# #3 Assignment - Scientific Computations - CMPT 383

Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

## I. Introduction

Regarding scientific programming, what programming languages should we use? What software libraries should we build upon? Our answers to such questions can be defined on the languages and/or paradigms we use.

Numerical methods for scientific problems, especially in engineering and science, are frequently related to solving problems for large matrices (such as image processing). Many of the most efficient algorithms for large-scale matrix computations are based on approximations of the given matrix by small matrices (e.g. kernel convolution and Petrov-Galerkin projections).

Traversing matrices is one of the heaviest numerical computations as, for a given matrix **m** with *r* rows and *c* columns, it has known time complexity of $O(r \times c)$. Imagine that we have a bitmap image **img** of size $800 \times 600$ with *RGB* colors, for example. Now imagine that we want to convert **img** into its grayscale representation. In that case, it would take a time of $800 \times 600 \times 3$ (we multiply by 3 because we must consider the space occupied by $red$, $green$ and $blue$ values of each pixel) to traverse and convert every RGB pixel of **img** into its grayscale representation.

The work below will try to prove that, for all (or most) different paradigms here compared *(Functional Programming (Haskell), Object-oriented Programming (Java), Scripting Language (Python) and Imperative Programming (Fortran))*, it should take about the same time complexity to traverse and apply map functions to matrices, but the same might not be true for memory allocation (some paradigms might take more memory).

## II. Applying numerical computations for different paradigms and programming languages

### Functional Programming - Haskell

Functional programming in general appears to have many beautiful ways of traversing matrices, given (in theory) the lack of side effects and its foundation of applying mathematical functions to the real world, however, in practice, this may not work as beautiful as it seems because a bigger problem may be divided into many small problems (function calls, as high order functions are always broken down in other smaller functions), which may impact on the speed of a given program, once that it would be faster if the process was done linearly[5] (the way that imperative languages would try to solve this category of problems). Haskell will be the representative of the Functional Programming paradigm for this assignment.

Haskell has many libraries for matrix handling, however, I chose to use the *hmatrix* library as it is comparable with *Python's numpy* library and it also provides helpers for common matrix operations.

On the code below, you will see how one would convert a bitmap image into its grayscale representation in haskell.

```
import Codec.BMP
import System.Environment
import System.IO
import Numeric.LinearAlgebra

main
 = do    [srcFile, dstFile] <- getArgs

  handle <- openFile srcFile ReadMode
  mBMP <- hGetBMP handle

  case mBMP of
   Left  err -> print err
   Right bmp ->
     do let rgb = unpackBMPToRGBA32 bmp
        let (width, height) =
          bmpDimensions bmp

        let m'    =
          toGrayScale width height rgb
        let svBmp' =
          packRGBA32ToBMP width height m'
        handle' <-
          openFile dstFile WriteMode
        hPutBMP handle' svBmp'
```

The code above will open a bitmap image, get its height and width and then will traverse its pixel matrix and convert every single pixel into its grayscale (using the function *toGrayScale*).

We realize that the program above will take most of its time to read, process and write the output of the given *bmp* image. Also, *toGrayScale* will run in linear time $O(|rows| \times |colums|)$. In order to be full functional, it would break every cell of the matrix down and apply a recursive function rebuilding and copying the matrix by passing it as a parameter to "keep" the state of the *still-to-build* matrix. This

will be evaluated later (on Assignment #5).

**Object-oriented Programming - Java**

The way Object-oriented programming paradigm behaves regarding scientific programming can be related to the Object-oriented language we are using, because such paradigm is often implemented in multi-paradigm languages. For example, *Java, Erlang, Objective-C and Pony* will most likely have different ways and use different methods for processing numerical computations. That happens because these languages are not only Object-oriented, but also imperative (for Java and Objective-C) and Process-Calculus-based(Erlang and Pony), a paradigm very close to Functional Programming, where "everything" is a process rather than a function. That is to say, Java will be representative language for this category, but what is true to Java might **not** be true to other different *OOP languages*.

Java is one of the *C-family* languages[11] and for this reason, scientific programming in pure Java can be **very** similar to scientific programming in C (not accounting direct access to pointers). In fact, it would be very hard to identify and distinguish scientific programs in pure Java or/and in pure C (not considering classes and particularities of the Object-oriented paradigm).

