LEVERAGING COMPILED LANGUAGES TO OPTIMIZE PYTHON FRAMEWORKS

By

Ethan Mayer

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

December 16, 2023.

Nashville, Tennessee

Approved:

Gabor Karsai, Ph.D.

Abishek Dubey, Ph.D.

ACKNOWLEDGMENTS

I would like to thank Professor Gabor Karsai for his help and guidance during my graduate career. I would also like to thank my family for their never-ending support and generous funding that allowed me to complete the work required to write this thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

The objective of this thesis was to investigate possibilities of improving the performance of frameworks such as the Resilient Information Architecture Platform for Smart Grid framework (RIAPS) [1], a large, embedded software framework written entirely in Python. Due to the nature of the Python programming language, fast performance was traded for ease of use and accessibility, and the result was a platform that is not focused on being a strictly real-time system. However, novel techniques were investigated to determine which is the ideal solution to enhance the performance of RIAPS. Using faster, compiled languages, such as C++, performance-centric parts of RIAPS can be sped up significantly. This is then implemented and tested in order to quantify the performance improvement. All of the code developed for this research was written by me can be found on GitHub (https://github.com/EthanMayer/Leveraging-Compiled-Languages-to-Optimize-Python-Frameworks) under the open-source Apache License Version 2.0.

I.1 RIAPS

RIAPS is a software platform that allows developers to build applications for modern systems using a component-oriented approach. Recently, the push towards distributed, real-time, and embedded computing necessitated the creation of a framework to support this. The primary application of this software is the “Smart Grid”, the future of the internet-connected energy grid, where RIAPS can serve as a foundational tool for creating distributed computing systems. RIAPS was created by the Vanderbilt Institute of Software Integrated Systems (ISIS) with funding from

the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy. RIAPS includes tools such as an application run-time system (component model, messaging framework, security framework), run-time services (discovery service, deployment service, time synchronization service), design-time tools (modeling language, code generators), and operation services (application deployment and control). RIAPS aims to provide programmers with both design and run-time tools for building embedded software that can be used in areas such as the Smart Grid.

The platform is written entirely in Python due to its ease of use and general convenience. Consequently, performance is largely limited by the language's single core nature, and it cannot be considered a real-time system. "Single core" in this instance means that all Python code will ultimately run on a single processor. Although the *threading* Python library is used to handle the execution of developer applications, this is simply a convenient abstraction that is ultimately executed sequentially for each instance of the Python Interpreter. Thus, since no true parallel multithreading takes place, RIAPS acts as a single-core program with both the RIAPS platform and developer application sharing time on a single processor. Although the platform is designed to be run in a distributed environment, with several different nodes distributed throughout a Smart Grid, the speed of each individual node in the network is limited by that node's single-core performance. Theoretically, multiple actors can be run on the same node with multiple components, each in a separate actor. However, this is a very performance-intensive solution since a separate Python interpreter would be used for each actor.

The ideal solution would consist of using parallel software threads to handle running the developer applications on separate cores. True threads can run on separate cores in parallel. With these threads, the RIAPS platform can be separated from the developer application(s) and run in

parallel. Thus, RIAPS's overhead can be separated from the developer code via separate threads, and each developer application thread would run faster, resulting in a more real-time system. *POSIX Threads*, otherwise known as *pthreads*, can be used to spawn separate, parallel threads so there is no performance bottleneck occurring on a single processor core. Since RIAPS is written in Python, a different language will have to be used to run the developer application while using a communication library, such as *ZeroMQ* (*ZMQ*), to communicate back with RIAPS.

CHAPTER II

Background

This chapter will lay the foundation for the motivation for this thesis by explaining the context of the research. Once the technical aspects of languages used in RIAPS are fully presented, the reason for this research will become clear. The details of both Python and Cython are discussed, as well as the motivating factors driving this project.

II.1 Python

Python [2] is one of the most popular modern programming languages. Unlike traditional compiled languages, such as C or C++, Python is an interpreted language; the CPython interpreter, written in C, interprets Python code (in the form of bytecode) that runs on a virtual machine. The Python interpreter, as it is currently implemented, does not take advantage of multiple processor cores that may be available to it. Usually, there exists only one interpreter to run Python programs, thus in general, two Python threads cannot be run in parallel, only concurrently on a single processor core. A Global Interpreter Lock (GIL) exists that only allows the Python interpreter to be used by a single Python thread at a time. Unlike C or C++, there exists no true multithreading, and all "threading" done in Python is ultimately sequentially executed via sharing time on a single processor. Thus, the multicore computing power that accompanies all modern CPUs cannot be leveraged by Python to improve performance. This limitation is visualized in Figure II.1 below.

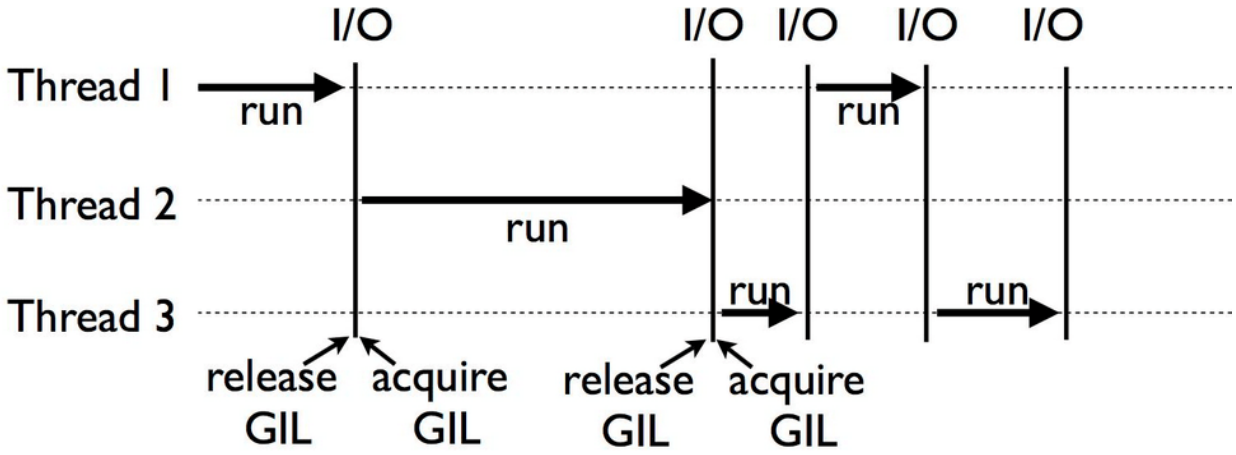

Figure II.1: Python GIL Visualization with Concurrent Python Threads [3].

ZeroMQ, also known as *ZMQ*, is the library used for communication in RIAPS and the other implementations in this research [4]. *ZMQ* is a fast and lightweight open-source networking library that provides a framework for communicating with different threads. This library is available in multiple languages, including Python and C++, and also allows messaging between them using objects such as sockets and ports. It offers multiple different transport methods, most notably transports such as in-process and inter-process, and allows for easy implementations of common communication patterns, such as communication between a pair of sockets. It should be noted that, due to the thread-safe nature of ZMQ and the prevalence of threads in both RIAPS and this research, a socket must first be created by the thread that intends to use it.

II.2 Cython

Cython [5], commonly described as “Python at the speed of C”, is a programming language that is a superset of the Python programming language with the additional, optional ability to declare and use C types and functions. Cython is a compiled language that generates C/C++ files that, when compiled, are automatically wrapped in interface code, producing extension modules that can be

imported by regular Python code. The intended use case of Cython is for developers to pinpoint slow and generally computationally intensive components of their project and then either replace them with Cython or write Cython code to interface with a C/C++ file written to do the same job. The step-by-step Cythonize process can be seen below in Figure II.2. In this figure, *setup.py* is the Python file that is normally used for all Cython projects that defines aspects of the Cythonize process and initiates it. *hello.pyx* is the Cython file, as denoted by the *.pyx* file type, that will be Cythonized. After calling *setup.py*, the Cython compiler generates *hello.c* from the *hello.pyx* Cython code. This *hello.c* file is then compiled, by an ordinary C compiler, into the shared library file *hello.so*. From here, the actual Python program, *launch.py*, can import the Cython function from this shared library.

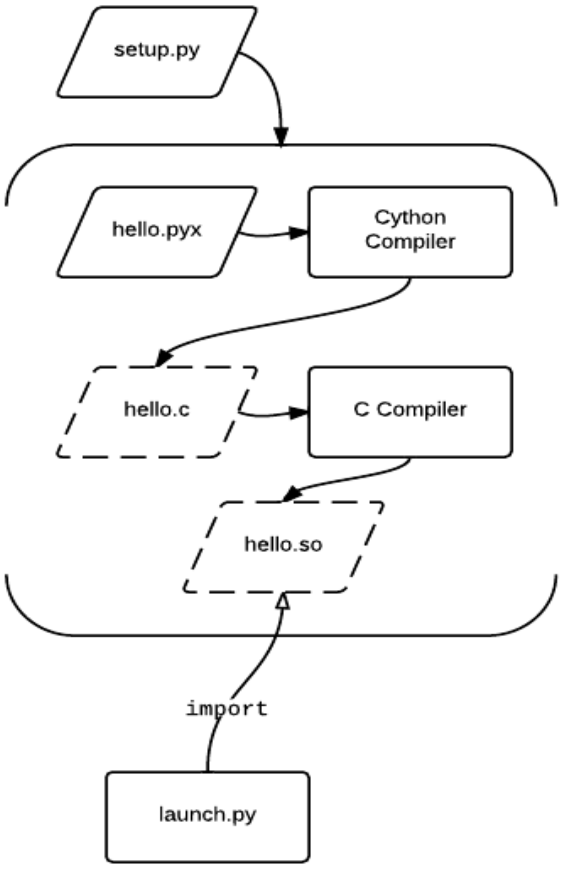

Figure II.2: Cythonize Process [6].

As stated previously, since Cython is a superset of Python, all Python code and libraries natively work with Cython, meaning Python code can call Cython code and vice versa. Additionally, Cython can import and use any C/C++ library as long as a Cython header file is made to wrap the C header files. In fact, several Python libraries already have Cython modules, such as ZMQ, since Cython is used in the backend of the libraries to speed them up. Since the result is a mixture of Python and C/C++ code, Cython gives developers a large amount of flexibility between the ease of use of Python and the speed and technical ability of C/C++. Thus, this option seems particularly enticing when considering methods to speed up Python-based RIAPS.

II.3 Motivation

In the last decade, single-core performance has started to plateau. As Moore's law slows down, the rate at which single-core performance is improving is reducing drastically. There are many factors contributing to this slowdown; the inability to keep doubling the number of transistors on a die is hindering potential maximum performance. Additionally, more transistors and faster clock frequencies lead to more heat production and energy consumption. The ability to draw heat off a die with the same surface area as a former die with lower heat production is difficult. The drastic increase in energy consumption for marginally faster clock speeds is leading to a slow-down in the rate at which single-core performance is increasing.

To counteract this, modern CPUs have put more emphasis on including multiple independent cores on each chip. As newer generations of CPUs are released, more individual cores continue to be added to each chip. Even if single-core performance increases slow down, adding additional cores can allow the industry to keep up exponential speed gains. On paper, benchmarks show multicore performance increasing exponentially and far outpacing single-core performance. This

performance difference was recently quantified by Dr. Abinash Roy in 2009 and visualized in Figure II.3 [7].

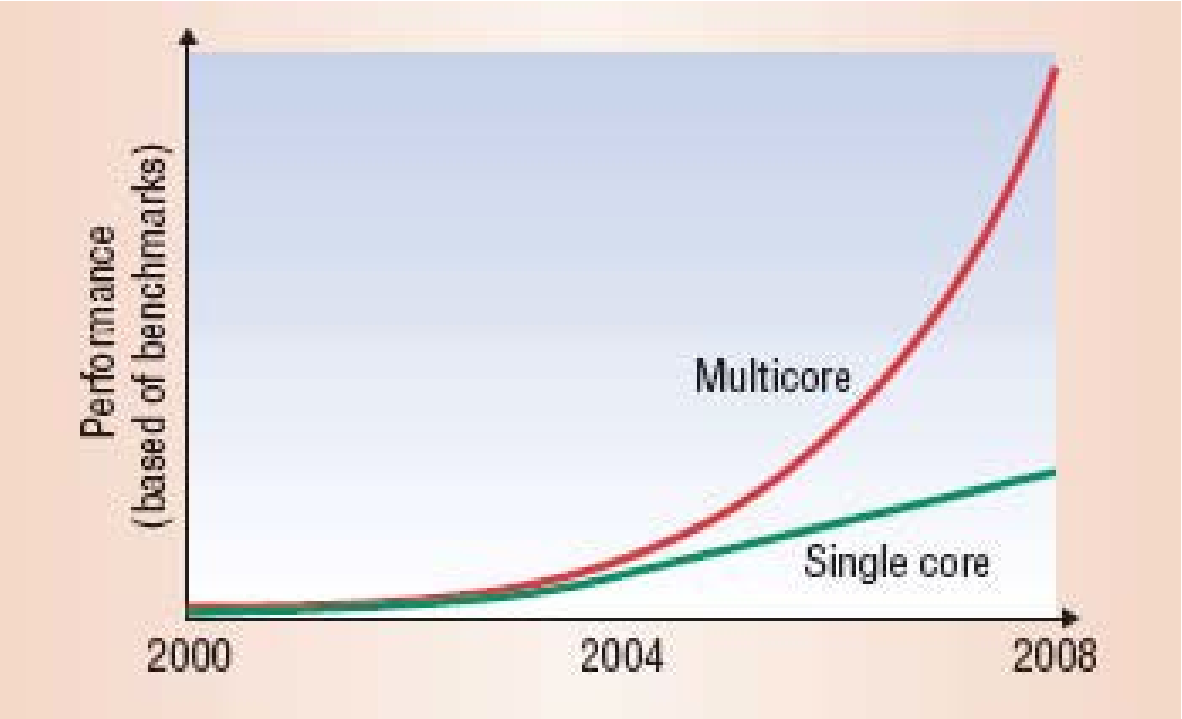

Figure II.3: Performance comparison between single-core and multi-core Intel processors using SPECint2000 and SPECfp2000 benchmarks [7].

According to their research, increasing the performance of a single core by 13% will require a 20% clock frequency increase, leading to a 73% increase in power consumption. On the other hand, decreasing clock frequency by 20% reduces performance by 13% but also reduces energy consumption by 49%. Thus, if multiple cores can be effectively utilized, they can each be individually slower and still yield faster overall results with less energy consumption. The future of fast and energy-efficient computation is multicore, so modern frameworks must be designed with this in mind.

Since Python is strictly single-core, RIAPS as a framework is also strictly single-core. Python's *threading* library allows the use of Python threads, which are non-preemptive and concurrent, and synchronization primitives, such as locks, that in tandem can be used to separate tasks into concurrent threads within the same Python interpreter. Although the *threading* Python library is used as an abstraction to separate the developer-defined application from the RIAPS runtime, all code is ultimately executed sequentially via sharing time on a single core. Sharing time on a single processor is done via context switching, which occurs when the processor swaps from executing one thread to another. The default context switching interval in Python is 5 milliseconds, so each thread is run for that amount of time before the next thread is run. A visualization of this single-threaded behavior can be seen below in Figure II.4. Each *ComponentThread* is a Python thread that runs a developer's application code. These Python threads are placed vertically in the visual below to denote their concurrent, not parallel, execution.

