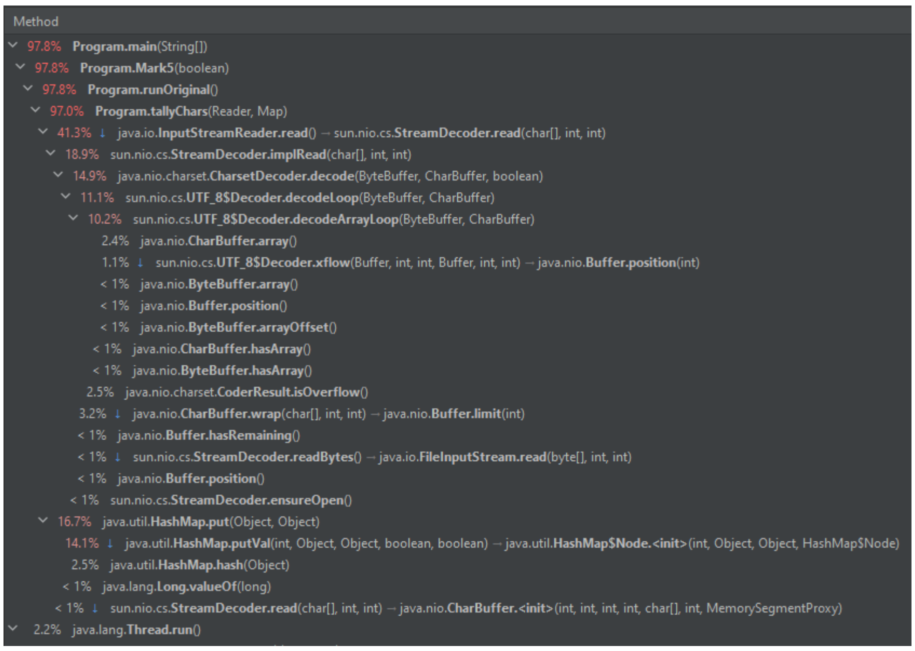
2.2 Current Performance

We measured the program using the Mark5 method supplied by Sestoft.[2]
This is the current console readings when the program is run without any changes to the code. So a baseline run time of 51 miliseconds with a standard deviation of 1.07 miliseconds.

```
-------------------------------------------
Mean               Sdev               Count
-------------------------------------------
60,2 ms +/-    20,39 ms                 2
54,7 ms +/-     4,41 ms                 4
51,4 ms +/-     1,62 ms                 8
51,1 ms +/-     1,90 ms                16
50,7 ms +/-     0,28 ms                32
50,7 ms +/-     0,64 ms                64
51,0 ms +/-     1,07 ms               128
-------------------------------------------
```

2.3 Bottleneck(s)

Profiling with the Java Flight Recorder in IntelliJ gives us the following indication of the programs shortcommings:



```
Method
97.8% Program.main(String[])
 97.8% Program.Mark5(boolean)
  97.8% Program.runOriginal()
   97.0% Program.tallyChars(Reader, Map)
    41.3% ↓ java.io.InputStreamReader.read() → sun.nio.cs.StreamDecoder.read(char[], int, int)
     18.9% sun.nio.cs.StreamDecoder.implRead(char[], int, int)
      14.9% java.nio.charset.CharsetDecoder.decode(ByteBuffer, CharBuffer, boolean)
       11.1% sun.nio.cs.UTF_8$Decoder.decodeLoop(ByteBuffer, CharBuffer)
        10.2% sun.nio.cs.UTF_8$Decoder.decodeArrayLoop(ByteBuffer, CharBuffer)
          2.4% java.nio.CharBuffer.array()
          1.1% ↓ sun.nio.cs.UTF_8$Decoder.xflow(Buffer, int, int, Buffer, int, int) → java.nio.Buffer.position(int)
          < 1% java.nio.ByteBuffer.array()
          < 1% java.nio.Buffer.position()
          < 1% java.nio.ByteBuffer.arrayOffset()
         < 1% java.nio.CharBuffer.hasArray()
         < 1% java.nio.ByteBuffer.hasArray()
        2.5% java.nio.charset.CoderResult.isOverflow()
       3.2% ↓ java.nio.CharBuffer.wrap(char[], int, int) → java.nio.Buffer.limit(int)
       < 1% java.nio.Buffer.hasRemaining()
       < 1% ↓ sun.nio.cs.StreamDecoder.readBytes() → java.io.FileInputStream.read(byte[], int, int)
       < 1% java.nio.Buffer.position()
      < 1% sun.nio.cs.StreamDecoder.ensureOpen()
    16.7% java.util.HashMap.put(Object, Object)
     14.1% ↓ java.util.HashMap.putVal(int, Object, Object, boolean, boolean) → java.util.HashMap$Node.<init>(int, Object, Object, HashMap$Node)
     2.5% java.util.HashMap.hash(Object)
    < 1% java.lang.Long.valueOf(long)
   < 1% ↓ sun.nio.cs.StreamDecoder.read(char[], int, int) → java.nio.CharBuffer.<init>(int, int, int, int, char[], int, MemorySegmentProxy)
2.2% java.lang.Thread.run()
```

