UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO LUIS DA SILVA GUIO SOARES

ESTUDO DE GO E OS SEUS RECURSOS PARA CONCORRÊNCIA

Modelo em Latex

RIO DE JANEIRO
2019

JOÃO LUIS DA SILVA GUIO SOARES

ESTUDO DE GO E OS SEUS RECURSOS PARA CONCORRÊNCIA

Modelo em Latex

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Silvana Rossetto

Co-orientador:

RIO DE JANEIRO

2019

CIP - Catalogação na Publicação

R484t

Ribeiro, Tatiana de Sousa
Titulo / Tatiana de Sousa Ribeiro. -- Rio de Janeiro, 2018.
44 f.

Orientador: Maria da Silva.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2018.

1. Assunto 1. 2. Assunto 2. I. Silva, Maria da, orient. II. Titulo.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

JOÃO LUIS DA SILVA GUIO SOARES

ESTUDO DE GO E OS SEUS RECURSOS PARA CONCORRÊNCIA
Modelo em Latex

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em ___ de _____ de _____

BANCA EXAMINADORA:

Nome do Professor Orientador
Titulação (Instituição)

Nome do Professor1
Titulação (Instituição)

Nome do Professor2
Titulação (Instituição)

Em homenagem a todos que me engradeceram para a conclusão dessa jornada.

AGRADECIMENTOS

Agradeço à todo o corpo docente do departamento de Ciência da Computação da UFRJ, em especial à professora Silvana Rossetto.

Epígrafe: É um item onde o autor apresenta a citação de um texto que seja relacionado com o tema do trabalho, seguido da indicação de autoria do mesmo.
(texto iniciando do meio da página alinhado à direita)

"Few are those who see with their
own eyes and feel with their own hearts."

Albert Einstein
(Nome do autor da epígrafe)

RESUMO

Resumo em português. O texto deve ser digitado ou datilografado em um só parágrafo com **espaçamento simples** e conter de **150 a 500** palavras. Utilizar a terceira pessoa do singular, os verbos na voz ativa e evitar o uso de símbolos e contrações que não sejam de uso corrente. O resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão **Palavras-chave:**, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chave: latex. abntex. editoração de texto.

ABSTRACT

Abstract in english. The text should be typed or typed in a single paragraph with **single spacing** and contain between 150 and 500 words. Use the third person singular, the verbs in the active voice and avoid the use of symbols and contractions that are not of current use.

Keywords: latex. abntex. text editoration.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

Fig.	Area of the i^{th} component
456	Isto é um número
123	Isto é outro número
Bibliot.	Biblioteconomia
Inform.	Informática
ABNT	Associação Brasileira de Normas Técnicas
I^2C	Inter-Integrated Circuit
SRAM	Static Random-Access Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
LED	Light-Emitting Diode
MLP	Modulação por Largura de Pulso
PWM	Pulse-Width Modulation
PID	Proportional–Integral–Derivative
RAM	Random-Access Memory
API	Application Programming Interface
GPL	GNU General Public License
GNU	GNU's Not Unix
iid	Independente e identicamente distribuídas

LISTA DE SÍMBOLOS

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence
\$	subcampo

SUMÁRIO

1 INTRODUÇÃO

(Importante: somente a partir da introdução as páginas do trabalho são numeradas. Utilizam-se algarismos arábicos, sendo que, a contagem das páginas inicia na folha de rosto.)

O texto deverá ser digitado em espaço 1,5. O parágrafo deverá apresentar um recuo na primeira linha a 1,25cm da margem esquerda, não contendo espaçamento entre um parágrafo e outro.

Devem ser digitados em espaços simples: as citações com mais de 3 linhas (citações longas), notas, resumo, referências, legendas de ilustração e de tabelas, bem como partes da capa e da folha de rosto.

Os títulos das seções devem ser separados do início do texto que os precedem ou os sucedem por um espaço (1,5).

2 CONCORRÊNCIA

Concorrente, na sua etimologia, vem do latim *concurrens*, que significa "acontece ao mesmo tempo"[citar um dicionário]. Em computação, está comumente associado a ideia de não dependência entre programas em execução, o que permitiria uma simultaniedade de diversos fluxos de execuções numa única máquina [citar algo].

Antes de surgirem sistemas operacionais multitarefas os programas que eram sempre computados do inicio ao fim. Este recebia um conjunto de entrada de dados (*input*), e retornava um conjunto de saída de dados (*output*). Com o surgimento dos sistemas operacionais multitarefas, ou multiprogramação, a execução concorrente desses programas tornou-se possível. Porém, neste contexto, é importante levantar que a nível de instruções numa unidade central de processamento (CPU, ou processador) ainda não existia numa simultaniedade, ou um paralelismo, real. Num determinado instante do tempo, apenas uma única instrução pode ser executada por um processador. O que é feito para permitir a concorrência de programas é uma troca de contexto entre os diversos "fluxos de processamento", definidos como processos (2.1), que são executados pelo processador, dando assim a impressão da simultaniedade das tarefas, apesar de estarem sendo executadas uma única por vez, ainda que sem necessitar que um programa termine para que comece a executar um outro. Com o surgimento de processadores com mais de um núcleo de processamento, o paralelismo de computações tornou-se possível, tendo então um cenário onde há de fato mais de uma instrução sendo executadas ao mesmo tempo.

2.1 PROCESSOS E THREADS

Processos são a abstração oferecida pelo sistema operacional para um programa em execução. O sistema operacional fica responsável pela execução concorrente dos diversos processos da máquina, que permite o uso convencional de computadores atualmente, em que um usuário executa diversos programas simultaneamente (BRYANT; RICHARD; RICHARD, 2003).

É por meio da abstração de processos que um programa se comunica com as outras entidades, podendo ser *hardwares* disponíveis e ou mesmo outros processos. Em sistemas operacionais modernos, cada processo recebe um espaço de memória para sua execução. Assim evita-se conflitos com outros processos, evitando problemas de integridade dos dados do seu programa.

Os processos, porém, não são a menor abstração de um fluxo de execução de um programa. Um processo possui uma ou mais *threads* no sistema operacional, que permite em máquinas modernas o uso dos processadores com mais de um núcleo de processamento. São as chamadas de *threads*.

Threads (do inglês, linha ou fio) é a unidade de fluxo de execução de um programa. Um processo é composto de diversas *threads*, e a memória e o código é compartilhada entre elas, enquanto cada uma possui seus próprios registradores e ponteiro de instrução. A troca de execução entre *threads* é menos custosa do que entre processos, dado que existe uma troca de contexto muito mais complexa. *Threads* podem ser coordenadas preemptivamente, onde há uma disputa pelo uso de recurso (núcleos de processamento), ou cooperativas em que a troca de fluxo é controlada. Em sistemas operacionais modernos, as *threads* do sistema operacional, também chamadas de *kernel threads* são comumente preemptivas. As instruções sendo executadas no processador sempre pertencem a algum processo, que pode ter uma ou mais *threads* executando.

Por consequência da *thread* utilizar o espaço de memória do processo, temos que ela não consegue existir fora de um processo: sem o processo, não teria qual o código deve ser executado e nem onde a informação seria guardada. Todo processo possui, então, pelo menos uma *thread*, comumente chamada de *main thread*, ou fluxo de execução principal.

O modelo de threads veio para suprir a necessidade de mais velocidade de processamento a partir de *multithreading*, onde diversos fluxos executam paralelamente em núcleos diferentes do processador para acelerar uma determinada execução. Áreas de inteligência artificial, computação numérica, e servidores em geral se beneficiaram deste modelo.

2.1.1 *Kernel thread* vs *User-space thread*

Existem dois tipos principais de *threads* que diferem como você pensa na concorrência de seu problema, que são as chamadas de espaço de usuário (*user-space*), e as já mencionadas *kernel threads*. A diferença reside na responsabilidade pela coordenação de execução dos fluxos: na de espaço de usuário, um programa é o responsável por essa coordenação, sem a visão global que o sistema operacional possui. Exemplos são o ambiente de tempo de execução do Go e a máquina virtual do Java, não restrito a estes. Já nas *kernel threads*, como o próprio nome indica, o sistema operecional fica responsável por essa coordenação. É comum em sistemas UNIX e em Windows essas *threads* serem do padrão POSIX.

2.2 COMUNICAÇÃO ENTRE FLUXOS DE EXECUÇÃO

No início do capítulo 2, foi mencionado que um programa de computador apenas recebia um conjunto de *input*, realizava suas computações, e retornava um *output*. Neste cenário, não existe a preocupação sobre a disponibilidade de recursos: se um programa necessitasse acessar um determinado arquivo, ou algum *hardware* da máquina, há uma garantia de que aquele recurso está disponível e pronto para o uso.

No momento em que os processos passam a concorrer por tempo de processamento, toda esta garantia é perdida. Agora, dois ou mais programas diferentes executando concorrentemente numa máquina podem necessitar escrever em um mesmo arquivo num

disco, por exemplo, ao mesmo tempo que um ou mais outros processos estão lendo desse arquivo. Tanto o *output* quanto o *input* dos programas estão comprometidos nesse cenário, e podemos ter diversos erros de execução. Um programa deve ter sua corretude garantida, deve sempre ter um resultado previsível e correto.

Com essa simples mudança já é trazido a tona um dos maiores problemas da computação concorrente: o sincronismo de recursos. E disso tem-se o surgimento da necessidade de comunicação entre os programas, ou entre os fluxos de execução (processos e *threads*).

No modelo tradicional de processos, surgiu a comunicação por troca de mensagem: os programas se comunicam por ferramentas oferecidas pelo sistema operacional, para que possam consumir os recursos do sistema operacional de maneira correta e síncrona. Um processo que não utilizo recursos externos a ele é muito mais simples do ponto de vista de sincronia: se ele apenas está utilizando recursos pertencentes a sua memória, não há conflito com outros processos.

2.2.1 Comunicação interprocessos

Comunicação interprocessos (IPC) descreve a capacidade de processos trocarem mensagens entre si enquanto rodam num mesmo sistema operacional. O sistema operacional oferece os mecanismos necessários para essa troca. No unix, por exemplo, é oferecido o enfileiramente de mensagem (POSIX, ou System V), encadeadores de entrada e saída de dados, chamados de *pipes*, e RPCs. Além disso, é necessário oferecer também recursos de sincronização, como semáforos e *locks* citeunixnetworkprogrammingvolume2. Os recursos de sincronização serão aprofundados na seção 2.2.3.

[ferramentas ipc sobre a rede]

2.2.2 Comunicação por memória compartilhada

Com a introdução de *threads*, como dito na seção 2.1, esses fluxos de execução passam então a compartilhar a memória. É possível então que, uma determinada alteração nesta memória do processo possa influenciar o resultado de outras *threads*. Assim, criaram-se ferramentas de sincronia por meio de memória compartilhada. Um mesmo processo pode ter diversos fluxos de execução concorrentes tendo um tempo de processamento disponibilizado pelo processador. Estes fluxos, chamados de threads, compartilham o mesmo espaço de memória entre si.

Um programa precisa, por exemplo, pegar resultados númericos e realizar duas operações distintas: pegar o maior dos seus valores e calcular a sua média. No fim, ambos os valores devem escritos na saída padrão com a ordem de primeiro o maior valor, e depois a média. Finalmente, o processo é então terminado. Para isso, foi decidido que dois fluxos de execução seriam criados, cada um para uma tarefa. Em *threads* preemptivas, não temos garantias da ordem de conclusão, e nem que quando um fluxo terminou já tem-se ambos

os resultados. Surge então a necessidade da comunicação da finalização desses outros dois fluxos de execução.

O problema apresentado pode ser facilmente resolvido por meio de variáveis globais que indicam a completude das operações: no momento que ambas as variávies estivessem com um determinado valor, seria determinado que suas execucões completaram, e agora é possível ser feito a impressão dos resultados. A memória compartilhada então, pode ser utilizada para fazer este tipo de comunicação entre as linhas de execução: um fluxo só deve prosseguir dado que o programa atingiu um determinado estado.

2.2.3 Condição de corrida

Se o problema indicado em 2.2.2 mudar, e agora desejar-se criar mais de um fluxo de execução para calcular a média. Os números são somados concorrentemente numa variável global, e no final é divido pela quantidade de números somados, por qualquer um dos fluxos, se o outro ainda não tiver realizado. O acesso simultâneo na memória compartilhada não é garantido: as instruções realmente executadas no processador muitas vezes diferem do código escrito numa linguagem de mais alto nível. Uma única operação de soma podem ser três instruções: ler um valor da memória, realizar e escrever novamente. O sistema operacional é responsável pela execução concorrente de instruções, e não se preocupa se as abstrações de alto nível da linguagem são respeitadas ou não.

[imagem de condição de corrida]

Assim, existe o problema de **condição de corrida**. O resultado do programa passa a ser dependente da ordem que as instruções são executadas no processador. Isso é inaceitável, pois os programas devem ter seu resultado previsível e correto. Para que seja possível eliminar a condição de corrida que surge na memória compartilhada, sistemas operacionais oferecem, geralmente, dois tipos de ferramentas: trancas (*locks*) e semáforos. [fontes]

Retornando ao exemplo: toda vez que for realizada uma soma a variável global, é desejado uma garantia de que aquela soma não será influenciada por outros fluxos, e assim o resultado não seja dependente da ordem das instruções a serem executadas. Para isso, é possivel construir uma **seção crítica** do código: antes de realizar a soma, o fluxo tenta adquirir um *lock*. Como o próprio nome indica, a ideia é proporcionar uma espécie de fechadura, que no momento que um fluxo fechar essa tranca (ou adquire o *lock*), nenhum outro fluxo conseguirá trancar, até que o outro abra a fechadura (ou libere o *lock*). Assim, temos garantia que, no momento que for realizada a soma, nenhum outro fluxo estará acessando a variável.

[insire imagem ou pseudocódigo desta lógica]

Um outro recurso oferecido é o semáforo binário: a ideia é que, ao invés de adquirir ou liberar *locks*, você incremente/decremente um valor numa estrutura de dado que seja segura concorrentemente. O fluxo, ao chegar na seção crítica, pede para decrementar o

valor do semáforo. Se este valor estiver 1, a operação é realizada. Senão, aguarda-se a mudança para o valor. Após a completude da seção crítica, o fluxo deve então incrementar o valor do semáforo, para indicar que o trecho de código está disponível para execução. Apesar de recursos diferentes, a premisa continua a mesma: criar seções críticas do código, e garantir que um único fluxo de execução esteja executando o trecho. [fontes]

2.2.4 Sincronização por condição lógica

No problema continuado na seção 2.2.3, um último empecilho ficou a ser resolvido: após o final da soma, apenas uma *thread* deve realizar a divisão. Assim, um fluxo adquire novamente um *lock*, ou incrementa um semáforo binário, checa o valor de variável de condição, que indica se já foi executado o último passo. Se não feito, realiza a divisão, altera o estado da variável de condição e libera o *lock*, ou decrementa o semáforo binário, terminando sua execução. Se já foi feito, apenas segue para o fim dessa linha de execução.

Com esta última lógica, o problema finalmente pode ser considerado resolvido: a soma dos valores não depende mais da ordem de execuções e a média é corretamente calculada no final da execução. O último desafio a ser resolvido, é um tipo de sincronização por **condição lógica**. O problema a ser resolvido requisita um tipo de sincronia entre as threads para um determinado tipo de comportamento ser permitido. A solução proposta faz uso de uma **variável de condição**, que indica um determinado estado da aplicação, porém a modificação neste estado poderia ser modificado por diversos fluxos, o que implica na necessidade de protege-la numa seção crítica.

