Concurrency Study of Python 2.7

MD. SAIDUL HOQUE ANIK, BUET, Bangladesh
MOHAMMAD MASUDUR RAHMAN KHAN, BUET, Bangladesh

Python provides a rich set of concurrency libraries. But not all of it's implementations perform equally well executing multi-threaded programs due to the presence of Global Interpreter Lock in some of the variants. In this paper, we have stacked up the most popular implementation of Python, the CPython, with two other widely known implementations of Python namely IronPython and Jython to measure their concurrency skill. These three distributions are fundamentally different among each other. Our experiment shows how well the last two implementations perform against CPython in executing concurrent programs. Finally, we have explored why some implementation perform better than the other from a developer's point of view.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: Python, Concurrency, Performance

ACM Reference Format:
Md. Saidul Hoque Anik and Mohammad Masudur Rahman Khan. 2017. Concurrency Study of Python 2.7. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Python is currently a popular language for not only engineering purpose, but also for scientific researches. This often involves working with big data and large number of inputs. To make this process faster, one might be tempted to use concurrency libraries. Making a proper use of concurrency features might speed up a program drastically, but failing to do so results in catastrophe.

Python, however, poses a different challenge. Some of it's implementations are written in such a way that only a single thread can run at a time. Due to this restriction, one might find the concurrent program acting in an unexpected manner. In this paper, we are going to compare three major implementations of Python with regards to concurrency and then we shall try to investigate why they perform in this way.

2 MOTIVATION

The most popular implementation of Python is written in C programming language, known as CPython. CPython has a mutex named Global Interpreter Lock. What GIL does is that it prevents multiple threads from executing Python bytecodes at the same time. Due to this restriction, the interpreter can be executed on only one processor core at a time. The usage of GIL provides an easy integration of C libraries that are not thread-safe. This provides a smoother implementation workflow for the language developers, but at the same time creates challenge for the Python programmers who want to use multi-threading in their programs.

CPython is written in native C language. PyPy, another variant of Python, is the meta-circular[1] implementation of Python. So both of these implementations suffer from the same problem. Writing

[1] Meta-circular interpreter means the interpreter is writter by a more basic version of the same interpreter. Python written by a more basic version of Python would be a meta-circular interpreter.

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

concurrent programs in any of these two implementations will not provide much of the speed-ups the developers would be hoping for.

Fortunately, there are other implementations that do not use GIL. Among them, there are IronPython and Jython. IronPython is written in .NET framework, and Jython is written in Java. The first one uses the Microsoft Framework for language development, the other one is deployed through a managed language (Java VM). The limitation of GIL does not apply to any of these implementations. In this paper, we are going to explore how a concurrent program behaves in these three types of implementations of Python.

3 METHODOLOGY

Our goal is to find out which implementation of Python performs better when a concurrent program in interpreted by it. For this purpose, we are going to write a non-concurrent code that performs a significantly time consuming task, and compute the elapsed time. Then we shall compute the elapsed time again, but this time by dividing the task into two equal portions, and running the sub-tasks into separate threads. The two elapsed time will give us some insights of the internal mechanism of the interpreters that we are using.

4 ANALYSIS OF DIFFERENT LANGUAGE IMPLEMENTATIONS

Among the implementations of Python, we find that there are three major divisions considering the source language. Some implementations, such as CPython etc. are written in native language (C language). Some implementations like IronPython are developed using Microsoft's Common Language Framework (.NET Framework). Other implementations include Jython which uses Java VM.

The native implementation written in C language provides fast compilation and execution of non-concurrent programs in. However, it comes with the restriction of the Global Interpreter Lock which affects the performance of CPU-bound programs. However, it greatly increases the speed of single-threaded programs.

The Common Language Runtime Framework of Microsoft is a great platform for creating programming languages. The .NET Framework provides a powerful set of base library including the concurrency modules that are identical to that of C# programming language.

