Concurrency in Ruby

Final Project

Course: CSCI 4060U

Professor: Dr. Jeremy S. Bradbury

University of Ontario Institute of Technology

By Calvin Lo

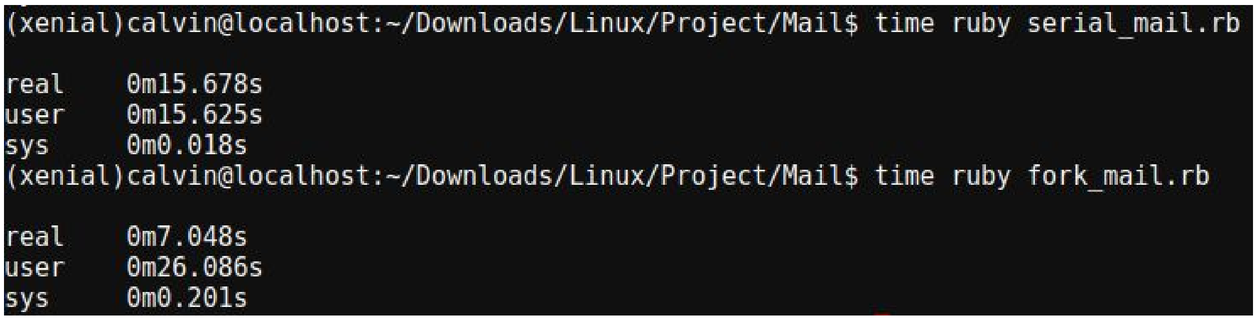
100514352

Introduction

Ruby is a programming language that “balanced functional programming with imperative programming.” [1] In this paper, we will discuss multiprocessing, multithreading, and how to achieve parallelism in Ruby. Concurrency improves the performance of a program and take advantage of the multi core computers. As Ruby is commonly used in web programming such as websites and web applications. Improving the performance in Ruby program can help to improve the web performance as well. Therefore, concurrency in Ruby is a good area to explore and discuss. Multiprocessing and multithreading in Ruby can easily implemented by using to defined classes. Parallel processing should have a better performance than serial processing. We will talk about the performance on different approaches. Additionally, we will discuss the limitation and problems when using multithreading in Ruby. In Ruby, concurrency means multiple threads run in a overlapping time. It includes the process of initialization, execution and completion. [2] On the other hand, parallelism is two threads running at the same time. [2] The difference is parallelism guaranteed the multithreading task will run simultaneously while concurrency is not necessary. This paper will also analyze the problem and solution when implementing real parallelism in Ruby programming.

Multiprocessing

The first approach to implement concurrency in Ruby is multiprocessing. Ruby has a Kernel class that support POSIX. It means it can used the POSIX kernel function but not the Window kernel. The first way to generate a process is using the Kernel.system. This method will execute the task in a subprocess and return true if run successfully. [4] However, the limitation of this method is the output limited to the main program destination. To create a independent subprocess, we can used Kernel.fork. It will make a copy the current process. A parent process can run the function to create multiple subprocess. The subprocess will return the process ID. [3] The program need to be able to wait for all process. To do this, we need to implement the Process.wait function. This function will wait for the process finish its job and return the process ID. [4] With the modificated mailing program originally created by Eqbal Quran [2], we can see the time difference of serial processing and multiprocessing.



```
(xenial)calvin@localhost:~/Downloads/Linux/Project/Mail$ time ruby serial_mail.rb

real    0m15.678s
user    0m15.625s
sys     0m0.018s
(xenial)calvin@localhost:~/Downloads/Linux/Project/Mail$ time ruby fork_mail.rb

real    0m7.048s
user    0m26.086s
sys     0m0.201s
```

When we run the mailing program in serial, the time expected in around 15 seconds. On the contrary, multiprocessing run around 2 time faster in the above result. Also, the user timing information showed the program did run in a multiprocessing setting. Multiprocessing increase the speed of a simple program.

However, there are some limitations and drawbacks. Multiprocessing is a concurrency that required a high memory usage. In addition, there are no optimization of copy-on-write in some Ruby implementation. [2] Copy-on-write is a computer technology to copy or duplicate a process. The memory use will significantly increase when creating a high amount of subprocess.

The another drawback about multiprocessing in Ruby is the difficulty in sharing data. All process in Ruby is independent when using fork. To share data between process or return a result from process is difficult. You will need to implement a pipe method to read and write the data. It will increase the memory use. When implementing the pi approximation method in Ruby with multiprocessing, the main challenge is how to share a pi variable between process. It is only sharing between subprocesses but also return back to the main process. The solution using IO pipe method to read and write the data in a separate file. The reason to apply multiprocessing to this program is to improve the performance. However, the memory use in copying process is already high and adding the IO pipe will increase the memory use. Therefore, it is the a good approach method to use multiprocessing here.

