UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALEXANDRE DE ROSSO CRESTANI

# Uma Comparação Empírica em Velocidade de Processamento entre C++, Go, Rust, Python e JavaScript

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Profa. Dra. Renata Galante

Porto Alegre
2024

*“ The only limit to our realization of tomorrow will be our doubts of today. “*

— FRANKIN D. ROOSEVELT

## AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão à minha família, em particular para minha mãe, Marinei de Rosso, e meu pai, José Moacir Crestani. O apoio incondicional, o amor e a compreensão de vocês foram a base que me permitiu perseguir meus objetivos com foco e determinação. Sem o suporte constante de vocês, este percurso não teria sido possível. Também gostaria de agradecer à Professora Dra. Renata Galante, cuja orientação foi fundamental na revisão do conteúdo e na melhoria da escrita deste trabalho. Um agradecimento especial à Universidade Federal do Rio Grande do Sul (UFRGS), pela oferta de uma educação de alto nível e acessível que abriu caminhos para oportunidades tanto no âmbito profissional quanto pessoal, enriquecendo minha vida com novas amizades que levarei com muito carinho. Finalmente, gostaria de agradecer a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho.

## RESUMO

Devido as restrições físicas crescentes e aos custos elevados associados a aquisição e manutenção de componentes de *hardware*, a indústria de *software* tem se concentrado na seleção de ferramentas e linguagens de programação que maximizem a eficiência dos recursos existentes. Na atualidade, com a crescente demanda por *softwares* mais sofisticados e eficientes, especialmente para atender a uma base de usuários em constante expansão na Internet, a busca constante por métodos inovadores que facilitem o desenvolvimento é intensificada. Nesse contexto, linguagens de programação inteiras são continuamente atualizadas (Python 3.12.1, C++ 23, Go 1.21 e Rust 1.75 em 2023) ou mesmo criadas (Go em 2009 e Rust em 2015) com finalidade de atender essas demandas da comunidade. Diante desse cenário dinâmico, torna-se crucial realizar um estudo comparativo entre algumas dessas opções. Este trabalho tem como objetivo comparar cinco das linguagens de programação mais utilizadas para situações diversas: C++, Go, Rust, Python e JavaScript. Ao analisar essas linguagens, concentramos nossa atenção na velocidade de processamento e no subjetivo relato de experiência do autor ao decorrer do trabalho quanto à facilidade de escrita, leitura e manutenção de códigos nas diferentes linguagens. Operações alvo dessa análise são: chamadas de funções vazias, iterações em laços, calculos aritiméticos simples, chamadas de funções recursivas simples de complexidade $O(n)$, chamadas de funções recursivas com complexidade $O(3^n)$ e, por fim, manipulação de arquivos e *strings* com testes de leitura e escrita com tradução de números inteiros para *strings* e vice versa. Essa abordagem nos permite compreender melhor qual linguagem oferece a melhor experiência, desempenho e eficiência no uso de recursos de *hardware*. Contrariamente ao esperado, C++ não foi o mais performático em vários testes, sendo superado por Rust, que mostrou excelentes resultados devido ao seu compilador eficiente. Go e JavaScript tiveram desempenhos satisfatórios em suas áreas específicas. Python, no entanto, desapontou, exibindo altos tempos de execução e falhando em alguns testes por uso excessivo de recursos. Este estudo sugere Rust e C++ como as melhores escolhas para aplicações de alto desempenho, embora o detrimento de um pelo outro possa se justificar pela aplicação alvo a ser desenvolvida. Go e JavaScript se destacaram pelo equilíbrio entre simplicidade e desempenho dentro de seus nichos. Python mostrou-se menos adequado para sistemas fora do ensino ou uso didático.



**Palavras-chave:** Desempenho. C++. Go. Rust. Python. JavaScript.

**An Empirical Comparison in Processing Speed between C++, Go, Rust, Python, and JavaScript**


## ABSTRACT

Due to increasing physical constraints and the high costs associated with the acquisition and maintenance of hardware components, the software industry has focused on selecting tools and programming languages that maximize the efficiency of existing resources. Currently, with the growing demand for more sophisticated and efficient software, especially to meet an ever-expanding user base on the Internet, the constant search for innovative methods that facilitate development is intensified. In this context, entire programming languages are continuously updated (Python 3.12.1, C++ 23, Go 1.21, and Rust 1.75 in 2023) or even created (Go in 2009 and Rust in 2015) to meet these community demands. Given this dynamic scenario, it becomes crucial to conduct a comparative study among some of these options. This work aims to compare five of the most widely used programming languages for various situations: C++, Go, Rust, Python, and JavaScript. In analyzing these languages, we will focus our attention on processing speed and the subjective experience report of the author throughout the work regarding the ease of writing, reading, and maintaining code in different languages. Operations targeted in this analysis are calls to empty functions, iterations in loops, simple arithmetic calculations, calls to simple recursive functions of complexity $O(n)$, calls to recursive functions with complexity $O(3^n)$, and finally, file and string manipulation with reading and writing tests, including the translation of integers to strings and vice versa. This approach allows us to better understand which language offers the best experience, performance, and efficiency in the use of hardware resources. Contrary to expectations, C++ was not the most performant in various tests, being surpassed by Rust, which showed excellent results due to its efficient compiler. Go and JavaScript had satisfactory performances in their specific areas. Python, however, was disappointed, displaying high execution times and failing in some tests due to excessive resource use. This study suggests Rust and C++ as the best choices for high-performance applications, although the detriment of one for the other may be justified by the target application to be developed. Go and JavaScript stood out for their balance between simplicity and performance within their niches. Python proved to be less suitable for systems outside of teaching or didactic use.



**Keywords:** Performance, C++, Go, Rust, Python, JavaScript.

## LISTA DE FIGURAS

## LISTA DE ALGORITMOS

## LISTA DE TABELAS

**SUMÁRIO**

# 1 INTRODUÇÃO

Devido às limitações físicas estarem se tornando cada vez mais restritivas em relação aos avanços na concepção de componentes de hardware e aos altos preços para a aquisição e manutenção dos mesmos (WALDROP, 2016), a indústria - principalmente de *software* - passou a intensificar esforços na escolha de ferramentas e linguagens de programação que otimizam o uso dos limitados recursos disponíveis. Essa transição acarreta implicações significativas para os profissionais que irão não só ter de lidar com essas novas ferramentas mas também terão de saber de mais detalhes quanto às diferentes características que o uso de diferentes linguagens podem proporcionar tanto ao longo do desenvolvimento quanto no resultado final de um projeto de *software*.

Quando tratamos de prós e contras no âmbito de linguagens de programação, programadores, cientistas e engenheiros da computação podem apresentar opiniões diferentes e fortemente definidas embasadas em suas experiências pessoais no uso de cada linguagem. Em contraste, geralmente pouca informação objetiva e de qualidade está disponível sobre os méritos objetivos e relativos de diferentes linguagens.

A literatura científica e de engenharia oferece muitas comparações entre linguagens de programação - de diferentes maneiras e com diferentes restrições. Algumas são discussões puramente teóricas com uma visão científica e abstrata sobre a construção da linguagem, enquanto outras se mostram não mais do que uma amostra das opiniões dos autores. Algumas são *benchmarks* que comparam uma única implementação de um determinado programa com fim de expressar o consumo de recursos para uma dada tarefa. Essas comparações, embora úteis, são um pouco limitadas por fornecerem um dado que é somente válido sobre restrições extremamente rígidas e específicas, o que acabam trazendo um pouco de dúvida quanto à validade na extrapolação dos resultados para outras aplicações.

O presente trabalho visa fornecer algumas informações objetivas e fundamentais sobre rotinas de uso comum e diário nas práticas de programação em diferentes linguagens de modo que essas informações, mesmo que tenham sido adquiridas sobre uma análise pontual, possam ser extrapoladas para outros cenários de uso tendo como base as mesmas linguagens, a saber, C++, Go, Rust, Python e JavaScript. Para essa análise comparativa, a fim de melhor validar as diferenças entre as linguagens, o mesmo conjunto de programas (isto é, conjunto de implementações do mesmo conjunto de requisitos) é considerado para cada linguagem de programação, ainda mantendo equivalência em operação.

Essas linguagens foram escolhidas por ranquear dentre as mais usadas com a finalidade de implementação de aplicações gerais em códigos abertos no Github (GITHUB, 2023) e por abrangerem tanto o campo das linguagens compiladas quanto das interpretadas.

## 1.1 Problema de Pesquisa

A utilização de diferentes linguagens de programação para a manipulação de dados em diferentes tipos de problemas é algo comum na vida de muitos programadores e cientistas de dados. Também é fato que a aquisição e a manutenção de componentes de *hardware* podem representar um gasto expressivo no balanço financeiro de uma empresa de *software* ao tentar manter seus servidores e *data centers*. Além disso, pesquisas na área geralmente acabam focando em algoritmos ou operações muito específicas, de modo que dificilmente são replicadas no cotidiano profissional. Nesse contexto, uma pergunta se torna não só válida como cada vez mais relevante: qual seria o impacto no uso de diferentes linguagens de programação ao executarem programas equivalentes no quesito de gestão e consumo eficiente de recursos de *hardware* para rotinas simples e comuns?

## 1.2 Objetivo

O objetivo deste trabalho é realizar uma análise comparatória entre o desempenho das linguagens C++, Go, Rust, Python e JavaScript ao executar rotinas simples e comumente usadas, definindo-as aqui, para o escopo deste trabalho, como: chamadas de funções vazias, iterações em laços, calculos de multiplicação e divisão, chamadas de funções recursivas simples de complexidade $O(n)$, chamadas de funções recursivas com complexidade $O(3^n)$, e leitura e escrita de arquivos junto com tradução de números inteiros para *strings* e vice versa, onde o número sendo escrito e lido do arquivo tem de estar no formato de *string* e deve ser transformado em um inteiro para ser usado pelo programa.

## 1.3 Metodologia

O estudo deve ser executado tentando manter as maiores condições de equivalência possível, distribuindo os cálculos de manipulações a serem testadas em funções que tem suas execuções feitas de modo isolado e em ambiente controlado (BADGETT;

MYERS et al., 2023). Espera-se, assim, ser possível medir com maior precisão o impacto da linguagem escolhida ao abordar diferentes problemas enfrentados cotidianamente numa aplicação genérica padrão.

Para melhor salientar as diferenças entre essas linguagens de programação, buscar-se-á ranquear as mesmas em dois fatores, sendo um deles objetivo, embasado em dados obtidos de forma empírica, e um deles subjetivo, baseado na experiência do autor ao escrever os casos de teste.

- **Fator 1:** Tempo de execução do teste proposto (objetivo - quanto menor, melhor);
- **Fator 2:** Facilidade na escrita e no uso eficiente dos recursos da linguagem para uma experiência agradável e produtiva (subjetivo às notas do autor).

Ao final desta análise, espera-se que o leitor possa estar munido de informações relevantes para habilitá-lo na escolha da linguagem de programação que melhor satisfaça as necessidades de um dado produto de *software* a ser implementado.

## 1.4 Justificativa

A justificativa para este estudo reside na necessidade crescente de compreender como diferentes linguagens de programação afetam o uso de recursos em sistemas computacionais, um aspecto de fundamental importância para a indústria de *software*. Em um cenário onde os limites físicos dos componentes de *hardware* tornam-se cada vez mais restritivos e o custo associado à aquisição e manutenção de *hardware* continua a ser uma preocupação significativa, a eficiência no uso dos recursos disponíveis é um fator crítico.

Este estudo é vital por várias razões. Primeiramente, ele procura preencher a lacuna entre as experiências subjetivas dos profissionais de TI e as análises objetivas sobre as linguagens de programação. As opiniões individuais, embora valiosas, podem ser tendenciosas e não refletir completamente o desempenho real das linguagens em diferentes contextos. Portanto, a análise objetiva proposta neste trabalho oferece uma base mais sólida para a tomada de decisões informadas.

