Capítulo 3

Programação Multithread: Modelos e Abstrações em Linguagens Contemporâneas

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

Resumo

Este capítulo oferece uma introdução à programação concorrente, com foco em linguagens que possuem suporte nativo a multithreading. São apresentados os fundamentos da concorrência, os principais modelos de implementação de threads e, em seguida, analisadas as abordagens adotadas por C++, Rust, Go e Elixir. O estudo destaca como cada linguagem organiza suas construções para programação concorrente e inclui dois estudos de caso que ilustram a aplicação prática dos conceitos. O conteúdo serve como referência inicial para aprofundar o domínio técnico e explorar criticamente as potencialidades das linguagens abordadas.

Abstract

This chapter provides an introduction to concurrent programming, focusing on languages with native support for multithreading. It presents the fundamentals of concurrency, the main thread implementation models, and analyzes the approaches adopted by C++, Rust, Go, and Elixir. The study highlights how each language structures its constructs for concurrent programming and includes two case studies that illustrate the practical application of the discussed concepts. The content serves as an initial reference for deepening technical knowledge and critically exploring the capabilities of the featured languages.

3.1. Apresentação

Este texto se apresenta como um material ao apoio ao estudo da programação concorrente e multithread utilizando ferramentas de programação contemporâneas. A proposta é combinar uma abordagem teórica com exemplos práticos, oferecendo uma visão abrangente dos fundamentos, modelos e estratégias adotados para explorar o multithreading em C++, Rust, Go e Elixir.

Este material cobre os conceitos e mecanismos disponíveis nas linguagens escolhidas, destacando seus principais recursos para programação concorrente. O texto apresentado, portanto, não se propõe a realizar comparações de desempenho entre as linguagens nem a discutir, de forma aprofundada, critérios para a escolha de uma linguagem em detrimento de outra, considerando diferentes cenários de aplicação.

Nas seções iniciais, Seção 3.2 e Seção 3.3, são apresentados os fundamentos da programação multithread e as principais estratégias de implementação, com o objetivo de uniformizar a terminologia adotada ao longo do texto. As seções seguintes abordam, em sequência, os modelos de execução, as formas de comunicação entre tarefas e os mecanismos de sincronização disponíveis nas quatro linguagens de programação contemporâneas selecionadas para compor este material: C++ (Seção 3.4), com seu modelo de threads nativos e controle explícito por mutexes; Rust (Seção 3.5), que combina segurança de memória com mecanismos de sincronização seguros por construção; Go (Seção 3.6), cujo modelo baseado em goroutines e canais simplifica a programação concorrente; e Elixir (Seção 3.7), que segue o modelo de atores, promovendo isolamento entre processos e tolerância a falhas. A Seção 3.8 discute a implementação de dois casos de estudo nas diferentes linguagens, e a Seção 3.9 apresenta as considerações finais.

3.2. Fundamentos do Multithreading

A crescente complexidade dos sistemas computacionais e a evolução das arquiteturas de hardware, especialmente com a ubiquidade dos processadores multicore, tornaram a programação concorrente um elemento central no desenvolvimento de software moderno [Sutter and Larus 2005]. Como reflexo, linguagens tradicionais passaram a incorporar mecanismos de concorrência de forma nativa, e as linguagens recentes têm sido concebidas para operar em contextos de programação paralela [McCool et al. 2012, Rauber and Rünger 2010].

3.2.1. Motivações e Contexto

A programação concorrente tornou-se indispensável para utilizar efetivamente as arquiteturas multicore, que emergiram como substitutas das arquiteturas monoprocessadas diante das limitações físicas no aumento da frequência dos processadores e da demanda por maior poder computacional [Gupta et al. 2021]. Atualmente, qualquer equipamento computacional, de laptops a smartphones, dispõe de múltiplos núcleos de processamento. No entanto, a simples presença desse recurso não garante ganhos de desempenho: cabe ao software explorar esse paralelismo. Apesar dos avanços no desenvolvimento de novas abstrações, a maioria das linguagens de programação ainda exige que o programador expresse explicitamente o uso de múltiplos fluxos. Isso pode ocorrer por meio de threads, tarefas assíncronas ou outras abstrações [Skillicorn and Talia 1998]. Além disso, é necessário empregar mecanismos para coordenar esses fluxos na cooperação por dados.

Neste contexto, é importante entender com clareza os conceitos de concorrência e paralelismo. Ainda que bastante relacionados, esses conceitos expressam ideias distintas, e é fato que os autores os empreguem de formas diferentes [Pacheco 2022]. A concorrência pode ser entendida pela presença de múltiplas tarefas em progresso em um programa, independentemente de serem executadas no mesmo instante de tempo real (tempo de re-

lógico). Um sistema multitarefa é um exemplo típico: mesmo em uma máquina com apenas um núcleo, diversas tarefas, como leitura de entrada, atualização de tela e execução de processos em segundo plano, podem estar em andamento por meio de revezamento no acesso aos recursos de processamento. Esse comportamento caracteriza um sistema concorrente. Por outro lado, o paralelismo implica a execução simultânea dessas tarefas, com múltiplos núcleos cooperando diretamente na resolução de um problema. Assim, todo programa paralelo é também concorrente [Pacheco 2022].

É também usual associar *concorrência* a uma propriedade da aplicação (um programa descreve atividades concorrentes entre si) e *paralelismo* a uma propriedade do hardware (uma arquitetura possui recursos suficientes para suportar a execução de duas ou mais de atividades simultâneas) [Pacheco 2022][Herlihy et al. 2020]. Assim, é comum que a expressão "programação concorrente" se refira à programação voltada à exposição da concorrência da aplicação, enquanto "programação paralela" seja relacionada à programação focada em extrair o máximo de desempenho de uma determinada arquitetura.

Neste texto, a distinção entre concorrência e paralelismo, assim como entre programação concorrente e paralela não é destacada. De toda forma, o leitor deve observar que o enfoque é dado para a exploração da programação multithread pela exposição da concorrência de atividades concorrentes nos programas e expectativa de execução paralela sobre o hardware multiprocessado disponível.

As aplicações que se beneficiam de múltiplos fluxos são numerosas e crescentes. Sistemas embarcados, como os de controle automotivo, precisam lidar com sensores, atuadores e lógica de controle em tempo real. Servidores web modernos atendem centenas de requisições simultâneas, exigindo alto grau de concorrência. Ambientes de computação científica e aprendizado de máquina exploram paralelismo para reduzir o tempo de programação de tarefas computacionalmente intensivas. Em todos esses contextos, o domínio de técnicas de programação multithread é um diferencial essencial.

3.2.2. Infraestrutura de Suporte ao Multithreading

O estudo da programação multithread segue-se aos estudos das arquiteturas de computadores e dos sistemas operacionais. Embora a disponibilidade de uma arquitetura paralela não seja um requisito essencial para a programação multithread, a onipresença de processadores multicore torna praticamente inconcebível o desenvolvimento de um programa em que não seja prevista sua programação sobre uma arquitetura multiprocessada. Da mesma forma, os sistemas operacionais atuais são capazes de gerenciar centenas, e até milhares, de fluxos de programação nas diversas unidades de processamento providas por tais arquiteturas [Tanenbaum and Bos 2022][Silberschatz et al. 2018].

As arquiteturas paralelas consideradas neste texto tratam-se de multiprocessadores [Culler et al. 1999], compostos por múltiplos núcleos de processamento compartilhando um espaço de endereçamento comum. Este modelo se apresenta em duas organizações arquiteturais: UMA (Uniform Memory Access) e NUMA (Non-Uniform Memory Access). No modelo UMA, presente em equipamentos como notebooks e desktops, todos os núcleos compartilham uma única memória principal, na qual o acesso a qualquer posição de memória a partir de qualquer núcleo se dá em tempo constante. Nas arquiteturas NUMA, típicas em servidores de maior porte, o acesso à memória é segmentado em regiões, cada

uma mais próxima a determinados núcleos, o que reduz a contenção no acesso à memória e permite uma escalabilidade mais eficiente em termos de número de núcleos disponíveis. Embora mecanismos de escalonamento e estratégias de programação possam explorar de forma eficiente as características de uma arquitetura NUMA, este texto se concentra, de forma geral, no modelo básico de programação sobre uma arquitetura multiprocessada.

No sistema operacional, a abstração fundamental para a programação de programas é o processo. Cada processo possui seu próprio espaço de endereçamento e contexto de programação, sendo representados no sistema operacional por uma estrutura de dados chamada PCB (Process Control Block). Um processo sequencial convencional possui um único fluxo de execução, denominado neste texto como thread. Um processo, ao lançar múltiplos fluxos de execução é dito multithread. Os threads, por sua vez, são considerados processos leves em contraste aos processos convencionais, mais pesados para serem manipulados pelo sistema operacional. Threads compartilharem o espaço de endereçamento e outras estruturas com os demais threads do mesmo processo, o que torna sua criação e escalonamento mais eficientes. O padrão POSIX Threads (Pthreads) é implementado nativamente nos principais sistemas operacionais modernos, como Linux, macOS e Windows, e serve de base para diversas linguagens e bibliotecas de programação concorrente.

O suporte ao multithreading também se estende ao hardware. Processadores modernos incluem mecanismos que garantem a atomicidade de certas operações, evitando interferências indesejadas entre threads que compartilham dados. Instruções atômicas, como `compare-and-swap` ou `test-and-set`, são fundamentais para a implementação de estruturas de sincronização como mutexes e semáforos. Além disso, recursos como registradores de controle, caches com políticas de coerência e instruções de barreira de memória contribuem para a programação eficiente de programas concorrentes em ambientes com múltiplos núcleos.

3.2.3. Princípios da Programação Concorrente

A programação concorrente lida com a coordenação de múltiplas unidades de programação com o objetivo de aproveitar os recursos disponíveis em sistemas computacionais com múltiplos núcleos. Para compreender as diferentes formas de concorrência, é importante distinguir os conceitos de processo, thread e tarefa, utilizados para determinar o encapsulamento de uma unidade de trabalho nos programas. Um processo é uma instância isolada de execução, com seu próprio espaço de endereçamento e recursos de sistema, como descritores de arquivos e registradores. Os threads, por sua vez, são unidades de programação mais leves que compartilham o espaço de memória e recursos do processo ao qual pertencem. Essa leveza torna o uso de threads mais eficiente em termos de criação, comunicação e escalonamento, ainda que traga a necessidade de mecanismos de sincronização para lidar com o acesso simultâneo a dados compartilhados na memória do processo. Já as tarefas representam unidades lógicas de trabalho, independentes em termos conceituais, mas que podem ser implementadas como threads, processos ou até corrotinas, dependendo do ambiente de programação adotado.