Uma outra solução para o mesmo problema é utilizar um **semáforo contador**. A ideia é de, imediatamente antes de iniciar um *thread* incrementar um valor num semáforo (não preso aos valores 1 e 0). Após criar todas as *threads* somadoras, ele aguarda que o semáforo chegue ao valor 0. No fluxo de uma *thread* somadora qualquer, ela deve decrementar o valor deste mesmo semáforo ao finalizar as somas que lhe pertencem. Assim, quando todas as *threads* auxiliares da soma terminarem, a *thread* principal pode então realizar a divisão, sem a possibilidade de condições de corrida. A ideia é que forma-se uma barreira, que só pode ser ultrapassada quando todas determinadas *threads* terminarem suas execuções.

2.2.5 Deadlocks Starvations

3 A LINGUAGEM DE PROGRAMAÇÃO GO

Go (*Golang*, abreviação de Go programming language) é uma linguagem de programação de código aberto, lançada em 2009 pela Google. O alto tempo de compilação e a dificuldade de manutenção de código C++ para os sistemas de alto desempenho motivaram a criação da linguagem. É uma linguagem compilada com *garbage collector*, com uma sintaxe similar a C/C++, porém com mecanismos de linguagens modernos, como funções de primeira ordem, *closures*, e um modelo similar ao de orientação de objeto por meio de *interfaces* (BINET, 2018).

Go é uma linguagem concorrente, dado algumas de suas primitivas: *goroutines*, semelhante à *green threads* do Java, permitindo que funções sejam invocadas de maneira concorrente através da diretiva *go*; *channels*, utilizada para fazer a comunicação entre *goroutines*; e *select*, utilizado para o controle de fluxo em códigos concorrentes, em que os casos são a partir de leituras ou escritas em um canal. Além disso, é disponibilizado também a biblioteca básica *sync*, que possuí diversas implementações básicas de concorrência inclusive estruturas clássicas, como *mutexes*, e estruturas complexas, como um *HashMap* seguro para escrita entre *goroutines*.

3.1 ESPECIFICAÇÕES

Go é uma linguagem compilada, com tipagem estática e recursos para tipagem dinâmica. A linguagem se assemelha a C, porém injeta recursos novos de linguagens modernas, sempre a fim de servir a um propósito de simplificação.

3.1.1 Tipos de dados

A linguagem possui já pré declarada uma vasta gama de tipos de variáveis (PIKE et al.,):

- Tipos booleanos: `bool`, representa o valor `true` ou `false`, para operações de lógica booleana.

Código 3.1 – Exemplo do tipo booleano

```
package main

import "fmt"

func main() {
        var a int64
        var b complex128
        var c float64
        a = 100
        b = 1 + 10i
        c = 10.99999999999999
        fmt.Println(complex(float64(a), 0) + b)
        fmt.Println(float64(a) + c)
}
```

- Tipos numéricos: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float32`, `float64`, `complex64`, `complex128`. Cada tipo representa se é um `unsigned integer`, `integer`, `float` ou `complex`, com a quantidade de bits utilizados para construir a variável. Existem 5 `alias` para esses dados: `byte` para `uint8`, `rune` para `int32`, `uint` para `uint32` ou `uint64` e `int` para `int32` ou `uint64`, dependendo do compilador. Por fim, existe o `uintptr`, que é um *unsigned integer* grande o suficiente para guardar valores de um ponteiro.

Código 3.2 – Exemplo de tipos númericos

```
package main

import "fmt"

func main() {
        var a int64
        var b float64
        c := 10
        a = 20
        b = 10.9999999999
        fmt.Println(int(a) + c)
        fmt.Println(float64(a) + b)
}
```

- Tipos para texto: o tipo *string* guarda um conjunto de bytes, sendo que seu tamanho é sempre um valor não negativo. *Strings* em Go são imutáveis.

Código 3.3 – Exemplo de texto

```
package main

import "fmt"

func main() {
    a := "Hello"
    b := "        !\n"
    fmt.Printf("%s", a + b)
}
```

- *Arrays*: são sequências de elementos que compartilham o mesmo tipo. Podem ser multi dimensionais.

- *Slices*: são uma descrição para algum tipo de array existente, provendo acesso a uma parte do array, podendo ser ou não o todo. Um *slice* não inicializado tem sempre valor *nil*. Como *arrays*, eles podem ser multi dimensionais, porém as dimensões não são fixas.

Código 3.4 – Exemplo de array e slice

```
package main

import "fmt"

func main() {
        odds := [5]int{1, 3, 5, 7, 9}

        var firstThreeOdds []int = odds[0:3]
        fmt.Println(firstThreeOdds)
}
```

- *Structs*: Estruturas são sequências de elementos que possuem nome e tipo definidos *a priori*. Quando um campo não possue nome, apenas tipo, ele é considerado um campo embutido. Assim, a estrutura ganha também os campos que o seu campo embutido possui. Estruturas e interfaces devem ser declaradas indicando a criação de um novo tipo por meio da palavra reservada `type`.

- Ponteiros: semelhante a um ponteiro em C, guarda o local de memória de uma variável do tipo base do ponteiro. Um ponteiro não inicializado é do tipo *nil*

Código 3.5 – Exemplo de estrutura e ponteiro

```
package main

import "fmt"

type foo struct {
        Bar string
        Baz int
}

func main() {
        v := &foo{"Hello", 1}

        fmt.Printf("%+v\n", v)
}
```

- Funções: Uma variável do tipo função que possui o mesmo conjunto de parâmetros e resultados que o definido na variável. O último parâmetro pode receber uma quantidade não definida, sendo assim uma *variadric function*. Os nomês do paramêtros devem ser todos definidos ou todos indefinidos. São definidas por meio da palavra reservada `func`.

Código 3.6 – Exemplo de funções

```
package main

import "fmt"

func multiplication(a int, x int) int {
        return a * x
}

func main() {
        x := 2
        fmt.Println(multiplication(10, x))
}
```

- Métodos: Um método é uma função que está associada a uma determinada estrutura. O método pode ser tanto associado a uma estrutura quanto a um ponteiro para uma estrutura. O conjunto de métodos de uma estrutura é chamada de *method set*.

- Interfaces: Semelhante as interfaces de orientação a objeto, definem funções que devem ser implementadas como métodos para que, dado uma estrutura, esta estrutura seja considerada que implementa determinada interface. São definidas pela palavra reservada `func`.

Código 3.7 – Exemplo de interface e método

```
package main

import "fmt"

type Vector interface {
        Sum(Vector) Vector
}

type Vector2 struct {
        x, y float64
}

func (v Vector2) Sum(v2 Vector2) Vector2 {
        return Vector2{v.x + v2.x, v.y + v2.y}
}

func main() {
        v1 := Vector2{1, 1}
        fmt.Println(v1.Sum(Vector2{10, 10}))
}
```

- Mapas: Um tipo de dicionário chave-valor não ordenado, ou um *hashmap*, que associa um determinado tipo de variável (a chave) a outro (o valor), criando assim pares. A chave de um mapa pode ser outro mapa.

Código 3.8 – Exemplos de mapas

```
package main

import "fmt"

func main() {
        urls := map[string]string{
                "http":  "http://example.com",
                "https": "https://example.com",
        }
        fmt.Println(urls["http"])
}
```

- Canais: Estruturas de comunicação entre códigos concorrentes, que possuem mecanismos de `sending` e `receive`.

É possível, como indicado no caso númerico, criar um pseudônimo (*alias*) para um determinado tipo. No próprio caso númerico, temos os exemplos de `byte`, que é declarado pela própria linguagem a partir do trecho `type byte uint8`. Este comando, presente

no código fonte da linguagem, cria um *alias* do tipo `uint8` chamado `byte`. Utiliza-se o símbolo `:=` para declaração de variáveis implícitas, onde o tipo do código é inferido a partir do resultado da expressão a direita.

3.1.2 Packages

Semelhante a linguagens orientadas a objetos, a linguagem Go utilizao o recurso de pacotes (*packages*). Na primeira linha de cada código, deve-se explicitar qual é o que o código pertence. A visibilidade de uma função ou de uma estrutura é definida pela capitalização do primeiro caracter de uma variável. Ou seja, caso a primeira letra do nome da variável seja maiscula, significa que ela é visível fora do seu pacote.

Os pacote por padrão possuem nomes todos minusculos, e sua importação deve ser feita no inicio do programa, imediatamente após a definição de qual pacote aquele arquivo pertence.

O *main package*, ou pacote principal, deve conter uma função chamada `main`, que define o inicío do fluxo do programa.

3.1.3 Controles de fluxo

O controle de fluxo em Go é feito por meio condicionais ou de repetição. Para diminuir redundância, e simplificar a maneira de escrever um código, existe apenas uma única estrutura de repetição, o `for`. Para estruturas condicionais, existem o `if...else`, `switch` e o `select`, sendo o último utilizado para controle de fluxo em códigos concorrentes.

O `if` sempre incluí uma condicional, e pode conter uma declaração (*statement*) anterior a sua condicional, separada por meio de um ponto e vírgula, e deve também sempre se utilizar chaves para definir o escopo do fluxo. Caso deseja utilizar também o `else`, que executa quando a condição não é verdadeira, este deve estar exatamente após a chave que fecha o fluxo do `if`.

Código 3.9 – Diferentes usos das condicionais clássicas

```
package main

import "fmt"

func main() {
    a := 10
    if a < 10 {
        fmt.Println("'a' menor que 10")
    }
    if a < 5 {
        fmt.Println("'a' menor que 5")
    } else {
        fmt.Println("'a' maior que 5")
    }
    if b := a / 2; b < 10 {
        fmt.Println("'a/2' menor que 10")
    }
}
```

Temos que, para o valor inicial de `a = 10`, as impressões feitas serão "'a' maior que 5"e "'a/2' menor que 10". A nova variável criada na declaração dentro da cláusula condicional só existe dentro do fluxo do `if`, como o caso da variável `b` do exemplo..

O *switch* é um tipo de controle de fluxo para multiplas escolhas. Você deve passar uma expressão, e cada possível fluxo a ser seguido é definido por uma expressão precedido da palavra `case`. Caso você deseja um comportamento padrão caso nenhuma das expressões corresponde a expressão inicial. As expressões de caso podem ser devem poder executar a comparação `v == t`, onde `v` é a expressão inicial do *switch* e `t` é a expressão do caso.

O `for` segue a declaração de C, sem a necessidade de parenteses, em que é definido pela palavra reservada, por declarações a serem feitas anteriormente ao inicio da repetição, a condição de parada e o que deve ser executado no fim de cada passo, separados por um ponte e vírgula. Um `for` pode ser também declarado apenas utilizando a condição de parada, ou sem nenhum texto, que é o equivalente a um `while (1)` do C. Este tipo de estrutura de repetição é mostrado em 3.13, num exemplo sobre *goroutines*. O trecho de código apresentado na listagem 3.10 ilustra um exemplo básico do uso do `for`.

Código 3.10 – Exemplo do uso de for

```
package main

import "fmt"

func main() {
    a := [5]int{1, 2, 3, 4, 5}
    for i := 0; i < len(a); i++ {
        fmt.Printf("a[%d]: %d\n", i, a[i])
    }
}
```

O mesmo código acima pode ser feito de maneira mais simples, por meio de um *for :=* *range*, recurso para iterar sobre um determinado tipo de dado, sendo estes um entre *maps*, *arrays*, *slices* ou *arrays*. Sempre é retornado dois valores: uma chave e um valor, onde a chave é o índice no caso de *arrays*, *slices* ou *strings*, e para *maps* é a variável definida como chave do mapa. O código 3.11 mostra como ficaria o programa 3.10 utilizando o `range`.

Código 3.11 – Iterando em um array por meio da palavra reservada `range`

```
package main

import "fmt"

func main() {
    a := [5]int{1, 2, 3, 4, 5}
    for i, v := range a {
        fmt.Printf("a[%d]: %d\n", i, v)
    }
}
```

3.1.4 Interfaces e Métodos

Uma interface pode ser definida como um "protótipo", ou um tipo não definido, em que a intenção de uso é para abstrair qual o tipo de uma variável, apenas definindo quais são os métodos que ela implementa, qualquer que seja a estrutura real da variável.

Um bom uso de interfaces é para abstrair a comunicação entre os componentes do seu sistema (*drivers* de rede, ou para acesso a um banco de dados, com a própria lógica). Um exemplo é, dado um programa que envia logs de processos para um servidor remoto, podendo ser tanto via UDP quanto por TCP, dependendo de uma *flag* em sua inicialização, queremos que o programa execute a mesma lógica em qualquer um dos casos, sendo indiferente qual é protocolo da camada de transporte.

Código 3.12 – Exemplo de interfaces com redes

```
package main

type Connection interface {
        Read([]byte) (int, error)
        Write([]byte) error
}

type TCP struct{}

func SendStringToConn(value string, conn Connection) {
        err := conn.Write([]byte(value))
        if err != nil {
                panic(err)
        }
}

func (t *TCP) Read(dest []byte) (int, error) {
        // implement read from tcp socket
        return 0, nil
}

func (t *TCP) Write(input []byte) error {
        // implement write to tcp socket
        return nil
}

func main() {
        tcpSocket := &TCP{}
        a := "Hello world!"
        SendStringToConn(a, tcpSocket)
}
```

Na função `SendStringToConn` em 3.12, não se sabe qual o tipo de conexão será feita (TCP ou UDP). Basta que seja uma estrutura de dado qualquer que implemente as funções `Read` e `Write` com os mesmos parâmetros e retornos. A linguagem Go já fornece, de maneira completa, o que foi mostrado: no pacote `net`, existe a interface `Conn`, que é uma interface que tanto a estrutura `UDPConn` quanto a `TCPConn` implementam. Estes recursos serão explorados no capítulo 4, na implementação de um Chat em 4.3.2.

A implementação de um método para estrutura se dá semelhante a uma função, porém ela é precedida por um ponteiro ou uma variável do tipo da estrutura que esse método pertence. O nome desse campo é o receptor do método (*method receiver*). Os métodos implementados foram o `Read` e `Write` para a estrutura `TCP`, necessários para que essa estrutura seja considerada que implementa a interface `Connection`

3.2 RECURSOS PARA CONCORRÊNCIA

Go foi desenhada de maneira a facilitar a escrita de códigos concorrentes. Para isso, foram criados alguns recursos que cuidam da criação, comunicação e controle dentro de fluxos concorrentes. Será explicado a seguir os principais recursos: *goroutines*, *channels*, *select* e a biblioteca *sync*.

3.2.1 Goroutines

Goroutines são um tipo de *user space threads*, gerenciadas pelo escalonador do Go. Como dito anteriormente, para invocar uma *goroutine* basta fazer a invocação de uma função qualquer precedida pela diretiva Go. Com um tamanho reduzido de inicialmente 2KBs, a criação de uma *goroutine* é leve e não custosa, em comparação com uma *kernel thread*, de 2MB.

User space threads (*threads* do espaço de usuário) são um tipo de mecanismo de concorrência que é gerenciado por algum escalonador próprio, diferente das *kernel space threads* que são gerenciadas diretamente pelo *kernel* do sistema operacional.

Goroutines são o mecanismo de criação de fluxos concorrentes da linguagem.

Podendo ainda lidar diretamente com gerenciamento de processos, a principal maneira de executar fluxos concorrentes na linguagem é por meio das *goroutines*.

Por funções serem valores de primeira classe, muitas vezes a criação do fluxo concorrente pode vir de uma função anônima.