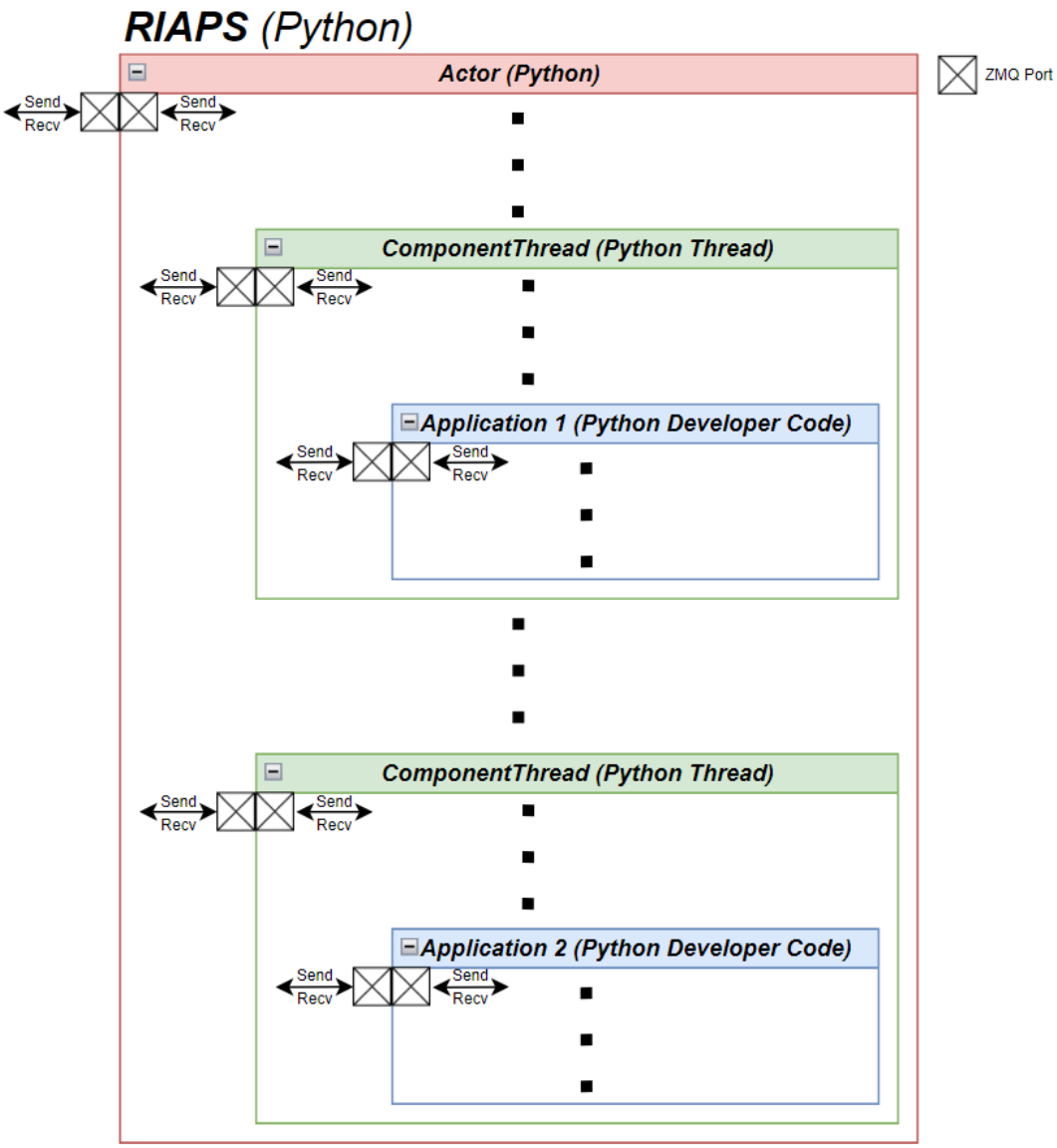

Figure II.4: RIAPS Python Hierarchy.

Due to this single-core nature and considering the slow-down of single core performance improvements, large performance improvements will not be seen when upgrading processors or microcontrollers. In contrast, upgrading to multiple cores opens up the opportunity for significant performance increases. Additionally, the base energy cost of running a Python program at a certain speed would be much higher than running an equivalent, perfectly multithreaded program at the

same speed. RIAPS is intended to be distributed across embedded devices, which are generally energy-conscious and not using high clock frequencies. Thus, a more modern and multithreaded approach is warranted.

CHAPTER III

Implementation

This chapter will discuss the implementation of the solutions created during this research. As explained in **Chapter II,** the motivation for these implementations was to design a proof-of-concept multicore solution for RIAPS. As the project progressed, multiple different prototypes in different programming languages were made in order to determine the most optimal solution for RIAPS. The two primary implementations created were a Python + Cython Prototype and a Python + C++ Prototype.

III.1 Cython Investigation

A significant amount of research went into investigating whether Cython would be a suitable solution to making RIAPS a more real-time system. Cython's intended function is to be used as a drop-in replacement for the backend of slow Python code or libraries in order to speed them up; RIAPS's needs fit this requirement well. Replacing the mechanism that handles spawning threads for the developer's application would contribute greatly toward making RIAPS a more real-time system. Since this proposed solution is intended to be a drop-in replacement through the use of Cython, a very minimal amount of RIAPS that has been written in Python would need to change.

The first step was to study Cython and create a proof-of-concept prototype before attempting to integrate anything into RIAPS. Fortunately, since Cython is simply a superset of Python, there was only minimal syntax that needed to be learned. Even when using C-style syntax, such as for static typing, Python's normal syntax patterns were followed, making the process a smooth transition.

Compiling Cython was simple, and when importing from other Python programs, there was no noticeable difference from the importer's perspective when compared to importing normal Python libraries.

This process became more difficult when implementing multithreading. The goal was to demonstrate using Cython as a bridge between a pure Python program and pure POSIX threads. While researching, it became apparent that using the *pthread* library to spawn true POSIX threads within Cython was a relatively novel and rare approach to parallelization in Cython. In Cython, when a C library is to be imported and used, a Cython header file must be made, similar to a C header file, declaring all classes and functions in Cython syntax. For many C libraries, such as the standard library, *ZeroMQ*, and others, these header files already exist because of their use in Python. However, due to its novel nature, for the *pthread* library, a custom Cython header file had to be created to allow the import and use of *pthread* in Cython. Once I created this header file that declared all classes and functions in *pthread*, it could be imported by Cython.

The goal was to change the RIAPS hierarchy seen previously in Figure II.2 to a truly parallelized environment seen below in Figure III.1. Each ComponentThread is a pthread launched by Cython that runs a developer's application code. These pthreads are placed horizontally in the visual below to denote their parallel and simultaneous execution. The only change that was required for this implementation was converting the comp.py Python file that usually handles component threads into a comp.pyx Cython file. Thus, all Python syntax stays the same while additional C-style syntax was added to import the pthread library and launch pthreads instead of Python threads. Since the developer application and RIAPS runtime are distinct and not directly reliant on each other, parallelizing the developer application threads would be an ideal solution. Not only would the RIAPS overhead be free to handle more tasks, but the developer application

could also be more performance-intensive without slowing down the overall framework execution. This is especially important for modern CPUs with many separate processor cores.

The goal was to change the RIAPS hierarchy seen previously in Figure II.2 to a truly parallelized environment seen below in Figure III.1. Each *ComponentThread* is a *pthread* launched by Cython that runs a developer's application code. These *pthreads* are placed horizontally in the visual below to denote their parallel and simultaneous execution. The only change that was required for this implementation was converting the *comp.py* Python file that usually handles component threads into a *comp.pyx* Cython file. Thus, all Python syntax stays the same while additional C-style syntax was added to import the *pthread* library and launch *pthreads* instead of Python threads. Since the developer application and RIAPS runtime are distinct and not directly reliant on each other, parallelizing the developer application threads would be an ideal solution. Not only would the RIAPS overhead be free to handle more tasks, but the developer application could also be more performance-intensive without slowing down the overall framework execution. This is especially important for modern CPUs with many separate processor cores.

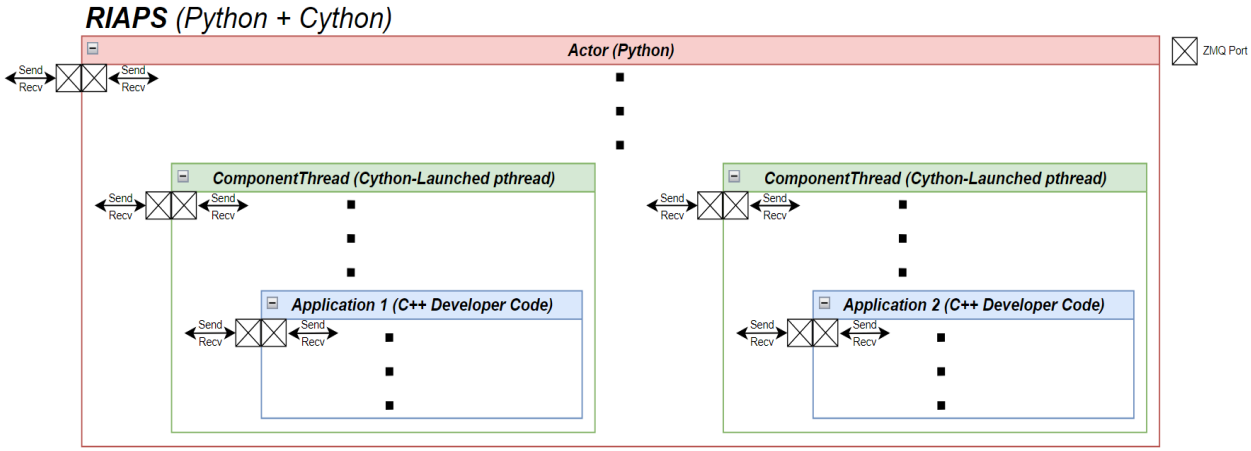

Figure III.1: RIAPS Python + Cython Hierarchy.

Problems arose when considering the interactions between the Python framework and C-based *pthreads*. Firstly, debugging is a challenge when compared to Python or C individually. Cython's usual debugging process uses an extension for the GNU debugger (*cygdb*) so that Cython code can be debugged via the command line. With this, the convenience of visual debugging via IDE tools usually present for Python and C is lost. Additionally, using *cygdb* alone is not sufficient to debug the entire program. The upper-level Python running RIAPS may also have to be debugged. Additionally, *cygdb* will not account for the *pthreads* spawned by Cython, so these will have to be debugged separately if needed. Working on a production-level and security-minded framework such as RIAPS requires a significant amount of attention to detail when it comes to solving complex issues, so difficulties experienced in debugging could be prohibitive. A comprehensive multi-level Cython debugger has been investigated [8], and a graphical overview of it can be seen below in Figure III.2. This debugger intended to unify the debugging of the multiple layers of Cython into a comprehensive debugger complete with a GUI. However, using this debugger proved challenging and time-consuming, and questions arose whether this comprehensive debugging solution was worth the effort to use. While it allowed for comprehensive debugging of Cython programs, the time required for the setup and learning process made it not worth using.

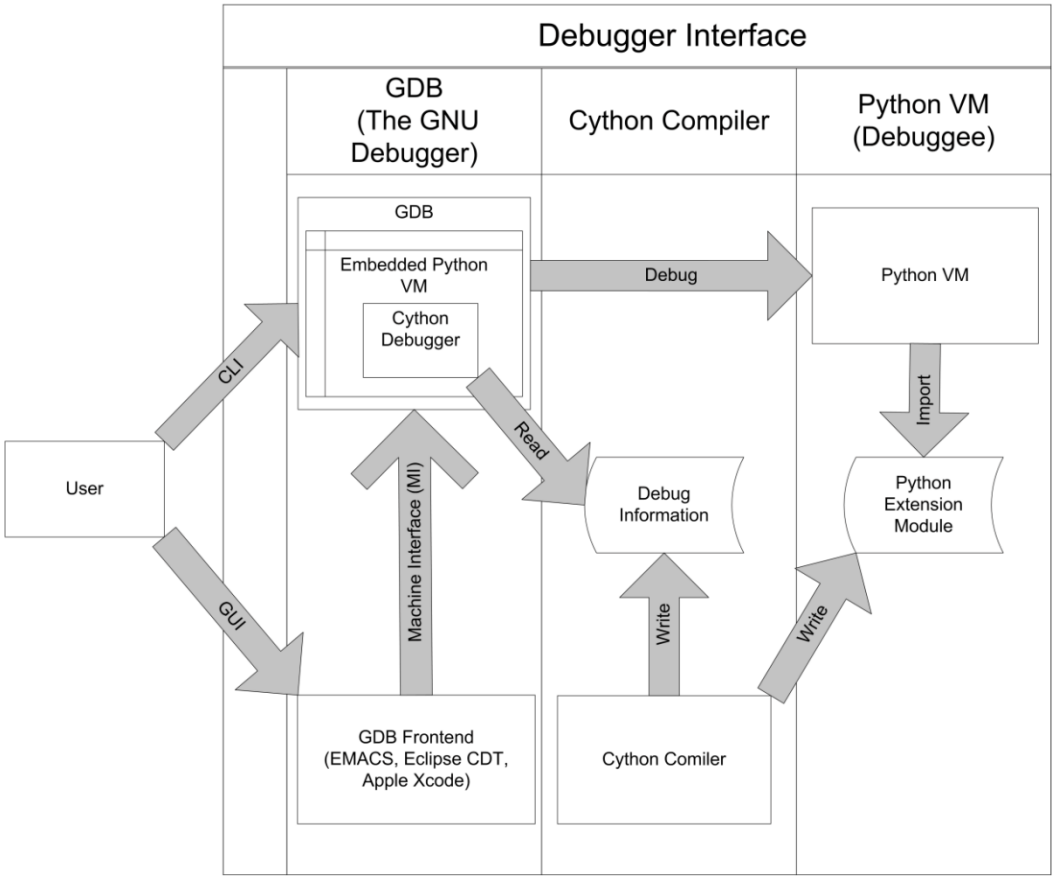

Figure III.2: Cython Debugger Graphical Overview [8].

Memory management across all of these different levels of the hierarchy also presents another challenge. One of the benefits of Cython was the seamless integration with Python, thus making it an ideal facilitator when transferring Python data structures to the *pthreads*. Using cross-language libraries, such as *pybind11*, would allow the C layer to receive and operate directly on Python objects used by RIAPS. However, since the Global Interpreter Lock would still have to be used when accessing these data structures, performance could be lost, potentially to the point of defeating the purpose of this research.

When taking all of these inconveniences and challenges into account, there may be more practical alternatives to integrating the speed of compiled languages in RIAPS. Thus, the research

pivoted at this point towards eliminating this middle ground and thus eliminating the added complexity from it. Cython was originally chosen due to the novelty of this particular use case and for the perceived ease of integration so that it can act as a bridge between Python and C. However, the added complexity outweighed the benefits.

III.2 C++ Investigation

After realizing that Cython was likely not the ideal solution to optimizing RIAPS, a simplified approach was the new goal. Going directly from Python to a compiled language, such as C++, would be a stark transition but would avoid the complexity of an intermediate layer. Thus, the research pivoted to directly bridging the gap between RIAPS's Python runtime and developer applications made with C++.

The core motivation and goal of the Python + C++ Prototype remains the same as the previous investigation; in order to observe a performance increase, the most inefficient aspect of RIAPS must be tackled. Developer application threads must use real, parallelizable threads, such as *pthreads*, so that a single processor core does not have to execute both the RIAPS overhead and developer application concurrently. Thus, the developer application must be written in C++. The goal remains to change the RIAPS hierarchy seen previously in Figure II.2 to a truly parallelized environment; however, in this implementation, Cython will not be used. The new proposed hierarchy diagram is seen below in Figure III.3. Each *ComponentThread* is a *pthread* launched directly by C++ that runs a developer's application code. These *pthreads* are placed horizontally in the visual below to denote their parallel and simultaneous execution.

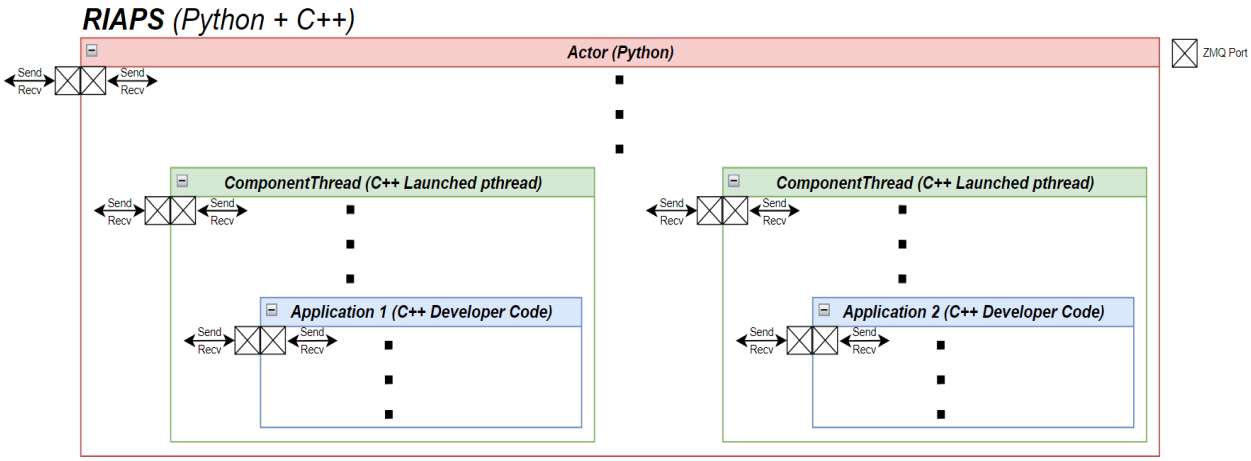

Figure III.3: RIAPS Python + C++ Hierarchy.

Since the Cython intermediate layer has been removed, Python will now directly interface with the lower C++ layer. Using the *CDLL* function from the *ctypes* Python library, C++ functions found in a shared library can be executed. A shared library, which is a *.so* file on Unix systems and a *.dll* file on Windows, is a compiled file that cannot be executed alone but instead is meant to contain functions that can be loaded and called by other executables or shared libraries at load time or runtime. This method is done as opposed to being directly copied by the linker at compile time and bundled into the executable.