Além da especificação das unidades de trabalho, a sincronização é um dos principais desafios na programação concorrente [Herlihy et al. 2020]. Como múltiplos threads podem acessar simultaneamente variáveis ou estruturas de dados compartilhadas, é neces-

sário coordenar esses acessos para evitar inconsistências. A exclusão mútua garante que apenas um thread acesse determinada região crítica de código por vez. Mecanismos como mutexes, semáforos e barreiras são amplamente utilizados para esse fim, sendo estes os principais recursos para coordenação de atividasdes em ferramentas multithread clássicas, como Pthreads [Kleiman et al. 1996] e OpenMP [Chandra et al. 2001], e também em C++ e Rust. Quando a sincronização é negligenciada, surgem problemas como condições de corrida, em que o comportamento do programa depende da ordem de execução dos threads e pode levar a resultados incorretos ou imprevisíveis.

Uma alternativa à sincronização por exclusão mútua é a comunicação por troca de mensagens, na qual threads ou tarefas interagem por meio de canais de comunicação. Nesse modelo, em vez de múltiplas unidades de programação acessarem diretamente uma região compartilhada da memória, os dados são explicitamente enviados e recebidos entre elas. Isso favorece um estilo de programação em que a sincronização ocorre como parte da comunicação, reduzindo os riscos de condições de corrida e simplificando o raciocínio sobre a concorrência. Linguagens como Go e Elixir adotam esse modelo como principal mecanismo de interação entre tarefas, mas muitas bibliotecas em linguagens tradicionais também oferecem suporte a canais.

Outro aspecto importante diz respeito à granularidade, tanto da unidade de trabalho quanto das seções críticas. A granularidade em relação a unidade de trabalho refere-se ao comprimento, em termos de número de operações, que as unidades de trabalho descrevem [McCool et al. 2012] [Rauber and Rünger 2010]. Unidades de trabalho com poucas operações, sobre-expondo a concorrência da aplicação, podem gerar sobrecarga, devido ao custo de gerenciamento do grande número de atividades geradas. Por outro lado, unidades muito longas podem limitar o aproveitamento do paralelismo disponível, serializando a execução do programa e, potencialmente, não explorando eficientemente os recursos de hardware disponíveis. Essa limitação está relacionada à Lei de Amdahl [Amdahl 1967], segundo a qual o ganho de desempenho com paralelismo é limitado pela fração sequencial do programa.

A granularidade das seções críticas [Herlihy et al. 2020][McCool et al. 2012] possui outra natureza e diz respeito à quantidade de recursos (dados) adquiridos para execução de uma determinada seção crítica, influenciando diretamente o tempo de bloqueio. Neste caso, uma abordagem de granularidade grossa utiliza um único ponto de sincronização, um mutex, por exemplo, para proteger diversos dados compartilhados. Esta estratégia simplifica o código, embora aumente a possibilidade de serialização indesejada, uma vez que pode forçar a exclusividade ao acesso a um dado não utilizado em determinada seção crítica, mas necessário em outra. Por outro lado, uma granularidade fina, por exemplo, individualizando cada dado compartilhado a um mutex distintos, tem potencial para ampliar o paralelismo que pode ser explorado, mas acaba por elevar a complexidade do programa ao mesmo tempo em que eleva também o custo computacional de manipulação dos diversos recursos de sincronização.

Por fim, a programação concorrente está sujeita a problemas clássicos de implementação [McCool et al. 2012][Herlihy et al. 2020], como condição de corrida, deadlock e starvation. O deadlock ocorre quando dois ou mais threads ficam bloqueados indefinidamente esperando por recursos que nunca serão liberados. O starvation se dá quando

um thread é repetidamente preterido no acesso a recursos, não conseguindo progredir. A prevenção e mitigação desses problemas requer estratégias específicas, como ordenação consistente de aquisição de recursos, uso de algoritmos de escalonamento justos e aplicação de técnicas de detecção e recuperação.

3.2.4. Panorama das Linguagens Abordadas

A concepção de multithreading surgiu no contexto de sistemas operacionais, como uma estratégia para permitir a existência de múltiplos fluxos de controle dentro de um mesmo processo, visando melhorar a responsividade e o aproveitamento dos recursos disponíveis. A biblioteca POSIX Threads (pthreads) [Kleiman et al. 1996] tornou-se uma representação canônica dessa abordagem, fornecendo uma API padronizada e portátil para criação, sincronização e gerenciamento de threads no nível do sistema. Ainda que ferramentas oferecidas na forma de uma biblioteca não sejam as mais convenientes ao desenvolvimento de programas multithread [Boehm 2005], é fato que ela se popularizou neste uso. A linguagem Ada [Burns and Wellings 1998] nos anos 1970, por sua vez, introduziu conceitos de concorrência no nível da linguagem de programação, oferecendo suporte a tarefas e ao mecanismo de sincronização por *rendezvous* (mecanismo semelhante à troca de mensagens ou canais, que não utiliza dados em memória para comunicação). Já no final do século 20, emerge o padrão OpenMP, resultado de um esforço coordenado para fornecer uma interface de programação de alto nível voltada à exploração eficiente de paralelismo em arquiteturas multiprocessadas, especialmente em aplicações científicas.

Pthreads, OpenMP e Ada, entre outras, são as ferramentas de programação frequentemente utilizadas para o ensino da programação concorrente [Silva et al. 2022] [Pacheco 2022] [Andrews 1999] [Cavaleiro and Santos 2007] [Cavalheiro 2009] [Pilla et al. 2009][Cavaleiro and Du Bois 2014], ainda que Ada em menor escala atualmente. No entanto, o mercado está repleto de novas abordagens para programação concorrente, disponibilizadas em novas linguagens de programação ou em linguagens de programação consolidadas, revisitadas no atual contexto da ubiquidade dos multiprocessadores pela ploriferação das arquiteturas de processadores multicore. Neste texto são abordadas quatro dessas novas abordagens, oferecidas por C++, Rust, Go e Elixir. Cada uma dessas linguagens adota estratégias distintas quanto à unidade de concorrência, formas de comunicação entre tarefas e mecanismos de sincronização.

O C++, nas suas versões a partir de C++11, utiliza um modelo baseado em threads nativos, acessados por meio da biblioteca `std::thread`. A criação, o controle e a sincronização dos threads são de responsabilidade do programador, que deve recorrer a primitivas como mutexes e locks para garantir exclusão mútua em regiões críticas. A comunicação entre threads se dá por memória compartilhada, exigindo sincronização cuidadosa para evitar problemas como condições de corrida e deadlocks. As versões mais recentes de C++ (a atual especificação estável é a C++23, embora a maioria dos compiladores esteja suportando efetivamente a versão C++20) incorporaram muitas funcionalidades ao suporte originalmente ofereceido por C++11.

Rust também oferece threads nativos, mas incorpora mecanismos de segurança baseados em seu sistema de tipos. As regras de ownership e borrowing, verificadas em tempo de compilação, restringem o compartilhamento de dados entre threads, prevenindo

erros comuns em modelos de memória compartilhada. A linguagem ainda oferece suporte a tarefas assíncronas escalonadas por um runtime leve, permitindo tanto concorrência baseada em threads quanto orientada a eventos.

Go adota um modelo centrado em goroutines, unidades leves de concorrência gerenciadas pela linguagem. A comunicação entre essas unidades ocorre preferencialmente por meio de canais (`chan`), promovendo o compartilhamento seguro de dados por troca de mensagens. Embora o uso de memória compartilhada e sincronização explícita com mutexes seja possível, a abordagem recomendada favorece canais, refletida no lema: "não comunique compartilhando memória; compartilhe memória comunicando".

Elixir segue o modelo de atores, herdado da máquina virtual Erlang. Cada ator é um processo leve e isolado, com sua própria mailbox para recebimento de mensagens. Toda comunicação ocorre por meio de troca assíncrona de mensagens, sem memória compartilhada. O modelo é supervisionado, permitindo que falhas em processos sejam detectadas e tratadas por supervisores, o que favorece a criação de sistemas robustos e tolerantes a falhas.

3.3. Implementações e Modelos

Embora a ideia de ferramentas para programação multithread remeta, em geral, a modelos baseados na exploração do espaço de endereçamento compartilhado em arquiteturas multiprocessadas, diferentes ferramentas podem adotar estratégias variadas de implementação e oferecer modelos de programação distintos. Nesta seção, são discutidas tanto as abordagens de implementação dessas ferramentas quanto os principais modelos de programação multithread que elas proporcionam.

3.3.1. Modelos de Implementação de Ferramentas

A implementação de ferramentas para programação multithread envolve diferentes estratégias de gerenciamento das unidades de concorrência, tarefas ou threads, criadas pelo programador. Essas estratégias estão associadas a decisões quanto ao nível de abstração oferecido, ao grau de envolvimento com o sistema operacional e à política de escalonamento adotada. Em linhas gerais, podem ser identificados dois modelos principais [Kleiman et al. 1996]: o modelo 1×1, no qual cada unidade de concorrência do programa corresponde diretamente a um thread do sistema operacional, e o modelo $N\times M$, no qual as N unidades de concorrência lançadas pelo programa são gerenciadas em espaço de usuário e distribuídas sobre M threads do sistema providas pelo runtime da ferramenta (normalmente, N é muito maior do que M). Até o advento dos multicores, ferramentas de programação também eram implementadas sobre o modelo $N\times1$, em que todas as unidades de concorrência lançadas por um programa eram multiplexadas sobre um único thread. No entanto, com a virtual extinção das plataformas monoprocessadas, esse modelo deixou de ser relevante, e é raramente adotado por ferramentas contemporâneas.

No modelo 1×1, também conhecido como modelo thread sistema, cada unidade de execução lançada pelo programa em execução é diretamente gerenciada pelo sistema operacional. Em termos de nomenclatura, usualmente as ferramentas de programação implementadas sobre este modelo adotam o termo *thread* para designar a unidade que descreve a concorrência no programa. Também é comum que os threads criados sejam

individualmente identificados e esta identificação conhecida como retorno da operação de criação de um thread.

A estrutura de implementação de ferramentas sob o modelo 1×1 não requer a introdução de estratégias de mapeamento dos threads sobre os recursos de processamento disponíveis e a gestão de sua execução, uma vez que a infraestrutura de escalonamento, sincronização e controle de recursos é delegada ao sistema operacional. Ferramentas neste modelo oferecem grande liberdade para construção de algoritmos, uma vez que, de posse do identificador de um thread, seja este representado por uma variável ou um objeto, qualquer ponto do programa pode sincronizar com o thread em execução, tendo este thread sido criado neste ponto, ou não.

Como exemplos de ferramentas 1×1, este texto aborda as linguagens C++ e Rust que são implementadas sob este modelo, ainda que extensões de Rust promovam o uso do modelo N×M discutido a seguir. Outra ferramenta implementada neste modelo é suportada nas principais implementações do padrão POSIX threads (Pthreads) e também Java, desde sua versão 1.0.