Código 3.13 – Exemplo de função anônima concorrente

```
package main

import (
    "fmt"
    "time"
)

func processData() {
    // Pega dados de um servidor remoto e processa-os
}

func main() {
    go func() {
        for {
            fmt.Println("working...")
            time.Sleep(30 * time.Second)
        }
    }
    processData()
}
```

Podemos ver que no exemplo 3.13, no inicio do processo iniciamos uma *goroutine* que executa a função anônima que entra num loop sem condição de parada imprimindo na linha de comando um texto, para dar alguma mensagem para o usuário (no caso, que está trabalhando).

Perceba que a *goroutine* criada não tem algum fluxo de parada. O que encerra sua execução é a finalização da *main thread*, o fluxo principal da função main. Esse comportamento é semelhante ao encontrado numa *kernel thread*, em que o fim da execução da thread principal causa consequentemente o fim do processo e de todas as outras threads.

As goroutines optam por serem multitarefas de maneira cooperativa, em que o escalonador só permite a execução de uma goroutine ser se tiver ao menos uma única *thread* livre. Isso significa que, uma *goroutine* não irá interrompar a outra caso não haja a possibilidade de criar novas *kernel threads*. Você pode controlar a quantidade de *kernel threads* a serem criadas a partir da variável de ambiente GOMAXPROCS. Por padrão, o valor dela é, caso não seja especificado, exatamente a quantidade de processadores disponíveis.

Dessa maneira, no exemplo 1, caso haja apenas um único núcleo disponível e decidimos executarmos o programa com a variável de ambiente `GOMAXPROCS=1`, que define o número máximo de *kernel threads*, a execução da *goroutine* só será possível quando ocorrer algum tipo de parada na função `processData`, como ler de um canal, escrever no canal, espera de IO do sistema operacional, entre outros. Caso fosse especificado um número maior que 1 de *kernel threads* com a variável `GOMAXPROCS`, a *goroutine* teria então espaço para ser alocado. Quem é responsável por gerenciar esse tipo de *threads* é o próprio sistema operacional, então o código iria executar concorrentemente a partir das preempções de processos/*threads*, como em outros linguagens que fornecem concorrência.

Por fim, *goroutines*, apesar do seu nome, não se comportam como corrotinas. Não é possível cessar sua execução e retomar do mesmo ponto de maneira idiomática pela linguagem, sendo necessário assim uma implementação de corrotinas utilizando os recursos do Go. No próximo capítulo, um exemplo implementação de corrotinas assimétricas em Go será demonstrado.

3.2.2 Channels

Enquanto *goroutines* são o mecanismo de criação de diferentes fluxos de execução, *channels* (ou canais) são o mecanismo de comunicação entre esses diversos fluxos. Existem dois tipos de *channels*, buferizados e não buferizados, sendo os buferizados não bloqueantes e os não buferizados bloqueantes.

Channels são criados utilizando o comando `make`, que aloca um determinado tipo de dado na memória e retorna a sua referência. Como a linguagem possui um *garbage collector*, não é necessário a liberação de memória explícita. Para acessa-lo, utilizá-se a palavra reservada `<-`, diferenciando-se assim de variáveis comuns.

Código 3.14 – Exemplo simples de um canal

```
package main

import "fmt"

func main() {
        ch := make(chan string)
        go func() {
                ch <- "Hello world!"
        }()
        a := <-ch
        fmt.Println(a)
}
```

3.2.3 Comunicação por compartilhamento de memória vs Compartilhar de memória para se comunicar

Canais foram criados com um tipo de filosofia diferente da usual de outras linguagens que permitem concorrência. *Channels*, em conjunto com a característica cooperativa das *goroutines*, permitem estruturar o código de maneira que a comunicação entre as *goroutines* seja feita diretamente pelos canais. Utilizando a característica bloqueante dos canais, é possível criar esperas em um fluxo de acordo com a mudança em um determinado *channel*.

Utilizaremos o recurso de sincronia da biblioteca `sync` chamado `WaitGroup`, um contador seguro entre *goroutines*, utilizado comumente para realizar a espera de conclusão dos fluxos concorrentes. Além disso, também temos o uso do `defer`, que "agenda"a execução de uma função para o fim da função onde foi declarado, não importando se ocorreu um erro ou não. É utilizado, não limitado a esses, para executar finalizações desejadas num fluxo, como fechamento de sockets, ou como no exemplo 3.15, incrementar o contador do `WaitGroup`.

Código 3.15 – Comunicando por canais

```
package main

import (
        "fmt"
        "sync"
        "net/http"
)

func main() {
        downloadSuccess := make(chan bool)
        wg := sync.WaitGroup{}
        wg.Add(1)
        go func() {
                defer wg.Done()
                if <-downloadSuccess == true {
                        fmt.Println("Download success")
                } else {
                        fmt.Println("Download failed")
                }
        }()
        wg.Add(1)
        go func() {
                defer wg.Done()
                _, err := http.Get("http://example.com")
                if err != nil {
                        downloadSuccess <- false
                        return
                }
                downloadSuccess <- true
        }()
        wg.Wait()
}
```

No exemplo 3.15, a *main thread* cria um canal booleano compartilhado por todas as *goroutines*. Lembrando que, as invocações das *goroutines* não iniciam sua execução, caso não haja *kernel threads* disponíveis, devido a sua propriedade cooperativa. Assim, por exemplo, numa execução com `GOMAXPROCS=1`, apenas na chamada do `wg.Wait` é liberado a execução de alguma das outras duas `goroutines`.

Caso a primeira venha a ser executada, ela tentará ler do canal `downloadSuccess`. Como ainda não foi preenchido, ela então libera a sua execução e fica numa espera. Assim, a outra *goroutine* pode então executar e preencher o canal com `true` caso nenhum erro no download seja encontrado.

Caso a segunda fosse executada antes da primeira, ao tentar escrever no canal e perce-

ber que não há nenhuma outra recebendo este dado, ela iria liberar a execução para que alguma outra *goroutine* pudesse declarar a intenção de ouvir esse canal.

O exemplo mostra a essência da comunicação de dados em Golang. Muitas vezes a intenção de compartilhamento de memória é para permitir que diversos fluxos de execução fiquem observando uma mudança. Esse tipo de observação, normalmente causa espera ocupada, onde é gasto ciclos de instruções numa *thread* para que ela veja se o valor de uma variável foi modificado ou não. Neste modelo, não há espera ocupada: a *goroutine* só está executando enquanto há algum comando a ser executado, e libera a execução para outras *goroutines* quando é feito algum tipo de bloqueio, como no caso a leitura de um canal não buferizado.

3.2.4 Select

Enquanto *goroutines* são utilizadas para a criação de fluxos concorrentes e *channels* são as estruturas de dados utilizadas para comunicação entre os fluxos, a diretiva *Select* é responsável pelo controle de fluxo em códigos concorrentes. Semelhante a um switch, a ideia é que possa ser usados com variáveis do tipo *channel*.

Uma das premissas de *Go* é a de legibilidade. O *select* permite então a escrita de fluxos comuns em códigos sequencias clássicos. A diretiva de seleção adiciona a possibilidade de decisões de múltiplos fluxos em códigos concorrentes, além de permitir o comportamento de tomar alguma ação padrão caso nenhum requisito seja cumprido. Assim, é possível definir um padrão para caso nenhum dos casos sejam executados, e a possibilidade de uma escrita mais compreensível em casos de múltiplas escolhas.

Um exemplo de uso do select é num cenário onde esperamos receber dados de múltiplas fontes, porém sem uma ordem definida.

Código 3.16 – Exemplo de uso do select

```
package main

func getLastTemperatureValue() float32 {
    // reads from sensor
}

func getLastMoistureValue() float32 {
    // reads from sensor
}

func processTemperature(temp float32) {
    // send to server
}

func processMoisture(moisture float32) {
    // send to server
}

func readTemperature(temperatureCh chan float32) {
        // ...
        for {
            temperatureCh <- getLastTemperatureValue() //
        }
}

func readMoisture(moistureCh chan float32) {
        // ...
        for {
            moistureCh  <- getLastMoistureValue() //
        }
}

func main() {
        tempChannel := make(chan float32)
        moistureChannel := make(chan float32)
        go readMoisture(moistureChannel)
        go readTemperature(tempChannel)
        for {
                select {
                case temp := <-tempChannel:
                        processTemperature(temp) //
                case moisture := <-moistureChannel:
                        processMoisture(moisture) //
                }
        }
}
```

No código 3.16, mostra-se uma possível cenário em que, um servidor central que recebe diversos sinais de sensores, caracterizados pelas funções `getLastTemperatureValue()` e `getLastMoistureValue()`, e os processa de acordo com as suas chegadas, definidas pelas funções `processTemperature` e `processMoisture`, sem uma ordem definida. Pela definição da declaração `select`, apenas um único caso será executado.(PIKE et al.,) Isso significa que ainda é possível ocorrer `starvations`, dado o comportamento pseudo-aleatório da escolha caso os dois casos estejam prontos.

Além disso, o `select` podem ser utilizados para permitir usar os canais de uma maneira não bloqueante. O recebimento ou envio de uma mensagem num canal é primeiramente validado se é possível de ocorrer. Caso não seja possível, o código segue o fluxo padrão caso exista, senão apenas termina a execução do trecho. No código 3.17, o select irá selecionar algum dos canais, caso o envio já tenha sido concluído. Se nenhuma outra *goroutine* estiver enviando no canal, ele apenas segue o padrão, e finaliza o programa.

Código 3.17 – Exemplo de select com leitura não bloqueante

```
package main

import "fmt"

func main() {
        ch, ch2 := make(chan int), make(chan int)
        go func() {
                ch <- 10
        }()
        go func() {
                ch <- 20
        }()
        select {
        case n := <-ch:
                fmt.Printf("Received value %d\n", n)
        case n := <-ch2:
                fmt.Printf("Received value %d\n", n)
        default:
                fmt.Println("Couldn't read from any channel")
        }
}
```

4 PADRÕES E APLICAÇÃO DE CONCORRÊNCIA EM GO

No último capítulo foi mostrado os recursos, com foco em concorrência, que a linguagem Go provê, analisando suas regras, funcionamento e implementações. O que será atacado são propostas de resoluções de problemas usuais de concorrências, e suas devidas análises. Assim, será possível então comparar com implementações de outras linguagens, sobre os aspectos de legibilidade, manutenabilidade e desempenho.

4.1 PROBLEMAS CLÁSSICOS DE CONCORRÊNCIA

Nesta seção, será abordado dois problemas clássicos de concorrência: o de produtor/-consumidor, e o de leitores e escritores. Serão feitas propostas de resolução com suas variações.

4.1.1 Produtor-consumidor

O problema de produtor consumidor, também conhecido como o problema do *buffer* limitado proposto por Dijkstra, como um problema de sincronização do uso de recursos compartilhados (DIJKSTRA, 1972). O problema a ser resolvido é como realizar a sincronização de, num fluxo (no caso, o *buffer* finito de informação), produtores inserindo porções de dados e consumidores retirando as porções de dados presentes, na mesma ordem que foram inseridos e sem repetição no consumo, de maneira a não se perder informação e não ocorrer *deadlocks*. Numa implementação ingênua, o *deadlock* poderia ocorrer quando tanto os consumidores como os produtores desejam acessar o *buffer*. Dado que o *buffer* é finito, não há de se preocupar com *starvation* se implementado corretamente.

4.1.1.1 Um produtor e um consumidor

No cenário de um único produtor e um outro consumidor, o problema de sincronização é em comum trivial de ser resolvido em diversas linguagens. Não diferentemente em Go, a implementação é trivial e direta.

Código 4.1 – Código do produtor

```
package main

import (
        "math/rand"
        "fmt"
        "sync"
        "time"
)

func producer(stream chan []byte, streamSize int) {
        randomInput := make([]byte, streamSize)
        for {
                n, err := rand.Read(randomInput)
                fmt.Println("Producer writing randomInput")
                stream <- randomInput
        }
}
```

A função produtor é simples e direta. Começa criando um array de bytes que possui um tamanho representado pela variável `streamSize`, que será usado para enviar o valor para o canal. Depois, preenche esse *array* com valores aleatórios de bytes (aqui como um valor qualquer entre 0 e 255). A fim de um melhor compreendimento da execução, ele imprime que ocorrerá a escrita no canal. Ocorre então, no canal passado como parâmetro da função, a escrita.

Código 4.2 – Código do consumidor

```
package main

import (
        "fmt"
        "sync"
        "time"
)

func consumer(stream chan []byte, streamSize int) {
        streamBytes := make([]byte, streamSize)
        for {
                streamBytes <- stream
                fmt.Printf("Read from stream %d bytes!\n%v\n",
                    streamSize, streamBytes)
        }
}
```

A função consumidor foi também muito simplificada. Como na função produtor, o

array de bytes é inicializado com um tamanho definido, desta vez porém será usado para a leitura do canal passado como parâmetro. Após receber o valor, ele imprime o valor recebido.

Código 4.3 – Código da função principal

```
package main

import (
        "math/rand"
        "time"
        "sync"
)


func main() {
        wg := sync.WaitGroup{}
        rand.Seed(time.Now().UTC().UnixNano())
        streamSize := 32
        stream := make(chan []byte, 1)

        wg.Add(1)
        go consumer(stream, streamSize)
        go producer(stream, streamSize)
        wg.Wait()
}
```

Resta então, na função principal, a inicialização dos parametros que serão utilizados nas função. Ele cria uma nova semente para o gerador aleatório do Go, utilizando o tempo atual no padrão *epoch* do Unix. Além disso, ele cria um `WaitGroup`, para impedir o fim da função principal, que engatilharia o fim do processo, e assim o consumidor e não iria produzir. Alem disso, ele cria o canal que será compartilhado pelas *goroutines*, como um buffer de um único slot.

O código é o suficiente para suprir o seguinte cenário: dado um produtor e um consumidor, em que o produtor gera uma sequência aleatória de 32 bytes, coloca num buffer que só aceita uma única sequência de bytes e o consumidor obterá essa sequência. Não importa a ordem que o código seja executado, o resultado é sempre o mesmo, dado que o consumidor espera o produtor caso este ainda não tenha produzido nada, ao mesmo tempo que o produtor aguarda o consumidor enquanto o *buffer* estiver lotado.

4.1.1.2 N produtores e M consumidores

Neste cenário, existe a necessidade de coordenar a inserção no *buffer* entre os produtores e a retirada do *buffer* entre os consumidores. Então, temos as mesmas condições

inicias propostas pelo cenário anterior, apenas modificando o número de produtores e consumidores.

Mais uma vez, é trivial a resolução desse problema por meio dos mecanismos de concorrência que a linguagem Go provê. Dado que não é necessário fazer a coordenação explícita no acesso do canal, pois a estrutura é feita para esta comunicação, apenas será instanciado um número arbitrário N de produtores e um número arbitrário M de consumidores, recebido por meio de argumentos passados no momento de execução do programa.

Código 4.4 – Código do produtor N-M

```
package main

import (
        "math/rand"
        "fmt"
        "sync"
        "time"
)

func producer(stream chan []byte, streamSize int) {
        randomInput := make([]byte, streamSize)
        for {
                n, err := rand.Read(randomInput)
                fmt.Println("Producer writing randomInput")
                stream <- randomInput
        }
}
```

O código do produtor não foi alterado em relação ao 4.1.

Código 4.5 – Código do consumidor N-M

```
package main

import (
        "fmt"
        "sync"
        "time"
)

func consumer(stream chan []byte, streamSize int) {
        streamBytes := make([]byte, streamSize)
        for {
                streamBytes = <-stream
                fmt.Printf("Read from stream %d bytes!\n%v\n",
                   streamSize, streamBytes)
        }
}
```

O código do consumidor não foi alterado em relação ao 4.2.

