



**UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE**

**CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS**

*Colegiado de Ciência da Computação*

**Curso de Bacharelado em Ciência da Computação**

## **Um estudo sobre bugs de concorrência em aplicações Go open-source**

Trabalho de Conclusão de Curso

Alonso Lucca Fritz



Cascavel-PR

2024

**UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE**

**CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS**

*Colegiado de Ciência da Computação*

**Curso de Bacharelado em Ciência da Computação**

Alonso Lucca Fritz

## **Um estudo sobre bugs de concorrência em aplicações Go open-source**

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel.

Orientador(a): Marcio Seiji Oyamada

Cascavel-PR

2024

# Lista de figuras

Figura 1 – <b>Bug #43112</b> . Relacionado a Deadlock. Retirado de Docker (2023b) . . . . .	40
Figura 2 – <b>Resolução de <i>Bug</i> #43112</b> . Relacionado a <i>Deadlock</i> . Retirado de Docker (2023b) . . . . .	41

# Lista de tabelas

Tabela 1 – <b>Quantidade de locais que criam <i>goroutines</i></b> . Aplicação, quantidade de locais onde criam <i>goroutines</i> com funções normais, utilizando funções anônimas, total de criações e criações por mil linhas de código. Adaptada de (TU et al., 2019). . . . .	31
Tabela 2 – <b>Uso de primitivas de concorrência</b> . A coluna referente a <i>Mutex</i> contempla <i>Mutex</i> e <i>RWMutex</i> . Adaptada de (TU et al., 2019). . . . .	32
Tabela 3 – <b>Contagem e classificação de Bugs</b> . Adaptado de Lu et al. (2008) . . . . .	35
Tabela 4 – <b><i>Descobertas sobre as características reais dos bugs de concorrência do mundo real e suas implicações</i></b> para a detecção de <i>bugs</i> de concorrência. Adaptada de (LU et al., 2008) . . . . .	36
Tabela 5 – <b>Total de <i>logs de commits</i></b> . Quantidade total de resultados para os critérios de busca definido. Retirado de (DOCKER, 2023b), (TERRAFORM, 2023), (LABS, 2023b) . . . . .	38

# Lista de códigos

Código 1 – Exemplo de Condição de Corrida . . . . .	15
Código 2 – Exemplo de <i>Deadlock</i> . Adaptado de (COX-BUDAY, 2017). . . . .	16
Código 3 – Exemplo de <i>Goroutine</i> . . . . .	21
Código 4 – Exemplo de <i>WaitGroup</i> . . . . .	23
Código 5 – Exemplo de <i>RWMutex</i> . . . . .	24
Código 6 – Exemplo de <i>Cond</i> . . . . .	26
Código 7 – Exemplo de <i>Once</i> . . . . .	27
Código 8 – Exemplo de <i>Channels</i> . . . . .	28
Código 9 – Exemplo de <i>Select</i> . . . . .	30

# Lista de abreviaturas e siglas

IaC	Infrastructure as Code - Infraestrutura como Código
SMTP	<i>Simple Mail Transfer Protocol</i> - Protocolo de Transferência de Correio Simples
POSIX	<i>Portable Operating System Interface</i> - Interface de Sistema Operacional Portátil
CSP	<i>Communicating Sequential Processes</i> - Comunicação de Processos Sequenciais Comunicantes
CPU	Central Processing Unit - Unidade de Processamento Central

# Sumário

<b>1</b>	<b>Introdução</b>	<b>7</b>
1.1	Objetivos Gerais	9
1.2	Objetivos Específicos	10
<b>2</b>	<b>Fundamentação Teórica e Trabalhos relacionados</b>	<b>11</b>
2.1	A linguagem de programação Go	11
2.2	Concorrência	12
2.2.1	Processos	13
2.2.2	<i>Threads</i>	14
2.2.3	Condição de Corrida	14
2.2.4	<i>Deadlocks</i>	16
2.2.5	Sincronização por Memória Compartilhada	18
2.2.6	Sincronização por Passagem de Mensagem	18
2.3	Concorrência em Go	19
2.3.1	<i>Goroutines</i>	20
2.3.2	Pacote <i>Sync</i>	21
2.3.2.1	<i>WaitGroup</i>	22
2.3.2.2	<i>RWMutex</i>	23
2.3.2.3	<i>Cond</i>	24
2.3.2.4	<i>Once</i>	25
2.3.3	<i>Channels</i>	26
2.3.4	<i>Select</i>	28
2.4	Aplicações Go	29
2.5	Trabalhos Relacionados	31
<b>3</b>	<b>Metodologia</b>	<b>37</b>
3.1	Aplicações	37
3.2	<i>bugs</i>	37
3.3	Classificação	38
	<b>Referências</b>	<b>42</b>

# 1

## Introdução

Com a evolução atual dos dispositivos de hardware, o hardware multi-core tornou-se uma realidade não somente para grandes corporações e servidores de ponta, mas também para computadores *desktop*, *notebooks*, *tablets*, *smartphones* e outros dispositivos que possuem mais de um núcleo de processamento. Conforme destacado por [Posovszky \(2020\)](#), as Unidades de Processamento Central (CPUs), popularmente conhecidos como processadores, têm experimentado incrementos significativos em sua capacidade computacional a cada iteração tecnológica nas últimas décadas. Ao longo dos últimos 40 anos, observou-se um aumento no desempenho dos microchips em um fator impressionante de 10.000 desde 1978. Esse avanço tecnológico suscitou a formulação de duas leis fundamentais, sendo a Lei de Escalonamento de Dennard, que postula a possibilidade de manter a queda atual de corrente/tensão e, ao mesmo tempo, preservar a confiabilidade entre circuitos integrados; e a Lei de Moore, que prevê a duplicação do número de transistores por chip a cada ano. Contudo, é importante notar que o aumento de desempenho por geração está diminuindo com cada novo desenvolvimento. A estagnação dessa curva de aumento de desempenho nos últimos 15 anos reflete o declínio das duas leis mencionadas. Em torno de 2004, o Escalonamento de Dennard chegou ao seu fim, e, mais recentemente, em 2011, a Lei de Moore foi demonstrada como não mais válida ([HENNESSY; PATTERSON, 2011](#)). A partir de 2010, fica evidente que o aumento no processamento de um núcleo específico está praticamente estagnado. Nesse contexto, a introdução de processadores com múltiplos núcleos paralelos tem se destacado como uma estratégia para conferir um impulso adicional de desempenho, contudo, isso só se concretiza se a aplicação for desenvolvida para execução paralela.

Consequentemente a quantidade de desenvolvimento de sistemas de software concorrentes aumentou, trazendo consigo a necessidade das empresas por desenvolver sistemas que utilizem ao máximo os recursos disponíveis dos dispositivos, dessa forma a computação concorrente tem se tornado crucial para se aproveitar da evolução do hardware. Porém escrever programas concorrentes não é trivial e essa complexidade tem atingido a comunidade de desenvolvimento de software, desde que escrever programas concorrentes de boa qualidade e corretos tornou-se



importante (STOICA et al., 2014).

Escrever programas concorrentes corretamente não somente é difícil, como a maioria dos programadores pensam de forma sequencial, portanto, cometem erros facilmente ao escrever programas concorrentes, além disso a natureza não determinística torna extremamente difícil para os desenvolvedores detectarem, diagnosticarem e corrigirem *bugs* relacionados. Estes chamados *bugs* de concorrência, são causados por problemas de sincronização em software *multithread* e podem causar bilhões de dólares em perdas econômicas, tornando-se uma ameaça à confiabilidade do uso de softwares *multithread* (LU et al., 2008).

Estudos software de código aberto realizados por Lu et al. (2008) mostraram que geralmente leva vários meses para os desenvolvedores corrigirem corretamente os *bugs* de concorrência. Além disso, os *bugs* de concorrência são os mais difíceis de corrigir corretamente entre os tipos de *bugs* comuns (YIN et al., 2011), com muitos patches incorretos lançados. Partindo deste ponto podemos dizer que desenvolver programas concorrentes de forma correta é um desafio que ainda necessita de conhecimento e avanços em diversos cenários, incluindo entre eles a detecção de *bugs* de concorrência, teste e verificação de modelo de programas concorrentes e o design de uma linguagem de programação concorrente.

Algumas linguagens entretanto passaram por diversas adaptações para então poder disponibilizar o conjunto de ferramentas necessárias para o desenvolvimento concorrente, desde que inicialmente essas linguagens não nasceram com esse tema específico em mente. A linguagem Java criou um modelo de *green threads*, um tipo de mecanismo de concorrência que facilita o uso de fluxos concorrentes, sem utilizar múltiplos processadores, e só depois disponibilizou *threads* nativas do Java (ORACLE, 2000). C++ disponibilizou uma biblioteca própria de *threads* apenas na versão de 2011, antes tendo de utilizar *threads* POSIX, ou a biblioteca *pthreads* do C (ISO, 2011). A linguagem Python, por exemplo, passou a oferecer os recursos de *async/await*, que permite a escrita de códigos assíncronos, com o lançamento da versão 3.5, em setembro de 2015 (PRANSKEVICHUS; SELIVANOV, 2015), estes são alguns exemplos de adaptações realizadas pelas linguagens para atenderem as necessidades atuais de hardware.

A linguagem Go, originalmente desenvolvida pelo Google em 2009 (GO, 2023), tem conquistado rapidamente seu espaço na indústria de software, segundo pesquisa realizada por StackOverflow (2023) 13,24% dos desenvolvedores de todo o mundo escrevem seus códigos em Go. Sendo uma linguagem adotada em uma ampla gama de aplicações, desde bibliotecas como gRPC (GOOGLE, 2023) e *softwares* de alto nível como clientes SMTP (HECTANE, 2023), sistemas de infraestrutura como código, também conhecido como IaC (TERRAFORM, 2023), bancos de dados (LABS, 2023a) até sistemas de infraestrutura em nuvem, como sistemas de contêineres (DOCKER, 2023a). Uma das propostas fundamentais do Go é simplificar e tornar mais segura a programação concorrente, buscando soluções que minimizem os erros nesse contexto, sendo esse o foco de nossa pesquisa.

Um dos principais objetivos do design da linguagem Go é aprimorar a programação

concorrente em comparação com as linguagens tradicionais *multithreaded*. Para alcançar esse objetivo, o Go concentra seu design de concorrência em dois princípios essenciais: primeiramente tornar as *threads*, chamadas *goroutines*, leves e de fácil criação; e em segundo adotar o uso de mensagens explícitas, conhecidas como canais, para facilitar a comunicação entre essas *threads*. Esses princípios fundamentais não apenas introduzem novas primitivas e bibliotecas de concorrência, mas também promovem uma reinterpretação das semânticas já existentes (Tu et al., 2019).

Compreender o impacto dessas novas primitivas e mecanismos de concorrência do Go em relação aos *bugs* de concorrência é crucial. Esses tipos de *bugs* são notoriamente complexos de diagnosticar e têm sido amplamente estudados em linguagens de programação tradicionais *multithread*. Tu et al. (2019), realizou um estudo sobre *bugs* de concorrência na linguagem Go, nas seguintes aplicações *open-source*: Docker, Kubernetes, gRPC, etcd, CockroachDB e BoltDB. Conforme o autor avaliou, existe uma incerteza se os mecanismos de concorrência realmente tornam a programação em Go mais acessível e menos propensa a erros de concorrência quando comparada às abordagens tradicionais.

Os *bugs* foram coletados realizando uma busca nos repositórios *open-source* de projetos populares em Go verificando a recorrência de *bugs* de concorrência. Assim como feito por Tu et al. (2019) e Lu et al. (2008), o objetivo desse trabalho não é corrigir *bugs* de concorrência, mas sim estudar e classificar quais os fatores que ocasionaram o *bugs*, quais as características necessárias para sua manifestação e como ele foi corrigido.

## 1.1 Objetivos Gerais

Este trabalho é uma mesclagem dos estudos realizados por Tu et al. (2019), uma pesquisa sobre *bugs* de concorrência na linguagem Go em aplicações de mundo real, propondo uma nova taxonomia, ou forma de classificar os *bugs* de concorrência considerando o design da linguagem, além de colocar em pauta a discussão entre memória compartilhada e a passagem de mensagem na programação concorrente buscando entender se a linguagem realmente cumpre a promessa de simplificar e tornar mais segura a programação concorrente; e por Lu et al. (2008) que realizou um estudo sobre *bugs* de concorrência em aplicações que utilizavam linguagem C, sua metodologia concentra seu estudo em três principais aspectos: padrão de bug, condições de manifestação e estratégia de correção de *bugs*. Será realizado um estudo sobre os *bugs* de concorrência na linguagem de programação Go, e em como suas primitivas e mecanismos de concorrência afetam de alguma forma os *bugs* de concorrência em aplicações *open-source* como Docker (DOCKER, 2023a), Terraform (TERRAFORM, 2023) e CockroachDB (LABS, 2023a). Go é uma linguagem que possui estruturas tanto para memória compartilhada quanto passagem de mensagens, no entanto ele incentiva o uso de canais em vez de memória compartilhada, considerando assim a passagem explícita de mensagens menos propensa a erros, assim como proposto por Tu et

al. (2019) podemos olhar para uma questão antiga e fundamental da programação concorrente, qual dos mecanismos de comunicação entre *threads* é menos propenso a erros, passagem de mensagens ou memória compartilhada.

