Revista Terra & Cultura: Cadernos de Ensino e Pesquisa
ISSN 2596-2809

ANÁLISE QUANTITATIVA DO DESEMPENHO DE CONCORRÊNCIA EM JAVA, KOTLIN E GO

QUANTITATIVE ANALYSIS OF CONCURRENCY PERFORMANCE IN JAVA, KOTLIN, AND GO

Leonardo Moraes da Silva[1]
Simone Sawasaki Tanaka[2]

RESUMO

Este artigo realiza um estudo comparativo entre duas linguagens de programação orientadas a objetos, Java e Kotlin, e uma linguagem que não se enquadra nesse paradigma, Go. O paradigma de programação orientada a objetos é central no campo do desenvolvimento de *software*, pois facilita a criação de programas estruturados e de fácil manutenção. Java é famosa pela sua portabilidade e é uma das linguagens mais utilizadas, ao passo que Kotlin é uma opção recente e de fácil compreensão, muito usada na criação de aplicativos Android. Go, por outro lado, uma criação do Google, é notável pelo seu apoio ao processamento paralelo em sistemas distribuídos. Este trabalho tem como propósito realizar uma análise considerando o critério de concorrência entre as três linguagens; Java, Kotlin e Go. Através dos dados recolhidos dos sistemas criados, o objetivo é fornecer uma comparação ampla e útil, ajudando os programadores a selecionar a linguagem mais conveniente para os seus projetos.

Palavras-chave: programação orientada a objeto; concorrência; Java; Kotlin; Go.

ABSTRACT

This article conducts a comparative study among two object-oriented programming languages, Java and Kotlin, and a language that does not fall within this paradigm, Go. The object-oriented programming paradigm is central in the field of software development, as it facilitates the creation of structured and easily maintainable programs. Java is renowned for its portability and is one of the most widely used languages, while Kotlin is a recent and easily comprehensible option, commonly employed in the development of Android applications. Go, on the other hand, a creation of Google, is notable for its support for parallel processing in distributed systems. The purpose of this work is to perform an analysis considering the criterion of concurrency among the three languages; Java, Kotlin, and Go. Through the data collected from the created systems, the objective is to provide a comprehensive and useful comparison, aiding programmers in selecting the most suitable language for their projects.

Keywords: object-oriented programming; concurrency; Java; Kotlin; Go.

[1] Graduando do Curso de Ciência da Computação do Centro Universitário Filadélfia de Londrina
[2] Docente do departamento de Computação do Centro Universitário Filadélfia de Londrina - UniFil

1 INTRODUÇÃO

A programação é uma das áreas mais dinâmicas da tecnologia, apresentando um desenvolvimento acentuado e múltiplas possibilidades de linguagens de programação. Entender as diferenças entre as inúmeras linguagens existentes é fundamental para escolher a opção de melhor custo-benefício para cada projeto, considerando as necessidades do projeto em conjunto com as vantagens e facilidades da linguagem. Reconhecendo essa necessidade, este trabalho tem como objetivo realizar uma análise comparativa de três linguagens de programação existentes e muito utilizadas, sendo elas: Kotlin, Java e Go.

No vasto mundo da programação encontramos diversas linguagens, cada uma baseada em um paradigma diferente. Um paradigma é uma abordagem ou estilo de programação que define como os desenvolvedores resolvem problemas e estruturam seu código. Exemplos de paradigmas são o paradigma funcional, orientada a objetos, entre outros (Roy et al., 2003). A programação orientada a objetos se destaca por sua abordagem clara e intuitiva, que representa as entidades do mundo real através de objetos dotados de certas propriedades e métodos (Bezerra, 2007).

Este paradigma permite aos desenvolvedores criar programas organizados, modulares e escaláveis, sendo uma opção frequente para múltiplos projetos (Bezerra, 2007). Entre as linguagens orientadas a objetos, Java é uma das mais populares, com uma ampla comunidade de desenvolvedores (Zeichick, 2022).

A portabilidade é um dos pontos fortes do Java, permitindo que seja executado em diferentes sistemas operacionais (Schildt, 2022). Em contraste, a linguagem Kotlin foi projetada como uma alternativa ao Java, com ferramentas modernas para os programadores. Embora muitos desenvolvedores considerem o Java mais fácil de aprender e usar, o Kotlin, uma linguagem mais recente, vem ganhando popularidade, especialmente no desenvolvimento de aplicativos Android. O Kotlin permite o tratamento de erros com maior segurança, tornando-a uma escolha atraente para desenvolvedores que buscam uma linguagem mais segura e eficiente (Jemerov; Isakova, 2017).

Por outro lado, a linguagem Go, também estudada neste trabalho, foi desenvolvida pelo Google com o objetivo de suportar processamento paralelo e

escalabilidade em sistemas distribuídos. A linguagem Go simplifica o trabalho do desenvolvedor em ambientes que exigem alto desempenho e eficiência no gerenciamento de recursos (Donovan; Kernighan, 2015). Essa linguagem tem sido amplamente utilizada em projetos que envolvem grandes volumes de processamento de dados, como os do próprio Google, devido à sua simplicidade e poder de execução. Um dos principais benefícios do Go é sua capacidade de gerenciar simultaneidade de forma eficiente, permitindo a criação de sistemas altamente escaláveis e robustos. (Donovan; Kernighan, 2015).

Levando em consideração essas diferenças e peculiaridades, este trabalho realizará um levantamento de dados considerando o critério da concorrência, onde serão analisadas duas linguagens de programação orientadas a objetos (Java e Kotlin) e com a linguagem Go, servindo como um contraponto por ser de um paradigma distinto. Esse fator pode ser decisivo para a decisão de qual linguagem de programação utilizar em um projeto. O objetivo dessa pesquisa é fornecer informações para ajudar os desenvolvedores a tomar a melhor decisão ao definir qual linguagem utilizar em seus projetos.