O modelo N×M introduz uma camada de abstração que intermedia as unidades de concorrência do programa, usualmente referenciadas como *tarefas* e um conjunto de unidades de execução em nível de sistema operacional (threads sistema) mantido pela ferramenta em um núcleo de execução. Nesse caso, o runtime da linguagem é responsável tanto pelo escalonamento das tarefas sobre os threads do núcleo de execução, delegando ao sistema operacional a responsabilidade do mapeamento dos threads sobre os recursos físicos de processamento. Essa abordagem permite que operações de criação, suspensão e retomada de execução ocorram inteiramente no espaço de usuário, reduzindo significativamente o custo associado a essas operações. Como consequência, torna-se viável lidar com milhares ou até milhões de unidades de concorrência leves. Linguagens como Go e Elixir exemplificam esse modelo, assim como OpenMP e também Java a partir de sua versão 21, como recurso Virtual Threads.

Tabela 3.1: Comparação entre os modelos de implementação 1x1 e NxM

Aspecto	**Modelo 1x1**	**Modelo NxM**
Unidade de Execução	Thread	Tarefa
Custo (comparado)	Maior	Menor
Mapeamento	1 unidade de concorrência → 1 thread do sistema	N unidades → M threads do sistema (com N > M)
Escalonamento	Sistema operacional	Ambiente de execução (runtime)
Criação e troca de contexto	Mais custosas, dependem do sistema operacional	Mais leves, realizadas em espaço de usuário
Escalabilidade	Limitada por overhead do sistema e número de threads	Alta, viabiliza milhares ou milhões de unidades leves
Integração com ferramentas do sistema	Total (visibilidade de threads nativos)	Parcial ou ausente (threads não visíveis ao sistema)
Exemplos de linguagens	Pthreads, C++, Rust	OpenMP, Go, Elixir

Uma vez que a criação de tarefas no modelo N×M tem menor custo que a criação de threads no modelo 1×1, é usual que ferramentas de programação implementadas neste modelo sejam a opção para aplicações que necessitem expor um grande número de atividades concorrentes. A Tabela 3.1 apresenta uma comparação entre estes dois modelos.

3.3.2. Modelos para Programação Multithread

Independente do modelo de implementação adotado pela ferramenta de programação, estas oferecem diferentes modelos de programação. Esses modelos podem ser classificados em duas grandes categorias [Rauber and Rünger 2010]: compartilhamento de memória e troca de mensagens.

Embora seja comum que ferramentas de programação forneçam recursos de programação a ambos os modelos, memória compartilhada e troca de mensagens, as vocações das ferramentas se posicionam, majoritariamente, sobre um ou outro. Como atrativo das ferramentas baseadas em memória compartilhada, destaca-se a simplicidade da comunicação por meio de escritas e leituras em um espaço de endereçamento comum. Esse modelo, no entanto, exige que o programador gerencie a sincronização entre os fluxos de execução concorrentes. Já a troca de mensagens incorpora sincronização implícita entre as partes comunicantes, ao custo de gerenciar o endereçamento e a entrega das mensagens.

Modelos baseados em compartilhamento de memória são os mais tradicionais e consistem na execução concorrente de múltiplas threads que acessam um espaço de memória comum. Um exemplo típico é o modelo *thread* + *lock*, adotado por bibliotecas como POSIX Threads, Java Threads, `std::thread` em C++ e as primitivas de `std::thread` e `Mutex` em Rust. Nessa abordagem, a sincronização entre as unidades concorrentes é feita explicitamente por meio de mecanismos como mutexes, semáforos e variáveis de condição. Embora ofereça grande flexibilidade, esse modelo exige cuidado redobrado do programador para evitar condições de corrida, impasses e outros erros difíceis de depurar.

Também compõe parte dos modelos baseados em compartilhamento de memória o paralelismo de dados, no qual múltiplas threads executam o mesmo conjunto de operações sobre diferentes partes de uma estrutura de dados. Nesse paradigma, o foco não está na divisão do programa em tarefas distintas, mas na divisão dos dados a serem processados, permitindo que cada thread opere sobre uma fatia independente do conjunto de entrada. Essa abordagem é especialmente eficiente em aplicações de natureza numérica ou científica, como simulações físicas, álgebra linear e processamento de imagens, quando grandes volumes de dados homogêneos devem ser manipulados de forma iterativa.

Linguagens e bibliotecas que oferecem suporte ao paralelismo de dados geralmente fornecem construções que abstraem o particionamento dos dados e a criação das threads. Em C++, a biblioteca `Parallel STL`, introduzida no C++17, permite aplicar algoritmos padrão, como `std::sort` ou `std::transform`, com execução paralela utilizando a política `std::execution::par`. Em Java, a API `Stream` com `.parallel()` permite aplicar transformações a coleções de forma concorrente, com a divisão automática dos dados e execução em múltiplos núcleos. No contexto do OpenMP, o paralelismo de dados é frequentemente expresso com a diretiva `#pragma omp for`, que distribui as iterações de um laço entre múltiplos threads, cada um responsável por

uma fatia do intervalo de índices.

Essa forma de paralelismo tende a ser mais previsível e escalável do que modelos centrados na coordenação entre tarefas, já que minimiza a necessidade de sincronização entre as threads. No entanto, o paralelismo de dados pressupõe que as operações sobre diferentes porções do conjunto sejam independentes entre si, o que nem sempre é possível ou trivial de garantir. Uma variação desse paradigma é o modelo de regiões paralelas ou paralelismo estruturado, em que o programador especifica blocos de código a serem executados em paralelo, delegando à ferramenta a responsabilidade por criar, distribuir e sincronizar as threads. OpenMP é o principal exemplo dessa abordagem, permitindo que diretivas de compilação definam paralelismo sobre laços, seções ou tarefas sem necessidade de gerenciamento explícito de threads.

Como alternativa dentre os modelos de programação baseados em memória compartilhada, encontra-se o paralelismo de tarefas. Nessa estratégia, a ênfase recai na decomposição lógica do trabalho em tarefas independentes, sem que o programador precise controlar as threads que as executam. Ferramentas como `std::async` em C++, o módulo `Task` em Elixir, o suporte a tarefas em OpenMP e os executores em Java com `CompletableFuture` ou `ExecutorService` exemplificam essa abordagem.

Modelos de programação concorrente baseados em troca de mensagens também são largamente explorados em arquiteturas multiprocessadas. Neste modelo, apesar de fluxos de execução compartilharem um espaço de endereçamento comum, é evitado o compartilhamento de dados em memória, sendo privilegiada a comunicação explícita entre unidades de execução autônomas. O modelo de atores é uma das formas mais conhecidas, no qual cada ator encapsula seu próprio estado e responde a mensagens assíncronas. Esse paradigma é adotado pela máquina virtual Erlang, utilizada por Elixir, onde processos leves reagem a mensagens sem acesso compartilhado à memória.

Outro modelo de troca de mensagens relevante é o de processos sequenciais comunicantes (CSP), que fundamenta a programação em canais de comunicação entre unidades concorrentes. Go adota esse modelo por meio da combinação de *goroutines* e *channels*, permitindo que processos concorrentes interajam de forma sincronizada e segura por meio de trocas explícitas de mensagens.

3.3.2.1. Abordagens Funcionais

Além dos modelos tradicionais de programação multithread baseados em compartilhamento de memória ou troca de mensagens, algumas ferramentas contemporâneas oferecem suporte a abordagens funcionais [Thomas 2022]. Essas abordagens propõem uma mudança de paradigma, eliminando a necessidade de compartilhamento de estado entre unidades de execução, favorecendo o isolamento e a imutabilidade dos dados.

Um exemplo é a incorporação de construtores nas linguagens para os padrões paralelos de *pipelines* e *streams*, nos quais transformações são aplicadas de maneira encadeada a uma sequência de dados. Em vez de explicitar a divisão de trabalho entre múltiplas unidades de execução, o programador compõe funções puras, ou seja, de funções que operem apenas sobre os parâmetros recebidos e retornem exclusiamente o resultado da

computação sobre estes parâmetros. Estas funções operam sobre elementos de um espaço de dados de forma independente, permitindo a exploração do paralelismo. A `Stream API` de Java oferece essa abordagem. Com o uso do modificador `.parallel()`, pipelines declarativos como `stream.map(...).filter(...).collect(...)` podem ser automaticamente paralelizados pela plataforma.

De forma análoga, linguagens funcionais como Elixir promovem a composição de pipelines com o operadores `|>`, em que listas ou fluxos de dados são transformados por meio de funções puras. Bibliotecas como `Flow` permitem que esses pipelines sejam executados de forma concorrente e distribuída, utilizando internamente múltiplos processos leves (baseados no modelo de atores). Em cenários de processamento de eventos em larga escala, frameworks como `Broadway` oferecem uma abstração de fluxo concorrente que combina particionamento, paralelismo e tolerância a falhas de maneira declarativa.

Linguagens como C++ e Rust também têm incorporado elementos desse paradigma. No C++17, a `Parallel STL` permite aplicar algoritmos sobre coleções com políticas como `std::execution::par`, sem exigir controle direto de threads. Em Rust, bibliotecas como `rayon` oferecem iteradores paralelos que preservam a semântica funcional e facilitam a paralelização segura sobre coleções.

3.3.3. Foco nas Linguagens Analisadas

Nesta seção foram apresentados os principais modelos empregados na implementação de programas multithread, com foco nas abordagens oferecidas por C++, Rust, Go e Elixir. A Tabela 3.2 resume como essas linguagens estruturam suas abstrações para os principais aspectos da programação concorrente. Ainda que todas ofereçam meios para adotar diferentes modelos de concorrência, cada uma revela uma vocação predominante, delineada por decisões de projeto que influenciam diretamente o estilo de programação incentivado. As seções seguintes, dedicadas ao estudo individual dessas linguagens, têm por objetivo destacar essas vocações e discuti-las em maior profundidade.

Tabela 3.2: Modelos de concorrência nas linguagens abordadas.

Aspecto	C++	Rust	Go	Elixir
Unidade	Thread nativa	Thread nativa / Tarefa assíncrona	Goroutine	Processo leve (actor)
Comunicação	Memória compartilhada	Memória compartilhada (com propriedade e empréstimo)	Troca de mensagens e memória compartilhada	Troca de mensagens
Mecanismo	`std::thread`	`std::thread` / `thread::spawn`	`go` (goroutines)	`spawn` (processos leves)
Mapeamento	Manual	Automática (via tarefa)	Automática	Supervisionada
Sincronização	Seções críticas (`mutexes`, `locks`)	Exclusão mútua + garantias de compilação (propriedade e empréstimo)	Canais; mutexes opcionais	Mailbox com isolamento

Complementando a sumarização da presente seção, a Tabela 3.3 apresenta alguns dos principais recursos disponibilizados pelas ferramentas estudadas para a implementação dos diferentes modelos de concorrência discutidos ao longo do texto. Embora não exaustiva, a tabela oferece um panorama representativo das possibilidades técnicas e estilísticas favorecidas por cada linguagem. Estes recursos estão entre aqueles discutidos para caracterizar cada ferramenta.

