Marios Yiannakou Student ID: 10333274

Benchmarking Programming Languages

BSc Computer Science, University of Manchester
Supervised by Bijan Parsia
Academic Year 2021/2022

Abstract

This paper introduces a modern redesign and repurpose of the "Computer Language Shootout" project. The original project uses collaborative efforts to solve a set of algorithms in a plethora of languages.

This project is a first attempt on recreating the shootout project with a primary purpose of being used as a teaching tool for the introduction of data structures and optimised algorithm implementation to the novice programmer regardless of the hardware available to the participants. Loading up a quick environment with pre-installed compilers and an IDE has proved to greatly speed up the learning process by saving on set up costs and providing a ready to code environment to focus on learning a language.

Table of Contents

1 History and Introduction

The "Computer Language Benchmarks Game", previously known as "The Great Computer Language Shootout", is an open-source project that is meant to compare how different known algorithms can be implemented in different languages. The solutions are measured using a single framework to capture usage parameters like elapsed time, CPU and thread utilisation, and memory loaded and used to produce a ranked list. It was originally introduced by a programmer named Doug Bagley with the first trace of an online activity January 2001, to compare the major scripting languages and, later, with compiled languages as an experiment and teaching tool "…I'm doing it so that I can learn about new languages, compare them in various (possibly meaningless) ways, and most importantly, have some fun." (Bagley, 2001) with an update on Spring of 2004 stating that the project had been frozen during the Fall of 2001 with no clear motivation of being restarted, and even migrated to a new endpoint as an archive. During October 2004, the endpoint completely went offline. Since then, the project was resurrected by independent programmers which created their own spin on the benchmarking requirements and methods, while keeping an aspect of the original repository alive.

However, this paper refers to a particular version of the shootout by Brent Fulgham in the same year the original project was retired and is actively maintained by Isaac Gouy since 2008. It is noted that the project has been maintained since 2008 but the official GitLab repository's earliest commit originates in 2018 due to what is described as "the Debian Alioth hosting service EOL" (The Computer Language 22.03 Benchmark's Game , 2022).

The project seemed like an intriguing idea, as an upcoming software engineer already familiar with a few languages, it would be interesting to see how my learning capabilities have evolved and if I would now be capable to teach myself different languages and if so, at what level. This has the potential of a being powerful teaching tool or profiler for programmers of any level to either tinker or learn seeing as how people begin coding or write their first line of code at a younger age (StackOverflow).

A teacher could provide their students with a single repository that can be forked, modified, executed, and compared, provide a tidy environment for an individual programmer trying to learn a new language, or compare how different systems can run a set of algorithms in different languages. It can provide insight for freelancers and companies to manage costs between running dedicated hardware in or take advantage of the convenience and flexibility of cloud computing.

2 Design

This section focuses on the motivation for the file structure hierarchy and the design choices made while re-creating the Computer Language Shootout in a novice-oriented manner. It explains the target audience, what measures have been put into place to ensure a controlled and fair environment, and how the project can be set up from scratch.

2.1 Motivation

During the first attempt to download and run the original Computer Language Shootout project, it was clear that the provided instructions intended to be a "what to do after you have set up the project" rather than "how to set up the project". The source code and its many configuration files were scoured through for three days, hoping to find a comment or an instruction file stating specific steps how to set up and run it, or even the required system pre-requisites. After a highly frustrating process, a stack overflow question[1], and a few cups of coffee, the original shootout project was up and ready to run.

This project lacked documentation, or at least accessibility to it, and automation. A lot of the heavy lifting was left up to user error. The program should be taking advantage of the users' environment to

[1] https://stackoverflow.com/questions/69700644/cant-install-python-2-gtop-module-in-virtual-environment

automate the process whilst allowing the ability to customise the configuration. Such an approach has a clear advantage in troubleshooting. It is more likely that failures in a system with documented requirements are more straightforward to debug than leaving everything to human error. Common failures mean that an FAQ or community forum is highly likely to emerge where troubleshooting can be "outsourced", alleviating the developer from solving random bugs in different peoples' systems. For example, I created a Discord server[2] that people interested in this project can join and discuss their opinions and optimisation methods, or for novice programmers to ask for help from a more advanced user.

2.2 Inspiration

Before any documentation is to be written, a primary purpose and scope need to be clearly defined. Regardless of the application, this project must have a structure that would be easy to understand, follow, and expand (Mic, Debray, & Peterson, 1993). Given the project's root and a destination directory, anyone with enough knowledge about the project should easily be able to navigate the hierarchy with minimal assistance or deviation from the true path using only semantic grammar. Different implementations must be broken down in a manner that is easy to expand. For example, what if someone wanted to add a second implementation of an existing algorithm in an existing language, do you keep the best implementation or both? What if someone wanted to add a completely new algorithm in a new language? Most importantly, it should support any operating system on any machine. The only problem that should arise is incompatibility between a machine and a programming language, but never with the project.

2.3 Understanding the target audience

A safe assumption is that, at a minimum, a novice level programmer who is either self-learning or currently in high-school or university level academics with some understanding of variable declaration, functions, classes, conditionals, and data structures would tackle this project. So, it seems doubtful that someone without fundamental programming knowledge would stumble across this project. This is meant as a fully open-source project where everyone can contribute, regardless of skill level, but the project is intended purely for the novice programmer to clone and attempt to beat the score of the given implementation as a teaching tool. The public repository's main purpose, though, is to act as a "Fort Knox" to keep "golden" data for comparison. An intermediate or expert programmer is free to contribute by optimising existing implementations or providing better ones whilst abiding to a contribution standard that should be made publicly available in the repository's documentation.

This can raise many questions as to who the official authority is to distinguish between what is to be included in the "golden" data which have a direct impact to the learning experience of anyone who decides to use them. Regarding repository ownership, I plan to be sole owner of the repository with full administrative privileges.

2.4 Breaking things down

This project is intended to be taken on as a year-long university module, with soft and hard deadlines to create an environment for learning a generous but conservative number of languages in balance. The participant should come out of this project with better time management and estimation, the ability to tackle simultaneous tasks that do not necessarily interconnect, and programming skills. It is essential to understand that this project's scope is yearly for a good reason. The participant should spend a considerable time laying down the foundations of the project rather than rushing into an "I will do as I go" mentality. For example, researching different technologies, understanding one's capabilities, strengths, and weaknesses before writing a single line of code, and creating a list of what should be achieved by the end. Planning can alleviate some of the stress in the future, and visualising this data

[2] https://discord.gg/9S4tqXWb

means that a user can focus on achieving those tasks instead of remembering them. Task management tools such as Trello[3], Monday[4], and Jira[5] are great ways of keeping tasks in order with integrated calendars to set internal deadlines. Providing one person with the task of learning several new languages and testing them on implementing different programs at different skill levels with them is no easy feat. Thought must go into what kind of languages these will be, the users' experience and syntax familiarity, and user testing.

2.4.1 Tools

I decided to begin by downloading and running the original benchmarks game and another repository that compares different languages in a prime sieve implementation. After successfully running both repositories and getting results, I decided to dig into the file structure used, the documentation available and the means of benchmarking, such as the scripts used. This gave me an idea of the approach other, more experienced; programmers have followed and the kind of results they received in terms of accuracy, verboseness, and comparison method.

I decided to focus on understanding how the repositories worked internally and try to reuse as much of that code as possible. Unfortunately, limited by my knowledge and experience, I could not achieve this, thus reserved to using publicly available and tested tools. The list of choices considered is shown in *Table 1* below:

Table 1 - Usage profiling tools

Tool Name	Measurements available	Time Accuracy
time (bash)	Track time elapsed	0.001 s
time (Linux)	Track time elapsed, CPU utilisation, and RSS memory	0.01 s
cgmemtime[6]	Track RSS memory	N/A
top	Track time elapsed, CPU utilisation, and memory	0.01 s
syrupy[7]	Track time elapsed, CPU utilisation, and memory	0.000001 s

By evaluating the information these tools could provide and comparing them with the content of the two other projects, I decided to narrow it down to Syrupy. Syrupy is an open-source Python script that wraps over the ps command to take snapshots of a given process's system resource usage. Running in Python means that the tool is potentially limited by the language's capabilities in how fast it takes snapshots and how accurate it is. However, this tool provides all required measurements in a single package, increasing the reliability of the data captured rather than using a different tool for each measurement.

The tool displays its results in a tabular form, with each row representing a second of elapsed time, which can be processed in a UNIX environment using pipes to extract and manipulate the data as required. It is worth mentioning that the tool uses both the stdout and stderr data stream, making it easier to work with and separate the logging data from general information displayed by the script. I used UNIX pipes with the awk or readarray commands to split the output into its respective values and average them.

[3] https://trello.com/
[4] https://monday.com/
[5] https://www.atlassian.com/software/jira
[6] https://github.com/gsauthof/cgmemtime
[7] https://github.com/jeetsukumaran/Syrupy

```
muffin@muffin:~/benchmarking-programming-languages
$ python dependencies/syrupy/syrupy.py -S -C --no-raw-process-log python implementations/python/sieve/sieve_run.py
SYRUPY: Executing command 'python implementations/python/sieve/sieve_run.py'
    PID        DATE      TIME      ELAPSED    CPU    MEM       RSS     VSIZE       CMD
    183   2022-04-16  16:30:30        00:00    0.0    0.0      5396     11768  python implementations/python/sieve/sieve_run.py
    183   2022-04-16  16:30:31        00:01   99.0    0.1     10180     15324  python implementations/python/sieve/sieve_run.py
    183   2022-04-16  16:30:32        00:02    100    0.1     10180     15324  python implementations/python/sieve/sieve_run.py
    183   2022-04-16  16:30:33        00:03    100    0.1     10180     15324  python implementations/python/sieve/sieve_run.py
    183   2022-04-16  16:30:34        00:04    100    0.1     10180     15324  python implementations/python/sieve/sieve_run.py
    183   2022-04-16  16:30:35        00:05    100    0.1     10180     15324  python implementations/python/sieve/sieve_run.py
    183   2022-04-16  16:30:36        00:06   86.5    0.0         0         0  [python] <defunct>
SYRUPY: Completed running: python implementations/python/sieve/sieve_run.py
SYRUPY: Started at 2022-04-16 16:30:30.022571
SYRUPY: Ended at 2022-04-16 16:30:37.065022
SYRUPY: Total run time: 0 hour(s), 00 minute(s), 07.042451 second(s)
muffin@muffin:~/benchmarking-programming-languages
$
```

Figure 1 - Syrupy output

```
muffin@muffin:~/benchmarking-programming-languages
$ python dependencies/syrupy/syrupy.py -S -C --no-raw-process-log python implementations/python/sieve/sieve_run.py 2> /dev/null
    PID        DATE      TIME      ELAPSED    CPU    MEM       RSS     VSIZE       CMD
    254   2022-04-16  16:30:52        00:00    0.0    0.0      5756     12024  python implementations/python/sieve/sieve_run.py
    254   2022-04-16  16:30:53        00:01   99.0    0.1     10064     15328  python implementations/python/sieve/sieve_run.py
    254   2022-04-16  16:30:54        00:02    100    0.1     10064     15328  python implementations/python/sieve/sieve_run.py
    254   2022-04-16  16:30:55        00:03    100    0.1     10064     15328  python implementations/python/sieve/sieve_run.py
    254   2022-04-16  16:30:56        00:04    100    0.1     10064     15328  python implementations/python/sieve/sieve_run.py
    254   2022-04-16  16:30:57        00:05    100    0.1     10324     15568  python implementations/python/sieve/sieve_run.py
    254   2022-04-16  16:30:58        00:06   84.0    0.0         0         0  [python] <defunct>
muffin@muffin:~/benchmarking-programming-languages
$
```

Figure 2 - Syrupy output with no stderr

Having tested different tools that required many installs and uninstalls, it dawned on me that dependency conflicts can occur. This called for a sandbox environment where different dependencies and tools can be installed with minimal repercussions. Since the tools consisted of Python and Bash scripts and Linux packages, the only virtual environment available was for Python, as I could not find an Ubuntu virtual environment. I chose the PyEnv[8] version manager as it is widely known in the community and has a virtualenv[9] plugin to customise the Python version and environment on a per directory level. As a result, I can install different versions of Python (i.e., 2.a.b, 3.x.y) and create different instances of the installed versions, each with an isolated list of packages.