Espera-se oferecer uma base sólida para essa tomada de decisão, fornecendo informações relevantes sobre as três linguagens e permitindo aos desenvolvedores selecionar a opção que melhor atenda às suas necessidades e objetivos.

2 RESULTADOS ESPERADOS

Espera-se através deste artigo obter uma fundamentação sobre quais linguagens foram mais eficazes, considerando o tempo de execução e o uso da memória. A análise será conduzida levando em consideração a aplicação de concorrência em operações I/O. Com base nos dados coletados, espera-se auxiliar os desenvolvedores na otimização do potencial de seus projetos, fornecendo informações para a tomada de decisão sobre qual linguagem se adequa melhor ao projeto quando o fator decisivo for a concorrência.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Paradigmas

Os paradigmas de programação representam abordagens distintas para a criação de *software*, cada uma com seu conjunto de conceitos e técnicas. Entre os principais paradigmas estão o imperativo, lógico, funcional, orientado a eventos e orientado a objetos (Silveira; Cavalcanti, 2021).

A escolha do paradigma adequado depende das características do problema a ser resolvido e do conhecimento do desenvolvedor. Compreender diferentes paradigmas é essencial para os programadores, pois amplia sua capacidade de expressar soluções de maneira mais versátil e genérica. Isso permite escolher a linguagem de programação mais apropriada para abordar problemas específicos e facilita a adaptação a novas linguagens. Os paradigmas de programação desempenham um papel fundamental no desenvolvimento de *software*, influenciando a estrutura e a organização do código. Ao explorar e dominar diferentes paradigmas, os desenvolvedores podem tornar-se mais proficientes e versáteis em suas atividades, contribuindo para a criação de programas mais eficientes e flexíveis (Sebesta, 2018).

O paradigma de orientação a objetos é fundamentado em conceitos como classes, objetos, herança, polimorfismo, encapsulamento e abstração (PERUGINI, 2021). Ao contrário do paradigma procedural, que se concentra em funções e procedimentos, o paradigma orientado a objetos enfatiza a relação entre os objetos e suas interações (Bezerra, 2007). O desenvolvimento de sistemas de *software* baseados neste paradigma foi cunhado por Alan Kay e vem sendo disseminado entre a comunidade de desenvolvedores (Bezerra, 2007). Para que o paradigma orientado a objetos seja considerado, alguns princípios devem ser respeitados. O primeiro deles é que qualquer coisa pode ser um objeto. Além disso, os objetos realizam tarefas solicitando serviços de outros objetos e pertencem a uma determinada classe. Essa classe agrupa objetos semelhantes e serve como um repositório para o comportamento associado ao objeto. Por fim, as classes são organizadas em hierarquias (Perugini, 2021).

Um paradigma concorrente é um modelo de programação que permite a

execução simultânea de múltiplas tarefas dentro de um programa. Este modelo é especialmente relevante em sistemas de hardware paralelo e é comumente implementado usando "*threads*". Em computação científica, a concorrência é vital para o paralelismo de dados e é frequentemente realizada por métodos alternativos como troca de mensagens e operações vetoriais. O objetivo é otimizar o uso dos recursos, melhorar a performance e tornar os sistemas mais responsivos e eficientes (Daleiden, 2016).

3.2 Linguagens de programação

As linguagens de programação são sistemas de comunicação, assim como as linguagens naturais, mas se distinguem pela ausência de ambiguidade. Enquanto as línguas humanas podem ter múltiplas interpretações, as linguagens de programação possuem sintaxe e semântica precisas. Este texto explora a evolução das linguagens de programação, começando com códigos matemáticos e a linguagem Assembly nos anos 1940. À medida que os computadores se desenvolviam, surgiram linguagens de alto nível, como FORTRAN e COBOL na década de 1950. Nos anos 60 e 70, o paradigma estruturado dominou, e a linguagem C emergiu. A década de 80 trouxe a programação orientada a objetos, enquanto os anos 90 viram a expansão da *internet* e o uso de linguagens como Java e JavaScript. A escolha de uma linguagem depende do problema, mas a habilidade de programação transcende qualquer linguagem específica (Bertolini *et al.*, 2019).

Desenvolvida pela JetBrains, o Kotlin é uma linguagem de programação estaticamente tipada. A sua criação emergiu da necessidade de uma linguagem que conseguisse ir além das limitações do Java, mantendo, no entanto, a compatibilidade com a plataforma Java JVM (Jemerov; Isakova, 2017). Um dos aspectos mais conhecidos do Kotlin é a sua sintaxe concisa e clara. Isso permite que se escreva menos código, reduzindo, assim, a quantidade de erros. A linguagem oferece ainda recursos modernos, como inferência de tipo, funções de extensão e programação funcional (Jemerov; Isakova, 2017).

Em relação à sintaxe, o Kotlin se sobressai pela sua simplicidade e concisão. Comparado ao Java, o Kotlin reduz a rigidez formal e prioriza a expressividade e a robustez do código. Isso é evidenciado pelas funções de extensão do Kotlin, que

permitem adicionar novas funcionalidades a uma classe de maneira mais expressiva e menos formal. Além disso, a maneira como o Kotlin lida com exceções, não se apoiando em exceções verificadas, reflete uma abordagem de sintaxe mais segura e simplificada (Bose *et al*., 2018).