Tabela 3.3: Abordagens para controle da concorrência em ferramentas de programação multithread.

Abordagem	Ferramenta	Forma de Implementação
Seções críticas	C++	`std::thread`, `std::mutex`, `std::lock_guard`, `std::shared_mutex`
	Rust	`thread::spawn`, `std::sync::Mutex`, `std::sync::RwLock`, `std::sync::Arc`
	Go	Pacote `sync` com `Mutex`, `WaitGroup`, `RWMutex`
Paralelismo de Tarefas	C++	`std::async` com `std::future`
	Rust	`thread::spawn` com `JoinHandle`; uso de `move` para paralelismo leve
Modelo de Atores	Elixir	Cada processo leve atua como ator. Uso de primitivas `send`/`receive`, e módulos como `GenServer`
	Elixir	Supervisores organizam os processos em árvores de supervisão com tolerância a falhas
Troca de Mensagens	Go	Goroutines com canais (`chan`), síncronos ou com buffer
	Rust	Biblioteca `std::sync::mpsc` e alternativa `crossbeam::channel`
Paralelismo de Dados	Rust	Biblioteca `rayon` com `par_iter()`
Funcional	Elixir	Imutabilidade; bibliotecas `Flow` (pipelines concorrentes) e `Broadway` (fluxos concorrentes tolerantes a falhas)

3.4. C++

C++[1] [Williams 2019] é uma linguagem criada na década de 1980 e passou por sucessivas revisões que ampliaram significativamente suas capacidades expressivas e operacionais. No contexto da programação multithread, o padrão C++11 trouxe a introdução da classe `std::thread`, permitindo a criação e manipulação de threads nativas de forma mais simples e com um ciclo de vida bem definido. Além disso, C++11 também trouxe recursos como mecanismos de sincronização e operações atômicas. Em versões subsequentes, como C++14, melhorias incrementais foram feitas no suporte a paralelismo, enquanto o C++17 introduziu o conceito de algoritmos paralelos integrados à STL. Nesse novo contexto, as operações da STL passaram a permitir a paralelização direta por meio de políticas de execução oferecendo recursos para programação paralela sem a necessidade de um controle explícito sobre os threads. O C++20, por sua vez, expandiu ainda mais o suporte a paralelismo e sincronização, trazendo novas primitivas como barreiras e travas de contagem, além de aprimorar ainda mais a exploração de paralelismo na biblioteca

[1] `https://isocpp.org`

padrão.

3.4.1. Modelo Básico

Em C++, o suporte nativo a threads é realizado por meio da classe `std::thread`. A criação de um thread ocorre pela construção de um objeto dessa classe, ao qual deve ser associado um objeto invocável, como uma função, um functor ou uma expressão lambda. Assim que o objeto `std::thread` é instanciado, o thread associado inicia sua execução imediatamente, sendo o objeto invocável executado sobre este thread. Esse modelo permite integrar o gerenciamento de threads à programação orientada a objetos da linguagem, conferindo ao thread o comportamento de um objeto comum, com ciclo de vida e escopo bem definidos. O Código 1 ilustra a criação com as diferentes formas de objetos invocáveis. Neste código são criados três threads, `t1`, `t2` e `t3`, cada thread sendo criado com um tipo de objeto invocável diferente: uma função, um functor (representado pelo método `operator()` na classe Functor, e uma expressão lambda.

O controle do thread criado se dá por meio do próprio objeto `std::thread`. É responsabilidade do programador finalizar explicitamente o thread por meio de uma chamada ao método `join()`, que faz com que o fluxo chamador aguarde o término do thread, assegurando sua conclusão antes da continuação do programa. Alternativamente, pode-se utilizar o método `detach()`, que dissocia o objeto do thread em execução, permitindo que ele continue sua programação de forma independente, sendo sua finalização gerenciada pelo sistema operacional. Caso nenhum dos dois métodos seja chamado antes da destruição do objeto `std::thread`, o programa será abruptamente encerrado com uma chamada a `std::terminate()`.

Os threads criados em C++ podem ser considerados *nomeados*, uma vez que o objeto que os encapsula possui um identificador explícito no escopo da função onde é criado; assim, todos os threads são identificados individualmente. Além disso, quando o objeto `std::thread` está em estado *joinable*, ou seja, não foi *detached*, ele pode ser submetido a uma e apenas uma sincronização por chamada a join(). Após essa chamada, o thread é considerado finalizado, e o objeto não é mais *joinable*.

O uso de `std::thread` segue o princípio de RAII (Resource Acquisition Is Initialization), no qual a alocação e liberação de recursos são associadas ao ciclo de vida de objetos. Nesse modelo, um thread é criado na construção do objeto `std::thread` e deve ser adequadamente finalizado antes que esse objeto seja destruído. Caso contrário, o programa será encerrado de forma abrupta. O RAII reforça a ideia de que a responsabilidade pela aquisição e liberação dos recursos do thread é do próprio objeto, permitindo que o controle sobre o ciclo de vida do thread seja exercido de forma segura e previsível. Isso reduz a propensão a erros como vazamento de recursos, como memória e mecanismos de sincronização, evitando panes no programa, condições de corrida ou terminação inesperada, e favorecendo uma estruturação mais clara do programa concorrente.

3.4.2. Sincronizações e Compartilhamento de Dados

Em C++, toda comunicação entre threads se dá por meio do compartilhamento de dados na memória. Isso implica que os escopos de vida das variáveis compartilhadas devem estar ativos enquanto as threads acessam tais dados. Como consequência, múltiplas refe-

```
void foo() {
  std::cout << "Executando uma função foo." << std::endl;
}
class Functor {
public:
  void operator()() {
    std::cout << "Executando um functor." << std::endl;
  }
};
int main() {
  Functor f;
  std::thread t1(foo);
  std::thread t2(f);
  std::thread t3([]() {
       std::cout << "Executando uma função lambda." << std::endl;
  });
  // ...
  t1.join();
  t2.join();
  t3.join();
  return 0;
}
```

Código 1: Modelo básico de criação de threads em C++

rências simultâneas a uma mesma região de memória podem ocorrer, exigindo mecanismos de controle de acesso para garantir a integridade dos dados e evitar comportamentos indeterminados.

O controle eficiente de acesso a dados compartilhados pode ser realizado por meio de operações atômicas, disponibilizadas pela na classe `std::atomic`. Esta classe oferece operações realizadas diretamente sobre os recursos de hardware para operações atômicas, garantindo que uma modificação em uma variável ocorra de forma indivisível. Isso impede que outras threads interfiram no meio de uma operação, assegurando a consistência dos dados. No entanto, as operações atômicas são aplicáveis a dados de uma única palavra de memória, como um inteiro, o que as tornam adequadas principalmente para contadores, flags e indicadores de estado. As variáveis atômicas suportam operações como incremento, troca e comparação-condicional (`compare_exchange_strong`), sendo muito eficientes, pois não envolvem bloqueios. Um exemplo de uso da atomicidade em C++ pode ser ilustrada por:

```
std::atomic<int> valor(10);
int esperado = 10, novo_valor = 20;
```

```
if (valor.compare_exchange_strong(esperado, novo_valor))
    std::cout << "Sucesso, valor = " << valor.load() << std::endl;
else
    std::cout << "Falha, valor = " << valor.load()
              << ". Esperado valor: " << esperado << std::endl;
```

Neste exemplo de uso de `atomic`, a variável `valor` é protegida contra condições de corrida por meio do uso da função atômica `compare_exchange_strong`, que realiza uma substituição condicional: ela apenas altera o valor caso ele ainda seja igual ao esperado. Essa abordagem é comum em algoritmos *lock-free*, em que o controle de concorrência é feito sem o uso de bloqueios explícitos. A classe `std::atomic` é uma classe `template`, cujo tipo de dado é especificado entre os sinais de menor e maior, como em `std::atomic<int>`. O tipo fornecido deve ser um tipo trivial de dados, isto é, tipos escalares como `int`, `bool`, `char`, ponteiros (`T*`) ou tipos enumerados. A biblioteca também oferece especializações para ponteiros e suporte adicional por meio da classe `std::atomic_flag`, útil para operações ainda mais básicas de sinalização.

Para seções críticas com construções mais elaboradas, envolvendo um número maior de operações ou estruturas de dados mais complexas do que as suportadas pelos templates ao `std::atomic`, é indicado o uso de `std::mutex`. Este mecanismo de sincronização permite ao programador delimitar seções de código, ditas seções críticas, a serem executadas em regime de exclusão mútua com outras seções críticas sincronizadas pelo mesmo mutex. Um mutex deve ser explicitamente adquirido, com uma chamada ao método `lock()`, antes do acesso a dados compartilhados. Após o uso deve ser explicitamente liberado, pela invocação da operação `unlock()`. A omissão do desbloqueio, por falha lógica ou desvio de controle (como exceções), pode comprometer toda a aplicação. Para evitar esse risco, é recomendável o uso de mecanismos de guarda. Em comparação com variáveis atômicas, o uso de mutex não é tão eficiente, embora ofereça maior flexibilidade e permita seções críticas complexas, envolvendo múltiplas variáveis.

O `std::lock_guard` é um objeto RAII que adquire o mutex na construção e o libera automaticamente na destruição. Sua simplicidade o torna ideal para escopos curtos e bem definidos. Já o `std::unique_lock` oferece maior flexibilidade, permitindo adiar a aquisição, liberar e reacquirir o mutex manualmente, ou combiná-lo com variáveis de condição. Ambos os mecanismos evitam vazamentos de locks mesmo em situações de exceção.

Em situações em que múltiplos mutexes precisam ser adquiridos simultaneamente, a chamada direta à operação `lock()` sobre objetos mutex em ordem arbitrária pode levar a deadlocks, especialmente se threads distintas tentarem adquirir os mesmos mutexes em ordens diferentes. Para tratar esse problema, o C++ oferece a função `std::lock`, que realiza a aquisição conjunta de dois ou mais mutexes de forma atomizada e livre de deadlocks.

3.4.3. Programação Assíncrona

A biblioteca padrão de C++, a partir de C++11, sua primeira versão com suporte à concorrência, oferece mecanismos de programação assíncrona com suporte nativo à execução de

tarefas em segundo plano. O principal recurso para esse fim é a função `std::async`, que recebe um *objeto invocável* e retorna imediatamente um objeto `std::future`. Esse objeto retornado representa o resultado eventual da computação e pode ser usado posteriormente para sincronização e acesso ao valor retornado. Um exemplo da criação assíncrona de threads, e uso de variáveis futuras, é apresentado no Código 2.