However, when a C++ function in the shared library is loaded and called by Python, this function is not necessarily executed in parallel with the calling Python program. This C++ function must instead act more like a *main* function. No core functionality was executed here except for launching *pthreads* so that other C++ functions, also loaded from the shared library, can be executed in parallel.

Since there is less of a bridge between the Python and C++ layers, all communication occurred through sending *ZMQ* messages back and forth between RIAPS and the developer application C++ threads. Since the original implementation of RIAPS held this same design philosophy, not many

additional changes would be required to be made to RIAPS. As discovered with Cython in the previous section, Python objects can no longer be sent as-is due to the GIL requirement. In order for C++ to manipulate these Python data structures directly, the GIL would have to be acquired by the *pthread* each time they are accessed, which introduces the same performance issues seen in Python due to the presence of a single Python interpreter.

In order to ensure the communication system is language-agnostic, all data structures being exchanged between Python and C++ were serialized into a universal format, such as JSON. While Python's native serialization methods, such as *pickle*, do not include native support for complex data structures, such as named tuples, a library such as *jsonplus* was used to handle all JSON-related functions. Similarly in C++, nlohmann's *json* library was used for the deserialization and serialization of all message data after being received and before being sent, respectively.

Before integrating into RIAPS, a prototype was implemented as a proof-of-concept of this design. As described above, an upper-level Python program will load and call a C++ main function from a shared library. This main function will then spawn *pthreads* that run in parallel and handle receiving/processing/sending data to and from the Python level via JSON. A visualization of this prototype can be seen below in Figure III.4.

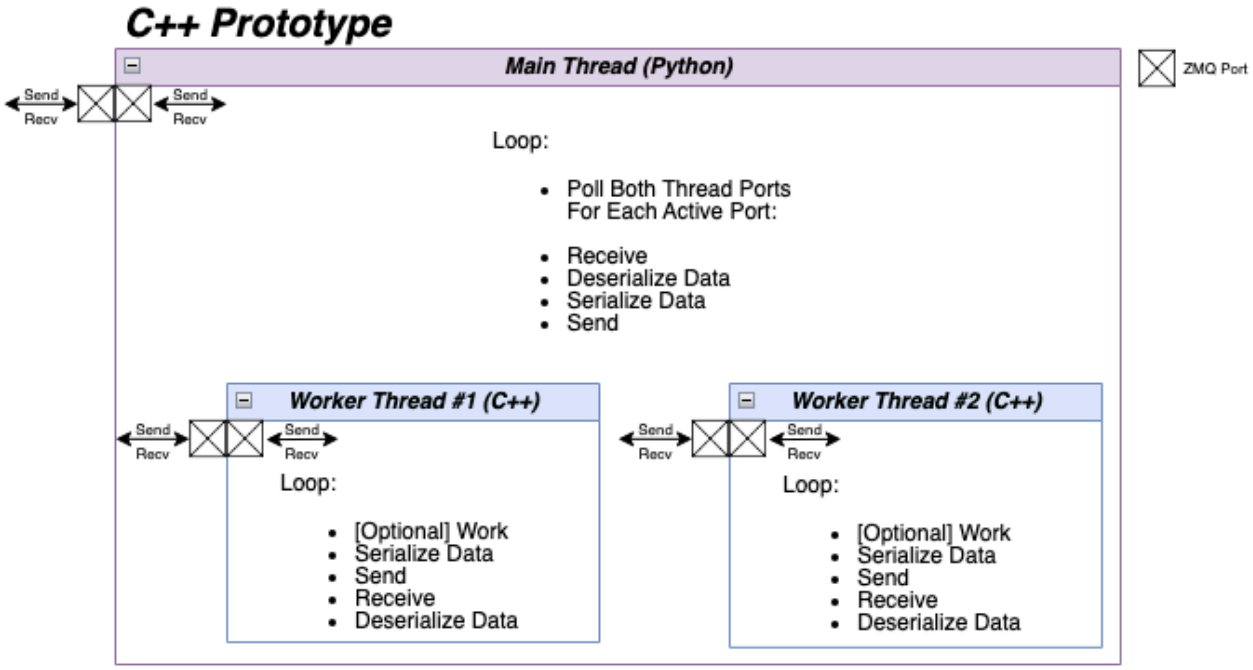

Figure III.4: Python + C++ Prototype Implementation.

The general flow of the prototype is as follows: the "main" thread, written in Python, launched a C++ function from a shared library. After launching this function and passing the correct parameters to it, which are necessary to setup the ZMQ ports, etc., the only purpose of the Python main thread was to send and receive messages while JSON serializing and deserializing them, respectively. The *ZMQ Poller* was used to handle detecting incoming messages on the two ports, one for each thread, so that no time is wasted being blocked while waiting to receive from one of the threads.

Each of the C++ worker *pthreads* executed in parallel simultaneously since no locking devices, such as mutexes or semaphores, are present due to the absence of shared resources. Each thread performed some sort of work, of which there are five options: no work, light math computation (iteratively calculate the Fibonacci number of the square root of the current message number), heavy math work (iteratively calculate the Fibonacci number of the current message number),

function calls (recursively calculate the Fibonacci number of the current message number), and memory allocation (allocate and free memory between each message). All of this work was in addition to the normal JSON serialization done upon sending a message and deserialization done upon receiving a message. Once the total number of target messages was achieved, the program exited.

III.3 Python Control

The primary purpose of the Python Control Implementation was to generate test data representative of a baseline of performance. Since RIAPS is programmed entirely with Python, a control in this same form was needed to perform comparisons against the other tests performed for this research. The Python Control Implementation functioned identically to the other Python + C++ Prototype, but the entire program was written in pure Python. Thus, due to the GIL, all threading was ultimately executed sequentially, and these results are expected to be the slowest in all cases. The work done by the Python main thread and the Python worker threads was identical in each respective test case to the work described above for the Python + C++ Prototype. A visualization of this workflow can be seen below in Figure III.5.

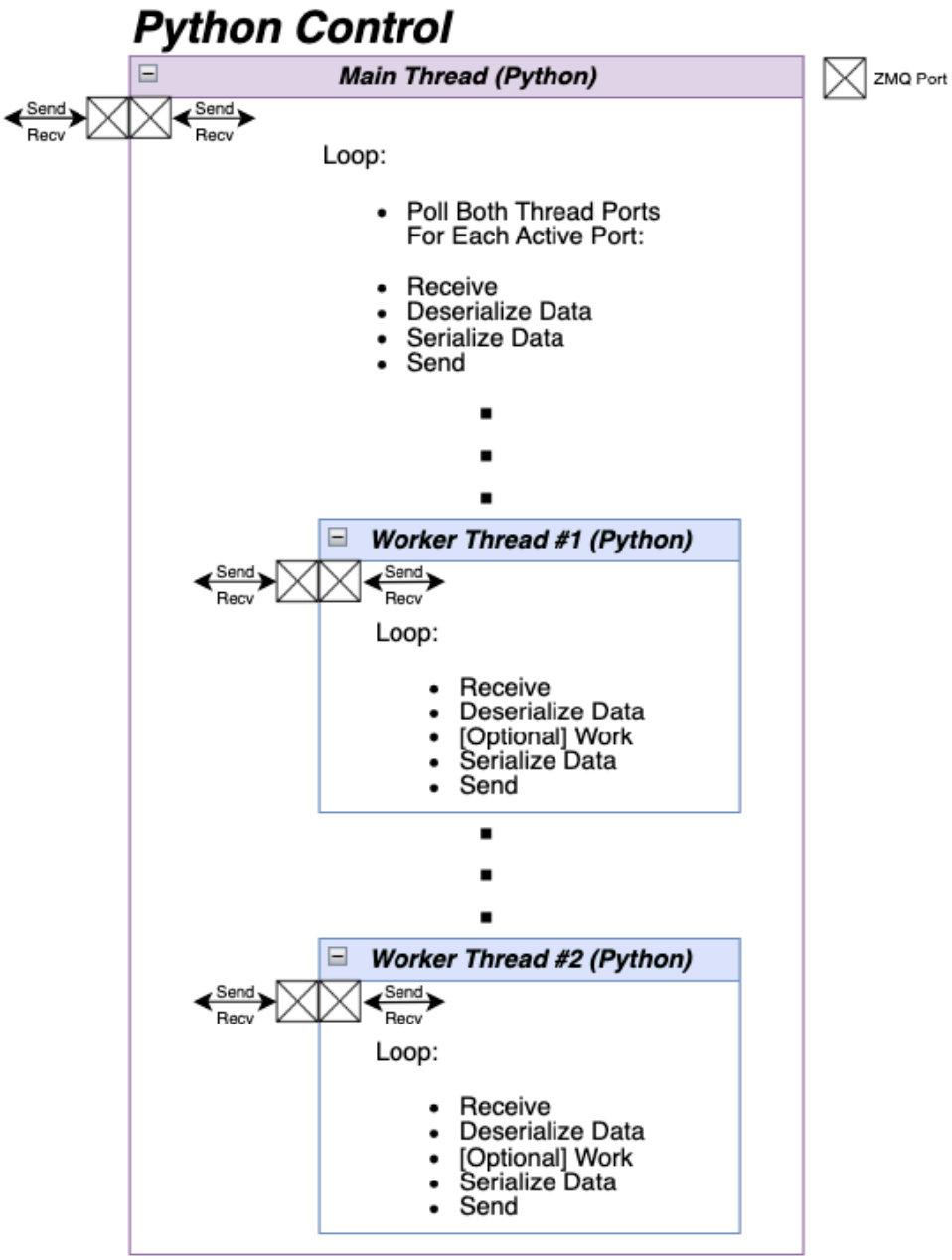

Figure III.5: Python Control Implementation.

CHAPTER IV

Results

This chapter will discuss the results of the prototypes implemented in **Chapter III**. Before actual integration into RIAPS can occur, these prototypes need to produce quantitative data to serve as justification for their proof-of-concept designs. The Python + C++ Prototype will have test results shown alongside a Control Implementation that is written in pure Python. Several different test categories will be shown for different workloads. These tests are listed in Table IV.1. The parameters and variables present in each test are detailed in Table IV.2.

Test	Message Amount	Work	Notes
Messaging Tests	100k; 500k;	None	Only work done is JSON serialization/deserialization
Light Computational Load Tests	100k; 500k	Square Root Fibonacci	The Fibonacci number of the square root of the current message number is calculated iteratively
Heavy Computational Load Tests	1k; 5k	Fibonacci	The Fibonacci number of the current message number is calculated iteratively
Function Calls Tests	10; 40	Recursive Fibonacci	The Fibonacci number of the current message number is calculated recursively
Memory Allocation Tests	100k; 500k	Memory Allocation	100 data structures of 4 KB size are allocated and freed

Table IV.1: Test Details.

Test Parameter	Value	Notes
CPU	Apple M1	ARM; 8 cores; no thermal throttling or power saving; starts idle
RAM	16 GB	No memory bottlenecks, such as swapping, occur
Runs per Test	10	Ensures external factors are averaged out
Printing/Logging	None	No printing/logging done to avoid bottlenecks (writing to output/buffer)
Main Thread	Python	All implementations have a Python main thread
Worker Threads	2	All implementations use 2 worker threads; thread language varies
Communication	ZMQ	All communication is done via PAIR sockets
Message Format	JSON	Data serialized before being sent; messages deserialized after being received
Message Number	Variable	Number of pairs of messages sent/received varies
Work	Variable	Amount and type of work (if any) done between messages varies; see details for each test's work below

Table IV.2: Test Parameter and Variable Details.

For every test in this chapter, certain test parameters were able to be kept constant. Every test was run on an Apple M1 ARM 8-core processor, and there was no throttling present, whether for thermal reasons or for power-saving. No tasks or apps were actively running in the background, and the CPU started at idle for each test. The system has access to 16 gigabytes of RAM, so no memory issues occurred, such as swapping, to slow the tests down. Since there are many external factors that can affect the performance of a program being benchmarked, such as OS scheduling or interrupts, 10 runs were performed sequentially for each test to ensure these external factors can be

averaged out. Additionally, no printing or logging of any kind is done, so the test results will not be influenced by waiting for I/O or waiting to write to a buffer. During each test, a Python main thread launches two worker threads, both in Python for the Python Control Implementation or both in C++ for the Python + C++ Prototype, and these threads are communicated with via ZMQ PAIR sockets. Each pair of messages consists of one message sent to a worker thread and one message sent back to the main thread. All messages sent consist of data in JSON form. Upon receiving a message, it is deserialized and the data is extracted. When sending a message, the data that is to be sent is first serialized into JSON. This process is visualized in the message sequence diagram seen below in Figure IV.X. In this diagram, the main thread and both worker threads are shown with their *ZMQ* PAIR sockets and the messages between them. Two types of example tests, the 100,000 message tests and the 500,000 message tests, are shown in the diagram. The points at which the test timer starts and ends are clearly demarcated as to visually show which parts of the program are being timed for the results later in this chapter.

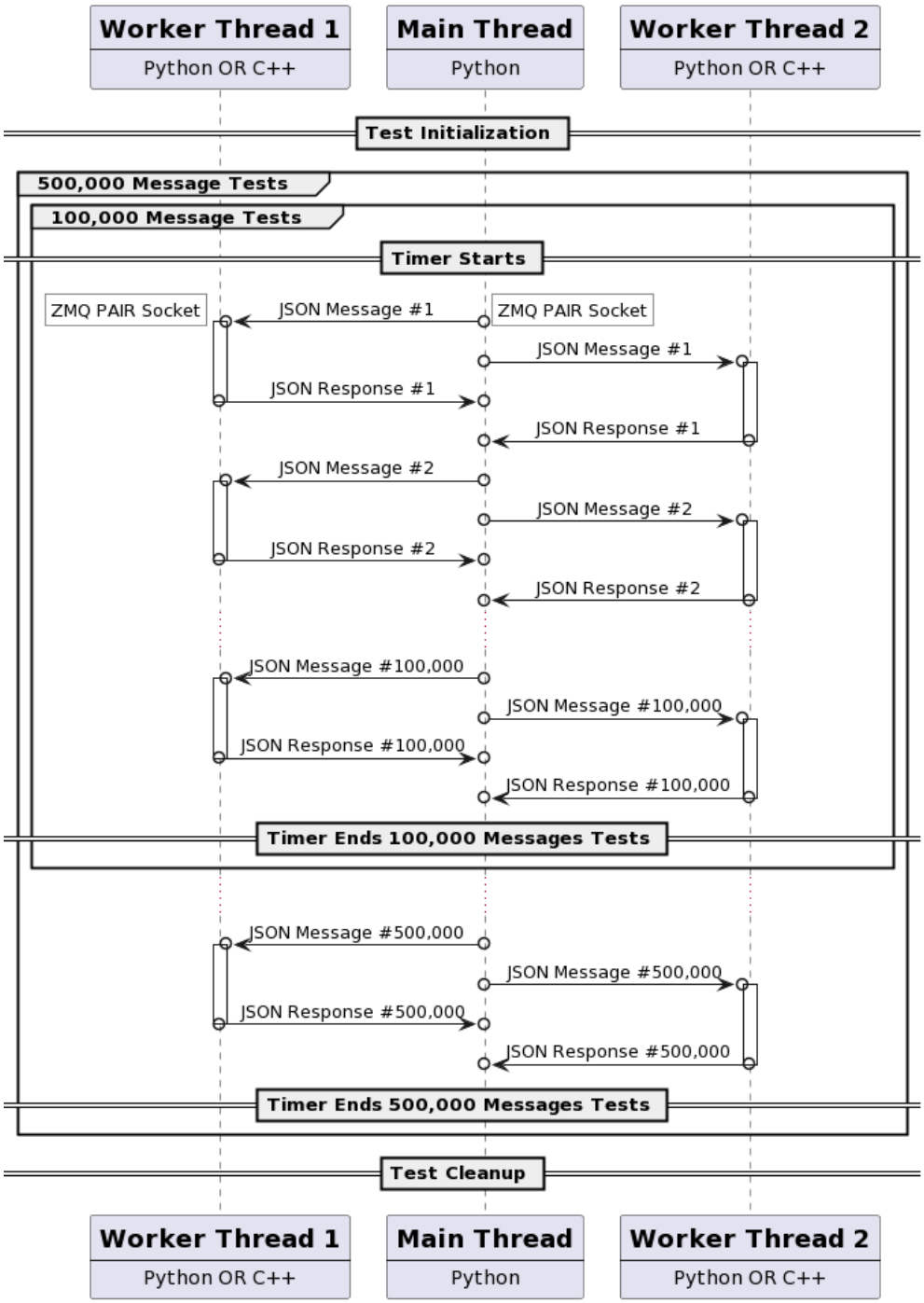

Figure IV.1: Test Message Sequence Diagram.