O foco principal deste trabalho é investigar como os mecanismos de concorrência do Go impactam a ocorrência e a natureza dos *bugs* de concorrência, bem como identificar e classificar seu comportamento, causa, condições de manifestação e estratégias de correção. A pesquisa visa aprofundar a compreensão *bugs* em aplicações concorrentes de mundo real em Go, em entender, se implementações em Go realmente tornam mais segura e simples a programação concorrente; e qual a relação da utilização de passagem explícita de mensagens e da memória compartilhada com a propensão a *bugs* de concorrência, com o objetivo de contribuir para o desenvolvimento de software mais confiável, e até no desenvolvimento e aprimoramento de ferramentas de diagnósticos eficazes tanto para essa linguagem quanto para *bugs* de concorrência de forma geral.

## 1.2 Objetivos Específicos

- Definir aplicações *open-source* a serem utilizadas como fonte de pesquisa para *bugs*;
- Identificar e Selecionar os *bugs* de concorrência em repositórios de aplicações *open-source*, segundo palavras chave definidas;
- Entender a taxonomia de *bugs* proposta por Tu et al. (2019), e aplica-la aos *bugs* selecionados
- Compreender as causas raiz dos *bugs* de concorrência identificados, incluindo o uso inadequado das primitivas de concorrência do Go;
- Investigar *issues* e *pull requests*, tempos de resposta e solução para problemas de concorrência encontrados;
- Analisar os padrões comuns de erro relacionados à concorrência observados nas aplicações *open-source* estudadas;
- Fornecer orientações e boas práticas para desenvolvedores e pesquisadores que desejam escrever código Go mais confiável e criar ferramentas de depuração e diagnóstico de *bugs* de concorrência mais eficazes para essa linguagem;
- Contribuir para o avanço do conhecimento sobre programação concorrente em Go e aprimorar a compreensão dos desafios e das soluções associados à concorrência nessa linguagem.

# 2

## Fundamentação Teórica e Trabalhos relacionados

Esta seção fornece uma contextualização sobre a linguagem Go, seus mecanismos de concorrência, incluindo seu modelo de *thread*, métodos de comunicação entre *threads*, mecanismos e outras primitivas relacionadas a sincronização de *threads*. Fornece também informações importantes com relação aos paradigmas tradicionais de concorrência, bem como seus *bugs* e principais definições e *bugs* conhecidos. E por fim é apresentado brevemente as três aplicações em Go selecionadas para estudo.

### 2.1 A linguagem de programação Go

*bugs* de concorrência representam desafios significativos para o desenvolvimento e a confiabilidade de sistemas de software. No âmbito de linguagens de programação projetadas para enfrentar tais desafios, o Go, também conhecido como Golang, surgiu como uma ferramenta poderosa e eficiente para programação concorrente. Esta introdução fornece uma breve visão geral da linguagem de programação Go, destacando seus princípios de design, recursos e contexto histórico, estabelecendo as bases para a exploração subsequente de *bugs* de concorrência em aplicativos Go de código aberto.

Go é uma linguagem de propósito geral explicitamente criada com programação de sistemas em mente (PIKE, 2023g). Desenvolvido como um projeto de código aberto, o Go tem como objetivo aprimorar a produtividade do programador, oferecendo uma linguagem expressiva, concisa e eficiente, com forte suporte para programação concorrente (PIKE, 2023a). A linguagem é fortemente tipada, possui coleta de lixo e utiliza uma sintaxe direta que permite análise fácil, tornando-a propícia para análises automáticas por ferramentas como ambientes de desenvolvimento integrado (PIKE, 2023b).

Programas em Go são construídos a partir de pacotes, facilitando a gestão eficiente de dependências. Os princípios de design do Go buscam inspiração na família C em termos de

sintaxe básica, com influências adicionais da família Pascal/Modula/Oberon para declarações e pacotes, e ideias de linguagens inspiradas por Processos Sequenciais Comunicantes (CSP), como Newsqueak e Limbo, para concorrência (PIKE, 2023e).

A linguagem de programação Go, apresentada ao público como um projeto de código aberto em 2009, angariou uma comunidade ampla de desenvolvedores chamados "*gophers*" (PIKE, 2023d). Seu sucesso superou as expectativas iniciais, tornando-se a linguagem de escolha para inúmeros programadores em todo o mundo (PIKE, 2023d).

Notavelmente, o Go possui uma extensa biblioteca de tempo de execução, integrada a cada programa Go. Essa biblioteca implementa recursos críticos, como coleta de lixo, concorrência e gerenciamento de pilhas (PIKE, 2023h). É essencial esclarecer que o tempo de execução do Go difere de ambientes tradicionais de máquinas virtuais, já que os programas Go são compilados antecipadamente para código nativo da máquina (??).

O Go enfrenta os desafios da concorrência construindo sobre as ideias de CSP (PIKE, 2023e). Este modelo enfatiza interfaces de nível mais alto para simplificar o código, desviando da complexidade associada a detalhes de baixo nível, como mutexes e barreiras de memória. As primitivas de concorrência do Go, inspiradas pelo CSP, incluem o conceito inovador de canais como objetos de primeira classe (PIKE, 2023e).

A linguagem de programação Go, com suas raízes na família C e influências do CSP, oferece uma abordagem única para a programação concorrente. Seus princípios de design, sistema de tempo de execução eficiente e ênfase na simplicidade contribuem para sua popularidade entre os desenvolvedores. As seções subsequentes desta irão explorar as características, e primitivas que dão suporte ao diferencial da programação concorrente (PIKE, 2023c).

## 2.2 Concorrência

O conceito de concorrência na computação está associado à ideia de independência da ordem de execução entre programas, onde fluxos distintos não possuem uma dependência causal entre si, permitindo que sejam considerados como fluxos concorrentes (LAMPORT, 2019). A palavra "concorrência" pode ter diferentes interpretações para aqueles na área de computação, sendo algumas vezes utilizada de forma intercambiável com termos como "assíncrono", "paralelo" ou "*threaded*". Alguns consideram essas variações como tendo os mesmos significados, enquanto outros as consideram inteiramente distintas entre si (COX-BUDAY, 2017). A programação concorrente tem tido cada vez de mais importância na ciência da computação, destacando sua complexidade, desafios e a necessidade de um estudo cuidadoso. Segundo Cox-Buday (2017) apesar dos desafios, a linguagem de programação Go oferece primitivas de concorrência que prometem tornar os programas mais claros e eficientes.

Entretanto, o código concorrente é conhecido por ser notoriamente difícil de ser desenvol-

vido corretamente, muitas vezes exigindo iterações para alcançar o funcionamento esperado. *bugs* em códigos concorrentes podem permanecer não detectados por longos períodos, sendo revelados apenas quando ocorrem mudanças nas condições temporais, como aumento na utilização de disco ou maior número de usuários no sistema. A comunidade científica, ciente desses desafios, identificou questões comuns relacionadas ao código concorrente, possibilitando discussões sobre como esses problemas surgem, por que ocorrem e como podem ser solucionados (COX-BUDAY, 2017).

### 2.2.1 Processos

Em um sistema operacional, a representação de um programa em execução é denominada processo. Essa abstração possibilita a execução simultânea de vários processos no mesmo sistema, aparentando que cada um tem uso exclusivo do *hardware*. O conceito de simultaneidade refere-se à intercalação de instruções entre processos, sendo que, na maioria dos sistemas, há mais processos para execução do que núcleos disponíveis. Nos sistemas tradicionais, apenas um programa era executado de cada vez, ao passo que processadores *multicore* mais modernos têm a capacidade de executar diversos processos ou *threads* simultaneamente. Mesmo em um único núcleo, a ilusão de execução simultânea entre processos é obtida por meio da troca de contexto, um mecanismo operado pelo sistema operacional (BRYANT; O'HALLARON, 2011). Essa flexibilidade no uso de *threads* dentro de um processo destaca a adaptação dos sistemas operacionais modernos para atender às demandas de eficiência e concorrência na execução de programas.

Em sistemas operacionais contemporâneos, cada processo é alocado em um espaço de memória exclusivo, prevenindo assim problemas de integridade dos dados do programa, uma vez que cada processo possui seu próprio espaço de endereçamento. O sistema operacional desempenha o papel de fornecer e proteger esse espaço de memória contra acessos indevidos por outros processos. A comunicação com dispositivos ocorre frequentemente por meio de chamadas de sistema, operações realizadas por um processo para requisitar ao sistema operacional a execução de uma tarefa específica, como desenhar um ponto na tela (TANENBAUM; FILHO, 2016).

Além disso, os processos em sistemas operacionais modernos não são a menor abstração de um fluxo de execução, uma vez que podem conter uma ou mais *threads*, fluxos de execução independentes. *Threads* compartilham o espaço de memória e código de um processo, mas cada uma possui seus próprios registradores e ponteiros de instrução. Embora compartilhem dados, as *threads* não interferem umas nas outras, garantindo a independência de execução. A criação e o gerenciamento de *threads* são responsabilidades do sistema operacional, que provê recursos como espaço de memória e registradores, além de controlar a execução das *threads* (TANENBAUM; FILHO, 2016).

### 2.2.2 Threads

Em sistemas operacionais modernos, o conceito de processo evoluiu para além de um único fluxo de controle, permitindo que um processo consista em múltiplas unidades de execução conhecidas como *threads*. As *threads* operam no contexto do processo, compartilhando o mesmo código e dados globais. Essa abordagem destaca a importância crescente das *threads* como um modelo de programação vital, especialmente diante da necessidade de concorrência em servidores de rede, por exemplo (BRYANT; O'HALLARON, 2011).

A relevância das *threads* é evidenciada pela facilidade de compartilhamento de dados entre elas, comparado com a complexidade de compartilhar dados entre processos distintos. Além disso, as *threads*, em geral, são mais eficientes que processos isolados. O uso de *multithread* não apenas atende à demanda por concorrência, mas também representa uma estratégia para acelerar a execução de programas em sistemas com múltiplos processadores disponíveis (TANENBAUM; FILHO, 2016).

Atualmente podemos destacar dois tipos principais de *threads*, sendo elas as *kernel threads* e as chamadas *user-space threads*, ou *threads* de espaço de usuário. A diferença essencial entre elas está no detentor da responsabilidade pela coordenação de execução de fluxos. Sendo para as *kernel threads*, o sistema operacional é o responsável por essa gerência, é comum em sistemas UNIX e Windows essas *threads* serem do padrão POSIX. Enquanto para as de espaço de usuário, um programa criado por um usuário que é o responsável por essa gerência, não tendo uma visão global que o sistema operacional possui, como exemplo conhecido podemos citar a máquina virtual do Java com o recurso de *green threads*, outro exemplo, importante para esse trabalho é o escalonador de Go, que gerencia as *goroutines* (TANENBAUM; FILHO, 2016), mostradas em subseção 2.3.1.

### 2.2.3 Condição de Corrida

Em um contexto de abstração para *threads*, uma condição de ocorrência ocorre sempre que um programa depende de uma *thread* atingir um objetivo x em seu fluxo de controle antes que outra *thread* atinja o objetivo y (BRYANT; O'HALLARON, 2011). E como iremos trabalhar com Go, dentro desse contexto pode-se considerar que uma condição de corrida ocorre quando o programa não fornece o resultado correto para algumas intercalações das operações de múltiplas *goroutines* (DONOVAN; KERNIGHAN, 2015).

No contexto geral então uma condição de corrida ocorre quando duas ou mais operações devem ser executadas em uma ordem correta, mas o programa não foi escrito de forma a garantir que essa ordem seja mantida, se manifestando no que é chamado de corrida de dados, onde uma operação concorrente tenta ler uma variável enquanto, em um determinado momento, outra operação concorrente tenta escrever na mesma variável (COX-BUDAY, 2017).