Sendo uma linguagem de programação versátil, o Kotlin é utilizado em uma ampla gama de aplicações, desde o desenvolvimento Android até aplicações *server side*. Com o advento do Kotlin/Native, a linguagem expandiu suas possibilidades para o desenvolvimento *cross-platform*, possibilitando o compartilhamento de código entre diferentes plataformas (Petrova, 2022).

A linguagem de programação Go, também conhecida como Golang, é um produto moderno da indústria da computação, desenvolvida para atender às necessidades específicas do cenário atual da computação. Sua concepção se deu no Google, em 2007, pelas mãos de Robert Griesemer, Rob Pike e Ken Thompson, e foi apresentada ao público em 2009 (Donovan; Kernighan, 2015). A motivação para a criação do Go veio da insatisfação com as linguagens de programação existentes para sistemas do Google, notadamente C++ e Java. A visão para o Go é ser uma linguagem simples de compreender e programar, sem sacrificar a eficiência no tempo de compilação e no desempenho de execução (Donovan; Kernighan, 2015).

A simplicidade e eficiência são dois dos principais atrativos do Go. A linguagem possui uma sintaxe clara e consistente, o que a torna fácil de aprender e ler (Balbaert, 2012). Além disso, Go é fortemente tipada e possui um sistema de coleta de lixo automática, proporcionando segurança. É particularmente adequado para programação concorrente e em rede, devido aos seus recursos nativos para *goroutines* e canais (Chisnall, 2012).

Java é uma das linguagens de programação mais populares e influentes do mundo. Criada em 1995 por James Gosling e sua equipe na Sun Microsystems, o Java foi originalmente desenvolvido para dispositivos eletrônicos, como *set-top boxes* e televisores. No entanto, rapidamente se tornou popular para o desenvolvimento de aplicativos para a *web* devido à sua portabilidade, segurança e simplicidade (Gos Ling; Holmes; Arnold, 2005). O funcionamento do Java se baseia no conceito de "*Write Once, Run Anywhere*", que permite que os desenvolvedores escrevam um único código-fonte que pode ser executado em qualquer plataforma que suporte a Máquina Virtual Java (JVM). A JVM é responsável por converter o código-fonte Java

em *bytecode*, que é uma representação intermediária que pode ser executada em diferentes sistemas operacionais e dispositivos (Gosling; Holmes; Arnold, 2005). Várias características-chave do Java, como a herança de classes, a capacidade de criar e gerenciar *threads* e a coleta de lixo automática, facilitam o desenvolvimento de aplicativos complexos e eficientes (Gosling *et al*., 2000).

3.3 Concorrência

A concorrência ocorre quando diversos processos ou *threads* são executados simultaneamente e interagem com recursos compartilhados (Rogliano *et al*., 2022). A palavra ’concorrência’, na sua etimologia, vem do latim *concurrens*, que significa ’que corre junto’ (Cunha, 2019). Este significado etimológico ilustra perfeitamente a natureza da concorrência no domínio dos sistemas de computação. É uma realidade frequente onde diferentes segmentos de um *software* têm a capacidade de funcionar de maneira independente e simultânea, trazendo benefícios como melhoria de desempenho e uso mais eficiente de recursos, embora também apresenta desafios consideráveis (Herlihy; Moss, 1993).

Quando devidamente implementada, a concorrência pode melhorar o desempenho ao permitir a execução simultânea de múltiplos processos ou *threads*, aproveitando assim a capacidade total de processamento do sistema. Além disso, ela otimiza a utilização de recursos, habilitando diferentes partes do programa a operar de forma autônoma e paralela (Herlihy; Moss, 1993).

Concorrência também permite que partes de um programa sejam executadas fora de ordem ou em ordem parcial sem afetar o resultado final. *Threads* são unidades de um programa que executam de forma independente e podem ser agendadas de duas maneiras: cooperativa e preemptiva. Os *threads* cooperativos devolvem voluntariamente o controle para o agendador, enquanto os *threads* preemptivos dependem do agendador para decidir qual deles será executado em seguida. Embora operem de forma independente, os *threads* ainda interagem entre si, o que pode levar a *bugs* específicos que não seriam presentes em programas sequenciais (Rogliano *et al*., 2022). O termo ’user space’ designa um espaço de memória no sistema operacional onde programas de usuário são executados. *Threads* de ’user space’ são coordenadas por um programa ou biblioteca específica, sem intervenção direta do

sistema operacional. Esse conceito é aplicado em linguagens como Go através das *goroutines*. Estas são *threads* de espaço de usuário gerenciadas pelo escalonador de Go, notáveis por sua leveza: iniciam com apenas 2KBs de memória, em comparação com os 2MB das *kernel threads*. Para criar uma *goroutine*, basta preceder uma chamada de função com a diretiva "go". O escalonador de Go adota uma gerência cooperativa de tarefas, permitindo a execução de uma única *goroutine* por *kernel thread* disponível. Assim, Go combina a eficiência de *threads* de 'user space' com um modelo de gerência otimizado (Soares, 2019).

4 METODOLOGIA

A metodologia utilizada para a condução deste estudo será delineada neste capítulo. Abordaremos os métodos e técnicas empregados na coleta de dados, assim como as ferramentas e procedimentos adotados na análise. O entendimento deste segmento é crucial para a compreensão dos resultados e das conclusões que serão apresentadas.

4.1 Configuração do ambiente

Para a coleta de dados proposta neste artigo, foram desenvolvidas três aplicações em Java, Kotlin e Go. Cada uma delas segue a mesma lógica de execução, adaptando-se às particularidades de suas respectivas linguagens. Nestes testes, variamos o número de *threads* ou *goroutines* de 1 a 4, executando cada uma 15 mil vezes. Quando executamos o cenário com apenas uma *thread* ou *goroutine,* coletamos métricas sobre memória utilizada e tempo de execução.