A escolha da linguagem de programação não afeta apenas o desempenho técnico, mas também tem implicações econômicas significativas. Uma escolha inadequada pode levar a um consumo excessivo de recursos, aumentando os custos operacionais e reduzindo a competitividade da empresa no mercado. Ao fornecer uma comparação empírica e detalhada de linguagens populares, como C++, Go, Rust, Python e JavaScript, este es-

tudo oferece uma visão valiosa para a tomada de decisões mais embasadas que podem levar a economias significativas e a melhorias no desempenho.

Por fim, a pesquisa aborda uma questão de crescente interesse na comunidade de tecnologia: como diferentes linguagens de programação se comportam sob cargas de trabalho semelhantes. Em um mundo onde a eficiência de recursos e a otimização de desempenho são cruciais, entender essas diferenças pode ser a chave para desenvolver soluções mais robustas e eficientes. Ao fornecer uma análise comparativa detalhada, este estudo contribui para um entendimento mais profundo das forças e fraquezas de cada linguagem considerada, capacitando os profissionais a fazer escolhas mais informadas e estratégicas em seus projetos de *software*.

## 1.5 Organização do Texto

O restante do texto está organizado da seguinte forma. O Capítulo 2 traz conceitos importantes para o entendimento do trabalho, apresentando algumas das mais relevantes definições e contextualizações necessárias. O Capítulo 3 faz uma apresentação em paralelo à comparações entre a proposta deste trabalho e de trabalhos que abrangem assuntos relacionados. O Capítulo 4 entra com mais detalhes sobre a metodologia, as implementalções e a avaliação experimental prática realizada. O Capítulo 5, por vez, busca trazer uma avaliação mais aprofundada dos resultados observados. Por fim, o Capítulo 6 traz considerações finais sobre os testes realizados e os resultados obtidos, além de ideias e discussões sobre possíveis trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, o enfoque central é a contextualização da evolução da computação ao longo da história. Não apenas se definem aspectos superficiais, mas também se explora profundamente as razões que impulsionam o surgimento e desenvolvimento de cada elemento. As linguagens tidas como alvo desta pesquisa recebem uma seção específica que explora suas características distintas, fornecendo detalhes técnicos e analisando as vantagens e desvantagens que um programador pode antecipar ao se deparar com cada uma delas.

A Seção 2.1 apresenta uma contextualização histórica a respeito da concepção e evolução das linguagens de programação como um todo. As Seções 2.2 e 2.3 apresentam uma contextualização junto à definição dos termos ”linguagem compilada” e ”linguagem interpretada”, respectivamente.

## 2.1 Contextualização Histórica

A história da computação é uma jornada fascinante que abrange séculos de inovação, desde os primórdios dos dispositivos mecânicos até os avançados sistemas computacionais que permeiam nossa vida cotidiana. A ciência e a engenharia da computação evoluíram em paralelo com a criação e aprimoramento das linguagens de programação, desempenhando papéis cruciais na transformação da sociedade e impulsionando o progresso tecnológico (PARSONS, 2019).

Nos seus primórdios, máquinas mecânicas como a Máquina Analítica de Charles Babbage no século XIX (HAECKER, 2022) e a Máquina de Turing proposta em 1936 por Alan Turing (MOL, 2018) estabeleceram os fundamentos teóricos da computação. Contudo, foi somente por volta dos anos de 1940 que os primeiros computadores eletrônicos, como o ENIAC (Electronic Numerical Integrator and Computer) (O’REGAN; O’REGAN, 2018), emergiram, sendo pioneiras e essenciais para a resolução de cálculos complexos, acelerando processos que antes demoravam meses, marcando então uma nova era na história da computação.

Paralelamente ao desenvolvimento de *hardware*, a necessidade de interação mais eficaz e acessível com os computadores levou ao surgimento das linguagens de programação. Visando a facilitação e o aumento na efetividade e velocidade do ato de programar, a evolução das agoras nascidas linguagens de programação foi constante e acelerada, desde

linguagens de baixo nível como Assembly em 1940, até linguagens conhecidas como de alto nível em suas épocas, como Fortran, COBOL e LISP em 1950.

A década de 1970 foi, por sua vez, marcada na história com o advento dos computadores pessoais, tendo empresas como Apple, Microsoft e IBM liderando a revolução. Esses computadores pessoais, como o Apple II e o IBM PC, trouxeram a computação para os lares e escritórios, democratizando o acesso à tecnologia (HARTIGH et al., 2016). Concomitantemente, o desenvolvimento de sistemas operacionais como o Unix, criado por Ken Thompson e Dennis Ritchie nos Laboratórios Bell (SPINELLIS; LOURIDAS; KECHAGIA, 2016), também teve um impacto significativo. O Unix contribuiu para a disseminação de padrões e práticas na programação, influenciando gerações de sistemas operacionais subsequentes.

No campo das linguagens de programação, a década de 1970 também testemunhou a criação da linguagem C por Dennis Ritchie nos Laboratórios Bell. O C, lançado oficialmente em 1978 com o livro *'The C Programming Language'*, escrito por Ritchie e Brian Kernighan (KERNIGHAN; RITCHIE, 1978), rapidamente se tornou uma linguagem de programação amplamente adotada. Sua eficiência e portabilidade foram fundamentais para o desenvolvimento de muitas distribuições do sistema operacional Unix e, posteriormente, para muitos outros projetos.

Além disso, o início do desenvolvimento da linguagem de programação C++, uma extensão da linguagem C com recursos de programação orientada a objetos. Criada por Bjarne Stroustrup, o C++ foi lançado no início da década de 1980 com o livro *'The C++ Programming Language'* (STROUSTRUP, 1985) e se tornou uma linguagem de programação poderosa e versátil, influenciando muitas outras linguagens que surgiram posteriormente e sendo amplamente usada e atualizadas até os dias de hoje.

A evolução das linguagens de programação não apenas acompanhou, mas também impulsionou a rápida expansão da computação em diversos setores. Desde a revolução dos computadores pessoais até os sistemas distribuídos e a ascensão da Internet, as linguagens de programação desempenharam um papel fundamental, permitindo que desenvolvedores expressassem suas ideias de maneira eficaz e eficiente, contribuindo assim para a complexa tapeçaria da era digital em que vivemos.

Avançando no tempo e agora retratando o presente, linguagens como Go, Rust, Python e JavaScript refletem a diversidade e a adaptabilidade contínua no campo da programação. Cada uma delas atende a necessidades específicas e contribui para a inovação tecnológica em suas respectivas áreas de aplicação. O panorama em constante evolução

das linguagens de programação continua a moldar a forma como concebemos e desenvolvemos *software*, destacando a importância de acompanhar as tendências e os avanços nesta fascinante jornada da computação.

## 2.2 Linguagens Compiladas

A história das linguagens compiladas remonta aos primórdios da programação de computadores. Antes do surgimento das linguagens compiladas, os programadores precisavam escrever códigos diretamente em linguagem de máquina ou em linguagens assembly, uma tarefa complexa e facilmente propensa a erros. Com o tempo, os desenvolvedores começaram a perceber a necessidade de ferramentas que facilitassem a programação e aumentassem a eficiência na criação de *software*.

A ideia por trás das linguagens compiladas é traduzir o código-fonte escrito pelo programador em uma linguagem de alto nível para o código de máquina específico da arquitetura do computador de destino. Esse processo é chamado de compilação e resulta na criação de um programa executável que pode ser diretamente executado pelo computador sem a necessidade do código-fonte original.

No entanto, foi somente com o desenvolvimento do Fortran (FORmula TRANslation), na década de 1950, que as linguagens compiladas ganharam verdadeira popularidade. O Fortran foi uma das primeiras linguagens de programação de alto nível e foi projetado especificamente para cálculos científicos e engenharia. Ele permitia que os programadores expressassem algoritmos em termos mais próximos do pensamento humano, facilitando a programação para uma gama mais ampla de profissionais.

Assim, podemos entender uma linguagem de programação compilada como sendo aquela em que o código-fonte, escrito pelo programador, é traduzido integralmente para código de máquina ou código intermediário por um programa chamado compilador antes da execução do programa. Essa tradução é o que chamamos de compilação (SARTORELLO, 2022).

O compilador analisa o código-fonte, verifica sua sintaxe e semântica, e o converte em uma forma que pode ser executada pelo *hardware* do computador ou por uma máquina virtual. O resultado desse processo é geralmente um arquivo executável ou um código intermediário que pode ser executado por um interpretador ou uma máquina virtual. O processo de compilação oferece várias vantagens, incluindo:

- **Desempenho Otimizado:** O código compilado é traduzido para a linguagem de máquina, o que muitas vezes resulta em um desempenho mais eficiente em comparação com linguagens interpretadas. Além disso, compiladores modernos tem a capacidade de fazer um refinamento no processo de otimização do executável gerado, possibilitando uma eficácia ainda maior do programa e ao mesmo tempo não exigir um esforço equivalente por parte do programador.
- **Detecção de erros:** Muitos erros podem ser identificados durante a fase de compilação - variando entre linguagens e compiladores para a mesma linguagem. Isso ajuda os desenvolvedores a corrigir e previnir problemas antes que o programa seja executado.
- **Proteção do código-fonte:** Como o código-fonte é traduzido para uma forma que não é diretamente legível por humanos e, mesmo com a ajuda de *decompilers*, a lógica do programa torna-se mais difícil de ser compreendida por pessoas não autorizadas.
- **Independência da plataforma:** Uma vez compilado, o código pode ser executado em qualquer máquina que tenha compatibilidade com o executável gerado.

Exemplos de linguagens de programação compiladas incluem C, C++, Go, Rust e Fortran. Cada uma dessas linguagens possui um compilador dedicado que traduz o código-fonte para código de máquina ou código intermediário, tornando-as adequadas para uma variedade de aplicações, desde sistemas de baixo nível até desenvolvimento de *software* de grande escala.

### 2.2.1 C++

Diferentemente da época em que foi criada, hoje a linguagem de programação C++ é considerada por muitos como uma linguagem de baixo nível. Sua história data do início dos anos 80, quando a linguagem de programação C foi estendida com recursos de programação orientada a objetos, dando origem ao C++. O nome ”C++” foi inspirado na operação de incremento ”++” presente na linguagem C.

O C++ tem se destacado no cenário de desenvolvimento de *software* devido a suas características únicas que o diferenciam de outras linguagens no mercado. Uma das principais características é sua capacidade de oferecer tanto programação orientada a objetos quanto programação de baixo nível, proporcionando aos desenvolvedores uma

flexibilidade notável (CPLUSPLUS, 2024).

A linguagem C++ também é conhecida por sua eficiência e controle direto sobre a memória, tornando-a uma escolha popular para o desenvolvimento de sistemas embarcados, drivers de *hardware* e *softwares* que exigem otimização de recursos. C++ se mostra tão performática e de relativo fácil acesso que acabou sendo também uma das linguagens mais usadas com propósitos de avaliar o desempenho de outras linguagens em *benchmarks* e análises comparativas.

Pode-se ainda destacar:

- **Gerenciamento de Memória**: Permite controle detalhado sobre a alocação e desalocação de memória, utilizando operadores como `new` e `delete`.
- **Biblioteca Padrão**: Inclui a Standard Template Library (STL), uma biblioteca fruto de uma comunidade madura e extremamente focada na evolução da linguagem ao longo de anos que fornece uma abordagem extremamente abrangente de estruturas de dados, algoritmos e outros utilitários e genéricos.
- **Compilação**: Como uma linguagem compilada, C++ geralmente resulta em código de máquina mais rápido em comparação com linguagens interpretadas. O compilador de C++ ainda fornece *flags* de otimização adicionais para dar ainda mais controle sobre o programa gerado ao programador, possibilitando otimizações e aperfeiçoamentos ainda mains finos.
- **Desempenho**: É conhecida por seu alto desempenho, sendo uma escolha popular para desenvolvimento de sistemas e *softwares* que exigem eficiência.
- **Complexidade**: C++ é considerada uma linguagem mais complexa, exigindo compreensão aprofundada de conceitos como gerenciamento de memória e ponteiros.