Código 4.6 – Código da função principal N-M

```
package main

import (
        "fmt"
        "math/rand"
        "os"
        "strconv"
        "sync"
        "time"
)

func main() {
        if len(os.Args) < 4 {
                fmt.Printf("Usage: %s <number of consumers> <number of
                    producers> <buffer size>\n", os.Args[0])
                os.Exit(1)
        }
        consumers, err := strconv.Atoi(os.Args[1])
        if err != nil {
                fmt.Println("Failed to convert consumer argument as
                    number")
                panic(err)
        }
        producers, err := strconv.Atoi(os.Args[2])
        if err != nil {
                fmt.Println("Failed to convert producer argument as
                    number")
                panic(err)
        }
        bufferSize, err := strconv.Atoi(os.Args[3])
        if err != nil {
                fmt.Println("Failed to convert buffer size argument as
                    number")
                panic(err)
        }
        wg := sync.WaitGroup{}
        rand.Seed(time.Now().UTC().UnixNano())
        streamSize := 32
        stream := make(chan []byte, bufferSize)

        wg.Add(1)
        for i := 0; i < consumers; i++ {
                go consumer(stream, streamSize)
        }
        for i := 0; i < producers; i++ {
                go producer(stream, streamSize)
        }
        wg.Wait()
}
```

As modificações, em comparações ao 4.3, podem ser resumidas em três trechos:

- Obtenção das variáveis de ambiente, e conversão para números (quantidade N de produtores, e quantidade M de consumidores e para fins ilustrativos o tamanho x do buffer), por meio do array `os.Args`, que possui as entradas passadas pela execução do programa, e pela função `strconv.Atoi`, semelhante ao `atoi` da biblioteca `stdlib`.
- Alteração do tamanho do *buffer*, para ilustrar que pode ser um tamanho qualquer de escritas antes dos produtores bloquearem
- *Loops* para criação de vários consumidores e vários produtores

4.1.2 Leitores e escritores

O problema dos leitores e escritores pode ser definido da seguinte maneira: um recurso é compartilhado por diversos processos que precisam de ou o escrever ou o ler, de maneira exclusiva (BALLHAUSEN, 2003). Normalmente é resolvido por algum tipo de estrutura que permita exclusões mútuas, como *mutexes*.

O problema possui uma característica importante diferente do problema de consumidor e produtor: vários leitores podem realizar uma leitura simultaneamente, enquanto apenas um escritor pode realizar a escrita por vez. Considere sempre que os tempos de leitura e escritas são tempos finitos.

Existem três cenários comuns:

- Prioridade para a leitura
- Prioridade para a escrita
- Sem prioridade (ou sem *starvation*)

4.1.2.1 Leitores e escritores com prioridade para leitura

No caso de prioridade para a leitura, o problema segue as seguintes condições, considerando inicialmente o recurso como "livre"(sem nenhum processo utilizando-o) com um valor qualquer preenchido. A requisição de acesso ao recurso contempla o tempo de obtenção de um *mutex* até a liberação do mesmo.

- Caso um processo **leitor** requisite o acesso ao recurso, ele irá:
  - Ler o recurso, caso não tenha nenhum escritor utilizando.
  - Aguarda a liberação do recurso, caso um escritor esteja utilizando.

- Caso um processo **escritor** requisite o acesso ao recurso, ele irá:
  - Escrever no recurso, caso não tenha nenhum escritor nem nenhum leitor utilizando.
  - Agurdar a liberação do recurso, caso tenha algum outro processo utilizando. No caso dos leitores, ele aguarda o fim de todos os leitores..

Nestas condições, temos o cenário de que existe sempre uma garantia de que os leitores terão uma oportunidade de ler. Dado a característica de os processos leitores sempre lerem caso tenha um leitor, é possível entrar num estado onde não é possível prever quando o recurso será liberado pelos leitores, e assim podendo causar *starvation* para os escritores. Dado que ao menos um leitor peça acesso ao recurso enquanto um escritor está utilizando, temos que este será eventualmente atendido, dado a característica dos mutexes.

Código 4.7 – Código do escritor

```
package main

import (
        "crypto/rand"
        "fmt"
        "sync"
)

type Writer struct {
        ID      int
        Writers *sync.Mutex
}

func (w *Writer) start(stream []byte, size int) {
        for {
                w.Writers.Lock()
                fmt.Printf("Writer %d wrinting byte string %v\n", w.ID,
                   stream)
                rand.Read(stream)
                w.Writers.Unlock()
        }
}
```

O escritor começará um loop infinito, em que ele pede acesso ao recurso por meio do mutex chamado de `Writers` e a função `Lock`. Ao ter o acesso, ele insere um número aleatório de bytes na stream compartilhada. Por fim, ele libera o acesso ao recurso chamando a função `Unlock`.

Código 4.8 – Código do leitor

```
package main

import (
        "fmt"
        "sync"
)

type Reader struct {
        ID      int
        Count   chan int
        Writers *sync.Mutex
}

func (r *Reader) start(stream []byte, size int) {
        for {
                count := <-r.Count
                if count == 0 {
                        r.Writers.Lock()
                }
                r.Count <- count + 1

                fmt.Printf("Reader %d read byte string %v\n", r.ID,
                    stream)

                count = <-r.Count
                if count == 1 {
                        r.Writers.Unlock()
                }
                r.Count <- count - 1
        }
}
```

O código dos leitores é um pouco diferente. Como queremos que todos os leitores leiam o dado, precisamos fazer com que apenas o primeiro leitor faça a requisição do recurso. Assim, utiliza-se o canal `Count` para transmitir a quantidade de leitores em um determinado momento. Esse canal é iniciado com tamanho um no *buffer* e inserido o valor 0. Assim, todo leitor que inicia deve consumir o valor e acrescentar em uma unidade. Após a leitura, ele deve novamente pegar a quantidade de escritores e validar se ele é o ultimo sendo executado. Se sim, ele deve liberar o recurso para os escritores. Por fim, ele decrementa o valor e o guarda no *buffer*.

Código 4.9 – Código da função principal

```
package main

import (
        "fmt"
        "os"
        "strconv"
        "sync"
        "time"
)

func main() {
        if len(os.Args) < 3 {
                fmt.Printf("Usage: %s <number of readers> <number of
                    writers>\n", os.Args[0])
                os.Exit(1)
        }
        numReaders, err := strconv.Atoi(os.Args[1])
        if err != nil {
                fmt.Println("Failed to convert consumer argument as
                    number")
                panic(err)
        }
        numWriters, err := strconv.Atoi(os.Args[2])
        if err != nil {
                fmt.Println("Failed to convert producer argument as
                    number")
                panic(err)
        }

        streamSize := 32
        stream := make([]byte, streamSize)
        readers := make(chan int, 1)
        readers <- 0
        writers := &sync.Mutex{}

        for i := 0; i < numWriters; i++ {
                writer := &Writer{ID: i + 1, Writers: writers}
                go writer.start(stream, streamSize)
        }
        for i := 0; i < numReaders; i++ {
                reader := &Reader{ID: i + 1, Readers: readers, Writers:
                    writers}
                go reader.start(stream, streamSize)
        }
        time.Sleep(100 * time.Second)
}
```

A função principal apenas faz as inicializações do problema, pegando como argumento de entrada a quantidade de leitores e escritores. Ela fica responsável também por criar o *mutex* e o canal que serão compartilhados entre as *goroutines*. Para fim de testes, ela apenas dorme por 100 segundos, antes de terminar o programa.

4.1.2.2 Leitores e escritores com prioridade para escrita

O problema é resolvido de maneira semelhante ao anterior. O que modifica são as condições de mudança do processo escritor e do processo leitor, para priorizar a escrita. Tendo as mesmas considerações do que as anteriores, temos:

- Caso um processo **leitor** requisite o acesso ao recurso, ele irá:
  - Ler o recurso, caso não tenha nenhum outro processo utilizando.
  - Ler o recurso, caso tenha um leitor utilizando e não tenha um escritor aguardando.
  - Aguardar fim de todos os escritores, caso tenha um escritor aguardando ou utilizando o recurso.
- Caso um processo **escritor** requisite o acesso ao recurso, ele irá:
  - Escrever no recurso, caso não tenha nenhum outro processo utilizando
  - Aguardar a liberação do recurso, um escritor utilizando.
  - Aguardar o fim dos leitores sem permitir que novos sejam iniciados, caso tenha pelo menos um leitor utilizando.

Código 4.10 – Código do escritor

```
package main

import (
        "crypto/rand"
        "fmt"
        "sync"
)

type Writer struct {
        ID      int
        Writers *sync.Mutex
        Readers *sync.Mutex
        Count chan int
}

func (w *Writer) Start(stream []byte, size int) {
    for {
        count := <-w.Count
        if count == 0 {
                w.Readers.Lock()
        }
        w.Count <- count + 1

        w.Writers.Lock()
        fmt.Printf("Writer %d writing byte string %v\n", w.ID, stream)
        rand.Read(stream)
        w.Writers.Unlock()

        count = <-w.Count
        if count == 1 {
                w.Readers.Unlock()
        }
        w.Count <- count - 1
    }
}
```

O escritor com prioridade fica semelhante ao código 4.7, porém com uma adição importante: agora, o primeiro escritor sempre deve pedir um novo mutex, chamado de `Readers`. Utiliza-se um próprio contador semelhante ao feito em 4.8, porém agora para escritores. Ao conseguir adquiri-lo, ele pede acesso ao recurso com os outros escritores, usando o mesmo mutex que antes. Assim, temos que diversos escritores irão competir entre si apenas por meio do mutex `Writers`, de acesso ao recurso, enquanto os leitores aguardarão todos os escritores finalizarem antes de conseguir de fato executar.

Código 4.11 – Código do leitor

```
package main

import (
        "fmt"
        "sync"
)

type Reader struct {
        ID      int
        Readers *sync.Mutex
        Writers *sync.Mutex
        Count   chan int
}

func (r *Reader) Start(stream []byte, size int) {
    for {
        r.Readers.Lock()
        count := <-r.Count
        if count == 0 {
                r.Writers.Lock()
        }
        r.Count <- count + 1
        r.Readers.Unlock()

        fmt.Printf("Reader %d read byte string %v\n", r.ID, stream)
        count = <-r.Count
        if count == 1 {
                r.Writers.Unlock()
        }
        r.Count <- count - 1
    }
}
```

O leitor sem prioridade fica ainda mais semelhante do 4.8, com uma importante diferença: agora, sua primeira ação é pedir acesso ao lock do mutex `Readers`. Assim, caso os escritores estejam escrevendo, todo leitor irá aguardar. Quando um leitor for liberado, ele pedirá o mutex do recurso, chamado de `Writers`, impedindo writers de acessarem. Ele utiliza a mesma lógica anterior para permitir diversos readers lendo simultaneamente. Por fim, antes de ler o recurso, ele libera o mutex `Readers`. Se um escritor conseguir adquiri-lo, nenhum outro leitor conseguirá ler o recurso. Caso contrário, mais leitores poderão realizar a leitura.

Código 4.12 – Código da função principal

```
package main

import (
        "fmt"
        "os"
        "strconv"
        "sync"
        "time"
)

func main() {
        if len(os.Args) < 3 {
                fmt.Printf("Usage: %s <number of readers> <number of
                    writers>\n", os.Args[0])
                os.Exit(1)
        }
        numReaders, err := strconv.Atoi(os.Args[1])
        if err != nil {
                fmt.Println("Failed to convert consumer argument as
                    number")
                panic(err)
        }
        numWriters, err := strconv.Atoi(os.Args[2])
        if err != nil {
                fmt.Println("Failed to convert producer argument as
                    number")
                panic(err)
        }

        streamSize := 32
        stream := make([]byte, streamSize)
        readers := &sync.Mutex{}
        writers := &sync.Mutex{}
        readerCount := make(chan int, 1); readerCount <- 0
        writerCount := make(chan int, 1); writerCount <- 0

        for i := 0; i < numWriters; i++ {
                writer := &Writer{ID: i + 1, Readers: readers, Writers:
                    writers, Count: writerCount}
                go writer.Start(stream, streamSize)
        }
        for i := 0; i < numReaders; i++ {
                reader := &Reader{ID: i + 1, Readers: readers, Writers:
                    writers, Count: readerCount}
                go reader.Start(stream, streamSize)
        }
        time.Sleep(100 * time.Second)
}
```

O código da função principal mantem-se muito semelhante, apenas criando-se os canais e os *mutexes* adicionais utilizados para a sincronia do problema.

4.1.2.3 Leitores e escritores sem starvation

Para a solução sem *starvation*, cada uma das filas de escritores e leitores não deixa aumentar caso chegue um processo do outro tipo. Isso significa que, caso um processo do tipo A esteja ocorrendo, e chegar um processo do tipo B, este processo será servido antes de outros processos do tipo A chegarem. Desta maneira, sempre há uma alteração de qual é o tipo de processo utilizando o recurso, e garantimos que não há *starvation*.

Seguindo os fluxos anteriores de lógica, temos:

- Caso um processo **leitor** requisite o acesso ao recurso, ele irá:
  - Ler o recurso, caso não tenha nenhum outro processo utilizando.
  - Ler o recurso, caso tenha um leitor utilizando e não tenha um escritor aguardando.
  - Aguardar fim de todos os escritores, sem permitir o inicio de novas escritas, caso tenha um escritor aguardando ou utilizando o recurso.
- Caso um processo **escritor** requisite o acesso ao recurso, ele irá:
  - Escrever no recurso, caso não tenha nenhum outro processo utilizando
  - Aguardar a liberação do recurso, um escritor utilizando.
  - Aguardar o fim dos leitores, sem permitir o inicio de novas leituras, caso tenha pelo menos um leitor utilizando.

Para este tipo de cenário, a linguagem Go já providencia um tipo de *mutex* específico para este problema, chamado de `RWMutex`, ou um *mutex* de escrita e leitura. Assim, é transparente escrever o código para este tipo de leitor e escritor, pois a linguagem já cuida da regra de acesso na hora de realizar o *lock* e o *unlock*.

Código 4.13 – Código do escritor

```
package writer

import (
        "crypto/rand"
        "fmt"
        "sync"
)

type Writer struct {
        ID    int
        Mutex *sync.RWMutex
}

func (w *Writer) start(stream []byte, size int) {
        for {
                w.Mutex.Lock()
                fmt.Printf("Writer %d writing byte string %v\n", w.ID,
                    stream)
                rand.Read(stream)
                w.Mutex.Unlock()
        }
}
```

Código do escritor fica extremamente simplificado: apenas faz um *lock* no *mutex* antes de escrever, e libera assim que termina. Na realidade, a lógica é a mesma que no 4.7, porém com a diferença do tipo do *mutex*.

Código 4.14 – Código do leitor

```
package reader

import (
        "fmt"
        "sync"
)

type Reader struct {
        ID    int
        Mutex *sync.RWMutex
}

func (r *Reader) start(stream []byte, size int) {
        for {
                r.Mutex.RLock()
                fmt.Printf("Reader %d read byte string %v\n", r.ID,
                   stream)
                r.Mutex.RUnlock()
        }
}
```

Código do leitor é também muito enxugado. O leitor adquire e libera o mutex por meio de duas funções diferentes: o `RLock` e o `RUnlock`. Estas funções permitem um uso síncrono da seção crítica seguindo as premissas do problema: diversos leitores podem acessar simultaneamente, porém apenas um escritor pode escrever no dado por vez, e nenhum leitor pode acessar o recurso enquanto estiver ocorrendo a escrita. A lógica fica semelhante ao 4.7 e 4.13, com a diferença exclusivamente de qual função está sendo utilizada.