Figur 1: Java Fligt Recorder Profile 1

As it can be seen, most of the ressources are spent within the *tallyChars()* method. We decided to start looking at the two following areas of the program:

- The file reading
- The HashMap

2.4 Hypothesis

We suspected that the file reading was slowed down a lot by only using the *FileReader*, since the *FileReader* reads a few bytes at a time and a *BufferedReader* wrapped around a *FileReader* will buffer a whole line and keep it ready. The use of *HashMap* seemed to us to be doing a lot of unnecessary work and we thought that an optimization of that could improve the overall run time some amount.

2.5 Solution(s)

We have done the following, all of which can be seen in the code (except the *BufferedReader* which have been further optimized):

- Implementing the wrapper *BufferedReader*
- Optimization of the *HashMap* and using *ArrayList*
- Further optimization of the file reading with the use of *java.nio.file.Files*

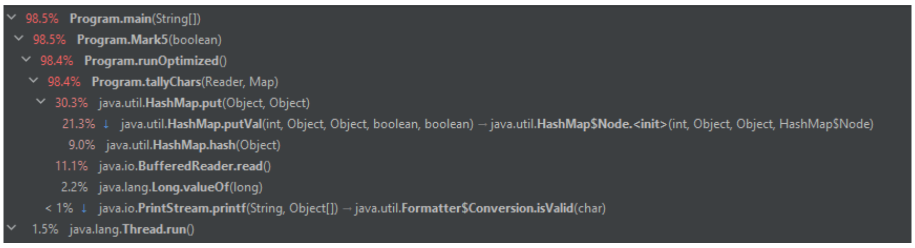
2.6 Optimized Performance

In this section the documentation for each improvement can be seen, along with our thoughts about the changes.

2.6.1 BufferedReader

The use of a BufferedReader alone have increased the program mean runtime by **32.2%**. Looking at the Profile produced by the Java Flight Recorder in IntelliJ after this change, confirms that this was an actual issue with the original code.

```
-------------------------------------------
Mean                 Sdev               Count
-------------------------------------------
39,8 ms +/-    14,74 ms                   2
34,1 ms +/-     1,53 ms                   4
33,7 ms +/-     1,45 ms                   8
33,5 ms +/-     0,85 ms                  16
32,7 ms +/-     0,29 ms                  32
33,1 ms +/-     1,00 ms                  64
33,9 ms +/-     1,39 ms                 128
-------------------------------------------
```



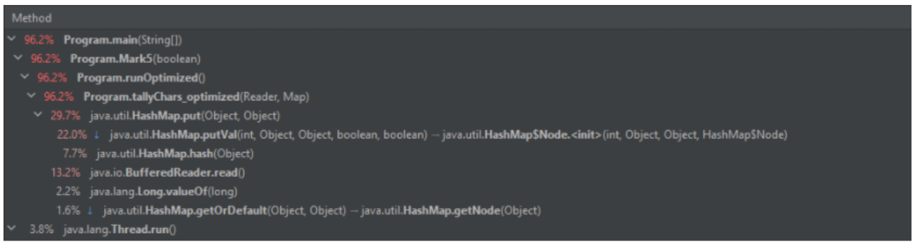
```
98.5% Program.main(String[])
 98.5% Program.Mark5(boolean)
  98.4% Program.runOptimized()
   98.4% Program.tallyChars(Reader, Map)
    30.3% java.util.HashMap.put(Object, Object)
      21.3% ↓ java.util.HashMap.putVal(int, Object, Object, boolean, boolean) → java.util.HashMap$Node.<init>(int, Object, Object, HashMap$Node)
       9.0% java.util.HashMap.hash(Object)
    11.1% java.io.BufferedReader.read()
     2.2% java.lang.Long.valueOf(long)
   < 1% ↓ java.io.PrintStream.printf(String, Object[]) → java.util.Formatter$Conversion.isValid(char)
1.5% java.lang.Thread.run()
```