Last, but not the least, Java VM is the top-notch programming language when it comes to concurrency. Multithreading features have been integrated deeply in this language from its very early stage of development. The combination of the powerful concurrency libraries with a managed environment makes Java ideal for multithreaded programming. So it is no surprise that the Java implemenation, Jython does not require any Global Interpreter Lock.

5 EXPERIMENTAL SETTINGS

Python, specially CPython, is still in it's active development stage. Thus, more than one stable version of CPython is available. However, the other two variants that we shall be using, i.e. IronPython and Jython, are both the implementation of Python 2.7 standards. So we have chosen the version of CPython accordingly.

5.1 Configuration Details

The configurations settings that are used in this experimental environment are given below.

Processor Configuration Core i5-3360M @ 2.80GHz, 2-Cores
Operating System Windows 7 Professional (SP1), 64-bit
CPython v2.7.13, 64-bit (AMD64)

IronPython v2.7.7, .NET 4.0 32-bit
Jython v2.7.0, Java HotSpot Client VM, jdk-1.8.0144

5.2 Source Code

A simple python function is used for this experiment that decrements one from a large number until it reaches zero. The divided subtasks do exactly the same work, but the large number is divided into two equal portions. The function calling procedure has already been described previously in Section 3.

The codes for three implementation of Python have subtle differences. The reason is that these variants implement the libraries of Python in slightly different ways. Their basic working procedures are, however, identical. The source codes can be found in Appendix A.

5.3 Other Experimental Settings

For measuring the execution time of the functions, the in-built library of Python known as ‘profile’ has been used. CPython and Jython produce the exact profiling for the target function. The version of IronPython that we are using, however, does not support the usage of ‘profile’ library directly. An easy yet effective workaround for this is to calculate the elapsed time from the ‘time’ library, which we have used in our source code. All of the source codes are compiled using the command line interface.

6 EXPERIMENTAL FINDINGS

This section is divided into two subsection. In the first subsection, we’ll show the calculated results. In the latter subsection, we shall try to analyze why this outcome has occurred.

6.1 Results

The program provided in Appendix A has both single-threaded and multi-threaded version. Table 1 shows the execution times for the variants of Python for multi-threaded and sequential (single-threaded) version of the program.

Table 1. Execution Time (Seconds)

	Thread1	Thread2	Sequential
Jython	6.327373	6.298202	9.5541
IronPython	1.7679	1.735856	3.249749
CPython	7.46432	7.419355	3.856387

Single-threaded vs Multi-threaded program performance is visible in Figure 1. We can see the combined side-by-side result in Figure 2.

Combined Performance Comparison chart considers each thread individually. On the other hand, we have taken the average of two thread execution time as concurrent execution time while calculating the overall improvement.

6.2 Result Analysis

Individually, it’s no surprise that CPython will perform poorly when concurrency is introduced in code due to Global Interpreted Lock. It is also expected that the other two variants will show significant improvements on the latter case.

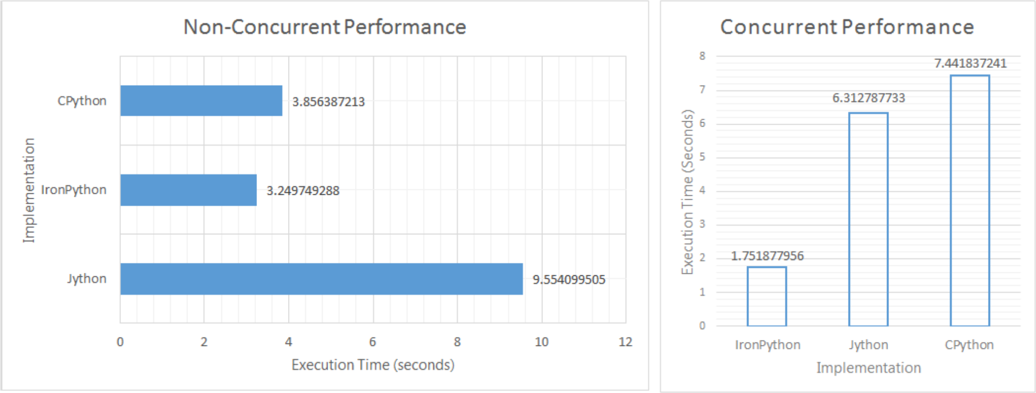

Fig. 1. Non-concurrent vs Concurrent Execution Performance

As for the comparison among the non-concurrent version of the program, CPython and IronPython provide more optimized code than Jython. As Java uses Just-In-Time compiler, it requires a bit of time to ‘profile’ the program to generate a more optimized version. Our experimental setting does not provide sufficient room for this to happen. This is why Java performs poorly during the non-concurrent phase.