Código 4.15 – Código da função principal

```
package main

import (
        "fmt"
        "os"
        "strconv"
        "sync"
        "time"
)

func main() {
        if len(os.Args) < 3 {
                fmt.Printf("Usage: %s <number of readers> <number of
                    writers>\n", os.Args[0])
                os.Exit(1)
        }
        numReaders, err := strconv.Atoi(os.Args[1])
        if err != nil {
                fmt.Println("Failed to convert consumer argument as
                    number")
                panic(err)
        }
        numWriters, err := strconv.Atoi(os.Args[2])
        if err != nil {
                fmt.Println("Failed to convert producer argument as
                    number")
                panic(err)
        }
        stream := make([]byte, 32)
        streamSize := 32
        mutex := &sync.RWMutex{}
        for i := 0; i < numWriters; i++ {
                writer := Writer{ID: i + 1, Mutex: mutex}
                go writer.start(stream, streamSize)
        }
        for i := 0; i < numReaders; i++ {
                reader := &Reader{ID: i + 1, Mutex: mutex}
                go reader.start(stream, streamSize)
        }
        time.Sleep(100 * time.Second)
}
```

Como há também menos recursos utilizados para sincronia, a função principal realiza menos inicializações que em 4.9 e 4.12.

4.1.3 Barbeiro dorminhoco

O problema do barbeiro dorminhoco é um outro tipo de problema clássico de concorrência. A ideia é que exista dois tipos de processos: um barbeiro e diversos clientes. O barbeiro deve dormir enquanto não houver nenhum cliente para ser atendido. Na chegada do cliente, o barbeiro é acordado e inicia assim o corte de cabelo. Se novos clientes chegarem enquanto o barbeiro estiver realizando um corte, eles devem ser enfileirados até um número máximo de "cadeiras"que o salão providencia.

Código 4.16 – Código do barbeiro

```
package main

import (
        "fmt"
        "sync"
)

type Barber struct {
        ID          int
        ReadyToCut chan int
}

func (b *Barber) start(seatsMutex *sync.Mutex, customerReady chan int,
    seats *int) {
        fmt.Println("Barber", b.ID, "started...")
        for {
                fmt.Println("Waiting costumer to arrive")
                costumerID := <-customerReady
                fmt.Println("Costumer", costumerID, "has arrived")
                seatsMutex.Lock()
                *seats++
                b.ReadyToCut <- b.ID
                fmt.Println("Will now start to cut")
                seatsMutex.Unlock()
                // CUT HAIR HERE
        }

}
```

No código do barbeiro, há a declaração da estrutura definida como `Barber`, que é responsável pela lógica do barbeiro. Ele possui um ID, e um canal que é utilizado para avisar aos clientes que ele está pronto para cortar. Seu fluxo então é:

- Aguarda a chegada de um cliente;
- Ao chegar o cliente, avisa-o de que está pronto para cortar;

- Corta o cabelo, e reinicia o fluxo;

Uma sincronização que foi necessária de ser feita é a alteração na quantidade de assentos. Dado que é uma variável compartilhada, precisamos sempre utilizar algum recurso de exclusão mútua, nesse caso um *mutex*, para garantir que não teremos problema de escritas/leituras concorrentes.

Código 4.17 – Código do cliente

```
package main

import (
        "fmt"
        "sync"
)

type Costumer struct {
        ID             int
        CostumerReady chan int
}

func (c Costumer) start(seatsMutex *sync.Mutex, barberReady chan int,
    seats *int) {
        fmt.Println("Costumer", c.ID, "started...")
        seatsMutex.Lock()
        if *seats > 0 {
                *seats--
                seatsMutex.Unlock()
                fmt.Println("waiting barber to receive my arrival")
                c.CostumerReady <- c.ID
                fmt.Println("barber knows i have arrived")
                barberID := <-barberReady
                fmt.Println("i will now have my hair cut by barber",
                    barberID)
                // HAVE HAIR CUT HERE
        } else {
                seatsMutex.Unlock()
                fmt.Println("No place to wait, going away...")
        }
}
```

No código do cliente, o fluxo esperado é o apresentado:

- Ao chegar, confere a quantidade de cadeiras. Se tiver algum lugar, senta ocupando. Senão, vai embora;
- Aguarda sua vez de ser atendido;

- Quando chegar sua vez, avisa ao barbeiro e inicia o corte.

Para que não tenha problemas de concorrência na leitura de assentos, é necessário o uso de um recurso de exclusão mútua, no caso o mesmo *mutex* utilizado pelo barbeiro e pelos outros clientes. A comunicação com o barbeiro é feito via canais, permitindo uma maneira simples e direta de comunicação. Não há uma necessidade de criação de uma estrutura de fila, dado que o próprio canal já enfileira corretamente os clientes.

Código 4.18 – Código da função principal

```
package main

import (
        "fmt"
        "sync"
        "time"
        "math/rand"
)

func main() {
        costumerChannel := make(chan int)
        costumers := []Costumer{}
        for i := 0; i < 5; i++ {
                costumers = append(costumers, Costumer{
                        ID:            i + 1,
                        CostumerReady: costumerChannel,
                })
        }
        barber := &Barber{ID: 1, ReadyToCut: make(chan int)}
        mutex := &sync.Mutex{}
        seats := new(int)
        *seats = 5
        go barber.start(mutex, costumerChannel, seats)
        for i := 0; i < 1000; i++ {
            turn := rand.
                fmt.Println(costumer.ID)
                go costumer.start(mutex, barber.ReadyToCut, seats)
        }
        time.Sleep(10 * time.Second)
}
```

Por fim, temos a função principal do código. Esta presente as criações de clientes, do barbeiro, do *mutex* utilizado e da quantidade de assentos. No final, temos um sleep simples apenas para evitar impedir a saída da rotina principal.

4.1.4 Filósofos comilões

O problema dos filósofos comilões foi criado por Dijkstra, e posteriormente elaborado por Tony Hoare. Um número qualquer de filósofos, inicialmente proposto como cinco, passam toda a sua vida pensando e comendo. Para comer, existe uma mesa redonda com o número de lugares o suficiente para quantidade de filósofos, com cada um destes com um lugar específico, e cada lugar possui um garfo posicionado a esquerda do assento. No centro da mesa, existe uma cumbuca de comida que o filósofo deve utilizar dois garfos, um localizado a esquerda do seu assento e outro posicionado na direita. Os garfos não podem ser usados simultaneamente. Por fim, deve-se manter uma contagem de quantos filósofos estão na sala comendo (DÍAZ; RAMOS, 1981, pg.323).

Nesta abstração, os filósofos seriam os fluxos de execução do seu programa. Os garfos, são os recursos que os fluxos disputam. Um dos cenários indesejados possíveis é em que cada um dos filosófos seguram um garfo, causando um *deadlock*.

4.2 PERFORMANCE EM GO

Já foi apresentado como concorrência em Go pode ser utilizado para diminuir a complexidade de problemas clássicos de concorrência. Nesta seção, iremos atacar a questão de se essas abstrações adicionais são performáticas ou não. Será utilizado o problema clássico de paralelismo no cálculo de multiplicação de matrizes.

O principal aspecto analisado será o tempo de execução dos programas, sendo posteriormente comparados com a linguagem C++.

4.2.1 Multiplicação de matrizes

O problema de multiplicação de matrizes é um ótimo exemplo para testes de paralelismo pois é um cenário em que não há necessidade de sincronia entre os diversos fluxos de execução. Isso se dá pela independência de cálculo de cada campo da matriz.

Considere uma matriz qualquer A, bidimensional, de tamanho N por$M$. Considere também uma matriz B qualquer de tamanho M por N', e considere a matriz resultado $C = A * B$, com dimensões N por N'.

Para cada par i,j, onde $1 \leq i \leq N$ e $1 \leq j \leq N'$, sendo $C_{i,j}$ cada elemento da matriz final, temos que o cálculo de um elemento é igual a:

$$C_{i,j} = \sum_{k=1}^{M} (A_{i,k} * B_{k,j}) \tag{4.1}$$

Em que $A_{i,k}$ é um elemento da matriz A e $B_{k,j}$ é um elemento da matriz B.

Assim, é possível observar que cada elemento da matriz final C depende exclusivamente dos valores de A e de B. Dado que as matrizes A e B não são alteradas a partir do inicio da multiplicação, o cálculo de cada campo é independente da execução do outro campo.

4.2.2 Código em Go

Código 4.19 – Código da estrutura `Matrix`

```
package main

import (
        "fmt"
        "math/rand"
        "sync"
)

type MatrixInt [][]int
func (a MatrixInt) Step(b, c MatrixInt, row, col int, wg *sync.WaitGroup
    ) {
        defer wg.Done()
        for i := 0; i < len(a); i++ {
                c[row][col] += a[row][i] * b[i][col]
        }
}
func (a MatrixInt) Multiply(b, c MatrixInt, parallel bool) {
        wg := &sync.WaitGroup{}
        for i := 0; i < len(a); i++ {
                for j := 0; j < len(a); j++ {
                    wg.Add(1)
                        if parallel {
                                go a.Step(b, c, i, j, wg)
                        } else {
                            c[i][j] += a[i][k] * b[k][j]
                        }
                }
        }
        if parallel {
            wg.Wait()
        }
}
func (a *MatrixInt) InitSquared(newSize, max int, fillRandomly bool) {
        *a = make([][]int, newSize)
        for i := 0; i < newSize; i++ {
                (*a)[i] = make([]int, newSize)
                if fillRandomly {
                        for j := 0; j < newSize; j++ {
                                (*a)[i][j] = rand.Int() % max
                        }
                }
        }
}
```

No código de multiplicação, foram criadas três métodos para a função MatrixInt, que é apenas um *alias* para um *slice* de *slice* de `int`.

- `InitSquared`: É responsável pela inicialização das matrizes. Pode iniciar a matriz com campos com valor 0 por padrão, ou preencher aleatoriamente, num valor máximo passado como parâmetro. É relevante ver que o *receiver* do método é um ponteiro, para uqe possamos atribuir o valor da nova matriz alocada a variável final utilizada, e não uma cópia.

- `Step`: Dado uma matriz qualquer A e duas matrizes B e C, e os campos i, j da matriz C, a função calcula o valor final de $C_{i,j}$.

- `Multiply`: responsável por executar a multiplicação. Cada campo de C é passado para o método `Step`, porém no modo paralelo é invocado uma *goroutine*, podendo ser então paralelizado. Por fim, se foi executado paralelamente, é realizado um wait no final da função para assegurar a sua completude.

Dada a abstração de *slices*, não foi necessário nenhum uso de ponteiros nas funções responsáveis pela multiplicação, apenas na `InitSquared`, dado que faz uma nova alocação de espaço para a variável. Assim, não é preciso uma gerência de memória explícita na hora de consumir e alterar esses *slices* já inicializados.

Código 4.20 – Código da função principal

```
package main

import (
        "fmt"
        "math/rand"
        "os"
        "strconv"
        "time"
)

func main() {
        A, B,  C := MatrixInt{}, MatrixInt{}, MatrixInt{}
        if len(os.Args) < 3 {
                fmt.Printf("Usage: %s <size of squared matrix> <parallel
                    : true or false>\n", os.Args[0])
                os.Exit(1)
        }
        n, err := strconv.Atoi(os.Args[1])
        if err != nil {
                fmt.Println("Failed to convert matrix size as number")
                panic(err)
        }
        parallel, err := strconv.ParseBool(os.Args[2])
        if err != nil {
                fmt.Println("Failed to convert option parallel")
                panic(err)
        }

        rand.Seed(time.Now().UTC().UnixNano())

        A.InitSquared(n, 1000, true)
        B.InitSquared(n, 1000, true)
        C.InitSquared(n, 0, false)

        start := time.Now()
        A.Multiply(B, C, parallel)
        end := time.Now().Sub(start)

        fmt.Println(end.String())
}
```

A função principal fica com o papel de preparar a execução da multiplicação. Ela inicializa as três matrizes A, B e C, valida os valores recebidos na execução do programa (tamanho da matriz) e se deve ser paralelizado. Além disso, ela inicializa uma nova

semente para geração dos números pseudoaleatórios, faz as chamadas de inicalização das matrizes, e realiza a contagem de tempo. Por fim, ele imprime o tempo passado já formatado para entendimento humano, utilizando o sistema internacional de medidas para indicar a unidade do tempo.

Para modificar a quantidade de threads utilizada pelo *runtime* do Go, foi utilizado a variável de ambiente `GOMAXPROCS`. Então, a chamada:

```
GOMAXPROCS=2 ./go_matrix 1000 true
```

Executa a multiplicação de duas matrizes de tamanho 1000 por 1000 de maneira paralela.

É interessante observar que, neste caso, a quantidade de *threads* criadas e executando em paralelo não fica sob responsabilidade de quem cria o código. O código é simplificado e não exige um conhecimento profundo de concorrência, apenas é utilizado uma barreira, que nada mais é nesse caso do que um contador seguro em fluxos concorrentes. Toda a relação com as *threads* que o sistema operacional coordena é abstraída, e basta indicarmos que cada passo será executado concorrentemente utilizando o comando `go` que o código já está paralelizado. Será mostrado, posteriormente, que no cenário do C++, é necessário um maior cuidado sobre quais partes da matriz serão calculados por quais *threads*.

4.2.3 Comparações com outras linguagens

Para realizar uma comparação de desempenho analisando o tempo de execução, foi criado um código em C++, uma linguagem comumente utilizada para criar aplicações de alto desempenho [carece de fontes].

Código 4.21 – Código de execução paralela

```
#include <cstdlib>

void MultiplyMatrix(int **A, int **B, int **C, int n, int threads, int
    id) {
     for (int i = id; i < n; i+=threads) {
          for (int j = 0; j < n; j++) {
               for (int k = 0; k < n; k++) {
                    C[i][j] = C[i][j] + A[i][k] * B[k][j];
               }
          }
     }
}

void InitMatrix(int **A, int n, bool random) {
     for (int i = 0; i < n; i++) {
          A[i] = (int*) malloc(sizeof(int) * n);
          for (int j = 0; j < n; j++) {
               if (random) {
                    A[i][j] = rand() % 10000;
               } else {
                    A[i][j] = 0;
               }
          }
     }
}
```

Inicialmente, a abordagem do código foi aproximar-se o máximo possível da implementação feita em Go, para que a diferença de performance não seja influenciada pela lógica da aplicação. Porém, dado a diferença de como funciona fluxo concorrentes em cada linguagem, a função de multiplicação tem uma diferença relevante. Cada campo é especificamente calculado por alguma das threads, diferentemente da solução em Go, onde qualquer thread livre poderia executar aquele cálculo. Em C++, não seria possível criar uma *thread* de sistema para cada índice da matriz resultado, dado o tamanho e custo desse tipos de *threads*.Ainda assim, ambos possuem a mesma complexidade de tempo, $O(N^3)$, dado a solução clássica mostrada do problema.

Existem diversas maneiras de particionar o trabalho para cada fluxo de execução. Além das já mencionadas, ainda seria possível fazer uma linha de saída para cada *thread*, diminuindo a quantidade de *threads* necessárias; várias linhas de saída da matriz para cada threads. Foi escolhido, como visto no código 4.21, a opção de cada *thread* calcular um determinado número de linhas com alternância: as linhas escolhidas em cada fluxo não são necessariamente contínuas. Isto pode afetar o desempenho na hora de trazer o

dado da memória, aumentando o acerto de *cache* em dados contíguos, porém não influencia drasticamente, pois a mudança de linha não ocorre tantas vezes no programa, em comparação ao acesso em si de um determinado campo.

A função de preenchimento segue a lógica semelhante a escrita em Go: ou inicializa o vetor com o valor 0, ou cálculo aleatoriamente o valor.

