Optimization of Letter Frequencies - Take Two

Anders Jacobsen, Dima Karaush

April 21, 2021

Contents

1 Introduction

We are using the project Letter Frequencies downloaded from `https://github.com/CPHBusinessSoftUFO/letterfrequencies` in this paper. The entire project with all files can be found here `https://github.com/Cosby1992/CPH-business-assingments/tree/master/Data%20Science/Assignment_3_optimization`.
We plan on measuring a benchmark on the original letter frequencies project. This will enable us to see witch aspects of the program is occupying the most runtime. We plan to use a number of warmup cycles, then run the code again, many times, save the results and in the end write it to a csv file for data exploration and visualization. We will measure all aspects of the original code, and we plan on splitting the code into three categories.

1. The initialization
2. The Letter frequency algoritm
3. The output to console

We expect to see the letter frequency algorithm to be the bottleneck in this code. That is because it is at this point in the code we read and handle the reading of the file. We think that we are able to improve the output to console aswell but not as much as the algoritm. We do not have a prediction for the initialization, we expect it to run too fast to have an infuence on the execution time, but, we are interested in weather our optimized solution will have a difference in the initialization execution time.

2 Test Enviroment

2.1 PC and operating system

OS	Microsoft Windows 10 Pro
OS Version	10.0.19042 N/A Build 19042
System Type	x64-based PC
Processor(s)	Intel® Core™ i7-10700KF Processor, 16M Cache, up to 5.10 GHz
BIOS Version	American Megatrends Inc. 1.10, 21-05-2020
Total Physical Memory	32.688 MB
Disc(s)	Force Series™ MP510 980GB M.2 SSD (up to 3480MB/sec sequencial read)

2.2 Software Enviroment

IDE	Visual Studio Code
IDE version	1.55.2 x64
Language	Java
Language Version	15

3 Tools for Benchmarking

Timer klassen Benchmark timeren Lavet metoder statiske og kalder dem ved benchmarking

4 Benchmark for original

What is going on in the program As it is visible on Listing 1, the original letter frequencies program uses a FileReader and a HashMap<Integer, Long> to read the file and safe the letter frequencies. It manipulates the Hashmap through the static tallyChars method. Then the program uses the other static method, print_tally, to show the letters alligned with their frequency in the file.

```
public static void main(String[] args) throws FileNotFoundException
    , IOException {

    String filePath = "src/main/resources/FoundationSeries.txt";

    Reader reader = new FileReader(filePath);
    Map<Integer, Long> freq = new HashMap<>();

    tallyChars(reader, freq);

    print_tally(freq);
}
```

Listing 1: The main method of the original Letter Frequncies program without optimizations

4.1 The timer

To benchmark this program we used our Timer and BenchmarkTimer classes, descriped in section 3. We implemented our timer in such a way that we get three execution times, Initialization, tallyChars, print_tally.

```
public static double[] original() throws IOException{
    double initializeTime, tallyCharsTime, printTallyTime = 0.0;

    // Get the starting time
    Timer timer = new Timer();
    timer.start();
```

```
7
8       // Create the reader and hashmap
9       // required to run the methods
10      Reader reader = new FileReader("src/main/resources/
        FoundationSeries.txt");
11      Map<Integer, Long> letterFrequencyMap = new HashMap<>();
12
13      initializeTime = timer.milli();
14      timer.start();
15
16      OriginalForBenchmark.tallyChars(reader, letterFrequencyMap);
17
18      tallyCharsTime = timer.milli();
19      timer.start();
20
21      OriginalForBenchmark.print_tally(letterFrequencyMap);
22
23      printTallyTime = timer.milli();
24
25      double[] times = { initializeTime, tallyCharsTime,
        printTallyTime };
26
27      return times;
28  }
```

Listing 2: Timer implementation

As it is visible on Listing 2, we've moved all the parts of the program to a static method witch uses the Timer class to get the execution time after each part of the program with `timer.milli()` witch returns the time since `timer.start()` as a `double` representing milliseconds with multiple decimal points. To get accurate benchmarks we are using a warmup sequence before doing the actual timing.

4.2 Warmup

The warmup session is performed to fight against the JIT[1] compiler in the JVM[2]. That means that the code we run can end up being optimized on the fly (or just in time), meaning that our measurements will not be accurate when we are benchmarking the program. The warmup session is using every class and variable the real timing is using, making sure that the Jit compiler has compiled everything we are using in our benchmark.

```
1   public static void multipleRunsOriginal(int iterations, int
        warmUpIterations, BenchmarkTimer timer) throws IOException {
2
3       double[] temp = new double[3];
4
5       // Warm-up
6       for (int i = 0; i < warmUpIterations; i++) {
7           temp = original();
8
```

[1] Just-In-Time Compiler
[2] Java Virtual Machine

```
9           for (int j = 0; j < 3; j++) {
10              timer.addWarmupTime(j, temp[j]);
11          }
12      }
13
14      // Real test
15      for (int i = 0; i < iterations; i++) {
16          temp = original();
17
18          for (int j = 0; j < 3; j++) {
19              timer.addRealTime(j, temp[j]);
20          }
21      }
22  }
```

Listing 3: Warmup implementaion

As seen on Listing 3, we run a number of warmup iterations identical to the real timing before we are timing the real iterations. Since all of our timing is returned from the `Original()` method, it doesn't really matter how much time we use on saving the times afterwards. The important thing is that the code is completely compiled before taking the real times.

4.3 Saving the results

From Listing 3 it is also visible that we save our times in the `BenchmarkTimer` class, descriped in section 3. This class can store multiple list's of values in an array. We use this class to save the times until we can write them to a csv file for further analysis.

