**test plan and results for**

**Parallel Programming in Python:**

**Determining the Most Efficient Way to Perform CPU-Intensive Multiprocessing Operations**

1 Introduction 2

1.1 Scope 2

1.1.1 In Scope 2

1.1.2 Out of Scope 2

1.2 Quality Objective 2

2 Test Methodology 3

2.1 Overview 3

2.2 Test Levels 3

2.3 Test Completeness 3

3 Resource & Environment Needs 4

3.1 Test Environment 4

4 Testing Results 4

4.1 Testing Level 1 4

4.2 Testing Level 2 4

5 Conclusion 5

5.1 Testing Level 1 Results 5

5.2 Testing Level 2 Results 5

5.2 Final Conclusions 5

# Introduction

This software was designed to perform CPU-intensive tasks in parallel using the multiprocessing module in Python so that the efficiency of each of three different methods of data can be measured quantitatively.

The research hypothesis that is being tested with this software is twofold:

* The simpler a solution in the module is in functionality the more efficient it will be.
* The simplest, and therefore most efficient, solution in the module would be simplex pipes, followed by duplex pipes. The most complex, and therefore least efficient, solution available in the module would be using a multiprocessing pool.

## Scope

### In Scope

The three features found in the multiprocessing module of Python that will be tested by this software are:

* Simplex Pipes
* Duplex Pipes
* Multiprocessing pools

### Out of Scope

This software will not test any features outside of the multiprocessing module of Python. It will also only focus on situations where subprocesses are running CPU-intensive tasks, and will only use two different tasks that have been chosen to test the efficiency of the chosen channels between the main processes and its subprocesses. This software will also only be designed for use within one operating environment, and the differences in performance found using other operating systems and other computer architectures will remain out of scope for this testing.

## Quality Objective

The overall objective that I plan to achieve with this testing is to harness data that can be used to determine which feature of the Multiprocessing module in Python is most efficient when performing CPU-intensive tasks with subprocesses running in parallel.

# Test Methodology

## Overview

The testing methodology selected for this project was the WaterFall method. This Software Engineering model was selected because the requirements for this project were clearly defined and stated beforehand, and there was very little possiblity of unplanned events or uncertainity that would need to be accomodated. Since there was little to no chance of the requirements changing throughout the duration of the project this seemed like the easiest model to plan and manage.

## Test Levels

There will be two levels of testing performed during this project:

* The first level of testing will be conducted by sending small amounts of data sent through the 3 testing channels, which will be to attempt to determine how much overhead each of the channels has based on its fundamental construction.
* The next level of testing will be conducted by sending large amounts of data through the 3 testing channels, which will be to measure the efficiency of each of the channels during actual tasks and with the CPU under strain.

## Test Completeness

Testing will be deemed complete for each level of testing when:

* Level 1: Each multiprocessing method has been utilized 100 times using a small amount of data through each CPU-intensive function. The small amount of data for this test will be a list of prime numbers from 1 to 10 for the primes function and a list of hash digests from 1 to 10 using SHA512 for the encryption function. These functions will be completed 10 times each in the main process, and the main process itself will be looped 10 times overall for 100 total passes.
* Level 2: Each multiprocessing method has been utilized 100 times using a small amount of data through each CPU-intensive function. The large amount of data for this test will be a list of prime numbers from 1 to 10,000 for the primes function and a list of hash digests from 1 to 10,000 using SHA512 for the encryption function. This represents 1,000 times more input into and output from each to be processed using each multiprocessing method. . These functions will be completed 10 times each in the main process, and the main process itself will be looped 10 times overall for 100 total passes.

# Resource & Environment Needs

## Test Environment

The following hardware was used in the creation and testing of this software:

* Intel Core i5-9300H CPU
* 8.0 GB Physical Main Memory
* 25.8 GB Virtual Main Memory

The following software was used in the creation and testing of this software:

1. Windows 10.0 Home Build 19041
2. Visual Studio Code 1.55.1
3. Python 3.8.8 64-bit programming environment

# Testing Results

## Testing Level 1

Small Data Testing:

Total processing time for Simplex Pipes: 126.641 (2 minutes 6.6 seconds)

Total processing time for Duplex Pipes: 123.299 (2 minutes 3.3 seconds)

Total processing time for Multiprocessing Pool: 170.399 (2 minutes 50.4 seconds)

Average time for each main loop: 41.600 seconds

Total overall processing time for program: 416.002 (6 minutes 56 seconds)

## Testing Level 2

Big Data Testing:

Total processing time for Simplex Pipes: 429.067 (7 minutes 9 seconds)

Total processing time for Duplex Pipes: 425.004 (7 minutes 5 seconds)

Total processing time for Multiprocessing Pool: 467.325 (7 minutes 47.3 seconds)

Average time for each main loop: 89.142 (1 minute 29.1 seconds)

Total overall processing time for program: 891.423 (14 minutes 51.4 seconds)

# Conclusion

## Testing Level 1 Results

The first round of testing revealed a clear overhead difference between the two modules using pipes and the multiprocessing pool module. The function using simplex pipes outperformed the multiprocessing pool function by 43.8 seconds per execution on average, and the function using duplex pipes did so by 47.1 seconds on average. The advantage that the function using duplex pipes had over the function utilizing simplex pipes was much slimmer at 3.3 seconds per execution on average. It should be noted, however, that although the processing time advantage for the function utilizing duplex pipes was small, it was also consistent – there was not a single pass of any of the functions where the function using simplex pipes outperformed the function using duplex pipes.

## Testing Level 2 Results

The second round of testing was able to verify the observations revealed by the first. It clearly demonstrated that the overhead difference between the functions remained proportionally the same, regardless of the amount of data used as input or collected as output. In this test, the function using simplex pipes outperformed the multiprocessing pool function by 38.3 seconds per execution on average, while the function using duplex pipes outperformed the multiprocessing pool function by 42.3 seconds per execution on average. The duplex pipes function again had a slim but consistent processing time advantage over the function using simplex pipes, this time of 4 seconds per execution on average.

## Final Conclusions

The data gained from testing this software shows that the most efficient method for parallel processing in Python is the duplex pipe. In both rounds of testing, regardless of the amount of input applied and output collected, the duplex pipe method took less processing time than either of the other two methods tested.

The initial hypothesis is partially supported by the results in that the multiprocessing pool is the least efficient method of parallel processing data that was tested here. This validates the hypothesis that there must be processing overhead involved in managing the multiprocessing pool that removes some CPU power that could otherwise be directed at performing the subprocesses.

The initial hypothesis is challenged by the results in that the duplex pipes functions proved more efficient than the simplex pipes functions were. A possible explanation for this result would be that the creation of two separate pipes created more overhead than was saved by each pathway for data being more efficient. Testing this hypothesis would be outside the scope of this project, but would make an interesting addition to future iterations of this project.

The final reflection that we get from these results is that a programmer wanting to harness the parallel processing power of Python should consider the method that is used for multiprocessing carefully. If the final product is designed to run briefly or to handle smaller amounts of data it could easily be argued that the time and effort saved by using the much more user-friendly multiprocessing pool module could make it a viable choice. If, however, the final software is designed to handle very large amounts of data or run for very long time periods than duplex pipes would appear to be the most efficient choice available of the three methods tested in this project.