Código 4.22 – Código de execução paralela

```
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <chrono>
#include <thread>

int main(int argc, char *argv[]) {
    srand(time(NULL));
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <size of matrix> <number
            of threads>" << std::endl;
        exit(1);
    }
    int matrix_size = atoi(argv[1]);
        if (matrix_size == 0) {
        std::cout << "Cannot use 0 as matrix size" << std::endl;
        exit(1);
    }
    int threads = atoi(argv[2]);
    if (threads < 1 || threads > 8) {
        std::cout << "Number of threads must be between 1 and 8" << std
            ::endl;
        exit(1);
    }
    int **A = (int**) malloc(sizeof(int*) * matrix_size);
    int **B = (int**) malloc(sizeof(int*) * matrix_size);
    int **C = (int**) malloc(sizeof(int*) * matrix_size);
    InitMatrix(A, matrix_size, true);
    InitMatrix(B, matrix_size, true);
    InitMatrix(C, matrix_size, false);

    auto start = std::chrono::high_resolution_clock::now();
    auto ref_threads = new std::thread[matrix_size];
    for (int i = 0; i < threads; i++) {
        ref_threads[i] = std::thread(MultiplyMatrix, A, A, C,
            matrix_size, threads, i);
    }
    for (int i = 0; i < threads; i++) {
        ref_threads[i].join();
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> dur = end - start;
    std::cout << dur.count() / 1000 << std::endl;
}
```

Semelhantemente, a função principal recebe dois valores: o tamanho da matriz e, neste caso diferentemente em Go, a quantidade de *threads* a ser utilizada. É realizado as inicializações das matrizes, e após isso é realizado a multiplicação, a partir da execução paralela numa outra thread (no mínimo uma outra *thread*). A função principal aguarda então um `join` as *threads*. Por fim, imprimi o valor do tempo da multiplicação em segundos.

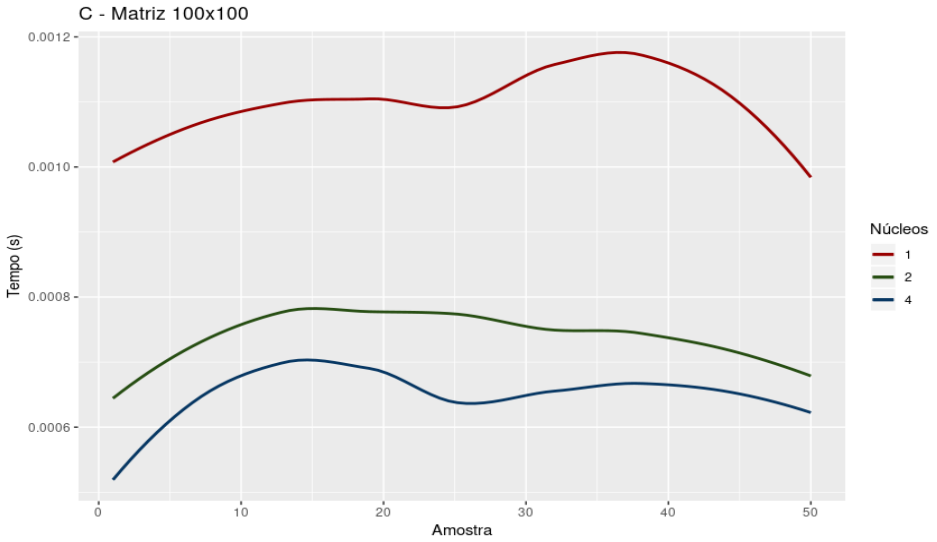
4.2.4 Medições

O compilador utilizado para o C++ foi o `clang`, utilizando *flags* de otimização `Ofast`, semelhante a uma `O3` com algumas outras otimizações. Em Go, foi utilizado o compilador padrão, disponibilizado pelo Google. Todos os testes rodaram na mesma máquina, com sistema operacional Ubuntu 18.04 LTS, com *kernel* do Linux versão 4.18.0-1018-azure x86_64. Foram realizados testes com os mesmos tamanhos de matrizes em ambas as linguagens, com tamanhos de $100x100$, $1000x1000$, e $5000x5000$. Utilizou-se 1, 2 e 4 núcleos de processamento simultâneos, na mesma máquina com CPU 2.4 GHz Intel Xeon® E5-2673 v3 (Haswell).

Foram realizados duas medições por linguagem. A diferença entre cada uma delas, é a ordem dos *loops* das multiplicações: ao inverter a ordem que percorre a matriz, trocando o *loop* mais a dentro com o acima, é possível termos melhores resultados, devido ao príncipio da localidade e *caches* do processador.

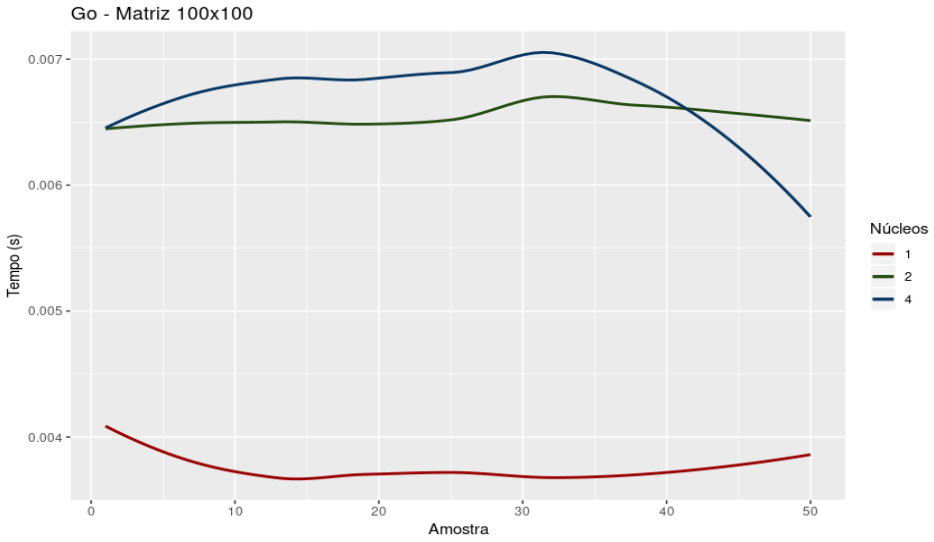
4.2.4.1 Código sem mudança no loop

Figura 1 – Medição do tempo de execução da multiplição de matriz em C



Tamanho da matriz 100x100, sem intervalo de confiança

Figura 2 – Medição do tempo de execução da multiplição de matriz em Go

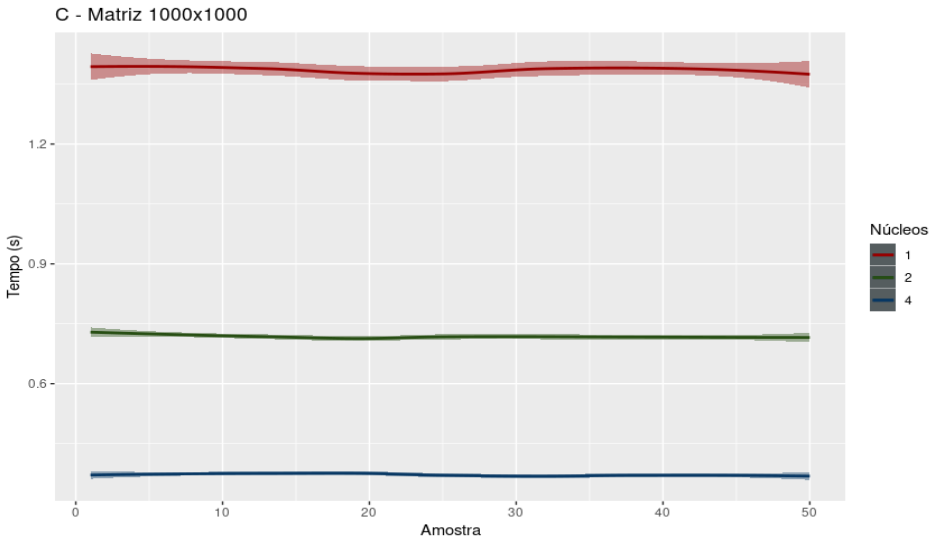


Tamanho da matriz 100x100, sem intervalo de confiança

Na primeira medição de C, temos uma grande variância das execuções. Para este cenário, o tempo de execução já é muito baixo, por volta dos 10 milissegundos. Apesar disso, utilizar *threads* já demonstrou uma melhoria de processamento, ainda que pouca. Não se trabalhou com o intervalo de confiança, dado a alta variabilidade dos dados, e que os resultados mantiveram-se muito baixos.

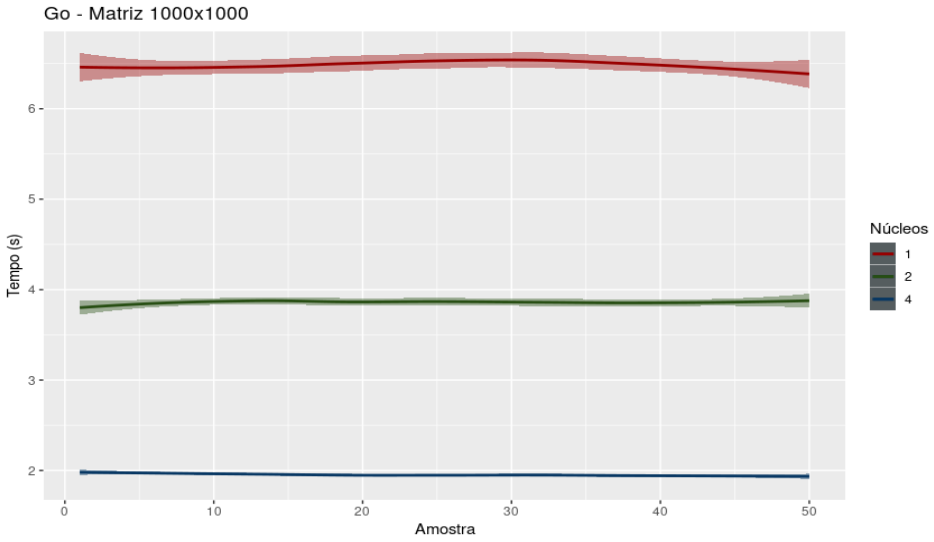
No mesmo cenário em Go, apesar de ter performado mais rápido que em C++, o custo de instanciamento e gerenciamento das diversas *goroutines* acabaram sendo maior que o ganho de executar-se paralelamente. Go performou bem, ainda que mais lento do que C.

Figura 3 – Medição do tempo de execução da multiplição de matriz em C



Tamanho da matriz 1000x1000

Figura 4 – Medição do tempo de execução da multiplição de matriz em Go

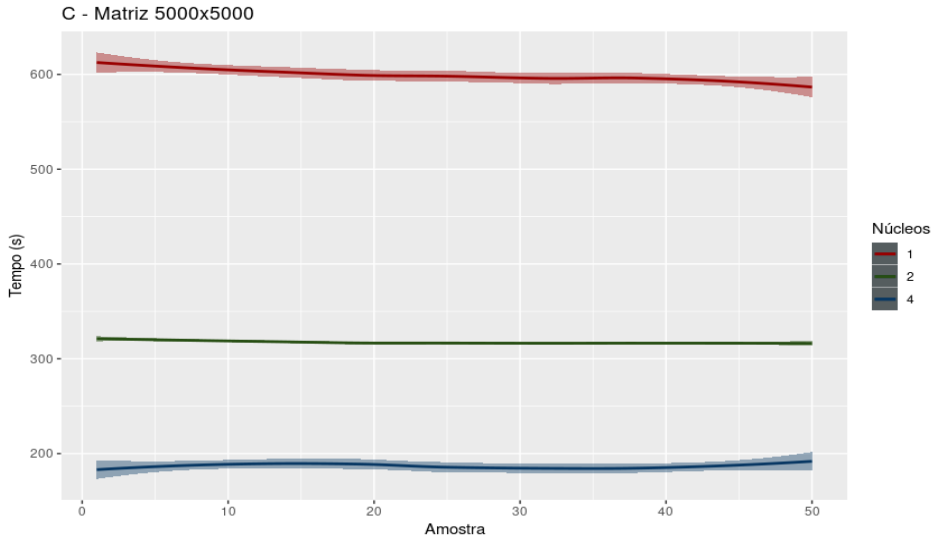


Tamanho da matriz 1000x1000

Ambos os gráficos 3 e 4 o intervalo de confiança é indicado como a área sombreada acima e abaixo da linha dos valores, construindo assim a evolução do intervalo de confiança calculado ao longo das amostras.

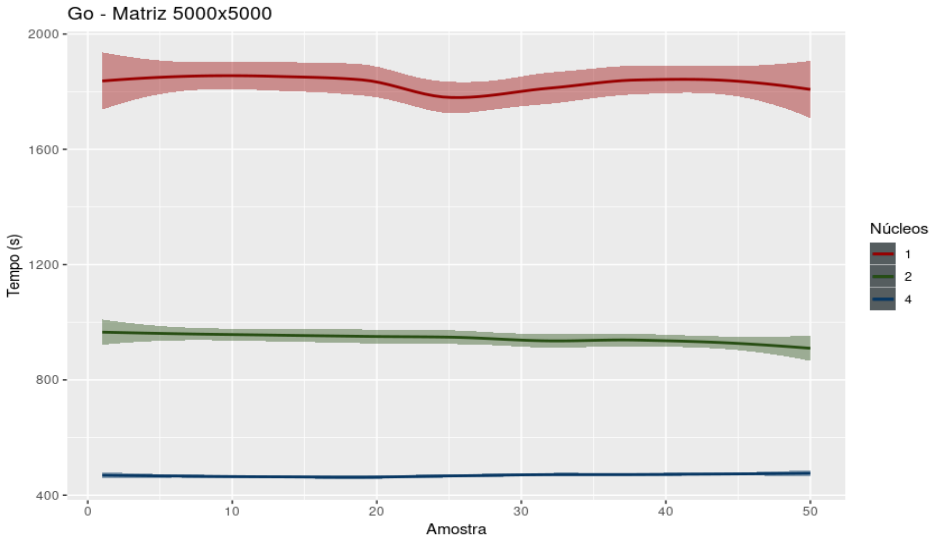
Novamente, temos que o tempo de execução em Go ficou inferior ao tempo de execução em C++. Isso é esperado: dado as abstrações, as questões de *garbage collector*, e a gerência das *goroutines* sendo alocadas e desalocadas de *kernel threads*, a maturidade do compilador de C++, entre diversos outros fatores, contribuem para que C++ mantenha-se mais performático. Ainda assim, já é possível notar no gráfico 4 o ganho de desempenho ao utilizar diversas *threads* em Go. Mesmo que a abstração de *goroutine* tenha um custo, a melhoria de performance ainda se encontra suficientemente próxima percentualmetne de C (2x mais rápido com duas *threads* e 4x mais rápido com quatro *threads*, um crescimento linear).