Um exemplo é a maneira mais clara de entender a natureza desse problema, considerando



o [Código 1](#). O termo *go* é utilizado para executar uma função concorrentemente, dessa maneira ao fazer isso é criado o que é denominado como *goroutines*, discutido em maior detalhes em [subseção 2.3.1](#). Dessa maneira nas linhas 8 e 9 são criadas duas *goroutines* que incrementam uma variável compartilhada chamada *counter* em um *loop*. A função *main* utiliza *sync.WaitGroup* para garantir que as duas *goroutines* concluam antes de imprimir o valor final de *counter*, isso é discutido em detalhes na [subseção 2.3.2.1](#).

Código 1 – Exemplo de Condição de Corrida

```
1 var counter int
2 var wg sync.WaitGroup
3
4 func main() {
5     runtime.GOMAXPROCS(2)
6     wg.Add(2)
7
8     go incrementCounter()
9     go incrementCounter()
10
11     wg.Wait()
12     fmt.Println("Final Counter:", counter)
13 }
14
15 func incrementCounter() {
16     defer wg.Done()
17     for i := 0; i < 10000000; i++ {
18         // Critic! Race condition here!
19         counter++
20     }
21 }
```

A condição de corrida ocorre porque duas *goroutines* estão concorrendo para acessar e modificar a variável *counter* sem uma sincronização adequada. Sendo esta uma situação em que o comportamento do programa depende da ordem de execução das *threads* ou *goroutines*.

Ambas as *goroutines* estão incrementando *counter* simultaneamente sem qualquer mecanismo de sincronização, o que pode resultar em leituras e gravações concorrentes na variável compartilhada.

Na maioria das vezes, as corridas de dados são introduzidas porque os desenvolvedores estão pensando no problema de forma sequencial assumindo que, porque uma linha de código está antes de outra, ela será executada primeiro (COX-BUDAY, 2017). De forma mais específica ocorre porque os desenvolvedores assumem que as *threads* seguirão alguma trajetória específica pelo espaço de estados de execução, esquecendo a regra fundamental de que programas com *threads* devem funcionar corretamente para qualquer trajetória viável (BRYANT; O'HALLARON, 2011).

Condições de corrida são perigosas porque podem permanecer latentes em um programa e aparecer raramente, talvez apenas sob carga intensa ou ao usar determinados compiladores,



plataformas ou arquiteturas. Isso torna difícil reproduzi-las e diagnosticá-las (DONOVAN; KERNIGHAN, 2015).

### 2.2.4 Deadlocks

Um cenário de deadlock ocorre quando todos os processos concorrentes aguardam uns aos outros. Nesse estado, o programa torna-se irrecuperável sem intervenção externa (COX-BUDAY, 2017). O tempo de execução (*runtime*) do Go faz tentativas de detectar alguns *deadlocks*, nos quais todas as *goroutines* devem estar bloqueadas ou "adormecidas" (Golang Team, 2023). No entanto, essa abordagem não oferece uma solução abrangente para prevenir *deadlocks*.

Para uma compreensão mais clara do que constitui um *deadlock*, o Código 2 apresenta um exemplo de *deadlock* devido à possibilidade de duas *goroutines* ficarem bloqueadas uma esperando pela outra.

Código 2 – Exemplo de *Deadlock*. Adaptado de (COX-BUDAY, 2017).

```
1 type value struct {
2     mu    sync.Mutex
3     value int
4 }
5 var wg sync.WaitGroup
6 printSum := func(v1, v2 *value) {
7     defer wg.Done()
8     v1.mu.Lock()
9     defer v1.mu.Unlock()
10
11     time.Sleep(2 * time.Second)
12     v2.mu.Lock()
13     defer v2.mu.Unlock()
14     fmt.Printf("sum=%v\n", v1.value+v2.value)
15 }
16 var a, b value
17 wg.Add(2)
18 go printSum(&a, &b)
19 go printSum(&b, &a)
20 wg.Wait()
```

Analisando as sessões críticas segundo Cox-Buday (2017):

- **Linha 8:** Aqui tentamos entrar na seção crítica, a *goroutine* adquire a trava (*lock*) do *mutex* associado à estrutura *v1*. Isso significa que ela está bloqueando o acesso a qualquer outra *goroutine* que tente adquirir o mesmo *mutex*. No entanto, observe que a *goroutine* também chama *v2.mu.Lock()* (linha 11) mais tarde no código.
- **Linha 9:** O uso de *defer* garante que o *mutex* seja desbloqueado mesmo se ocorrer uma exceção ou se a função retornar. Neste caso, a *goroutine* desbloqueia o *mutex* associado a *v1* quando a função *printSum* é concluída, portanto usamos a instrução *defer* para sair da seção crítica antes que *printSum* retorne.

- **Linha 11:** Este é um atraso de 2 segundos adicionado intencionalmente para simular trabalho, aumentando as chances de ocorrer um *deadlock*. Durante esse tempo, a primeira *goroutine* ainda mantém o *lock* de v1, mas está prestes a solicitar o *lock* para v2.
- **Linha 12:** A segunda *goroutine* tenta adquirir o *lock* para o *mutex* associado a v2. No entanto, este *mutex* está atualmente bloqueado pela primeira *goroutine*.

O que ocorre em suma é que a primeira chamada para *printSum* trava v1 e, em seguida, tenta travar v2, mas enquanto isso, a segunda chamada para *printSum* travou v2 e tentou travar v1, dessa forma ambas as *goroutines* esperam uma pela outra infinitamente. Dessa forma fica muito claro que estamos lidando com um *deadlock*, mas para validarmos um *deadlock* devemos cumprir quatro condições, conhecidas como Condições Coffman, sendo elas a base para técnicas de detecção, prevenção e correção de *deadlocks*. Sobre as Condições de Coffman segundo Cox-Buday (2017).

1. **Exclusão Mútua:** Um processo concorrente detém direitos exclusivos sobre um recurso em qualquer momento.
2. **Condição de Espera:** Um processo concorrente deve simultaneamente deter um recurso e estar aguardando por um recurso adicional.
3. **Não Preempção:** Um recurso detido por um processo concorrente só pode ser liberado por esse processo.
4. **Espera Circular:** Um processo concorrente (P1) deve estar aguardando em uma cadeia de outros processos concorrentes (P2), que, por sua vez, estão aguardando por ele (P1).

Ao analisar o [Código 2](#) por essa perspectiva:

1. A função *printSum* requer direitos exclusivos tanto para "v1" quanto para "v2", portanto, ela atende a condição de exclusão mútua.
2. Como *printSum* mantém a posse de "v1" ou "v2" e está aguardando pelo outro, temos a condição de espera.
3. Não existe nenhuma maneira para que as *goroutines* sejam antecipadas
4. A primeira chamada de *printSum* está aguardando a segunda chamada, a recíproca também é verdadeira, portanto temos a espera circular.

Essas condições permite não apenas identificar mas também evitar *deadlocks* ao garantir que ao menos uma das Condições de Coffman não seja verdadeira, supostamente é evitado o *bug* relacionado a *deadlocks*, porém na prática essas condições se mostram difíceis de serem analisadas e prevenidas (COX-BUDAY, 2017).

### 2.2.5 Sincronização por Memória Compartilhada

A memória compartilhada, uma técnica de sincronização concorrente em Go, permite que múltiplas *goroutines* acessem simultaneamente a mesma variável ou recurso, evitando a sobrecarga de comunicação entre elas. Para implementar a memória compartilhada em Go, utilizam-se estruturas como *channels* e *mutexes*. *Channels* possibilitam comunicação assíncrona entre *goroutines*, enquanto *mutexes* protegem o acesso exclusivo a recursos compartilhados. Contudo, essa eficiente técnica é mais propensa a condições de corrida. Por outro lado, a passagem de mensagens, uma alternativa que envolve o envio de mensagens entre *goroutines*, é menos propensa a condições de corrida, mas pode apresentar menor eficiência em termos de desempenho. A escolha entre essas técnicas é crucial, e a análise de [Tu et al. \(2019\)](#) sobre 171 *bugs* de concorrência em Golang destaca a importância de usar a memória compartilhada com cautela, visto que a maioria dos *bugs* identificados estava relacionada a condições de corrida ([TU et al., 2019](#)).

O Go oferece suporte a acessos tradicionais à memória compartilhada, proporcionando primitivas de sincronização como *Mutex*, *RWMutex*, *Cond* e *atomic*. O *RWMutex* no Go difere da implementação *pthread\_rwlock\_t* em C, conferindo maior privilégio a solicitações de trava de escrita. Introduzindo a primitiva "Once", o Go assegura que uma função seja executada apenas uma vez, sendo valioso para a inicialização única de variáveis compartilhadas por várias *goroutines*. Além disso, o "WaitGroup" é empregado para permitir que múltiplas *goroutines* concluam seus acessos a variáveis compartilhadas antes que uma *goroutine* de espera prossiga. Contudo, o uso inadequado do *WaitGroup* pode resultar em *bugs* bloqueantes e não-bloqueantes ([TU et al., 2019](#)). Essas ferramentas oferecidas pelo Go refletem a preocupação em equilibrar eficiência e segurança em ambientes concorrentes.

### 2.2.6 Sincronização por Passagem de Mensagem

A sincronização por passagem de mensagem é uma técnica de sincronização de *threads* que utiliza a troca de mensagens entre *threads* para garantir a consistência de dados ([TU et al., 2019](#)). Essa técnica é baseada no princípio de que *threads* só podem acessar dados compartilhados se receberem uma mensagem autorizando-os a fazê-lo.

Uma das vantagens da sincronização por passagem de mensagem é que ela é mais segura do que a sincronização por memória compartilhada, pois reduz a possibilidade de condições de corrida ([CORMEN et al., 2009](#)). Isso ocorre porque, nesta sincronização, as *threads* só podem acessar dados compartilhados se receberem uma mensagem autorizando-os a fazê-lo. Isso elimina a possibilidade de um *thread* acessar dados compartilhados que estão sendo modificados por outro *thread*.

Outra vantagem da sincronização por passagem de mensagem é que ela é mais flexível do que a sincronização por memória compartilhada ([CORMEN et al., 2009](#)). Isso ocorre porque,

na sincronização por passagem de mensagem, os *threads* podem trocar mensagens de qualquer tipo, incluindo dados estruturados complexos. Isso permite que os *threads* compartilhem dados de forma mais eficiente e flexível.

No entanto, a sincronização por passagem de mensagem também apresenta algumas desvantagens. Uma das desvantagens é que ela pode ser menos eficiente do que a sincronização por memória compartilhada. Isso ocorre porque, na sincronização por passagem de mensagem, os *threads* precisam trocar mensagens para acessar dados compartilhados. Essa troca de mensagens pode adicionar *overhead* ao sistema (TANENBAUM; FILHO, 2016).

Outra desvantagem da sincronização por passagem de mensagem é que ela pode ser mais complexa do que a sincronização por memória compartilhada. Isso ocorre porque, na sincronização por passagem de mensagem, os desenvolvedores precisam implementar mecanismos para gerenciar a troca de mensagens entre *threads* (TANENBAUM; FILHO, 2016).

## 2.3 Concorrência em Go

A importância da concorrência no cenário computacional contemporâneo, especialmente no contexto de máquinas *multicore* e servidores web lidando com múltiplos clientes. Go, como linguagem, merece atenção ao incorporar uma variante da comunicação de processos sequenciais (CSP) com canais de primeira classe. A adoção do CSP é motivada não apenas pela familiaridade de um dos projetistas da linguagem, mas também pela sua integração fácil a um modelo de programação procedural sem exigir mudanças profundas na estrutura da linguagem. Essa composição de funções executando de forma independente dentro de código procedural regular permite que o Go una a concorrência à computação. (PIKE, 2012).

Embora o modelo de concorrência do Go prove ser vantajoso para cenários típicos, ele vem com uma ressalva significativa: o Go não é puramente seguro em termos de memória na presença de concorrência. A linguagem permite compartilhamento, incluindo a passagem de ponteiros por canais, o que é tanto idiomático quanto eficiente. Apesar de alguma decepção por parte de especialistas em concorrência e programação funcional que esperavam uma abordagem de escrita única para semântica de valor, semelhante a linguagens como Erlang, as escolhas de design do Go priorizam a familiaridade e adequação ao domínio do problema. A linguagem promove programação concorrente simples e segura, confiando em convenções e treinamento para incentivar programadores a adotar um paradigma de passagem de mensagens como uma forma de controle de propriedade. O lema "Não se comunique compartilhando memória, compartilhe memória se comunicando" encapsula a abordagem do Go, promovendo simplicidade e robustez em software de rede concorrente (PIKE, 2012).

### 2.3.1 Goroutines

*Goroutines*, fundamentais no Go, simplificam o uso de concorrência ao serem funções que operam de maneira independente ou *coroutines*, multiplexadas em um conjunto de *threads*. O sistema de tempo de execução, conforme destacado por Pike (2023f), gerencia automaticamente situações bloqueantes, movendo *coroutines* entre *threads*, garantindo eficiência na utilização dos recursos do sistema.