2.4.2 Data Collection

Even with the tools chosen and tested, all they provide is a set of raw data with no context. The project must have some method of quantifying competency and skill to produce an outcome with a ranking system. Programming can be seen as a collection of concepts and semantics that can design, build, and execute a tool or algorithm. Rather than focusing on the programming itself, which each language can offer in diverse ways, the project focuses on the ability of the user to comprehend and apply these concepts to create a particular algorithm with known output. One way of quantifying such abstract data is by encapsulating them in other measurements and understanding the distinct phases of research (as defined by myself):

1. Comprehending the project
2. Material-gathering
3. Studying
4. Implementing
5. Testing

Unfortunately, human comprehension is a very subjective topic; no one can really say whether one has "understood" something until they are proven wrong. For instance, how does one check that someone has understood a subject successfully? A popular way to grasp a feel of one's understanding is to teach

[8] https://github.com/pyenv/pyenv
[9] https://virtualenv.pypa.io/en/latest/

the same topic to someone else, preferably to someone with a deeper understanding so any mistakes made can be caught and corrected. As Rick Garlikov suggests, being able to explain or recite something does not necessarily mean you have understood it (Garlikov). This way, rather than the user blindly explaining something and assessing themselves, an unbiased third party can take over the assession while they focus on the explanation. This is a crucial step that should not be skimmed over as a solid understanding of the core subject, which can differ based on the project conductor's application, is more likely to affect the user's material gathering and learning phases. Even more so if this were to be used as a teaching tool where the primary purpose was to teach a programming language.

The resources used during the learning period directly affect how a user perceives the information and at what depth. The internet is vast and filled with knowledge, among other things. Said knowledge need not always be peer-reviewed or even sourced. Everyone can promote their ideas in their own way and explain why they believe it to be the best fit. For the inexperienced, this can be overwhelming and confusing. Perhaps a user found a working solution from an unknown source with no credits or found a solution that leads them in the right direction but does not work out of the box in a credible source. Who is to say which is best and which to continue with? Should the user stick with the unsourced solution that works or focus their time on a credible community that may not give them the solution but can help steer them to the resources required? Although the answer may seem obvious to some, the user's willingness to research and the depth of understanding they wish to achieve is an unavoidable limiting factor. For example, sometimes a user wants to find a solution to their problem, not the reason behind why theirs was not working. This holds more so in the novice tier, where some programmers may not understand a language feature.

The project needs to define whether the measurement will be timeboxed or open until the user feels confident. In other words, would it be best to allow someone one week of studying at their own pace then testing them, or allow the user to spend as much time as needed to study a particular language or subject and provide the test when they feel comfortable? Would the user be allowed to solve mini problems to test their progressive learning, for example, on websites such as HackerRank? The amount of time used for learning is an excellent collection pool that can accurately represent the user's ability to comprehend the material. Due to the conditions under which the project was taken, timeboxing provided a balance between time and task managing. Soft deadlines helped evenly distribute the workload throughout the academic year and give a reasonable estimate of the workload one could tackle, keeping in mind that as a university student I had more than just one module to work on simultaneously.

It is assumed that the user has understood and is comfortable implementing the pseudocode version during algorithm implementation. This does not mean that the user should have no access to external resources but should theoretically be able to convert the algorithm's pseudocode into source code with minimal help. Consequently, any external help should be noted accordingly and considered in the final assessment for each user. Whether the implementation phase should be timeboxed or recorded is up to the person conducting the project.

Testing and validating one's solution and results is arguably more important than the implementation itself, and just as crucial as finding suitable sources to learn from, despite testing being more controversial regarding what is proper testing and when to stop. Considering the implementation is being evaluated, testing should, in theory, act as input to a black-box environment. This gives the potential of a single test suite for all implementations, with a few exceptions on helper functions a user has created, consequently enforcing a community approach where multiple novice programmers work in unison. The suggested approach is to enforce a 100% code coverage model for unit tests in toy algorithms and 80% for the markdown parser. Code coverage does not reflect completeness but ensures that every part of the codebase runs successfully. Since the toy algorithms should have no user-based

input but hard-coded values, no edge cases should arise; thus, it seems validating enough. On the other hand, the markdown parser relies on user input, and some edge cases will happen, which raises the question of when and how we know the code has been sufficiently tested?

2.4.3 Control & Consistency

Taking into account the unique nature of people, no one can account for all the variables. Most one can do is setup a controlled environment to restrict them as much as possible, increasing the validity of collected data and fair assessment for programmers of any level.

The most apparent and straightforward control that needs enforcement is the hardware. Running all the benchmarks on the same hardware will produce much more accurate results if these were for comparison. I had assumed that the CPU architecture, core/thread count, clock speeds, primary memory capacity and speed play a significant role in executing the algorithm, while background tasks and temperature can affect the overall performance of the machine. This was found to not be the case with the current tools and benchmarks that were run. These variables can be finely tuned using different cloud services such as Amazon's AWS, Google's GCP, and Microsoft's Azure which can create custom specked instances of most operating systems.

While developing, each programmer creates a lab environment where they work, using their tools, configurations, and plugins. Such tools and plugins could provide an unfair advantage. Code auto-complete, library and file indexing, and overlay documentation can speed up a user's performance. When comparing two or more users in a controlled environment, tools and plugins should be either limited to the pre-installed kit and provided to all users partaking in the project. I would go as far as to say that some extra time should be allowed for all users to have a feel of the installed tools. Docker[10] is an excellent solution for this, virtualisation software that can create lightweight and portable instances of an image. An *image* is a set of instructions used to build a Docker *container*, an executable software package. In other words, Docker is a virtual machine engine that launches quickly and uses few resources from the host machine as opposed to a virtual machine software such as Oracle's VM VirtualBox[11], whose purpose is to emulate hardware using software. The latter tend to be resource heavy and take a long time to load based on the resources manually allocated from the host machine. Docker allows the observer to create an isolated environment where all packages, language compilers, and tools are pre-installed that can be expanded and executed locally or on the aforementioned cloud platforms with no setup required, and cross-platform support.

Looking at the current education system, the assessment coursework and examinations and their respective marking schemes remain the same across students of all backgrounds and skill levels. This is not because all students are of the same calibre but because all students have gone through the same teaching phase. It is not a perfect system but considering the longevity and scale, applying the same type of system to a small-scale project such as this is a safe assumption. Therefore, the learning and implementation phases should be timeboxed and used to assess the same language and algorithm. It should be mentioned that the learning phase is more flexible than the implementation phase in the sense that both phases have a maximum timestamp but not a minimum. If (one of) the participant(s) believes they feel comfortable with the current amount of studying they have made and can solve the algorithm, the learning phase can be pre-maturely stopped to progress to the implementation phase. Consequently, if said participant later decides that extra resources will be required to assist with the implementation, these are counted towards their implementation phase time budget, and the scoring system should reflect this as a penalty to their score.

[10] https://www.docker.com/
[11] https://www.virtualbox.org/

2.5 The benchmarks

This is the part of the project where the "curriculum" is to be crafted in detail such that the participant has a clear end-goal. These are broken down into two distinct components that are meant to test different things.

Toy or "100-line" problems are meant to introduce or give a flavour of a language rather than make the user proficient in that language. They are an excellent way of exercising conditionals, recursion, and object-oriented programming, but do not, as the results have shown, spark great confidence in the competency of the programmer. By the end of this project, I expected that my ability to code and read in each language would significantly increase as I have previous experience in programming and can say that I ended up achieving the ability to at least recognise and read code written in the languages that I had worked with, although there is no quantifiable way of measuring or presenting this.

Even though toy problems can introduce a language and some of its features, they are unsatisfactory for deeper learning; hence a mid-size program to be exercised in parallel was required. This ended up being a shaved-down version[12] of the Common Mark Markdown Parser[13] that I specked. This was meant to be an expansion of the toy programs to produce an almost production capable program, specifically specked for a learner to apply as many programming features as possible like classes, conditionals, object-oriented programming, etc. The markdown parser was chosen as the program's specifications are available online where parts can be inserted or extracted without affecting other sections (mostly), making it a perfect candidate for an assertion program.

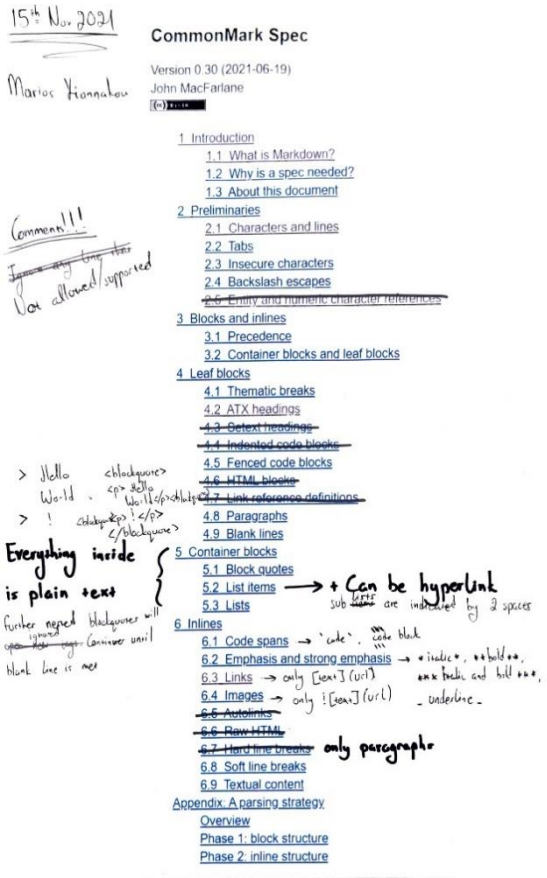


Figure 3 - Draft of the Common Mark Markdown Specification

[12] https://github.com/Mariosyian/benchmarking-programming-languages/tree/submission/markdown-parser
[13] https://spec.commonmark.org/0.30/

3 Implementation & Results

This section focuses on my personal approach, goals, and how some of my decisions were swayed from one implementation to another. This can be thought of as a ledger book of what I was able to achieve in, roughly, one academic year.

3.1 Implementation

The implemented version of this project did not assess the participant but used them as part of the control group. Since the combined number of participants for the active and control groups was one, participant evaluation was impractical. The "Benchmarking Programming Languages" project compares how different languages would perform to one another when running the same implementation of an algorithm written by the same user, along with some measurement of their competency in learning said languages. Hence most of the scoring system and assessment criteria mentioned in the previous section have been tweaked to fit this purpose.

After deciding on and setting up my tools and working environment, as mentioned in the previous section, I began to plan the languages and algorithms to be benchmarked. Evaluating the amount of time available to me and the responsibility of additional university modules, exams, and coursework, I decided that implementing at least one toy algorithm for each of five to six different languages would produce sufficient data and some interesting results.

During the planning phase, I had hoped to create a detailed outline of the year ahead, allowing me to distribute my workload and present my supervisor with a draft of what my idea would look like. Unfortunately, even with generous estimates that included debugging time, I could not accurately estimate how long tasks or debugging a problem would take due to the random nature of university work. Regardless of the outcome, setting up some time management tools freed some brainpower to focus on implementing the project rather than remembering finer details such as "What will I work on today?" or "Where did I leave off yesterday?".