Figur 2: Java Fligt Recorder Profile 2

2.6.2 HashMap refactoring

After modifying *HashMap* and the methods *tallyChars()* and *print_tally()* we are now at an **40.4%** overall optimization.

```
------------------------------------------
Mean               Sdev               Count
------------------------------------------
36,3 ms +/-    12,22 ms                 2
30,2 ms +/-     1,49 ms                 4
30,9 ms +/-     0,86 ms                 8
31,2 ms +/-     2,04 ms                 16
30,0 ms +/-     1,26 ms                 32
31,0 ms +/-     0,78 ms                 64
29,8 ms +/-     0,62 ms                 128
------------------------------------------
```



```
Method
⌄ 96.2% Program.main(String[])
 ⌄ 96.2% Program.Mark5(boolean)
  ⌄ 96.2% Program.runOptimized()
   ⌄ 96.2% Program.tallyChars_optimized(Reader, Map)
    ⌄ 29.7% java.util.HashMap.put(Object, Object)
       22.0% ↓ java.util.HashMap.putVal(int, Object, Object, boolean, boolean) → java.util.HashMap$Node.<init>(int, Object, Object, HashMap$Node)
        7.7% java.util.HashMap.hash(Object)
      13.2% java.io.BufferedReader.read()
       2.2% java.lang.Long.valueOf(long)
       1.6% ↓ java.util.HashMap.getOrDefault(Object, Object) → java.util.HashMap.getNode(Object)
⌄ 3.8% java.lang.Thread.run()
```

Figur 3: Java Fligt Recorder Profile 3

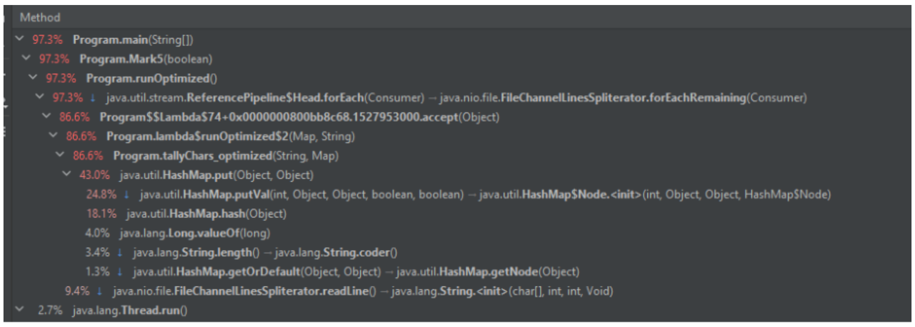
2.6.3 java.nio.file.Files

After changing from *BufferedReader* to this lambda expression from *java.nio.file.Files*

```
Files.lines(Paths.get(fileName)).forEach(line -> tallyChars_optimized(line,
     freq))
```

the run time decreased further. But as of now we are unsure why this is the case, as most people deem *java.nio* and *java.nio* equal. We have also tried to increase the buffer in the *BufferedReader* to be able to contain the whole file, but that did nothing for the run time. As of this change the run time is now **66%** faster than the initial run time of the program.

```
------------------------------------------
Mean               Sdev               Count
------------------------------------------
22,7 ms +/-    10,48 ms                2
18,0 ms +/-     1,76 ms                4
17,6 ms +/-     1,23 ms                8
17,1 ms +/-     0,44 ms                16
17,3 ms +/-     0,61 ms                32
16,8 ms +/-     0,16 ms                64
17,0 ms +/-     0,34 ms                128
------------------------------------------
```