### 2.2.2 Go

Go, também conhecida como Golang (GOOGLE, 2024), é uma linguagem de programação de alto nível e moderna, desenvolvida no final dos anos 2000 pela Google sob a liderança de Robert Griesemer, Rob Pike e Ken Thompson. Criada para ser eficiente, concisa e fácil de usar, Go atende às necessidades de desenvolvimento de *software* em larga escala.

Embora compartilhe a simplicidade e eficiência do C, Go se destaca por suas inovações destinadas às demandas contemporâneas de desenvolvimento de *software*. Sua

abordagem em relação à concorrência e à manutenção da simplicidade, sem sacrificar a eficiência, por exemplo, são algumas de suas características mais notáveis e motivos pelos quais a linguagem é escolhida como base para projetos.

A seguir, detalhamos os principais aspectos de Go:

- **Gerenciamento Automático de Memória**: Inclui um coletor de lixo eficiente, simplificando a gestão de memória e reduzindo a probabilidade de vazamentos.
- **Concorrência**: O modelo de concorrência de Go, utilizando *goroutines* e *channels*, facilita a criação de programas concorrentes de forma extremamente fácil e eficaz, ideal para sistemas escaláveis e serviços Web com muitas requisições simultâneas.
- **Manutenção e Gerenciamento de Erros**: Go adota uma metodologia distintiva para o manejo de erros que ocorrem durante a execução de programas. Esta estratégia rigorosa exige que o programador esteja consciente das potenciais fontes de erro e os trate especificamente. Essa característica da linguagem é simultaneamente admirada e criticada por diferentes programadores, refletindo um espectro de opiniões sobre sua eficácia e praticidade.
- **Simplicidade de Sintaxe**: A linguagem é conhecida por sua sintaxe simplificada, que facilita a leitura e a manutenção do código.
- **Abordagem Minimalista com Bibliotecas**: Go oferece uma biblioteca padrão que, embora abrangente, não apresenta tantas funcionalidades como as presentes em linguagens mais maduras como C++ ou Python.

Go, portanto, estabelece-se como uma linguagem moderna e prática, adaptada às exigências do desenvolvimento de *software* contemporâneo, com foco especial na concorrência e escalabilidade.

### 2.2.3 Rust

Rust (MOZILLA, 2022) é uma linguagem de programação moderna, lançada em 2015 pela Mozilla Research, que combina segurança e desempenho. Projetada como uma alternativa a C e C++, Rust oferece um sistema de tipos forte e um sistema de propriedade exclusivo, minimizando bugs de concorrência e de gerenciamento de memória.

Rust se destaca pelo gerenciamento de memória inovador. Seu sistema de propriedade garante, em tempo de compilação, a segurança do acesso à memória, eliminando vazamentos e violações sem a necessidade de um coletor de lixo.

Rust é notável por suas abstrações de alto nível sem *overhead* significativo, mantendo o desempenho comparável ao código otimizado manualmente. Seu sistema de tipos expressivo facilita a escrita de código robusto e compreensível, e o compilador oferece mensagens de erro detalhadas para correção eficiente de problemas.

A linguagem proporciona uma abordagem segura à programação concorrente utilizando os conceitos de *ownership* (propriedade) e *borrowing* (empréstimo). O conceito de propriedade refere-se à maneira como Rust gerencia a memória através da atribuição de cada porção de memória a uma única variável, garantindo que não haja acessos simultâneos indesejados. Por outro lado, o conceito de empréstimo permite que outras partes do código usem esses dados temporariamente sob regras estritas, evitando erros comuns em programação concorrente. Adicionalmente, Rust oferece o Cargo, um sistema de gestão de pacotes que facilita a administração de dependências e a construção de projetos.

- **Gerenciamento de Memória Inovador**: Utiliza um sistema de propriedade único para evitar erros comuns de gerenciamento de memória, como vazamentos e violações de acesso.
- **Programação Concorrente Segura**: Suporta programação concorrente de forma segura através dos conceitos de *ownership* e *borrowing*, evitando condições de corrida.
- **Abstrações de Zero-Custo**: Oferece abstrações de alto nível que não comprometem o desempenho, mantendo a eficiência do código otimizado manualmente.
- **Sistema de Tipos Forte**: Possui um sistema de tipos expressivo que ajuda na criação de código mais robusto e compreensível.
- **Mensagens de Erro Detalhadas**: O compilador de Rust é conhecido por fornecer *feedback* detalhado e útil, auxiliando na rápida identificação e correção de erros.
- **Sistema de Pacotes Integrado (Cargo)**: Inclui uma ferramenta de gerenciamento de pacotes que facilita a gestão de dependências e a construção de projetos.

## 2.3 Linguagens Interpretadas

Enquanto as linguagens compiladas têm uma longa história que remonta aos primórdios da programação de computadores, as linguagens interpretadas surgiram como uma abordagem alternativa para facilitar o desenvolvimento de *software*. O conceito de linguagem interpretada está intimamente ligado à execução do código-fonte de forma di-

reta, sem a necessidade de uma etapa de compilação prévia.

Em uma linguagem interpretada, o código-fonte escrito pelo programador não é traduzido antecipadamente para código de máquina ou código intermediário. Em vez disso, um programa chamado ”interpretador” lê e executa o código-fonte linha por linha em tempo real. Essa execução dinâmica permite uma interação mais flexível durante o desenvolvimento e elimina a necessidade de gerar um executável separado antes da execução.

O termo ”interpretada” refere-se ao processo em que as instruções escritas pelo programador são interpretadas e executadas diretamente pelo interpretador, sem a criação prévia de um arquivo executável. Isso implica que o código-fonte permanece legível e pode ser alterado durante a execução do programa. Existem diversas vantagens associadas às linguagens interpretadas, incluindo:

1. **Desenvolvimento interativo:** Como não é necessário compilar o código antes de executá-lo, os desenvolvedores podem realizar alterações no código-fonte e ver os resultados imediatamente, facilitando o desenvolvimento interativo.
2. **Facilidade de Depuração:** A depuração em linguagens interpretadas é frequentemente mais fácil, já que os erros podem ser identificados e corrigidos durante a execução do programa, sem a necessidade de recompilar.
3. **Portabilidade:** Por não estar vinculado a uma arquitetura específica, mas sim à versão do programa interpretador, o código-fonte interpretado é mais portátil. Nesse cenário, o interpretador é o encarregado de lidar com as diferenças de *hardware* e sistema operacional.

Exemplos de linguagens de programação interpretadas incluem Python, JavaScript, Ruby e PHP. Essas linguagens são amplamente utilizadas em desenvolvimento Web, automação, *scripting* e outras aplicações onde a flexibilidade e a rapidez no desenvolvimento são prioridades.

### 2.3.1 Python

Python (LUTZ, 2013) é uma linguagem de programação interpretada e de alto nível, amplamente reconhecida por sua simplicidade e legibilidade. Criada nos anos 80 por Guido van Rossum, ela foi projetada para ser fácil de entender e escrever, o que a torna uma escolha ideal para ambos programadores iniciantes e profissionais experientes.

Como os códigos escritos em Python são executados diretamente pelo programa interpretador, não possuem a necessidade de compilação prévia. Isso contribui para um ciclo de desenvolvimento mais rápido, pois permite a execução de programas imediatamente após a escrita do texto.

Outros aspectos notáveis de Python incluem:

- **Tipagem Dinâmica**: Python é uma linguagem de tipagem dinâmica, o que significa que o tipo de uma variável é determinado em tempo de execução, não em tempo de compilação. Isso aumenta a flexibilidade do código, mas também exige um cuidado maior por parte do programador para evitar erros de tipo.
- **Gerenciamento de Memória Automatizado**: Python gerencia automaticamente a memória através de um coletor de lixo, o que reduz a complexidade do código e previne vazamentos de memória.
- **Biblioteca Padrão Abrangente**: A linguagem vem com uma vasta biblioteca padrão, oferecendo módulos para realizar uma variedade de tarefas, desde expressões regulares até comunicação em rede.
- **Multi-paradigma**: Python suporta múltiplos paradigmas de programação, incluindo programação orientada a objetos, imperativa e, em menor grau, funcional.
- **Portabilidade e Interoperabilidade**: Sendo multiplataforma, Python pode ser executado em diversos sistemas operacionais. Além disso, bibliotecas como ctypes e cffi permitem a interação com código em C, facilitando a integração com legados e otimizações de desempenho.
- **Comunidade e Ecossistema**: Python possui uma comunidade ativa e um ecossistema rico em *frameworks* e bibliotecas, como Django para desenvolvimento Web e NumPy para computação científica, que ampliam ainda mais suas capacidades.

Python se estabeleceu como uma linguagem essencial em diversos campos, como desenvolvimento Web, análise de dados, aprendizado de máquina e automação de tarefas, devido à sua facilidade de aprendizado e versatilidade.

### 2.3.2 JavaScript

Frequentemente abreviado como JS, JavaScript (CROCKFORD, 2008) é uma linguagem de programação de alto nível conhecida por ser de extrema importância e fundamental no desenvolvimento Web moderno. Criada nos anos 90 por Brendan Eich, Ja-

vaScript foi inicialmente desenvolvida para adicionar interatividade a páginas Web, mas rapidamente evoluiu para uma linguagem de programação completa e robusta.

JavaScript é executada predominantemente em navegadores Web, tornando-a uma das linguagens mais acessíveis e amplamente utilizadas no mundo. Além disso, com o advento de ambientes como Node.js, JavaScript também se tornou popular no desenvolvimento de servidores e aplicações *backend*.

As principais características de JavaScript incluem:

- **Interpretada e Dinâmica**: JavaScript é uma linguagem interpretada, com suporte para tipagem dinâmica. Isso permite rápido desenvolvimento e prototipagem, embora possa levar a bugs relacionados a tipos em tempo de execução.
- **Manipulação do DOM**: Uma das principais funcionalidades de JavaScript é a capacidade de manipular o Document Object Model (DOM) das páginas Web, permitindo modificar conteúdos e estilos dinamicamente.
- **Eventos e *Callbacks***: JavaScript é fortemente baseada em eventos, tornando-a ideal para desenvolver interfaces de usuário reativas e interativas.
- **Funções de Primeira Classe**: Em JavaScript, funções permitem o uso de técnicas como **callbacks**, funções de alta ordem e programação funcional.
- **Ecossistema Rico**: A comunidade de JavaScript desenvolveu um vasto ecossistema de bibliotecas e *frameworks*, como React, Angular e Vue.js para desenvolvimento *frontend*, e Express.js para o *backend*.
- **Assíncrono e Não-bloqueante**: JavaScript possui um modelo de operação assíncrono e não-bloqueante, especialmente útil para desenvolvimento Web e aplicações que necessitam de alto desempenho em operações de I/O.
- **Node.js**: Com a introdução do Node.js, JavaScript expandiu seu alcance para o desenvolvimento de servidores, proporcionando um ambiente de execução fora do navegador.

JavaScript hoje não é apenas fundamental para o desenvolvimento *frontend* em aplicações Web, mas também é uma peça-chave em muitas outras áreas, incluindo desenvolvimento *backend* e desenvolvimento de aplicativos móveis (com *frameworks* como React Native).

# 3 TRABALHOS RELACIONADOS

Este capítulo dedica-se à exposição e análise de estudos correlatos contemporâneos. São enfatizados, na Seção 3.1, os aspectos notáveis identificados em cada pesquisa. Posteriormente, na Seção 3.2, são estabelecidas as relações destes trabalhos com o estudo atual. É importante mencionar que uma parcela significativa das publicações anteriores enfoca comparações com a linguagem C++, o que reforça a percepção comum de sua superioridade em termos de desempenho e capacidade de gerenciamento de memória de modo efetivo.

## 3.1 Artigos em Destaque

Nesta seção, são apresentados de maneira sucinta os artigos já publicados que abordam temas relevantes ao presente trabalho.