For each test involving C++ *pthreads*, their CPU affinity was set to ensure that each of the two *pthreads* were run simultaneously on separate CPU cores. Processor affinity is used to force the

execution of a thread on a certain CPU core. Although the *pthread* library has its own functions to set CPU core affinity, these only work on Linux, and these tests were run on MacOS. Thus, the *thread_policy_set* MacOS API function was used to ensure that each *pthread* will be executed simultaneously on different CPU cores [9].

IV.1 Messaging Tests

IV.1.1 Messaging Test 1 Results

For the first Python Control Implementation messaging test, 100,000 pairs of messages were sent and received between the main thread and each worker thread for a total of 200,000 pairs of messages, or 400,000 total messages. Besides the JSON serialization/deserialization process described previously, no other computational work was done in between messages. The results of this test can be seen below in Figure IV.2.

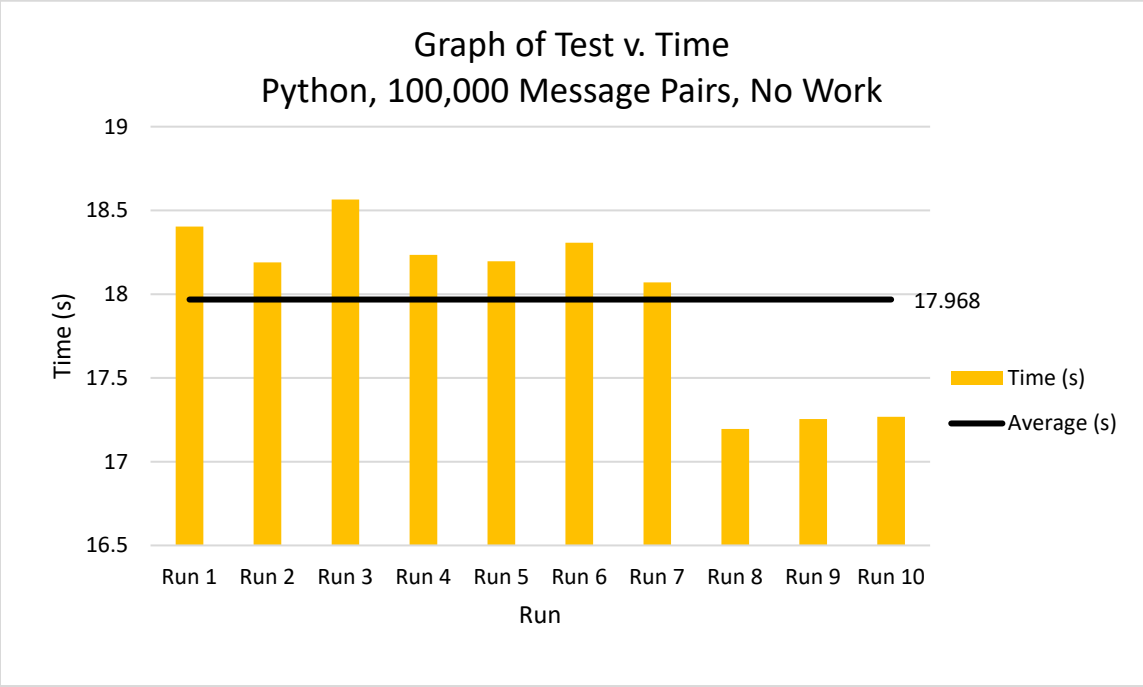

Figure IV.2: Python Control Messaging Test 1 Results.

Out of the 10 runs performed for this first Python Control Implementation test, the time per run ranged from the quickest at 17.195 seconds to the longest at 18.565 seconds with the average time per test being 17.968s. Since there was no further work to do besides sending and receiving messages, these tests were relatively quick.

For the first Python + C++ Prototype messaging test, 100,000 pairs of messages were sent and received between the main thread and each worker thread for a total of 200,000 pairs of messages, or 400,000 total messages. Besides the JSON serialization/deserialization process described previously, no other computational work was done in between messages. The results of this test can be seen below in Figure IV.3.

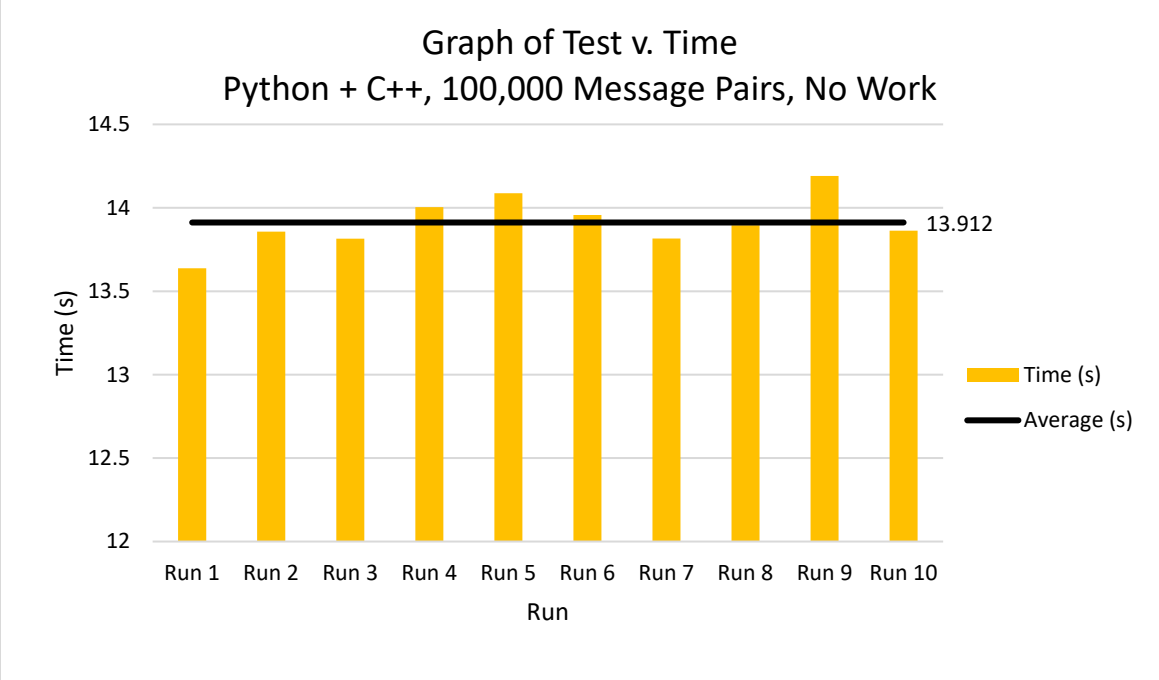

Figure IV.3: Python + C++ Prototype Messaging Test 1 Results.

Out of the 10 runs performed for this first Python + C++ Prototype messaging test, the time per run ranged from the quickest at 13.638 seconds to the longest at 14.191 seconds with the average

time per test being 13.912 seconds. As expected, this prototype was able to send and receive the same number of messages as the Python Control Implementation in less time. Using C++ worker threads led to an average speed-up of 23%.

IV.1.2 Messaging Test 2 Results

For the second Python + C++ Prototype messaging test, 500,000 pairs of messages were sent and received between the main thread and each worker thread for a total of 1,000,000 pairs of messages, or 2,000,000 total messages. This test was identical to the previous Messaging Test 1 but with more messages to simulate a higher throughput application. Besides the JSON serialization/deserialization process described previously, no other computational work was done in between messages. The results of this test can be seen below in Figure IV.4.

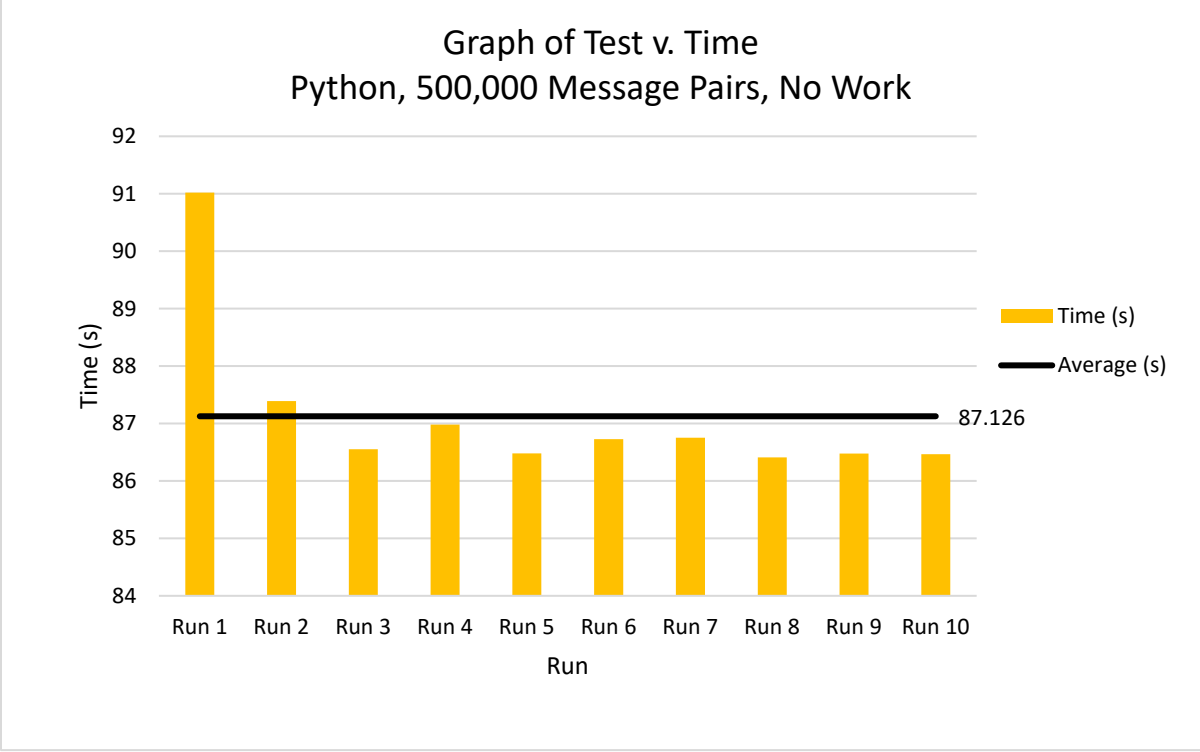

Figure IV.4: Python Control Messaging Test 2 Results.

Out of the 10 runs performed for this second Python Control Implementation test, the time per run ranged from the quickest at 86.411 seconds to the longest at 91.021 seconds with the average time per test being 87.126 seconds. Each run here ran longer than the previous Python Control Implementation test due to the significant increase in messages. When compared, this test averaged handling 400% more message pairs in about 385% more time. As seen in the results graph, this test benefited from Python's caching mechanism, which allowed each run after the first run to consistently complete on average around 5% faster. However, the initial slow run was necessary for this future speed improvement.

For the second Python + C++ Prototype messaging test, 500,000 pairs of messages were sent and received between the main thread and each worker thread for a total of 1,000,000 pairs of messages, or 2,000,000 total messages. As with the Python Control Implementation for this test, this test was identical to the previous Messaging test but with more messages to simulate a higher throughput application. Besides the JSON serialization/deserialization process described previously, no other computational work was done in between messages. The results of this test can be seen below in Figure IV.5.

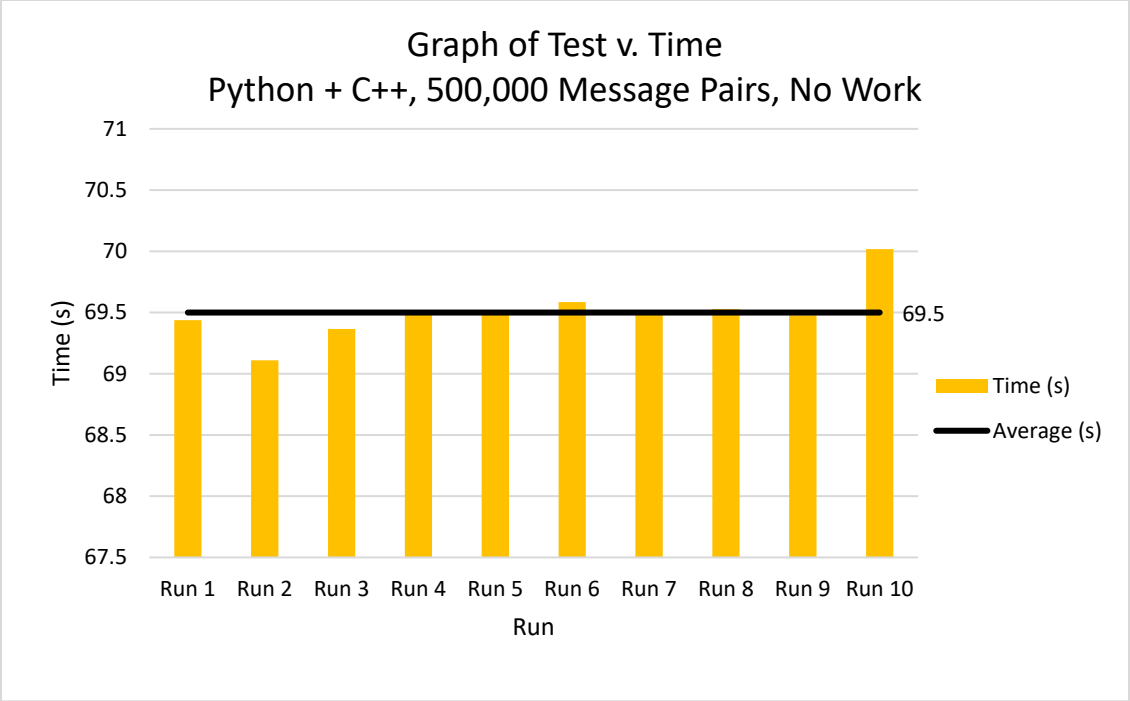

Figure IV.5: Python + C++ Prototype Messaging Test 2 Results

Out of the 10 runs performed for this second Python + C++ Prototype messaging test, the time per run ranged from the quickest at 69.11 seconds to the longest at 70.018 seconds with the average time per test being 69.5 seconds. As with the previous messaging test, this prototype was able to send and receive the same number of messages as the Python Control Implementation in less time. Using C++ worker threads led to an average speed-up of 20%. When compared to the previous Python + C++ messaging test, this test averaged handling 400% more message pairs in about 400% more time, a linear scaling.

Of the two messaging tests performed here, the Python + C++ Prototype implementation consistently performed faster than its Python Control Implementation counterpart. A summary of the results can be seen below in Table IV.3. Since no work was done in this test, this is the worst-case scenario for performance for the Python + C++ Prototype. Still, the speed-up over the Python

Control Implementation averaged 23% and 20% for each test, respectively, representing the floor for performance increases.

Test Metric	Messaging Test 1	Messaging Test 2
Number of Messages Sent to Each Thread	100,000	500,000
Python Control Average Time (s)	17.968	87.126
Python Control Variance (s)	0.224	1.768
Python Control Standard Deviation (s)	0.494	0.329
Python Control Range (s)	1.370	4.610
Python + C++ Prototype Average Time (s)	13.912	69.500
Python + C++ Prototype Variance (s)	0.022	0.045
Python + C++ Prototype Standard Deviation (s)	0.148	0.212
Python + C++ Prototype Range (s)	0.553	0.908
Python + C++ Prototype Average Speed-Up	23%	20%

Table IV.3: Messaging Test Results.

IV.2 Light Computational Load Tests

IV.2.1 Light Computational Load Test 1 Results

For the first Python Control Implementation test with light computational load, 100,000 pairs of messages were sent and received between the main thread and each thread for a total of 200,000 pairs of messages, or 400,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages to simulate a light workload that may be performed by a developer application. To simulate

computational work, a function to calculate the Fibonacci number of an input number was created. Thus, each worker thread received an integer representing the number of the current message, found the Fibonacci number corresponding to the square root of this number, and sent it back to the main thread. The results of this test can be seen below in Figure IV.6.