Para cenários com múltiplas *threads* ou *goroutines*, além dessas métricas, capturamos a utilização de memória em momentos específicos e calculamos sua média aritmética. Também avaliamos a memória antes e depois das operações de leitura e escrita para determinar a média de memória usada durante a execução. Além disso, calculamos o *speedup*, *overhead* e eficiência das *threads* ou *goroutines* em questão.

Quanto ao desenvolvimento da aplicação, optou-se por evitar o uso de frameworks, utilizando apenas as dependências estritamente necessárias. Esse

cuidado garantiu que os dados coletados fossem o mais fidedignos possível, mantendo condições igualitárias para todos os cenários testados.

O ambiente de execução das aplicações foi configurado em uma máquina virtual da Oracle, usando a imagem Oracle-Linux-8.8-2023.08.31-0. *Shape* da máquina virtual utilizada foi a "VM.Standard.E2.8" que está equipada com um processador AMD EPYC™ 7551 de 2.0 GHz , 8 OCPU e 64GB de memória.

Para configurar nosso ambiente de desenvolvimento em sistemas Linux, um *script* automatiza diversas etapas cruciais, garantindo inicialmente que todos os pacotes do sistema estão atualizados. Utilizamos o IntelliJ para desenvolvimento em Java e Kotlin e o GoLand para Go, ambas as ferramentas fornecidas pela JetBrains. O Git, na versão 2.39.3, e o GitHub são empregados para o controle de versão, permitindo um gerenciamento eficiente do código-fonte. Para o desenvolvimento em Java, optamos pela versão 17.0.8 do JDK, que é de suporte de longo prazo (LTS). Isso contribui para a estabilidade dos nossos projetos. Complementarmente, o Apache Maven na versão 3.9.4 é utilizado para o gerenciamento de pacotes em projetos Java e Kotlin.

Para projetos em Go, utilizamos a versão 1.21.1 da linguagem, complementada com configurações de ambiente adicionais para otimizar o desenvolvimento. No que tange ao armazenamento de dados, o PostgreSQL é empregado.

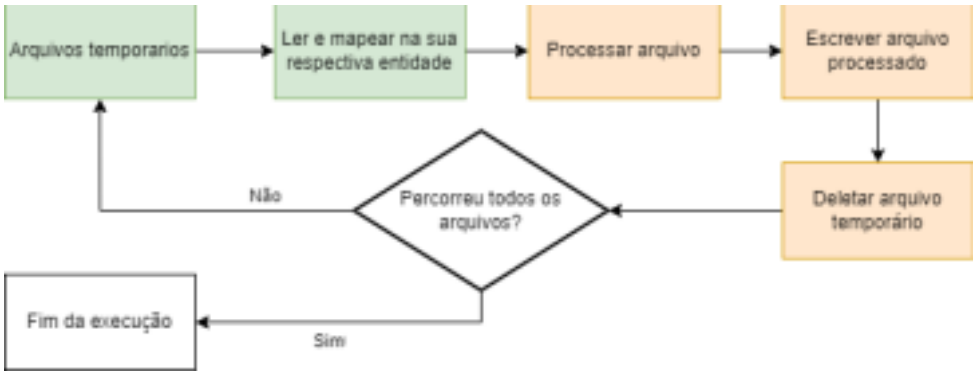
Todas as instalações e configurações ocorreram de forma bem-sucedida e sem dificuldades.

4.2 Desenvolvimento

No processo de construção de uma funcionalidade que requer o processamento paralelo de arquivos, tanto em Java e Kotlin quanto em Go, a lógica fundamental segue uma abordagem semelhante, embora as implementações difiram em seus detalhes.

O ponto de partida em ambas as linguagens envolve percorrer uma lista de arquivos temporários que serão lidos, processados, escritos e deletados de forma concorrente. Como pode ser visto na Figura 1.

Figura 1 – Esquema de execução sem concorrência



Fonte: Autor

Em Java e Kotlin, um *CountDownLatch* é utilizado, inicializado com o número total de arquivos temporários, agindo como um semáforo que faz a *thread* principal esperar pela conclusão das tarefas concorrentes. Já em Go, o pacote *sync* oferece o *sync.WaitGroup*, que serve a um propósito similar. O *WaitGroup* é incrementado com cada nova *goroutine* e decrementado após sua conclusão.

Um ponto de distinção notável ocorre aqui: enquanto em Java e Kotlin já existe uma infraestrutura de gerenciamento de concorrência, como o *ExecutorService*, que pode lidar com a alocação e gestão de *threads*, em Go foi necessário criar um *pool de threads* manual para gerir as *goroutines*. Isso é feito através de uma estrutura chamada *WorkerPool*, que organiza e executa tarefas em *goroutines* de forma controlada.

Após a preparação inicial, ambos os exemplos entram em um *loop* para iterar sobre cada arquivo temporário. Em Java e Kotlin, o *ExecutorService* declarado é responsável por submeter tarefas anônimas para a execução concorrente. Em Go, uma nova *goroutine* é lançada diretamente dentro do *loop*, mas gerenciada pelo *Worker Pool* criado anteriormente.

O trabalho real acontece na tarefa anônima ou *goroutine*, onde o arquivo é lido, os dados são processados e, finalmente, as alterações são escritas de volta ao disco. Em Java e Kotlin, isso é feito pelas operações I/O de *read* e *write*.

