Capítulo

1

Concorrência sem Memória Compartilhada

Gerson Geraldo H. Cavalheiro, André Rauber Du Bois, Alexandro Baldassin

Resumo

Este texto apresenta alternativas ao modelo tradicional de programação concorrente baseado em memória compartilhada, explorando mecanismos que deslocam o foco do controle explícito de threads para a coordenação por meio de dados e comunicação. Inicialmente, são revisados os mecanismos básicos de criação de threads em C++, Rust, Go e Elixir, estabelecendo uma base comum para comparação. Em seguida, discute-se a coordenação por espera explícita, evidenciando suas limitações, e introduz-se a abstração de futures como forma de integrar produção de resultados e sincronização. Na sequência, é abordado o modelo de comunicação por troca de mensagens, com ênfase no papel dos canais como mecanismo estruturador da interação entre unidades de execução. Exemplos nas linguagens consideradas ilustram como essas abstrações se manifestam em diferentes contextos. Ao final, destaca-se a mudança de perspectiva que caracteriza modelos baseados em comunicação, nos quais a sincronização emerge da troca de dados e a coordenação deixa de ser centrada no gerenciamento direto de threads. O texto serve como material complementar para uma apresentação sobre concorrência além do modelo de memória compartilhada.

1.1. Introdução

Multithreading é um modelo de programação no qual um processo é estruturado como um conjunto de múltiplos threads de execução, cada um representando um fluxo de controle independente dentro de um mesmo espaço de endereçamento e compartilhando os recursos do processo [Silberschatz et al. 2018, Tanenbaum and Bos 2022]. Esse modelo permite concorrência intraprocesso, na medida em que múltiplos threads podem progredir de forma intercalada ao longo do tempo, e, quando suportado pelo hardware e pelo escalonamento do sistema operacional, pode possibilitar execução paralela em múltiplas unidades de processamento [Silberschatz et al. 2018, Tanenbaum and Bos 2022].

Em contraposição ao modelo de execução sequencial tradicional associado à arquitetura de von Neumann, no qual as instruções são concebidas como uma única sequência ordenada de operações, o multithreading introduz a existência de múltiplos fluxos de

controle que podem evoluir de forma concorrente dentro de um mesmo programa. Nesse contexto, a noção de ordem de execução deixa de ser global e passa a ser parcialmente definida, exigindo mecanismos adicionais para coordenar a interação entre os diferentes fluxos e garantir a consistência dos resultados produzidos [Silberschatz et al. 2018, Tanenbaum and Bos 2022].

A disponibilização do multithreading nas ferramentas de programação atuais ocorre, em geral, por meio de abstrações que se organizam segundo dois modelos principais: compartilhamento de memória ou troca de mensagens.

O modelo de implementação baseado em compartilhamento de memória é, possivelmente, o mais explorado no contexto acadêmico. Entre seus representantes mais difundidos destacam-se o Pthreads [Kleiman et al. 1996] e o OpenMP [Chandra et al. 2001]. Já o modelo de troca de mensagens consolidou-se em diferentes ambientes e linguagens, como Elixir [Gospodinov 2021] e Go [Cox-Buday 2017], sendo também suportado por linguagens multiparadigma como Rust [Team 2021].

Neste texto, o foco recai sobre mecanismos de programação em ambiente multithreading que vão além do uso convencional de seções críticas em implementações baseadas em memória compartilhada. As linguagens utilizadas como base são C++ [Williams 2019], Rust, Go e Elixir. Caso o leitor deseje maiores informações sobre essas e outras linguagens, inclusive cobrindo recursos relacionados ao compartilhamento de dados em memória, também podem ser consultado o material divulgado em [Cavalheiro et al. 2025, Cavalheiro 2009, Cavalheiro and Santos 2007].