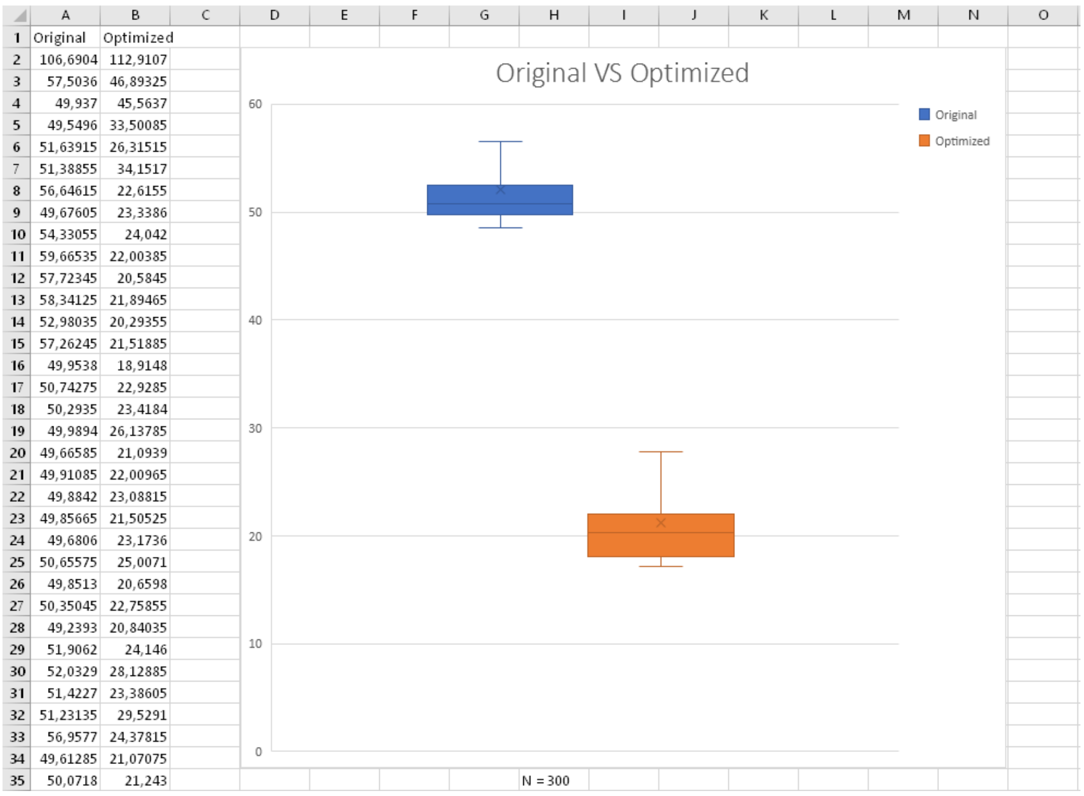


```
Method
⌄ 97.3%  Program.main(String[])
 ⌄ 97.3%  Program.Mark5(boolean)
  ⌄ 97.3%  Program.runOptimized()
   ⌄ 97.3% ↓ java.util.stream.ReferencePipeline$Head.forEach(Consumer) → java.nio.file.FileChannelLinesSpliterator.forEachRemaining(Consumer)
    ⌄ 86.6%  Program$$Lambda$74+0x0000000800bb8c68.1527953000.accept(Object)
     ⌄ 86.6%  Program.lambda$runOptimized$2(Map, String)
      ⌄ 86.6%  Program.tallyChars_optimized(String, Map)
       ⌄ 43.0%  java.util.HashMap.put(Object, Object)
          24.8% ↓ java.util.HashMap.putVal(int, Object, Object, boolean, boolean) → java.util.HashMap$Node.<init>(int, Object, Object, HashMap$Node)
          18.1%  java.util.HashMap.hash(Object)
         4.0%  java.lang.Long.valueOf(long)
         3.4% ↓ java.lang.String.length() → java.lang.String.coder()
         1.3% ↓ java.util.HashMap.getOrDefault(Object, Object) → java.util.HashMap.getNode(Object)
      9.4% ↓ java.nio.file.FileChannelLinesSpliterator.readLine() → java.lang.String.<init>(char[], int, int, Void)
⌄ 2.7%  java.lang.Thread.run()
```

Figur 4: Java Fligt Recorder Profile 4

2.7 Comparison

This section goes into detail of the speed improvements.



	A	B
1	Original	Optimized
2	106,6904	112,9107
3	57,5036	46,89325
4	49,937	45,5637
5	49,5496	33,50085
6	51,63915	26,31515
7	51,38855	34,1517
8	56,64615	22,6155
9	49,67605	23,3386
10	54,33055	24,042
11	59,66535	22,00385
12	57,72345	20,5845
13	58,34125	21,89465
14	52,98035	20,29355
15	57,26245	21,51885
16	49,9538	18,9148
17	50,74275	22,9285
18	50,2935	23,4184
19	49,9894	26,13785
20	49,66585	21,0939
21	49,91085	22,00965
22	49,8842	23,08815
23	49,85665	21,50525
24	49,6806	23,1736
25	50,65575	25,0071
26	49,8513	20,6598
27	50,35045	22,75855
28	49,2393	20,84035
29	51,9062	24,146
30	52,0329	28,12885
31	51,4227	23,38605
32	51,23135	29,5291
33	56,9577	24,37815
34	49,61285	21,07075
35	50,0718	21,243

Figur 5: Boxplot of performance between original and optimized solution

We ran both versions of the program 300 times to better understand the performance increase. The results are visualized above in a box-and-whiskers plot. As you can see, the original program lies at above 50ms per run while the optimized program lies at just above 20ms.

2.8 Conclusion

During the project we tried to identify the bottlenecks causing the slow run time first by reviewing the code and then looking at the profiles from the Java Flight Recorder. We found that the main problems was with the file reading, which also had the greatest impact on performance. We achieved an optimization of the program with a run time **66%** faster than the initial run time of the program.

Litteratur

[1] K. Østerbye, “Letter frequencies.” https://github.com/CPHBusinessSoftUFO/letterfrequencies.

[2] P. Sestoft, “Microbenchmarks in java and c#.” https://datsoftlyngby.github.io/soft2021spring/resources/ee799a67-SestoftMicrobenchmarking.pdf.