A abordagem baseada em `std::async` abstrai o controle explícito de threads, permitindo que a lógica da aplicação foque na decomposição do problema em tarefas independentes, sem lidar diretamente com criação, junção ou gerenciamento de threads. A execução da função fornecida pode ocorrer de forma imediata ou adiada, dependendo da política de lançamento especificada.

A política `std::launch::async` força o início imediato da tarefa em uma thread separada, enquanto `std::launch::deferred` adia sua execução até que o resultado seja requisitado, por meio de `future::get`. Quando nenhuma política é especificada, a decisão fica a critério da implementação. A chamada a `get()` bloqueia a thread chamadora até que o resultado esteja disponível, realizando, portanto, uma sincronização implícita.

```cpp
int resposta() {
  return 42;
}
int main() {
  std::future<int> f = std::async(std::launch::async, resposta);
  // ...
  int resultado = f.get(); // bloqueia até computacao() terminar
  std::cout << "Resultado: " << resultado << "\n";
  return 0;
}
```

Código 2: Criação assíncrona de threads em C++

Essa forma de programação é útil para tarefas computacionalmente intensivas ou operações de I/O que podem ser realizadas em paralelo com a lógica principal. No entanto, o uso de `std::async` deve ser feito com atenção ao modelo de execução adotado pela implementação, especialmente em relação ao número de threads disponíveis e à política de escalonamento do sistema subjacente. O padrão C++ não especifica se tarefas iniciadas com `std::async` usam threads dedicados ou um thread pool. O padrão define que com `std::launch::async`, a execução deve ser assíncrona, mas a implementação é livre para criar uma nova thread ou reutilizar uma de um pool. Já com `std::launch::deferred`, a execução ocorre no próprio thread chamador quando `get()` ou `wait()` é invocado. Na prática, as implementações variam: libstdc++ (GCC) e libc++ (Clang) criam uma nova thread para cada tarefa iniciada com `std::launch::async`, sem utilizar thread pools. Em contraste, o MSVC (Visual

Studio) adota um thread pool interno, o que reduz o overhead associado à criação de threads e permite um gerenciamento mais eficiente das tarefas. REFERENCIA

3.4.4. Algoritmos Paralelos

A partir do C++17, a STL passou a contar com suporte à execução paralela de seus algoritmos por meio de políticas que indicam se a operação deve ocorrer de forma sequencial, paralela ou paralela com vetorização. Ainda que essa extensão promova o uso de paralelismo de forma declarativa, ela exime o programador da necessidade de gerenciar diretamente threads ou empregar mecanismos explícitos de sincronização.

A principal interface para esse modelo é fornecida pelas versões sobrecarregadas dos algoritmos padrão da STL, que aceitam uma política de execução como primeiro argumento: `std::execution::seq` e `std::execution::par`. A política `seq` indica que o algoritmo deve ser executado de forma sequencial, sendo equivalente, portanto, ao comportamento anterior ao C++17. A política `par` habilita execução paralela das operações, com cada invocação podendo ocorrer em threads distintos. Existe também a política `std::execution::par_unseq`. Esta política combina paralelismo oferecido com a política `par` com a exploração do potencial de execução vetorial do processador, permitindo maior grau de exploração do paralelismo de dados. O trecho a seguir exemplifica os algoritmos paralelos, aplicando uma função lambda para atualizar cada elemento de um vetor com o valor de seu quadrado:

```
std::for_each(std::execution::par, v.begin(), v.end(), [](int& x) {
    x *= x;
});
```

O uso de esqueletos para descrição da concorrência permite delegar ao compilador e à biblioteca padrão a escolha das estratégias de mapeamento da concorrência gerada pelo programa no paralelismo disponibilizado pelo hardware, melhorando a expressividade do código e facilitando a manutenção. Deve ser observado mais uma vez que o padrão não especifica como os threads devem ser criados ou gerenciados para viabilizar a execução paralela dos algoritmos. Cabe à implementação da STL decidir se utilizará criação direta de threads, um banco de threads reutilizáveis (thread pool) ou se delegará essa responsabilidade a uma biblioteca paralela subjacente, como Intel TBB ou Microsoft PPL. Na prática, a maioria das implementações modernas opta pelo uso de thread pools, tanto por eficiência quanto por escalabilidade, evitando o custo associado à criação e destruição frequente de threads. Na stdlibc++, a estratégia adotada é de thread pool, sendo que o número de threads no pool é igual ao número de *cores* (virtuais) disponívies.

Embora a abordagem dos algoritmos paralelos da STL seja de aplicação simples, deve ser observado que não é garantida a ordem de execução entre as iterações. Isso inviabiliza o uso de lógicas cuja execução de diferentes passos da iteração devam respeitar uma sequência específica. Por esse motivo, funções puramente funcionais ou que operem de forma independente sobre cada elemento da faixa são as mais indicadas nesse contexto.

3.4.5. Barreiras e Travamentos

A versão C++20, além de promover aprimoramentos incrementais na interface de `std::atomic`, introduziu novos mecanismos de sincronização, como as barreiras (`std::barrier`) e os travamentos (`std::latch`). Essas novas ferramentas visam oferecer um controle mais preciso sobre o fluxo de execução de threads em cenários de paralelismo.

A `std::barrier` permite que um conjunto de threads se sincronize em um ponto específico de execução antes de continuar, garantindo que todas as threads atinjam um marco determinado antes de prosseguir. Esse mecanismo é particularmente útil em situações que envolvem fases de processamento que precisam ser concluídas de forma conjunta por todas as threads, como em pipelines de dados ou etapas de computação paralela.

Já o `std::latch` oferece uma forma de sincronizar threads até que um número pré-determinado de threads tenha alcançado uma etapa específica de execução. A diferença fundamental entre `latch` e `barrier` é que, uma vez que o valor de contagem do latch é alcançado, ele não pode ser reutilizado, tornando-o adequado para cenários em que a sincronização precisa ocorrer apenas uma vez. Isso é útil em casos onde se deseja garantir que um conjunto de threads complete uma tarefa antes de permitir que o programa continue, sem necessidade de reuso da mesma barreira.

3.4.6. Mais sobre sincronização

O C++ oferece ainda outros mecanismos de sincronização além do tradicionais `std::mutex` e das de `std::atomic`. Um destes recursos é o `std::shared_mutex`, que oferece uma semântica de leituras concorrentes e escrita exclusiva a dados compartilhados. Seu uso permite que múltiplas threads adquiram acesso de leitura simultaneamente, enquanto garante exclusividade para acessos de escrita. Ele é útil em situações onde operações de leitura são muito mais frequentes do que as de escrita. Para usá-lo, empregam-se as classes `std::shared_lock` (para leitura) e `std::unique_lock` (para escrita). Um exemplo simples de uso:

```
std::shared_mutex smutex;
void leitor() {
  std::shared_lock lock(smutex);
  ... // acesso somente leitura
}
void escritor() {
  std::unique_lock lock(smutex);
  ... // acesso de escrita
}
```

Outro recurso que se apresenta como uma alternativa ao uso individualizado das operações `lock()` sobre mutex é a classe `std::scoped_lock`, que permite travar

múltiplos mutexes simultaneamente em uma única chamada, evitando deadlocks por aquisição desordenada. Ele é particularmente útil quando diversas seções críticas devem ser protegidas de forma coordenada:

```
std::mutex a, b;
void funcao() {
  std::scoped_lock lock(a, b);
  ... // código protegido
}
```

Outro mecanismo importante para coordenação entre threads é a `std::condition_variable`, que permite a uma thread esperar por uma determinada condição antes de prosseguir. Ela é normalmente usada em conjunto com `std::unique_lockstd::mutex` e permite que threads sejam notificados por meio das funções `notify_one()` ou `notify_all()`. Essa abordagem é útil, por exemplo, para implementar filas de trabalho, buffers compartilhados ou qualquer estrutura que dependa de condições de disponibilidade de dados ou recursos. A verificação da condição deve sempre ser feita dentro de um laço, pois sinais espúrios podem ocorrer. O exemplo a seguir ilustra uma situação simples em que um thread produtor acorda uma thread consumidor após alterar uma variável de controle:

```
std::mutex m;
std::condition_variable cv;
bool pronto = false;
void produtor() {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  {
   std::unique_lock lock(m);
   pronto = true;
  }
  cv.notify_one();
}
void consumidor() {
  std::unique_lock lock(m);
  cv.wait(lock, [] { return pronto; });
  std::cout << "Consumidor acordado.\n";
}
```

Neste exemplo, a variável `pronto` controla quando o consumidor pode prosseguir. A `std::condition_variable` depende de um `std::mutex` para coordenar o acesso à variável compartilhada e garantir consistência. O método `wait()` libera temporariamente o mutex enquanto a thread está bloqueada, e o readquire automaticamente quando a notificação ocorre e a condição é satisfeita.

```cpp
void tarefa(std::stop_token st, int i) {
  while (!st.stop_requested()) {
    std::cout << "Trabalhando...\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
  }
  std::cout << "Thread " << i << " cancelada com segurança.\n";
}
int main() {
  std::jthread t1(tarefa,1);
  std::jthread t2(tarefa,2);
  std::this_thread::sleep_for(std::chrono::seconds(2));
  t1.request_stop();
  t2.request_stop();
  return 0;
}
```

Código 3: Cancelamento cooperativo seguro.

Por fim, o C++ disponibiliza a `std::condition_variable_any`, uma versão mais flexível da `std::condition_variable`. Enquanto esta última funciona exclusivamente com `std::unique_lockstd::mutex`, a variante `any` pode ser usada com qualquer tipo de objeto de sincronização que implemente os métodos `lock()` e `unlock()`, como `std::shared_mutex` ou mutexes personalizados. Isso permite, por exemplo, empregar variáveis de condição em cenários que envolvam acesso concorrente para leitura, algo inviável com a versão tradicional. Internamente, a semântica de uso é semelhante, envolvendo uma verificação de condição dentro de um laço e a liberação automática do bloqueio durante o tempo de espera. No entanto, o programador deve garantir que o tipo de trava utilizada ofereça o comportamento correto de exclusão mútua ou compartilhada, conforme o caso.

3.4.7. Cancelamento Cooperativo

O padrão C++20 introduziu a classe `std::jthread`, que representa uma evolução segura da tradicional `std::thread`. A principal vantagem deste recurso é que sua estrutura interna gerencia automaticamente a finalização da thread, realizando uma chamada a `join()` em seu destrutor, eliminando a necessidade de fazê-lo manualmente. Essa implementação evita encerramentos abruptos por destruição de threads ainda ativos, reduzindo o risco de erros comuns associados à gestão manual do seu ciclo de vidas.

