Final Project: Multiprocessing Performance Analysis with Python

Jonathan King & Kuber Sethi

CS 314: Principles of Programming Languages

Due Date: May 12th, 2019

1 Project Overview

This project aims to explore how multiprocessing affects the performance of various workloads, where it excels, and where it doesn't. In particular, we used the MultiProcessing library in Python, and implemented a diverse set of workloads for testing ranging from Web Scraping popular websites, to famous Computer Science Algorithms.

1.1 Project Goals

We had two main goals with this project:

- **Exploration of Advanced Python:** Both of us were beginners in Python at the beginning of this course, and really enjoyed the Python topics discussed in class. We wanted to expand on our Python knowledge, and multiprocessing seemed interesting and useful. Additionally, as Rutgers students we can take advantage of the extremely powerful machines that iLab provides, so multiprocessing was attractive from that angle as well. After completing this project, both of us feel more proficient in Python, and enjoyed learning an advanced feature of Python like multiprocessing.
- **Understand the Strengths and Weaknesses of Multiprocessing:** Both of us had heard about multiprocessing from outside study, and other coursework, but we didn't know how it worked in the real world, on real tasks. For example, we often heard that some video games struggle to use multiple cores, but we didn't really understand what that meant. With this project, we wanted to see why some software allowed us to use all of our CPUs, and others didn't. We wanted to learn which types of tasks excelled with multiprocessing, and which didn't.

2 Project Methodology / Design

With such large goals, we first had to plan out the specific design of our project, and specific workloads. A big challenge we faced was how we were going to develop on the iLabs, and deal with dependencies.

2.1 Challenges

We faced a few challenges while working on this project. The first was dealing with multiple python environments across our individual laptops as well as the iLabs. In addition, it was difficult to use tools like pip and conda as many installs required write access to certain folders that we do not have access to. This is why we were unable to install pyinstaller correctly so that we could create an executable. However, we were able to use Anaconda successfully to export a list of dependencies needed for our application.

2.2 Workload Type 1: Web Scraping

- **A: Weather Finder**
  This workload explores the common task of web scraping - navigating to a URL and extracting some data from it through a program. We built a program that would retrieve weather data for a list of cities, and store them in a CSV file. Multiprocessing was used to speed up the gathering of data for each city, and for certain types of data.

2.3 Workload Type 2: Mathematical Computation

- **A: Generating Prime Numbers**
  Mathematical Computation is often where multiprocessing shines, so we wanted to see for ourselves how it performed. Generating prime numbers is a computationally expensive operation, and given that we're writing them to a file, it gives us more room to explore how multiprocessing impacts performance.

- **B: Determining if a Number is Prime**
  Similar to the previous workload, we implemented this to compare the performance between generating a prime number, vs checking if a number was prime.

2.4 Workload Type 3: File I/O

- **A: File Compression**
  Compression is infamous for being taxing on CPUs, and we wanted to see how multiprocessing could affect it's performance. We used the zlib library, a popular compression library, and chose the texts of various Disney plays as compression examples. The workload would compress and decompress these texts several times, split across multiple processes.

- **B: Image Encoding**
  Combining aspects of Web Scraping with File I/O, this workload goes to https://www.reddit.com/r/pic/, retrieves the top images, and encodes them into a string with Base64. This is extremely computationally expensive, especially with high resolution images, so multiprocessing was a perfect fit here.

2.5 Workload Type 4: Computer Science Algorithms

- **A: Sorting Algorithms**
  We wanted to see how multiprocessing could improve sorting performance, so we wrote a program that could compare performance across many sorting algorithms. In this test, we used multiprocessing to split up the sorting of 100000 arrays of size 100. Since the workload was evenly split up between several processes, the overall runtime decreased.

- **B: Fibonacci Generator**
  While the Fibonacci Sequence is rooted in Mathematics, it's implementation falls in Computer Science. Although there are many different ways to implement Fibonacci, with differing performance, we chose the naive brute force method as it would allow us to observe the impact of multiprocessing over all else.

3 Program Functionality:

We utilized the Pool class in the multiprocessing library. The Pool class has a map funciton that works similarly to the vanilla map python function. However, this map function takes an extra argument which specifies how the mapping is going to be split up between threads. We utilized this to test the runtimes of each program using various number of threads. We then added functionality using Matplotlib that would print the results in a line graph. This can be generated in the python application or one can look over the sample graph.png that we have provided.

4 Workload Results:

Please note the graph.png file in the zip folder. The tests were run on cd.cs.rutgers.edu, an iLab machine with an Intel i7-7700 @ 3.6Ghz, containing 8 vCores. Calculating primes had the greatest runtime decrease when we implemented multiprocessing. Sorting and encoding images were close behind, however it seemed that the runtimes would plateau after about 6 processes. Other processes actually did not decrease in runtime at all. Rather, trying to multiprocess the algorithm actually made the overall program slower, due to the increase in overhead outweighing the performance benefit. This was the case in examples like Fibonacci, where the multiple processes were unable to work together to compute faster, and instead just added more overhead.

5 Analysis of Results:

We found some very interesting results after running our tests. For example, we noticed that there were some tests that did not decrease runtime when implementing multiprocessing. The graph (which is included in the zip folder) shows this clearly. Compression, for example, did not get any faster when we implemented multiprocessing. This was somewhat expected, as we were aware that setting up a new process would take a lot of overhead. In the case of compressing files, it actually takes longer to set up the new process than it

does to run the algorithm on a single process. Therefore, in situations like this, it is not worth using multiprocessing. Additionally, recursive programs like Fibonacci generation are unable to take advantage of multiprocessing, since processes lack shared memory access, so they are unable to build off of each other to improve performance. Multithreading would perform better for these types of programs, but in Python, this is difficult due to the Global Interpreter Lock.

6 What We Learned:

We learned that it is very important to design projects in advance. One must not rush into development especially with larger projects. For example, it makes sense to conclude that multiprocessing would decrease the overall runtime in all situations, however we learned that this is simply not the case. In addition, we learned the importance of containers and virtual environments. We both had to make sure that we had the same exact python environments as well as the same dependencies in order for our program to run the same on both of our computers. We used Anaconda in order to work with our virtual environments, but we tried many other solutions as well, such as Jupyter notebooks, PipEnv, etc.

Additionally, while this project was very practical with it's implementation and use of the MultiProcessing library in Python, we came across some interesting theory as well. For example, we learned about Amdahl's Law, which gives the theoretical maximum increase in performance for parallel programs. We also came across design philosophies for Python, like why multiprocessing is so prominent for performance increases versus in other languages, where multithreading is more common. This is due to Python history, where the Global Interpreter Lock was needed to deal with memory management issues, and how it has stuck with the language till today, making multithreaded programs useless.

Going forward, both of us will definitely explore more with the MultiProcessing library. We feel like we improved heavily with this project, and are now much better Python programmers. Each workload may have been relatively short in length, but after this project we learned how to use the basics of NumPy, MatPlotLib, BeautifulSoup, JSON querying, and more. Although we didn't explicitly cover processes or the other APIs we used in this course, we covered a wide range of diverse material that inspired us to create such a diverse set of workloads.