### 3.1.1 Comparação de Desempenho e Facilidade de Escrita

O artigo *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program* (PRECHELT, 2000) analisa 80 versões do programa *phonecode* feitas por 74 programadores em diversas linguagens como C, C++, Java, Perl, Python, Rexx e Tcl. Este estudo compara desempenho, uso de memória, tamanho e confiabilidade do código, destacando que linguagens de *script* são geralmente mais produtivas e rápidas em certas tarefas, embora consumam mais memória que C, C++ ou Java.

Os resultados mostram ainda que a escrita em linguagens de *script* leva metade do tempo e resulta em códigos menores comparados a C, C++ ou Java, sem diferenças significativas em confiabilidade. Embora linguagens de *script* tenham maior consumo de memória e possam ser mais lentas na inicialização em comparação com C e C++, elas podem ser mais rápidas que o Java em fases principais.

O estudo conclui que para certos problemas, linguagens de *script* são alternativas eficientes, considerando produtividade e facilidade de escrita, apesar de algumas limitações em desempenho e uso de memória.

### 3.1.2 C++ vs Python: Análise de Desempenho e Memória

O artigo *Comparative Analysis of C++ and Python in Terms of Memory and Time* (ZEHRA et al., 2020) compara C++ e Python, focando em gerenciamento de memória e eficiência de tempo. O estudo destaca que C++ exige gerenciamento manual de memória, permitindo controle detalhado, enquanto Python utiliza coleta de lixo automática, facilitando a escrita dos códigos para os programadores, mas aumentando o uso de memória e reduzindo levemente a velocidade.

A análise inclui tarefas como criação e ordenação de vetores e manipulação de elementos, demonstrando que C++ supera Python em termos de desempenho e eficiência de memória. Assim, o estudo conclui que Python é mais adequado para iniciantes e projetos com menos demandas de eficiência de processamento, devido à sua facilidade de uso e legibilidade. Por outro lado, C++ é recomendado para aplicações que requerem alto desempenho, oferecendo maior eficiência e controle de memória.

### 3.1.3 C++ vs Python: Desempenho sobre Bibliotecas Dinamicamente Lincadas

O estudo *Programming PHREEQC Calculations with C++ and Python: A Comparative Study* (MüLLER DAVID L. PARKHURST, 2011) compara C++ e Python em desempenho e memória, enfocando no estudo e na modelagem geoquímica com a biblioteca IPhreeqc. Realizado pelo US Geological Survey, ele destaca a maior facilidade e rapidez de programação em Python devido à sua natureza interpretada, que dispensa compilação. Python é indicado para prototipagem rápida e acessível a programadores menos experientes, enquanto C++ exige mais conhecimento técnico.

Apesar dos modelos em Python terem tempos de execução ligeiramente superiores aos de C++, eles são mais adaptáveis a diferentes plataformas sem necessidade de alterações. A eficiência da API IPhreeqc é ressaltada na simplificação da programação de modelos de transporte reativo.

O trabalho conclui que Python é preferível a C++ pela facilidade de uso, manutenção e portabilidade, apesar de tempos de execução maiores. Estas vantagens são especialmente relevantes no caso em questão, onde a experiência em programação por parte dos geólogos pode ser limitada, tornando Python uma opção mais prática.

### 3.1.4 C++ vs JavaScript: Comparação de Desempenho em Manipulação de Vetores

O artigo *Performance Comparison of C++ and JavaScript (Node.js – V8 Engine)* (STEFANOSKI; KARADIMCHE; DIMITRIEVSKI, 2019) analisa as diferenças de desempenho entre C++ e JavaScript em tarefas como inserção em *arrays* dinâmicos e algoritmos de busca e ordenação. Os resultados mostram que C++ geralmente tem melhor desempenho. Na inserção em *arrays*, C++ é ligeiramente mais rápido, mas JavaScript se destaca em cenários com grande volume de dados. Para busca linear, C++ é consistentemente mais rápido, favorecendo tarefas com intensa manipulação de dados. Na comparação de algoritmos de ordenação, JavaScript supera C++ no Selection Sort, enquanto C++ é mais eficiente no Radix Sort, especialmente em *arrays* menores.

O estudo conclui que C++ é superior em tarefas que exigem mais processamento, mas JavaScript é ainda uma alternativa viável, particularmente em desenvolvimento Web com demandas de entrada e saída de dados. A análise é útil para programadores na escolha entre C++ e JavaScript, considerando desempenho, facilidade de desenvolvimento e contexto de aplicação.

### 3.1.5 C++ vs Rust: Comparação de Desempenho em Algoritmos de Ordenamento

O estudo *Is Rust C++-fast? Benchmarking System Languages on Everyday Routines* (IVANOV, 2022) compara C++ e Rust, focando em algoritmos de ordenação como Merge Sort e Insertion Sort, e operações em dicionários. O objetivo é avaliar a aplicabilidade prática das linguagens em programação cotidiana.

Na análise de ordenação, observa-se variação no desempenho do Merge Sort e Insertion Sort conforme o tamanho dos dados, sendo influenciada pela linguagem. Algoritmos de ordenação híbridos demonstram superioridade em ambas as linguagens, com um desempenho particularmente eficaz em C++. Nas implementações otimizadas, C++ supera Rust.

Em termos de estruturas de dados, mapas *hash* mostram-se mais eficientes do que árvores binárias para operações de inserção e exclusão, com C++ realizando essas operações mais rapidamente. O estudo ressalta a necessidade de pesquisas futuras com dados não aleatórios para uma análise mais representativa. Esta pesquisa oferece *insights* detalhados sobre o desempenho de C++ e Rust em algoritmos e estruturas de dados comuns, destacando as diferenças baseadas na linguagem e na implementação específica.

### 3.1.6 C vs Rust: Comparação de Concorrência

O artigo *A Comparison of Concurrency in Rust and C* (PFOSI RILEY WOOD, 2019) analisa a concorrência em Rust e C, focando na eficácia, segurança e facilidade de implementação. Rust se destaca pela garantia de segurança de memória e prevenção de condições de corrida em tempo de compilação, enquanto C oferece um paradigma de concorrência mais tradicional e de baixo nível com Pthreads.

O estudo utiliza *benchmarks* como multiplicação de matrizes, Blackscholes e produto escalar, avaliando desempenho e facilidade de implementação. Os resultados mostram que, apesar de Rust evitar erros comuns de concorrência e não adicionar sobrecarga de *threading*, tende a ser mais lento que C e pode ter comportamentos de bloqueio inesperados. Em desempenho, C alcança melhor escalabilidade com mais *threads*, enquanto Rust varia em eficiência, sendo menos eficaz em tarefas como multiplicação de matrizes.

Rust apresenta uma curva de aprendizado devido ao seu rigoroso sistema de propriedade e regras de concorrência. Isso assegura segurança de memória, mas complica a implementação de algoritmos simples. C, por outro lado, permite uma abordagem mais flexível e direta, mas com menor segurança.

O estudo conclui que Rust é poderoso e seguro para programação concorrente, mas pode ser mais lento e desafiador de implementar em comparação com C. Esta análise é útil para programadores ao escolherem entre Rust e C para projetos que envolvem concorrência, ponderando entre segurança, usabilidade e desempenho.

## 3.2 Relação com este Trabalho

O artigo (PRECHELT, 2000), apesar de datado, apresenta uma perspectiva valiosa sobre a produtividade do programador e o desempenho das linguagens, focando em diversas linguagens e um programa alvo implementado por diferentes programadores. Essa análise ressoa com a ênfase do presente trabalho na eficiência de processamento e na experiência do usuário, preenchendo uma lacuna importante sobre a produtividade do desenvolvedor em diferentes linguagens.

Quanto a comparação de C++ contra linguagens interpretadas, temos os artigos de Zehra (ZEHRA et al., 2020) e Müller (MüLLER DAVID L. PARKHURST, 2011) trazendo ideias e dados contra Python e o trabalho de (STEFANOSKI; KARADIMCHE; DIMITRIEVSKI, 2019) fazendo destaques contra JavaScript. Todos os artigos trazem

ideias valiosas sobre comparações empíricas considerando C++ e essas duas linguagens interpretadas de maneiras diferentes, proporcionando uma completude de casos que ajuda na compreensão deste trabalho.

Já ao compararmos C++ e Rust, os estudos de (PFOSI RILEY WOOD, 2019) e (IVANOV, 2022) oferecem uma comparação direta sobre o desempenho dessas linguagens em cenários não tratados pelo presente trabalho, como análise em gerenciamento de vetores e aplicações concorrentes.

Em resumo, cada trabalho relacionado foi escolhido por sua relevância específica e contribuição para a compreensão do desempenho, eficiência e usabilidade das linguagens de programação estudadas. Juntos, eles ajudam a construir um quadro mais completo e detalhado, fornecendo um contexto necessário para avaliar as conclusões do presente estudo e destacando áreas onde pesquisas adicionais podem ser necessárias.

Tabela 3.1: Comparação de temas abordados sobre tempo de execução

| Foco | PRECHELT | ZEHRA | MULLER | STEFANOSKI | IVANOV | PFOSI | Este Trabalho |
|---|---|---|---|---|---|---|---|
| Aritimética | | | | | | | X |
| Laços | | | | | | | X |
| Vetores | | X | | X | X | | |
| Strings | X | | | | | | X |
| Concorrência | | | | | | X | |
| Arquivos | | | | | | | X |
| Recursões | | | | | | | X |
| Segurança | X | X | X | | | X | |
| Facilidade | X | | X | X | X | X | X |
| Uso de Memória | | X | X | | | X | |

Fonte: Autor

# 4 AVALIAÇÃO EXPERIMENTAL

Neste capítulo, focamos na análise empírica e comparativa entre as linguagens de programação selecionadas. Detalhamos o processo completo, desde a preparação até a execução dos testes, fornecendo informações detalhadas e essenciais para quem desejar replicar ou validar os experimentos. Adicionalmente, explicamos os critérios e metodologias utilizados na elaboração dos testes, incluindo uma discussão aprofundada sobre as razões e as técnicas empregadas em cada etapa da implementação, visando uma compreensão abrangente dos procedimentos e dos objetivos dos testes.

A Seção 4.1 apresenta uma visão geral da configuração utilizada para os testes, detalhando componentes de *hardware* e versões de *softwares* - como emuladores e sistemas operacionais, e configurações fora do padrão de fábrica, quando realizadas. Na Seção 4.2 são discutidas as versões das linguagens de programação escolhidas e as razões por trás dessas escolhas. A Seção 4.3 descreve a arquitetura adotada para acomodar e executar os testes de cada funcionalidade em cada linguagem, justificando as decisões tomadas para a arquitetura final. Finalmente, a Seção 4.4 eferece uma explanação mais detalhada sobre os testes em si, abordando seu propósito, o que pretendem avaliar, por que foram escolhidos e como buscam mensurar a funcionalidade alvo.

## 4.1 Bancada de Testes

Para assegurar um ambiente de teste isolado das operações comuns em um computador *desktop* padrão, os experimentos foram conduzidos em uma máquina virtual utilizando o *software* VirtualBox.

A máquina física hospedeira operava com o sistema operacional Windows 11 Pro versão 22H2 e foi configurada com um processador Ryzen 5 3600x; placa gráfica RX 5700 XT; 16 GB de memória RAM operando a 3200 MT/s em configuração dual channel para otimizar o desempenho com o processador; e um SSD NVMe de 500 GB, com velocidades de leitura e escrita de até 3500 MB/s e 2000 MB/s, respectivamente.

A máquina virtual criada com o VirtualBox na versão 7.0 foi configurada com o sistema operacional Ubuntu 22.04 LTS, numa instalação mínima para garantir um ambiente de teste limpo e eficiente. Esta máquina virtual teve alocação de 4 núcleos de processamento, 8 GB de memória RAM e 100 GB do armazenamento SSD do sistema hospedeiro. Todas as outras configurações foram deixadas como padrão.