`std::jthread` introduz ainda suporte embutido ao cancelamento cooperativo, por meio de travas de contagem implementadas por mecanismos `std::stop_token` e `std::stop_source`. O cancelamento é iniciado pelo emissor de um `stop_source`, e as threads associadas recebem um `stop_token` que pode ser consultado periodicamente para verificar se a execução deve ser encerrada.

O Código 3 ilustra o cancelamento coletivo com uso de travas de contagem. No exemplo, cada thread recebe um `std::stop_token` como primeiro argumento da função executada. Esse token permite à função detectar, de forma não intrusiva, se foi solicitado seu encerramento. No corpo da função `tarefa`, a verificação é feita com `st.stop_requested()`, que retorna `true` quando o pedido de cancelamento é emitido. A verificação ocorre dentro de um laço, permitindo que a thread finalize sua execução de maneira controlada assim que o sinal de parada for detectado.

A chamada `request_stop()` realizada no `main` aciona o mecanismo de cancelamento, comunicando a intenção de finalização aos threads. Como o cancelamento é cooperativo, cabe à função em execução respeitar o pedido e interromper seu fluxo. A estrutura do `std::jthread` garante, por meio de seu destrutor, que o método `join()` seja invocado automaticamente, assegurando que o programa principal espere o término das threads antes de prosseguir.

3.4.8. RAII e Gerenciamento de Recursos

Em alguns pontos desta seção foi mencionado o termo RAII (Resource Acquisition Is Initialization). RAII é um padrão de design em C++ que vincula a alocação e liberação de recursos ao ciclo de vida de objetos. Quando um objeto RAII é criado, ele adquire um recurso no seu construtor, como memória, arquivos ou mecanismos de sincronização e mesmo a criação de um thread. Ao sair de escopo, o destrutor do objeto é chamado, liberando o recurso de forma automática. Esse comportamento elimina a necessidade de gerenciamento manual de recursos, minimizando o risco de erros como vazamentos de memória ou falhas na liberação de recursos, especialmente em cenários com exceções ou retornos antecipados de funções.

Na programação multithread em C++, o RAII é amplamente utilizado para garantir o gerenciamento correto de threads e sincronização de dados. A classe `std::thread` segue o padrão RAII, pois o thread é iniciado na construção do objeto e precisa ser explicitamente finalizado com `join()` ou `detach()` antes da destruição do objeto. Se o thread não for finalizado corretamente, o programa chamará `std::terminate()`, forçando o término do programa. O uso de mutexes também é gerido por meio de RAII, como demonstrado pelo `std::lock_guard`, que adquire um mutex no momento da criação do objeto e o libera automaticamente quando o objeto sai de escopo. Além desses casos, o RAII pode ser empregado no gerenciamento de recursos temporários em operações assíncronas, onde objetos como `std::unique_lock` e `std::shared_lock` permitem maior flexibilidade na sincronização de múltiplos threads, assegurando a correta liberação de recursos mesmo em cenários complexos de concorrência.

3.5. Rust

Rust[2] [Team 2021][Troutwine 2018] teve sua primeira versão estável em 2015. Desde então tem se destacado como uma linguagem moderna voltada ao desenvolvimento de sistemas seguros e eficientes. A linguagem combina controle de baixo nível com garantias de segurança em tempo de compilação, utilizando o sistema de ownership e as regras de empréstimo para evitar condições de corrida e acessos inválidos a dados. As abstra-

[2]https://www.rust-lang.org

ções oferecidas por sua biblioteca padrão incluem criação e gerenciamento de threads, estruturas de sincronização, como mutexes e tipos atômicos, além de canais para troca segura de mensagens entre threads. Para aplicações que demandam alta escalabilidade, Rust também oferece suporte a programação assíncrona baseada em futures, com sintaxe `async/await` e integração com runtimes como Tokio. O ecossistema da linguagem é fortemente apoiado pelo Cargo, uma ferramenta que gerencia pacotes, dependências e builds de forma integrada, facilitando a adoção de bibliotecas especializadas para concorrência e paralelismo.

3.5.1. Modelo Básico

Rust oferece um modelo de concorrência construído a partir de três mecanismos principais: threads nativos, tarefas assíncronas e canais de comunicação. Cada um desses pilares é integrado à linguagem de forma a respeitar o sistema de ownership e garantir segurança em tempo de compilação.

3.5.1.1. Threads Nativos

A criação de um thread nativo ocorre com a chamada à função `thread::spawn`, que requer como parâmetro um fechamento (closure). O thread é criado para executar esta closure. A captura de variáveis é explicitada pelo uso da palavra-chave `move`, resultando na transferência da posse dos dados instanciados no escopo envolvente ao lançamento do thread para o escopo do novo thread. Este mecanismo de propriedade inviabiliza o compartilhamento de dados, impedindo assim referências inválidas e compartilhamento inseguro. Observe que apenas as variáveis efetivamente utilizadas no closure serão movidos para o novo thread, os demais dados do escopo envolvente não são afetados. O Código 4 mostra exemplos básicos de criação de threads com closures, incluindo o uso de `join` para aguardar sua conclusão.

```
use std::thread;
fn main() {
  let x = 8;
  let handle = thread::spawn(move || x * x);
  let r = handle.join().unwrap();
  println!("{}", r);
}
```

Código 4: Modelo básico de criação de threads em Rust

O objeto retornado por `thread::spawn` representa a handle do thread criado e permite sincronização explícita com a chamada ao método `join`. Os threads são, assim, identificados individualmente. Caso `join` não seja chamado, o thread ainda será executado normalmente, mas não há garantia de que será concluído antes do término da função

principal, podendo levar à finalização prematura do programa. Assim, todo thread criado deverá ser submetido a uma, e no máximo a uma, operação de sincronização por `join`, não sendo possível ativar um mecanismo de `detach`, como em C++. No exemplo, o thread retorna um valor, e a chamada a `join()` tem como resultado um `Result<T>`, onde `T` é o tipo do valor retornado pela closure do thread. O método `.unwrap()` é então chamado nesse Result. O `unwrap()` é uma forma de lidar com o possível erro que pode ocorrer durante a execução da thread. Se o thread completar sua execução com sucesso e retornar um valor, o `unwrap()` extrairá esse valor do Result. Por outro lado, se o thread entrar em pânico (devido a um erro irrecuperável), a chamada a `unwrap()` fará com que o thread principal também entre em pânico e o programa seja finalizado.

3.5.1.2. Programação Assíncrona

O suporte à programação assíncrona é realizado por meio dos construtores `async` e `await`. Esse suporte é implementado por meio da transformação automática de funções assíncronas em estruturas que implementam a *trait* `Future`. No entanto, a biblioteca padrão não fornece um mecanismo de escalonamento para essas `futures`. As opções recaem em desenvolver esta lógica de agendamento no próprio programa ou usar bibliotecas, ou *crates*, na terminologia Rust, consolidadas que proveem estes serviço. Dentre as opções, Tokio é bastante popular.

No modelo oferecido pelo Tokio, cada tarefa assíncrona é representada por um valor que implementa a *trait*, algo como interface no contexto Rust, `Future`. Esse valor encapsula o estado da computação e pode ser agendado em um runtime baseado em múltiplos threads, que executa as tarefas de forma cooperativa. O programador declara uma tarefa com a palavra-chave `async`, e seu corpo só é executado quando o `Future` resultante for submetido ao runtime por meio do método `spawn` ou `spawn_blocking`.

A macro `tokio::main` transforma a função principal do programa em uma tarefa assíncrona e inicializa automaticamente o runtime. A seguir, o Código 5 ilustra a criação de tarefas concorrentes com Tokio.

```
use tokio::task;
#[tokio::main]
async fn main() {
 let t = task::spawn(async { println!("Executando uma tarefa"); });
 t.await.unwrap();
}
```

Código 5: Criação de tarefas assíncronas com Tokio

A função `task::spawn` inicia imediatamente a execução da tarefa assíncrona no pool de threads mantido pelo runtime. O método `await` sobre o valor retornado permite aguardar a conclusão da tarefa, de forma semelhante à chamada `join` usada

com threads do sistema operacional. No entanto, ao contrário dos threads nativos, o uso de `await` não bloqueia a thread atual, mas apenas suspende a tarefa em execução, permitindo o avanço de outras tarefas cooperativas no mesmo runtime.

No código exemplo, também é observado que a função `main` também é assíncrona. Isso é necessário, por, em Tokio, somente uma tarefa assíncrona pode invocar `await` sobre outra tarefa. A macro `#[tokio::main]` providencia a execução assíncrona da função main.

3.5.1.3. Canais de Comunicação

Embora Rust ofereça diferentes mecanismos para compartilhamento de dados entre threads e tarefas assíncronas, o principal meio para troca de informações entre essas entidades é por meio de canais. O modelo de canais adotado segue o padrão `mpsc` (multiple producer, single consumer), no qual múltiplos produtores podem enviar dados para um único consumidor. Esse modelo minimiza a necessidade de acesso compartilhado à memória, garantindo que as mensagens sejam entregues de forma segura, sem risco de condições de corrida.

A criação de um canal é realizada por meio da função `mpsc::channel`, que retorna dois objetos: o transmissor (`Sender`) e o receptor (`Receiver`). O transmissor envia as mensagens, enquanto o receptor as recebe. O receptor bloqueia até que uma mensagem esteja disponível, o que torna o modelo simples e eficiente para comunicação síncrona. Um exemplo desse mecanismo é apresentado no Código 6, no qual o thread principal cria um canal e um novo thread. Este último envia uma mensagem para o thread principal, que a recebe usando a função `recv()`, que bloqueia até a chegada da mensagem.

```
use std::sync::mpsc;
use std::thread;
fn main() {
    let (tx, rx) = mpsc::channel();
    let t = thread::spawn(move || {
                let mensagem = String::from("Olá do thread!");
                tx.send(mensagem).unwrap();
            });
   let recebido = rx.recv().unwrap(); // recebe a mensagem
   println!("Mensagem recebida: {}", recebido);
   t.join().unwrap();
}
```

Código 6: Canais de Comunicação

Os canais de comunicação são thread-safe, ou seja, podem ser usados por múlti-

plos threads ou tarefas assíncronas sem causar problemas de concorrência. Além disso, o transmissor (`Sender`) pode ser clonado, permitindo que várias threads ou tarefas assíncronas enviem mensagens para o mesmo receptor.