Figura 5 – Medição do tempo de execução da multiplição de matriz em C



Tamanho da matriz 5000x5000

Figura 6 – Medição do tempo de execução da multiplição de matriz em Go

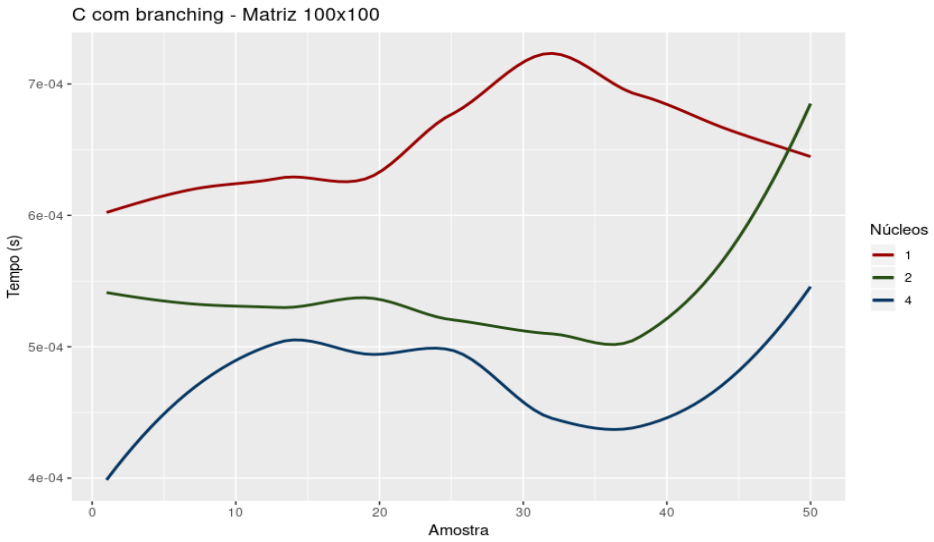


Tamanho da matriz 5000x5000

Por fim, nas execuções com a matriz de tamanho $5000x5000$, os resultados são mais satisfatório para Go. Enquanto nos cenários de $1000x1000$ e $100x100$, o melhor cenário de Go não tenha atingido o pior cenário em C++, agora temos que o cenário de quatro *threads* ficou próximo de 1.5 vezes mais rápido que o cenário de uma *thread* em C++, e quase atingiu o cenário de duas *threads* em C++. Tanto em Go quanto em C++, a paralelização continua linear.

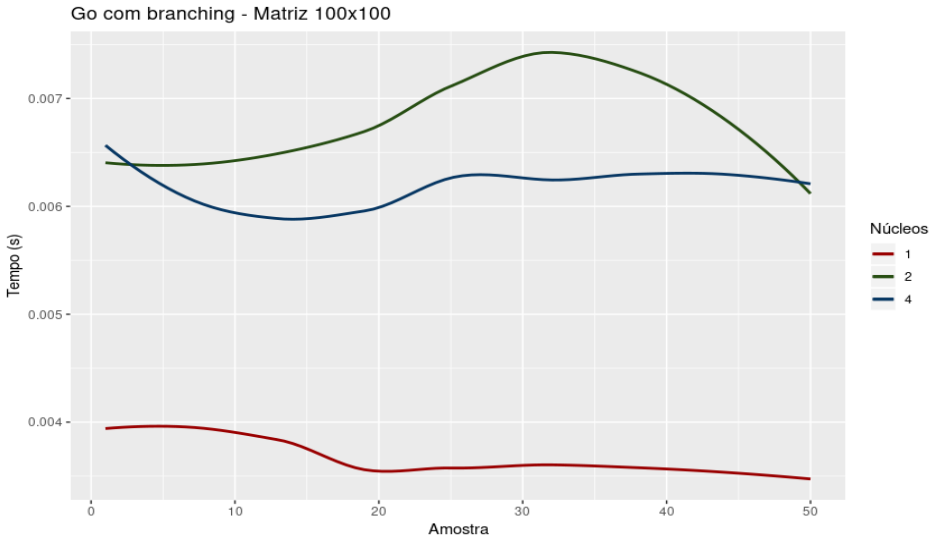
4.2.4.2 Código com mudança no loop

Figura 7 – Medição do tempo de execução da multiplição de matriz otimizada em C



Tamanho da matriz 100x100, sem intervalo de confiança

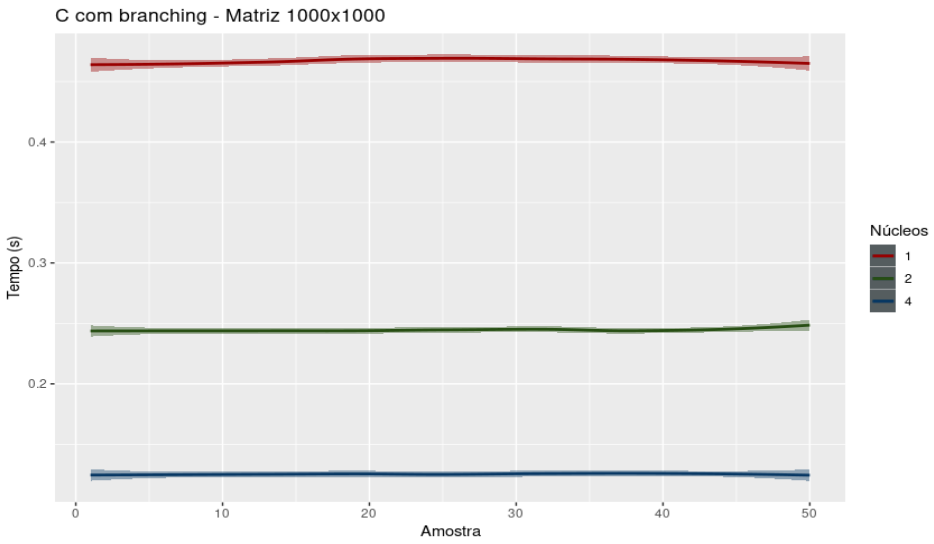
Figura 8 – Medição do tempo de execução da multiplição de matriz otimizada em Go



Tamanho da matriz 100x100, sem intervalo de confiança

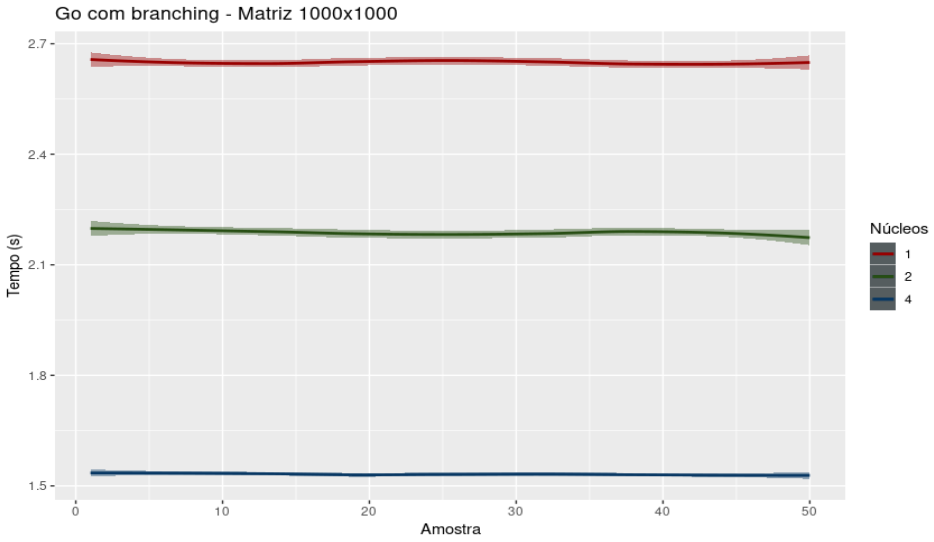
Com a mudança da ordem de *loop*, houve um aumento na variação do cenário em C++, enquanto em Go se manteve mais bem comportado. O cenário de matrizes pequenas apenas ilustram que, num ambiente de baixo processamento, nem sempre paralelização será uma solução.

Figura 9 – Medição do tempo de execução da multiplição de matriz otimizada em C



Matriz 1000x1000

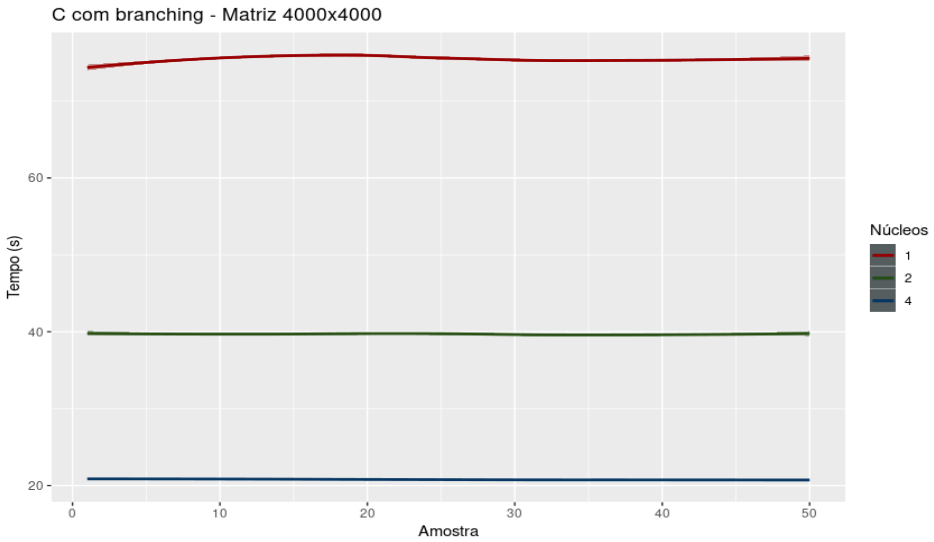
Figura 10 – Medição do tempo de execução da multiplição de matriz otimizada em Go



Matriz: 1000x1000

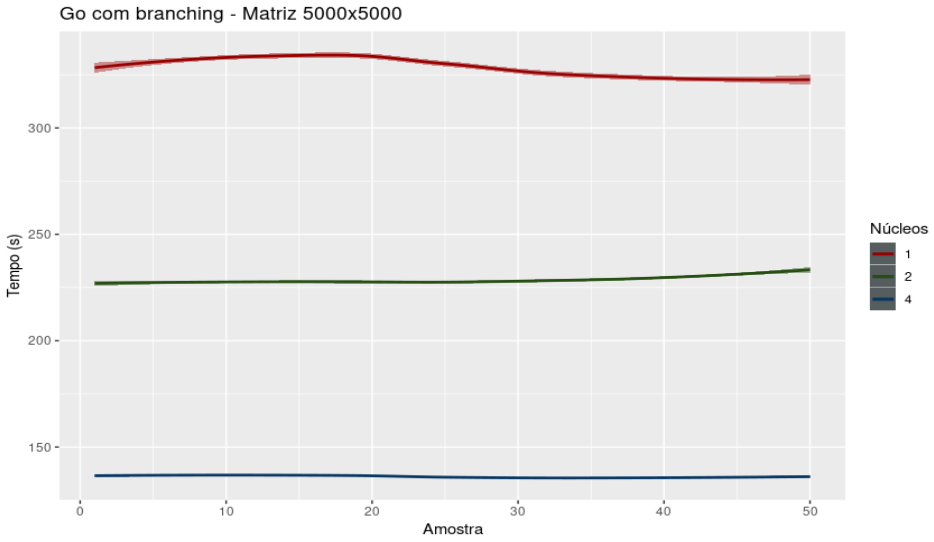
No cenário da matriz $1000x1000$, já é possível ver nos gráficos 9 e 10 que o tempo de execução não está sendo mais linear entre as execuções: o ganho com mais acertos de *cache* é perdido a medida que aumenta-se a paralelização, devido aos novos *cache miss*, dado que a agora o dado deve ser particionado entre os *cores* do CPU.

Figura 11 – Medição do tempo de execução da multiplição de matriz otimizada em C



Matriz: 5000x5000

Figura 12 – Medição do tempo de execução da multiplição de matriz otimizada em Go



Matriz 5000x5000

4.3 SISTEMAS DISTRIBUÍDOS

Dentro da área de concorrência, no modelo convencional de processos, existe os complexos problemas de comunicação interprocessos, muitas vezes essa comunicação se dando por via de alguma rede de computadores. [explicar mais aqui]

4.3.1 Aplicação cliente/servidor

Uma aplicação cliente/servidor é um modelo de comunicação entre processos, cujos servidores são responsáveis por prover algum tipo de recurso enquanto os clientes consomem este recurso. Neste caso, recurso é qualquer tipo de processamento a ser executado ou dado a ser transmitido, como por exemplo, páginas *HTML*, consultas a banco de dados, chamadas de procedimento remotos (RPC), entre outros recursos possíveis. Esses processos podem estar em computadores diferentes, realizando a troca de mensagens a partir da rede.

Neste tipo de modelo, é comum esta comunicação ser realizada a partir de uma *sockets* oferecidos pelo sistema operacional, exposto em alguma porta, e trafega na camada de transporte por meio dos protocolos UDP ou TCP.

Código 4.23 – Exemplo de servidor

```
package main

import (
        "bufio"
        "fmt"
        "net"
        "strings"
)

func main() {
        ln, _ := net.Listen("tcp", ":3000")

        for {
                conn, err := ln.Accept()
                if err != nil {
                        fmt.Println("Can't listen to socket")
                        break
                }
                go func() {
                        message, _ := bufio.NewReader(conn).ReadString('
                            \n')

                        fmt.Print("Message received:", string(message))

                        newMessage := strings.ToUpper(message)

                        conn.Write([]byte(newMessage + "\n"))
                }()
        }
}
```

No código 4.23, temos a função principal do programa servidor. Ele reserva um socket na porta 3000, com endereço de IP qualquer, num loop infinito aceita as conexões que chegam e, caso não ocorra nenhum erro, ele le os bytes enviados, transforma numa string e envia o valor desse texto caixa alto de volta para quem enviou o pacote. Como estamos utilizando o tipo de socket TCP, temos a possibilidade de escrever a resposta.

Código 4.24 – Exemplo de client

```
package main

import (
        "bufio"
        "fmt"
        "net"
        "os"
)

func main() {
        conn, err := net.Dial("tcp", "127.0.0.1:3000")
        if err != nil {
                fmt.Println("Server is down")
                os.Exit(1)
        }
        reader := bufio.NewReader(os.Stdin)
        fmt.Print("Text to send: ")
        text, _ := reader.ReadString('\n')

        fmt.Fprintf(conn, text+"\n")

        message, _ := bufio.NewReader(conn).ReadString('\n')
        fmt.Print("Message from server: " + message)
}
```

Já no código 4.24, temos a função do programa cliente. Ele funciona de maneira mais simples. Utilizando novamente a biblioteca `net`, ele requisita um socket TCP na porta 3000 no endereço local (mesma máquina). Em caso de erro, termina o programa. Após corretamente conseguir conectar-se ao socket, ele aguarda o input de texto do usuário que será enviado para o servidor. Após o envio de um texto (finalizado a partir do input `n`), é feito a leitura da resposta e sua impressão.

Neste simples exemplo, já é possível ver muitas das funções de Go sendo utilizadas. A biblioteca padrão `net` já possui uma solução completa para essa comunicação com *sockets*, e o uso das *goroutines* para responder os sockets concorrentemente é uma solução elegante para problemas em sistemas distribuídos. Por fim, ainda faz-se uso do recurso de *closures*.

4.3.2 Chat

Chat é um tipo de sistema de trocas de mensagens instantâneas, onde vários usuários trocam mensagens por meio de uma interface de entrada e saída de dados. As mensagens podem ser centralizadas num servidor, e esse fica responsável por repassar a mensagem para os outros clientes, ou as mensagens são passados diretamente entre os clientes.

O chat funciona semelhante ao exemplo apresentado de servidor e cliente: um cliente irá mandar uma mensagem pro servidor, e este reproduzirá para todos os clientes que estejam no chat no momento (diferentemente do exemplo anterior, que é só uma troca de mensagem entre o cliente e o servidor). Para isso, é necessário salvar quais são os clientes atualmente conectados, e também remove-los caso algum saia.