No sentido de atingir eficiência, o tempo de execução do Go implementa pilhas redimensionáveis e limitadas para *goroutines*. A dinâmica de ajuste do tamanho da pilha possibilita a criação de inúmeras *goroutines* com uma quantidade moderada de memória. Essa escolha de design diferencia as *goroutines* das *threads* tradicionais, contribuindo para a reputação do Go como uma linguagem que facilita a programação escalável e concorrente, conforme destacado por Pike (2023f).

*Goroutines*, sendo unidades fundamentais de organização em programas Go, merecem compreensão aprofundada. Todo programa Go, ao iniciar, possui pelo menos uma *goroutine*, denominada *goroutine* principal, automaticamente criada e iniciada durante o processo (COX-BUDAY, 2017). A criação de novas *goroutines* é realizada pela instrução `go`, que, sintaticamente, precede uma chamada de função ou método comum, levando à execução da função em uma *goroutine* recém-criada (DONOVAN; KERNIGHAN, 2015).

A definição de *goroutines* em Go, como atividades executadas concorrentemente, difere de um programa sequencial, onde chamadas para diferentes funções ocorrem de maneira linear. Donovan e Kernighan (2015) introduzem a ideia de que, em um programa concorrente com duas ou mais *goroutines*, chamadas para diferentes funções podem estar ativas simultaneamente, proporcionando novas possibilidades de execução.

*Goroutines*, é uma característica distintiva do Go, diferenciam-se por não serem *threads* do sistema operacional e também por não serem *green threads*, que são *threads* gerenciadas pelo tempo de execução de uma linguagem. Elas representam um nível mais elevado de abstração conhecido como *coroutines*. As *coroutines*, por sua vez, são sub-rotinas concorrentes, como funções, *closures* ou métodos em Go, caracterizadas pela não preempção, ou seja, não podem ser interrompidas. *Coroutines* proporcionam vários pontos ao longo dos quais é possível a suspensão ou reentrada (COX-BUDAY, 2017). Embora a sintaxe para lançar *goroutines* seja semelhante à chamada de *threads* em outras linguagens, Donovan e Kernighan (2015) esclarecem que as diferenças entre *threads* e *goroutines* são predominantemente quantitativas.

O Código 3, representa uma simples de utilização de *goroutines* em Go. Este exemplo cria duas *goroutines* anônimas usando a palavra-chave `go` antes de uma função anônima. A função `printMessage` é chamada em cada *goroutine* para imprimir uma mensagem específica. A `sync.WaitGroup` é usada para aguardar a conclusão de ambas as *goroutines* antes de sair da função `main`. Isso garante que a função `main` aguarde a conclusão de todas as *goroutines* antes de

Código 3 – Exemplo de *Goroutine*

```
1 func main() {
2     var wg sync.WaitGroup
3     wg.Add(2)
4
5     go func() {
6         defer wg.Done()
7         printMessage("Hello from Goroutine 1")
8     }()
9
10    go func() {
11        defer wg.Done()
12        printMessage("Hello from Goroutine 2")
13    }()
14
15    wg.Wait()
16    fmt.Println("Main function exiting")
17 }
18
19 func printMessage(message string) {
20     fmt.Println(message)
21 }
```

terminar.

As duas *goroutines* são criadas de forma concorrente para executar as funções anônimas que chamam *printMessage*. A utilização de *defer wg.Done()* garante que o contador do *WaitGroup* seja decrementado quando as *goroutines* terminam.

Ao executar esse código, você verá mensagens *"Hello from Goroutine 1"* e *"Hello from Goroutine 2"* sendo intercaladas na saída, demonstrando a execução concorrente das *goroutines*.

A singularidade das *goroutines* no Go está relacionada à sua profunda integração com o tempo de execução da linguagem. Ao contrário de outras implementações, as *goroutines* não especificam seus próprios pontos de suspensão ou reentrada. O tempo de execução do Go monitora dinamicamente o comportamento em tempo de execução das *goroutines*, suspendendo-as automaticamente quando bloqueadas e retomando-as quando desbloqueadas. Isso as torna, de certa forma, preemptivas, mas apenas nos pontos específicos em que a *goroutine* fica bloqueada. Essa colaboração eficaz entre o tempo de execução e a lógica de uma *goroutine* as configura como uma classe especial de *coroutine* (COX-BUDAY, 2017).

### 2.3.2 Pacote *Sync*

O pacote *sync* contém as primitivas de concorrência mais úteis para a sincronização de acesso à memória em um nível mais baixo, como *locks* de exclusão mútua com exceção dos tipos *Once* e *WaitGroup*. Sincronização em níveis mais altos é mais bem realizada através de canais de comunicação (PIKE, 2023h). Go construiu um novo conjunto de primitivas de concorrência sobre as primitivas de sincronização de memória compartilhada, fornecendo assim um conjunto

maior de elementos para trabalhar, essas operações têm seu uso, principalmente em escopos pequenos, como uma *struct*, cabendo assim ao projetista decidir quando a sincronização por memória compartilhada é apropriada (COX-BUDAY, 2017).

Quando a modelagem por troca de mensagem se mostra difícil e complexa, o Go oferece recursos clássicos, tais como *mutexes* e semáforos, por meio da biblioteca *sync* (COX-BUDAY, 2017). Além disso, a biblioteca *sync* disponibiliza outras estruturas que podem ser usadas de maneira complementar aos canais (*channels*) e *goroutines*.

### 2.3.2.1 *WaitGroup*

O mecanismo de *WaitGroup* no Go oferece uma forma eficaz de coordenar a execução de *goroutines*, permitindo que a *goroutine* principal aguarde a conclusão de um conjunto específico delas. Conforme descrito por Pike (2023h), a principal funcionalidade da *WaitGroup* envolve a chamada do método *Add* para definir o número de *goroutines* a serem aguardadas, seguido por cada *goroutine* chamando o método *Done* quando conclui sua execução. A utilização do método *Wait* possibilita que a *goroutine* principal bloqueie sua execução até que todas as *goroutines* tenham sido finalizadas.

Conforme sugerido por Cox-Buday (2017), o *WaitGroup* se destaca como uma ferramenta valiosa para a espera de operações concorrentes quando o resultado dessas operações não é relevante ou quando existem outros meios de coletar esses resultados. No entanto, se essas condições não forem atendidas, a autora recomenda o uso de canais e uma instrução *select* como uma alternativa. A versatilidade e utilidade do *WaitGroup* são evidentes, justificando sua apresentação como um recurso fundamental no desenvolvimento de aplicações concorrentes em Go.

O Código 4 demonstra um exemplo básico de como utilizar o *WaitGroup* to aguardar a finalização de *goroutines*.

Na função principal (*main*), declaramos uma variável do tipo *sync.WaitGroup* chamada *wg*. Essa variável é crucial para sincronizar a execução entre a função principal e as *goroutines*, permitindo que a função principal aguarde a conclusão de todas as *goroutines* antes de encerrar.

Adicionamos 2 ao contador do *WaitGroup*, indicando que estamos esperando duas *goroutines*. Em seguida, criamos duas *goroutines* anônimas utilizando a palavra-chave *Go*. Cada *goroutine* executa uma função anônima que imprime uma mensagem específica, como "*Hello from Goroutine 1*" ou "*Hello from Goroutine 2*".

Ao utilizar *defer wg.Done()*, garantimos que o contador do *WaitGroup* seja decrementado quando cada *goroutine* termina, permitindo que a função principal saiba quando todas as *goroutines* concluíram a execução. Chamamos *wg.Wait()* para bloquear a execução da função principal até que o contador do *WaitGroup* retorne a zero, indicando que todas as *goroutines* terminaram.



Código 4 – Exemplo de *WaitGroup*

```
1 func main() {  
2     var wg sync.WaitGroup  
3     wg.Add(2)  
4  
5     go func() {  
6         defer wg.Done()  
7         printMessage("Hello from Goroutine 1")  
8     }()  
9  
10    go func() {  
11        defer wg.Done()  
12        printMessage("Hello from Goroutine 2")  
13    }()  
14  
15    wg.Wait()  
16    fmt.Println("Main function exiting")  
17 }  
18  
19 func printMessage(message string) {  
20     fmt.Println(message)  
21 }
```

Por fim, imprimimos uma mensagem indicando que a função *main* está saindo. A execução das *goroutines* demonstra a natureza concorrente, e a utilização do *WaitGroup* garante uma coordenação eficiente entre as diferentes partes do programa, proporcionando um controle ordenado da execução concorrente.

### 2.3.2.2 *RWMutex*

Uma abordagem para otimizar o acesso concorrente à memória compartilhada é a utilização do *sync.RWMutex*, conforme proposto por Cox-Buday (2017). O *sync.RWMutex* compartilha o mesmo propósito fundamental que um *Mutex*, protegendo o acesso à memória compartilhada, mas oferece uma funcionalidade adicional de controle mais refinado. Ao solicitar um bloqueio para leitura, o *RWMutex* concede acesso a múltiplos leitores simultâneos, desde que nenhum bloqueio de escrita esteja sendo mantido. Essa característica proporciona uma flexibilidade valiosa, permitindo um equilíbrio entre concorrência e exclusividade no acesso a dados compartilhados. No Código 5 demonstra a utilização do *sync.RWMutex* para controle de acesso concorrente.

Este código cria três funções: *producer*, *observer*, e *test*. A função *producer* representa uma *goroutine* que realiza operações de *Lock* e *Unlock* em um *Locker* fornecido, simulando uma seção crítica. A função *observer* é uma *goroutine* que apenas adquire um *Lock* e o libera imediatamente. A função *test* simula um teste onde uma quantidade especificada de leitores (*observers*) e um produtor (*producer*) operam simultaneamente em uma seção crítica protegida por um *sync.Locker* (que pode ser um *sync.Mutex* ou um *sync.RWMutex*).



Código 5 – Exemplo de *RWMutex*

```

1 func producer(wg *sync.WaitGroup, l sync.Locker) {
2     defer wg.Done()
3     for i := 5; i > 0; i-- {
4         l.Lock()
5         l.Unlock()
6         time.Sleep(1 * time.Millisecond)
7     }
8 }
9
10 func observer(wg *sync.WaitGroup, l sync.Locker) {
11     defer wg.Done()
12     l.Lock()
13     defer l.Unlock()
14 }
15
16 func test(count int, mutex, rwMutex sync.Locker) time.Duration {
17     var wg sync.WaitGroup
18     wg.Add(count + 1)
19     beginTestTime := time.Now()
20     go producer(&wg, mutex)
21     for i := count; i > 0; i-- {
22         go observer(&wg, rwMutex)
23     }
24     wg.Wait()
25     return time.Since(beginTestTime)
26 }
27
28 func main() {
29     tw := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
30     defer tw.Flush()
31     var m sync.RWMutex
32
33     fmt.Fprintf(tw, "Readers\tRWMutex\tMutex\n")
34     for i := 0; i < 5; i++ {
35         count := int(math.Pow(2, float64(i)))
36         fmt.Fprintf(
37             tw,
38             "%d\t%v\t%v\n",
39             count,
40             test(count, &m, m.RLocker()),
41             test(count, &m, &m),
42         )
43     }
44 }

```

2.3.2.3 *Cond*

A implementação de uma variável de condição, como descrita por Pike (2023h), é fundamental para coordenar a execução de *goroutines* em Go, proporcionando um ponto de encontro para a comunicação entre elas em relação à ocorrência de eventos. Cada variável de condição, representada pela estrutura *Cond*, está associada a um *Locker* (frequentemente um *Mutex* ou *RWMutex*), que deve ser mantido durante alterações na condição e chamadas ao método

*Wait*. É crucial notar que, conforme as diretrizes apresentadas por Pike (2023h), uma variável de condição não deve ser copiada após o primeiro uso. No contexto do modelo de memória do Go, a *Cond* estabelece uma ordem de sincronização, indicando que chamadas para *Broadcast* ou *Signal* ocorrem "sincroniza antes" de chamadas para *Wait* que desbloqueiam a execução.

Seguindo as recomendações de Cox-Buday (2017), é destacado que um "evento" refere-se a qualquer sinal arbitrário entre *goroutines*, indicando apenas a ocorrência de um evento sem carregar informações adicionais. Em muitos casos, aguardar por esses sinais torna-se essencial antes de prosseguir com a execução de uma *goroutine*. A definição de eventos, conforme Cox-Buday (2017), destaca a simplicidade e eficácia do uso de canais para muitos casos de uso, onde o *Broadcast* corresponde ao fechamento de um canal, e o *Signal* equivale ao envio em um canal. Essas abstrações facilitam a comunicação entre *goroutines*, contribuindo para o desenvolvimento de sistemas concorrentes robustos em Go.