O restante do texto está organizado como segue. Na Seção 1.2 são apresentados os mecanismos básicos de criação de threads nas linguagens consideradas, estabelecendo uma base comum para comparação. Na Seção 1.3 discute-se a coordenação de execução concorrente, iniciando pela espera explícita e avançando para a abstração de futures. Na Seção 1.4 aborda-se a comunicação por troca de mensagens, incluindo o uso de canais e sua realização nas linguagens. Por fim, a Seção 1.5 apresenta uma síntese das abordagens discutidas.

1.2. Criando Threads

Nesta seção são apresentados mecanismos básicos de criação de threads em C++, Rust, Go e Elixir. É importante observar que este texto não cobre todos os mecanismos oferecidos pelas linguagens, tampouco esgota a discussão sobre os mecanismos apresentados. No entanto, o que é apresentado é suficiente para compreensão dos demais conceitos que serão desenvolvidos.

Antes de avançar, é necessário estabelecer uma distinção conceitual. Termos como processo, thread e tarefa referem-se a unidades de execução, porém em níveis distintos de abstração. Um processo caracteriza-se como uma unidade de isolamento, com espaço de endereçamento próprio. Um thread é uma unidade de execução associada a um processo, compartilhando seus recursos. Já uma tarefa representa uma unidade lógica de trabalho, frequentemente gerenciada por um runtime e que pode ou não corresponder diretamente a um thread ou processo. As linguagens e ferramentas adotam nomenclaturas próprias para essas unidades, sendo necessário observar o significado atribuído em cada contexto.

1.2.1. C++

A biblioteca padrão de C++ oferece suporte a multithreading por meio da classe `std::thread`. Um thread é criado pela construção de um objeto dessa classe, ao qual se associa um objeto invocável, como uma função, um functor ou uma expressão lambda. A execução do thread inicia imediatamente após a construção do objeto.

O Código 1 ilustra essas três formas de criação. Cada thread executa de forma concorrente ao fluxo principal do programa.

```cpp
#include <iostream>
#include <thread>

void foo() {
  std::cout << "Executando foo\n";
}

class Functor {
public:
  void operator()() {
    std::cout << "Executando functor\n";
  }
};

int main() {
  Functor f;

  std::thread t1(foo);
  std::thread t2(f);
  std::thread t3([]() {
    std::cout << "Executando lambda\n";
  });

  t1.join();
  t2.join();
  t3.join();
}
```

Código 1: Criação de threads em C++

O controle do thread é realizado por meio do próprio objeto `std::thread`. A chamada ao método `join()` bloqueia o fluxo chamador até a finalização do thread. Alternativamente, o método `detach()` permite que o thread execute de forma independente. Caso o objeto `std::thread` seja destruído enquanto ainda estiver associado a um thread em execução, ocorre a chamada a `std::terminate()`, encerrando o programa.

Esse modelo associa explicitamente o ciclo de vida do thread ao objeto que o representa, exigindo que o programador gerencie sua finalização de forma adequada.

1.2.2. Rust

Rust oferece suporte a criação de threads nativos por meio da função `std::thread::spawn`. Essa função recebe como argumento uma expressão invocável, tipicamente uma closure, que será executada concorrentemente em um novo thread. A execução do thread inicia imediatamente após a chamada a `spawn`.

O Código 2 ilustra a criação de um thread em Rust. A closure utiliza a palavra-chave `move`, que transfere a posse das variáveis capturadas para o novo thread, garantindo segurança no acesso aos dados.

```rust
use std::thread;

fn main() {
  let x = 8;

  let handle = thread::spawn(move || {
    x * x
  });

  let r = handle.join().unwrap();
  println!("{}", r);
}
```

Código 2: Criação de threads em Rust

A função `spawn` retorna um objeto do tipo `JoinHandle`, que representa o thread criado. Esse objeto permite sincronização explícita por meio do método `join()`, que bloqueia o fluxo chamador até a conclusão do thread. O valor retornado pela closure é disponibilizado como resultado da chamada a `join()`, encapsulado em um `Result`.