Durante essas operações, métricas como tempo de execução e memória usados são capturadas em ambas as linguagens. Em Java e Kotlin, essas métricas são armazenadas em listas sincronizadas, enquanto em Go, os canais são uma opção mais idiomática para armazenar esses dados de forma segura em um ambiente

multi-thread.

Ao final de cada tarefa ou *goroutine*, o contador de controle de concorrência (*CountDownLatch* ou *WaitGroup*) é decrementado. Após todas as tarefas serem concluídas e o contador chegar a zero, a *thread* principal ou o fluxo de execução principal é retomado, e um objeto contendo todas as métricas coletadas é retornado, seja ele um *ExecutionResult* em Java e Kotlin ou uma estrutura de dados correspondente em Go.

Em resumo, para replicar essa funcionalidade em ambas as linguagens, é necessário um mecanismo para gerenciar tarefas em concorrência (como *Executor Service* em Java e Kotlin ou *goroutines* em Go), um mecanismo para sincronizar a conclusão das tarefas (*CountDownLatch* ou *sync.WaitGroup*), e uma estrutura de dados segura a concorrência para armazenar métricas e outros dados relevantes.

4.3 Configuração Experimental

Para avaliar o desempenho de nossos protótipos de concorrência implementados em diversas linguagens de programação, conduzimos um conjunto de experimentos utilizando um intervalo de 1 a 4 *threads*. Cada execução coletou 15.000 vezes os dados, totalizando 60.000 coletas de dados brutos por linguagem analisada. Este processo envolveu a criação de 22 arquivos temporários que foram consistentemente reutilizados em todas as execuções para simular operações de I/O.

Adicionalmente, a utilização da memória foi monitorada registrando o consumo inicial antes do início de cada tarefa de processamento e após a conclusão de cada uma das operações de I/O (*read*, processamento de dados, *write*), como pode ser visto no exemplo do código 2.1. A média aritmética da utilização de memória foi então calculada a partir desses registros. O tempo de execução foi também meticulosamente monitorado: foi calculado desde o início da operação de concorrência até a conclusão total das tarefas. A média aritmética dos tempos de execução para o processamento dos 22 arquivos simulados foi então calculada para fornecer uma visão compreensiva do desempenho.

Listing 2.1 – Exemplo da captura de memória

```
procedimento executeInConcurrency()
    initialMemory := getMemoryNow()

    readFile()
    finalMemory := getMemoryNow()
    saveMemory:= saveUsedMemory(initialMemory, finalMemory)

    processFile()
    finalMemory := getMemoryNow()
    saveMemory:= saveUsedMemory(initialMemory, finalMemory)

    writeFile()
    finalMemory := getMemoryNow()
    saveMemory:= saveUsedMemory(initialMemory, finalMemory)
fimprocedimento

funcao saveUsedMemory(initialMemory, finalMemory)
    return finalMemory - initialMemory
fimfuncao
```


Fonte: Autor

4.4 Tratamento dos Dados e Análise Estatística

Após a fase de coleta, os dados passaram por múltiplos estágios de tratamento e análise. Inicialmente, empregamos o cálculo do Intervalo Interquartil (IQR) para identificar e remover *outliers*.

Seguindo isso, o *speedup* foi calculado utilizando a fórmula $\frac{T_1}{T_n}$ onde T_1 representa o tempo de execução com uma única *thread* e T_n é o tempo com *n threads*. Este indicador é crucial para avaliar melhorias de desempenho atribuíveis à concorrência. Adicionalmente, a eficiência foi avaliada mediante a fórmula da eficiência, que serve como um índice para entender a eficácia na utilização dos recursos computacionais.

$$\text{Eficiência} = \frac{Speedup}{Tn}$$

Em relação ao cálculo de *overhead*, no contexto de concorrência, adotamos a seguinte fórmula: *Overhead* = T_n − Tempo de execução de uma thread. Essa métrica

compara o tempo de execução quando utilizando T_n com o tempo de execução da única *thread*.

Ademais, efetuamos cálculos de métricas estatísticas incluindo o desvio padrão, para entender a dispersão dos dados em torno da média. Também foram calculadas a média aritmética, mediana e moda, fornecendo uma visão abrangente da distribuição dos dados coletados.

5 RESULTADOS OBTIDOS

Neste capítulo, serão descritos os resultados obtidos através da coleta de dados e da análise desses dados coletados. A partir dos resultados, iremos descrever nossa conclusão para avaliar as questões propostas inicialmente no estudo.

5.1 Execução com uma *Thread*

Em nossa análise de execuções em ambiente de *thread* único, os resultados obtidos são apresentados na Tabela 1. A linguagem Go completou a operação em 58.6 milissegundos, Java em 78.79 milissegundos e Kotlin em 86.32 milissegundos. Nessa métrica específica, Go apresentou o melhor desempenho, seguido por Java e, por último, Kotlin.

No entanto, quando observamos o gerenciamento de memória, a narrativa muda. Kotlin, apesar de ter o pior tempo de execução, mostrou um gerenciamento de memória mais eficiente, terminando a operação com menos memória do que tinha no início, com uma diferença de -31.55 MB. Go também demonstrou um bom resultado, usando apenas 6.23 MB de memória. Java, por sua vez, mostrou o maior consumo, com 1,06 GB de memória usada em média.

Tabela 1 – Análise de execução em ambiente de *thread* único.