A estrutura *sync.Cond* em Go é utilizada para implementar variáveis de condição, proporcionando um ponto de encontro para *goroutines* que desejam esperar ou anunciar a ocorrência de eventos. O Código 6 demonstra o uso prático de *sync.Cond*. Neste exemplo, a estrutura *Data* contém um valor inteiro e uma variável de condição (*sync.Cond*). Duas *goroutines* são criadas: uma para atualizar o valor de *Data* a cada 2 segundos e sinalizar a condição, e outra para esperar pelo sinal de atualização e imprimir o valor.

A função *sync.NewCond(&sync.Mutex)* cria uma nova variável de condição associada a um *mutex*. O método *Signal()* é usado para notificar a *goroutine* adormecida que a condição foi satisfeita. A função *Wait()* é usada para aguardar até que a condição seja satisfeita, liberando temporariamente o *mutex* associado.

O uso do *sync.Cond* facilita a coordenação eficiente entre *goroutines*, permitindo que elas cooperem na execução de tarefas específicas quando as condições desejadas são atendidas. Este tipo é particularmente útil em casos onde canais podem ser inadequados ou quando se deseja coordenar o comportamento de múltiplas *goroutines*.

Como ocorre com a maioria das outras funcionalidades do pacote *sync*, o *Cond* é mais eficaz quando utilizado em um escopo específico e restrito, ou quando exposto a um escopo mais amplo por meio de um tipo que o encapsula (COX-BUDAY, 2017).

#### 2.3.2.4 *Once*

Como uma nova primitiva introduzida pelo Go, *Once* é projetado para garantir que uma função seja executada apenas uma vez (TU et al., 2019). Um *Once* é um objeto que realizará exatamente uma ação e não deve ser copiado após o primeiro uso (PIKE, 2023h). Podemos acompanhar no Código 7 um programa representando a utilização de *sync.Once*.

Neste exemplo, a função *initializer* será executada apenas uma vez, independentemente de quantas *goroutines* chamem *once.Do(initializer)*. Isso é alcançado pelo mecanismo

Código 6 – Exemplo de *Cond*

```
1 type Data struct {
2     value int
3     cond *sync.Cond
4 }
5
6 func main() {
7     data := &Data{
8         value: 0,
9         cond: sync.NewCond(&sync.Mutex{}),
10    }
11
12    go func() {
13        for {
14            time.Sleep(2 * time.Second)
15            data.cond.L.Lock()
16            data.value++
17            fmt.Println("Updated value:", data.value)
18            data.cond.Signal()
19            data.cond.L.Unlock()
20        }
21    }()
22
23    go func() {
24        for {
25            data.cond.L.Lock()
26            data.cond.Wait()
27            fmt.Println("Updated value:", data.value)
28            data.cond.L.Unlock()
29        }
30    }()
31
32    select {}
33 }
```

interno do *sync.Once*, que garante que a função seja executada apenas na primeira chamada e, subsequentemente, seja ignorada.

O uso do *sync.Once* é especialmente útil quando há a necessidade de realizar uma inicialização única em um programa concorrente, evitando a repetição desnecessária da mesma operação por várias *goroutines*. Isso é particularmente útil em situações em que a inicialização é custosa ou envolve acesso a recursos compartilhados.

### 2.3.3 Channels

Os canais, primitivas de sincronização em Go inspiradas no CSP de Hoare, são empregados não apenas para sincronizar acesso à memória, mas principalmente para facilitar a comunicação entre *goroutines*, conforme destacado por Cox-Buday (2017). Funcionando como condutores de fluxo de informações, os canais permitem a transmissão de valores entre partes distintas do programa, proporcionando uma forma eficiente de compor funcionalidades em programas de

Código 7 – Exemplo de *Once*

```
1 func main() {
2     var once sync.Once
3
4     initializer := func() {
5         fmt.Println("Initializing...")
6     }
7
8     go func() {
9         once.Do(initializer)
10        fmt.Println("Hello from Goroutine 1.")
11    }()
12
13    go func() {
14        once.Do(initializer)
15        fmt.Println("Hello from Goroutine 2.")
16    }()
17
18    go func() {
19        once.Do(initializer)
20        fmt.Println("Hello from Goroutine 3.")
21    }()
22
23    select {}
24 }
```

qualquer tamanho. A comunicação por meio de canais simplifica a interação entre *goroutines*, permitindo que valores sejam transmitidos e lidos, sem a necessidade de conhecimento detalhado entre as partes envolvidas. A criação de canais pode ser realizada de maneira concisa com o operador `:=`, evidenciando a simplicidade e a eficácia dessa primitiva em Go (COX-BUDAY, 2017). Em paralelo, se as *goroutines* são as atividades concorrentes em um programa Go, os canais representam as conexões entre elas, funcionando como mecanismos de comunicação para a transmissão de valores específicos entre *goroutines*. Cada canal é associado a um tipo específico de elemento, sendo expresso, por exemplo, como *chan int* para um canal que transmite valores inteiros. A criação de canais é realizada utilizando a função *make*, como exemplificado por `'ch := make(chan int)'`, proporcionando uma abordagem simples e poderosa para facilitar a comunicação efetiva entre as atividades concorrentes.

No Código 8, temos um canal chamado `messageChannel` que é usado para comunicação entre duas *goroutines*: uma produtora e outra consumidora. A *goroutine* produtora envia mensagens para o canal, e a *goroutine* consumidora as recebe.

A relação dos canais com a sincronização por passagem de mensagem está no fato de que os canais proporcionam um meio seguro e eficiente para que as *goroutines* se comuniquem trocando mensagens. A sincronização por passagem de mensagem refere-se à troca de dados entre *threads* ou processos para coordenar suas atividades. Em Go, os canais desempenham um papel essencial nesse modelo de comunicação, garantindo que a troca de mensagens seja

Código 8 – Exemplo de *Channels*

```
1 func main() {
2     messageChannel := make(chan string)
3
4     var wg sync.WaitGroup
5
6     wg.Add(1)
7     go func() {
8         defer wg.Done()
9         for i := 1; i <= 5; i++ {
10             message := fmt.Sprintf("Mensagem %d", i)
11             messageChannel <- message
12         }
13         close(messageChannel)
14     }()
15
16     wg.Add(1)
17     go func() {
18         defer wg.Done()
19         for message := range messageChannel {
20             fmt.Println("Recebido:", message)
21         }
22     }()
23
24     wg.Wait()
25 }
```

sincronizada e segura para ambientes concorrentes.

Ao utilizar canais, as *goroutines* podem se comunicar sem a necessidade de *mutexes* ou outras estruturas de sincronização, simplificando o desenvolvimento de programas concorrentes e garantindo a prevenção de condições de corrida. Isso torna os canais uma ferramenta poderosa para implementar a sincronização por passagem de mensagem em Go.

### 2.3.4 *Select*

A instrução *select* em Go desempenha um papel crucial na composição de canais, conectando efetivamente *goroutines* e permitindo a formação de abstrações mais amplas em um programa concorrente, conforme enfatizado por Cox-Buday (2017). Ao ser a "cola que une os canais," o *select* pode ser localmente empregado em uma única função ou tipo, bem como globalmente, na interseção de diversos componentes em um sistema. Além de unir componentes, as instruções *select* podem ser estrategicamente utilizadas em pontos críticos do programa para integrar canais de forma segura, incorporando conceitos como cancelamentos, limites de tempo, espera e valores padrão.

A funcionalidade do *select* assemelha-se a um *switch*, apresentando vários casos e um padrão opcional, conforme destacado por Donovan e Kernighan (2015). Cada caso no *select* especifica uma comunicação, representando operações de envio ou recebimento em algum canal,

junto a um bloco associado de declarações. O *select* aguarda até que uma dessas comunicações esteja pronta para prosseguir, realizando a comunicação associada ao caso correspondente e executando as declarações relacionadas. Caso nenhum caso esteja pronto, um *select* sem casos (*select*) aguarda indefinidamente.

O *select* funciona como um mecanismo de controle de fluxos concorrentes, proporcionando decisões condicionais em operações de escrita ou leitura em canais, conforme mencionado por Cox-Buday (2017) e Donovan e Kernighan (2015). Assim como no *switch*, é possível escolher um comportamento padrão através do caso especial *default*, acionado se todas as operações resultarem em bloqueio. O *select* possibilita a escrita de fluxos de controle comuns em códigos sequenciais clássicos, enriquecendo códigos concorrentes com a capacidade de tomar decisões em múltiplos fluxos. Essa abordagem flexível e poderosa do *select* torna-o uma ferramenta essencial na construção de programas concorrentes robustos em Go.

No Código 9, temos duas *goroutines* produtoras que enviam mensagens para dois canais diferentes (ch1 e ch2). A *goroutine* consumidora utiliza *select* para receber mensagens de ambos os canais. O *select* permite que a *goroutine* consumidora aguarde a comunicação em qualquer um dos canais prontos para enviar, garantindo que a execução não seja bloqueada.

A relação com canais é fundamental no contexto do *select*. O *select* é frequentemente utilizado em conjunto com canais para coordenar a comunicação entre *goroutines*. Cada caso dentro do *select* representa uma possível operação de leitura ou escrita em um canal, e o *select* aguarda até que uma dessas operações esteja pronta para prosseguir. Isso proporciona uma forma eficiente de lidar com múltiplos canais de forma concorrente, permitindo a tomada de decisões dinâmicas com base na disponibilidade de comunicação em diferentes canais.

## 2.4 Aplicações Go

Atualmente Go é a décima linguagem mais popular no Github, ocupando também a décima oitava posição quando considerarmos o percentual de crescimento por contribuidores (GITHUB, 2023). As linguagens de programação não estão mais restritas ao âmbito do desenvolvimento de software convencional. Observamos uma equivalência significativa entre as linguagens mais amplamente utilizadas em projetos desenvolvidos em 2023 em comparação com aquelas predominantes no GitHub. Alguns casos atípicos notáveis incluem Kotlin, Rust, Go e Lua, que demonstraram um aumento mais expressivo em popularidade em projetos mais recentes hospedados no GitHub (GITHUB, 2023). No momento dessa pesquisa foram encontrados 566k repositórios escritos em Go (GITHUB, 2024).

Este estudo selecionou três aplicações de mundo real escritas em Go, incluindo um sistema de contêiner (Docker), um banco de dados (CockroachDB) e um sistema de automação e provisionamento de infraestrutura como código (Terraform). Essas aplicações são projetos de código aberto que ganharam ampla utilização em ambientes de data center.

Código 9 – Exemplo de *Select*

```
1 func main() {
2     ch1 := make(chan string)
3     ch2 := make(chan string)
4
5     go func() {
6         for i := 0; i < 5; i++ {
7             time.Sleep(time.Second)
8             ch1 <- fmt.Sprintf("Message from Channel 1: %d", i)
9         }
10        close(ch1)
11    }()
12
13    go func() {
14        for i := 0; i < 5; i++ {
15            time.Sleep(2 * time.Second)
16            ch2 <- fmt.Sprintf("Message from Channel 2: %d", i)
17        }
18        close(ch2)
19    }()
20
21    for {
22        select {
23            case msg, ok := <-ch1:
24                if ok {
25                    fmt.Println(msg)
26                } else {
27                    fmt.Println("Channel 1 closed.")
28                }
29            case msg, ok := <-ch2:
30                if ok {
31                    fmt.Println(msg)
32                } else {
33                    fmt.Println("Channel 2 closed.")
34                }
35        }
36    }
37 }
```

O Docker é um sistema de containerização que permite a criação, o empacotamento e o envio de aplicações como unidades executáveis portáteis. É um dos sistemas de containerização mais populares do mundo, com mais de 67 mil estrelas no GitHub. O Docker é mantido ativamente por uma comunidade de desenvolvedores e é usado por empresas de todos os tamanhos (DOCKER, 2023b).

O CockroachDB é um banco de dados relacional distribuído que oferece alta disponibilidade, escalabilidade e desempenho. É uma alternativa popular ao PostgreSQL e ao MySQL, e é usado por empresas como Netflix, Spotify e Uber. O CockroachDB tem mais de 28 mil estrelas no GitHub e é mantido ativamente pela equipe da Cockroach Labs (LABS, 2023b).

O Terraform é uma ferramenta de automação e provisionamento de infraestrutura como código (IaC) que permite aos desenvolvedores gerenciar recursos de infraestrutura, como

servidores, redes e armazenamento, usando um único arquivo de configuração. O Terraform é uma ferramenta popular para DevOps e arquitetos de infraestrutura, e é usado por empresas de todos os tamanhos. O Terraform tem mais de 40 mil estrelas no GitHub e é mantido ativamente pela equipe da HashiCorp ([HASHICORP, 2023](#)).