Diferentemente de C++, Rust não oferece um mecanismo explícito de `detach`. Caso o `JoinHandle` não seja utilizado para sincronização, o thread continuará sua execução de forma independente, sem garantia de conclusão antes do término do programa. O modelo de ownership da linguagem impõe restrições ao compartilhamento de dados entre threads, exigindo transferência de posse ou o uso de abstrações seguras, o que contribui para evitar erros comuns em programação concorrente.

1.2.3. Go

Em Go, a unidade básica de execução concorrente é a *goroutine*. Uma goroutine é criada de forma simples, prefixando a chamada de uma função com a palavra-chave `go`. A execução da função é então agendada pelo runtime da linguagem, ocorrendo de forma concorrente ao fluxo principal do programa.

O Código 3 ilustra a criação de goroutines em Go. Duas tarefas são lançadas a partir da função `main`, uma utilizando função nomeada e outra utilizando uma função anônima.

```go
package main

import (
  "fmt"
  "sync"
)

func f(nome string, wg *sync.WaitGroup) {
  defer wg.Done()
  fmt.Println("Executando:", nome)
}

func main() {
  var wg sync.WaitGroup

  wg.Add(2)

  go f("goroutine 1", &wg)

  go func() {
    defer wg.Done()
    fmt.Println("Executando: goroutine 2")
  }()

  wg.Wait()
  fmt.Println("Função main retornou.")
}
```

Código 3: Criação de goroutines em Go

O runtime de Go é responsável por gerenciar a execução das goroutines, abstraindo detalhes de criação e escalonamento de threads do sistema operacional. Por padrão, o programa é encerrado assim que a função `main` retorna, mesmo que existam goroutines em execução. Por esse motivo, mecanismos de sincronização, como `sync.WaitGroup`, são utilizados para garantir que as goroutines concluam sua execução antes do término do programa.

1.2.4. Elixir

Em Elixir, a unidade básica de execução concorrente é o processo leve, gerenciado pela máquina virtual BEAM. Esses processos são independentes entre si, não compartilham memória e executam de forma concorrente.

A criação de um processo é realizada por meio da função `spawn`, que recebe como argumento uma função a ser executada. A execução ocorre de forma assíncrona em relação ao fluxo que realizou a chamada.

O Código 4 ilustra a criação de um processo em Elixir.

```
defmodule Demo do
  def saudacao do
    IO.puts("Olá do processo!")
  end
end

spawn(Demo, :saudacao, [])
```

Código 4: Criação de processos em Elixir

A função `spawn` retorna um identificador do tipo `pid`, que representa o processo criado. Diferentemente das linguagens apresentadas anteriormente, não há um mecanismo direto equivalente a `join` para aguardar a finalização de um processo. A coordenação entre processos é realizada por meio de troca de mensagens, tratada em seções posteriores.

O runtime da BEAM é responsável pelo gerenciamento e escalonamento dos processos, permitindo a criação de um grande número de unidades concorrentes com baixo custo.

1.3. Coordenação de Execução Concorrente

Nos mecanismos apresentados na Seção 1.2, a coordenação entre fluxos de execução ocorre de forma explícita, por meio de operações que suspendem o fluxo chamador até a conclusão de outro. Esse modelo, embora simples ainda que efetivo, exige controle manual da ordem de execução e não integra diretamente a produção e o consumo de resultados. Recursos de mais alto nível são oferecidos para que o programador tenha sua tarefa simplificada. Importante um lembrete ao leitor: seções críticas (mutex, semáforo) e execução condicional (variáveis de condição) preveem compartilhamento de memória e este texto não trata esse assunto.

1.3.1. Coordenação por Espera Explícita

A forma mais direta de realizar essa coordenação é por meio de operações de espera explícita. Nesses casos, o fluxo chamador é suspenso até que o fluxo concorrente finalize sua execução. Em C++ e Rust, essa coordenação é realizada por meio da chamada ao método `join()`, que bloqueia até o término do thread. Em Go, um comportamento equivalente pode ser obtido com o uso de `sync.WaitGroup`, que permite aguardar a conclusão de múltiplas goroutines. Já em Elixir, não há um mecanismo direto equivalente a `join`. A coordenação entre processos é realizada por meio de troca de mensagens, tipicamente com o uso de `send` e `receive`, permitindo que um processo aguarde explicitamente por uma mensagem que sinalize a conclusão de outra atividade.