When concurrency is introduced, CPython immediately takes the back-seat, leaving the competition entirely between Java and .NET. As we have mentioned earlier, the JIT of Java HotSpot requires method profiling to generate faster and well-optimized code. Therefore, it becomes the runner-up, leaving the first place for IronPython.

When we see the overall picture, we see that IronPython performs astonishingly well among the three with a staggering improvement of 46.09%. This is not just because the other two competitors are not well-suited for the job, but it is due to the nature of Common Language Runtime (CLR) Framework of Microsoft that generates better low level codes in generic scenarios.

7 CONCLUSION

In this paper, we have performed an experiment that depicts the concurrency performance among the three major implementation of Python. We have performed a comparison among the implementation by a native language, a managed language, and a Framework. Finally, we have tried to shed some light on the insight of why the results are the way they are. In future, we would like to explore more concurrent libraries and features of the newer versions of Python and see how they perform is in other implementations as well.

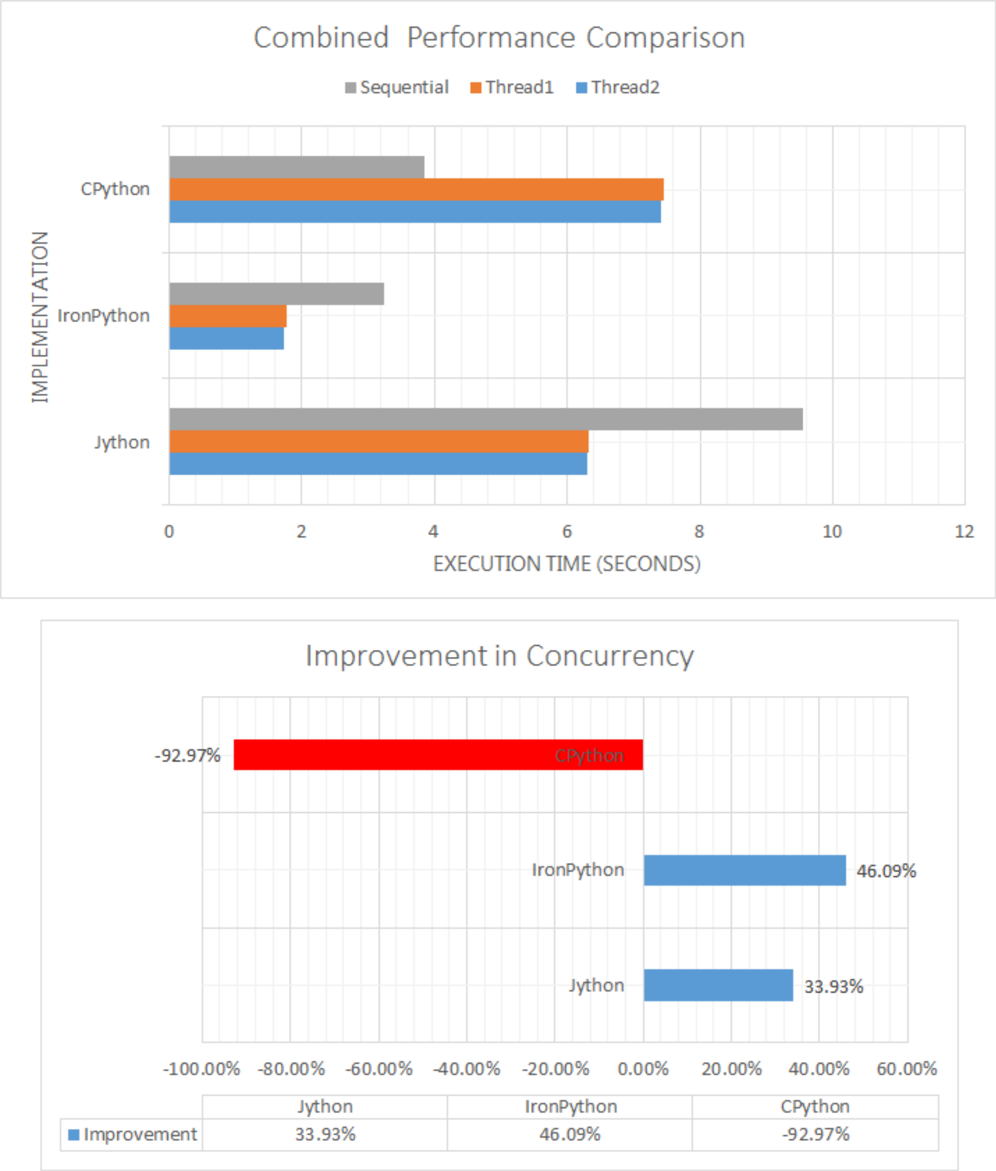

Fig. 2. Performance Evaluation in Concurrency

A APPENDIX

This section contains the code snippets used for concurrency comparison. Code snippets are slightly different in the three languages as they vary in Python library implementation.

A.1 CPython

```
#from System.Threading import Thread, ThreadStart
```

```
import threading
import time
import profile

COUNT = 100000000
def count2():
        n = COUNT/2
        t1 = time.clock()
        while (n>0):
                n-= 1
        t2 = time.clock()
        print "Thread Done\n" + str(t2-t1)

def count():
        n = COUNT/2
