#3 Assignment - Scientific Computations - CMPT 383

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

I. Numerical Computation

Numerical Computational 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 800×600 with *RGB* colors, for example. Now imagine that we want to convert **img** into its grayscale representation. It 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 $blue$) 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, Object-oriented Programming, Scripting Language and Imperative Programming)*, it should take about the same time complexity to traverse matrices, but the same might not be true for the memory allocation (some paradigms might take more memory).

II. Paradigms and Programming Languages

Actor/Message Passing/OOP - Erlang

Erlang is designed for concurrency and is used in many big projects whose main concerns are related to performing asynchronous tasks, such as Facebook Messenger. Erlang is a pure functional language and features single assignment and eager evaluation. In its actor model, each object is an actor. Messages can be exchanged among actors, which will be buffered in the object's mailbox. Upon receiving a message, an action will be taken by the actor. Messages are not guaranteed to be delivered in the same order as they were sent, however they are guaranteed to be always delivered.

Concurrent programming in **Erlang** is built-in is done by sending asynchronous messages with and identifier so called *Pid*(process identifier). The caller is able to do that by executing a command responsible by spawning a new *Pid*: $pid = spawn(f)$, where f is a function. See the example below:

```
start() ->
    spawn(fun() -> sendMessage("Hi") end).

sendMessage(Message) ->
    io:fwrite("~p",[Message]).
```

The code above will call $spaw$ sending a new concurrent process that evaluates fun. The new process runs in parallel with the caller and prints the message *"Hi!"*.

Similarly, to receive a message passed by an *spawned* function f, one will use the keyword $receive$ and pattern match the input. See below:

```
addFn() ->
    receive
      {X, Y} ->
        io:fwrite("X + Y is: ~p~n",[X + Y]),
        addFn();
    Other ->
       io:fwrite("Unknown"),
       addFn()
    end.

start() ->
    pid = spawn(fun() -> addFn() end),
    pid ! {6, 10}.
```

In the code snippet above, the output will be 16, because $\{6, 10\}$ matches with the pattern $\{x, y\}$.

Functional Programming - Haskell

Given the lack of side effects, a Haskell program appears to have many opportunities for automatic parallelization, however, in practice, this may not work as beautiful as it seems because it creates many small items (high order functions are broken down by many other smaller functions) which can not be efficiently scheduled.

Haskell provides a mechanism to allow the user to control the of concurrency by indicating what computations may be done. To use the power of concurrency one must use the $Control.Concurrent$ module. For that, the use of $MVars$ is very important once they do the managing concurrent access to shared resources, and communicating across threads. See the code below:

```
import Control.Concurrent (forkIO,
```

```
                                    threadDelay)

main = do
  res <- newEmptyMVar

  forkIO (do
    threadDelay (10000)
    putMVar result 42
    putStrLn "Wait! What's the question?")

  putStrLn "Waiting for answer..."
  value <- takeMVar result
  putStrLn ("The answer is: " ++
              show value)
```

The output for the code above will be:

Waiting for answer...
// After 10 seconds
The answer is: 42
Wait! What's the question?

Object-oriented Programming - Java
Java is a high-level programming language that allows multi-threading, meaning that a given program does not have only one thread (the main thread), it may other threads that can run concurrently in order to solve one or more tasks. Many times, these different threads may be doing different tasks such as event-handling while the main thread is displaying screen elements, for example. We have multitasking when two or more processes share common resources (processing, memory and such).

In Java, you must implement the *Runnable* interface or extend the class *Thread* if you want your code to be divided in multiple threads, allowing multitasking and consequently concurrency. Having a **Thread** class that implements *Runnable*, you must implement a method named $run()$ (defined by the *Runnable* interface). This method is called when your **thread object** is *started*. Check the example below:

```
class ConcurrencyDemo implements Runnable {
  private String tName;
  private Thread t;

  ConcurrencyDemo( String tName) {
    this.tName = tName;
    System.out.println("Creating " +
                       tName );
  }

  public void start () {
    System.out.println("Initiating " +
                        tName );
    if (null == this.thread) {
      this.t = new Thread (this, tName);
      this.t.start ();
    }
  }

  public void run() {
    // your business logic will be here
  }
}

public class ExecThread {
 public static void main(String args[]) {
   ConcurrencyDemo t1 =
     new ConcurrencyDemo( "1st Thread");
   t1.start();
 }
}
```

The code above will create an object that will run concurrent with our main thread.

Obs.: We also realize that Java can be very verbose when dealing with concurrency.

Scripting Language - Python *Python* has a number of different concurrency **constructs** such as *threading*, *queues* and *multiprocessing*. The *threading* module used to be the primary way of accomplishing concurrency. Python is also largely used and you can find many open source libraries over the internet related to concurrent programming, once it requires a lot of legwork in order to implement (and understand) it, if you compare with languages prepared for that, such as *Erlang*.