O modelo de coordenação baseado em espera explícita, exemplificado pelo uso de `join()` em C++ e Rust e por `WaitGroup` em Go, apresenta uma característica comum: a sincronização é externa ao resultado da computação. O programador deve explicitamente controlar quando aguardar a conclusão de uma atividade concorrente, separando o fluxo de execução da obtenção dos resultados produzidos.

Embora simples, essa abordagem impõe uma estrutura de controle rígida, na qual a ordem de execução deve ser gerenciada manualmente. Essa limitação motiva a introdução de abstrações que integrem a produção e o consumo de resultados em um único mecanismo.

1.3.2. Futures como Abstração

Uma alternativa ao modelo de coordenação por espera explícita consiste em representar o resultado de uma computação concorrente como um valor que será disponibilizado no futuro. Essa ideia é formalizada pela abstração de *future*.

Um *future* representa o resultado de uma computação que pode ainda não ter sido concluída. Em vez de suspender explicitamente o fluxo de execução para aguardar a finalização de uma tarefa, o programa passa a manipular esse resultado de forma indireta, podendo acessá-lo quando necessário.

Essa abordagem integra a produção e o consumo de resultados em um único mecanismo. A sincronização deixa de ser uma operação externa e passa a estar associada ao próprio valor, que encapsula tanto o resultado quanto o estado da computação. Dessa forma, a coordenação entre fluxos de execução torna-se mais declarativa, reduzindo a necessidade de controle manual da ordem de execução.

1.3.3. Futures nas Linguagens

A abstração de *futures* é disponibilizada de forma explícita em C++ e Rust, ainda que com diferenças na forma de uso e no modelo de execução subjacente.

Em C++, a biblioteca padrão oferece os tipos `std::future` e `std::promise`, que permitem representar e manipular resultados produzidos por tarefas concorrentes. Um `future` pode ser obtido, por exemplo, a partir da função `std::async`, que executa uma função de forma concorrente e retorna imediatamente um objeto associado ao resultado futuro da computação. O valor pode ser obtido posteriormente por meio do método `get()`, que bloqueia até que o resultado esteja disponível.

Embora Go e Elixir não ofereçam um tipo explícito de *future* em suas bibliotecas padrão, a ideia pode ser representada por meio de seus mecanismos nativos de comunicação. Em Go, canais podem ser utilizados para transportar o resultado de uma goroutine, permitindo que o consumidor aguarde o valor quando necessário. Em Elixir, o mesmo papel é desempenhado pela troca de mensagens entre processos, em que o recebimento de uma mensagem pode ser interpretado como a obtenção de um resultado futuro.

Dessa forma, ainda que a abstração de *futures* não seja uniformemente exposta em todas as linguagens, sua ideia fundamental, a associação entre resultado e sincronização, pode ser observada em diferentes modelos de concorrência.

```cpp
#include <future>
#include <iostream>

int f() {
  return 313;
}

int main() {
  std::future<int> fut = std::async(f);
  int r = fut.get();
  std::cout << r << std::endl;
}
```

Código 5: Uso de `std::future` em C++

```rust
async fn f() -> i32 {
  313
}

#[tokio::main]
async fn main() {
  let r = f().await;
  println!("{}", r);
}
```

Código 6: Uso de `async/await` em Rust

1.4. Comunicação por Troca de Mensagens

Nesta seção, o foco passa do modelo de coordenação baseado na espera por resultados para um modelo no qual a interação entre fluxos de execução ocorre por meio da troca de dados. Em vez de apenas aguardar a disponibilização de um resultado, considera-se a comunicação explícita entre unidades de execução. Esse modelo elimina a necessidade de compartilhamento direto de memória e estabelece a comunicação como mecanismo central de coordenação em sistemas concorrentes.