Multithreading

Multithreading is a technology that create multiple threads to finish different tasks. A multithreading program expected to run parallel which means multiple threads can run at the same time. It will result in a faster runtime. Ruby has a Thread class that support most of the general threading methods. The syntax of threading in Ruby is similar to pthread in C. Creating a thread in Ruby used the *new* method in the Thread class. In the block of code, you can define the function that the thread will execute. Similar to pthread, we need to *join* all threads before the main thread exited. This will make sure the main thread will wait for all threads finished their jobs before quit.

```
t = Thread.new { function() }
t.join
```

The context of code to create multiple threads in similar to pthread in C. You can use a loop in Ruby to create multiple threads that run the same function. Compare to pthread in C, Ruby allows directly code in the block without using another function. It means it allow multiple functions in a single Thread as well. Thread in Ruby will start the thread once CPU resources become available.[5] You don't need to start a thread. Thread in Ruby allows user to retrieve the current thread and the main thread by using the functions Thread.current and Thread.main. One of the special features of Thread in Ruby is it can store variables in a thread. It is different than pthread and OpenMP. Pthread and OpenMP do not support an individual variable of a single thread. You need to use a global variable or list to store a values of a specific thread. Here is an example on how you can assign and read a variable from a thread.

```
Thread.current["data"] = "mydata"
t["data"] #myData
```

Compare to multiprocessing, thread can store and share data between any threads included the main thread. It improve the functionality of concurrency.

Mutex is an important concept in threading. It can lock a variable that blocked any other thread to access the variable. Mutex can implement in Ruby by the *Mutex* class. *ConditionVariable* class provided the function of signal and wait of a lock. You can simply create a Mutex and ConditionVariable and use it in a thread. *Mutex.synchronize* is the method that initialize Mutex in a thread. Below is a sample code on handing mutual exclusion in Ruby threading.

```
m = Mutex.new
cv = ConditionVariable.new
t1 = Thread.new {
   m.synchronize {
      ...
      cv.wait(mutex)
      ...
   }
}

t2 = Thread.new {
   m.synchronize {
      ...
      cv.signal
      ...
   }
}
```

Deadlock solved by using the *signal* and *wait* methods. A multithreading program need to make sure there are no chance that all threads are waiting for a lock variable. Ruby has a documentation that provided a good tutorial on how to implantation thread.

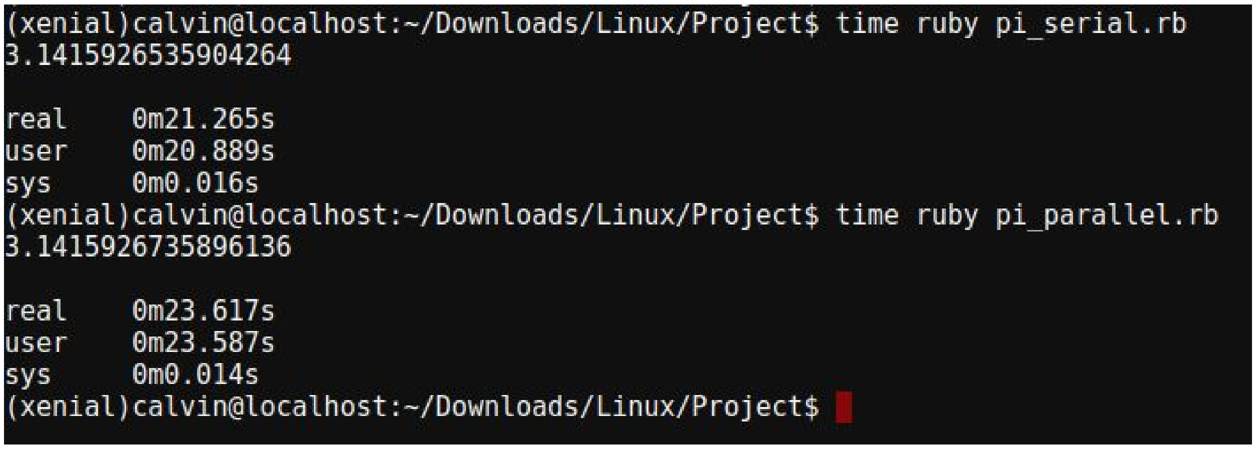
One of the drawbacks of multithreading in Ruby is the limitation of how many threads can be created. [2] Depends on the hardware, you can only create a limited number of threads.

