BenchMarking Algorithm Lab

CSUN Comp 182 Ariel Kohanim

December 19, 2018

1 Introduction

For this assignment I integrated the function:

$$\int_1^{100} 1/x dx$$

There are many methods to approximate an integral, such as by using Simpson's method or Trapezium. The following program integrates the formula above using the rectangular midpoint. This was done in 3 different methods: 1. for-loop 2. sequential stream 3. parallel stream

2 Time Complexity

The time complexity of the three methods approximating the same integral are all: $O(n)$.

This is because we are running a for loop n times, these functions have a time complexity of an upper bound of $O(n)$ and lower bound $\Omega(n)$ and therefore have $\Theta(n)$ since in the input for the number of times the for loop runs is 'n' for all 3 methods.Therefore all functions grow linearly with respect to 'n'.

Compilers have 40+ years of experience optimizing loops and the virtual machine's JIT compiler is especially apt to optimize for-loops over arrays with an equal stride like the one in our benchmark. Streams on the other hand are a very recent addition to Java and the JIT compiler does not (yet) perform any particularly sophisticated optimizations to them.

The ultimate conclusion to deduce from this benchmark experiment is that sequential streams are not always slower than loops. Yes, streams are sometimes slower than loops, but they can also be equally fast; it depends on the circumstances.

3 Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs. That means how much memory, in the worst case, is needed at any point

in the algorithm. The for-loop is more efficient than streams with respect to space complexity for this algorithm. The for loop has a space complexity of $\Theta(1)$ because the loop runs 'n' times, its is a fixed constant variable. Streams have a higher space complexity relative to the for loop. The time complexity for streams is $\Theta(n)$

4 Code

```
1             private static double rectangular(double a, double b, int n, FPFunction f, int m
2   {
3     double range = checkParamsGetRange(a, b, n);
4     double modeOffset = (double)mode / 2.0;
5     double nFloat = (double)n;
6     double sum = 0.0;
7     for (int i = 0; i < n; i++)
8     {
9       double x = a + range * ((double)i + modeOffset) / nFloat;
10      sum += f.eval(x);
11    }
12    return sum * range / nFloat;
13  }
14
15  //streams
16  public static double rectangularStream(double a, double b, int n, FPFunction f, int mode)
17  {
18    double range = checkParamsGetRange(a, b, n);
19    double modeOffset = (double)mode / 2.0;
20    double nFloat = (double)n;
21
22
23    double sum = IntStream.range (0,n)
24            .mapToDouble(i -> a + range * ((double)i + modeOffset) / nFloat)
25            .map(x -> f.eval(x))
26            .reduce(0, (y, z) -> y+z);
27        return sum * range / nFloat;
28
29  }
30
31   public static double rectangularParallelStream(double a, double b, int n, FPFunction f, i
32  {
33    double range = checkParamsGetRange(a, b, n);
34    double modeOffset = (double)mode / 2.0;
35    double nFloat = (double)n;
36
```

```
37
38      double sum = IntStream.range (0,n)
39              .parallel()
40              .mapToDouble(i -> a + range * ((double)i + modeOffset) / nFloat)
41              .map(x -> f.eval(x))
42              .reduce(0, (y, z) -> y+z);
43          return sum * range / nFloat;
44
45  }
```

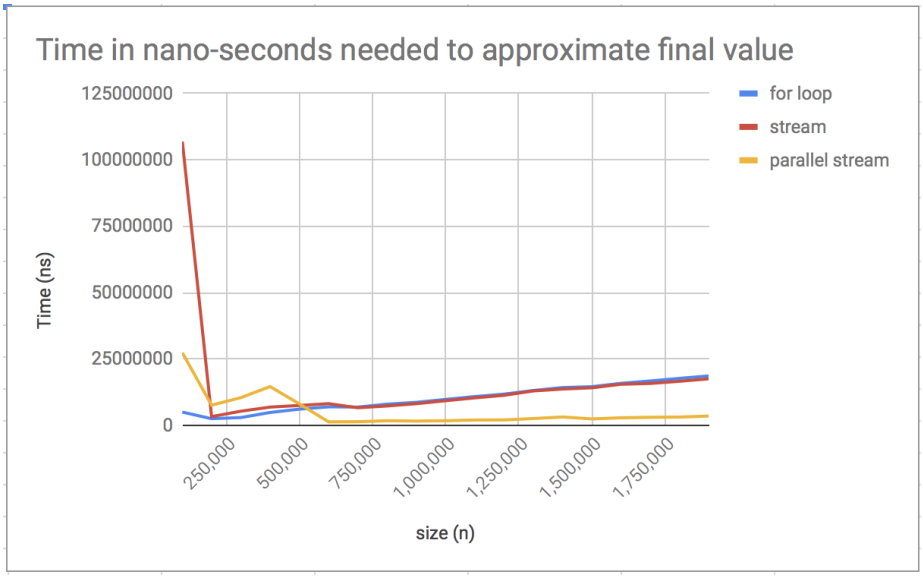

Figure 1: Graph of data