1.4.1. Comunicação por Mensagens

Uma alternativa à coordenação baseada em compartilhamento de memória é a comunicação por troca de mensagens, na qual unidades de execução interagem por meio do envio e recebimento explícito de dados. Nesse modelo, em vez de múltiplos threads acessarem diretamente uma região compartilhada da memória, os dados são transferidos entre as unidades de execução por meio de mecanismos de comunicação, evitando o acesso concorrente direto a estruturas compartilhadas.

Esse modelo favorece um estilo de programação no qual a sincronização ocorre como parte do próprio processo de comunicação. Ao enviar ou receber uma mensagem, estabelece-se implicitamente uma relação de coordenação entre as unidades envolvidas, o que reduz os riscos de condições de corrida e simplifica o raciocínio sobre o comportamento concorrente do programa.

Em linguagens como Go, a comunicação é realizada por meio de canais, que permitem a troca segura de dados entre goroutines, promovendo tanto a transmissão de valores quanto a sincronização entre os participantes da comunicação. Operações de envio e recebimento são utilizadas para transferir dados, podendo inclusive bloquear a execução até que a contraparte esteja pronta, estabelecendo um ponto de coordenação entre as tarefas concorrentes.

Já em Elixir, a comunicação ocorre por meio de troca de mensagens entre processos leves, utilizando primitivas como `send` e `receive`. Nesse modelo, processos são criados de forma independente e interagem exclusivamente por mensagens, sendo comum a construção de soluções nas quais múltiplos processos cooperam por meio desse mecanismo para produzir um resultado final.

Assim, a comunicação por mensagens estabelece um modelo no qual a interação entre unidades de execução é explícita e estruturada, deslocando o foco do controle de acesso à memória para a troca organizada de informações entre componentes concorrentes.

1.4.2. Canais como Abstração de Comunicação

Canais constituem uma abstração para comunicação entre unidades de execução concorrentes, permitindo a troca de dados de forma segura sem a necessidade de acesso direto a memória compartilhada. Por meio dessa abstração, valores são explicitamente enviados e recebidos, estabelecendo uma forma estruturada de interação entre tarefas.

Além de viabilizar a transmissão de dados, os canais também promovem sincronização entre as unidades de execução envolvidas. Em particular, operações de envio e recebimento podem impor bloqueio até que a contraparte esteja pronta para prosseguir, fazendo com que o próprio mecanismo de comunicação atue como ponto de coordenação entre tarefas concorrentes.

Os canais podem ser definidos com ou sem buffer. Em canais sem buffer, o envio de um valor exige que haja simultaneamente uma operação de recebimento correspondente, estabelecendo uma sincronização direta entre emissor e receptor. Já canais com buffer permitem que um número limitado de valores seja armazenado temporariamente, possibilitando certo desacoplamento entre produção e consumo de dados.

Outro aspecto relevante é a possibilidade de especialização do uso dos canais, restringindo sua direção para envio ou recebimento. Essa característica permite explicitar o papel das diferentes partes do programa na comunicação, contribuindo para uma organização mais clara das interações entre tarefas.

Dessa forma, os canais estabelecem uma forma de comunicação tipada e estruturada, na qual a troca de dados e a sincronização são tratadas de maneira integrada, favorecendo a construção de programas concorrentes que evitam os problemas associados ao

uso de memória compartilhada.

1.4.3. Comunicação nas Linguagens

A comunicação por troca de mensagens manifesta-se de forma explícita nas linguagens consideradas, sendo possível observar diferentes formas de estruturar essa interação entre unidades de execução.

