ASSIGNMENT 8 – PARALLELISM

Advanced programming paradigms

Question 1 – *Parallel collections*

In this first exercise, you will experiment on how parallel collections can be easily used to improve the performance on standard operations in FP[1].

Sometimes, a program needs to integrate a function (i.e., calculate the area under a curve). It might be able to use a formula for the integral, but doing so is not always convenient, or even possible. An easy alternative approach is to approximate the curve with straight line segments and calculate an estimate of the area from them. This is called the *trapezoidal rule*, which is a technique used to approximate the value of a definite integral $\int_a^b f(x)dx$.

To compute the value of the integral numerically, we can apply the following algorithm (by supposing $f(x)$ is continuous on $[a, b]$).

1. Start by dividing $[a, b]$ into n sub intervals of equal length $\Delta = \frac{b-a}{n}$.
2. Compute $f(x)$ at each point to obtain $y_0 = f(x_0), y_1 = f(x_1), \ldots, y_n = f(x_n)$.
3. As straight lines are formed between the points (x_{i-1}, y_{i-1}) and (x_i, y_i) for $1 \leq i \leq n$, we can approximate the integral using n trapezoids

The following figure illustrates how the trapezoidal method can be applied to a general function:

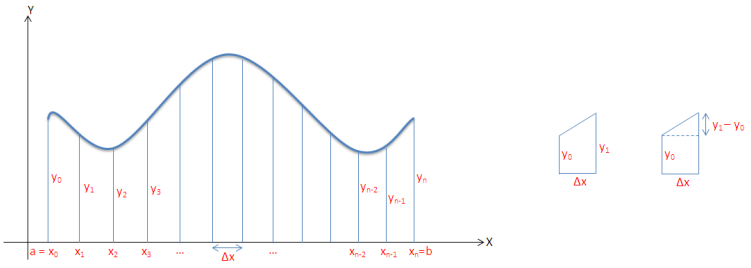

Figure 1 – The trapezoidal method in general and trapezoid area computation

To compute one of the trapezoids area, we can use the following formula:

$$A = y_0\Delta x + \frac{1}{2}(y_1 - y_0)\Delta x$$
$$A = \frac{(y_0 + y_1)\Delta x}{2}$$

In the end, adding the area of n trapezoids is an approximation of the integral

$$\frac{(y_0 + y_1)\Delta x}{2} + \frac{(y_1 + y_2)\Delta x}{2} + \ldots + \frac{(y_{n-1} + y_n)\Delta x}{2} \approx \int_a^b f(x)dx \tag{1}$$

If we have very many divisions to compute the integral (to have a precise values for instance), it might be interesting to compute the y_i values in parallel. However, even though this equation enables parallel computation, it requires to compute most values twice (y_1 for instance). As it would be a bit silly to try to accelerate an inefficient computation, we can rewrite (1) as follows:

$$\approx \frac{\Delta}{2}(y_0 + 2y_1 + \ldots + 2y_{n-1} + y_n) \tag{2}$$
$$\approx \frac{\Delta}{2}(y_0 + y_n + 2\sum_{i=1}^{n-1} y_i) \tag{3}$$

[1] This exercise is adapted from the *SIGCSE 2013 Workshop on Scala*.

This latest solution requires now to compute each y_i once. As a result, the sum can be easily executed in parallel, a feature that we will leverage in the exercise.

(a) Define a method called `integrate` to compute the integral of a general function (which can be defined as a parameter). This function should use the trapezoidal integration technique described earlier. The complete prototype of the function should be as follows:

```
1 def integrate(a: Double, b: Double, nIntervals: Int, f: (Double => Double))
```

Check that your function returns the proper result. For instance, $\int_1^2 sin(x)\ dx \approx 0.956449$

(b) The function you integrated above was composed of a single mathematical function $sin(x)$. Scala offers a way to compose (in a mathematical sense) functions. Using the `compose` operator, compute $\int_0^1 sin(cos(x))\ dx$. The result should be ≈ 0.738643.

(c) Measure the time[2] required for computing a result using 20^6 intervals.

(d) The performance of the method you have developed so far is not optimal. In fact, applying various operations in sequence to a list is not really efficient because all the intermediary result have to be computed and stored. Take for instance the following case:

```
1 (1 to 10e6.toInt).
2   map(Math.sin(_)).
3   map(_ * 12).
4   map(Math.pow(_, 4))
```

This requires to create 3 different collections and apply the operation on each intermediary result. As we did with the `Stream` class, it is possible to create *lazy collections* with the `view` method. Thus, writing:

```
1 (1 to 10e6.toInt).
2   view.
3   map(Math.sin(_)).
4   map(_ * 12).
5   map(Math.pow(_, 4))
```

consumes far less memory and goes almost two times faster. In addition, if all the results are not required, the results are not computed. Thus, in the second case, if we only require the first element of the result, only a single value will be computed!

Apply this technique to the integration method and measure the time required to compute a result. Why is it faster?

.....

.....

.....

(e) Implement now a new version of the integration using *lazy parallel collections*. Please note that the change on the collections themselves should be a matter of adding a `.par` somewhere. Measure the time taken to compute the numerical integral of the function using both the parallel and sequential collections. What speed-up do you get on your machine in the parallel version compared to the view version? Does it correspond to the speed-up you were expecting considering the number of logical processors your machine possesses?

[2] To measure time you can download the `utils` package on the Moodle website

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Question 2 – *Using futures in a real-world example*

A typical use case for futures is to asynchronously read results from the Web. In this exercise, you will use futures to read values from two public REST APIs to know the exchange rate for Bitcoins in CHF.

(a) Write a method returning a `Future[String]` to read the content of a web page. Note that you can read data from a URL simply by using the `Source.fromURL(...).mkString` method. To make it work, import `scala.io.Source`.

(b) Access the data from the following URL `https://api.bitcoinaverage.com/ticker/USD/` which provides the exchange rate from Bitcoins to USD. The returned `String` is formatted in JSON.

(c) Access the data from the URL `https://api.bitcoinaverage.com/ticker/CHF/` which gives the exchange rate from Bitcoins to CHF. This is also formatted as JSON.

(d) To parse JSON in Scala[3], download the two JAR files from the Moodle website (`lift-json_XXX.jar` and `paranamer-2.1.jar`). You need then to include those JAR into your Eclipse workspace, *Right-click on your project → Build Path → Configure build path → Add JAR ...*). When you are done, the following program should execute without problem:

```
1  import net.liftweb.json.DefaultFormats
2  import net.liftweb.json.parse
3
4  object JSONSimple extends App {
5    implicit val formats = DefaultFormats
6
7    // Sample JSON data
8    def bitcoin_json() = {
9      """{
10       "ask": 458.05,
11       "bid": 456.59,
12       "last": 456.41,
13       "timestamp": "Mon, 14 Apr 2014 12:09:35 -0000",
14       "total_vol": 93280.72
15       }"""
16   }
17   val jVal = parse(bitcoin_json) // JSON to val
18
19   // Primitive extraction a la XPATH
20   val a = (jVal \ "bid").extract[Double]
21   println("Extraction of 'bid': " + a)
22 }
```

(e) Using futures composition with comprehensions, parse the results from the two URLs and display the USD vs CHF rate.

[3]There many libraries to parse JSON in Scala. The one you will be using here comes from the *Lift!* framework which is typically used for creating web services. In this exercise, you will be using only the JSON library part of that framework. The library allows for several types of JSON parsing, using extraction for case classes, XPATH style extraction .... It also enables the creation of JSON data from Scala classes. The example above is very simplistic.