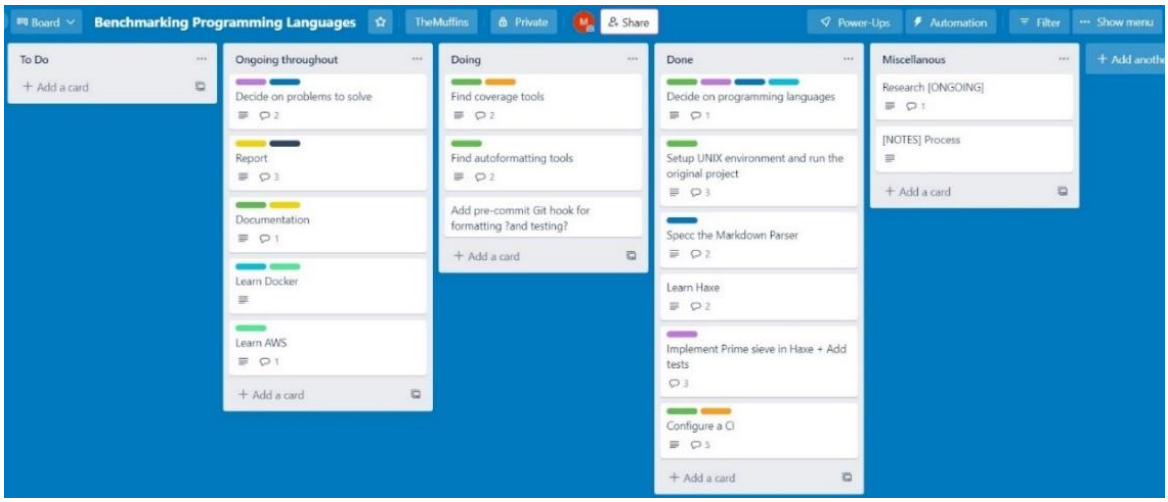


Figure 4 - Trello board used during the project

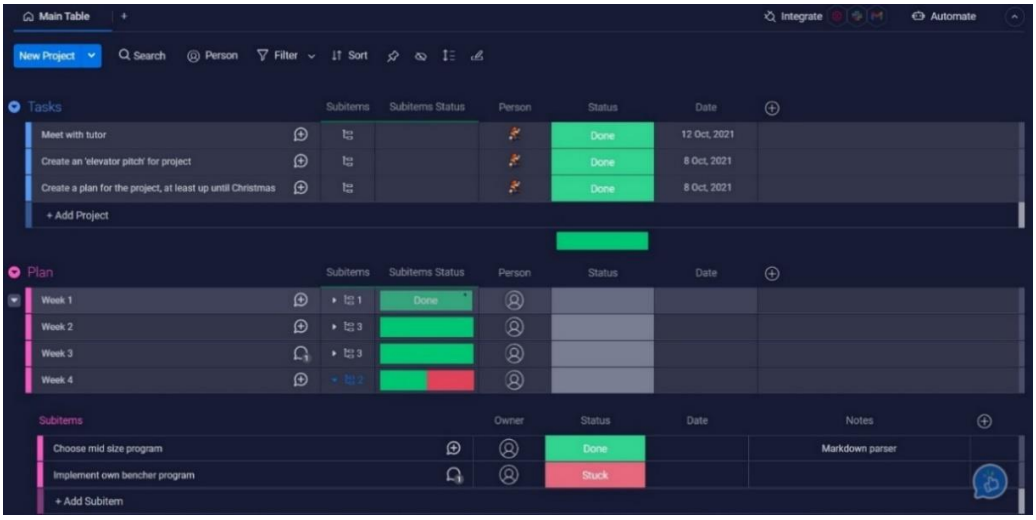


Figure 5 - Monday board used during the project

With the design finished, an online development lab is required. I used GitHub and CircleCI. I chose CircleCI as I am more comfortable with its interface, and it is a recommended app by GitHub itself. The CI environment safeguards against merging code that does not abide by the project specifications, such as adequate testing coverage, and can build and run the app to notify the observer if testing or running fails or not. CircleCI seems to require no downloads or setups, no self-hosting, and their free tier account provided more than enough resources for what I needed.

Upon deciding on the algorithms and languages to implement, I needed a method of producing a comparable result between them. Since this project was developed on a Linux system, I figured that a bash script is the most lightweight and dependable solution to run any command and utilise UNIX pipes to manipulate and redirect output. I decided to use naming conventions and file handling commands to automate the running process. When a user introduces a new algorithm or language to the project, a few things must be kept in mind.

1. Naming directories
2. Naming files
3. Updating the benchmarks script, testing script and the CI config

When designing the system, I decided that a tree hierarchy whose root contains the utility scripts, implementation and dependency directories, and documentation would work magnificently as it is quick and semantically logical to navigate. I debated whether the implementation structure should contain a set of subdirectories whose name must be the programming language name in all lowercase alphanumeric characters or a set of subdirectories whose name must be the algorithm name in all lowercase alphanumeric characters. These would then proceed to have the appropriate subdirectory set like a set of directories named after the algorithms that the language named in the parent directory implements or the name of the languages that implement the algorithm named in the parent directory. A graph representation of each structure is seen in *Figure 6* below:

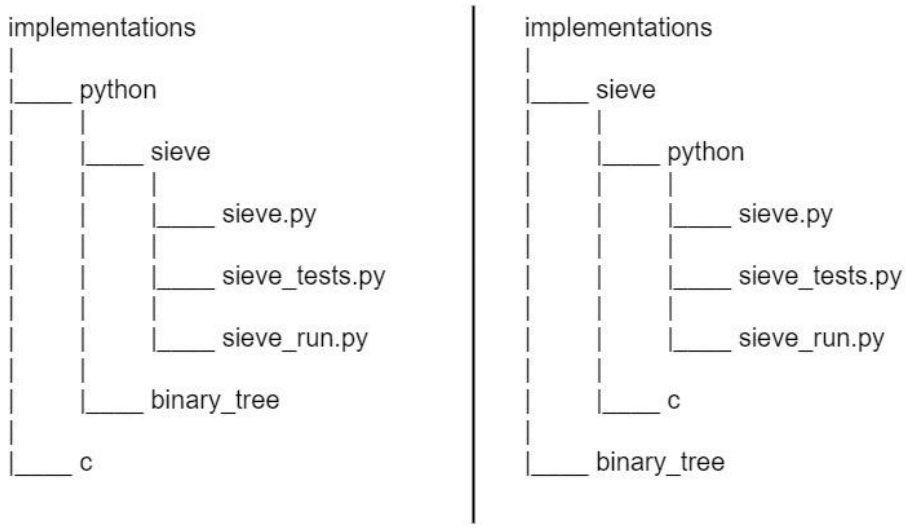


Figure 6 - The two suggested file structures.
Language parent with algorithm children (left) and algorithm parent with language children (right)

I decided to follow the language parent structure which classifies the algorithms by language purely by coincidence. My original thought was that if a user were to search for an algorithm, they would search for a specific language implementation of the algorithm; hence "Find the python implementation of a prime sieve" would be interpreted as "Go to the python directory containing the prime sieve algorithm", due to the semantic processing of a sentence. It should be noted here that the above sentence can just as equally been phrased as "Find the prime sieve implementation in python", which would then suggest the algorithm parent file structure, so it was pure coincidence that the file structure ended up being what it is (Friederici, Gunter, Hahne, & Mauth, 2004). Going back with a clearer mind about how the project unravelled, I would have chosen the algorithm parent directory hierarchy, although it makes no real difference on the benchmarking nor the script logic.

The languages chosen for this project were split into two types of languages. "Home" languages that signify a programming language that I am familiar with and capable of coding in with minimal assistance from documentation or Internet browsing, and "exotic" languages which are programming languages I have little to no confidence coding in without assistance. I decided to pursue a 1:2 ratio, such that for each "home" language I introduce in the project, there exist two "exotic" languages. This is a requirement to balance out the learning strain of the participant, explained with an example below. With the time limit, my capabilities, and programming knowledge as limiting factors, I decided on using a total of six languages. Said languages (in alphabetical order) being:

- Home
    - Java
    - Python
- Exotic
    - C
    - GoLang
    - Haxe
    - Rust

These languages were chosen based on their industry application, reputation, or capabilities. C is considered by many the golden standard of efficient programming languages, so benchmarking programming languages almost implies that C would be one of the contestants. GoLang and Rust are two emerging languages, as suggested by *Figure* 7 below, that are being integrated into the industry.

Benchmarking Programming Languages

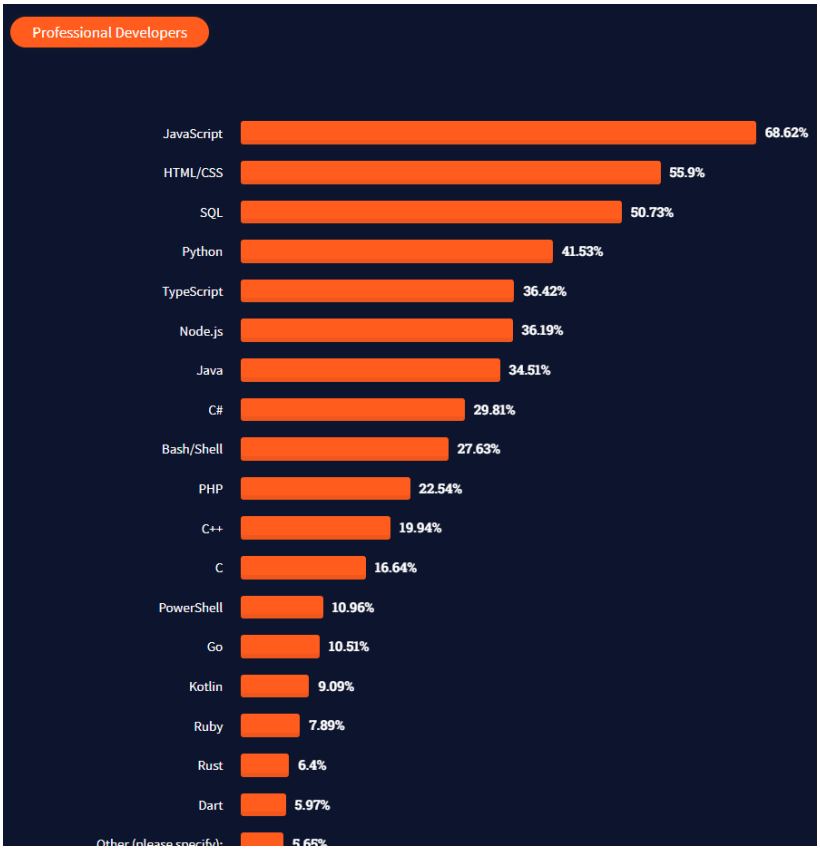


Figure 7 - StackOverflow Developer Survey 2021 - Programming language preference

Rust is also notorious for extremely verbose and helpful compiler messages, and documentation, while GoLang is famous for its quirky syntax, satirically discussed in the GoLang Reddit forum[14]. Haxe was introduced to me by my supervisor and was suggested as this is a "code once, produce many" type of languages. Haxe has the capability of translating its source code into a plethora of other languages including C++, Java, and Python.

```python
def sieve(upper_bound):
    """
    Checks all integer numbers from 2 up to, and including, `upper_bound` and computes
    all the prime numbers, then returns them.

    This algorithm goes through the multiples of each number as it ranges up to the
    `upper_bound` number, placing each prime number in a set, and non prime numbers
    in a different set. This is because accessing a set element is faster than
    re-calculating if a number is prime or not.

    :param upper_bound: The integer to act as an upper bound for the program to
        check up, and including, to.
    :returns: An unsorted set of all the prime numbers that are less or equal to
        `upper_bound`.
    """
    if upper_bound == 0 or upper_bound == 1:
        return []
    elif upper_bound == 2:
        return [2]

    primes = set()
    non_primes = set()
    for i in range(2, upper_bound + 1):
        for j in range(i):
            current_num = i * j
            if (current_num > upper_bound) or (current_num in non_primes):
                continue

            if is_prime(current_num):
                primes.add(current_num)
            else:
                non_primes.add(current_num)

    return primes
```

```python
    def contains(self, array, element):
        return python_internal_ArrayImpl.indexOf(array, element, None) != -1

    def sieve(self, upper_bound):
        if (upper_bound == 0) or ((upper_bound == 1)):
            return []
        primes = list()
        non_primes = list()
        _g_min = 2
        _g_max = upper_bound + 1
        while _g_min < _g_max:
            i = _g_min
            _g_min = _g_min + 1
            _g = 0
            _g1 = i
            while _g < _g1:
                j = _g
                _g = _g + 1
                current_num = i * j
                if (current_num > upper_bound) or self.contains(
                    non_primes, current_num
                ):
                    continue
                if self.isPrime(current_num):
                    primes = primes + [current_num]
                else:
                    non_primes = non_primes + [current_num]
        return primes
```