Em Go, a comunicação ocorre por meio de canais, que permitem a troca direta de valores entre goroutines. No exemplo do cálculo da sequência de Fibonacci (Código 7), uma goroutine é responsável por gerar os valores e enviá-los por um canal, enquanto outra goroutine realiza o consumo desses valores. O envio é realizado com o operador $<-$ aplicado ao canal, enquanto o recebimento utiliza o mesmo operador em sentido inverso. Esse mecanismo estabelece uma relação direta entre produtor e consumidor, na qual a comunicação também atua como forma de sincronização, uma vez que o envio e o recebimento podem bloquear até que ambas as partes estejam prontas para prosseguir.

```go
func fib(n, ch chan <- uint64) {
  if n <= 1 {
    ch <- uint64(n)
    return
  }
  ch1 := make(chan uint64)
  ch2 := make(chan uint64)
  go fib(n-1, ch1)
  go fib(n-2, ch2)
  res1 := <-ch1
  res2 := <-ch2
  ch <- res1 + res2
}
```

```go
func main() {
  n
  resultCh := make(chan uint64)
  go fib(n, resultCh)
  result := <-resultCh
}
```

Código 7: Fibonacci em Go com sincronização via canais.

Já em Elixir, a comunicação é realizada por meio do envio e recebimento explícito de mensagens entre processos (Código 8). No exemplo do cálculo de Fibonacci, um processo envia mensagens contendo resultados intermediários, enquanto outro processo utiliza a construção `receive` para aguardar e tratar essas mensagens. Cada processo possui sua própria fila de mensagens, e a interação ocorre exclusivamente por meio desse mecanismo, sem qualquer compartilhamento direto de memória.

```elixir
defmodule Fib do
  def compute(n) when n <= 1, do: n
  def compute(n) do
    parent = self()
    spawn(fn ->
      send(parent, {:fib1, compute(n - 1)})
    end)
    spawn(fn ->
      send(parent, {:fib2, compute(n - 2)})
    end)
    receive do
      {:fib1, res1} ->
        receive do
          {:fib2, res2} ->
            res1 + res2
        end
    end
  end
end
defmodule Main do
  def main(_) do
    n = 10
    result = Fib.compute(n)
  end
end
Main.main([])
```

Código 8: Fibonacci em Elixir com troca de mensagens explícitas.

Esse modelo pode ser observado também no padrão produtor-consumidor imple-

mentado em Elixir, no qual um processo produtor envia mensagens representando dados produzidos, enquanto um processo consumidor as recebe e processa. A coordenação entre esses processos ocorre inteiramente por meio da troca de mensagens, evidenciando um modelo no qual a comunicação é o elemento central de organização do fluxo concorrente.

Assim, tanto em Go quanto em Elixir, a comunicação por mensagens não apenas viabiliza a troca de dados, mas também estrutura a coordenação entre unidades de execução, integrando comunicação e sincronização em um único mecanismo.

1.4.4. Modelos de Concorrência Baseados em Comunicação

Os mecanismos apresentados nas seções anteriores evidenciam uma mudança de perspectiva na forma de estruturar programas concorrentes. Em contraste com o modelo centrado em threads, no qual a coordenação é realizada por meio do controle explícito da execução, as abordagens baseadas em comunicação deslocam o foco para a interação entre unidades de execução.

Nesse contexto, a concorrência deixa de ser organizada em torno da gestão de threads e passa a ser estruturada a partir do fluxo de dados entre entidades independentes. A comunicação não atua apenas como meio de troca de informações, mas também como mecanismo de sincronização, uma vez que a disponibilidade dos dados passa a determinar o progresso da execução.

Abstrações como futures e canais materializam essa mudança. Em vez de bloquear explicitamente a execução até o término de uma atividade, o programa passa a reagir à disponibilidade de resultados ou mensagens, permitindo uma composição mais direta entre produção e consumo de dados.

Como consequência, o controle da execução torna-se menos centralizado, e a coordenação emerge da própria estrutura de comunicação do programa. Esse deslocamento reduz a necessidade de gerenciamento explícito de estados compartilhados e favorece a construção de programas concorrentes mais modulares.

1.5. Discussão Final

Neste texto foram apresentadas diferentes formas de estruturar programas concorrentes, variando desde modelos centrados no controle explícito de threads até abordagens baseadas na comunicação entre unidades de execução. Mecanismos baseados em compartilhamento de memória não foram tratados.