Código 4.25 – Código da função principal do servidor

```
package main

import (
        "bufio"
        "fmt"
        "log"
        "math/rand"
        "net"
        "strconv"
        "time"
)

func main() {
    server, err := net.Listen("tcp", ":3000")
    if err != nil {
        panic(err)
    }
    clients := make(map[string]net.Conn)
    dead := make(chan string)
    messages := make(chan string)
    rand.Seed(time.Now().UTC().UnixNano())
    go receivePeople()
    for {
        select {
        case message := <-messages:
            for name, conn := range clients {
                go func(conn net.Conn) {
                    _, err := conn.Write([]byte(message))
                    if err != nil {
                        dead <- name
                    }
                }(conn)
            }
            log.Printf("New message: \"%s\". Broadcast to %d clients\n",
                message[:len(message)-2], len(clients))
        case name := <-dead:
            go func() {
                log.Printf("Client %s disconnected\n", name)
                delete(clients, name)
                messages <- fmt.Sprintf("O usuario %s saiu do chat.\n",
                   name)
            }()
        }
    }
}
```

O código principapl é responsável por inicializar o servidor, que escuta uma porta qualquer (definida como 3000). Além disso, é responsável por criar os mapas que serão utilizados para comunicar a saída de um usuário e o envio de novas mensagens. A função `receivePeople` fica responsável por lidar com a chegada de novos usuários.

O servidor então fica loopando entre duas possibilidades: ou recebeu alguma mensagem no canal de mensagens, ou recebeu um usuário que a conexão foi encerrada. No caso de uma nova mensagem, ele itera pelo mapa dos usuários que estão no chat e escreve a mensagem no socket para cada um deles. Já quando é notificado de uma conexão morta, ele deleta o cliente do mapa.

Código 4.26 – Código da função de recebimento de novos usuarios

```
func receivePeople() {
    for {
        conn, err := server.Accept()
        if err != nil {
            panic(err)
        }
        go func(conn net.Conn) {
            reader := bufio.NewReader(conn)
            _, err = conn.Write([]byte("Welcome to the chat! Please,
                type your name!\n"))
            if err != nil {
                break
            }
            name, err := reader.ReadString('\n')
            if err != nil {
                break
            }
            name = name[:len(name)-2]
            if _, ok := clients[name]; ok {
                for {
                    temp := name + strconv.Itoa(int(rand.Int31n(10e07)))
                    if _, ok := clients[temp]; ok {
                        continue
                    }
                    name = temp
                    break
                }
            }
            clients[name] = conn
            log.Printf("New client: %s! There are %d clients on the chat
                \n", name, len(clients))

            messages <- fmt.Sprintf("O usuario %s entrou do chat.\n",
                name)
            for {
                message, err := reader.ReadString('\n')
                if err != nil {
                    break
                }
                messages <- fmt.Sprintf("[%s]: %s", name, message)
            }
            dead <- name
        }(conn)
    }
}
```

O código da função de novos usuários 4.26 funciona como um loop que aceita novos sockets. Para cada novo socket que ele receber, ele pede ao usuário um nome para ser identificado. Caso o nome ja exista, é gerado um sufixo aleatório até que o nome não exista. Por fim, a função manda uma nova mensagem pro canal de mensagens, que avisa a chegada de um novo usuário para outros clientes. Por fim, ele inicializa a função do novo usuário, que irá ficar em loop ouvindo aquela conexão, até que ocorra algum erro. Quando ocorre um erro, o usuário é adicionado no canal de conexões derrubadas. Todo esse tratamento, após ser aceito a nova conexão, é feito assincronamente, para permitir o tratamento de multiplos clientes chegando simultaneamente.

4.3.2.1 Chat entre clientes

pode nao ter?

4.4 CORROTINAS

Corrotinas é uma proposta de controle abstração de controla, introduzida no inicio dos anos 1960. É atribuído a Conway, que descreveu corrotinas como "subrotinas que agem como o programa mestre"(MOURA; IERUSALIMSCHY, 2009). É comum ao falar sobre corrotinas falar também sobre continuações que são um tipo de representação abstrata do controle de estado da aplicação, que refere-se ao resto de instruções que devem ser executadas pelo seu programa. As características fundamentais definidas por Christopher Marlin (MARLIN, 1980) das corrotinas são:

- Os dados locais de uma corrotina persistem entre sucessivas chamadas;
- A execução de uma corrotina é suspensa quando o controle a deixa, e só retorna no mesmo local que foi deixado. Isto é, no momento que é o fluxo deixa a execução de uma corrotina, ela deve só deverá retornar, para o mesmo local que deixou, quando for novamente chamada.

Não é comum um mecanismo genérico de corrotinas ser disponibilizado por linguagens de programações atuais, apesar de ser comum a presença de geradores, como em Python (citar site), novas versões de Javascript (citar MDN), C (citar documentacao da microsoft), entre outros. Lua disponibiliza corrotinas assimétricas completas, que como demonstrado por Ana Lúcia (MOURA; IERUSALIMSCHY, 2009), é possível construir *one shot continuations*, ou continuações de um turno, subcontinuações, e corrotinas simétricas. Assim, foi criado uma proposta de corrotina assimétrica em Golang, modelada próxima do funcionamento da de Lua.

4.4.1 Implementação em Go

Código 4.27 – Código de definição e criação de corrotinas em Go

```
package coroutines

type Coroutine struct {
        base           func(...interface{}) []interface{}
        yield          chan interface{}
        resume         chan interface{}
        dead, started bool
}

func Create(base func(*Coroutine, ...interface{}) []interface{}) *
    Coroutine {
        coro := &Coroutine{
                yield:   make(chan interface{}),
                resume:  make(chan interface{}),
                dead:    false,
                started: false,
        }
        coro.base = func(args ...interface{}) []interface{} {
                rets := base(coro, args...)
                coro.dead = true
                return coro.Yield(rets...)
        }
        return coro
}
```

O código acima demonstra a criação de uma estrutura de corrotina. A estrutura guarda qual foi a função utilizada para construção (chamada de `base`), dois canais utilizados para transferencia de dados, `yield` e `resume` e por fim duas flags, que indicam o estado da corrotina.

Na criação da corrotina, para que fosse permitido ter a visibilidade da variável sendo utilizada (no caso, a variável de retorno `coro`), foi separado a criação da estrutura e a atribuição da função passada como argumento. Para que a função pudesse aceitar parâmetros indefinidos, e ainda retornar uma lista qualquer de parâmetros, foi utilizado o recurso de um argumento variádrico do tipo de uma interface vazia. O primeiro argumento, porém, deve ser a própria corrotina. Isso é necessário para fazer chamadas de `Yield` dentro da corrotina, já que é necessário uma referência para essa função, que não está disponível no momento da chamada da função `Create`.

Código 4.28 – Continuação do código de corrotinas: *resume* e *yield* das corrotinas

```
func (c *Coroutine) Resume(args ...interface{}) []interface{} {
        if c.dead {
                panic("Cannot resume a dead coroutine")
        }
        if !c.started {
                c.started = true
                go c.base(args...)
        } else {
                for _, value := range args {
                        c.resume <- value
                }
                close(c.resume)
        }
        list := []interface{}{}
        for yieldedValue := range c.yield {
                list = append(list, yieldedValue)
        }
        c.yield = make(chan interface{})
        return list
}

func (c *Coroutine) Yield(rets ...interface{}) (list []interface{}) {
        for _, value := range rets {
                c.yield <- value
        }
        close(c.yield)
        if len(rets) > 0 {
                list = []interface{}{}
                for newArg := range c.resume {
                        list = append(list, newArg)
                }
        }
        c.resume = make(chan interface{})
        return list
}
```

As funções de *resume* e *yield* (do inglês, resumir e produção/produzir) finalizam as três funções básicas propostas pela biblioteca do Lua. A ideia é que, você nunca possa resumir corrotinas já mortas (que tiveram seu processamento finalizado). O resume é utilizado tanto para inicio quanto para reativação das corrotinas. Na sua primeira execução, ele deve sempre chamar a função basicamente por meio de uma gorrotina. Ao final de toda chamada do resume, ele coleta os dados produzidos pela corrotina, que pode ser uma chamada vinda de dentro da função base, ou feita após a finalização da corrotina. Para utilizar o recurso de `range` em canais, é necessário um fechamento dos canais, e por não

haver uma maneira segura de saber se um canal encontra-se fechado, foram utilizados dois canais, em que cada uma das rotinas que leiem é responsável pela recriação desse canal.

Com essas funções, é possível reproduzir o problema proposto no artigo, um exemplo de percorrer uma árvore binária por meio de corrotinas, comparando os seus valores a medida que são impressos.

4.4.2 Exemplo de uso

Código 4.29 – Exemplo de corrotinas: percorrendo árvores binárias

```
package main

import (
        "fmt"
        "tcc-coroutines/coroutines"

        "golang.org/x/tour/tree"
)

func main() {
        var find func(*coroutines.Coroutine, ...interface{}) []interface
            {}

        A := tree.New(2)
        B := tree.New(3)

        find = func(c *coroutines.Coroutine, args ...interface{}) []
            interface{} {
                t := (args[0]).(*tree.Tree)
                if t != nil {
                        find(c, t.Left)
                        c.Yield(t.Value)
                        find(c, t.Right)
                }
                return nil
        }
        stepTree1 := coroutines.Create(find)
        stepTree2 := coroutines.Create(find)
        v1 := stepTree1.Resume(A)
        v2 := stepTree2.Resume(B)
        for len(v1) > 0 || len(v2) > 0 {
                if len(v1) > 0 && (len(v2) == 0 || v1[0].(int) < v2[0].(
                    int)) {
                        fmt.Printf("%v, ", v1[0])
                        v1 = stepTree1.Resume()
                } else {
                        fmt.Printf("%v, ", v2[0])
                        v2 = stepTree2.Resume()
                }
        }
        fmt.Printf("\n")
}
```

Apesar de algumas idiossincrasias do Go, como de tratar variáveis que podem ser de qualquer valor, por meio de *interfaces* vazias, a lógica do programa não é afetada: é realizado a instância das árvores iniciais. Depois, é definido a função base da corrotina, chamada de `find`, que apenas faz o acesso no nó da esquerda, retorna o valor por meio do *yield*, e depois acessa o nó direito. Caso o nó atual seja nulo, ele apenas retorna nulo.

Após isso, é feito a criação das duas corrotinas: uma para percorrer a árvore A, e outra para percorrer a árvore B. Nota-se que aqui, ainda não iniciou-se a execução da rotina. É coletado então os primeiros valores das árvores. Como o retorno é uma lista, o loop funciona enquanto a lista retornada tiver um tamanho maior que 0. Após isso, ele apenas checa se deve imprimir o valor da árvore A ou da árvore B, e chama novamente a corrotina referente ao nó da árvore impresso.

5 CONCLUSÃO

Onde se expõe o fechamento das ideias do estudo, são apresentados os resultados da pesquisa, e partindo da análise destes resultados, tiram-se as conclusões e se for necessário, as sugestões relativas ao estudo.

Observação: É opcional a apresentação dos desdobramentos relativos à importância, síntese, projeção, repercussão, encaminhamento e outros.

REFERÊNCIAS

BALLHAUSEN, H. Fair solution to the reader-writer-problem with semaphores only. **arXiv preprint cs/0303005**, 2003.

BINET, S. Go-HEP: writing concurrent software with ease and go. **Journal of Physics: Conference Series**, IOP Publishing, v. 1085, p. 052012, sep 2018. Disponível em: <https://doi.org/10.1088%2F1742-6596%2F1085%2F5%2F052012>.

BRYANT, R. E.; RICHARD, O. D.; RICHARD, O. D. **Computer systems: a programmer's perspective**. [S.l.]: Prentice Hall Upper Saddle River, 2003. v. 281.

DÍAZ, J.; RAMOS, I. **Formalization of Programming Concepts: International Colloquium, Peniscola, Spain, April 19-25, 1981. Proceedings**. [S.l.]: Springer Science & Business Media, 1981. v. 107.

DIJKSTRA, E. W. Information streams sharing a finite buffer. In: **Inf. Proc. Letters**. [S.l.: s.n.], 1972. v. 1, p. 179–180.

MARLIN, C. D. **Coroutines: a programming methodology, a language design and an implementation**. [S.l.]: Springer Science & Business Media, 1980.

MOURA, A. L. D.; IERUSALIMSCHY, R. Revisiting coroutines. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 31, n. 2, p. 6, 2009.

PIKE, R. et al. **The Go Programming Language Specification**. Disponível em: <https://golang.org/ref/spec>.

GLOSSÁRIO

abnTeX2 suíte para LaTeX que atende os requisitos das normas da ABNT para elaboração de documentos técnicos e científicos brasileiros. *veja* LaTeX

equilíbrio da configuração consistência entre os componentes. *veja também* componente

pai este é uma entrada pai, que possui outras subentradas.. 1) descriação da entrada componente.. .

APÊNDICES

APÊNDICE A – ANÁLISE DOS RELATÓRIOS MENSAIS DE USO DO SERVIÇO DE RENOVAÇÃO DE EMPRÉSTIMOS.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacus nisl, ultricies vitae semper eu, scelerisque nec enim. Curabitur posuere tortor orci, at porta leo laoreet et. Quisque ut congue dolor. Maecenas vel sagittis diam. Praesent fermentum eleifend mi, sit amet vehicula leo pellentesque quis. Curabitur mattis luctus pulvinar. Proin auctor est nec nulla pellentesque commodo. Donec nec justo eu magna aliquet eleifend. Curabitur tristique tortor id sem dignissim, a iaculis metus interdum. Phasellus bibendum velit sit amet interdum semper. Nam vestibulum dui quis nisi consectetur, id vehicula dolor faucibus.

APÊNDICE B – ANÁLISE DOS RELATÓRIOS MENSAIS DE USO DO SERVIÇO DE EMPRÉSTIMO DOMICILIAR.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacus nisl, ultricies vitae semper eu, scelerisque nec enim. Curabitur posuere tortor orci, at porta leo laoreet et. Quisque ut congue dolor. Maecenas vel sagittis diam. Praesent fermentum eleifend mi, sit amet vehicula leo pellentesque quis. Curabitur mattis luctus pulvinar. Proin auctor est nec nulla pellentesque commodo. Donec nec justo eu magna aliquet eleifend. Curabitur tristique tortor id sem dignissim, a iaculis metus interdum. Phasellus bibendum velit sit amet interdum semper. Nam vestibulum dui quis nisi consectetur, id vehicula dolor faucibus.

ANEXOS

ANEXO A – DEMONSTRATIVO DE FREQUÊNCIA DIÁRIA AGO./SET. 2001

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacus nisl, ultricies vitae semper eu, scelerisque nec enim. Curabitur posuere tortor orci, at porta leo laoreet et. Quisque ut congue dolor. Maecenas vel sagittis diam. Praesent fermentum eleifend mi, sit amet vehicula leo pellentesque quis. Curabitur mattis luctus pulvinar. Proin auctor est nec nulla pellentesque commodo. Donec nec justo eu magna aliquet eleifend. Curabitur tristique tortor id sem dignissim, a iaculis metus interdum. Phasellus bibendum velit sit amet interdum semper. Nam vestibulum dui quis nisi consectetur, id vehicula dolor faucibus.

ANEXO B – DEMONSTRATIVO DE FREQUÊNCIA DIÁRIA JAN./DEZ. 2002

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacus nisl, ultricies vitae semper eu, scelerisque nec enim. Curabitur posuere tortor orci, at porta leo laoreet et. Quisque ut congue dolor. Maecenas vel sagittis diam. Praesent fermentum eleifend mi, sit amet vehicula leo pellentesque quis. Curabitur mattis luctus pulvinar. Proin auctor est nec nulla pellentesque commodo. Donec nec justo eu magna aliquet eleifend. Curabitur tristique tortor id sem dignissim, a iaculis metus interdum. Phasellus bibendum velit sit amet interdum semper. Nam vestibulum dui quis nisi consectetur, id vehicula dolor faucibus.