## 2.5 Trabalhos Relacionados

O estudo realizado por [Tu et al. \(2019\)](#) é o primeiro estudo empírico sobre *bugs* de concorrência em aplicações amplamente utilizadas no mundo real e implementadas com a linguagem Go, foram estudados 171 *bugs* de concorrência em seis aplicações de código aberto dentre eles Docker, Kubernetes, etcd, gRPC, CockroachDB e BoltDB, ele se concentra em uma antiga questão fundamenta na programação concorrente e busca definir qual mecanismo de comunicação entre *threads*, passagem de mensagem ou memória compartilhada, qual é menos propensa a erros, sendo Go a linguagem ideal para esse estudo, desde que a mesma oferece estruturas para ambos os modelos, porém Go incentiva o uso de passagem de mensagem, utilizando a primitiva *channels* da linguagem, com a crença de que a passagem explícita de mensagens é menos propensa a erros.

Aplicação	Funções	Funções Anonimas	Total	Por 1000 linhas de código
Docker	33	112	145	0.18
Kubernetes	301	233	534	0.23
etcd	86	211	297	0.67
CockroachDB	27	125	152	0.29
gRPC-Go	14	30	44	0.83
BoltDB	2	0	2	0.22

Tabela 1 – **Quantidade de locais que criam *goroutines***. Aplicação, quantidade de locais onde criam *goroutines* com funções normais, utilizando funções anônimas, total de criações e criações por mil linhas de código. Adaptada de ([TU et al., 2019](#)).

Dessa maneira [Tu et al. \(2019\)](#) realiza uma análise estática e dinâmica do uso de *goroutines* e de outras primitivas de concorrência em Go nas aplicações estudadas, para isso foi feito uma coleta de dados no código-fonte dessas aplicações procurando pela quantidade de linhas de código-fonte que criam *goroutines* e posteriormente por ocorrências dos usos de diferentes primitivas, constatou que a média de locais de criação de *goroutines* por mil linhas de código varia de 0,18 a 0,83, pode-se avaliar os valores por aplicação na [Tabela 1](#), sendo em todas as aplicações a grande maioria utilizando funções anônimas exceto por Kubernetes e BoltDb. Destaca-se também que operações de sincronização de memória compartilhada são mais frequentes do que por passagem de mensagens, e *Mutex* é a primitiva mais amplamente utilizada em todas as aplicações. Para primitivas de passagem de mensagem, *chan* é a mais frequentemente utilizada, variando de 18,48% a 42,99%, [Tabela 2](#).



Aplicação	Memória Compartilhada					Passagem de Msg.		Total
	<i>Mutex</i>	<i>atomic</i>	<i>Once</i>	<i>WaitGroup</i>	<i>Cond</i>	<i>chan</i>	<i>Misc.</i>	
Docker	62.62%	1.06%	4.75%	1.70%	0.99%	27.87%	0.99%	1410
Kubernetes	70.34%	1.21%	6.13%	2.68%	0.96%	18.48%	0.20%	3951
etcd	45.01%	0.63%	7.18%	3.95%	0.24%	42.99%	0	2075
CockroachDB	55.90%	0.49%	3.76%	8.57%	1.48%	28.23%	1.57%	3245
gRPC-Go	61.20%	1.15%	4.2%	7.00%	1.65%	23.03%	1.78%	786
BoltDB	70.31%	2.13%	0	0	0	23.40%	4.26%	47

Tabela 2 – **Uso de primitivas de concorrência.** A coluna referente a *Mutex* contempla *Mutex* e *RWMutex*. Adaptada de (Tu et al., 2019).

No trabalho desenvolvido por Tu et al. (2019) o processo de coleta de *bugs*, foi feito uma busca nos históricos de *commits* do Github, nos repositórios das aplicações definidas no estudo, buscando em seus *logs* de *commits* pelas seguintes palavras-chave: "*race*", "*deadlock*", "*synchronization*", "*concurrency*", "*lock*", "*mutex*", "*atomic*", "*compete*", "*context*", "*once*" e "*goroutine leak*". Foram encontrados 3211 *commits* distintos correspondentes aos critérios de busca, uma amostragem aleatória foi retirada e foram estudados 171 *bugs*, propondo assim uma nova taxonomia para *bugs* de concorrência em Go, ou pode-se dizer um novo método para categorizar *bugs* de concorrência em Go.

A nova taxonomia proposta por Tu et al. (2019) classifica os *bugs* por duas visões, uma relacionada ao comportamento e outra a causa. A dimensão baseada no comportamento separa os *bugs* entre bloqueantes e não-bloqueantes, sendo os bloqueantes aqueles onde uma ou mais *goroutines* ficam involuntariamente presas em sua execução e não conseguem avançar e os não-bloqueantes aqueles que em vez disso, todas as *goroutines* podem concluir suas tarefas, mas seus comportamentos não são desejados, chamamos esses de *bugs* não-bloqueantes. A dimensão da causa separa os *bugs* em aqueles causados pelo uso inadequado de memória compartilhada e aqueles causados pelo uso inadequado de passagem de mensagens.

Constatou-se que dos 171 *bugs* existem um total de 85 *bugs* bloqueantes e 86 *bugs* não-bloqueantes, e há um total de 105 *bugs* causados por proteção incorreta de memória compartilhada e 66 *bugs* causados por passagem incorreta de mensagens. Os *bugs* bloqueantes ocorrem quando uma ou mais *goroutines* realizam operações que aguardam recursos e esses recursos nunca estão disponíveis. Foram descobertos que aproximadamente 42% dos *bugs* bloqueantes são causados por erros na proteção da memória compartilhada, enquanto 58% são causados por erros na passagem de mensagem. Os *bugs* bloqueantes coletados envolvidos com a má proteção de memória compartilhada foram encontrados 28 *bugs* bloqueantes causados por uso indevido de bloqueios (*Mutex*), incluindo dupla aplicação bloqueantes, aplicação de bloqueios em ordens conflitantes e esquecimento de desbloqueio.

*bugs* bloqueantes relacionados ao uso indevido de passagem de mensagem, são atualmente o principal tipo de bug bloqueante nas aplicações estudadas por Tu et al. (2019), erros no uso de

canais para passar mensagens entre *goroutines* causam 29 *bugs* bloqueantes. Muitos dos *bugs* bloqueantes relacionados a canais são causados pela falta de envio para (ou recebimento de) um canal ou pelo fechamento de um canal, o que resultará no bloqueio de uma *goroutine* que espera para receber (ou enviar) para o canal. Em 16 *bugs* bloqueantes, uma *goroutine* é bloqueada em uma operação de canal, enquanto outra *goroutine* é bloqueada em uma operação de trava (*lock*) ou espera (*wait*).

Foi descoberto que cerca de 90% dos *bugs* bloqueantes estudados são corrigidos ajustando primitivas de sincronização, a maioria dos *bugs* bloqueantes estudados (tanto os tradicionais de memória compartilhada quanto os de passagem de mensagem) pode ser corrigida com soluções simples, e muitas correções estão correlacionadas com as causas dos *bugs*. Sugere no fim que devido a alta correlação entre as causas e as soluções nos *bugs* bloqueantes do Go, juntamente com a simplicidade em suas correções, pode-se pensar em desenvolver ferramentas automatizadas ou semi-automatizadas para corrigir esses tipos de *bugs* bloqueantes.

Cerca de 80% dos *bugs* não-bloqueantes coletados são devido a acessos à memória compartilhada não protegidos ou protegidos de maneira incorreta. No entanto, nem todos compartilham as mesmas causas que *bugs* não-bloqueantes em linguagens tradicionais. Cerca de dois terços dos *bugs* não-bloqueantes de memória compartilhada são causados por causas tradicionais. As novas semânticas de *multithread* e novas bibliotecas do Go contribuem para o restante, cerca de um terço. Erros durante a passagem de mensagens também podem causar *bugs* não-bloqueantes e compreendem cerca de 20% dos nossos *bugs* não-bloqueantes coletados. Há 16 *bugs* não-bloqueantes causados pelo uso inadequado de *channels*. Outro tipo de *bug* de concorrência ocorre ao usar *channel* e *select* juntos. No Go, quando várias mensagens são recebidas por um *select*, não há garantia de qual delas será processada primeiro. Essa implementação não determinística do *select* causou 3 *bugs*. Conclui-se que há muitos menos *bugs* não-bloqueantes causados por passagem de mensagens do que por acessos a memória compartilhada.

Aproximadamente 69% dos *bugs* não-bloqueantes foram corrigidos restringindo o tempo, seja adicionando primitivas de sincronização como *Mutex*, ou movendo primitivas existentes. Dez *bugs* não-bloqueantes foram corrigidos eliminando instruções que acessam variáveis compartilhadas ou ignorando as instruções. Quatorze *bugs* foram corrigidos fazendo uma cópia privada da variável compartilhada, e esses *bugs* são todos relacionados à memória compartilhada. 24 *bugs* foram corrigidos por outras primitivas de concorrência e 19 *bugs* foram corrigidos sem o uso de nenhuma primitiva de concorrência.

O estudo concluiu que a passagem de mensagens não torna necessariamente os programas *multithread* menos propensos a erros do que a memória compartilhada, na verdade, a passagem de mensagens é a principal causa de *bugs* bloqueantes. Ela causa menos *bugs* não-bloqueantes do que a sincronização de memória compartilhada e, surpreendentemente, foi até usada para corrigir *bugs* causados por sincronização errada de memória compartilhada. Conclui-se que

a passagem de mensagens oferece uma forma limpa de comunicação entre *threads* e pode ser útil para passar dados e sinais. Mas eles são úteis apenas se usados corretamente, o que exige que os programadores entendam não apenas os mecanismos de passagem de mensagens, mas também outros mecanismos de sincronização do Go. O uso incorreto de bibliotecas Go pode causar *bugs* tanto bloqueantes quanto não-bloqueantes. Existe uma porção não negligenciável de *bugs* não-bloqueantes causados por erros durante a passagem de mensagens, e esses *bugs* não são cobertos por trabalhos anteriores.

O estudo realizado por [Lu et al. \(2008\)](#) abrange as características reais de *bugs* de concorrência, examinando cuidadosamente padrões de *bugs* de concorrência, manifestações e estratégias de correção de 105 *bugs* de concorrência em aplicações do mundo real de código aberto representativos cliente servidor, sendo eles MySQL, Apache, Mozilla e OpenOffice, revelando descobertas interessantes e oferecendo orientações úteis para detecção de *bugs* de concorrência, testes e definição de linguagens de programação concorrente.

O processo de coleta de *bugs* foi realizado utilizando um conjunto de palavras-chave como "*race(s)*", "*deadlock(s)*", "*synchronization(s)*", "*concurrency*", "*lock(s)*", "*mutex(es)*", "*atomic*", "*compete(s)*", e suas variações, filtrando assim meio milhão de relatórios de bug, destes foram escolhidos aleatoriamente 500 relatórios com descrições de causa raiz claras e detalhadas, códigos-fonte e informações de correção de *bugs*, depois de verificado manualmente para garantir que os *bugs* fossem realmente causados por suposições erradas dos programadores sobre a execução concorrente, dessa forma chegou-se a 105 *bugs* de concorrência. Nesse estudo os *bugs* foram separados em dois tipos sendo eles de *deadlock*, que ocorrem quando duas ou mais operações esperam circularmente uma pela outra para liberar o recurso adquirido; e de não *deadlock*, que ocorrem essencialmente quando as intenções de sincronização são violadas, ([LU et al., 2008](#)) os classifica em dois tipos principais com base em suas causas raízes, são estes, *bugs* de violação de ordem e *bugs* de violação de atomicidade. Uma violação de ordem se manifesta como um padrão composto por dois acessos sequenciais de *thread* a uma localização de memória compartilhada, onde pelo menos um dos acessos é uma gravação, enquanto que uma violação de atomicidade ocorre quando um conjunto de acessos em uma região atômica é interferido por outros acessos em uma *thread* diferente, e essa interferência resulta em comportamento não intencional do programa. Classificação a qual se difere do formato utilizado por [Tu et al. \(2019\)](#), o qual classifica os *bugs* em bloqueantes, ou seja, ocorrem quando uma ou mais *goroutines* realizam operações que aguardam recursos e esses recursos nunca estão disponíveis, a definição proposta é mais ampla do que a de *deadlocks* e situações em que não há espera circular, mas ao menos uma *goroutine* espera por recursos que nenhuma outra *goroutine* fornece, dessa maneira podemos ter *bugs* bloqueantes não relacionados a *deadlock*; e *bugs* não-bloqueantes, ou seja, ocorrem quando todas as *goroutines* podem concluir suas tarefas, mas seus comportamentos não são desejados, sendo 80% dos *bugs* não-bloqueantes coletados ocasionados por acessos à memória compartilhada não protegidos ou protegidos de maneira incorreta. No entanto, nem todos compartilham as mesmas causas que *bugs* não-bloqueantes em linguagens tradicionais.