## 4.2 Versões das Linguagens Testadas

Os experimentos utilizaram versões atualizadas das seguintes ferramentas e linguagens de programação: para C++ foi usado o compilador g++ (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0; para Go foi usada a versão go1.21.0 linux/amd64; para Rust foram usados rustc 1.75.0 (82e1608df 2023-12-21)1.75 com o gerenciador de pacotes Cargo na versão 1.75.0 (1d8b05cdd 2023-11-20); para Python o interpretador na versão 3.10.12; e para JavaScript o interpretador Node v20.10.0. Essas versões foram selecionadas por estarem dentre as mais recentes disponíveis para o Ubuntu 22.04 LTS na data de 4 de Dezembro de 2023, quando a bancada de testes foi preparada.

Comandos usados para compilação:

- **C++:** `g++ -O3 <input> -o <output>`
- **Go:** `go build <input>`
- **Rust:** `cargo build -release`

Comandos usados para a execução dos programas nas linguagens interpretadas:

- **Python:** `python3 <input>`
- **JavaScript:** `node <input>`

## 4.3 Arquitetura dos Testes

Para assegurar a mínima interferência externa nas execuções dos programas desenvolvidos e na avaliação das suas respectivas funcionalidades, optou-se por implementar cada teste em uma função distinta, dedicada exclusivamente a ele. Esta função de teste é invocada isoladamente pela função principal do programa, permitindo a mensuração precisa do tempo imediatamente antes e após a execução do teste. A estrutura de teste, consistente em todas as cinco linguagens utilizadas, segue o procedimento descrito a seguir no Algoritmo 1.

**Algoritmo 1** Arquitetura base para abrigar as funções de teste

```
function MAIN(argumentos)
    // Analiza entrada para verificar o número de iterações no teste
    iteracoes ← PARSEINPUT(argumentos)

    // Marca ponto no tempo de início do teste
    inicio ← NOW()

    // Chama a função alvo a ser testada e atribui valor retornado ao resultado
    resultado ← FUNCAOTESTE(iteracoes)

    // Marca o ponto no tempo de final do teste
    fim ← NOW()

    // Imprime resultado obtido e tempo decorrido para validação do teste
    PRINT(resultado)
    PRINT(fim − inicio)
end function
```

### 4.4 Especificações e Resultados

Os testes foram estruturados em 7 partes distintas, cada uma avaliando aspectos específicos do desempenho das linguagens em teste. Para cada categoria, uma função dedicada foi implementada, facilitando o isolamento das tarefas e a análise objetiva dos resultados.

Esta seção tem como objetivo apresentar cada uma dessas funções, explicando o objetivo por trás de cada uma e os resultados obtidos ao executar os *benchmarks* das mesmas. Ainda, a estruturação da apresentação de cada função está dividida em três etapas principais, sendo 1, elaboração da função bem como da ideia por trás do teste; 2, especificação to pseudocódigo definido que foi replicado em cada uma das linguagens analisadas; e 3, apresentação dos resultados observados de forma empírica para diferentes cargas de estresse em cada teste.

Os resultados foram calculados como uma média, derivada de 10 repetições de cada teste. Esse processo tem como objetivo reduzir a variabilidade entre as execuções

individuais. Ao repetir cada teste várias vezes, busca-se minimizar o impacto de variações acidentais ou de atividades não relacionadas (como aplicações sendo executadas em segundo plano no sistema operacional) que poderiam afetar os resultados. Dessa forma, a metodologia busca assegurar que os resultados sejam mais consistentes e representativos do desempenho real. Os códigos utilizados neste estudo estão disponíveis para consulta no seguinte endereço: https://github.com/Xandynhu/benchmark.

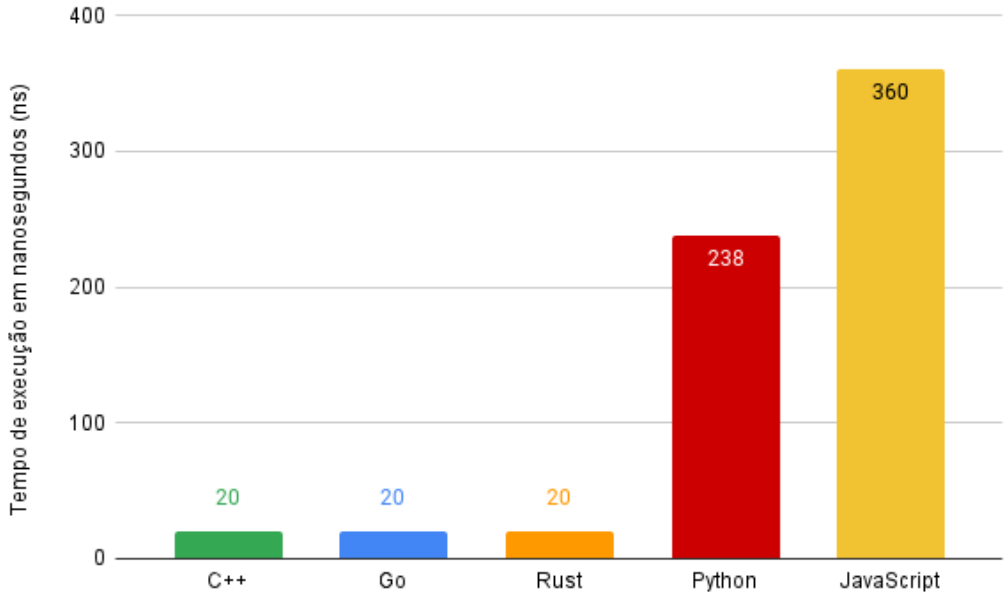
### 4.4.1 Função Vazia

O teste de chamada de função vazia foi intencionalmente criado da maneira simplista para que fosse possível observar o *overhead* adicional implícido das diferentes linguagens ao não executarem absolutamente nada.

A implementação consistiu de uma função que não realiza nenhuma operação e não possui nenhum parâmetro ou valor de retorno. Uma exemplificação mais direta pode ser vista na abstração em pseudocódigo do Algoritmo 2.

A Figura 4.1 exibe a média dos tempos de execução para o teste de chamada de uma função sem conteúdo, onde é observado um desempenho equivalente entre as linguagens compiladas, que demonstraram superioridade em comparação às linguagens interpretadas. É importante mencionar que o número de iterações, conforme descrito na arquitetura de teste padrão no Algoritmo 1, não é relevante para este teste específico.

Figura 4.1: Média de tempo de execução para teste Função Vazia




Fonte: Autor

**Algoritmo 2** Exemplificação do teste Função Vazia

```
function TESTEVAZIO
    // Não realiza operações
end function
```

### 4.4.2 Loop

O teste Loop visa analisar o desempenho de linguagens ao executar um laço de repetição. A função específica para este teste recebe um argumento inteiro, que determina o número de iterações do laço. Em cada iteração, a função realiza uma operação de AND bit a bit entre o valor representante da contagem atual da iteração e 1, acumulando o resultado. Ao final do laço, a função retorna o valor acumulado. O pseudocódigo a seguir ilustra a estrutura básica da função para este teste:

**Algoritmo 3** Exemplificação do teste de Laço

```
function TESTELOOP(n)
    resultado ← 0
    for i ← 0 ; i <n ; i++ do
        resultado ← resultado + i & 1
    end for
    return resultado
end function
```

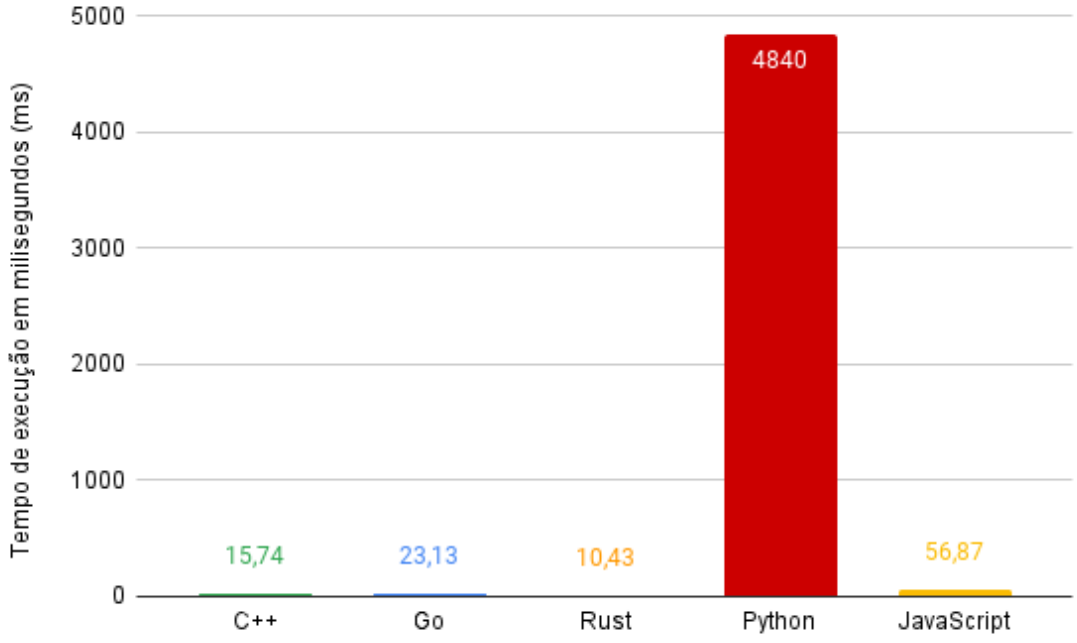
A seguir, a Tabela 4.1 detalha o tempo de execução em nanosegundos para quatro diferentes volumes de iterações ($n$), permitindo uma análise comparativa do desempenho escalável das linguagens. O aumento médio do tempo de execução, em comparação com a base assumida pelo desempenho de C++, varia consideravelmente entre as linguagens, com o Rust mostrando uma melhoria e o Python exibindo aumentos exorbitantes. Já a Figura 4.2 apresenta a média dos tempos de execução para o teste de laço com um volume significativo de iterações ($n = 100.000.000$ na tabela 4.1), evidenciando diferenças substanciais entre linguagens compiladas e interpretadas.

Tabela 4.1: Tempo em nanosegundos para o teste Laço

| | n=100 | n=10.000 | n=1.000.000 | n=100.000.000 | aumento médio |
|---|---|---|---|---|---|
| C++ | 60 | 1.470 | 142.350 | 15.740.000 | base |
| Go | 60 | 2.350 | 231.780 | 23.130.000 | 53,41% |
| **Rust** | **60** | **1.080** | **101.780** | **10.430.000** | **-27,52%** |
| Python | 3.810 | 436.070 | 49.710.000 | 4.840.000.000 | 30.107,15% |
| JavaScript | 2.070 | 81.460 | 634.570 | 56.870.000 | 1.847,89% |

Fonte: Autor

Figura 4.2: Média de tempo de execução para teste Laço com $n = 100.000.000$




Fonte: Autor

### 4.4.3 Cálculo

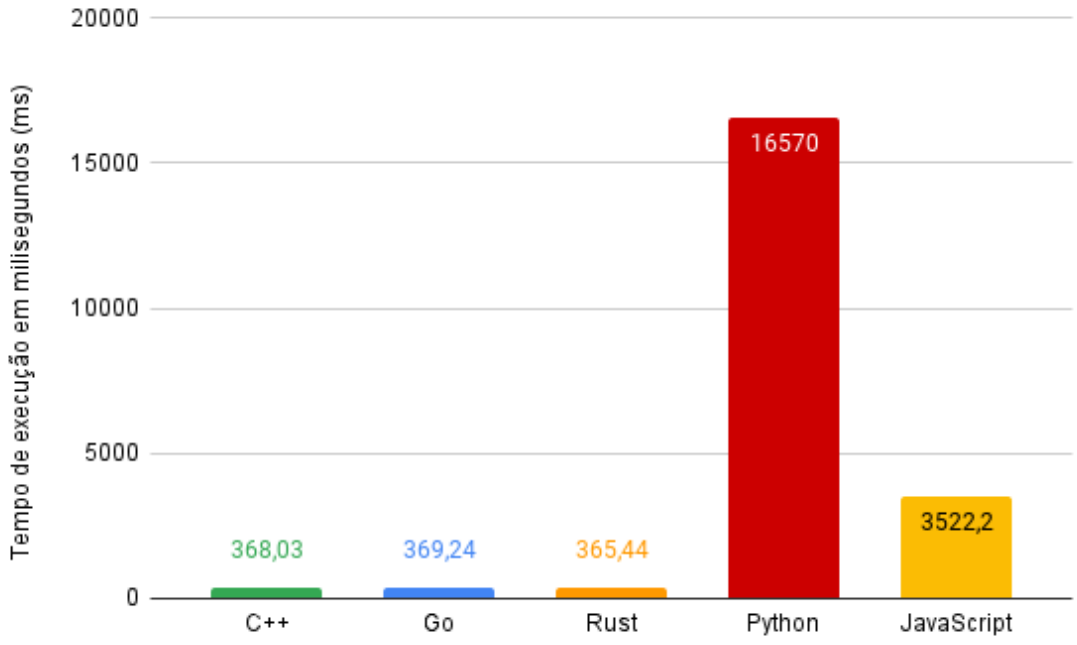
No teste Cálculo, a função envolvida executa um laço com um número predefinido de iterações, onde realiza operações aritméticas básicas (multiplicação e divisão) e operações bit a bit em cada iteração. O número de iterações é determinado por um argumento inteiro recebido pela função. A função retorna o resultado acumulado das operações ao término do laço.