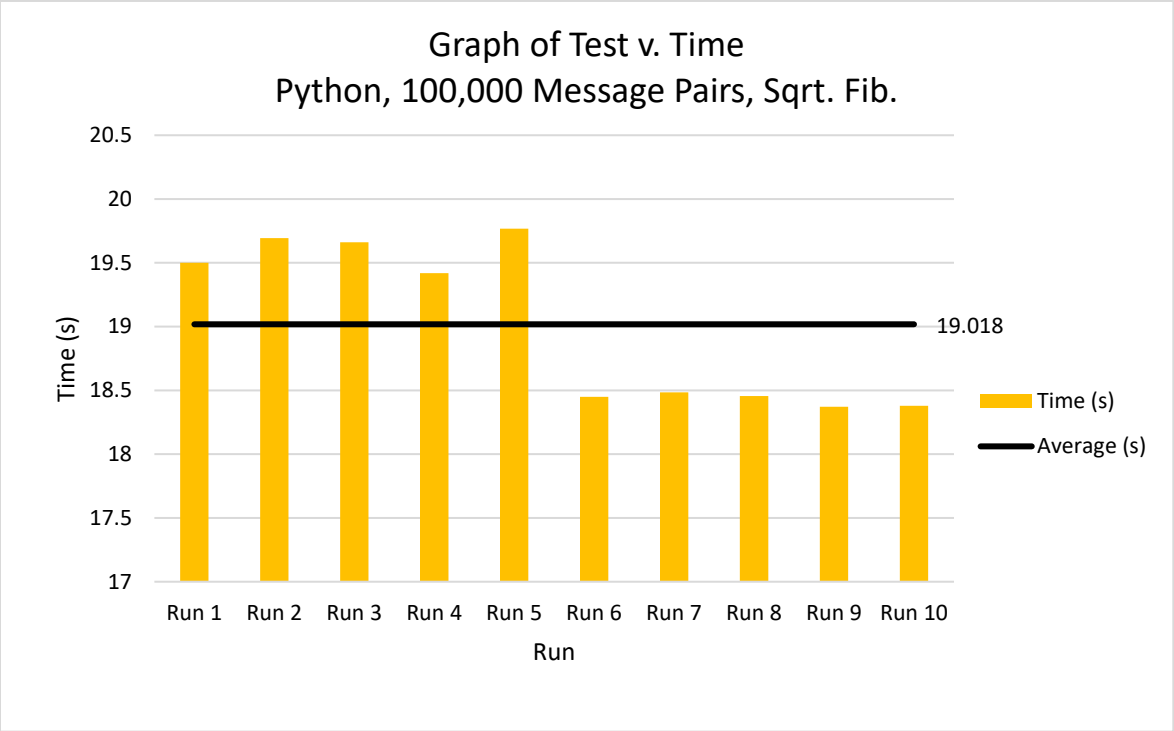

Figure IV.6: Python Control Light Computational Load Test 1 Results

Out of the 10 runs performed for this first Python Control Implementation test, the time per run ranged from the quickest at 18.372 seconds to the longest at 19.768 seconds with the average time per run at 19.018 seconds. Although the number of total message pairs remained the same, the average time per run here was 1.05 seconds longer than the first Python Control Implementation test due to the presence of the Fibonacci calculation, explained above, done by each thread in between each message. Thus, the presence of work here slowed the total execution of the program

down by 6%, a small yet noticeable slowdown of each run.

For the first Python + C++ Prototype test with light computational load, 100,000 pairs of messages were sent and received between the main thread and each thread for a total of 200,000 pairs of messages, or 400,000 total messages. In addition to the JSON serialization/deserialization process, further computational work is done by each worker thread in between messages in the same way as described above for the equivalent Python Control Implementation test. The results of this test can be seen below in Figure IV.7.

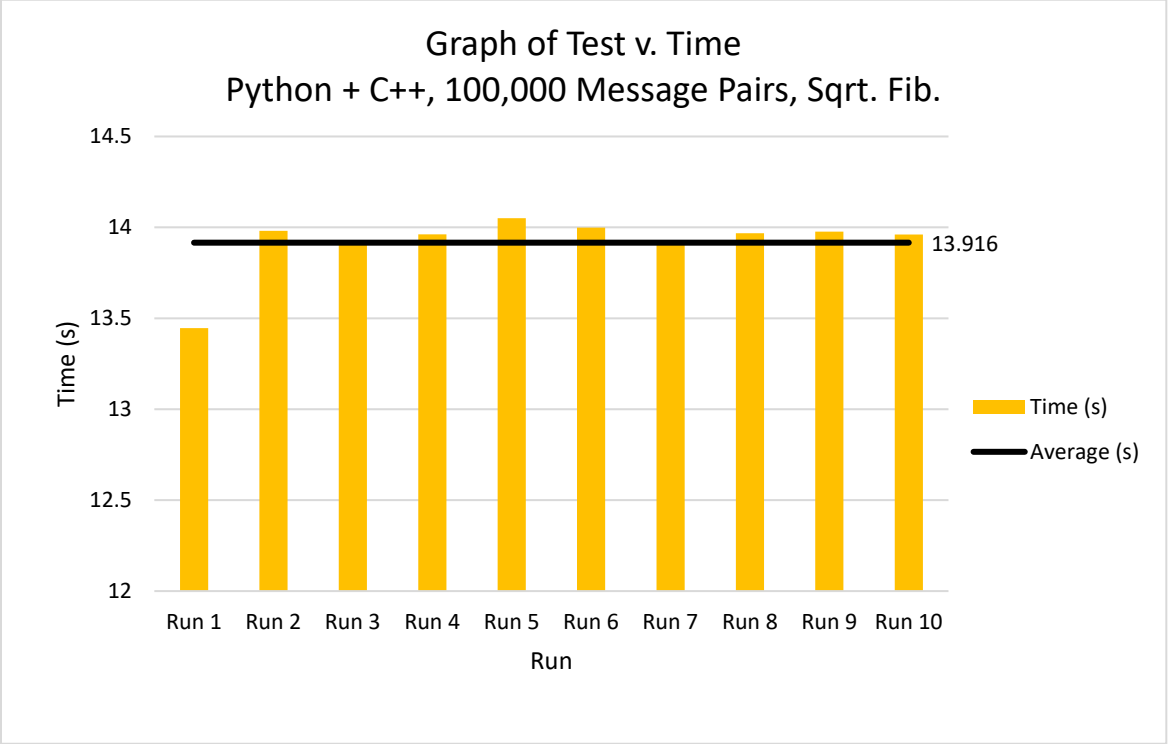

Figure IV.7: Python + C++ Prototype Light Computational Load Test 1 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 13.446 seconds to the longest at 14.051 seconds with the average time per run at 13.916 seconds. Although the Fibonacci calculation, explained above, is done by each

thread in between each message, the average time per run was unchanged from the Messaging Test 1 without the workload. Due to the speed of a compiled language such as C++ and other advantages, such as compiler optimizations and running in parallel on separate CPU processors, this workload ended up being negligible and did not slow down this test by a quantifiable amount of time, avoiding the 6% slowdown experienced by the Python Control Implementation during this test. Additionally, the Python + C++ Prototype experienced a 27% speed-up when compared to the equivalent Python Control Implementation test.

IV.2.2 Light Computational Load Test 2 Results

For the second Python Control Implementation test with light computational load, 500,000 pairs of messages were sent and received between the main thread and each thread for a total of 1,000,000 pairs of messages, or 2,000,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages via the square root Fibonacci calculation. The results of this test can be seen below in Figure IV.8.

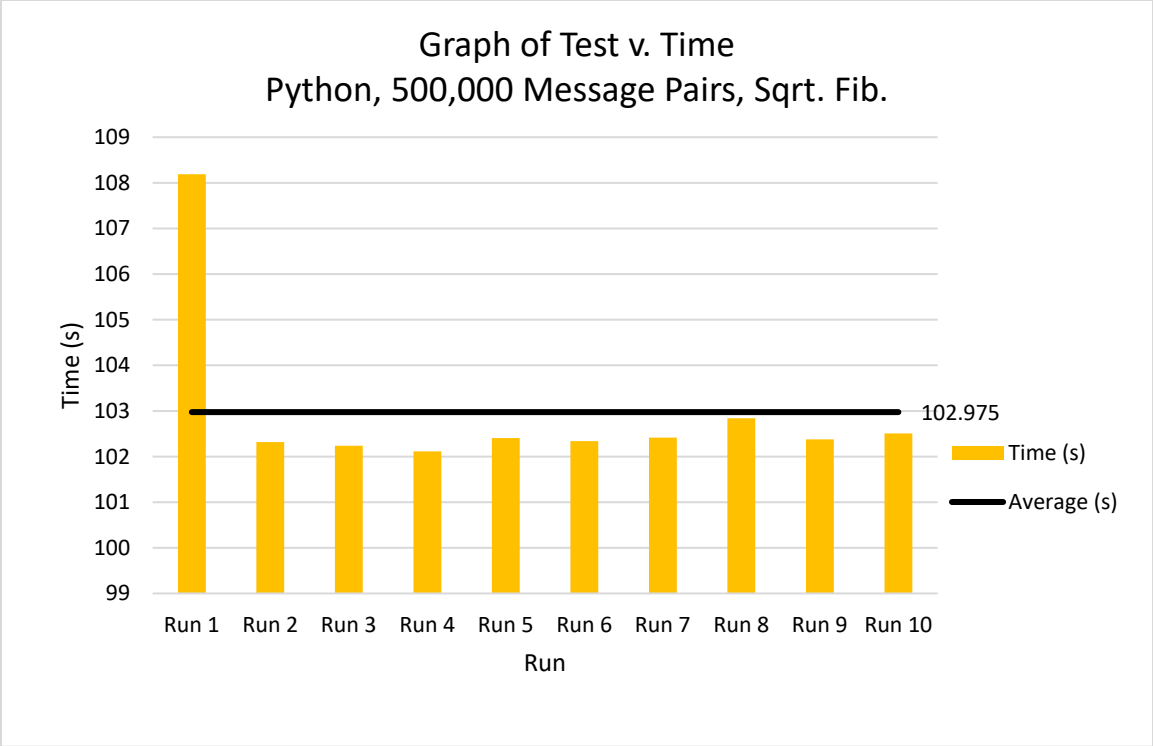

Figure IV.8: Python Control Light Computational Load Test 2 Results

Out of the 10 runs performed for this second Python Control Implementation test, the time per run ranged from the quickest at 102.115 seconds to the longest at 108.19 seconds with the average time per run at 102.975 seconds. Each run here ran an average of 18% longer than the previous no-work equivalent Python Control Implementation test, a large decrease in speed. When compared to the previous Python Control Implementation test in this section, this test averaged handling 400% more message pairs in about 441% more time; this scaling is worse than the one-to-one scaling seen in the messaging tests previously due to the presence of the computational work. Once again, the benefits of Python's caching mechanism are obvious from the initial outlier in the graph and allowed each run after the first run to consistently complete on average around 5% quicker.

For the second Python + C++ Prototype test with light computational load, 500,000 pairs of messages were sent and received between the main thread and each thread for a total of 1,000,000

pairs of messages, or 2,000,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages in the same way as above to simulate light computational load. The results of this test can be seen below in Figure IV.9.

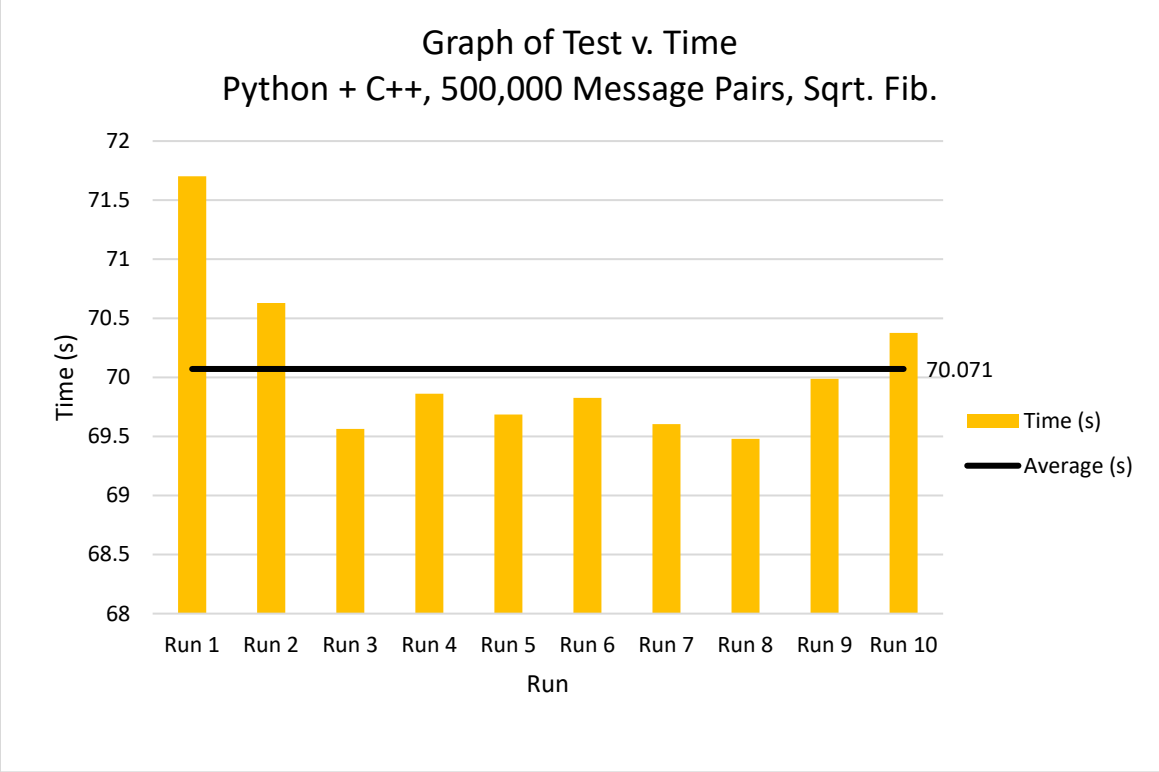

Figure IV.9: Python + C++ Prototype Light Computational Load Test 2 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 69.479 seconds to the longest at 71.702 seconds with the average time per run at 70.071 seconds. Once again, although the Fibonacci calculation was done by each thread in between each message, the average time per run is virtually unchanged from the equivalent message test without the workload. This workload continues to be negligible, and C++ manages to

avoid the 18% slowdown experienced by the Python Control Implementation during this test. The Python + C++ Prototype experienced a 32% speed-up when compared to the equivalent Python Control Implementation test.

Of the two light computational load tests performed here, the Python + C++ Prototype implementation performed faster than its Python Control Implementation counterpart in both tests. A summary of the results can be seen below in Table IV.4. Since computational load was introduced between each message, the average speed-up increased due to C++ excelling at mathematical calculations, boosting the average speed-up to 27% and 32% for each test, respectively.

Test Metric	**Light Computational Load Test 1**	**Light Computational Load Test 2**
Number of Messages Sent to Each Thread	100,000	500,000
Python Control Average Time (s)	19.018	102.975
Python Control Variance (s)	0.358	3.054
Python Control Standard Deviation (s)	0.598	1.748
Python Control Range (s)	1.396	6.075
Python + C++ Prototype Average Time (s)	13.916	70.071
Python + C++ Prototype Variance (s)	0.026	0.415
Python + C++ Prototype Standard Deviation (s)	0.162	0.664
Python + C++ Prototype Range (s)	0.605	2.223
Python + C++ Average Prototype Speed-Up	27%	32%

Table IV.4: Light Computational Load Test Results.

IV.3 Heavy Computational Load Tests

IV.3.1 Heavy Computational Load Test 1 Results

For the first Python Control Implementation test with heavy computational load, 1,000 pairs of messages were sent and received between the main thread and each thread for a total of 2,000 pairs of messages, or 4,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages to simulate a heavy workload that may be performed by a developer application. To simulate computational work, a function to calculate the Fibonacci number of an input number was created. Thus, each worker thread received an integer representing the number of the current message, found the Fibonacci number corresponding to the number, and sent it back to the main thread. The results of this test can be seen below in Figure IV.10.

