CHAPTER I
INTRODUCTION

1.1. Background

In high-level concurrency system architectures, the demand to process demand volumes instantly has shifted bottleneck performance fundamentally. The limitations of the system are no longer dominated by the inefficiency of the asymptotic time complexity of an algorithm, but rather by architectural limitations such as memory wall and I/O latency [1]. When the computational load and logical operations increase sharply, program execution time is often hampered by high numbers cache miss and overhead memory allocation [2]. Therefore, modern software engineering optimization can no longer rely solely on the reduction of mathematical steps but demands an approach that aligns the structure of the algorithm with the capabilities of the underlying hardware.

This need for alignment has become absolute as modern processor architectures evolve. When the physical speed limit of a single-core processor has been achieved, the semiconductor industry is permanently switching to multi-core processor technology [3]. This shift in hardware architecture underpins parallel computing techniques, in which software is developed to execute complex computing loads simultaneously on top of different physical cores [4]. However, when faced with data-heavy (I/O-bound) network scenarios, pure parallelism loses its efficiency. High wait times (idle) when the system retrieves or sends data causes bottleneck which holds the overall performance of the processor.

To solve the gridlock of such processor utilities, a concurrency approach is implemented. Concurrency improves execution time efficiency by performing context switching on the processor core that is idle to process other tasks in a interleaving [5]. In conventional programming languages, this feature is generally implemented through an extra-level additional library (library-level) that are vulnerable to adding to the complexity of the code and triggering overhead memory allocation [6]. These limitations have triggered a fundamental shift in modern programming language architectures towards the provision of built-in concurrency features (language-level built-in). This built-in approach is designed so that thread

management abstractions can be controlled directly by runtime language, relieving the burden of dependency from operating system-level threads so that execution becomes much lighter, safer, and more efficient [7].

Comparative analysis of the concurrency capabilities of modern programming languages has been evaluated through specific tests between Go and Java using matrix multiplication algorithms and PageRank [8]. The study proved Go's advantages on compilation speed and small workloads, while Java is more resilient when it comes to handling drastic surges in compute load. However, this comparison leaves an architectural inequality because Java relies on execution bytecode on top of the JVM, while Go is directly compiled into a standalone machine language [8]. To achieve a more equal comparison at the basic level (bare-metal), this study replaces Java with the Rust Programming Language. As a fellow compiled language which translates directly to the level of a machine like Go, the performance comparison between Go and Rust is expected to be able to present a fairer and more representative concurrency efficiency metric.

The Go Programming Language introduced by Google in 2009 is designed to optimize compilation speed and maintain syntactic simplicity [9]. The main attraction of the Go lies in the built-in concurrency model in the form of a lightweight Goroutine that is autonomously managed by scheduler for the execution of thousands of tasks that are much more efficient than thread Conventional Operating Systems [8]. Meanwhile, the Rust Programming Language came into existence in 2010 by the Mozilla Foundation [7], as a language statically typed that focuses on ensuring the security of data types (type safety), concurrency performance, and strict memory management that can secure memory allocation without overloading the system with garbage collection processes (garbage collector) [10]. Through the principle of ownership and data borrowing that is statically evaluated at compilation, Rust promises high-performance concurrent compute execution while being free from the threat of data race conditions.

As a follow-up to previous research, performance testing that focuses only on the execution of purely mathematical computations is often limited to measuring processor capacity (CPU-bound) [11]. Such a pure approach does not yet represent real workloads because modern system architectures are more often hampered by

the volume of data traffic (I/O-bound) rather than CPU speed. Meanwhile, the current industry trend is mostly implementing Go Programming Languages [12] than Rust [13] to handle massive data interactions. The use of both languages is generally focused on the development of critical infrastructure such as architecture Servers network and REST API services. Therefore, the scope of concurrency performance testing in this study needs to be expanded into network simulations to record the characteristics of data queue constraints in a representative manner.