**Algoritmo 4** Exemplificação do teste Cálculo

```
function TesteCalculo(n)
    resultado ← 5
    for i ← 0 ; i < n ; i++ do
        resultado ← resultado × ((i&1) + 1)
        resultado ← resultado ÷ ((i ≫ 1&1) + 1)
    end for
    return resultado
end function
```

A Tabela 4.2 fornece uma visao detalhada dos tempos médios de execução em nanosegundos para o teste de cálculo em diferentes volumes de iterações ($n$). Ainda na mesma tabela é possível observar um empate técnico entre as linguagens compiladas, com diferenças muito pequenas para serem consideradas. Quanto às interpretadas, Python e JavaScript apresentam aumentos significativos. A Figura 4.3, por vez e assim como a figura anterior, ilustra os resultados obtidos pelos testes de meior volume processados ($n = 100.000.000$).

Figura 4.3: Média de tempo de execução para teste de cálculo com $n = 100.000.000$




Fonte: Autor

Tabela 4.2: Tempo em nanosegundos para o teste Cálculo

| | n=100 | n=10.000 | n=1.000.000 | n=100.000.000 | aumento médio |
|---|---|---|---|---|---|
| C++ | 370 | 34.220 | 3.710.000 | 368.030.000 | base |
| Go | 371 | 34.560 | 3.660.000 | 369.240.000 | 0,30% |
| Rust | 360 | 34.550 | 3.520.000 | 365.440.000 | -1,70% |
| Python | 15.020 | 1.540.000 | 157.340.000 | 16.570.000.000 | 4.270,63% |
| JavaScript | 3.480 | 203.590 | 36.200.000 | 3.522.200.000 | 848,79% |

Fonte: Autor

### 4.4.4 Recursão Simples

O teste de Recursão Simples avalia o desempenho de linguagens em lidar com chamadas recursivas de complexidade linear. A função recebe um inteiro que especifica o número de chamadas recursivas a serem feitas e retorna o resultado das operações realizadas durante essas chamadas. Para esse teste foi implementada uma função recursiva simples de complexidade de espaço e tempo igual a $O(n)$, onde o resultado retornado final é igual à metade do número de iterações arredondado para cima.

**Algoritmo 5** Exemplificação do teste Recursão Simples

```
function TESTERECURSAOSIMPLES(iteracoes)
    if iteracoes = 0 then
        return 0
    end if
    return TESTERECURSAOSIMPLES(iteracoes − 1) + (iteracoes&1)
end function
```

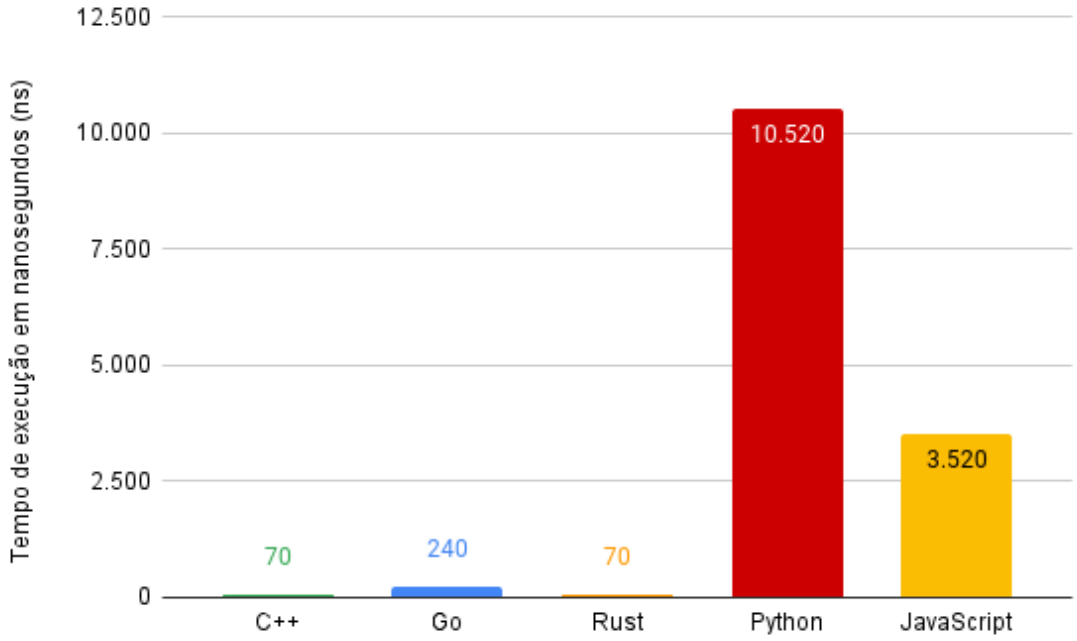
A Tabela 4.3 mostra os tempos de execução para recursão simples, onde Rust tem desempenho excepcional superando C++, enquanto Go, Python e JavaScript, por sua vez, além de menos performáticos, ainda enfrentam limitações com estouro de pilha em iterações mais altas. Já a Figura 4.4 complementa esses dados ilustrando que, para $n = 100$, C++ e Rust são marginalmente mais rápidos que Go, e substancialmente mais rápidos que Python e JavaScript, com este primeiro apresentando o tempo mais alto. Esses resultados enfatizam a eficiência das linguagens compiladas e sem gerenciadores de memória automáticos em tarefas de recursão e as desvantagens potenciais das linguagens interpretadas nesse contexto.

Tabela 4.3: Tempo em nanosegundos para o teste Recursão Simples

| | n=100 | n=10.000 | n=100.000 | n=1.000.000 | aumento médio |
|---|---|---|---|---|---|
| C++ | 70 | 2.310 | 233.030 | 23.950.000 | base |
| Go | 240 | 16.280 | 4.060.000 | $stack\ overflow$ | 604,76% |
| **Rust** | **70** | **1.110** | **105.220** | **11.920.000** | **-51,09%** |
| Python | 10.520 | $stack\ overflow$ | $stack\ overflow$ | $stack\ overflow$ | 14.928,57% |
| JavaScript | 3.520 | 331.660 | $stack\ overflow$ | $stack\ overflow$ | 9.593,07% |

Fonte: Autor

Figura 4.4: Média de tempo de execução para teste Resursão Simples com $n = 100$




Fonte: Autor

### 4.4.5 Recursão Tribonacci

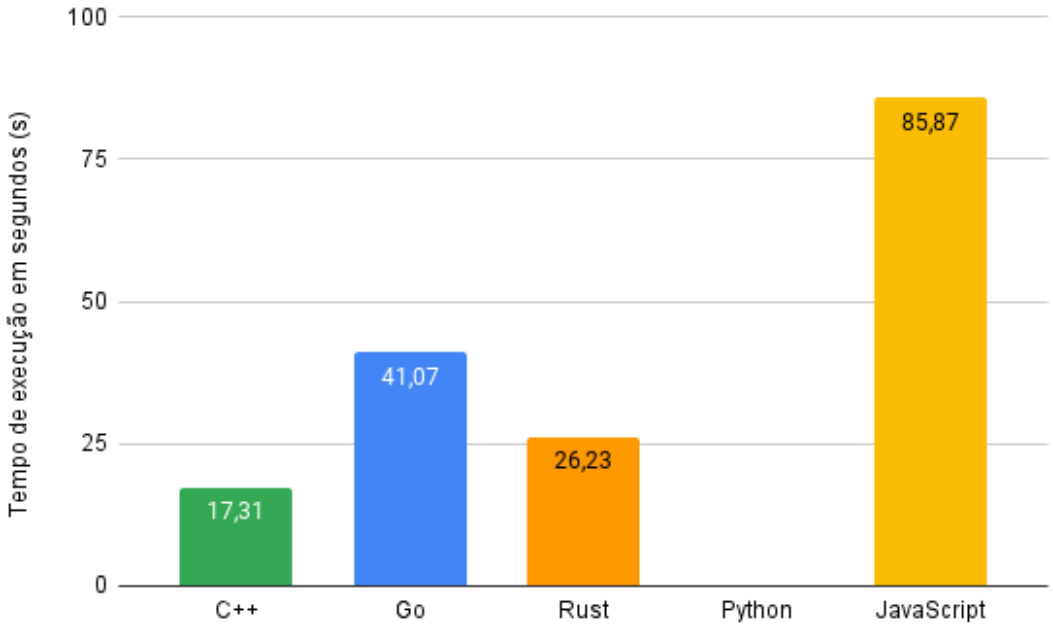
O teste Recursão Tribonacci utiliza uma função que implementa a sequência de Fibonacci de forma alterada recursiva, representando um desafio de complexidade exponencial $O(3^n)$ onde, em vez do número de posição $n$ assumir o valor igual a soma dos números em posições $n-1$ e $n-2$, ele recebe $n-1$, $n-2$ e $n-3$. A função recebe um inteiro que define o número de termos a serem calculados na sequência. Uma exemplificação do código do teste proposto pode ser visto no Algoritmo 6.

A Tabela 4.4 fornece os resultados médios em tempos de execução obtidos para o teste em diferentes volumes de iterações, destacando C++ como o mais performático e Python novamente não conseguindo completar o teste para a maior carga de trabalho. Os resultados para os testes realizados com um maior volume de iterações ($n = 40$) podem ser observados no gráfico da Figura 4.5. Vale ressaltar que, na figura, o resultado para Python não é mostrado pois o sistema não suportou o estresse causado pelo interpretador da linguagem e atingiu um estado de $stack\ overflow$, invalidando a captura de performance em tempo.