        t1 = time.clock()
        while (n>0):
                n-= 1
        t2 = time.clock()
        print "Thread Done\n" + str(t2-t1)

def f1():
        profile.run("count()")

def f2():
        profile.run("count2()")

#t1 = time.clock()

#t1 = threading.Thread(target=f1)
#t2 = threading.Thread(target=f2)
#t1.start()
#t2.start()
f1()
f2()
#thread.start_new_thread(count, (COUNT/2,))
print "Done\n"
```

A.2 IronPython

```
from System.Threading import Thread, ThreadStart
import time
import profile

COUNT = 100000000
def count2():
        n = COUNT/2
        t1 = time.clock()
        while (n>0):
                n-= 1
        t2 = time.clock()
        print "Thread1 Done\n" + str(t2-t1)

```

```
def count():
        n = COUNT/2
        t1 = time.clock()
        while (n>0):
                n-= 1
        t2 = time.clock()
        print "Thread2 Done\n" + str(t2-t1)

def f1():
        count()


def f2():
        count2()

tic = time.clock()

#t1 = Thread(ThreadStart(f1))
#t2 = Thread(ThreadStart(f2))
f1()
f2()
#t1.Start()
#t2.Start()
#thread.start_new_thread(count, (COUNT/2,))
print "Done\n"
```

A.3 Jython

```
#from System.Threading import Thread, ThreadStart
import thread
import time
import profile

COUNT = 100000000
def count2():
        n = COUNT/2
        t1 = time.clock()
        while (n>0):
                n-= 1
        t2 = time.clock()
        print "Thread Done\n" + str(t2-t1)

def count():
        n = COUNT/2
        t1 = time.clock()
        while (n>0):
                n-= 1
        t2 = time.clock()
        print "Thread Done\n" + str(t2-t1)

def f1():
        profile.run("count()")

```

```
def f2():
        profile.run("count2()")

#t1 = time.clock()

#t1 = thread.start_new_thread(f1, ())
#t2 = thread.start_new_thread(f2, ())
#t1.Start()
#t2.Start()
f1()
f2()
#thread.start_new_thread(count, (COUNT/2,))
print "Done\n"
```