Para auxiliar uma visão do conteúdo, os principais conceitos tratados estão sumarizados na Tabela 1.1. A apresentação desta tabela busca sintetiza essas diferenças entre C++, Rust, Go e Elixir no tema.

Tabela 1.1. Comparação entre modelos de concorrência nas linguagens consideradas

Linguagem	Unidade	Coordenação	Comunicação	Threads explícitas	Futures
C++	Thread	Espera (join)	Secundária	Sim	Sim
Rust	Thread/Tarefa	Espera e futures	Secundária	Sim	Sim
Go	Goroutine	Comunicação	Central	Não	Implícito
Elixir	Processo	Comunicação	Central	Não	Implícito

A tabela apresentada evidencia duas abordagens distintas para a construção de programas concorrentes. Em C++ e, pelo menos na sua forma nativa, em Rust, a estruturação da concorrência permanece centrada no thread, com coordenação realizada por mecanismos de espera explícita ou por abstrações que ainda mantêm relação direta com o fluxo de execução. Nesses casos, a comunicação desempenha papel secundário, sendo utilizada como complemento ao controle do programa.

Por outro lado, Go e Elixir adotam uma abordagem na qual a comunicação assume papel central. Nessas linguagens, as unidades de execução são tratadas como entidades independentes, e a coordenação ocorre por meio da troca de mensagens ou do uso de canais. Como consequência, a sincronização deixa de ser uma operação explicitamente controlada sobre threads e passa a emergir da própria interação entre os componentes do programa. Esse deslocamento caracteriza uma mudança de modelo, na qual o foco deixa de estar na gestão de execução e passa a privilegiar o fluxo de dados.

Referências

[Cavalheiro 2009] Cavalheiro, G. G. H. (2009). Programação com pthreads. In Mattos, J. C. B., Da Rosa Junior, L. S., and Pilla, M. L., editors, *Desafios e Avanços em Computação: O Estado da Arte*, pages 137–151. Editora e Gráfica Universitária - PREC UFPel, Pelotas.

[Cavalheiro et al. 2025] Cavalheiro, G. G. H., Baldassin, A., and Bois, A. R. D. (2025). Programação multithread: Modelos e abstrações em linguagens contemporâneas. In Musse, S. R. and dos Santos, A. P., editors, *44a Jornada de Atualização em Informática (JAI 2025)*. Sociedade Brasileira de Computação (SBC), Porto Alegre.

[Cavalheiro and Santos 2007] Cavalheiro, G. G. H. and Santos, R. R. (2007). Multiprogramação leve em arquiteturas multi-core. In Kowaltowski, T. and Breitman, K. K., editors, *Atualizações em Informática 2007*, pages 327–379. PUC-Rio, Rio de Janeiro.

[Chandra et al. 2001] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Cox-Buday 2017] Cox-Buday, K. (2017). *Concurrency in Go: Tools and Techniques for Developers*. O'Reilly Media, Inc., 1st edition.

[Gospodinov 2021] Gospodinov, S. (2021). *Concurrent Data Processing in Elixir: Fast, Resilient Applications with OTP, GenState, Flow, and Broadway*. The Pragmatic Bookshelf, Raleigh, North Carolina.

[Kleiman et al. 1996] Kleiman, S., Shah, D., and Smaalders, B. (1996). *Programming with Threads*. Sun Soft Press ; Prentice Hall, Mountain View, Calif. ; Upper Saddle River, NJ.

[Silberschatz et al. 2018] Silberschatz, A., Galvin, P. B., and Gagne, G. (2018). *Operating System Concepts*. Wiley, 10 edition.

[Tanenbaum and Bos 2022] Tanenbaum, A. S. and Bos, H. (2022). *Modern Operating Systems*. Prentice Hall, 5 edition.

[Team 2021] Team, T. R. (2021). *The Rust Programming Language*. Rust Core Team. Official language guide.

[Williams 2019] Williams, A. (2019). *C++ Concurrency in Action, Second Edition*. Manning, 2 edition.