**Algoritmo 6** Exemplificação do teste Recursão Tribonacci

```
function TRIBONACCI(n)
    if n < 3 then
        if n = 0 then
            return 0
        end if
        return 1
    end if
    return TRIBONACCI(n − 1) + TRIBONACCI(n − 2) + TRIBONACCI(n − 3)
end function
```

Figura 4.5: Média de tempo de execução para teste Resursão Tribonacci com $n = 40$




Fonte: Autor

Tabela 4.4: Tempo em nanosegundos para o teste Recursão Tribonacci

| | n=20 | n=30 | n=35 | n=40 | aumento médio |
|---|---|---|---|---|---|
| **C++** | **86.080** | **38.050.000** | **857.590.000** | **17.310.000.000** | **base** |
| Go | 187.930 | 84.530.000 | 1.780.000.000 | 41.070.000.000 | 120,24% |
| Rust | 133.920 | 58.580.000 | 1.230.000.000 | 26.230.000.000 | 52,74% |
| Python | 8.700.000 | 4.010.000.000 | 85.690.000.000 | $stack\ overflow$ | 10.006,88% |
| JavaScript | 344.270 | 163.810.000 | 3.440.000.000 | 85.870.000.000 | 315,82% |

Fonte: Autor

### 4.4.6 Leitura de Arquivo

O teste Leitura de Arquivo consiste em avaliar a performance de linguagens ao lerem um arquivo de texto. A função designada para o teste lê um arquivo cujo nome é fornecido como parâmetro. O arquivo contém números inteiros em formato de *string*, que são convertidos em inteiros e armazenados em um *array*. O número de linhas do arquivo é baseado em um parâmetro recebido pela função.

**Algoritmo 7** Exemplificação do teste Ler Arquivo

```
function TESTELERARQUIVO(iteracoes)
    // Concatena o numero de iterações para achar o nome arquivo e abre para leitura
    nomeArquivo ← "../arquivo/" + iteracoes + "_elements.txt"
    arquivo ← ABRIRARQUIVOPARALEITURA(nomeArquivo)

    // Cria vetor para armazenar os inteiros lidos
    resultado ← CRIARVETOR(iteracoes)

    // Lê cada linha, traduz o número de string para inteiro e armazena
    for i ← 0 ; i < iteracoes ; i++ do
        LERLINHA(arquivo, resultado[i])
    end for

    // Retorna o resultado
    return resultado
end function
```

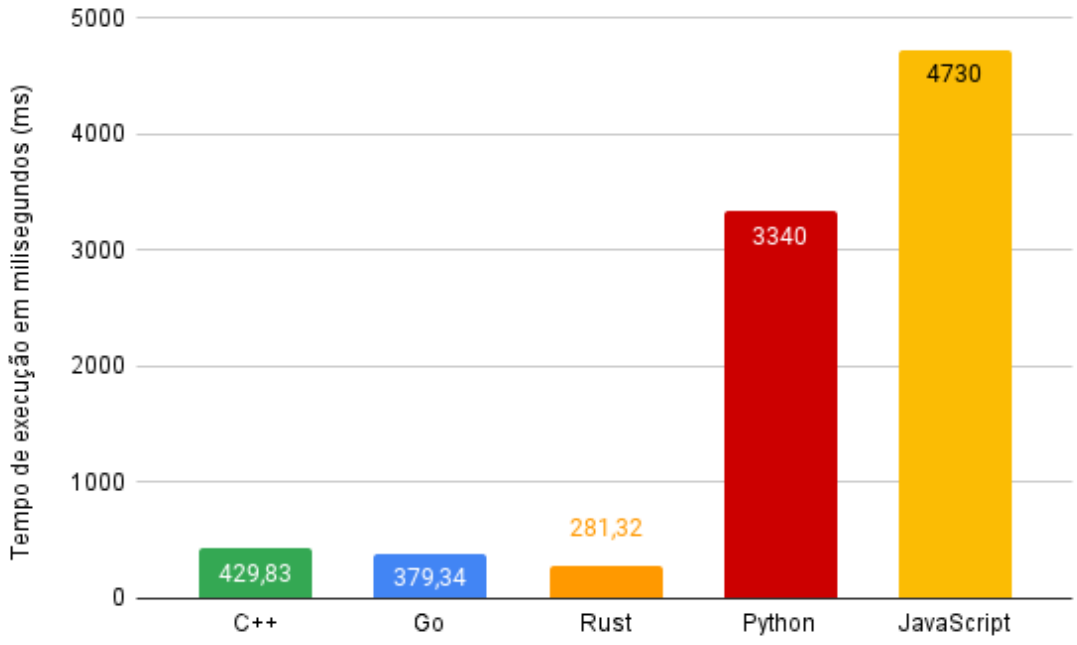
Na Tabela 4.5 e Figura 4.6, Rust e Go demonstram desempenhos superiores em leitura de arquivos, com tempos de execução menores do que C++, que serve como base. Python e JavaScript, por outro lado, apresentam tempos significativamente maiores, com JavaScript tendo o maior aumento percentual. A figura ilustra que, para um arquivo com 10.000.000 de linhas e núemros ($n = 10.000.000$), Rust é o mais rápido, seguido de perto por Go, enquanto Python e JavaScript mostram tempos de execução consideravelmente altos, refletindo a eficácia das linguagens compiladas em operações de I/O em comparação com as interpretadas.

Tabela 4.5: Tempo em nanosegundos para o teste Ler Arquivo

| | n=10.000 | n=100.000 | n=1.000.000 | n=10.000.000 | aumento médio |
|---|---|---|---|---|---|
| C++ | 381.120 | 3.600.000 | 40.200.000 | 429.830.000 | base |
| Go | 276.310 | 2.400.000 | 66.730.000 | 379.340.000 | -19,62% |
| **Rust** | **203.150** | **2.390.000** | **27.320.000** | **281.320.000** | **-34,08%** |
| Python | 1.270.000 | 15.530.000 | 150.000.000 | 3.340.000.000 | 302,26% |
| JavaScript | 1.860.000 | 18.600.000 | 207.980.000 | 4.730.000.000 | 417,01% |

Fonte: Autor

Figura 4.6: Média de tempo de execução para leitura de arquivo com $n = 10.000.000$




Fonte: Autor

### 4.4.7 Escrita em Arquivo

No teste Escrita em Arquivo, o foco é aferir a eficiência de linguagens na escrita de dados em um arquivo de texto. A função recebe a quantidade de números a serem escritos e um *array* de inteiros. Com esses dados, cada número inteiro deve ser escrito no arquivo de texto no formato de *string*. Cada elemento do *array* é escrito em uma linha separada do arquivo.

**Algoritmo 8** Exemplificação do teste Escrever Arquivo

```
function TESTEESCREVERARQUIVO(iteracoes, arr)
    // Concatena o numero de iterações para achar o nome arquivo e abre para escrita
    nomeArquivo ← "../arquivo/" + iteracoes + "_elements.txt"
    arquivo ← ABRIRARQUIVOPARAESCRITA(nomeArquivo)

    // Escreve, linha a linha, os números do array
    for value in arr do
        ESCREVERNOARQUIVO(arquivo, value)
    end for
end function
```

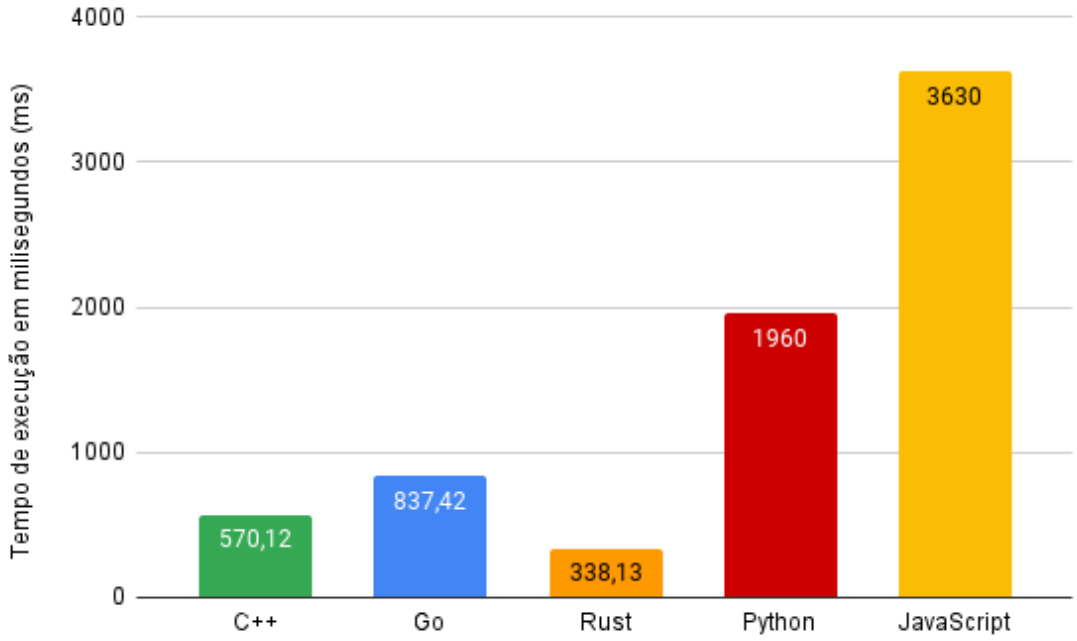
A Tabela 4.6 exibe o tempo necessário para escrever em um arquivo nas diferentes linguagens de programação abordadas e para várias quantidades de linhas e números ($n$). C++ é usada como referência, apresentando um aumento moderado no tempo conforme o número de operações cresce. Go mostra um aumento médio maior em comparação com C++, enquanto Rust destaca-se com uma diminuição de 38,29% no tempo médio de execução, evidenciando sua eficiência em operações de escrita. Por outro lado, Python e JavaScript mostram aumentos substanciais no tempo de execução, com Python atingindo um aumento de mais de 250%. Já Figura 4.7 ilustra a média de tempo de execução para a escrita de arquivos para a maior carga ($n = 10.000.000$), onde Rust se sobressai como a linguagem mais rápida. Python e JavaScript apresentam os tempos mais elevados.

Tabela 4.6: Tempo em nanosegundos para o teste Escrever Arquivo

| | n=10.000 | n=100.000 | n=1.000.000 | n=10.000.000 | aumento médio |
|---|---|---|---|---|---|
| C++ | 618.560 | 5.090.000 | 53.560.000 | 570.120.000 | base |
| Go | 642.240 | 7.030.000 | 80.300.000 | 837.420.000 | 42,50% |
| **Rust** | **384.590** | **3.330.000** | **32.800.000** | **338.130.000** | **-38,29%** |
| Python | 1.860.000 | 22.560.000 | 193.550.000 | 1.960.000.000 | 252,58% |
| JavaScript | 1.360.000 | 6.130.000 | 56.410.000 | 3.630.000.000 | 70,15% |

Fonte: Autor

Figura 4.7: Média de tempo de execução para escrita em arquivo com $n = 10.000.000$




Fonte: Autor

# 5 AVALIAÇÃO DE RESULTADOS

Este trabalho proporcionou uma análise detalhada do desempenho de várias linguagens de programação, cada uma com suas peculiaridades e otimizações. A discussão a seguir se concentra nas características dessas linguagens que podem ter influenciado os resultados obtidos nos testes.

## 5.1 Rust vs C++: Uma Questão de Eficiência e Segurança

Rust superou C++ em termos de eficiência de desempenho, um resultado que surpreendeu o autor deste trabalho e que pode ser atribuído a vários fatores. Primeiramente, Rust foi projetada com a ideia de *Zero-Cost Abstraction*, minimizando adições de latência e estresse em computações ao fazer abstrações de comandos, e ao mesmo tempo com um forte foco na segurança da memória sem sacrificar a performance. Isso é evidente em suas funcionalidades como *ownership* e *borrowing*, que minimizam erros comuns em C++ relacionados ao gerenciamento de memória. Além disso, Rust possui um compilador moderno que realiza otimizações agressivas, contribuindo para a eficiência em tempo de execução.

Por outro lado, C++ oferece um controle mais direto e fino sobre recursos de *hardware* e memória, um legado de sua longa história e evolução contínua. No entanto, abstrações introduzidas nas versões mais recentes de C++, bem como a necessidade de continuar suportando funcionalidades mais antigas, podem ter contribuído para um *overhead* adicional em tempo de execução que o compilador, ao ter que tratar de tantas funcionalidades novas e antigas, pode não conseguir otimizar tão bem quanto o compilador de Rust. Além disso, embora esses recursos aumentem a segurança e a facilidade de uso, eles podem comprometer ligeiramente a performance pura que C++ historicamente proporcionava. É importante notar que, com programadores altamente experientes e um investimento significativo em tempo e testes, além de um problema de escopo suficientemente pequeno, C++ poderia teoricamente ser otimizada a ponto de superar Rust nos testes propostos neste estudo. No entanto, isso exigiria um esforço considerável de tempo e dinheiro, dada a complexidade e a natureza menos restritiva da linguagem.