The code you will see below start 5 threads. If you think sequentially, the expected output would be 1 through 5, however, the operation does not wait for the threads to complete before moving on to the next print statement and the threading is not executed sequentially. To undertand it better, please check the following code and output:

```
from random import random
from threading import Thread
import time

def worker(number):
  slp = random.randrange(1, 10)
  time.sleep(slp)
  print("#{} ran asynchronously
          for {} seconds"
          .format(number, slp))

for n in range(5):
  thread = Thread(target=worker,
      args=(i,))
  thread.start()

print("What is the sequence?")
```

The output of the code above will be different every time you run it, but on my test was:

What is the sequence?
#1 ran asynchronously for 1 seconds
#3 ran asynchronously for 4 seconds
#5 ran asynchronously for 6 seconds
#2 ran asynchronously for 7 seconds
#4 ran asynchronously for 8 seconds

In the example above, I created 5 new threads (other than the main thread) by using $Thread(...)$ (but it is possible to subclass it and implement the code the same way one would do in a more object-oriented style, such as Java). Each one of them, will sleep for a random number of seconds and only them print *"#thread ran asynchronously for 'random' seconds"*.

Procedural - C
One must link the $pthread$ library in order to use most(if not all) of multithreading capabilities in **C**. There are a few particularities for that, such as:

1. In the $main()$ method, we declare a variable (let us call it tid or pid), with type of $pthread\_t$, which is type representing the id of a thread based on an integer (which is the same as explained on *Erlang* for pid), used to retrieve and handle the thread in the system.

2. After declaring tid, we then execute $pthread\_create()$, aiming to create a new concurrent process. It takes 4 arguments: a pointer to tid (or pid), a set of attributes related to tid, a *callback* of a function f_c (a function pointer, or a function address to a function that will be executed by the process) and the *args* that the function f_c will take.
3. Lastly, we will need to use $pthread\_join$ if we want the to wait for the calling thread related to the thread identifier tid (or pid) terminate.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *asyncFun(void *args);

int main()
{
   pthread_t tid;
   printf("Before Thread\n");
   pthread_create(&tid,
                  NULL,
                  asyncFun,
                  NULL);
   pthread_join(tid, NULL);
   printf("After Thread\n");
   return 0;
}

void *asyncFun(void *args)
{
   sleep(1);
   printf("Called from thread\n");
   return NULL;
}
```

The code above then creates a $thread$ and $waits$ until it returns (we do that by calling $pthread\_join$).

III. Evaluation

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

- Behaviour of program when it is handling thousands of random threads.
- Behaviour of program when possible *deadlocks* may occur.
- How long does a program take to respond to a given asynchronous request.

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

- **Erlang**: native profiling *fprof*.
- **Haskell**: *ghc* with option *-prof*.
- **Java**: *jprofiler* tool.
- **Python**: *profile.py* library.
- **C**: Mac OS native tool *Instruments*.

IV. Evaluation Environment

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

V. Discussion

I believe that, after evaluating all languages, **Erlang**, **C** and **Python** will have better performance than **Haskell** and **Java**. That will happen because *Erlang* is already designed to work with concurrency natively, whereas *C* is designed to work very close to the machine level and to the *OS*, having advantage over languages such as *Java* and *Haskell*. Likewise, most of *Python* libraries are developed in *C*, making it as fast as *C*.

Going further, without proper **benchmarks**, it will be hard to tell if *Erlang* will execute faster and better than *C* or *Python*. If you consider that *Erlang* does not have side effects by default (because it is also a functional language), one might say that it will be better if compared with many other powerful languages, however, side effects many times work as *shortcuts* in processing. Languages such as *Python* and *C* could use this factor to improve speed their code up through optimization, while *Erlang* would rely on its never-change state. It will be interesting to see how *Erlang* will handle

thousands of messages being received and sent from and to thousands of objects, while *C* will let the programmer avoid problems such as *deadlocks*.

That being said, in spite of being very reliable, *Haskell* will be executed very slow as said before, because 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, for example. Regarding *Java*, the *JVM* is already by itself slower than most of them, as it is another layer of abstraction. *Java* is expected to have other problems, such as bad handling of *deadlocks*.

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

- **1st place: C**
C is chosen to be the best one because it is one of the oldest languages amongst this set of languages. It also has very fast libraries and awesome compilers (*gcc* and *clang*). C is designed to work very close to *Assembly language* and is compiled, making its source as fast as our computers are.

- **2nd place: Erlang**
As said before, *Erlang* is designed for concurrent programming. It will probably run slower than C because it is compiled to *bytecode* and only then interpreted by its virtual machine (like Java). However, I am placing that before *Python* because of its roots based on multitasking.

- **3rd place: Python**
Python should be the third fastest language as its libraries are mostly written in C. The third place goes by the fact that it is an interpreted language.

- **4th 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 concurrency in Java is not a very good idea (may even be the reason why *Scala* has daily grown its community).

- **5th 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].