Java has the benefit of having multiple frameworks for numerical computations, such as *COLT, JLAPACK, ND4J, Matrix toolkit Java* and others. However, I chose to write numerical computations in pure Java for this assignment, as I would like to compare it with C (because the code is very similar, but there might be differences in runtime because of the different compilers for these languages and because Java's bytecode will be interpreted by the JVM).
The code below converts a bitmap image to its grayscale representation:

```
public class ProcessBMP {
 public static void main(String args[]) {
   BufferedImage img = ImageIO.read(
       getClass().getResource(bmp));

   int w = image.getWidth();
   int h = image.getHeight();
   int[][] matrix = new int[w][h];

  // r = |rows|, c = |columns|
  // w = width, h = height
  for (int r = 0; r < w; r++) {
    for (int c = 0; c < h; c++) {
      int color = image.getRGB(r, c);
      int gray  = image.toGrayScale(color);
      matrix[r][c] = gray;
    }
  }
 }
}
```

The code above will traverse the image as fast as C would do and in fact in very hard to distinguish the differences between that code and a given C code for the same task.

**Scripting Language - Python** As said before on previous assignments, scripting languages usually run slower than compiled languages (because of the time that is spent on just-in-time translations). Python has a slight advantage on other scripting languages because it uses many libraries written in pure C, boosting the process of interpreting programs.

Python is multi-paradigm and allows **imperative programming** as well, just like *Fortran, Java and C* and is reasonable to think that it will be one of fastest languages for this assignment. One point is that it is well known as a scientific programming language with numerous libraries out there, such as *NumPy* (which is mostly written in C). Lastly, Python allows "hacks" for performance when dealing with matrix handling. For this assignment, it was chosen to use *NumPy* for scientific programming.

The code below takes bitmap image as input and outputs its grayscale representation:

```
import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[...,:3],
      [0.299, 0.587, 0.114])

img = mpimg.imread(filename)
grayimg = rgb2gray(img)
mpimg.imsave(filename, grayimg)
```

Compared to all other seen snippets, so far, Python is the one with clearest code and helps us understand why the scientific community enforce the use of this tool. We still need to check how fast it is indeed (in comparison with other programming paradigms and languages, such as *Fortran and Java*).

**Imperative - Fortran**

Numerical computations are strongly encouraged by the *Imperative programming* paradigm. *Fortran, C and Assembly itself* are very powerful tools that help us to get close to the "bare" metal. The mentioned languages are very flexible and take full advantage of the Turing tape. On the other hand, they are subject of constant problems such as bugs, once they have many side effects. Depending on the program, we will choose speed over code safety (which would have on Haskell or other Functional Programming languages, for example). For this assignment, we will compare speed and therefore, this category is highly indicated to be one of the fastest ones. The chosen language for this assignment will be Fortran because of it was designed to work with scientific computations.

In spite of being a very old language (appeared for the

first time on 1957), Fortran is somewhat currently one of the most important language for scientific and engineering computation and actually, there are some libraries to work with newer languages, such as the transpiler *FORTRAN-to-Java (f2j)* that enables automatic translation of LAPACK to Java and many tools that do the same for C as well.

On the code below, we will see an example written in Fortran that will take an image as input and return its grayscale representation as output:

```
program grayscale
  implicit none
  integer ihpixf, jvpixf
  ! pixel and data size
  parameter(ihpixf = 128, jvpixf = 128)
  ! RGB image array (integer)
  character*1 rgb(3,ihpixf,jvpixf)
  ! three, 2D data array (float or double)
  real*8  dat(3,ihpixf,jvpixf)
  integer nframe, nf2

  do nframe = 1, 50
    nf2 = nframe
    call mkdata(dat,nf2)
    call mkbitmap(dat,rgb)
    call grayscale(rgb,nf2)
  enddo

  stop
end program grayscale
```

*Obs.: the code above was not written by me, you can see the original code on [18]*

## III. Evaluation

In order to evaluate all the languages and their capabilities related to scientific programming, it will be tested:

- Behaviour of program when copying a very large matrix (size $100000000 \times 100000000$).
- Behaviour of program when processing images (RGB image to grayscale representation).
- How long does a program take to apply **gaussian filter** on convolutions of size $25 \times 25$ on a image of size $1024 \times 1024$.

Tools for **measuring** that (*code profilers* and others):

- **Haskell**: *ghc* with option *-prof*.
- **Java**: *jprofiler* tool.
- **Python**: *profile.py* library.
- **Fortran**: *gprof* tool.

## IV. Evaluation Environment

Macbook Pro:
- 2.2GHz quad-core Intel Core i7 processor
- 16GB RAM

## V. Discussion

I believe that, after evaluating all languages, **Fortran**, **Python** and **Java** will have better performance than **Haskell** for this assignment. That will happen because these three languages base themselves on the imperative paradigm and are able to perform fast scientific computations, whereas *Haskell* will be constrained by the functional programming paradigm.

I also think that *Python* will be one of the fastest ones (together with Fortran) as it has libraries developed in *C*, making it as fast as *C*.

Going further, without proper **benchmarks**, it will be hard to tell if *Fortran* will execute faster and better than *Java* or *Python*.