Fiber Multithreading

Fiber is a lightweight thread that supports in Ruby. Fiber in Ruby require the user to start the fiber manually by using the *resume* method. This method will start the corresponding Fiber. However, the functionality in Ruby's Fiber is not well developed than normal Thread class. There are no function to deal with mutex exclusion or other situation of deadlock. Additionally, the performance of fiber is close to the Thread class. Fiber is basically a thread aims to provide a lightweight implementation. However, Fiber is a new class in Ruby which only introduced in Ruby 1.9. Thread still have a better use in Ruby.

Multithreading Parallelism Problem

The problem of multithreading is it does not support parallelism. I created a Ruby program that will approximate Pi in serial and parallel. The technology used in parallel is multithreading.

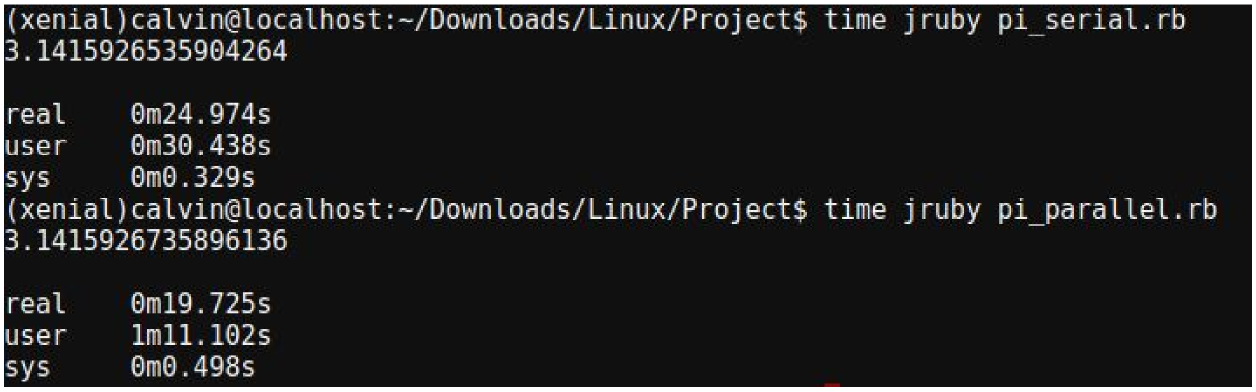


```
(xenial)calvin@localhost:~/Downloads/Linux/Project$ time ruby pi_serial.rb
3.1415926535904264

real    0m21.265s
user    0m20.889s
sys     0m0.016s
(xenial)calvin@localhost:~/Downloads/Linux/Project$ time ruby pi_parallel.rb
3.1415926735896136

real    0m23.617s
user    0m23.587s
sys     0m0.014s
(xenial)calvin@localhost:~/Downloads/Linux/Project$
```

As you see from the result above, there is no time difference between a serial program and parallel program. Ruby used the implementation of Ruby MRI. Ruby MRI, also known as CRuby, is the mainstream Ruby implementation used by the majority. The reason of not parallelizing is Ruby MRI used Global Interpreter Lock (GIL). It is a technology that synchronize the execution of threads and only allow one thread to run at a time.[2] Therefore, no matter how many threads you created, GIL will not allow the program to run the program simoustanly. It applied to every situations included in a multi core computer. Since Fiber is simply another implementation of Thread, it will not run multiple fibers at the same time as well. The thread will run in a random order but never in same time. Running only one thread at a time will limited the power of multithreading.

Solution to parallelism

To allow the multiple thread running at the same, there are no direct solution in Ruby MRI. You can not disable or stop the Global Interpreter Lock. It means there are no way to allow multiple threads to run at same time in Ruby MRI. However, the reason of this is the Ruby MRI used GLI. There are many different implementation of Ruby that used a different services. To allow real parallelism, you may need to consider a different implementation of Ruby. In this paper, we used JRuby to solve the problem. JRuby is another interpreter of Ruby. It used Java virtual machine (JVM). The most important part using JRuby here is it doesn't not implemented GLI. Below is the timing information about using JRuby to run the same pi approximation program.