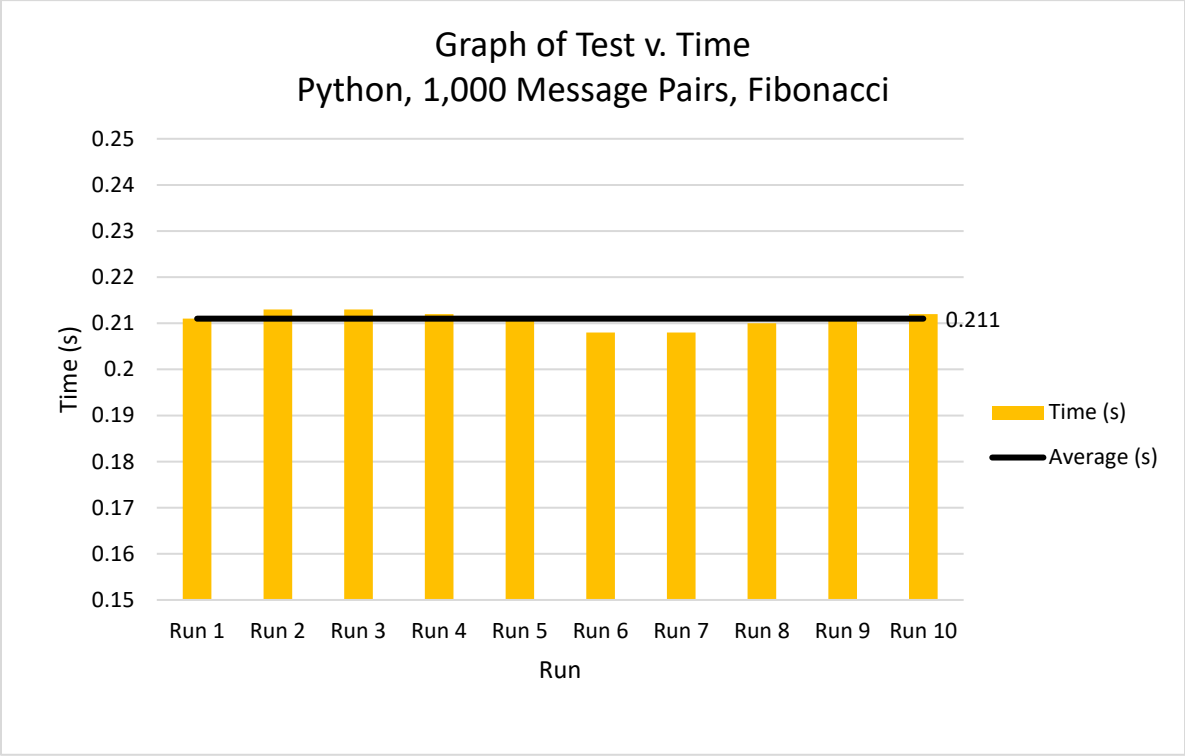

Figure IV.10: Python Control Heavy Computational Load Test 1 Results

Out of the 10 runs performed for this first Python Control Implementation test, the time per run ranged from the quickest at 0.208 seconds to the longest at 0.213 seconds with the average time per run at 0.211 seconds. The timescale for this test was in sub-seconds due to the relatively small number of messages. Although the number of total message pairs has significantly decreased compared to the previous tests, the computational work performed in between each message took much longer. In other words, the computational workload influenced test runtime far more than communication due to the increased workload. Unlike the previous Light Computational Load Test 1, where the maximum Fibonacci number to be calculated was 316, here the maximum Fibonacci number to be calculated was 1,000 due to the removal of the square root operator.

For the first Python + C++ Prototype test with heavy computational load, 1,000 pairs of messages were sent and received between the main thread and each thread for a total of 2,000 pairs of messages, or 4,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages in the same way as described above for the equivalent Python Control Implementation test. The results of this test can be seen below in Figure IV.11.

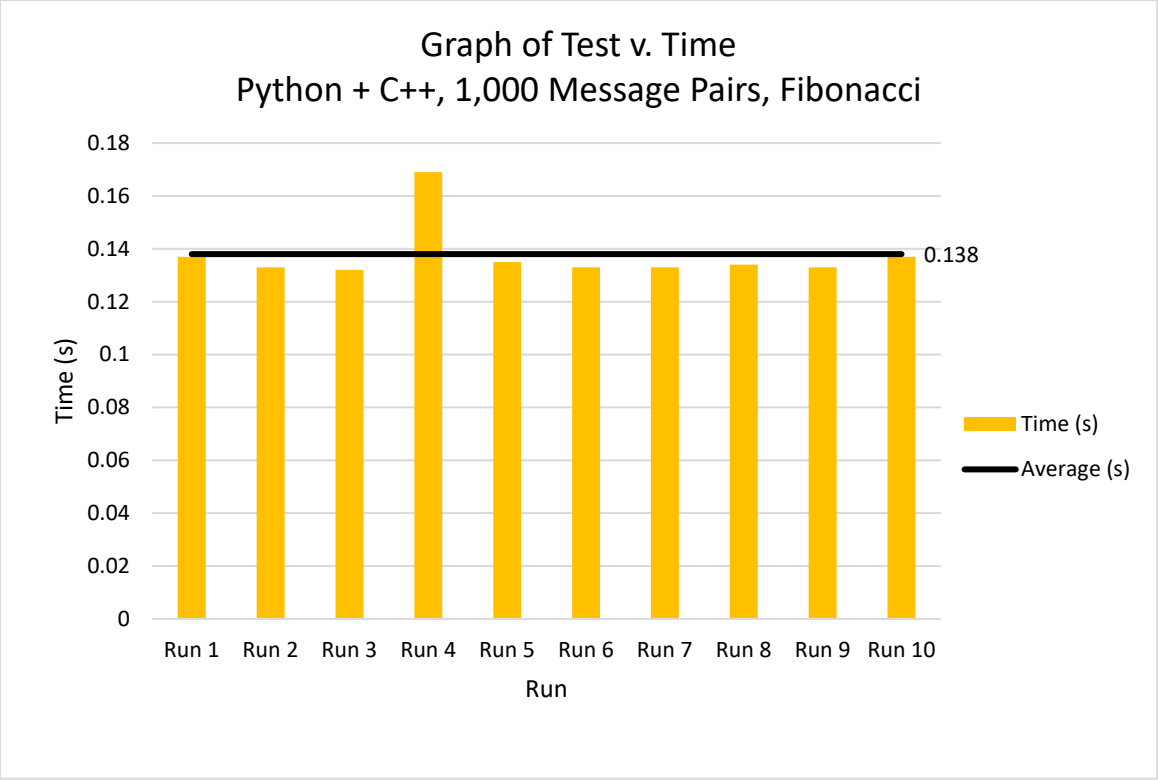

Figure IV.11: Python + C++ Prototype Heavy Computational Load Test 1 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 0.132 seconds to the longest at 0.169 seconds with the average time per run at 0.138 seconds. The timescale for this test was in sub-seconds due to the relatively small number of messages. As stated previously, the maximum Fibonacci number that was calculated here is 1,000, compared to 316 in the previous Light Computational Load Test 1. Here, the Python + C++ Prototype already experienced a 35% speed-up when compared to the equivalent Python Control Implementation test.

IV.3.2 Heavy Computational Load Test 2 Results

For the second Python Control Implementation test with heavy computational load, 5,000 pairs

of messages were sent and received between the main thread and each thread for a total of 10,000 pairs of messages, or 20,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages via the Fibonacci calculation. The results of this test can be seen below in Figure IV.12.

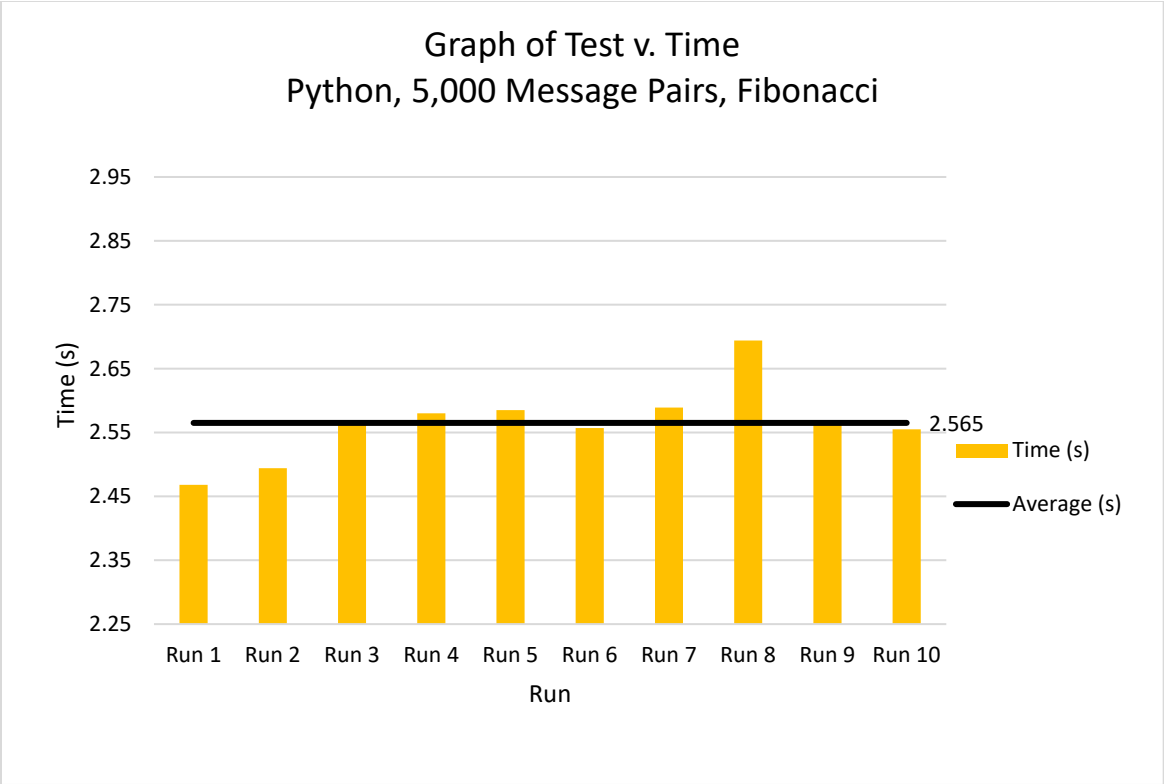

Figure IV.12: Python Control Heavy Computational Load Test 2 Results

Out of the 10 runs performed for this second Python Control Implementation test, the time per run ranged from the quickest at 2.468 seconds to the longest at 2.694 seconds with the average time per run at 2.565 seconds. Now that the number of messages has increased, the maximum Fibonacci number that was calculated here is 5,000 compared to 707 from the Light Computational Load Test 2. This work, which was performed in between each message, took up a significant

portion of the total time of each run.

For the second Python + C++ Prototype test with heavy computational load, 5,000 pairs of messages were sent and received between the main thread and each thread for a total of 10,000 pairs of messages, or 20,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages in the same way as above to simulate heavy computational load. The results of this test can be seen below in Figure IV.13.

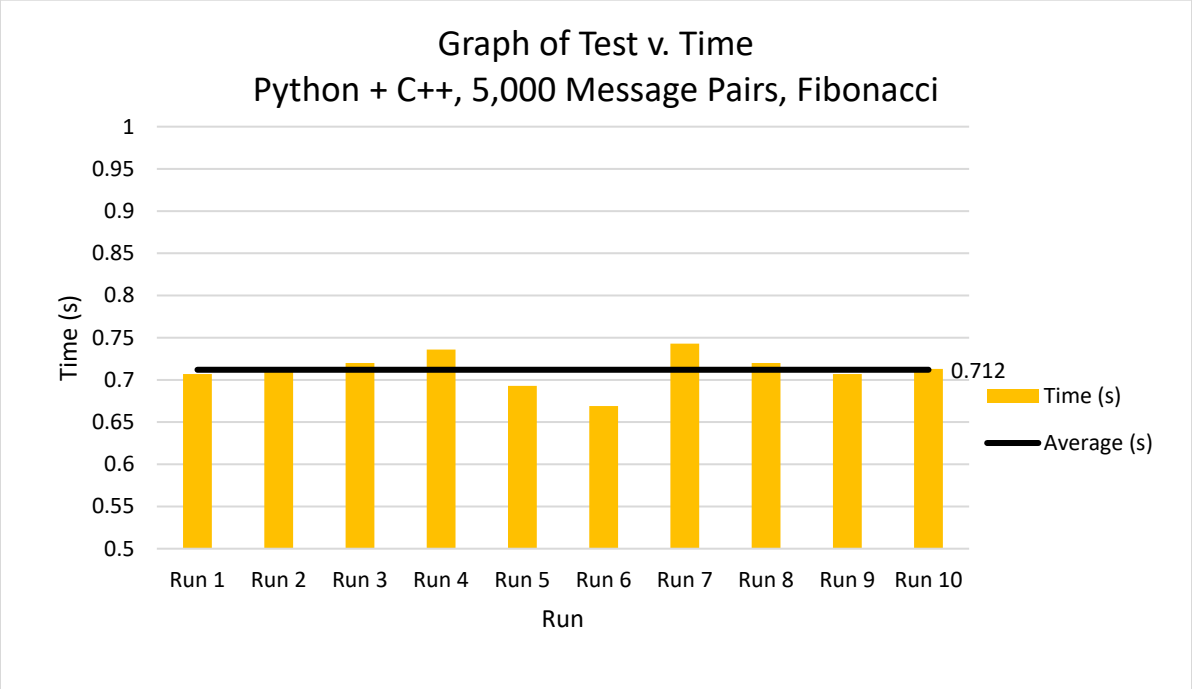

Figure IV.13: Python + C++ Prototype Heavy Computational Load Test 2 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 0.669 seconds to the longest at 0.743 seconds with the average time per run at 0.712 seconds. The timescale for this test was in sub-seconds due to the relatively small number of messages. Although a large Fibonacci calculation was done by each thread in between

each message, up to the 5,000th number, the runtime was still very quick. Here, C++'s computational advantage is obvious. The Python + C++ Prototype experienced a 72% speed-up when compared to the equivalent Python Control Implementation test.

Of the two heavy computational load tests performed here, the Python + C++ Prototype implementation performed faster than its Python Control Implementation counterpart in both tests by a significant amount. A summary of the results can be seen below in Table IV.5. Since heavy computational load was introduced between each message, the average speed-up significantly increased due to C++ excelling at mathematical computation, boosting the average speed-up to 35% and 72% for each test, respectively.

Test Metric	**Heavy Computational Load Test 1**	**Heavy Computational Load Test 2**
Number of Messages Sent to Each Thread	1,000	5,000
Python Control Average Time (s)	0.211	2.565
Python Control Variance (s)	0.000	0.003
Python Control Standard Deviation (s)	0.002	0.057
Python Control Range (s)	0.005	0.226
Python + C++ Prototype Average Time (s)	0.138	0.712
Python + C++ Prototype Variance (s)	0.000	0.000
Python + C++ Prototype Standard Deviation (s)	0.011	0.020
Python + C++ Prototype Range (s)	0.037	0.074
Python + C++ Average Prototype Speed-Up	35%	72%

Table IV.5: Heavy Computational Load Test Results.

IV.4 Function Calls Tests

IV.4.1 Function Calls Test 1 Results

For the first Python Control Implementation function calls test, 10 pairs of messages were sent and received between the main thread and each thread for a total of 20 pairs of messages, or 40 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages to simulate a function-call-heavy workload that may be performed by a developer application. To simulate this type of work, a function to recursively calculate the Fibonacci number of an input number was created. Thus, each worker thread received an integer representing the number of the current message, found the Fibonacci number corresponding to the number recursively, and sent it back to the main thread. The results of this test can be seen below in Figure IV.14.

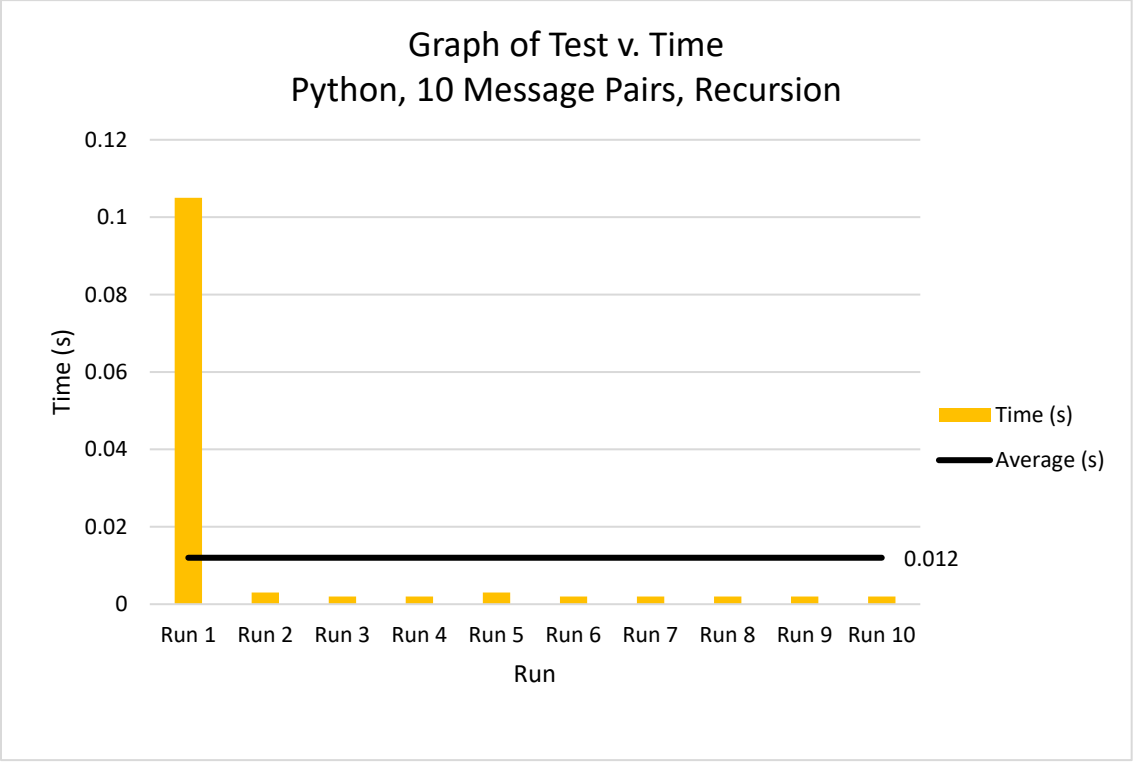

Figure IV.14: Python Control Function Calls Test 1 Results

Out of the 10 runs performed for this first Python Control Implementation test, the time per run ranged from the quickest at 0.002 seconds to the longest at 0.105 seconds with the average time per run at 0.012 seconds. The timescale for this test was in sub-seconds due to the relatively small number of messages. The initial outlier can be explained with Python's caching mechanism; thus, the range of run times was large here due to caching having a significant impact on subsequent run times. A 98% speedup was observed after the first run due to caching. Although 10 message pairs, and thus calculating the Fibonacci number of 10, may seem small, Python's speed slowed exponentially when dealing with recursive function calls. This is not surprising, since the time complexity of a recursive Fibonacci solution is $O(2^n)$, otherwise known as exponential. Thus, this light test served as a safe baseline.