Linguagem	Tempo de Execução	Memória
Kotlin	86.32	− 31.55 MB
Java	78.79	1,06 GB
Go	58.69	6.23 MB

Fonte: Autor

5.2 Execução com duas *Thread*

Na execução usando duas *threads*, o Go apresentou um desempenho superior, concluindo a tarefa em 35.68 milissegundos. Sua eficiência foi de 84%, apesar de estar abaixo do desempenho de Java e Kotlin. Java executou as mesmas tarefas em 41.27 milissegundos, com uma eficiência de 95%. Kotlin também executou a tarefa em 45.48 milissegundos, com eficiência igualmente alta, também de 95%. Um ponto notável é que Kotlin demonstrou uma gestão mais eficaz de memória, com uma média de 135.00 MB por execução, enquanto Go teve 283.82 MB e Java teve o maior uso com 968.37 MB em média por execução.

O *overhead* de Go, para concorrência, foi de apenas –23.01 milissegundos, significativamente menor em comparação com Java e Kotlin, que tiveram *overheads* de -37.52 e -40.84 milissegundos, respectivamente. Isso sugere uma que para esse tipo de operação, Java e Kotlin tiveram um melhor aproveitamento da adição de uma nova *thread* na execução da tarefa. Os resultados são apresentados na Tabela 2.

Tabela 2 – Análise de execução usando duas *threads*

Linguagem	Tempo de Execução	Overhead	Speedup	Eficiência	Memória
Kotlin	45.48	-40.84	1.9	0.95	135.00 MB
Java	41.27	-37.52	1.91	0.95	968.37 MB
Go	35.68	-23.01	1.68	0.84	283.82 MB

Fonte: Autor

5.3 Execução com três *Threads*

Em um cenário envolvendo três *threads*, a linguagem Go continua liderando em termos de tempo de execução, registrando uma média de 24,14 milissegundos. Ela é seguida por Java, que marcou 28 milissegundos, e Kotlin, que ficou com 30,69 milissegundos. Em termos de eficiência, Java e Kotlin demonstraram um desempenho notável, alcançando 94%, enquanto Go registrou uma eficiência um pouco mais baixa, de 82%. Apesar dessa diferença na métrica de eficiência, Go ainda se destaca por manter um desempenho superior em termos de velocidade.

Ao analisar os resultados associados ao *overhead*, fica evidente que as três linguagens obtiveram avanço significativo no desempenho. Houve uma melhoria de 10 milissegundos na linguagem Go ao comparar execuções com duas *threads*, alcançando um resultado final de -34.54 milissegundos. Isso indica que a adição de mais *goroutines* contribuiu para a otimização do desempenho. Em contraste, Java e Kotlin registraram *overheads* de -50,79 milissegundos e 55,31 milissegundos, respectivamente. No que se refere ao consumo de memória, Java liderou em uso de recursos, com um consumo médio de 4,40 GB. Go seguiu em segundo lugar, utilizando 929,91 MB, enquanto Kotlin ficou em terceiro lugar, com um consumo de 834,39 MB. Esses dados estão resumidos na Tabela 3 e é possível verificar os resultados sintetizados referente a simulação dos 3 *Análise de execução usando três threads*.

Tabela 3 – Análise de execução usando três *threads*

Linguagem	Tempo de Execução	Overhead	Speedup	Eficiência	Memória
Kotlin	30.69	-55.62	2.81	0.94	834.39 MB
Java	28.0	-50.79	2.81	0.94	4.40 GB
Go	24.14	-34.54	2.45	0.82	929.91 MB

Fonte: Autor

Ao avaliar o desempenho das três linguagens de programação em um cenário com quatro *threads*, pode-se observar através dos dados fornecidos na Tabela 4 que

Go mantém sua posição de liderança no quesito do tempo de execução, completando as tarefas em uma média de 19,97 milissegundos. Ele é seguido de perto por Java, que registrou 21,48 milissegundos, e por Kotlin, que anotou 23,5 milissegundos. No quesito eficiência, Java e Kotlin tiveram performances semelhantes, ambos alcançando um coeficiente de 90%, enquanto Go ficou um pouco atrás, com uma eficiência de 74%.

Ao analisar os mais recentes dados de *overhead* de execução, pode-se notar que todas as linguagens mostraram um aumento, porém de forma menos significativa se comparada às melhorias observadas com adições anteriores de novas *threads*. Go registrou um *overhead* de -38.72 milissegundos, enquanto Java e Kotlin apresentaram valores mais elevados, com -57.32 e -62.82 milissegundos, respectivamente.

Quando se trata do consumo de memória, Java continuou a ser o mais exigente, com um uso médio de 3,65 GB. Go e Kotlin foram significativamente mais conservadores neste aspecto. Notavelmente, esta foi a primeira vez em que Go superou Kotlin em gestão de memória, registrando um consumo de 848,48 MB contra 1 GB de Kotlin.

Tabela 4 – Análise de execução usando quatro *threads*

Linguagem	Tempo de Execução	Overhead	Speedup	Eficiência	Memória
Kotlin	23.5	-62.82	3.68	0.92	1 GB
Java	21.48	-57.32	3.67	0.92	3,65 GB
Go	19.97	-38.72	2.95	0.74	848.48 MB