Para cada um dos 105 *bugs*, sendo 74 *bugs* de concorrência não relacionados a deadlock e 31 *bugs* de deadlock. Para cada um desses [Lu et al. \(2008\)](#) examinou cuidadosamente seu relatório de *bug*, código-fonte correspondente, *patches* relacionados e discussões entre programadores, fornecendo, assim, uma compreensão relativamente completa dos padrões de *bugs*, condições de manifestação, estratégias de correção e processos de diagnóstico como podemos observar na [Tabela 3](#).

Aplicação	# de Bugs			Deadlock
	Não relacionados a Deadlock			
	Ordem	Atomicidade	Outros	
MySQL	14	1	1	9
Apache	7	6	0	4
Mozilla	29	15	0	16
OpenOffice	3	2	1	2
Total	51	24	2	31

Tabela 3 – **Contagem e classificação de Bugs**. Adaptado de [Lu et al. \(2008\)](#)

O estudo de ([LU et al., 2008](#)) inclui muitas descobertas interessantes e implicações para a detecção de *bugs* de concorrência, testes e design de linguagens de programação concorrente como pode ser melhor observado na [Tabela 4](#). Pesquisas futuras podem se beneficiar do estudo realizado por [Lu et al. \(2008\)](#) em vários aspectos, podendo utilizar este para projetar novas ferramentas de detecção de *bugs* para lidar com *bugs* de várias variáveis e violações de ordem; testar em pares as *threads* do programa concorrente e focar em ordens parciais de pequenos grupos de acessos à memória para aproveitar ao máximo o esforço de teste; podem ter melhores recursos de linguagem para suportar semânticas de "ordem" para facilitar ainda mais a programação concorrente.

Descobertas	Implicações
72 dos 74 <i>bugs</i> de concorrência não relacionados a <i>deadlock</i> examinados são abrangidos por dois padrões simples: violação de atomicidade e violação de ordem.	A detecção de <i>bugs</i> em programas concorrentes, os testes e o design de linguagens devem primeiro se concentrar nesses dois principais padrões de <i>bugs</i> .
24 dos 74 <i>bugs</i> de concorrência não relacionados a <i>deadlock</i> examinados são <i>bugs</i> de ordem, os quais não são abordados por trabalhos anteriores de detecção de <i>bugs</i> .	Novas técnicas de detecção de <i>bugs</i> são desejadas para lidar com <i>bugs</i> de ordem.
A manifestação de 101 dos 105 <i>bugs</i> de concorrência examinados envolve no máximo duas <i>threads</i> .	Os testes de programas concorrentes podem testar em pares as <i>threads</i> do programa, o que reduz a complexidade dos testes sem perder muito na capacidade de expor <i>bugs</i> essa descoberta sugere que os testes podem se concentrar nas ordens de execução entre acessos de cada par de <i>threads</i> . Essa técnica de teste em pares pode evitar que a complexidade dos testes aumente exponencialmente com o número de <i>threads</i> . Ao mesmo tempo, poucos <i>bugs</i> de concorrência seriam perdidos.
A manifestação de 7 de 31 <i>bugs</i> de concorrência de <i>deadlock</i> envolve apenas uma <i>thread</i> .	Este tipo de <i>bug</i> é relativamente fácil de detectar e evitar. Técnicas de detecção de <i>bugs</i> e linguagens de programação podem tentar eliminar esses <i>bugs</i> simples primeiro.
49 de 74 dos <i>bugs</i> de concorrência não relacionados a <i>deadlock</i> examinados envolvem apenas uma variável.	Focar nos acessos concorrentes a uma variável é uma boa simplificação para a detecção de <i>bugs</i> de concorrência.
34% <i>bugs</i> de concorrência não relacionados a <i>deadlock</i> envolve mais de uma variável	É preciso novas ferramentas de detecção de <i>bugs</i> de concorrência para lidar com <i>bugs</i> de concorrência de várias variáveis. <i>bugs</i> de concorrência de várias variáveis geralmente ocorrem quando acessos não sincronizados a variáveis correlacionadas causam um estado inconsistente no programa. Conexões semânticas entre variáveis são comuns e, portanto, <i>bugs</i> de concorrência de várias variáveis também são comuns.
30 de 31 dos <i>bugs</i> de concorrência de <i>deadlock</i> examinados envolvem no máximo dois recursos.	Testes de programas concorrentes orientados a <i>deadlock</i> podem testar em pares a ordem entre aquisição e liberação de dois recursos.
92% dos <i>bugs</i> de concorrência analisados manifestam-se quando é imposta uma ordem parcial entre não mais do que 4 acessos à memória.	Testar ordens parciais entre cada pequeno grupo de acessos pode expor a maioria dos <i>bugs</i> de concorrência e simplificar o espaço de <i>interleaving</i> de exponencial para polinomial.
67 de 74 dos <i>bugs</i> de concorrência examinados podem manifestar-se de forma determinística se certas ordens entre no máximo quatro acessos à memória forem aplicadas.	
30 de 31 dos <i>bugs</i> de <i>deadlock</i> examinados podem manifestar-se de forma determinística se certas ordens entre no máximo quatro operações de aquisição/liberação de recursos forem aplicadas.	O teste de programas concorrentes pode se concentrar na ordem parcial entre pequenos grupos de acessos. Isso simplifica o espaço de teste de <i>interleaving</i> de exponencial para polinomial em relação ao número total de acessos, com pouca perda na capacidade de expor <i>bugs</i> .
Apenas 20 dos 74 <i>bugs</i> de concorrência de não <i>deadlock</i> utilizam travas como principal estratégia de correção.	Não há uma solução única para corrigir <i>bugs</i> de concorrência.
61% dos <i>bugs</i> de <i>deadlock</i> analisados são corrigidos ao impedir que uma <i>thread</i> adquira um recurso (por exemplo, uma trava). Essa correção pode introduzir <i>bugs</i> de concorrência não relacionados a <i>deadlocks</i> .	Corrigir <i>bugs</i> de <i>deadlock</i> pode resultar na introdução de <i>bugs</i> de concorrência não relacionados a <i>deadlocks</i> . É necessário auxílio especial para garantir a correção das correções de <i>bugs</i> de <i>deadlock</i> .

Tabela 4 – *Descobertas sobre as características reais dos bugs de concorrência do mundo real e suas implicações* para a detecção de *bugs* de concorrência. Adaptada de (LU et al., 2008)

# 3

## Metodologia

Essa seção descreve e demonstra a metodologia utilizada durante o desenvolvimento deste estudo através das etapas, durante a escolha das aplicações a serem estudadas, a escolha dos *bugs* e da classificação dos *bugs*.

### 3.1 Aplicações

Foram selecionadas 3 aplicações de código aberto, sendo um sistema de contêineres (Docker) (DOCKER, 2023a), um sistema um banco de dados (CockroachDB) (LABS, 2023a) e uma ferramenta de *IaC* (Infraestrutura como Código) de código aberto para provisionar e gerenciar a infraestrutura de nuvem (TERRAFORM, 2023). Essas são aplicações de código aberto maduras e amplamente utilizadas em ambientes de mundo real. Todas as aplicações tem pelo menos 9 anos de histórico de desenvolvimento e são ativamente mantidas pelos desenvolvedores na atualidade, tendo crescendo cada vez mais em uso conforme o passar do tempo. As aplicações selecionadas tem tamanhos médios a grandes, com linhas de código variando de 9 mil a mais de 2 milhões. Foram escolhidas 3 aplicações com finalidades diferentes com o intuito de não se fechar a apenas um tipo de utilização das primitivas de concorrência em Go. CoackroachDB especificamente foi uma escolha influenciada pelo trabalho de Tu et al. (2019), enquanto Docker além de também fazer parte do estudo o qual este trabalho estende, foi escolhido assim como o Terraform, por seu tamanho, sua popularidade no ambiente de produção e pela familiaridade com as ferramentas.

### 3.2 *bugs*

Para coletar *bugs* de concorrência, foi filtrado os históricos de *commits* dos repositórios no Github das três aplicações, buscando em seus *logs* de *commits* por palavras chave relacionadas a concorrência: "*race*", "*deadlock*", "*synchronization*", "*concurrency*", "*lock*", "*mutex*", "*atomic*",

"*compete*", "*context*", "*once*" e "*goroutine leak*". Essas palavras chave são as mesmas utilizadas em estudos de *bugs* de concorrência [Lu et al. \(2008\)](#) e também no trabalho de [Tu et al. \(2019\)](#), o qual adiciona as novas palavras a busca como "*context*" e "*once*" pois estão relacionadas a novas primitivas ou bibliotecas de concorrência introduzidas pelo Go e "*goroutine leak*" pois está relacionado ao problema específico do Go. Além das palavras chaves, foram filtrados *logs* de *commits* que tenham sido criados depois de 2020 e que já tenham sido fechados antes de 2024 (2021-2023). Foram encontrados no total XXXX *bugs* distintos nas três aplicações que correspondem com o critério de busca estabelecido, a quantidade de resultados encontrados para cada aplicação pode ser encontrado na [Tabela 5](#).

Aplicação	# de logs de commits
Docker	857
Terraform	8647
CoackroachDB	21545
<b>Total</b>	31049

Tabela 5 – **Total de logs de commits**. Quantidade total de resultados para os critérios de busca definido. Retirado de ([DOCKER, 2023b](#)), ([TERRAFORM, 2023](#)), ([LABS, 2023b](#))

Em seguida, foi amostrado aleatoriamente os *commits* filtrados, eles foram identificados e verificados manualmente para garantir que os *bugs* sejam realmente causados por suposições erradas dos programadores sobre a execução concorrente ou por problemas na utilização de novas primitivas e paradigmas concorrentes do design da linguagem. Muitos dos *logs* de *commits* relacionados a *bugs* também mencionam os relatórios de *bugs* correspondentes, estes relatórios também são considerados para a análise do *bug*. Foram assim selecionados um total de XXX *bugs* com detalhes sobre a causa, códigos-fonte e informações sobre a correção de *bugs*.

### 3.3 Classificação

Para realizar a classificação dos *bugs* será utilizada uma mescla entre os métodos propostos por [Lu et al. \(2008\)](#) e [Tu et al. \(2019\)](#). Dessa forma focaremos em quatro aspectos das características de *bugs* de concorrência: padrão de *bug*, manifestação e estratégia de correção de *bugs*.

A maioria dos estudos sobre *bugs* de concorrência categorizam os *bugs* entre *bugs* de *deadlock* e *bugs* não relacionados a *deadlock* como podemos notar no estudo de [Lu et al. \(2008\)](#), onde *deadlocks* incluem situações em que há uma espera circular entre várias *threads* ([TU et al., 2019](#)) e onde *bugs* não relacionados a *deadlock* são aqueles nos quais as intenções de sincronização são violadas ([LU et al., 2008](#)). Porém para este trabalho estaremos utilizando a definição da dimensão de comportamento adotada por [Tu et al. \(2019\)](#), separando-a em *bugs* bloqueantes, quando uma ou mais *goroutines* ficam involuntariamente presas em sua execução e não conseguem avançar, sendo uma definição mais ampla do que *deadlocks* e inclui situações em



que não há espera circular, mas uma (ou mais) *goroutines* esperam por recursos que nenhuma outra *goroutine* fornece; e *bugs* não-bloqueantes que ocorrem sempre que todas as *goroutines* podem concluir suas tarefas, mas seus comportamentos não são desejados. Sendo assim os *bugs* serão classificados entre bloqueantes e não-bloqueantes.

No aspecto do padrão de *bug*, classificaremos os *bugs* bloqueantes em duas categorias utilizando a segunda dimensão de classificação proposta por [Tu et al. \(2019\)](#), aqueles ocasionados por uma má proteção de memória compartilhada, ou seja, aqueles causados por operações travadas, destinadas a proteger acessos à memória compartilhada ou e aqueles causados por uso indevido de passagem de mensagem. Os *bugs* não-bloqueantes também serão classificados em aqueles causados por falha em proteger a memória compartilhada e aqueles que possuem erros com passagem de mensagem.