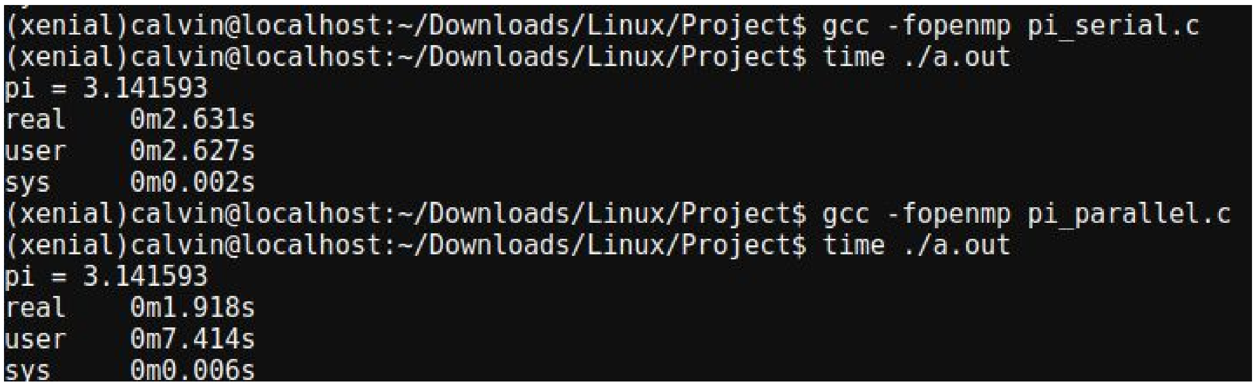


```
(xenial)calvin@localhost:~/Downloads/Linux/Project$ time jruby pi_serial.rb
3.1415926535904264

real    0m24.974s
user    0m30.438s
sys     0m0.329s
(xenial)calvin@localhost:~/Downloads/Linux/Project$ time jruby pi_parallel.rb
3.1415926735896136

real    0m19.725s
user    1m11.102s
sys     0m0.498s
```

Although the real time doesn’t have a significant difference, the user time has the difference. The user time information gives us the result that the program is running in multithreading. However, JRuby doesn’t support fork. It makes multiprocessing in JRuby harder. Parallelism in threading is important. We want a better performance when running a multithreading program. However, the Global Interpreter Lock limited the execution of thread. An alternative solution is using a different implementation of Ruby that without GLI but you may need to sacrifice some features in Ruby MRI. In JRuby, Fiber still cannot execute in real parallelism. All the testing is running in a Linux environment while the support of FIber in Linux is limited. Therefore, Fiber may not able to run expected. It is an another main topic to discuss about Fiber.

Comparison to OpenMP

OpenMP in C has a significantly faster performance in both serial processing and multithreading process than Ruby. Below is the performance of using OpenMP to do pi approximation.



```
(xenial)calvin@localhost:~/Downloads/Linux/Project$ gcc -fopenmp pi_serial.c
(xenial)calvin@localhost:~/Downloads/Linux/Project$ time ./a.out
pi = 3.141593
real    0m2.631s
user    0m2.627s
sys     0m0.002s
(xenial)calvin@localhost:~/Downloads/Linux/Project$ gcc -fopenmp pi_parallel.c
(xenial)calvin@localhost:~/Downloads/Linux/Project$ time ./a.out
pi = 3.141593
real    0m1.918s
user    0m7.414s
sys     0m0.006s
```

The program is testing with 8 threads. The time is less than 2 seconds for parallel processing while with JRuby, the time is 10 times slower. However, when you compare the serial processing, C is almost 10 times faster than JRuby as well. It means the slow runtime is because of the language of Ruby. Additionality, both OpenMP and JRuby Thread improved the performance of the program with concurrency.

Conclusion

In conclusion, concurrency can implemented in Ruby. Multiprocessing allows the program to copy main process and run the subprocess at the same time. However, it used many memory. Also, sharing data between process is difficulty and require even more memory. Although multiprocessing gives us a faster runtime, the limitations makes it not a good approach to concurrency. Multithreading used a similar syntax as pthread. The Thread class in Ruby supports multiple methods to achieve multithreading. The problem of multithreading is it does not support multithreading in Ruby MRI, which is the most used implementation of Ruby. The only solution to this is using an interpreter that without Global Interpreter Lock. JRuby is an alternative implementation that allow user to have a real parallelism.

References

[1] Ruby. (n.d.). Retrieved April 17, 2017, from https://www.ruby-lang.org/en/about/

[2] Eqbal Quran (n.d.). Ruby Concurrency and Parallelism: A Practical Tutorial. Retrieved April 17, 2017, from https://www.toptal.com/ruby/ruby-concurrency-and-parallelism-a-practical-primer

[3] Process. (n.d.). Retrieved April 18, 2017, from https://ruby-doc.org/core-2.1.2/Process.html#method-c-fork

[4] Thomas, D., Fowler, C., & Hunt, A. (2009). Programming Ruby: The pragmatic programmer's guide. (p.127-143)Raleigh, N. C.: Pragmatic Bookshelf.

[5] Ruby Multithreading. (n.d.). Retrieved April 18, 2017, from https://www.tutorialspoint.com/ruby/ruby_multithreading.htm

[6] Fiber. (n.d.). Retrieved April 18, 2017, from https://ruby-doc.org/core-2.2.0/Fiber.html