Fonte: Autor

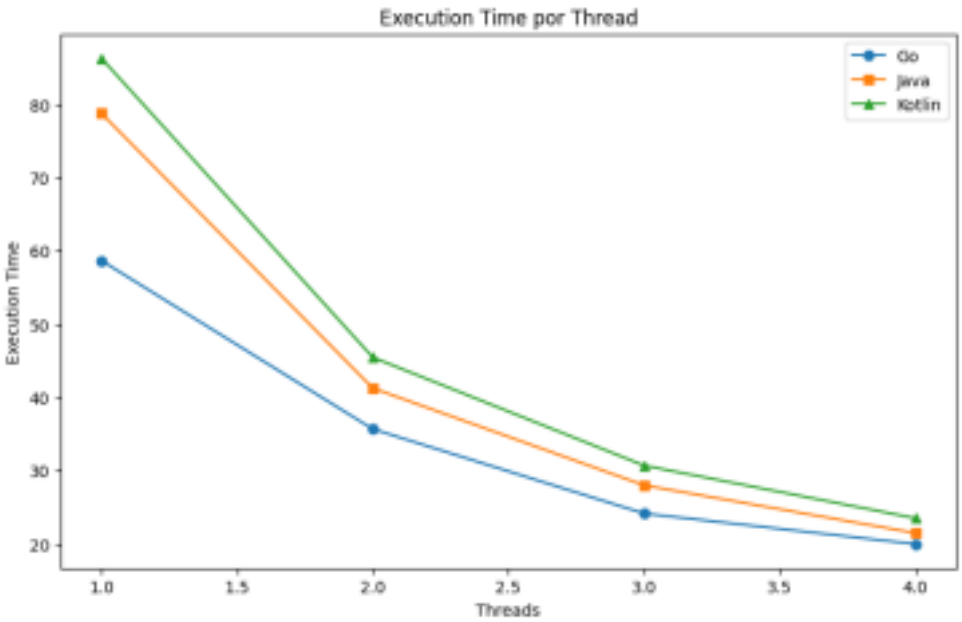
6 CONCLUSÃO

Ao comparar os cenários de execução com uma única *thread* e com múltiplas *threads/goroutines* em concorrência, notamos uma evolução significativa em nosso protótipo de pesquisa. Embora haja uma melhoria notável no desempenho em termos de tempo de execução, é importante salientar que essa eficiência vem com a

consequência de um aumento no consumo de recursos, especialmente no que se refere à memória.

Ao avaliar o desempenho entre as três linguagens, a redução no tempo de execução é notável e pode ser claramente visualizada na Figura 2. Em nosso cenário de testes, Go consistentemente liderou o melhor resultado em termos de tempo de execução mais rápido, seguido por Java e, posteriormente, Kotlin. Isso confirma que no cenário simulado, mesmo com o aumento do número de threads, Go mantém uma vantagem em velocidade.

Figura 2 – Gráfico do tempo de execução x Número de threads

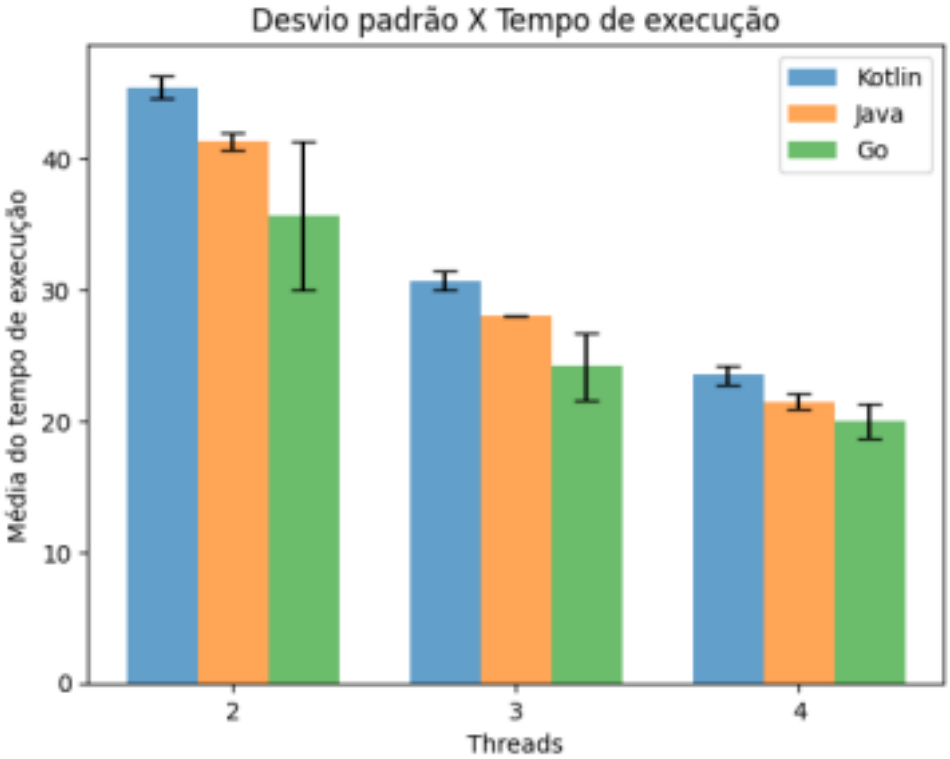


Fonte: Autor

Durante os testes envolvendo até três cenários de execução — de uma a três *threads* —, Kotlin apresentou a gestão de memória mais eficiente dentre as três linguagens avaliadas. Esta eficiência, no entanto, pode estar correlacionada ao fato de que Kotlin também exibiu a pior performance em tempo de execução nessas mesmas circunstâncias. Em contrapartida, ao elevar o cenário para quatro *threads*, a linguagem Go superou as demais tanto em tempo de execução quanto em gestão de memória. Java, por sua vez, demonstrou o uso de memória mais ineficiente em todos os cenários, sugerindo que sua implementação em contextos de concorrência pode acarretar em custos operacionais elevados.

A análise estatística do desvio padrão no tempo de execução trouxe *insights* adicionais. Java e Kotlin mostraram um desvio padrão baixo, indicando uma consistência na performance das tarefas executadas. Isso significa que o tempo de execução entre tarefas individuais nessas linguagens tende a ser muito similar. Go, entretanto, apresentou um desvio padrão significativamente maior, conforme ilustrado na Figura 3. No cenário com quatro *threads*, Go registrou um desvio padrão de 1.34, enquanto Java e Kotlin apresentaram valores de 0.58 e 0.69, respectivamente. Isso sugere que Go tem um intervalo de desempenho mais variável em comparação com as outras linguagens.