Figure 8 - My Python implementation (left), Haxe Python implementation (right)

[14] https://www.reddit.com/r/golang/comments/6d6uo2/is_go_really_as_awful_as_they_say/

As seen in *Figure 8* above the Python version of Haxe is less readable compared to a vanilla Python implementation, less "Pythonic", and longer (24 lines instead of 18). Furthermore, it is important to note that the Haxe version does not seem to understand or implement the Python provided structures such as `set()`, but instead uses `list()` as well as avoiding the `append()` function in favour of list addition. This would undoubtedly affect the learning capabilities of a novice programmer as it teaches wrong conventions and does not take advantage of built-in methods which are more optimised by the language authors and are possibly less expensive operations. It was extremely weird to see that Haxe is made aware that the Python addition or "+" operator is overloaded to accept the addition of lists, but not to use the built-in `append()` method considering the Haxe method of appending to a list uses a built-in method native to list objects. Another weird implementation, in my opinion, is Haxe's method of checking if an element exists inside a list. This does not return a boolean type but an integer due to the implementation of the `indexOf()` method, which returns the index of the element rather than if it exists. This is carried on to the Python implementation which generates unnecessary code although it could be argued that my way of writing the Haxe program could be at fault here. Testing the Python version of Haxe against my implementation was an interesting experiment and showed that the Python version of Haxe was tens of times slower than my Python implementation.

Going by my original statement, I intended to include at least one toy algorithm for each of the languages introduced left me with the following plan (in alphabetical order):

- A-Star
- Bellman-Ford
- Binary tree
- Breadth first search
- Insertion sort
- Prime sieve

These algorithms were not picked at random (predominantly). Most computer science students will have met or will meet during a high-school or university-level computer science course, and I believe act as a reasonable testing suite. Keep in mind that this project is not meant to deduct the participants' ability to translate the pseudocode of an algorithm to the source code of a particular language, but their ability to write code in a particular language. Can they write a conditional, create a loop, or create a class (if applicable)? The participant should solve the algorithm in one of their "home" languages, chosen at random, to reduce the learning variable so they can have a visual representation of the algorithm in a syntactically familiar source code. For each "home" language implementation of the algorithm, the same algorithm should be implemented in two randomly chosen "exotic" languages, turning it into a source-to-source translating exercise with no significant revision to the original solution. The observer could skip the random language choice but is encouraged to include it to avoid the development of a routine and challenge the user to focus on the implementation instead of which language comes next. Each language should be allowed a full working week of practice, suggested of around two hours per day for a total of ten hours, and another working week for implementation with unlimited time; of which both times should be recorded.

Furthermore, a mid-sized program was chosen that would test a deeper understanding of the participants ability to comprehend a problem and find a solution that is does not simply meet the specifications. Real world applications and input must be considered, unlike the toy algorithms which are constant. I chose the custom specked markdown parser mentioned in Section 2.5. This was only allowed to be implemented in a language after at least one toy algorithm was solved in said language, and was given a timebox of four weeks of implementation time only as the specifications were made by me.

In addition, I decided to include optional, mini "one-liner" programs that would be committed (or not) at the discretion of the participant. These were purely meant to act as a checklist of what, I believe, are the fundamentals of a programming language or a substitute to what would be considered equivalent to a "Getting Started" tutorial in case it was not provided.

```
1   // This is a program that asks the user for a grade between 0 and 100 (inclusive)
2   // and displays the appropriate letter grade.
3   READ input
4
5   IF the input is greater than 100 or less than 0 THEN
6       PRINT "Invalid grade"
7   ELSE IF the input is 100 THEN
8       PRINT "A+"
9   ELSE IF the input is greater than or equal to 90 THEN
10      PRINT "A"
11  ELSE IF the input is greater than or equal to 80 THEN
12      PRINT "B"
13  ELSE IF the input is greater than or equal to 70 THEN
14      PRINT "C"
15  ELSE IF the input is greater than or equal to 60 THEN
16      PRINT "D"
17  ELSE IF the input is greater than or equal to 50 THEN
18      PRINT "E"
19  ELSE IF the input is less than or equal to 40 THEN
20      PRINT "F"
21  ELSE
22      PRINT "Invalid grade"
```

Figure 9 - Pseudocode for conditionals mini program

Whilst writing the implementation of the first algorithm in one of my "home" languages simultaneously began testing the time management aspect of the implementation phase. When using a language that I am comfortable with, the learning phase was cut short by only studying the pseudocode and other implementations of the algorithm in the same language. On the other hand, "exotic" languages were heavily tilted towards the learning phase, and all took up the full five working days of two hours of studying per day to study the documentation and solve the "Get Started" tutorial or suggested mini programs. The time was managed manually using alarms or timers and were documented inside the commit message. Depending on the "home" implementations during the implementation phase of "exotic" languages, I noticed that having the "home" implementation available was a greater resource than switching over to online documentation or threads and increased consistency across the language implementations.

After the first algorithm was implemented in one of my "home" languages I began to work on the benchmark script and continuous integration configuration in parallel. This ensured that the repository was in working order and would be able to revise either of the files as the project scales. Considering the project's goal of accessibility and expandability, the benchmark script had to be rigid to account for user error while staying beginner-friendly and easy to manipulate. Although the script began as a hard-coded array reader traversing through the file structure, it has evolved into a more sophisticated process that can look intimidating to a novice programmer or Bash user but continues to use semantic grammar to allow anyone to easily manoeuvre it.

```
function bench_toy_programs() {
    cd $PROGRAMS_DIR || exit 3
    for language in $(find ./ -maxdepth 1 -type d | sed 1d); do
        # Get rid of the leading './'
        language=${language:2:${#language}}
        lang=$(regex $language)

        cd $language || exit 3
        for algorithm in $(find ./ -maxdepth 1 -type d | sed 1d); do
          algorithm=${algorithm:2:${#algorithm}}
           cd $algorithm || exit 3
           case $lang in
              rust)
                 rustc "${algorithm}_run.rs" -o "${algorithm}_run"
                 COMMAND="./${algorithm}_run"
                 if [ $TEST -eq 1 ]; then
                    echo "> Running Rust tests for $algorithm"
                    rustc --test "${algorithm}_test.rs" -o "${algorithm}_test"
                    ./${algorithm}_test
                    if [ $? -ne 0 ]; then
                       exit 1
                    fi
                 fi
                 ;;
```

Figure 10 - Snippet from one of the main functions of the benchmarking script

Notice how the function in *Figure 10* above is given a descriptive name such that someone could instantly identify its primary purpose. The rest of the script may look daunting to someone who has no prior experience with the Bash scripting language, but it is written so that any novice user can copy and paste specific parts to use as templates and plug in the appropriate values. It automatically captures all subdirectories in the current directory, traverses and checks them against a `case` statement to see if any compilation steps exist for the language that that directory represents.

The CircleCI configuration, on the other hand, is more complicated. CircleCI utilises the YAML language to create human-readable configuration files, but for someone who has not written anything like this before, it can be considered a learning curve and is usually not a skill taught in academics. Having no prior experience with configuration files nor the YAML language, I heavily relied on the documentation to get me started, add features, and debug.

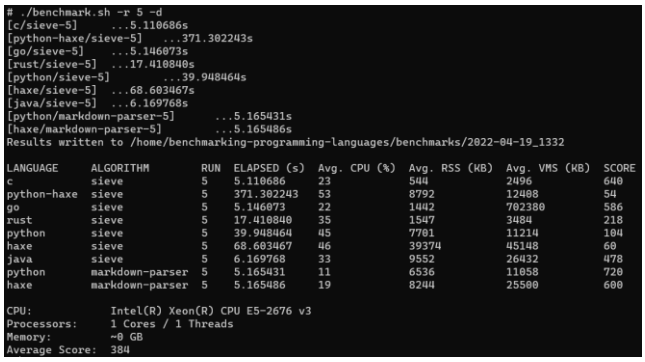
```
    rust:
      docker:
        - image: cimg/rust:1.57.0
      steps:
        - checkout
        # Format
        - run: echo "Checking formatting ..."
        - run: rustfmt --check $(pwd)/implementations/rust/**/*.rs
        # Build and Test
        - run: echo "Building and testing ..."
        - run: $(pwd)/circleCI.sh --rust
```

Figure 11 - Snippet from the CircleCI YAML configuration file

With a working version of the project, albeit on a small scale, it was time to address the universal aspect of the specifications. Universal means anyone should be able to run this project regardless of hardware or software, except for what is required. I created a Docker image[15] based off the Ubuntu base image and built from that, installing each language and its dependencies individually, cloning the codebase and running the script using its default configuration which can be overwritten using a Docker command.

I experimented with different systems and architectures throughout the academic year to ensure that the project's backbone was rigid. At this point I focused more on laying the foundations rather than building them up. This acted as a double-edged sword where the project ensures the next person picking this up needs to do minimal setup work but ate away on my implementation time. As a result, I only managed to implement one of the chosen algorithms in all the six languages described above and only two implementations of the markdown parser. This concern was discussed with my supervisor, and we concluded that the barebones structure of the project should precede the algorithm implementations since some data already exists. Since not much time would go into the algorithm implementations, I decided to explore an area that I read about at the beginning of the academic year but postponed until the universality of the project was complete. According to an article by Edward Walker (Walker, 2008), cloud computing's potential for High-Performance Computing (HPC) was caught and tested during the preliminary stages of its life, utilising Amazon's AWS Elastic Cloud Computing (EC2) service to rent out a moderately specked server matching the physical hardware Edward had available as close as possible. The article states that although the technology was not mature enough at the time, it would be interesting to revisit the idea down the line. Inspired by this article, I decided to take advantage of Amazon's enormous resources and EC2's free tier to run my project on what ended up being a 1 vCore / 1 vThread Intel Xeon E5-2676 v3 instance with 16GB of storage and 1GB RAM.