4.4 Benchmark of the original code

The actual benchmarking is initialised in the `main` method of the `Benchmark` class and can be seen on Listing 4 below.

```
1   public static void main(String[] args) throws IOException {
2
3       BenchmarkTimer timer = new BenchmarkTimer(3);
4
5       multipleRunsOriginal(500, 30, timer);
6
7       timer.writeRealTimesToCSV("time_data/
        multiple_run_real_times_new.csv");
8
9       printTallyTimes(timer);
10  }
```

Listing 4: Benchmark main method

Listing 4 shows that we run the benchmark 500 times and have 30 warmup iterations. This can be seen in line 5 where the method `multipleRunsOriginal (500, 30, timer);` initializes the benchmark. We also provide the method with a `BenchmarkTimer` that is used to keep track of the times measured through the benchmark. Afterwards we write all the measured times to a CSV file with the

name `multiple_run_real_times.csv` witch will be located in the `time_data` folder in the root of the project.

4.5 Results

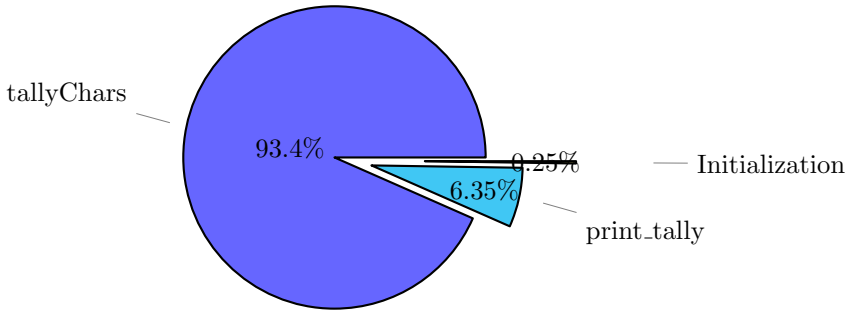
A python notebook is used to explore the run-times we got from our benchmark. The first thing we did was finding key statistical numbers like mean, std. dev. min, max. etc. to get a overview of our data.

	Initialization	**tallyChars**	**print_tally**
count	500	500	500
mean	0.105415	39.932451	2.716147
std	0.463613	1.277004	0.383040
min	0.056300	39.005500	2.047200
25%	0.070675	39.220900	2.582875
50%	0.084550	39.452850	2.694450
75%	0.093400	40.195350	2.844200
max	10.443300	51.593000	9.720600

Mean and standard deviation We can from this see that the average execution time and standard deviation of the tally algorithm is 39.932451 +- 1.277004 ms.

Min, max and quatiles It's clearly visible that the min, 25%, 50%, 75% lies very closely to the mean. And the Max value falls far from these times. A reason for that can be that that the PC is working on a lot of other stuff blocking the execution of the code. We eliminate these outliers to get a more accurate comparisson later on.

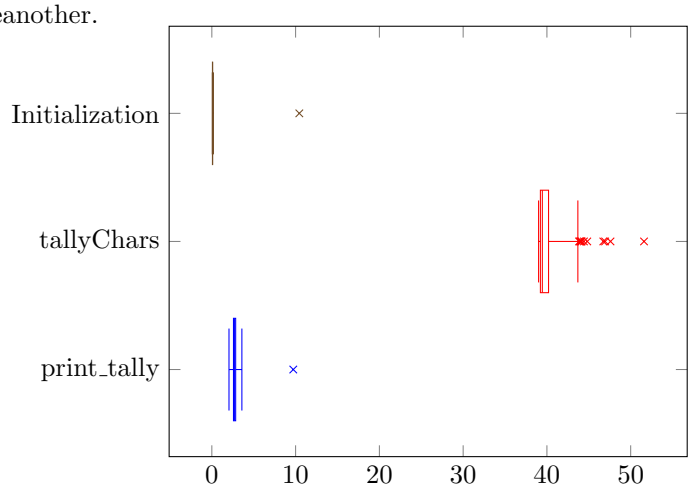
The next thing we can take a look at is how many percent each of the times is responsible for in the total runtime. To show that, we have made a pie-chart showing the percent of the total execution time, the part of the program is responsible for.



Percent of execution time As seen on the pie chart, the tallyChars method is responsible for 93,4% of the total runtime, this is a clear indication of where we need to focus our optimization. The output to the console only takes up 6,35% of the execution time. This means that optimization in this part of the program will have less infuence of the total runtime. However, the lowest amount of time is used in initialization, that only takes up 0,25% of the total execution time.

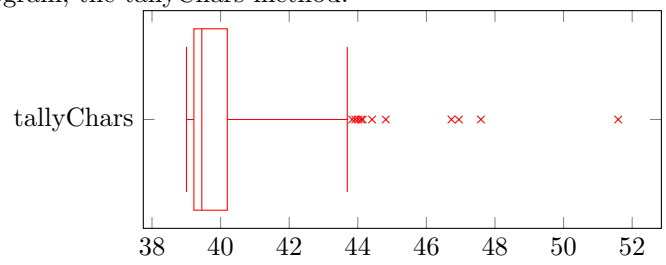
Boxplot of all times

We have collected all the times in a boxplot to see how they lie compared to oneanother.



Tally Chars boxplot

Here we have zoomed in on a boxplot of the most time consuming part of the program, the tallyChars method.



Optimization After exploring the data collected from the original benchmark we are now ready to optimize the code. From our data and data visualization we now know that we need to focus our optimization on the tallyChars method. In Section 5 we show what steps we took to optimize the program.

5 Optimization

What changes do we make to the program Show snippets of changes

6 Benchmark for Optimized

show the results of the benchmark

7 Benchmark Comparison

Compare the original vs the optimized times show tons of graphs and charts

8 Conclusion

Sum it up talk about most remarkable changes.