cphbusiness
COPENHAGEN BUSINESS ACADEMY

Exploration and Presentation Assignment 3

14. 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** - AMD Ryzen 7 3700X 8-Core 3.60 GHz
- **RAM** - DC 16 GB DDR4 3200 MHz
- **OS** - Windows 10 Pro 64-bit
- **JVM** - Oracle Corporation 15.0.2

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 reads a text file and tally up all the characters in the file and prints them out in the console with the most frequent printed first accompanied by the number of occurrences of each character.

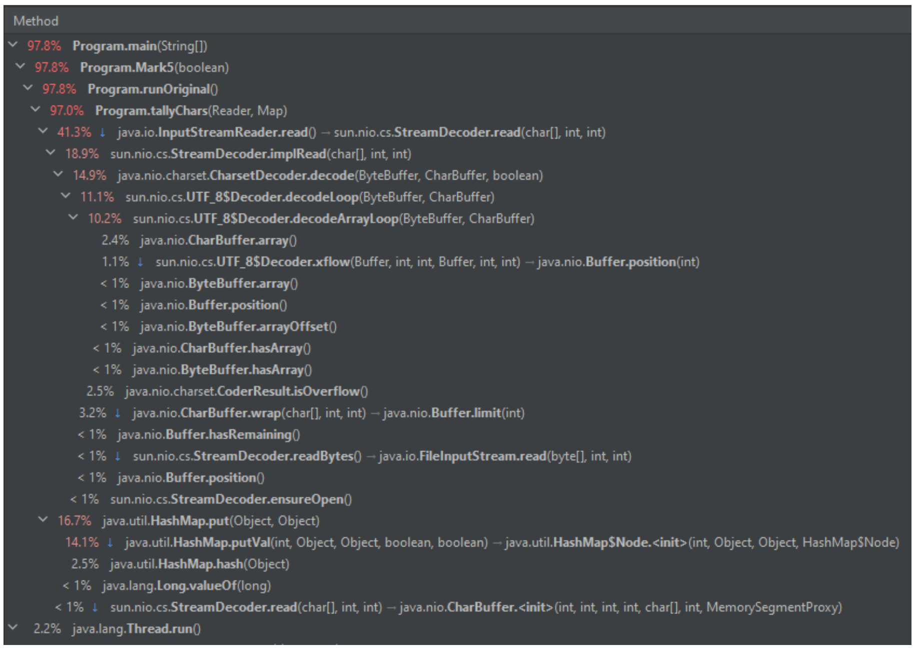
2.2 Current Performance

This is the current console readings when the program is run without any changes to the code. So a baseline run time of 34186746,0 nanoseconds with a standard deviation of 1958071,61 nanoseconds.

```
--------------------------------------------
Mean                Sdev                Count
--------------------------------------------
45161160,0 ns +/- 9633302,98           2
37220422,5 ns +/- 349034,17            4
37097426,3 ns +/- 237128,88            8
37167326,9 ns +/- 257313,56            16
37140620,9 ns +/- 194416,43            32
37137303,8 ns +/- 167730,54            64
37195579,7 ns +/- 120079,00            128
37262370,4 ns +/- 84030,46             256
34186746,0 ns +/- 1958071,61           512
--------------------------------------------
```