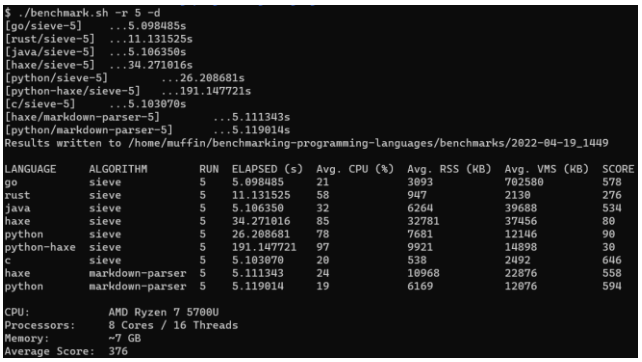
Table 2 - Benchmark results. AWS EC2 Instance (left), my personal computer (right)



```
# ./benchmark.sh -r 5 -d
[c/sieve-5]     ...5.110686s
[python-haxe/sieve-5]   ...371.302243s
[go/sieve-5]    ...5.146073s
[rust/sieve-5]  ...17.410840s
[python/sieve-5]        ...39.948464s
[haxe/sieve-5]  ...68.603467s
[java/sieve-5]  ...6.169768s
[python/markdown-parser-5]      ...5.165431s
[haxe/markdown-parser-5]        ...5.165486s
Results written to /home/benchmarking-programming-languages/benchmarks/2022-04-19_1332

LANGUAGE     ALGORITHM        RUN  ELAPSED (s)  Avg. CPU (%)  Avg. RSS (KB)  Avg. VMS (KB)  SCORE
c            sieve            5    5.110686     23            544            2496           640
python-haxe  sieve            5    371.302243   53            8792           12408          54
go           sieve            5    5.146073     22            1402           702380         586
rust         sieve            5    17.410840    35            1547           3484           218
python       sieve            5    39.948464    45            7701           11214          104
haxe         sieve            5    68.603467    46            39374          45148          60
java         sieve            5    6.169768     33            9552           26432          478
python       markdown-parser  5    5.165431     11            6536           11058          720
haxe         markdown-parser  5    5.165486     19            8244           25500          600

CPU:            Intel(R) Xeon(R) CPU E5-2676 v3
Processors:     1 Cores / 1 Threads
Memory:         ~0 GB
Average Score:  384
```



```
$ ./benchmark.sh -r 5 -d
[go/sieve-5]    ...5.098485s
[rust/sieve-5]  ...11.131525s
[java/sieve-5]  ...5.106350s
[haxe/sieve-5]  ...34.271016s
[python/sieve-5]        ...26.208681s
[python-haxe/sieve-5]   ...191.147721s
[c/sieve-5]     ...5.103070s
[haxe/markdown-parser-5]        ...5.111343s
[python/markdown-parser-5]      ...5.119014s
Results written to /home/muffin/benchmarking-programming-languages/benchmarks/2022-04-19_1449

LANGUAGE     ALGORITHM        RUN  ELAPSED (s)  Avg. CPU (%)  Avg. RSS (KB)  Avg. VMS (KB)  SCORE
go           sieve            5    5.098485     21            3093           702580         578
rust         sieve            5    11.131525    58            947            2130           276
java         sieve            5    5.106350     32            6264           39688          534
haxe         sieve            5    34.271016    85            32781          37456          80
python       sieve            5    26.208681    78            7681           12146          90
python-haxe  sieve            5    191.147721   97            9921           14898          30
c            sieve            5    5.103070     20            538            2492           646
haxe         markdown-parser  5    5.111343     24            10968          22876          558
python       markdown-parser  5    5.119014     19            6169           12076          594

CPU:            AMD Ryzen 7 5700U
Processors:     8 Cores / 16 Threads
Memory:         ~7 GB
Average Score:  376
```

Although the 1 vThread cloud machine should, on paper, not be comparable to my home machine that hosts an 8 Core / 16 Thread AMD Ryzen 7 5700U with 512GB of storage and 16GB RAM, the benchmarking results displayed in *Table 2* suggest that the Intel Xeon server CPU achieved a higher score in this singular run. Looking deeper into the machines specifications and the information provided by the script, secondary storage should not affect the calculations unless the primary storage runs short. Assuming each programming language utilises any form of garbage collection, (some of) the memory used by that process should be freed; thus, even in the EC2 instances, 1GB of RAM appears to be enough to run this project in its current state. If that was not the case and the primary storage capacity would have been exceeded, then the system would need to resort to its secondary

[15] https://hub.docker.com/r/mariosyian/benchmarking-programming-languages

storage medium as swap space which is tens, if not hundreds, of times slower than the primary storage (depending on the type of medium, i.e., HDD, SSD, NVMe) possibly affecting the overall performance. The result could hint towards the nature of the test not being stressful enough. The project is meant to test the ability of a system to run a set of algorithms in different language, thus the difficulty of the algorithm should not play an effect. An interesting measurement that was not taken into account would have been the thermals of the CPU package or each independent Thread but is realistically impractical as a closed and air-controlled environment is required for consistency.

The Xeon gets the advantage on the CPU utilisation score, which could be a fault of the Syrupy usage profiler and how it collects this data as it returns a percentage number documented as "*CPU time used divided by the time the process has been running (cputime/realtime ratio), expressed as a percentage.*" by the official repository. Perhaps the Intel Xeon architecture, which is meant to be run on Server computers, could have an effect compared to a consumer grade office laptop CPU. On the other hand, the resident set size (RSS) and virtual memory size (VMS) measurements which, despite being given the lowest weight, tend to give a bigger picture of how a language works internally. These have been given a low weight as in modern times, system memory is usually of little to no concern, as well as the fact that this project run with no problems on an EC2 instance with 1GB of primary memory.

Using the scoring system shown in *Table 3* below, we can deduce that although the elapsed time contributes the most points and is within a margin of error for most of the programs except the Python implementations. As a result, neither of the systems should have received an advantage as even the faster system still exceeds the maximum threshold and scores 0. This indicates that the scoring system may be flawed instead of the measurements being unfair.

Table 3 - Scoring system

```
331     # Calculate the score
332     # The score is out of 100 with a weighted distribution on each of the four measured
333     # properties as follows:
334     # - Time contributes to 50% (lower is better)
335     #   - 100% = 1 second
336     #   - 0% = 10 seconds+ TODO: Review again after more algorithms are introduced
337     # - Average CPU utilisation contributes to 30% (lower is better)
338     #   - 100% = 1%
339     #   - 5% = 100%
340     # - Average RSS contributes to 10% (lower is better)
341     #   - 100% = <=3000 ?? Based on algorithm ??
342     #   - 0% = >=10,000
343     # - Average VMS contributes to 10% (lower is better)
344     #   - 100% = <=6000 ?? Based on algorithm ??
345     #   - 0% = >=100,000
346     time_score=$(bc <<< "(10 / $elapsed_time) * 50")
347     cpu_score=$(((100 / $local_average_cpu) * 30))
348     rss_score=$(((3000 / $local_average_rss) * 10))
349     vms_score=$(((6000 / $local_average_vms) * 10))
```

	Weight (%)	Minimum Score	Maximum Score
TIME (s)	50	10s +	<= 1s
CPU (%)	30	100%	<= 1%
RSS (KB)	10	>= 10,000 KB	<= 3,000 KB
VMS (KB)	10	>= 100,000 KB	<= 6,000 KB

The Xeon AWS machine was explicitly used as it is the only free tier machine available and has only a single core with a single thread. This instance acted as a control for any language that uses multi-threading by default which should have an advantage in my home machine as, according to Docker's default resource allocation, it should be using as many resources as it requires.

Now that I knew the benchmark script was working, I carried forward with implementing the prime sieve algorithm in the other languages. While implementing, I noticed that different languages come with different core libraries in their "vanilla" form despite data structures being the algorithms' backbone. For example, C and Golang did not have set implementations out of the box, but unlike other languages, such as Java, it is not as simple as importing them using a package manager. C, Golang, and Haxe do not support sets, to begin with. An array had to be implemented with support functions to emulate a set, all much more complicated and time-consuming than importing a ready to

use API that comes with the language. I wanted to avoid external libraries as much as possible, especially if these were not provided by the languages original authors to avoid unfair advantages.

Another thing would be the flavour of each language. Most languages resemble the C-style syntax, which I was accustomed to from Java and found that it assisted with learning since I would compare it to what I already know, the coding flow felt more natural. I focused on learning the keywords and how certain features like for-loops and conditionals are written, although some languages like Rust care about the directory structure and need some internet browsing and tinkering to achieve a desirable state. Implementing the algorithm in a "home" language first helped visualise the results and the source code, so it became a matter of translating the source code from one language to another. I assume that if I were to write an algorithm in one of the "exotic" languages that I am not comfortable with, it would take longer because of the extra learning phase. That is to say, more time would have been spent learning and understanding the language rather than implementing the algorithm. This brings about a balance between implementing an unknown algorithm in a language I feel comfortable with and implementing a known algorithm with an unknown language. Each have their pros and trade-offs. Implementing an unknown algorithm in a "home" language allows the participant to focus all their resources into designing and testing and implementing a known algorithm in an "exotic" language allows the participant to focus their resources into learning the language.

Furthermore, I noticed a difference in code writing capabilities. Code writing style is a programmer's personal touch, convenience, and, frankly, something that is just not important. Different languages provided different naming and spacing conventions. These are trivial yet subjective "design features" that the original author of the language dictates. To resolve such issues, I decided that, wherever available, an autoformatting tool must be introduced and used upon committing a participant's changes using git hooks. Unfortunately, not all languages came with a built-in autoformatting tool, and it was not easy to find third party ones. GoLang, Haxe, and Rust came with a formatting tool that runs from the command line, so it was effortless to integrate into a git hook. Python has the famous "Python Black" tool, which is highly customisable, actively maintained, and can identify the correct files by itself, while C and Java have some tools that require some configuration before they can be used.

Regardless of the language, compilation steps, and underlying code, an algorithm should act as a black box where an input becomes an output, and the same input should give the same output (depending on the application). The only way to ensure this is through testing. This is where I noticed the biggest difference between languages from absolutely documented to, I had to spend a day finding a testing suite library. Python and Rust were the easiest to go from scratch to a fully implemented and running state. C was the most difficult, requiring additional research and libraries with little to no documentation easily accessible, finding myself depending on my IDE's ability to auto complete and suggest functions so testing my solution was a chore. Testing is an extremely important phase of development and can save future headaches, catch bugs before they happen, validate logic comparisons or calculations, and increase confidence in code rigidity thus, it is important to create a pleasant user experience to encourage it. The project required to know the overall code coverage before allowing a solution into the main branch. Any toy problem must have 100% code coverage, while a markdown parser must have 80%+. Having said that, either due to time constraints or a lack of code coverage tools, some implementations do not meet these requirements and have been carefully compared with known good test suites and source code to ensure that if there were to be a coverage tool introduced, they would be within a margin of error to the requirements. For example, Java has a variety of code coverage tools, none of which I was able to get working in my repository at the time of writing the code. The tools I tried and the problems I run into are shown in *Table 4* below.

Table 4 - Java coverage tools

JaCoCo[16]	Able to set up and run but gave 0% coverage on all implementations
Cobertura[17]	Unable to set up and use with documentation available
CodeCover[18]	Unable to set up and use with documentation available
Emma[19]	Able to set up but unable to give the right command to run the test class

This does not necessarily reflect on the quality of the documentation of the tools, but on my ability to comprehend and follow the instructions provided with the knowledge I had at the time.