Quando falamos do aspecto causa utilizaremos a mesma definição proposta por [Tu et al. \(2019\)](#), onde a ideia é categorizar as causas dos *bugs* de concorrência pelo modo como diferentes *threads* se comunicam, considerando que Go permite comunicação acessando memória compartilhada ou por passagem de mensagens, podendo assim ajudar programadores e pesquisadores a escolherem melhores maneiras de realizar a comunicação entre *threads* e a detectar e evitar erros potenciais ao realizar essa comunicação, além de fornecer dados para possíveis melhorias em ferramentas de detecção de *bugs* de concorrência.

Para o aspecto características de manifestação, é analisado a condição necessária para que cada bug de concorrência possa se manifestar, denominada por [Lu et al. \(2008\)](#) de condição de manifestação, depois é discutido sobre os *bugs* com base em quantas *goroutines*, quantas variáveis, quantas primitivas especializadas de concorrência e quantos acessos estão envolvidos em suas condições de manifestação

E por ultimo para a estratégia de correção de *bugs*, será estudado tanto a estratégia de correção do *patch* final quanto os erros em *patches* intermediários, também será analisado a vida útil dos *bugs* estudados, ou seja, o tempo desde a adição do código com *bug* no software até sua correção no software. Como observado a maioria dos *bugs* estudados por [Tu et al. \(2019\)](#) possui uma longa vida útil, ou seja, demoram para serem detectados desde sua adição ao código, porém assim que são identificados são rapidamente corrigidos, demonstrando a dificuldade de acionar e detectar esses tipos de erros e também que por serem corrigidos rapidamente, são *bugs* que não podem ser considerados triviais, merecendo assim uma análise mais detalhada.

Como exemplo, Docker#43112 na [Figura 1](#), a solicitação foi aberta em 30 de Dezembro de 2021, esse *bug* só foi finalizado e mesclado a ramificação principal em 16 de Janeiro de 2023. Trata-se de um erro de *deadlock*, onde caso *bp.bufLen* for igual a zero, então *bp.wait.Wait()* irá bloquear *func(bp \*BytesPipe) Read(p []byte) (n int, err error)*, então se caso *bp.bufLen* for maior ou igual a *blockThreshold* em *func(bp \*BytesPipe) Write(p []byte) (int, error)* irá bloquear em *bp.wait.Wait()*, nesse caso como existem dois *bp.wait.Wait()* acontece então o bloqueio, logo considerando o padrão desse bug segundo nossa classificação, este é um *bug* bloqueante, mais



especificamente de *deadlock*.

```

32 -     mu      sync.Mutex
33 -     wait    *sync.Cond
34 -     buf      []*fixedBuffer
35 -     buflen   int
36 -     closeErr error // error to return from next Read. set to nil if not closed.
37 }
38
39 // NewBytesPipe creates new BytesPipe, initialized by specified slice.
40
41 @@ -85,6 +86,9 @@ loop0:
42
43     85
44     86         // make sure the buffer doesn't grow too big from this write
45     87         for bp.buflen >= blockThreshold {
46
47
48     88             bp.wait.Wait()
49     89             if bp.closeErr != nil {
50     90                 continue loop0
51
52 @@ -129,7 +133,9 @@ func (bp *BytesPipe) Read(p []byte) (n int, err error) {
53
54     129             if bp.closeErr != nil {
55     130                 return 0, bp.closeErr
56     131             }
57
58     132             bp.wait.Wait()
59
60     133             if bp.buflen == 0 && bp.closeErr != nil {
61     134                 return 0, bp.closeErr
62     135             }

```

Figura 1 – **Bug #43112**. Relacionado a Deadlock. Retirado de [Docker \(2023b\)](#)

Para a resolução, [Figura 2](#), foram adicionadas variáveis e blocos de controle, além de se utilizar de `sync.Cond.Broadcast()`, para acordar todas as *goroutines* que estão aguardando por uma condição, evitando assim a espera circular que ocasiona o bloqueio, dessa maneira deixamos de ter um *deadlock*, apesar de possivelmente termos resolvido também o *bug* de concorrência, podemos notar durante as discussões a dificuldade de se replicar o *bug*, mesmo quando fornecido os detalhes para fazê-lo, ou até chegar a um consenso se o código submetido corrigiria de fato os problemas ou seria apenas uma solução paliativa, por fim esta é atualmente a correção em vigência para esse caso.

```
31  type BytesPipe struct {
32  +      mu      sync.Mutex
33  +      wait    *sync.Cond
34  +      buf      []*fixedBuffer
35  +      buflen   int
36  +      closeErr error // error to return from next Read. set to nil if not closed.
37  +      readBlock bool // check read BytesPipe is Wait() or not
38  }
39
40  // NewBytesPipe creates new BytesPipe, initialized by specified slice.
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87      // make sure the buffer doesn't grow too big from this write
88      for bp.buflen >= blockThreshold {
89  +          if bp.readBlock {
90  +              bp.wait.Broadcast()
91  +          }
92          bp.wait.Wait()
93          if bp.closeErr != nil {
94              continue loop0
95          }
96      }
97
98
99
100
101
102
103      if bp.closeErr != nil {
104          return 0, bp.closeErr
105      }
106
107  +      bp.readBlock = true
108      bp.wait.Wait()
109  +      bp.readBlock = false
110      if bp.buflen == 0 && bp.closeErr != nil {
111          return 0, bp.closeErr
112      }
113  }
```

Figura 2 – Resolução de *Bug #43112*. Relacionado a *Deadlock*. Retirado de [Docker \(2023b\)](#)

# Referências

BRYANT, R. E.; O'HALLARON, D. R. *Computer systems: a programmer's perspective*. [S.l.]: Prentice Hall, 2011. Citado 3 vezes nas páginas 13, 14 e 15.

CORMEN, T. et al. *Book: introduction to algorithms*. [S.l.]: MIT Press, 2009. Citado na página 18.

COX-BUDAY, K. *Concurrency in Go: Tools and Techniques for Developers*. [S.l.]: "O'Reilly Media, Inc.", 2017. Citado 16 vezes nas páginas 4, 12, 13, 14, 15, 16, 17, 20, 21, 22, 23, 25, 26, 27, 28 e 29.

DOCKER. *Docker - Build, Ship, and Run Any App, Anywhere*. 2023. Disponível em: <<https://www.docker.com/>>. Acesso em: 15 dez 2023. Citado 3 vezes nas páginas 8, 9 e 37.

DOCKER. *Docker v20.10.3*. 2023. Disponível em: <<https://github.com/docker/docker>>. Acesso em: 21 dez 2023. Citado 6 vezes nas páginas 2, 3, 30, 38, 40 e 41.

DONOVAN, A. A.; KERNIGHAN, B. W. *The Go programming language*. [S.l.]: Addison-Wesley Professional, 2015. Citado 5 vezes nas páginas 14, 16, 20, 28 e 29.

GITHUB. *The State of Open Source and AI*. 2023. Disponível em: <<https://github.blog/2023-11-08-the-state-of-open-source-and-ai/>>. Acesso em: 22 dez 2023. Citado na página 29.

GITHUB. *Repositórios Go*. 2024. Disponível em: <<https://github.com/search?q=go+language%3AGo&type=repositories&l=Go>>. Acesso em: 22 dez 2023. Citado na página 29.

GO. *Effective Go*. 2023. Disponível em: <[https://go.dev/doc/effective\\_go/](https://go.dev/doc/effective_go/)>. Acesso em: 15 dez 2023. Citado na página 8.

Golang Team. *Add a Partial Deadlock Detector For Go*. 2023. Disponível em: <<https://github.com/golang/go/issues/13759>>. Acesso em: 21 dez 2023. Citado na página 16.

GOOGLE. *A high performance, open source, general RPC framework that puts mobile and HTTP/2 first*. 2023. Disponível em: <<https://github.com/grpc/grpc-go>>. Acesso em: 15 dez 2023. Citado na página 8.

HASHICORP. *Terraform*. 2023. Disponível em: <<https://github.com/hashicorp/terraform>>. Acesso em: 21 dez 2023. Citado na página 31.

HECTANE. *Lightweight SMTP client written in Go*. 2023. Disponível em: <<https://github.com/hectane/smtp>>. Acesso em: 15 dez 2023. Citado na página 8.

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. [S.l.]: Elsevier, 2011. Citado na página 7.

ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. 3. ed. [S.l.], 2011. Disponível em: <<<https://www.iso.org/standard/50372.html>>>. Citado na página 8.

LABS, C. *CockroachDB - Distributed SQL for your applications*. 2023. Disponível em: <<https://www.cockroachlabs.com/>>. Acesso em: 15 dez 2023. Citado 3 vezes nas páginas 8, 9 e 37.

LABS, C. *CockroachDB v2.2.3*. 2023. Disponível em: <<https://github.com/cockroachdb/cockroach>>. Acesso em: 21 dez 2023. Citado 3 vezes nas páginas 3, 30 e 38.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. In: *Concurrency: the Works of Leslie Lamport*. [S.l.: s.n.], 2019. p. 179–196. Citado na página 12.

LU, S. et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. [S.l.: s.n.], 2008. p. 329–339. Citado 8 vezes nas páginas 3, 8, 9, 34, 35, 36, 38 e 39.

ORACLE. *JDK 1.1 for Solaris Developer's Guide*. [S.l.]: Sun Microsystems, 2000. Citado na página 8.

PIKE, R. *Go at Google: Language Design in the Service of Software Engineering*. 2012. Disponível em: <[https://go.dev/talks/2012/splash.article#TOC\\_13](https://go.dev/talks/2012/splash.article#TOC_13)>. Acesso em: 21 dez 2023. Citado na página 19.

PIKE, R. e. a. *Documentation*. 2023. Disponível em: <<https://go.dev/doc/>>. Acesso em: 21 dez 2023. Citado na página 11.

PIKE, R. e. a. *Frequently Asked Questions (FAQ)*. 2023. Disponível em: <<https://go.dev/doc/faq>>. Acesso em: 21 dez 2023. Citado na página 11.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): Does Go have a runtime?* 2023. Disponível em: <<https://go.dev/doc/faq#runtime>>. Acesso em: 21 dez 2023. Citado na página 12.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): What is the history of the project?* 2023. Disponível em: <<https://go.dev/doc/faq#history>>. Acesso em: 21 dez 2023. Citado na página 12.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): Why build concurrency on the ideas of CSP?* 2023. Disponível em: <<https://go.dev/doc/faq#csp>>. Acesso em: 21 dez 2023. Citado na página 12.

PIKE, R. e. a. *Frequently Asked Questions (FAQ): Why goroutines instead of threads?* 2023. Disponível em: <<https://go.dev/doc/faq#goroutines>>. Acesso em: 21 dez 2023. Citado na página 20.

PIKE, R. e. a. *The Go Programming Language Specification*. 2023. Disponível em: <<https://go.dev/ref/spec>>. Acesso em: 21 dez 2023. Citado na página 11.

PIKE, R. e. a. *sync package*. 2023. Disponível em: <<https://pkg.go.dev/sync@go1.21.5>>. Acesso em: 21 dez 2023. Citado 5 vezes nas páginas 12, 21, 22, 24 e 25.

POSOVSZKY, P. *Improving Data Locality in Distributed Processing of Multi-Channel Remote Sensing Data with Potentially Large Stencils*. Tese (Doutorado) — Ludwig-Maximilians-Universitat Munchen, 2020. Citado na página 7.

PRANSKEVICHUS, E.; SELIVANOV, Y. *What's New in Python 3.5*. 2015. Disponível em: <<URL>>. Acesso em: 15 dez 2023. Citado na página 8.

STACKOVERFLOW. *Stack Overflow Developer Survey 2023*. 2023. Disponível em: <<https://survey.stackoverflow.co/2023/#most-popular-technologies-new-collab-tools-learn>>. Acesso em: 02 fev 2024. Citado na página 8.

- STOICA, I. et al. The evolution of multicore processors: A survey of hardware trends and software challenges. *ACM Computing Surveys (CSUR)*, v. 47, p. 1–38, 2014. Citado na página 8.
- TANENBAUM, A.; FILHO, N. M. Sistemas operacionais modernos (vol. 4). *Único*, (Copyright, 1992), p. 152, 2016. Citado 3 vezes nas páginas 13, 14 e 19.
- TERRAFORM. *Terraform - Infrastructure as Code*. 2023. Disponível em: <<https://www.terraform.io/>>. Acesso em: 15 dez 2023. Citado 5 vezes nas páginas 3, 8, 9, 37 e 38.
- TU, T. et al. Understanding real-world concurrency bugs in go. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. [S.l.: s.n.], 2019. p. 865–878. Citado 11 vezes nas páginas 3, 9, 10, 18, 25, 31, 32, 34, 37, 38 e 39.
- YIN, Z. et al. How do fixes become bugs? In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.: s.n.], 2011. p. 26–36. Citado na página 8.