Figura 3 – Gráfico do desvio padrão x Tempo de execução



Fonte: Autor

Em resumo, no cenário que envolve quatro *threads*, Go se mostrou o melhor desempenho geral em nosso protótipo, que simulou uma situação de concorrência em pequena escala. Java obteve um resultado semelhante em tempo de execução, mas demonstrou um alto consumo de memória. Kotlin, apesar de eficiente na gestão de memória em cenários com menos *threads*, acabou por ter o pior tempo de execução geral.

6.1 Trabalhos futuros

Para trabalhos futuros, uma recomendação relevante seria a continuação da pesquisa atual, mas com um foco em escalar o número de recursos empregados. Isso pode incluir testes em cenários mais amplos, como o aumento do número de *threads* e *goroutines* utilizados. O objetivo seria entender como a escalabilidade afeta o desempenho e a eficácia dos algoritmos ou sistemas em estudo. A execução desses testes em um ambiente com recursos ampliados fornecerá dados mais robustos e possivelmente *insights* sobre limitações ou melhorias necessárias que só podem ser observadas em uma escala maior.

Uma outra direção futura para pesquisa e desenvolvimento seria investigar a relação entre o tempo de execução e o aumento nos custos operacionais. Isso forneceria *insights* valiosos sobre a escalabilidade inerente a cada linguagem de programação quando aplicada em cenários idênticos.

REFERÊNCIAS

BALBAERT, I. *The way to Go: A thorough introduction to the Go programming language*. [S.l.]: Universe, 2012.

BERTOLINI, C. *et al*. *Linguagem de programação I*. UFSM, NTE, UAB, 2019. Material Didático do NTE - Curso de Licenciatura em Computação. Disponível em: http://repositorio.ufsm.br/handle/1/18352. Acesso em: 2023 jun.2023.

BEZERRA, E. *Princípios de Análise e Projeto de Sistemas com UML*. [S.l.]: Elsevier Rio de Janeiro, 2007. v. 2.

BOSE, S. *et al*. A comparative study: java vs kotlin programming in android application development. *International Journal of Advanced Research in Computer Science*, v. 9, n. 3, p. 41-45, 2018.

CHISNALL, D. *The Go programming language phrasebook*. [S.l.]: Addison-Wesley, 2012.

CUNHA, A. da. *Dicionário etimológico da língua portuguesa*. Lexikon, 2019. Disponível em: https://books.google.com.br/books?id=4yiODwAAQBAJ. Acesso em: 2023 jun.2023.

DALEIDEN, P. M. *Empirical Study of Concurrent Programming Paradigms*. Thesis — UNLV Theses, Dissertations, Professional Papers, and Capstones, 2016. Disponível em: http://dx.doi.org/10.34917/9112055. Acesso em: 2023 jun.2023.

DONOVAN, A. A.; KERNIGHAN, B. W. *The Go programming language*. [S.l.]: Addison-Wesley Professional, 2015.

GOSLING, J.; HOLMES, D. C.; ARNOLD, K. *The Java programming language*. [S.l.]: Addison-Wesley, 2005.

GOSLING, J. et al. *The Java language specification*. [S.l.]: Addison-Wesley Professional, 2000.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. *In*: *Proceedings of the 20th annual international symposium on Computer architecture*. [S.l.: s.n.], 1993. p. 289-300.

JEMEROV, D.; ISAKOVA, S. *Kotlin in action*. [S.l.]: Simon and Schuster, 2017.

PERUGINI, S. *Programming languages: Concepts and implementation*. [S.l.]: Jones & Bartlett Learning, 2021.

PETROVA, E. *Kotlin Multiplatform Mobile Is in Beta – Start Using It Now!* [S.l.]: JetBrains, 2022. https://blog.jetbrains.com/kotlin/2022/10/kmm-beta/. Acesso em: 2023 jun.2023.

ROGLIANO, T. et al. *Technical Report: Unanticipated Object Synchronisation for Dynamically-Typed Languages*. [S.l.], 2022. Disponível em: https://hal.science/hal-03781743. Acesso em: 2023 jun.2023.

ROY, P. V. et al. The role of language paradigms in teaching programming. *In*: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2003. (SIGCSE '03), p. 269-270. Disponível em: https: //doi.org/10.1145/611892.611908. Acesso em: 2023 jun.2023.

SCHILDT, H. *Java: a beginner's guide*. [S.l.]: McGraw-Hill Education, 2022.

SEBESTA, R. *Conceitos de Linguagens de Programação - 11.ed.* Bookman Editora, 2018. Disponível em: https://books.google.com.br/books?id= J3RZDwAAQBAJ. Acesso em: 2023 jun.2023.

SILVEIRA, S.; CAVALCANTI, W. *Paradigmas de Programação: Uma introdução*. [S.l.: s.n.], 2021.

SOARES, J. L. d. S. G. *Um estudo sobre os mecanismos de concorrência da linguagem Go*. 2019. Disponível em: http://hdl.handle.net/11422/11452. Acesso em: 2023 jun.2023.

ZEICHICK, A. *Java é uma das principais linguagens para desenvolvedores de IA e ML, segundo estudo*. [S.l.]: Java Magazine, 2022. Disponível em: https://blogs.oracle.com/ oracle-brasil/post/java-linguagem-desenvolvedores-ia-ml-estudo. Acesso em: 2023 jun.2023.