That being said, in spite of being very reliable, *Haskell* will most likely execute slower than the other three chosen languages. As said before, it will try to break every single function in smaller functions (following Lambda Calculus principles), proving that Functional Programming Languages run slower than Imperative ones.

Regarding *Java*, the *JVM* might help, depending on the type of computations (with JVM warm up), however, it is hard to predict as the JVM itself is another layer of abstraction.

To conclude, I believe that the rank of languages will end with:

- **1st place: Python**
*Python* is chosen to be the best one (in comparison with the other ones in this assignment) because it has many libraries written in C, having advantage over other languages.

- **2nd place: Fortran**
As said before, *Fortran* is designed for scientific programming. It is also one of the oldest and most respect languages amongst this set of languages. It also has very fast libraries. It is very hard to say whether it will be faster or not than Python. It will require a very good evaluation on assignment #5.

- **3rd place: Java**
The *JVM* is slow, but not so slow. There still some advantage because of its *JVM's* warm-up. Some people say that scientific programming in Java is actually possible and there are a lot of support for it on its community).

- **4th place: Haskell**
I also believe strongly that *Haskell* will take the last position because of the reasons discussed before on this assignment.

I am looking forward to doing their benchmarks!

## REFERENCES


[1] "IWASEP", 2017. [Online]. Available: http://www4.ncsu.edu/ ipsen/ps/slides_iwasep.pdf. [Accessed: 28- Oct- 2017].

[2] "Freund_HLA", 2017. [Online]. Available: https://www.math.ucdavis.edu/ freund/freund_HLA.pdf. [Accessed: 28- Oct- 2017].

[3] "Big O notation", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Big_O_notation. [Accessed: 28- Oct- 2017].

[4] "Tweag I/O - Enter the matrix, Haskell style", Tweag.io, 2017. [Online]. Available: https://www.tweag.io/posts/2017-08-31-hmatrix.html. [Accessed: 28- Oct- 2017].

[5] "Functional vs. Imperative Programming", Ryanhmckenna.com, 2017. [Online]. Available: http://www.ryanhmckenna.com/2014/11/functional-vs-imperative-programming.html. [Accessed: 28- Oct- 2017].

[6] "hmatrix", 2017. [Online]. Available: http://dis.um.es/ alberto/material/hmatrix.pdf. [Accessed: 28- Oct- 2017].

[7] "For scientific computing, is Java useful in a way that C or Python aren't? - Quora", Quora.com, 2017. [Online]. Available: https://www.quora.com/For-scientific-computing-is-Java-useful-in-a-way-that-C-or-Python-arent. [Accessed: 28- Oct- 2017].

[8] "Scientific Computation", Introcs.cs.princeton.edu, 2017. [Online]. Available: https://introcs.cs.princeton.edu/java/90scientific/. [Accessed: 28- Oct- 2017].

[9] "Java And Cpp Platforms For Scientific Computing", 2017. [Online]. Available: https://inside.mines.edu/ dhale/papers/Hale06JavaAndCppPlatformsForScientificComputing.pdf. [Accessed: 28- Oct- 2017].

[10] P. Knoll and S. Mirzaei, "Scientific computing with Java", 2017.

[11] "List of C-family programming languages", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/List_of_C-family_programming_languages. [Accessed: 28- Oct- 2017].

[12] "Scientific Computing Tools for Python SciPy.org", Scipy.org, 2017. [Online]. Available: https://www.scipy.org/about.html. [Accessed: 28- Oct- 2017].

[13] "1.1. Python scientific computing ecosystem Scipy lecture notes", Scipy-lectures.org, 2017. [Online]. Available: http://www.scipy-lectures.org/intro/intro.html. [Accessed: 28- Oct- 2017].

[14] "A Primer on Scientific Programming with Python", 2017. [Online]. Available: https://hplgit.github.io/primer.html/doc/pub/half/book.pdf. [Accessed: 28- Oct- 2017].

[15] "Fortran", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Fortran. [Accessed: 28- Oct- 2017].

[16] "Programming in FORTRAN", Chem.ox.ac.uk, 2017. [Online]. Available: http://www.chem.ox.ac.uk/fortran/fortran1.html. [Accessed: 28- Oct- 2017].

[17] "Scientific computings future: Can any coding language top a 1950s behemoth?", Ars Technica, 2017. [Online]. Available: https://arstechnica.com/science/2014/05/scientific-computings-future-can-any-coding-language-top-a-1950s-behemoth/. [Accessed: 28- Oct- 2017].

[18] Sun.stanford.edu, 2017. [Online]. Available: http://sun.stanford.edu/ keiji/pixelfrt.for. [Accessed: 28- Oct- 2017].