BenchMarking Algorithm Lab

CSUN Comp 182 Ariel Kohanim

December 3, 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'.

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 the most efficient with respect to space complexity for this algorithm. The for loop has a space complexity of $\Theta(1)$ it is constant because n is a constant variable and the for loop stays on the stack-frame, jumping back and forth via references. Streams on the other hand utilize arrays, hence have a higher space complexity relative to the for loop. The time complexity is $\Theta(n)$ since streams needs to create and allocate space for an array for each iteration of n.

4 Code

```
          private static double rectangular(double a, double b, int n, FPFunction f, int m
{
  double range = checkParamsGetRange(a, b, n);
  double modeOffset = (double)mode / 2.0;
  double nFloat = (double)n;
  double sum = 0.0;
  for (int i = 0; i < n; i++)
  {
    double x = a + range * ((double)i + modeOffset) / nFloat;
    sum += f.eval(x);
  }
  return sum * range / nFloat;
}

//streams
public static double rectangularStream(double a, double b, int n, FPFunction f, int mode)
{
  double range = checkParamsGetRange(a, b, n);
  double modeOffset = (double)mode / 2.0;
  double nFloat = (double)n;


  double sum = IntStream.range (0,n)
          .mapToDouble(i -> a + range * ((double)i + modeOffset) / nFloat)
          .map(x -> f.eval(x))
          .reduce(0, (y, z) -> y+z);
      return sum * range / nFloat;

}

 public static double rectangularParallelStream(double a, double b, int n, FPFunction f, i
{
  double range = checkParamsGetRange(a, b, n);
  double modeOffset = (double)mode / 2.0;
  double nFloat = (double)n;


  double sum = IntStream.range (0,n)
          .parallel()
          .mapToDouble(i -> a + range * ((double)i + modeOffset) / nFloat)
          .map(x -> f.eval(x))
          .reduce(0, (y, z) -> y+z);
      return sum * range / nFloat;
```

```

}
```

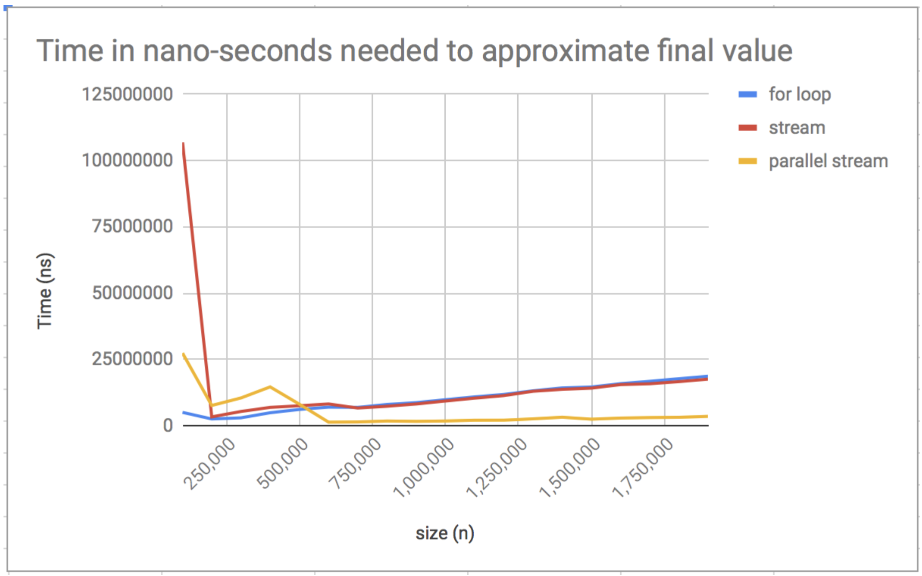

Figure 1: Graph of data