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*

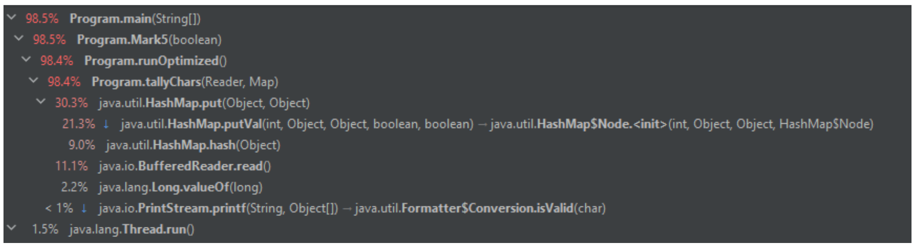
2.6 Optimized Performance

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 **40,87%**. 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
-------------------------------------------
26688810,0 ns +/- 8156130,62           2
21170010,0 ns +/- 1636441,34           4
20378076,3 ns +/- 165036,89            8
20086596,9 ns +/- 70131,53             16
20113604,4 ns +/- 95585,47             32
20121626,1 ns +/- 132419,65            64
20107895,8 ns +/- 106333,91            128
20220584,5 ns +/- 145384,37            256
20213537,6 ns +/- 40180,77             512
-------------------------------------------
```



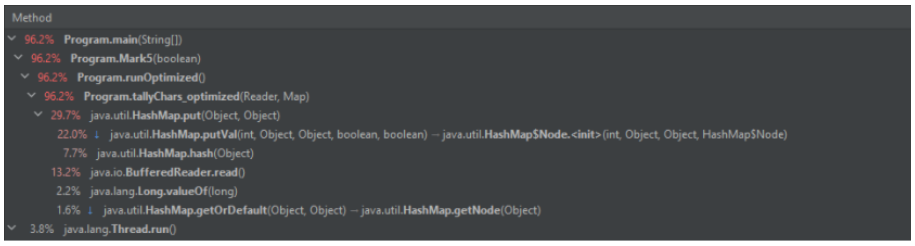
```
98.5%  Program.main(String[])
 98.5%  Program.Mark5(boolean)
  98.4%  Program.runOptimized()
   98.4%  Program.tallyChars(Reader, Map)
    30.3%  java.util.HashMap.put(Object, Object)
      21.3% ↓  java.util.HashMap.putVal(int, Object, Object, boolean, boolean) → java.util.HashMap$Node.<init>(int, Object, Object, HashMap$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 **42,37%** overall optimization.

```
-------------------------------------------
Mean                 Sdev                 Count
-------------------------------------------
24734475,0 ns +/- 6057492,36           2
20550197,5 ns +/- 1135309,37           4
19897790,0 ns +/- 169372,90            8
19688542,5 ns +/- 71463,82             16
20048747,5 ns +/- 322151,07            32
19751591,6 ns +/- 133747,31            64
19704611,6 ns +/- 73574,46             128
19760670,4 ns +/- 108017,90            256
19701593,6 ns +/- 47694,74             512
-------------------------------------------
```



```
Method
v 96.2%  Program.main(String[])
 v 96.2%  Program.Mark5(boolean)
  v 96.2%  Program.runOptimized()
   v 96.2%  Program.tallyChars_optimized(Reader, Map)
    v 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)
v  3.8%  java.lang.Thread.run()
```

Figur 3: Java Fligt Recorder Profile 3

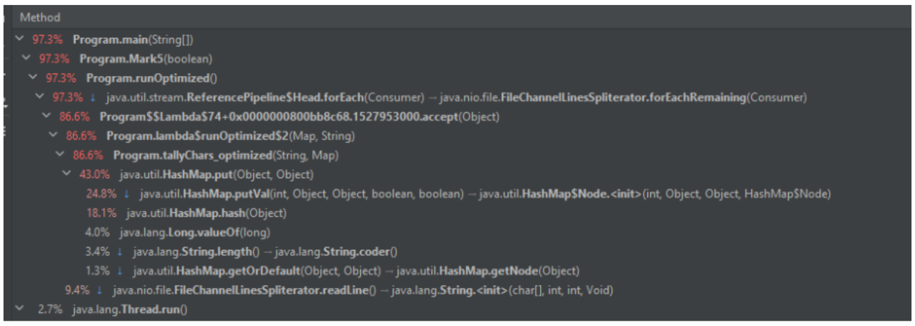
2.6.3 java.nio.file.Files

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 **51,06%** faster than the initial run time of the program.

```
-------------------------------------------
Mean                  Sdev                Count
-------------------------------------------
22515290,0 ns +/- 7673839,43          2
17604875,0 ns +/- 1064155,01          4
16822411,3 ns +/- 209192,50           8
16691758,8 ns +/- 115971,66          16
16571550,6 ns +/- 102956,38          32
16596552,0 ns +/- 92303,35           64
16584048,4 ns +/- 49308,56          128
16599184,4 ns +/- 45268,89          256
16731495,5 ns +/- 57025,00          512
-------------------------------------------
```



```
Method
v 97.3%  Program.main(String[])
 v 97.3%  Program.Mark5(boolean)
  v 97.3%  Program.runOptimized()
   v 97.3% ↓  java.util.stream.ReferencePipeline$Head.forEach(Consumer) → java.nio.file.FileChannelLinesSpliterator.forEachRemaining(Consumer)
    v 86.6%  Program$$Lambda$74+0x0000000800bb8c68.1527953000.accept(Object)
     v 86.6%  Program.lambda$runOptimized$2(Map, String)
      v 86.6%  Program.tallyChars_optimized(String, Map)
       v 43.0%  java.util.HashMap.put(Object, Object)
          24.8% ↓  java.util.HashMap.putVal(int, Object, Object, boolean, boolean) → java.util.HashMap$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)
v  2.7%  java.lang.Thread.run()
```

Figur 4: Java Fligt Recorder Profile 4

Litteratur

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