3.2 Evaluation

With the implementations and testing finished, and the benchmarking script producing structured results, I decided to run the project on different machines with different Operating Systems and specifications, listed in tables 5 and 6 below:

Table 5 - Development Machine (Natively runs the project)

Personal Laptop	Operating System	Windows 11 Home version 21H2 build 22000.613 (Ubuntu 20.04 5.10.16.3-microsoft standard-WSL2 x86_64)
	CPU	AMD Ryzen 7 5700U (8 Core / 16 Thread) @ 1.80 GHz
	RAM	16GB SODIMM @ 3200 MHz
	Storage	512GB BC711 SK Hynix (NVMe)

Table 6 - Testing Machines (Run Docker image)

Personal Desktop	Operating System	Windows 10 Pro version 21H2 build 19044.1645
	CPU	AMD Ryzen 9 3900X (8 Core / 16 Thread) @ 3.8GHz
	RAM	16GB DDR4 @ 3200 MHz
	Storage	1TB APS-SE20G-1T (NVMe)
AWS t2.micro	Operating System	ubuntu-focal-20.04-amd64-server-20211129
	CPU	Intel Xeon E5-2676 v3 (1 vCore / 1 vThread) @ 2.40GHz
	RAM	1GB
	Storage	16GB AWS gp2
AWS c3.4xlarge	Operating System	ubuntu-focal-20.04-amd64-server-20211129
	CPU	Intel Xeon E5-2676 v3 (8 vCore / 16 vThread) @ 2.40GHz
	RAM	16GB
	Storage	8GB AWS gp2
AWS mac1.metal	Operating System	Darwin Kernel Version 21.4.0
	CPU	Intel i7 8700B (6 Core / 12 Thread) @ 3.20GHz
	RAM	32GB
	Storage	100GB
Kamil's MacBook Pro	Operating System	macOS Catalina 10.15.3
	CPU	Intel i5 1038NG7 (4 Core / 8 Thread) @ 2.00 GHz
	RAM	16GB
	Storage	512GB

[16] https://github.com/jacoco/jacoco
[17] https://cobertura.github.io/cobertura/
[18] http://codecover.org/
[19] http://emma.sourceforge.net/

Benchmarking Programming Languages

The project was run successfully at least once on a Windows 10, Windows 11, Ubuntu (Linux), and MacOS image, so I am confident that the Docker image will provide universality regardless of the underlying Operating System or kernel in the host machine. The project was run five times per machine that I had full access to get a comparable set of data for each aspect of the measurements, as shown below. It is worth mentioning that if a participant were to replicate these machines and run the project does not guarantee that they would get the same results! The project was run on the same internet connection each time inside a Docker container which uses as many resources as it needs, so background tasks and running applications filling the systems RAM can cause a drop in performance although this theory was not tested.

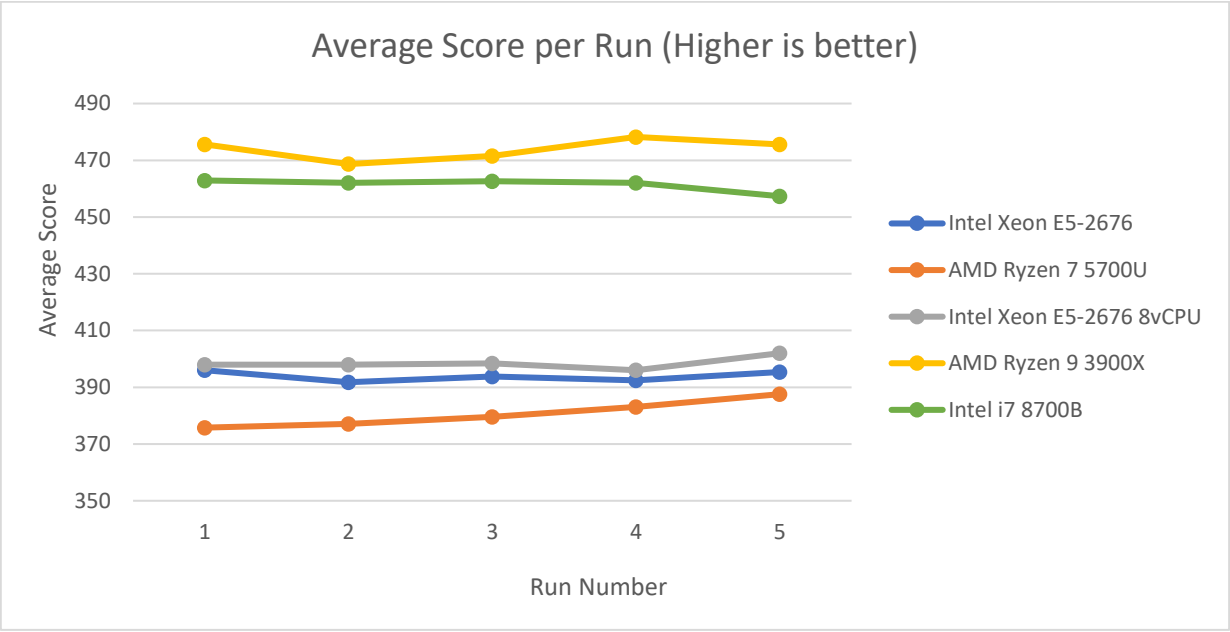


Figure 12 - Average score of each system per run over 5 runs

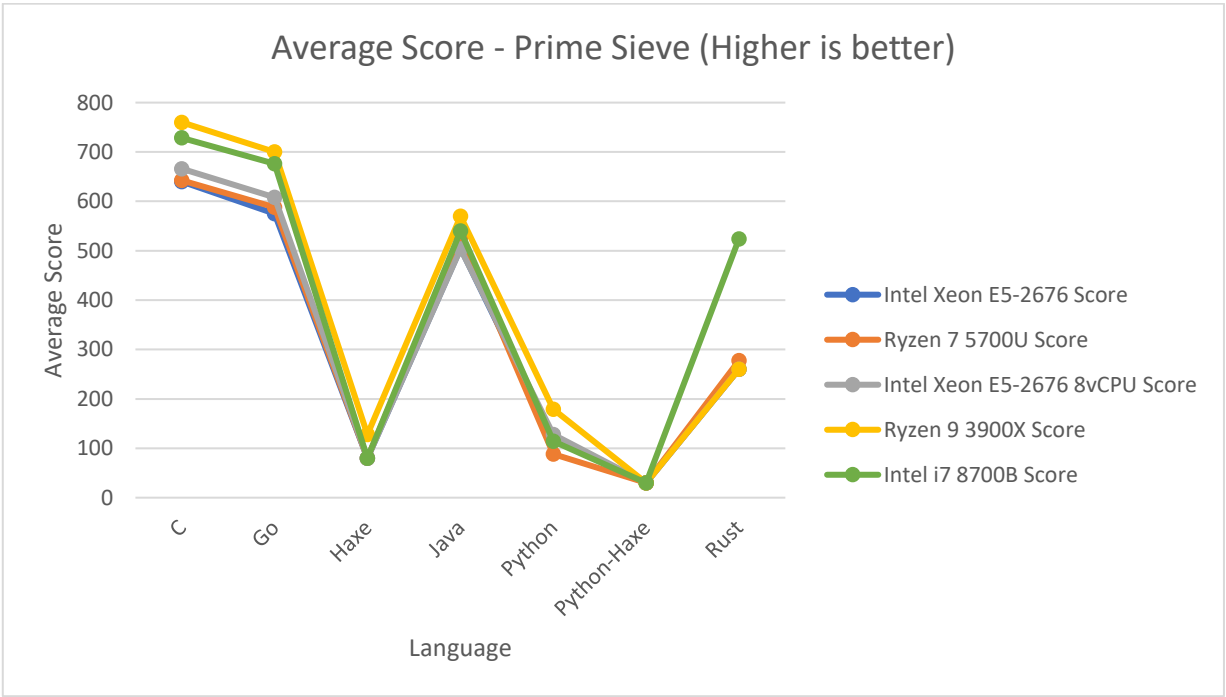


Figure 13 - Average Score of each system for each language over 5 runs for the Prime Sieve algorithm

Figures 12 and 13 above depict the project-wide average scores for all algorithms and individual language average scores for the prime sieve algorithm, respectively. This proves that even though different systems will receive different scores due to their specifications, the graphs show a constant trend across each system. This indicates that the system contributes as much as its specifications allow it to, rather than due to a compiler or CPU architecture bias, so a CPU with a high thread count that can distribute a workload to as many threads as possible potentially reports a low CPU percentage utilisation and could dominate the scoreboard.

Additionally, the Docker image is downloaded and ran on each system, hence any latency issues that could arise from AWS' EC2 instances due to network stability should have no impact on performance here. To test this, I ran the benchmarks for the AWS t2.micro instance on my personal desktop machine which has a dedicated Ethernet connection, a much faster connection than the WiFi connection that my laptop uses as seen in *Figure 14* below. Both connections were made in the same location so the ISP, router, and possibly network traffic were the same across all tests.

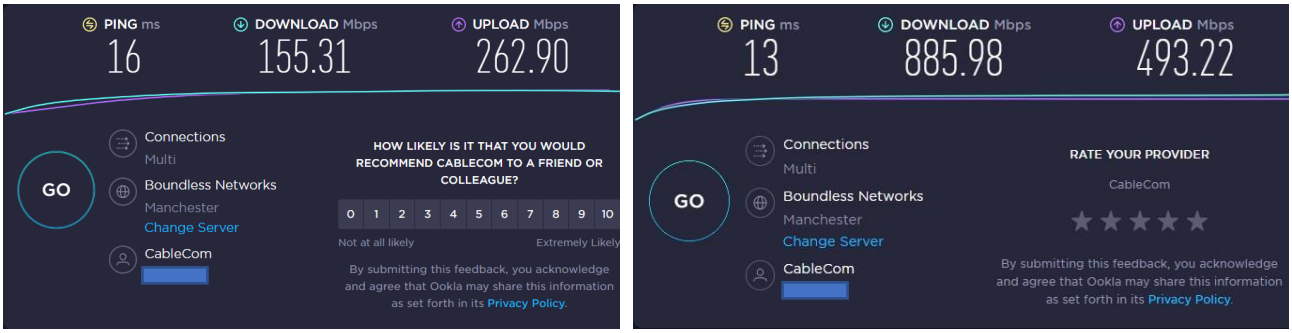


Figure 14 - Speed test comparison WiFi (left), Ethernet (right)

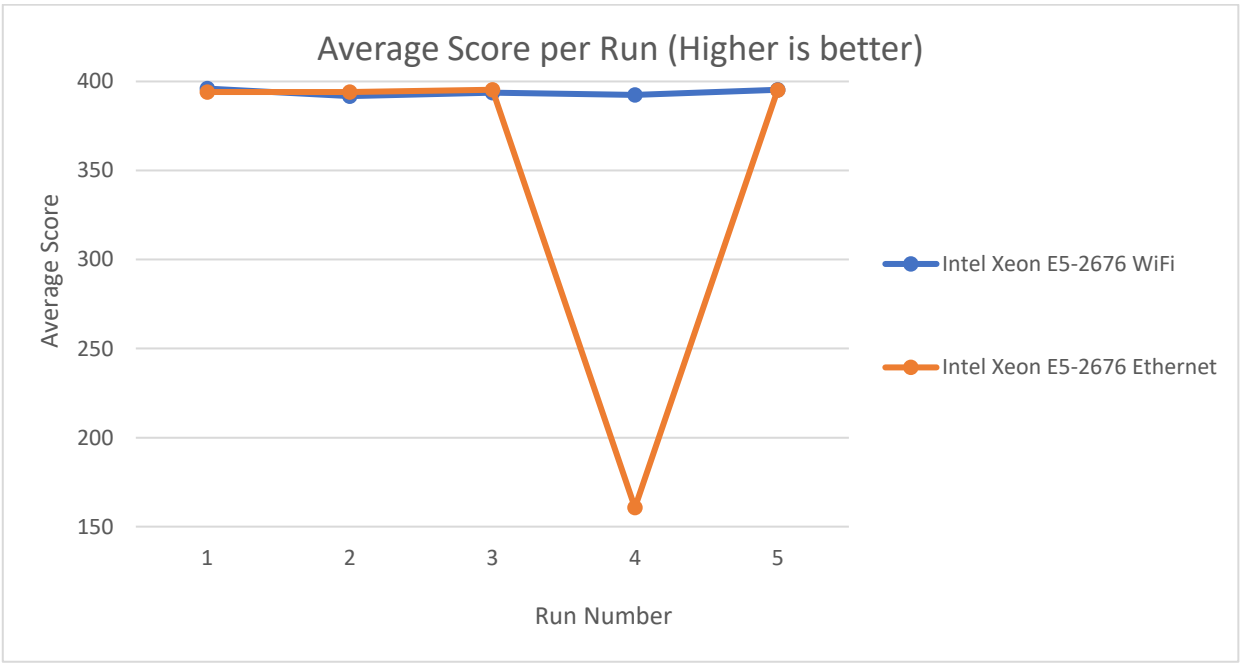


Figure 15 - Comparing 5 runs from a WiFi and Ethernet connection

Excluding a bad result on run number 4, *Figure 15* above suggests that the difference between tests run on the slower WiFi connection and the faster Ethernet connection made no difference. Despite the

WiFi connection being significantly slower than the Ethernet connection, it is not considered a slow connection but could suggest that the network speed will only affect the latency at which the terminal or GUI screen is being refreshed but have no effect on the outcome of the project.