Finding specific patterns within a text stack is one of the most resource-intensive REST API operational computational processes. Empirical data show that matching string can consume 70% to 80% of the processor's lifecycle and become a drag (bottleneck) most critical when data traffic grows [14]. To get around these high I/O queue constraints and CPU loads, modern architectures shift the text search process to the application service level by implementing matching algorithms string advanced. Through this approach, the application of specific algorithms for long texts such as Aho-Corasick and Knuth-Morris-Pratt (KMP) on REST API services has been proven to be able to produce a much higher and more stable level of execution efficiency [15].

Based on this urgency, this study intends to conduct a comparative study of the concurrency performance of the Go and Rust Programming Languages holistically. The test is performed by dividing the characteristics of the workload into two main domains. First, a pure numeric computation scenario to measure the upper limit of the processor's arithmetic capacity (CPU-bound) through Matrix Multiplication and PageRank algorithms. Second, a hybrid network server scenario to represent the robustness of data traffic queue management (I/O-bound) integrated internal calculation (CPU-bound) through the implementation of a REST API server based on the Aho-Corasick text search algorithm.

1.2. Problem Statement

Based on the background that has been described, the problems that will be raised in evaluating the performance comparison between the Go and Rust Programming Languages are formulated as follows:

1. How does the performance of pure computing execution time (CPU-bound) compare between Go and Rust programming languages in executing

embarrassingly parallel matrix multiplication algorithms and communication-intensive PageRank algorithms simultaneously in pure computing scenarios?

2. How do response time performance and throughput compare as a representation of network workloads (I/O-bound) when the two programming languages act as REST API servers that execute the Aho-Corasick text search algorithm concurrently?
3. How does the compile time efficiency and the final size of the compiled binary file generated by the compiler of the two programming languages compare across test scenarios?

1.3. Research Objectives

In accordance with the formulation of the problem determined, this research was carried out with the following objectives:

1. Measure and compare the performance of pure compute execution time (CPU-bound) between Go and Rust Programming Languages in executing matrix multiplication algorithms and PageRank algorithms concurrently in pure computing scenarios.
2. Measure and compare the robustness and stability of response time and throughput productivity as a representation of network load (I/O-bound) when the two languages serve incoming HTTP requests simultaneously as REST API servers executing the Aho-Corasick algorithm.
3. Determine the comparison of compile time efficiency and binary file size of all test application program code generated by Go and Rust compilers.

1.4. Research Benefits

The benefits that can be obtained from the implementation of this research are to provide an empirical foundation, quantitative data, and objective architectural considerations for tech architects and software developers in making choices between Go and Rust Programming Languages. These considerations are based on the real performance characteristics of the two languages when faced with three different spectrums of concurrency workloads: parallel independent computing (Matrix Multiplication), compute with memory-intensive synchronization

(PageRank), and asynchronous network services combined with complex memory calculations (REST API Server with Aho-Corasick algorithm).

1.5. Problem Limitations

To ensure the scope of the experiments and discussion of the research results remains focused, in-depth, and valid, the following problem boundaries were established:

1. Performance testing in a purely computational (CPU-bound) scenario was limited to two algorithms with contrasting characteristics: the traditional matrix multiplication algorithm (representing independent data decomposition) and the PageRank algorithm (representing an iterative graph algorithm with intensive inter-thread communication).
2. Hybrid network (I/O-bound) and internal computational load testing focused on the application's ability to serve concurrent incoming HTTP requests as a REST API server executing the Aho-Corasick multiple-pattern matching algorithm.
3. Network workload simulations (load testing) on the REST API server were conducted using an automated testing tool running locally (localhost) on the same experimental machine to isolate interfering variables such as latency and instability of the external internet network (ISP) and ensure the validity of the throughput and response time data.
4. Test applications built in both programming languages are purely focused on testing the efficiency of the backend architecture behind the scenes. Therefore, they lack a user interface, do not accept direct interactive input from users, and do not involve data storage procedures in a database (persistent storage).
5. Testing and capturing performance metrics (execution time, response time, throughput, compilation time, and binary file size) are conducted through two automated testing stages: functional unit testing and peak load testing using automation scripts.