For the first Python + C++ Prototype function calls test, 10 pairs of messages were sent and received between the main thread and each thread for a total of 20 pairs of messages, or 40 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages in the same way as described above for the equivalent Python Control Implementation test. The results of this test can be seen below in Figure IV.15.

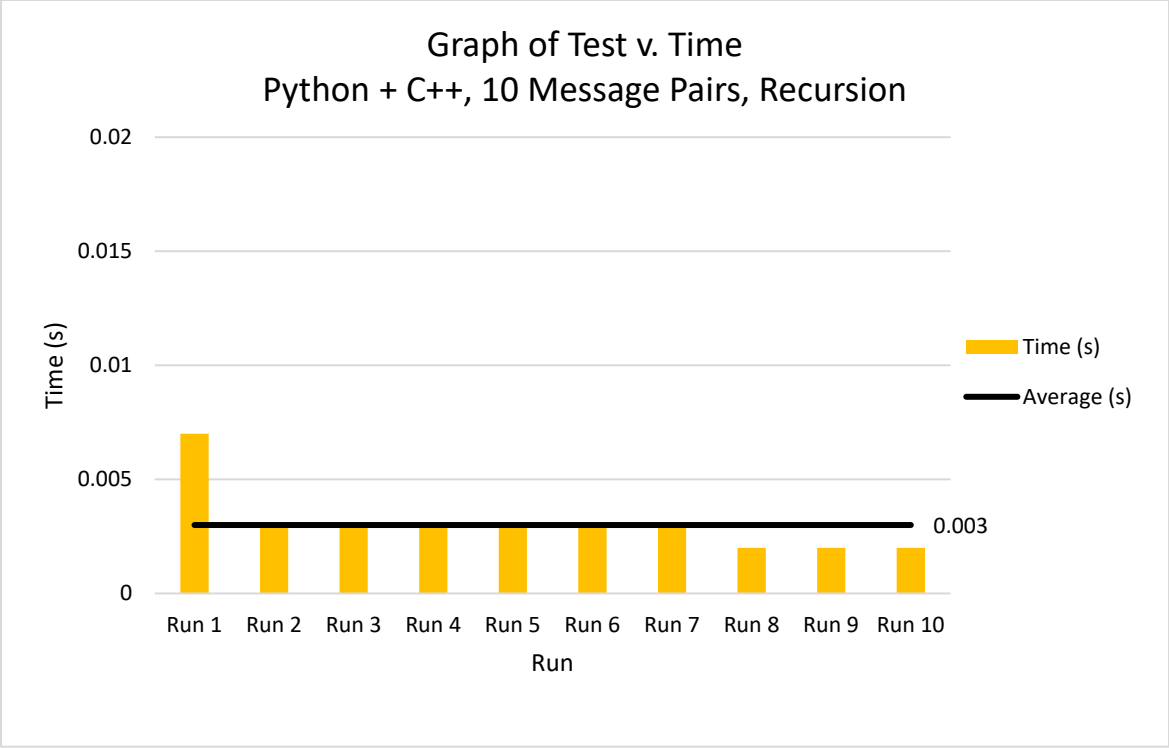

Figure IV.15: Python + C++ Prototype Function Calls Test 1 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 0.002 seconds to the longest at 0.007 seconds with the average time per run at 0.003 seconds. The timescale for this test was in sub-seconds due to the relatively small number of messages. Since C++ handled recursive function calls better, and the computation was relatively light, this implementation was able to run very fast. Here, the Python + C++ Prototype already ran 75% faster when compared to the equivalent Python Control Implementation test above.

IV.4.2 Function Calls Test 2 Results

For the second Python Control Implementation function calls test, 40 pairs of messages were sent and received between the main thread and each thread for a total of 80 pairs of messages, or

160 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages via the recursive Fibonacci calculation. The results of this test can be seen below in Figure IV.16.

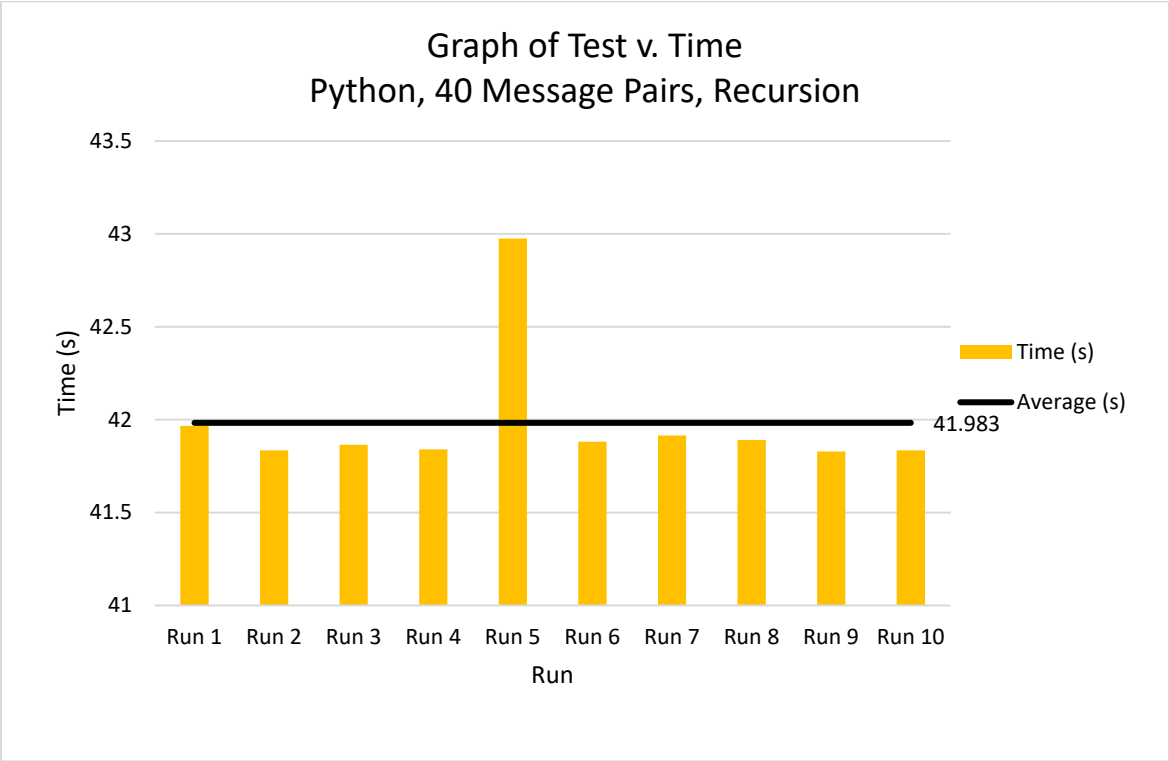

Figure IV.16: Python Control Function Calls Test 2 Results

Out of the 10 runs performed for this second Python Control Implementation test, the time per run ranged from the quickest at 41.829 seconds to the longest at 42.975 seconds with the average time per run being 41.983 seconds. As stated previously, Python's runtime rises exponentially when increasing the amount of function calls. Despite only performing Fibonacci calculations for a number that is 300% larger than the last function call test, the average runtime was a staggering 349,758% longer. It is sufficient to say that, of all the tests presented in this research, Python performed the worst at function calls.

For the second Python + C++ Prototype function calls test, 40 pairs of messages were sent and received between the main thread and each thread for a total of 80 pairs of messages, or 160 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages in the same way as above to simulate heavy function-call load. The results of this test can be seen below in Figure IV.17.

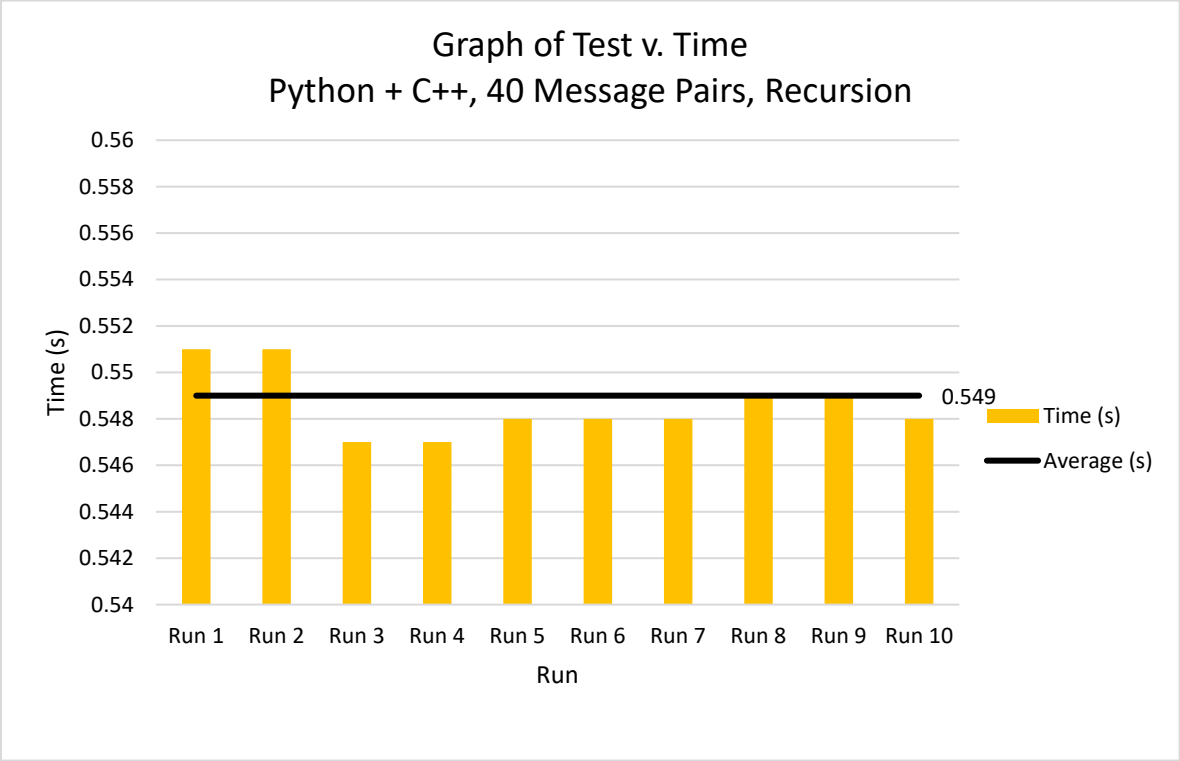

Figure IV.17: Python + C++ Prototype Function Calls Test 2 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 0.547 seconds to the longest at 0.551 seconds with the average time per run being 0.549 seconds. The timescale for this test was in sub-seconds due to the relatively small number of messages. While performing Fibonacci calculations for a number that was 300% larger than the last function call test, the average runtime was 18,200% longer. While this seems like a

large percentage, the original runtimes for the previous Function Calls Tests for this implementation were very small. Additionally, this was still 19 times faster scaling than Python when going from a small amount to a large amount of function calls. C++'s function-calling advantage was abundantly clear with a 99% speed-up when compared to the Python Control Implementation counterpart.

Of the two function calls tests performed here, the Python + C++ Prototype implementation performed staggeringly faster than its Python Control Implementation counterpart in both tests. A summary of the results can be seen below in Table IV.6. Due to the large number of function calls, C++ was able to achieve an average speed-up of 75% and 99% for each test, respectively.

Test Metric	**Function Calls Test 1**	**Function Calls Test 2**
Number of Messages Sent to Each Thread	10	40
Python Control Average Time (s)	0.012	41.983
Python Control Variance (s)	0.001	0.111
Python Control Standard Deviation (s)	0.031	0.333
Python Control Range (s)	0.103	1.146
Python + C++ Prototype Average Time (s)	0.003	0.549
Python + C++ Prototype Variance (s)	0.000	0.000
Python + C++ Prototype Standard Deviation (s)	0.001	0.001
Python + C++ Prototype Range (s)	0.005	0.004
Python + C++ Prototype Average Speed-Up	75%	99%

Table IV.6: Function Calls Test Results.

IV.5 Memory Allocation Tests

IV.5.1 Memory Allocation Test 1 Results

For the first Python Control Implementation memory allocation test, 100,000 pairs of messages were sent and received between the main thread and each thread for a total of 200,000 pairs of messages, or 400,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages to simulate a memory-heavy workload that may be performed by a developer application. To simulate this type of work, 100 data structures, each 4 KB in size, were created and then deleted by each thread in between each message. In Python, this operation takes the form of allocating lists of this size within a temporary scope that ceases to exist by the time the next message is sent. The results of this test can be seen below in Figure IV.18.

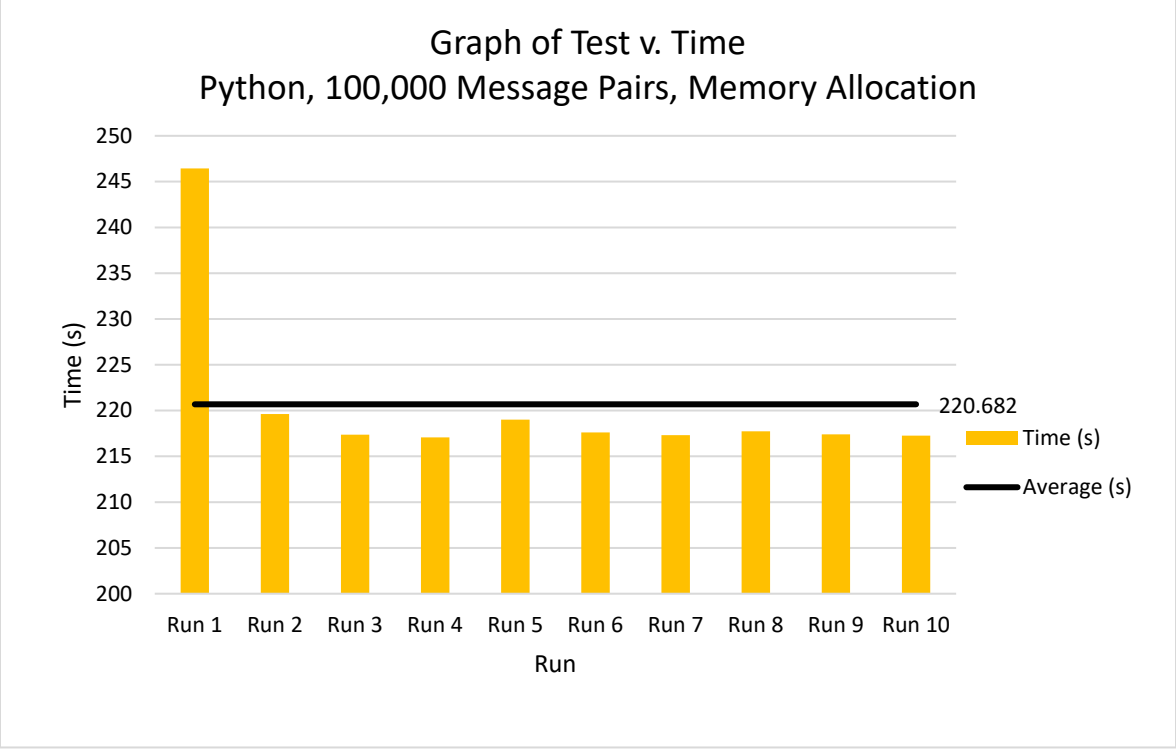

Figure IV.18: Python Control Memory Allocation Test 1 Results

Out of the 10 runs performed for this first Python Control Implementation test, the time per run ranged from the quickest at 217.077 seconds to the longest at 246.436 seconds with the average time per run at 220.682 seconds. Once again, the effects of caching were observed in the first outlier on the graph; subsequent runs gained a 12% speedup after the first run. Python was also slow at memory operations; this test is 1,128% slower than the equivalent Python Control Implementation results from Messaging Test 1.