## 5.2 Go: Simplicidade e Eficiência Moderada

Go demonstrou um desempenho satisfatório nos testes, alinhado com suas propostas de simplicidade e facilidade de uso. A linguagem foi projetada para ser eficiente sem sacrificar simplicidade e teve como motivação principal agilizar e facilitar a programação de aplicações que façam uma melhor utilização de múltiplos núcleos de processamento, funcionalidade essencial para aplicações modernas e que não foi testada no presente trabalho. O coletor de lixo de Go, embora introduza algum *overhead*, oferece um bom equilíbrio entre performance e segurança de memória. Esse compromisso é ideal para muitos cenários de produção, onde a legibilidade do código e a rapidez no desenvolvimento são tão importantes quanto a eficiência.

## 5.3 Python: Entre Demonstração e Produção

Os resultados desfavoráveis à Python nos testes levantam questões sobre seu uso fora do escopo de prototipagem e demonstrações de produtos. Python é conhecida por sua facilidade de uso e ampla biblioteca de suporte, tornando-a ideal para prototipagem rápida e desenvolvimento de *software* em pequena escala. No entanto, suas limitações em termos de desempenho se tornam aparentes em sistemas de produção de larga escala. Para mitigar isso, existem bibliotecas como NumPy e Cython que oferecem melhorias de desempenho ao permitir operações de nível mais baixo e integração com código C/C++. Essas ferramentas podem ajudar a superar algumas das limitações de desempenho, mas em aplicações onde a eficiência é crucial, linguagens mais robustas podem ser necessárias.

# 6 CONCLUSÃO

Nesse capítulo são apresentadas as conclusões e considerações finais sobre o trabalho realizado. Na Seção 6.1 é feita uma recapitulação sobre os assuntos abordados. A Seção 6.2 traz a opinião e conclusão do autor sobre os dados observados. Por fim, a Seção 6.3 aborda algumas ideias sobre trabalhos que podem ser feitos no futuro a fim de expandir os conhecimentos adquiridos neste estudo.

## 6.1 O que Foi Apresentado

A eficiência no uso dos recursos de *hardware* é crucial para o crescimento e a sustentabilidade de um ecossistema, especialmente no contexto da indústria de *software*. Neste estudo, foi introduzida uma análise comparativa para demonstrar como diferentes linguagens de programação podem oferecer resultados variados, dependendo de suas características intrínsecas. Através de testes empíricos, avaliou-se o desempenho de códigos equivalentes implementados em cinco linguagens distintas: C++, Go, Rust, Python e JavaScript.

Os resultados adquiridos neste estudo revelaram surpresas interessantes, que desafiaram as expectativas iniciais do autor. Notavelmente, C++ não se mostrou a linguagem mais eficiente em termos de desempenho, sendo superada por Rust por uma margem significativa. Por outro lado, alguns resultados confirmaram previsões anteriores, como a performance relativamente inferior de Go em comparação com as outras linguagens compiladas. Esse fenômeno pode ser atribuído à presença de um coletor de lixo na linguagem Go, que, apesar de sua eficiência, ainda introduz um *overhead* considerável em relação às linguagens compiladas que não utilizam tecnologias semelhantes.

Além disso, os testes destacaram a baixa performance de JavaScript, bem como resultados particularmente desfavoráveis para Python, que em muitos casos não conseguiu completar os testes devido ao uso excessivo de recursos. Esses achados levantam dúvidas sobre a viabilidade prática dessas tecnologias em sistemas de produção empresariais e outras organizações, sugerindo a necessidade de avaliação cuidadosa ao selecionar linguagens de programação para ambientes de produção, especialmente em contextos onde a eficiência de recursos é uma consideração crítica.

Além disso, o trabalho proporcionou uma visão abrangente, incluindo não apenas aspectos técnicos, mas também uma exploração da história e evolução dessas linguagens

de programação. Foram estabelecidas sete especificações de testes focadas em diferentes funcionalidades que se espera de linguagens tão robustas e reconhecidas. Estas especificações ajudaram a ilustrar os pontos fortes e fracos de cada linguagem em contextos específicos, fornecendo *insights* valiosos para a escolha da linguagem mais adequada para diferentes tipos de projetos de *software*.

## 6.2 Opinião do Autor

Na opinião do autor, cada linguagem de programação - com exceção de Python - apresentou um equilíbrio único entre eficiência, facilidade de uso, segurança e flexibilidade. A escolha da linguagem adequada para um projeto específico deve levar em conta esses fatores, além das competências da equipe de desenvolvimento e das demandas específicas do sistema em questão.

## 6.3 Trabalhos Futuros

A pesquisa realizada oferece um panorama abrangente sobre o desempenho de várias linguagens de programação em termos de eficiência e uso de recursos. Para expandir este trabalho, propõem-se várias direções para futuras investigações:

### 6.3.1 Aprofundamento nas Linguagens Compiladas

Uma exploração mais detalhada das linguagens compiladas (C++, Rust e Go) poderia ser realizada, focando em aspectos específicos e avançados de cada linguagem. Isso inclui o estudo de características como gerenciamento de memória, concorrência, e otimizações específicas do compilador. Além disso, poderia ser interessante avaliar como tanto as atualizações recentes quanto as mais antigas nas linguagens afetam o desempenho e a usabilidade.

### 6.3.2 C++ vs Rust: Estudo Comparativo de Desempenho e Segurança

Esta Seção se concentra na proposta de um estudo comparativo entre C++ e Rust, duas linguagens compiladas de alto desempenho, para entender suas vantagens e desvantagens em termos de desempenho e segurança. Especificamente, podería-se investigar como cada linguagem lida com a gestão de memória e a prevenção de erros comuns, como estouros de buffer, acessos inválidos à memória e vazamentos de memória. Além disso, pode-se realizar testes de *benchmarking* para comparar o tempo de execução, o uso de memória e a eficiência de CPU em várias aplicações, incluindo sistemas embarcados, aplicações Web de alta performance e *softwares* de processamento de dados ao mesmo tempo que buscaría-se manter um código legível e simples em detrimento do máximo de memória possível. Esta análise ajudaria a esclarecer em que cenários uma linguagem pode ser mais adequada do que a outra, oferecendo *insights* valiosos para desenvolvedores e arquitetos de *software* ao escolherem a tecnologia para novos projetos.

### 6.3.3 C++ vs Rust: Estudo Comparativo de Performance em Velocidade e Latência

Esta Seção propõe um estudo focado estritamente nos méritos objetivos de C++ e Rust em relação à performance de velocidade e latência em detrimento de qualquer subjetividade quanto à leitura e manutenção de boas práticas. O objetivo é realizar uma análise quantitativa e detalhada das duas linguagens em diferentes cenários de uso, como aplicações de alto desempenho, sistemas de tempo real, e processamento de grandes volumes de dados - . O estudo incluiria *benchmarks* rigorosos para medir o tempo de execução de tarefas específicas, a latência em operações críticas, e o comportamento sob cargas de trabalho intensas. Este comparativo permitiria identificar as diferenças práticas em termos de eficiência de execução entre C++ e Rust, fornecendo dados concretos sobre qual linguagem oferece a melhor performance em cenários que exigem resposta rápida e processamento eficiente. A análise também pode incluir aspectos como a otimização de compilador e o impacto de diferentes arquiteturas de *hardware* nas performances das linguagens.

### 6.3.4 Testes de Comunicação de Rede e Concorrência

Propõe-se a implementação de testes que avaliem a performance de comunicação de rede entre aplicações escritas nas diferentes linguagens. Isso envolveria medir a eficiência em cenários de I/O de rede, como transferência de dados e requisições HTTP. Além disso, seria valioso avaliar a capacidade das linguagens em lidar com concorrência e comunicações simultâneas, essencial em sistemas distribuídos e aplicações de alta disponibilidade.

### 6.3.5 Análise de Consumo de Recursos

Uma análise mais aprofundada do consumo de recursos, como uso de memória e CPU, seria extremamente relevante. Isso inclui não apenas a quantificação do consumo em diferentes cargas de trabalho, mas também entender como cada linguagem otimiza o uso desses recursos, especialmente em ambientes de produção de larga escala.

# REFERÊNCIAS


BADGETT, T.; MYERS, G. J. et al. **The Art of Software Testing**. [S.l.]: Wiley-Blackwell, 2023.

CPLUSPLUS. **C++ Language Tutorial**. 2024. Acessado em 5 de Janeiro de 2024. Disponível na Internet: <https://cplusplus.com/doc/tutorial/>.

CROCKFORD, D. **JavaScript: The Good Parts**. [S.l.]: O'Reilly Media, Inc., 2008. ISBN 978-0596517748.

GITHUB. **PYPL PopularitY of Programming Language**. 2023. Acesso em: 21 de Dezembro 2023. Disponível na Internet: <https://pypl.github.io/PYPL.html>.

GOOGLE. **Go Language Tutorial**. 2024. Acessado em 6 de Janeiro de 2024. Disponível na Internet: <https://go.dev/doc/>.

HAECKER, R. Sacramental engines: The trinitarian ontology of computers in charles babbage's analytical engine. **Religions**, MDPI, v. 13, n. 8, p. 757, 2022.

HARTIGH, E. D. et al. Platform control during battles for market dominance: The case of apple versus ibm in the early personal computer industry. **Technovation**, Elsevier, v. 48, p. 4–12, 2016.

IVANOV, N. Is rust c++-fast? benchmarking system languages on everyday routines. **arXiv preprint arXiv:2209.09127**, 2022.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. 1st. ed. [S.l.]: Prentice Hall, 1978.

LUTZ, M. **Learning Python**. 5. ed. [S.l.]: O'Reilly Media, Inc., 2013.

MOL, L. D. Turing machines. 2018.

MOZILLA. **The Rust Programming Language**. 2022. Acessado em 6 de Janeiro de 2024. Disponível na Internet: <https://doc.rust-lang.org/book/>.

MüLLER DAVID L. PARKHURST, S. R. C. M. Programming phreeqc calculations with c++ and python: A comparative study. MODFLOW, 2011.

O'REGAN, G.; O'REGAN, G. Edvac and eniac computers. **The Innovation in Computing Companion: A Compendium of Select, Pivotal Inventions**, Springer, p. 113–117, 2018.

PARSONS, R. **How Programming Languages Have Evolved**. 2019. Acesso em: 22 de Dezembro de 2023. Disponível na Internet: <https://www.thoughtworks.com/pt-br/insights/blog/how-programming-languages-have-evolved>.

PFOSI RILEY WOOD, H. Z. J. A comparison of concurrency in rust and c. 2019.

PRECHELT, L. An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program. Fakultat fur Informatik, Universitat Karlsruhe, 2000.

SARTORELLO, L. **O que é compilação?** 2022. Acessado em 23 de Dezembro de 2023. Disponível na Internet: <https://www.alura.com.br/artigos/o-que-e-compilacao>.

SPINELLIS, D.; LOURIDAS, P.; KECHAGIA, M. The evolution of c programming practices: a study of the unix operating system 1973–2015. In: **Proceedings of the 38th International Conference on Software Engineering**. [S.l.: s.n.], 2016. p. 748–759.

STEFANOSKI, K.; KARADIMCHE, A.; DIMITRIEVSKI, I. Performance comparison of c++ and javascript (node. js–v8 engine). **Research Gate**, 2019.

STROUSTRUP, B. **The C++ Programming Language**. 1st. ed. [S.l.]: Addison-Wesley, 1985.

WALDROP, M. M. More than moore. **Nature**, v. 530, n. 144, p. 1–4, 2016.

ZEHRA, F. et al. Comparative analysis of c++ and python in terms of memory and time. Preprints, 2020.