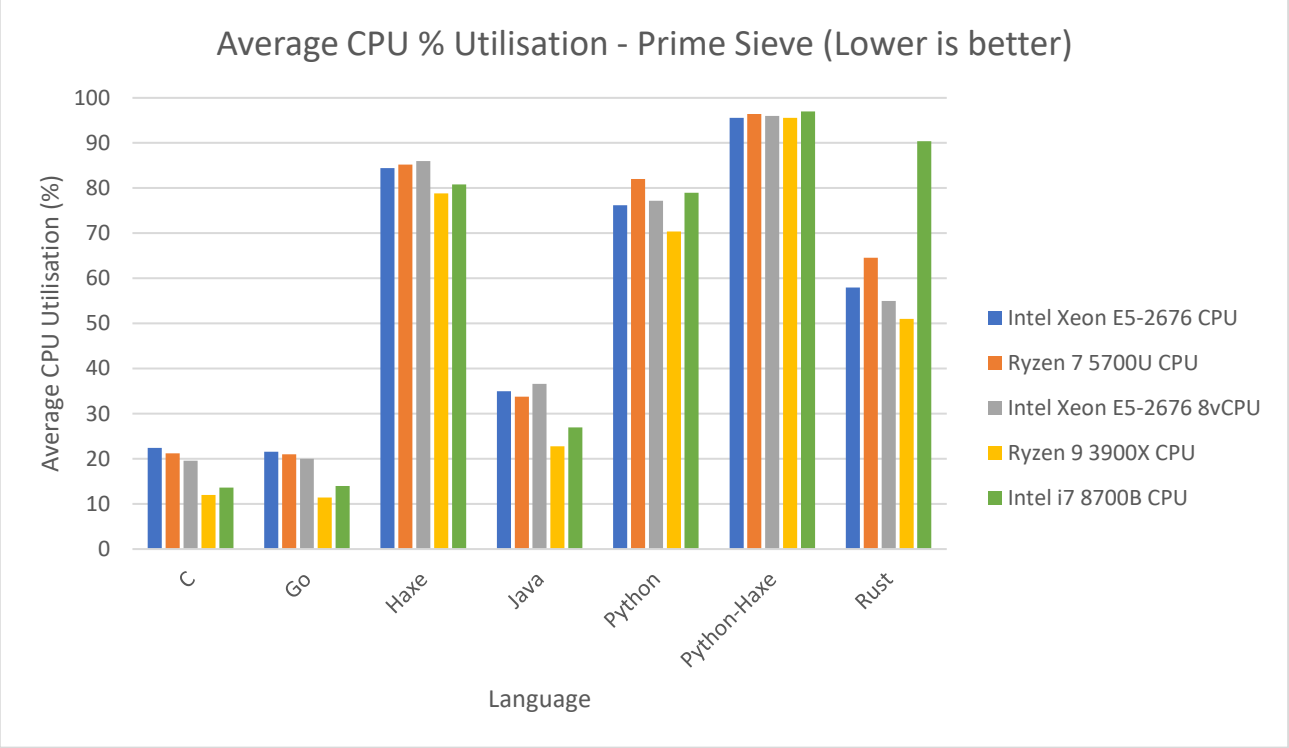


Figure 16 - Average CPU utilisation of each system over 5 runs for the Prime Sieve algorithm

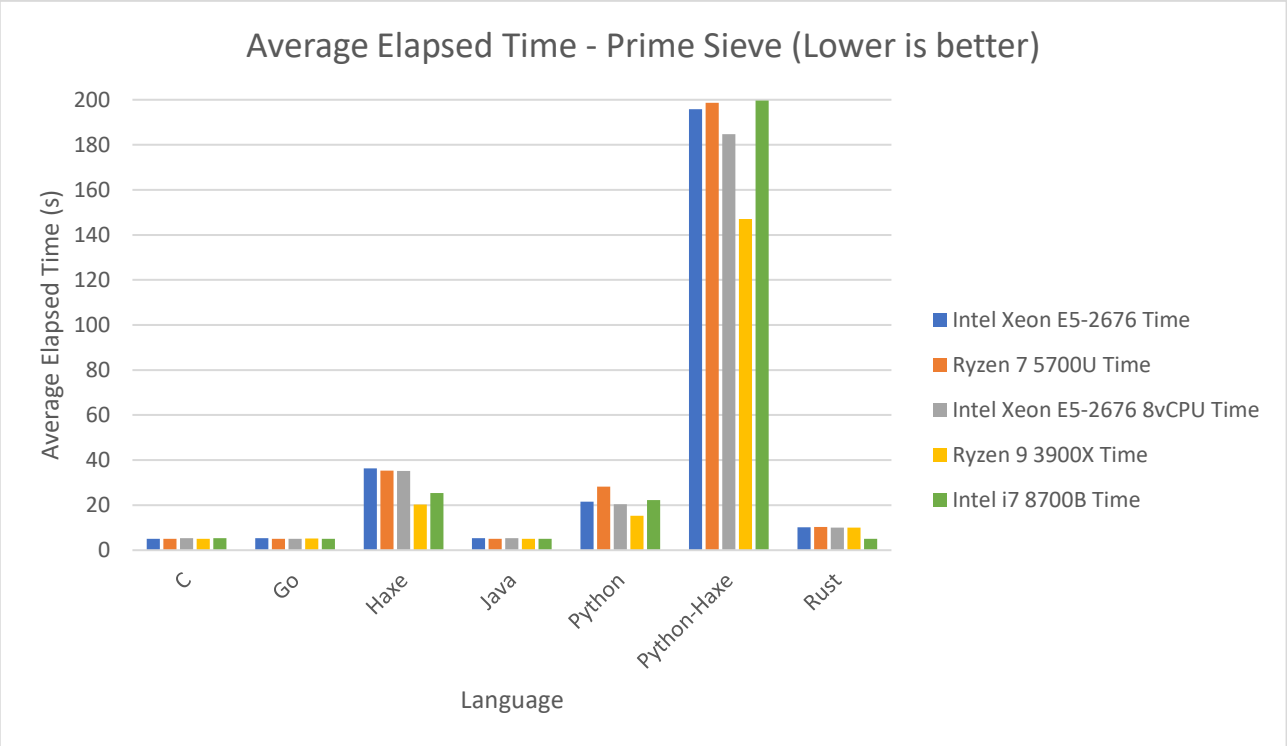


Figure 17 - Average elapsed time of each system over 5 runs for the Prime Sieve algorithm

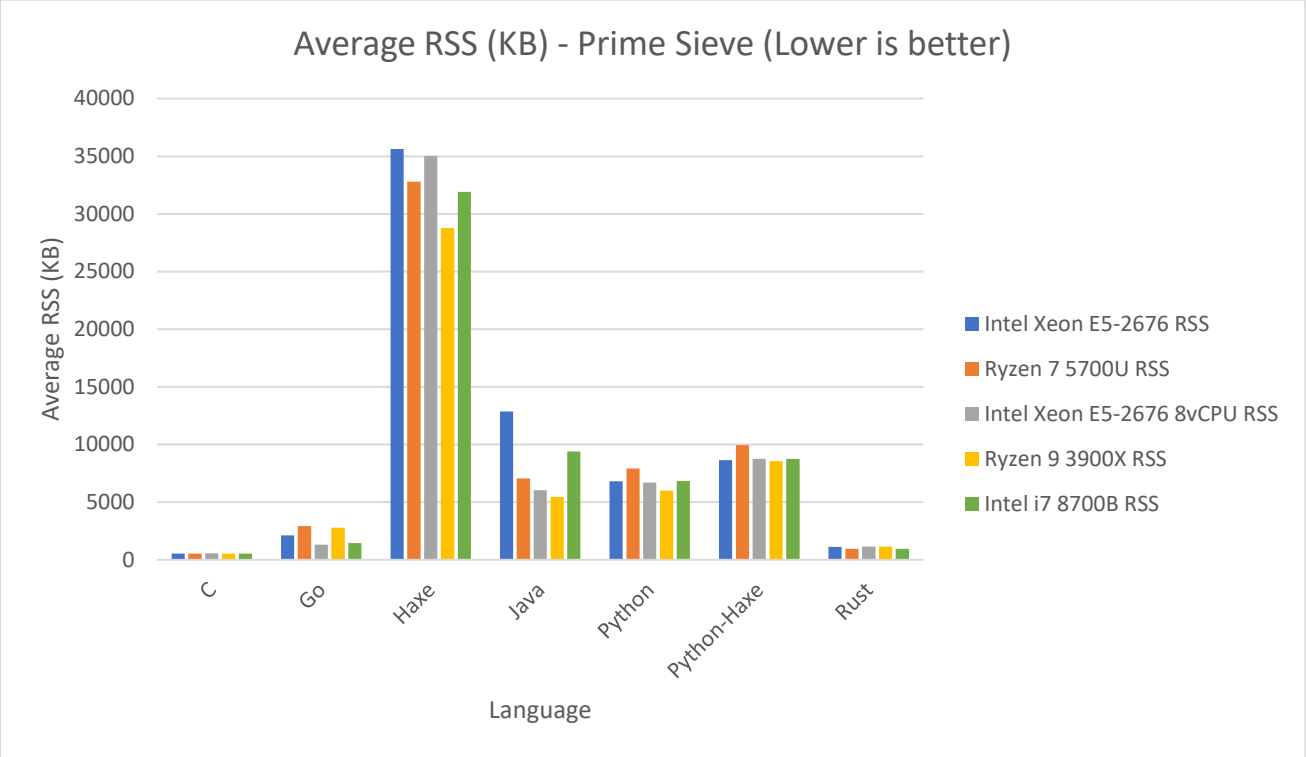


Figure 18 - Average Resident Set Size (RSS) of each system over 5 runs for the Prime Sieve algorithm