3.5.2. Compartilhamento Seguro de Dados

Em Rust, o compartilhamento seguro de dados entre threads e tarefas assíncronas é um princípio central garantido pela linguagem por meio de diversas abstrações de sincronização e controle de acesso. Essas ferramentas permitem que dados sejam compartilhados entre threads de forma controlada, evitando condições de corrida e garantindo a integridade dos dados. As principais ferramentas para garantir o compartilhamento seguro de dados incluem `Mutex<T>`, `RwLock<T>` e `Arc<T>`. Os dois primeiros são *wrappers* que envolvem um dado a ser compartilhado. O `Mutex<T>` garante acesso exclusivo, tanto para leitura quanto para escrita, enquanto o `RwLock<T>` permite leituras concorrentes, mas restringe a escrita a uma thread por vez, garantindo exclusividade nesse caso.

O *wrapper* `Arc<T>` (atomic reference counting) é utilizado para permitir o compartilhamento de um `Mutex<T>` ou `RwLock<T>` entre múltiplas threads, gerenciando de forma segura a contagem de referências e evitando problemas como referências inválidas ou uso incorreto de memória.

```rust
use std::sync::{Arc, Mutex}; use std::thread;
fn main() {
  let counter = Arc::new(Mutex::new(0));
  let handles = vec![];
  for _ in 0..10 {
    let counter = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
  }
  for handle in handles {
      handle.join().unwrap();
  }
  println!("Contador: {}", *counter.lock().unwrap());
}
```

Código 7: Exemplo de uso de Mutex em Rust

Além dessas abstrações, Rust também disponibiliza tipos atômicos como `AtomicBool`, `AtomicUsize` e outros tipos numéricos, que permitem a execução de operações seguras de leitura-modificação-gravação em dados compartilhados. Essas operações atômicas são realizadas sem a necessidade de bloqueios explícitos, o que pode

melhorar o desempenho em casos onde a sobrecarga de sincronização seria um problema.

3.5.3. Ferramentas e Padrões Avançados

Para o desenvolvimento de programas mutithread em Rust o programador também conta com estruturas de sincronização do tipo barreiras (`Barrier`), semáforos e variáveis de condição (`Condvar`). A exemplo dos recursos de exclusão mútua apresentados anteriormente, também devem ser encapsulados em *wrappers* de proteção ao acesso.

Caso o enfoque a ser dado seja na exploração de paralelismo de dados, a linguagem conta com o suporte de `rayon`. O uso desta *crate* é particularmente interessante por integrar-se de forma transparente com iteradores, oferecendo um modelo funcional de paralelização com mínima complexidade adicional. Ainda se referindo à disponibilidade de modelos de programação de mais alto nível, também é suportada a implementação de sistemas baseados no modelo de atores por *crates* específicas, como a `actix`. Por meio deste recurso, *atores*, as entidades concorrentes e isoladas de execução, comunicam-se exclusivamente por meio de mensagens assíncronas.

3.6. Go

A linguagem de programação Go[3] [Cox-Buday 2017] foi criada, no final da década de 2000, com o objetivo de oferecer uma linguagem eficiente e com bom suporte à concorrência. O modelo de concorrência desta linguagem é centrado no uso de goroutines, unidades leves de programação concorrente que são multiplexadas de forma eficiente sobre um conjunto de threads do sistema operacional. A criação e gerenciamento de goroutines não exige o controle direto dos threads por parte do programador. A comunicação entre goroutines é feita por meio de canais, que permitem troca segura de dados, reduzindo a necessidade de mecanismos de sincronização baseados em seções críticas. Esse modelo, fortemente influenciado pela notação de processos de CSP (Communicating Sequential Processes), promove um estilo de programação concorrente mais estruturado e seguro. Go também fornece bibliotecas padronizadas para controle de sincronização, timers, pools de goroutines e estruturas reativas, ampliando o suporte à construção de sistemas concorrentes robustos e escaláveis.

3.6.1. Modelo Básico

O suporte à concorrência em Go foi concebido como parte central da linguagem, oferecendo um modelo estruturado que favorece a decomposição de programas em unidades independentes que se comunicam por meio de canais. Essa organização estimula o pensamento em termos de atividades que cooperam entre si, em vez de manipulação direta de estados compartilhados. A comunicação por canais, portanto, é o mecanismo utilizado para troca de dados entre fluxos concorrentes. A contrapartida é que esse modelo estruturado reduz o número de alternativas para a implementação de algoritmos às suportadas pela própria estrutura oferecida.

A unidade de programação concorrente em Go é a goroutine, que consiste em uma tarefa gerenciada integralmente pelo ambiente de execução da linguagem, o qual é composto por threads do sistema operacional. A comunicação e a coordenação entre essas

[3] `https://go.dev`

tarefas ocorrem por meio de canais, que funcionam simultaneamente como meio de troca de dados e como mecanismo explícito de sincronização entre goroutines.

3.6.1.1. Goroutines: Criação e Gerenciamento

Goroutines são a principal abstração de concorrência em Go e representam tarefas que podem ser executadas de forma independente. Elas são significativamente mais leves do que os threads do sistema operacional, tanto em termos de custo de criação quanto de consumo de memória.

Para lançar uma tarefa concorrente em Go, basta prefixar a chamada de uma função com a palavra-chave `go`. Isso instrui o ambiente de execução da linguagem a agendar a execução da função como uma nova tarefa concorrente, com retorno imediato após o lançamento da nova tarefa. A função chamada pode ser uma função nomeada, uma lambda ou até um método de uma estrutura. O Código 8 ilustra a criação de duas goroutines a partir da função `main`.

```
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) // 2 goroutines
  go f("goroutine 1", &wg)
  go func() {
    defer wg.Done()
    fmt.Println("Executando: goroutine 2")
  }()
  // ...
  wg.Wait()
  fmt.Println("Função main retornou.")
}
```

Código 8: Modelo básico de criação de goroutines em Go

O ambiente de execução de Go gerencia o agendamento e a distribuição das gorou-

tines sobre um conjunto de threads do sistema operacional, por meio de um escalonador cooperativo de múltiplas filas, mantido pelo runtime de execução da linguagem. Esse escalonador permite que as goroutines sejam mapeadas dinamicamente sobre os threads nativos, gerenciados pelo sistema operacional, conforme necessário, promovendo boa utilização dos *cores* disponíveis e evitando sobrecarga com alternância de contexto excessiva. Como o escalonamento é interno à linguagem, o programador não precisa gerenciar diretamente a criação ou finalização de threads.

Por padrão, os programas são encerrados assim que a função `main` retorna, mesmo que existam goroutines em execução. Para evitar essa situação, devem ser utilizados mecanismos de sincronização, como `sync.WaitGroup`, para aguardar a conclusão de goroutines cujo processamento ainda não tenha sido finalizado.

3.6.1.2. Barreiras e Seções Críticas

Embora canais sejam a forma nativa para compartilhamento de informações e sincronização entre goroutines, existem situações em que o controle direto sobre o acesso a dados compartilhados é necessário. Nestes casos, o uso de mecanismos de exclusão mútua, como `sync.Mutex` e `sync.RWMutex`, pode ser mais apropriado, oferecendo uma maneira explícita de coordenar o acesso a seções críticas de código.

O `sync.Mutex` é o tipo mais básico de bloqueio e garante que apenas uma goroutine tenha acesso a uma seção crítica de cada vez. Quando uma goroutine adquire o mutex, outras goroutines que tentarem adquirir o mesmo mutex ficarão bloqueadas até que ele seja liberado. Por outro lado, o `sync.RWMutex` oferece um mecanismo mais sofisticado, permitindo leitura simultânea por várias goroutines, desde que nenhuma goroutine tenha adquirido um bloqueio exclusivo para escrita. Isso é útil quando há um alto volume de leituras e um baixo volume de modificações, proporcionando uma melhor performance em cenários com acesso concorrente a dados.

Além desses mecanismos de exclusão mútua, Go também oferece a estrutura `sync.WaitGroup`, que é um mecanismo de sincronização utilizado para aguardar a conclusão de um conjunto de goroutines. O `sync.WaitGroup` atua como uma barreira, permitindo que uma goroutine principal ou outra goroutine aguarde a conclusão de múltiplas tarefas concorrentes antes de prosseguir com a execução. Essa ferramenta é particularmente útil em cenários onde o sincronismo entre a execução de múltiplas goroutines é necessário, sem depender diretamente da comunicação por canais.

Embora os canais sejam frequentemente utilizados para comunicação, os mecanismos de exclusão mútua fornecem uma solução eficiente para coordenar o acesso a dados compartilhados em memória, especialmente quando a comunicação por canais não é suficiente ou desejada.

3.6.2. Comunicação Entre Goroutines

A comunicação entre goroutines em Go é feita de forma segura e eficiente usando canais. Go oferece um modelo de passagem de mensagens por canais que elimina a necessidade de bloqueios explícitos para compartilhar dados entre goroutines.

3.6.3. Comunicação Concorrente com Canais

Canais são o principal mecanismo de comunicação entre goroutines em Go. Eles permitem a troca de dados de forma segura entre tarefas concorrentes, promovendo tanto a transmissão de valores quanto a sincronização implícita entre os participantes da comunicação. Por meio dos canais, é possível construir programas concorrentes que evitam o uso de variáveis compartilhadas e os problemas típicos associados à sua manipulação.

Um canal é criado com a função `make`, especificando o tipo dos dados que serão transmitidos. As operações de envio e recebimento são realizadas com o operador ←. Ao enviar um valor para um canal, a goroutine emissora fica bloqueada até que outra goroutine esteja pronta para recebê-lo, a menos que a capacidade de bufferização do canal não esteja esgotada. Esse comportamento sincronizado permite que os canais sirvam como ponto de coordenação entre diferentes tarefas. Em casos em que o remetente não mais enviará valores, a função close permite sinalizar o encerramento do canal. Essa prática é especialmente útil quando o receptor está em um laço de leitura, evitando bloqueios ou a necessidade de lógica adicional para encerrar a leitura. Tais laços de leitura são implementados pelo construtor `for-range`, permitindo iterar sobre todas as mensagens recebidas e encerrando ao seu término.

```
for val := range ch {
  fmt.Println(val)
```

Além dos canais bidirecionais, é possível restringir a direção de uso de um canal, criando canais unidirecionais. Essa especialização pode ser útil para modularizar o código e reforçar garantias de segurança no uso da comunicação, indicando de maneira explícita se determinada função apenas envia ou apenas recebe dados por um canal.

Para lidar com múltiplos canais simultaneamente, Go oferece a construção `select`, que permite aguardar operações de envio ou recebimento em diferentes canais. O `select` avalia de forma não determinística os casos que estiverem prontos para prosseguir, o que permite, por exemplo, implementar mecanismos de multiplexação, onde diferentes fontes de dados podem ser atendidas de maneira reativa. Esse recurso também permite a implementação de timeouts e cancelamento de operações concorrentes, quando combinado com canais auxiliares ou temporizadores, como `time.After`.