For the first Python + C++ Prototype memory allocation test, 100,000 pairs of messages were sent and received between the main thread and each thread for a total of 200,000 pairs of messages, or 400,000 total messages. In addition to the JSON serialization/deserialization process, further memory-oriented computational work was done by each worker thread in between messages in the same way as described above for the equivalent Python Control Implementation test. The results of this test can be seen below in Figure IV.19.

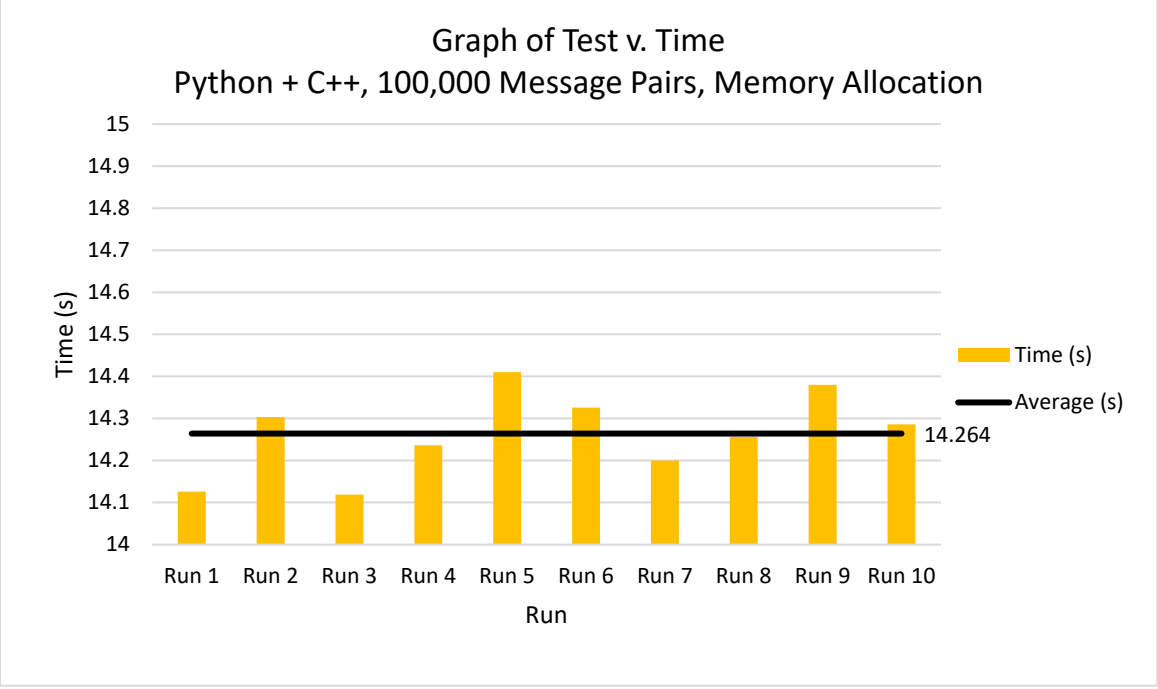

Figure IV.19: Python + C++ Prototype Memory Allocation Test 1 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 14.119 seconds to the longest at 14.41 seconds with the average time per run at 14.264 seconds. C++'s memory operations were fast due to its low-level nature, so this test was only 3% slower than the equivalent no-work Python + C++ Prototype test in Messaging Test 1 with the same number of messages. This implementation also ran 94% faster when compared to the equivalent Python Control Implementation test above.

IV.5.2 Memory Allocation Test 2 Results

For the second Python Control Implementation function calls test, 500,000 pairs of messages were sent and received between the main thread and each thread for a total of 1,000,000 pairs of messages, or 2,000,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread in between messages via the same memory operations as the previous test. The results of this test can be seen below in Figure IV.20.

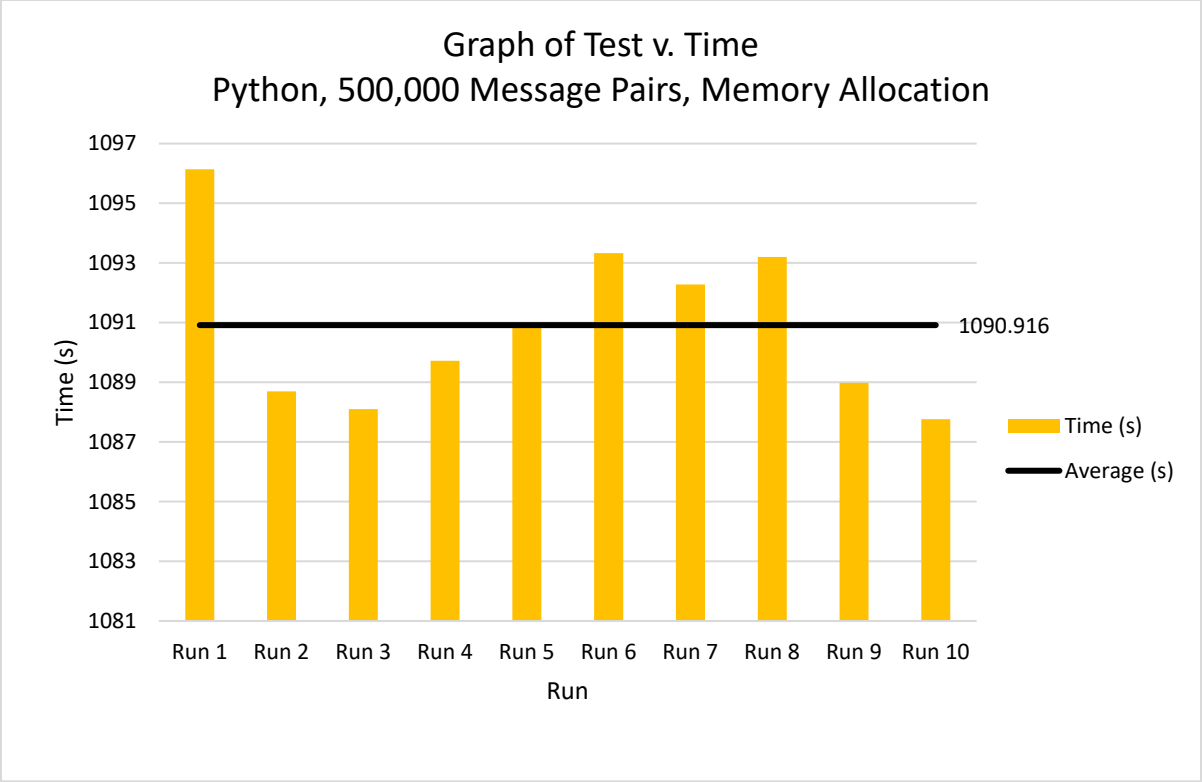

Figure IV.20: Python Control Memory Allocation Test 2 Results

Out of the 10 runs performed for this second Python Control Implementation test, the time per run ranged from the quickest at 1087.762 seconds to the longest at 1096.141 seconds with the average time per run at 1090.916 seconds. While caching seemed to have an effect on the overall test runtime early on, time per run rises later in the test and drops the overall caching speed advantage to only 1%. When compared to the equivalent 500,000 message no-work Python Control Implementation test, the average runtime here was 1,152% longer.

For the second Python + C++ Prototype memory allocation test, 500,000 pairs of messages were sent and received between the main thread and each worker thread for a total of 1,000,000 pairs of messages, or 2,000,000 total messages. In addition to the JSON serialization/deserialization process, further computational work was done by each worker thread

in between messages in the same way as above to simulate heavy memory operations load. The results of this test can be seen below in Figure IV.21.

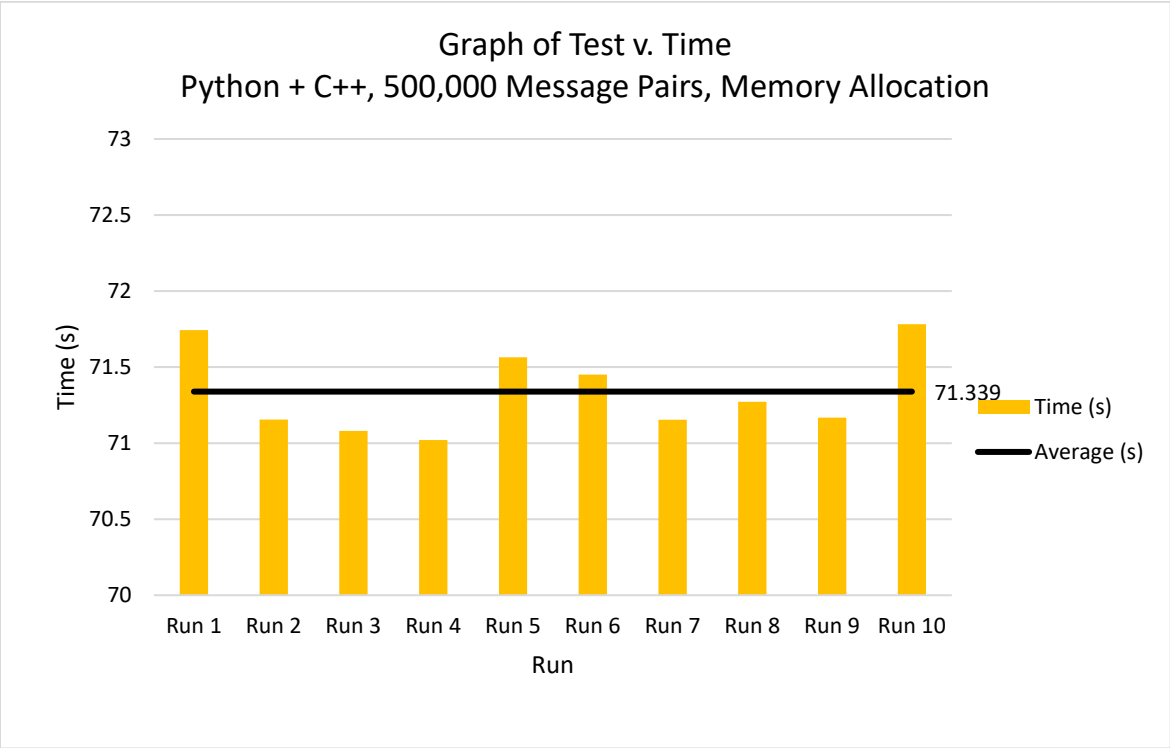

Figure IV.21: Python + C++ Prototype Memory Allocation Test 2 Results

Out of the 10 runs performed for this second Python + C++ Prototype test, the time per run ranged from the quickest at 71.021 seconds to the longest at 71.785 seconds with the average time per run at 71.339 seconds. Despite performing these memory operations between each message, this test only experienced a 3% increase in average runtime when compared to the equivalent Python + C++ Prototype results from Messaging Test 2. Similar to the Memory Allocation Test 1, the speedup here experienced by the Python + C++ Prototype was 93% when compared to its Python Control Implementation counterpart.

Of the two memory allocation tests performed here, the Python + C++ Prototype

implementation performed extremely fast in both tests, much faster than its Python Control Implementation counterpart and almost as fast as the same messaging tests without any work. A summary of the results can be seen below in Table IV.7. C++ was able to achieve an average speed-up of 94% and 93% for each test, respectively.

Test Metric	**Memory Allocation Test 1**	**Memory Allocation Test 2**
Number of Messages Sent to Each Thread	100,000	500,000
Python Control Average Time (s)	220.682	1090.916
Python Control Variance (s)	74.318	6.818
Python Control Standard Deviation (s)	8.621	2.611
Python Control Range (s)	29.359	8.379
Python + C++ Prototype Average Time (s)	14.264	71.339
Python + C++ Prototype Variance (s)	0.009	0.069
Python + C++ Prototype Standard Deviation (s)	0.093	0.263
Python + C++ Prototype Range (s)	0.291	0.761
Python + C++ Prototype Average Speed-Up	94%	93%

Table IV.7: Memory Allocation Test Results.

CHAPTER V

Conclusions and Future Work

This chapter will discuss the conclusions drawn from both the discussion of the implementation of the tests in **Chapter III** and the results of each test shown in **Chapter IV**. Once a final conclusion has been drawn on which design is the best and most beneficial for RIAPS, future work can be discussed as it relates to the details and intricacies involved with moving this proof-of-concept prototype design to full integration with a large framework such as RIAPS.

V.1 Conclusions

Python's drawbacks cause it to be a slow language, especially when compared to traditional, compiled languages. Due to the nature of Python and the existence of the GIL, no true parallel multithreading can be achieved, severely limiting its ability to fully use the computational resources available on modern computers and embedded devices. Since RIAPS is entirely programmed in Python, this distributed framework will suffer from the same performance issues.

Many different approaches were considered and attempted when looking for a solution to these performance issues in the Python-based RIAPS. Cython seemed like an efficient and easily integrated middle layer that would help facilitate the transition from high-level Python to lower-level parallelized *C* code. While integration with Python was easy, the interactions between Python and low-level compiled code were not necessarily easier via Cython. The added complexities of a middle layer, along with the fact that facilitating the transfer of Cython data structures to compiled code was slow due to the GIL, ruled Cython out as an effective solution.

Direct Python–C++ integration offers a fast and achievable solution for parallelism to optimize RIAPS. Compiling C++ functions to a shared library that can be accessed by Python gives Python scripts the ability to launch *pthreads*. Serializing Python data structures to a language-agnostic format, such as JSON, allows threads to execute free of the GIL. Prototype implementations and tests show the benefits and promise of this approach. Overall, the Python + C++ Prototype implementation performed better in every test compared to the Python Control Implementation. The worst-case scenario for a speed-up, which was shown with the communication-only messaging test, still yielded a 20%-23% average runtime reduction. When introducing load, depending on the type of load, speed increases of up to 99% can be observed. Thus, this direct Python–C++ integration improves RIAPS performance in every situation while not introducing any unmanageable drawbacks.

V.2 Future Work

The majority of future work to be done involves integrating this C++ prototype implementation into the RIAPS framework. Due to technical limitations, such as the fact that RIAPS can only run on Linux due to library dependencies, development directly onto RIAPS would have been time-consuming to set up properly in the timeframe and scope of this research. The Python + C++ Prototype implementation serves as a solid proof-of-concept for future work to implement true parallelization into RIAPS. All developer applications hosted by RIAPS would experience an increase in performance. Many developer applications, such as computationally intensive, high-function call, or memory-intensive ones, would benefit greatly from this truly parallel capability. RIAPS's component model, which currently serves as a convenient encapsulation of the developer

application, will be the target of the parallelization. Developer applications written in C++ can be compiled into shared libraries and then launched as a component within RIAPS.

References

[1] Ghosh, Purboday & Eisele, Scott & Dubey, Abhishek & Metelko, Mary & Madari, István & Völgyesi, Péter & Karsai, Gabor. (2020). Designing a Decentralized Fault-Tolerant Software Framework for Smart Grids and its Applications. Journal of Systems Architecture. 109. 101759. 10.1016/j.sysarc.2020.101759.

[2] Hasecke, Jan Ulrich, "python: a programming language changes the world," Available: https://brochure.getpython.info/media/releases/prerelases/psf-python-brochure-vol-1-final-content-preview. [Accessed Nov. 10, 2023].

[3] Beazley, David, "Understanding the Python GIL," Pycon 2010, Atlanta, GA, USA, 2010. Available: https://speakerdeck.com/dabeaz/understanding-the-python-gil?slide=10. [Accessed Nov. 10, 2023].

[4] The ZeroMQ Authors, "ZeroMQ: An open-source universal messaging library," Available: https://zeromq.org/. [Accessed Nov. 10, 2023].

[5] Behnel, Stefan & Bradshaw, Robert & Woods, David & Valo, Matúš & Dalcín, Lisandro, "Cython: C-Extensions for Python," Available: https://cython.org. [Accessed Nov. 10, 2023].

[6] Pokorny, David Brooks, "Cython CPython Ext Module Workflow," Available: https://commons.wikimedia.org/wiki/File:Cython_CPython_Ext_Module_Workflow.png. [Accessed Nov. 10, 2023].

[7] Roy, Abinash & Xu, Jingye & Chowdhury, Masud. (2009). Multi-core processors: A new way forward and challenges. 454 - 457. 10.1109/ICM.2008.5393510.

[8] M. Florisson, "Multi-Level Debugging for Cython," 14th Twente Student Conference on IT, vol. 14, no. 1, Jan. 2011.

[9] Apple, inc. “Apple Developer Function thread_policy_set,” Available: https://developer.apple.com/documentation/kernel/1418892-thread_policy_set. [Accessed Nov. 10, 2023].