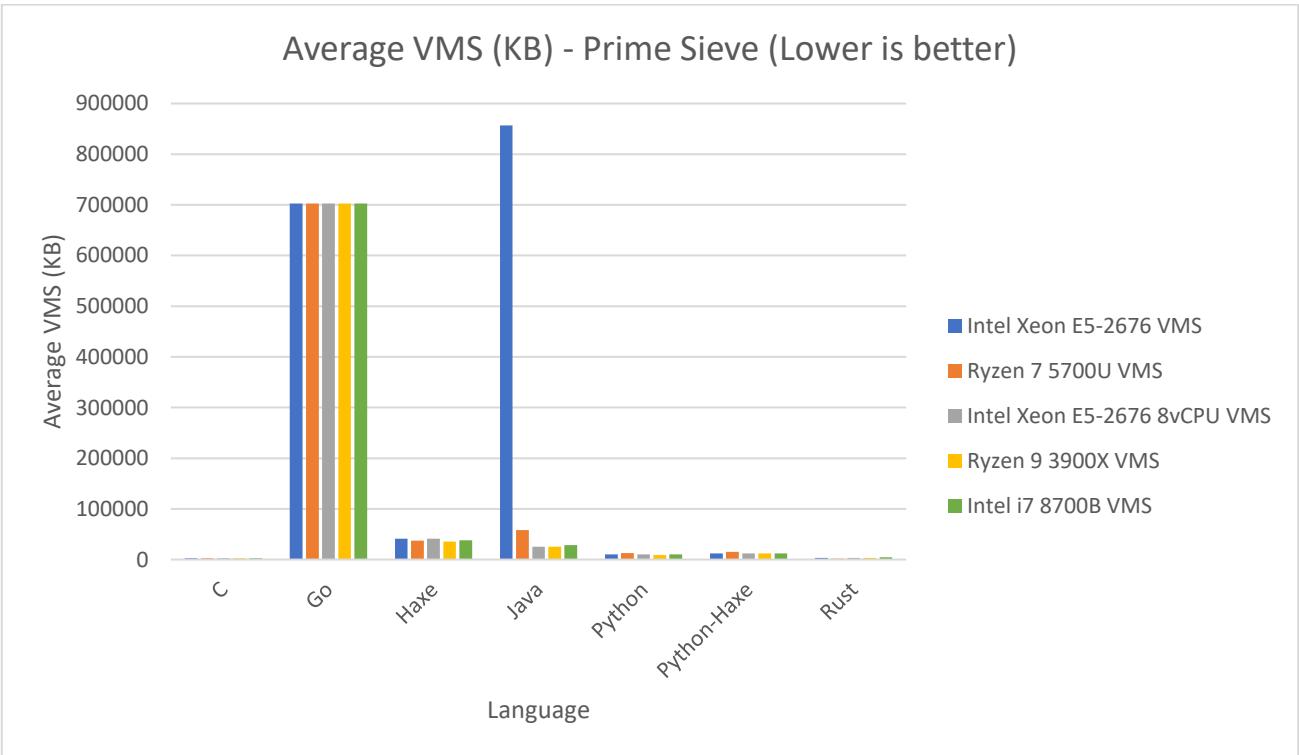


Figure 19 - Average Virtual Memory Size (VMS) of each system over 5 runs for the Prime Sieve algorithm

As discussed in figures 12 and 13, we can see that the system plays a significant role in the average score but not as much in some of the individual measurements. Figures 16, 17, 18, and 19 show that apart from the CPU percentage utilisation, which is exceptionally close, the C language comes out first in all other measurements with a distinct difference. This proves that even a moderate to low spec machine such as AWS' t2.micro could be a potential candidate for offloading CPU tasks such as sorting

or prime calculating algorithms. This can save costs on big corporations and research facilities that spend much of their resources on these tasks. For comparison, my AMD Ryzen 9 3900X machine costs £1,500, around £900 without the GPU, whereas an Amazon AWS t2.micro instance is included in the free tier accounts and can be run freely for 12 months with a time budget of 750 hours per month[20].

4 Challenges

This section talks about the challenges that I faced during the academic year and either how I overcame them, or what I did to mitigate them.

4.1 Hardware failure

Since this project was being developed for a Linux distribution, I decided to install and run a Linux distribution on my machine for native support. Even if all my files and notes were backed up on GitHub or physically in a notebook, any libraries, installed software, and environment configurations only exist within my local drive. Unfortunately, due to what I believe was a driver failure, the Linux distribution was unable to wake up my personal laptop from sleep which acted as my main development machine, and eventually destroyed it.

This left me without a development machine for a total of four working weeks until the manufacturer collected, diagnosed, and repaired/replaced my system. Four weeks during which I would use my personal desktop machine for development as university policy does not allow for external drive booting, or external software installs on their machines. Since the laptop contained all local configurations and special libraries that required download, I decided to extract my hard drive and clone it. A process that took about three working days due to the size of the drive and my inexperience in the matter.

After resuming operation with my Linux distribution transferred to my desktop, I was able to work during early mornings and late nights as university work and lectures did not allow me to always be present in my home. With an hour or two of breaks in between lectures, it was infeasible for me to be at home, make it back to university, and having done any work.

4.2 Ancient Libraries

To run the original benchmarks game, which was written in Python 2 an end-of-life product, a few pre-requisite libraries needed to be installed with no clear instructions on what these libraries were. To find out which libraries were needed I run the program and followed the runtime errors which would point out any missing libraries and I proceeded to install them. A particular library called gtop seemed to fail constantly in a virtual Python environment but was able to run it using my local systems version of Python 2.

I ended up locating the correct gtop library form my local systems Python 2 installation, and manually transferred it to my virtual environment. It was then revealed through listing my pip installation packages that I was running pygtop[21], which is different to the one required.

4.3 AWS Authorisation

During my testing phase and hoping between different operating systems through AWS, I found myself needing to create a macOS instance to ensure universality of my Docker image. I discovered that due to the way the physical Mac machines are being distributed, AWS requires users to fill in a form and

[20] https://aws.amazon.com/free/
[21] https://pypi.org/project/pygtop/

enter a queue to be reviewed for a quota increase of the available Mac machines they can use to create macOS instances.

Although the process was painless, it was lengthy and required an exchange of about 3 emails over a working week before I was granted my quota of a single Mac machine. It was also made clear that in order to rent a Mac machine from Amazon's servers, an extra Dedicated Host instance would need to be created for a minimum of 24 hours. This was not as much of a trouble to create, as it was financially.

4.4 Language Stubbornness

As mentioned, this project is meant to be primarily used as a teaching tool. This means that for most of the project's duration, the participant is to be studying and applying mostly known concepts in unknown languages. A large "unknown factor" such as this is bound to be associated with its own caveats.

Due to a subtle bug in the GoLang implementation of a Set, I was stuck on the same problem for about one working week. This costed me greatly and is what, I believe, cost me to only reach the one algorithm mark. Reflecting on this, I should have called it a special case or mitigating circumstance and would have either moved on to the next language for the prime sieve, or the next algorithm altogether starting with GoLang as my first "exotic" language.

5 Achievements

This section focuses on the achievements that contributed to the original repo, have personal significance as a milestone to my upcoming career as a software engineer, or I find to be worthy of mentioning.

5.1 Original repository

At the start of the academic year, I began by downloading and running the original Computer Language Benchmarks Game, but to contribute I needed to ask for permission to create an account in the repositories GitLab site. After being granted access to the site I got into contact with the original author and current maintainer of the repository with no luck until April 2022, where the author replied to my original email commenting on the lack of documentation and agreed to allow me to assist in writing a new, more verbose, version of it. One of the biggest achievements I managed during this academic year is the author committing[22] one of my suggestions communicated through an email thread.

5.2 Universality

A huge foundational step for this project is providing a singular source of truth that can run on most machines with practically no set up. The published Docker image contains the entire project source code and can be easily run and customised using Docker commands. The image can be run on most, if not all, cloud instances, and any machine that supports Docker.

5.3 Personal milestones

I do believe that I managed to grow significantly as a programmer and a software engineer through this project. I was able to explore new languages and technologies that I would have otherwise not had the chance to through my university modules. Rust and GoLang, as mentioned before, are emerging technologies that companies seem to favour in candidates, as well as Docker and AWS being an industry standard that I have not explored during my academic career.

[22] https://salsa.debian.org/benchmarksgame-team/benchmarksgame/-/commit/3e4991fab

6 Conclusion and Improvements

Looking with a bird's eye view over all the accumulated results, it is safe to deduct that this project could be renamed to "Which language can score closer to C". This conclusion is suggested by both the original and the prime sieve repository. Different posts on Stack Overflow or in verbal conversations suggest that a low-level language like C will represent data structures and variables differently from higher-level languages; hence its compiler can optimise[23]. This could explain why fundamental data structures are not natively supported in C and need to be introduced by the programmer, it was meant to be as clean as possible or "bare bones".

Going forwards, although the project is in a stable and working state, there are a few features I would like to add or change:

- Find or create a more accurate and verbose usage profiler to capture individual thread usage.

Despite the current usage profiler seems to be competent enough for a generic comparison of implementations and languages, it could provide a more verbose version of the data. The most important one being thread-by-thread utilisation, rather than an average CPU package report.

- Introduce analysis software for low-level languages (e.g., valgrind[24] for C).

Some languages that have manual memory allocation and deallocation usually require the assistance of analysis tools to ensure that program conventions have been followed and there are no memory leaks. This does not necessarily affect the performance or the score of the algorithm but since this is meant as a teaching tool, conventions must be followed.

- Add the source code size after comment purging and GZip compression.

This is inspired by the original repository which also accounted for the compressed size of a comment-purged version of the source code of each implementation.

- The repository's file structure should be changed from the "language to algorithm" pattern to an "algorithm to language" pattern to accommodate a more grammatically semantic approach.

This will make it easier to introduce an "algorithm" flag to the benchmarking script to run individual algorithms.

- Update and maintain documentation for all tools required and usage of the repository's scripts.

- Testing and test coverage should be enforced on all languages and algorithms.

As mentioned in Section 2.4.2 testing is an extremely important aspect of programming and a good practice. This should be enforced and taught in the earlier stages of someone's career as it can directly affect their learning path and how they approach a problem.

- The benchmarking script should become more robust and less prone to user error.

- The Docker image should become more lightweight and use existing Docker images of each language for robustness.

The Docker image will currently cost a user about 2GB worth of storage as it is a single Ubuntu base image whose originally size is about 30MB but builds and installs all required dependencies. Breaking this down into multiple Docker images that are managed by a parent Ubuntu image can be much more lightweight and provide more versatility.

[23] https://stackoverflow.com/a/9375783/5817020
[24] https://valgrind.org/

- Autoformatting should be introduced for all languages, and if not available, a link to conventional syntax documentation should be supplied.

Additionally, taking an example from the Prime Sieve project, the benchmark script could be used to check how many times each language can execute an algorithm with the same input within a certain period. This method comes with its own caveats such as relying on a computer's internal clock to accurately stop on the given threshold. Any extra time allocated could prove to be the turning point between two competing languages.

Referring to Edward Walker's article once more, it has been interesting to see how well cloud computing has evolved and that this data could be helpful to a freelancer, small company, or big corporation to save on maintenance and running costs. It was a generally painless procedure to go from nothing to launching an AWS EC2 instance, but due to the short amount of time I have used it I cannot make an absolute comment on the difference in maintenance of a physical system to an AWS cloud instance. With today's interconnected online world, it is safe to assume that most people, especially those who code for a living, have access to an internet connection. This gives the potential of a world of lightweight home machines that offload microservices and micro-tasks to cloud computing servers that are operational regardless of whether they are in use. Allocating these servers to customers means that less energy is being wasted by the consumer and the corporation hosting these servers (although there could be an argument that utilising these servers uses much more energy than they were using when idle). Even though I could not test this against an HPC, comparing these services with an average to the high specked machine should provide enough insight to anyone who wants to investigate further.

7 References

Bagley, D. (2001, January 24). The Great Computer Language Shootout. as seen in the WayBack Machine at https://web.archive.org/web/20010125021400/http://www.bagley.org/~doug/shootout/

The Computer Language 22.03 Benchmark's Game. (2022, March).

Friederici, A. D., Gunter, T. C., Hahne, A., & Mauth, K. (2004, January 19). The relative timing of syntactic and semantic processes in sentence comprehension. NeuroReport.

Garlikov, R. (n.d.). Understanding "Understanding" or What It Means To "Understand" Something? Irondale, Alabama, USA.

Mic, B., Debray, S. K., & Peterson, L. L. (1993, 11 1). Reasoning about naming systems. New York City, USA: Association for Computing Machinery.

Walker, E. (2008, October). Benchmarking Amazon EC2 for high-performance scientific computing. p. 6.