Canais podem ser definidos como com ou sem buffer. Um canal sem buffer exige que o remetente e o receptor estejam prontos simultaneamente para que a operação de envio ou recebimento seja completada. Já um canal com buffer permite que um número limitado de valores seja enviado mesmo que nenhuma goroutine esteja aguardando para recebê-los imediatamente. Isso introduz uma forma de desacoplamento entre remetente e receptor, útil em situações onde a produção e o consumo de dados ocorrem em ritmos distintos.

O uso de canais com buffer pode melhorar a performance de sistemas concorrentes, desde que o tamanho do buffer seja escolhido cuidadosamente, de acordo com as características da aplicação. Um buffer muito pequeno pode levar a bloqueios frequentes, enquanto um buffer muito grande pode introduzir latência e dificultar o controle de fluxo.

3.6.4. Sincronização e Exclusão Mútua

Embora Go ofereça mecanismos de comunicação entre goroutines por meio de canais, há situações em que é necessário controlar diretamente o acesso a recursos compartilhados. Para esses casos, a linguagem fornece estruturas de sincronização no pacote `sync`, como `Mutex` e `RWMutex`.

O tipo `sync.Mutex` implementa exclusão mútua. Ele permite que apenas uma goroutine acesse uma seção crítica por vez, impedindo condições de corrida quando múltiplas goroutines tentam modificar o mesmo recurso.

Para usá-lo, declara-se uma variável do tipo `sync.Mutex` e envolve-se o acesso ao dado com chamadas aos métodos `Lock()` e `Unlock()`. Exemplo no Código 9.

```
var m sync.Mutex var contador int
func incrementar() {
  m.Lock()
  contador++
  m.Unlock()
}
```

Código 9: Modelo básico de criação de goroutines em Go

É fundamental garantir que o método `Unlock()` seja sempre chamado, mesmo em casos de erro. Uma alternativa é, imediatamente após a obtenção do mutex, automatizar a liberação do mutex ao final da execução da função na qual o mutex foi adquirido, invocando sobre o objeto mutex a operação `defer`. Em termos práticos, a invocação da operação `defer` instrui o compilador a introduzir um código ao final da execução da função que executa o desbloqueio, independente se este término ocorreu normalmente ou por uma operação de pânico.

O tipo `sync.RWMutex` é uma variação do `Mutex` que permite múltiplas leituras simultâneas, desde que nenhuma escrita esteja em andamento. Ele é útil quando há predominância de operações de leitura sobre um mesmo recurso compartilhado.

O método `RLock()` bloqueia o mutex para leitura, enquanto `Lock()` bloqueia para escrita. A liberação deve ser feita com `RUnlock()` ou `Unlock()`, respectivamente.

3.6.5. Ferramentas e Bibliotecas Avançadas

Além das primitivas básicas de sincronização e comunicação, a linguagem Go oferece ferramentas adicionais que auxiliam na construção de sistemas concorrentes mais robustos e eficientes. Dentre elas, destaca-se o pacote `context`, usado para controlar o ciclo de vida de goroutines. Com ele, é possível propagar sinais de cancelamento e configurar limites de tempo para operações, o que facilita o gerenciamento cooperativo de tarefas, especialmente em aplicações com chamadas encadeadas e alto grau de concorrência. O

cancelamento explícito por meio de `context.WithCancel` ou a imposição de prazos com `context.WithTimeout` permitem evitar vazamentos de goroutines e garantir encerramento limpo de atividades.

Outro recurso útil é a estrutura `sync.Pool`, que implementa um pool de objetos reutilizáveis. Seu uso é recomendado em cenários nos quais há criação intensiva de objetos temporários, como buffers ou estruturas auxiliares. Ao minimizar a pressão sobre o coletor de lixo, essa estrutura pode contribuir significativamente para a redução do custo de alocação dinâmica e melhorar a performance de aplicações paralelas. No entanto, o conteúdo de um `sync.Pool` pode ser descartado a qualquer momento, e seu uso deve ser reservado a situações nas quais a perda de elementos armazenados não compromete a lógica da aplicação.

3.7. Elixir

A linguagem Elixir[4] [Marx et al. 2018] foi projetada para a construção de sistemas concorrentes, distribuídos e tolerantes a falhas. Baseada na máquina virtual Erlang (BEAM), herda um modelo de concorrência centrado em processos leves, isolamento completo e comunicação por troca de mensagens. Em contraste com abordagens baseadas em threads e memória compartilhada, Elixir adota o modelo de atores, no qual cada processo atua como uma unidade independente, com estado próprio e interação exclusivamente assíncrona. Essa arquitetura favorece a escalabilidade e a robustez, características centrais em aplicações como servidores web, sistemas de telecomunicações e pipelines de dados em tempo real. Nesta seção, analisamos os principais recursos oferecidos por Elixir para programação concorrente, começando pela criação e comunicação entre processos, passando por mecanismos de recepção e tratamento de falhas, até abstrações de mais alto nível disponíveis na linguagem e em seu ecossistema.

3.7.1. Modelo Básico

O modelo de concorrência adotado por Elixir baseia-se na criação massiva de processos leves, também chamados de processos Erlang. Esses processos são completamente isolados entre si, não compartilham memória e se comunicam exclusivamente por meio de troca de mensagens. Essa abordagem deriva diretamente da máquina virtual BEAM, desenvolvida para suportar sistemas distribuídos e tolerantes a falhas.

A criação de processos em Elixir é feita com a função `spawn`, que recebe como argumento uma função a ser executada de forma concorrente. Cada processo criado dessa forma possui seu próprio espaço de execução e pode ser identificado por um identificador do tipo `pid` (process identifier). Como esses processos são extremamente leves, ocupando apenas poucos kilobytes de memória, é comum a criação de milhares ou até milhões de processos em aplicações reais.

A seguir, um exemplo simples de criação de processos:

```
defmodule Demo do
  def saudacao do
```

[4] https://elixir-lang.org

```
    IO.puts("Olá do processo!")
  end
end
pid = spawn(Demo, :saudacao, [])
```

No exemplo acima, o processo principal invoca `spawn/3`, passando o módulo, a função e os argumentos a serem usados. A função `saudacao` será executada concorrentemente, imprimindo uma mensagem. O valor retornado por `spawn` é o `pid` do processo criado, que pode ser usado para interação posterior.

Embora este exemplo não envolva comunicação entre processos, ele já ilustra o princípio fundamental da concorrência em Elixir: a criação de unidades isoladas e independentes de execução.

3.7.2. Envio e Recepção de Mensagens

A comunicação entre processos em Elixir é feita por meio de troca de mensagens, utilizando um modelo assíncrono. Cada processo possui uma caixa de correio própria (mailbox), que armazena as mensagens recebidas até que sejam explicitamente processadas. O envio é realizado com o operador `send`, enquanto a recepção é feita com a construção `receive`.

O envio de mensagens em Elixir pode ser feito utilizando a forma `pid <- mensagem`, uma construção da linguagem que equivale a `send(pid, mensagem)`. Essa operação permite que qualquer processo envie uma mensagem a outro, desde que conheça seu pid. As mensagens podem ser de qualquer tipo de dado Elixir, incluindo átomos, tuplas, listas ou estruturas mais complexas.

A estrutura `receive` permite que um processo filtre e trate mensagens conforme padrões definidos, como no exemplo abaixo:

```elixir
defmodule Mensageiro do
  def iniciar do
    receive do
      {:ola, remetente} ->
        IO.puts("Recebido: olá de #{inspect(remetente)}")

      _mensagem ->
        IO.puts("Mensagem desconhecida: #{inspect(_mensagem)}")
    end
  end
end
pid = spawn(Mensageiro, :iniciar, [])
send(pid, {:ola, self()})
```

Código 10: Modelo básico de criação de processos em Elixir

Neste exemplo, o processo criado por `spawn` fica aguardando uma mensagem. Ao receber a tupla `:ola, remetente`, corresponde ao padrão e imprime uma saudação. Caso a mensagem não corresponda ao padrão esperado, uma resposta genérica é impressa. O uso de `self()` retorna o `pid` do processo atual (o remetente).

Por padrão, o `receive` bloqueia a execução até que uma mensagem seja recebida. É possível, no entanto, especificar um tempo limite com a construção `after`:

```elixir
receive do
  :ping ->
    IO.puts("pong")
after
  1000 ->
    IO.puts("Nenhuma mensagem recebida em 1 segundo")
end
```

Esse controle de tempo evita que um processo fique indefinidamente bloqueado esperando por uma mensagem que talvez nunca chegue. Caso o tempo especificado (em milissegundos) expire sem o recebimento de mensagens, a cláusula `after` é executada. No exemplo, 1.000 ms.

A estrutura `receive`, além de permitir a recepção de mensagens, oferece recursos expressivos para tratar diferentes padrões de mensagens. Cada cláusula dentro do `receive` especifica um padrão e as ações a serem tomadas quando esse padrão for correspondente. O mecanismo de correspondência segue as regras do pattern matching da linguagem, permitindo o uso de tuplas, listas, átomos e até guardas lógicas com `when` para maior controle.

Além da filtragem, é comum que o código precise lidar com mensagens inesperadas. Uma prática recomendada é incluir uma cláusula coringa, como `_ -> ...`, para capturar qualquer mensagem não tratada explicitamente, evitando que fiquem acumuladas na mailbox.

3.7.3. Monitoramento e Tolerância a Falhas

A concorrência em Elixir foi concebida com foco na construção de sistemas robustos e resilientes a falhas. Essa resiliência se apoia em dois pilares: a detecção de falhas entre processos e a capacidade de reinício automático, organizados sob a filosofia conhecida como "let it crash". Essa abordagem parte do princípio de que erros são inevitáveis e que os processos devem falhar rapidamente, delegando a recuperação ao sistema de supervisão.

Elixir fornece primitivas para que processos observem uns aos outros. A função `Process.monitor/1` cria uma relação de monitoramento unidirecional: o processo monitorado não tem conhecimento da existência do monitor. Se o processo monitorado for encerrado, o monitor receberá uma mensagem `:DOWN, ref, :process, pid, razao`, informando o motivo do término. O monitoramento é indicado quando não se deseja que os dois processos compartilhem destino, ou seja, que a falha de um não cause a finalização do outro.

Por outro lado, a função `link/1` estabelece um vínculo bidirecional. Se um dos processos falhar de maneira não tratada, o outro será automaticamente finalizado, a menos que intercepte o sinal de saída com mecanismos como `Process.flag(:trap_exit, true)`. Esse tipo de ligação é útil quando processos dependem mutuamente para garantir a consistência de uma operação.

Com base nesses mecanismos, Elixir organiza processos em estruturas hierárquicas chamadas árvores de supervisão (supervision trees). Um supervisor é um processo especial que monitora outros processos filhos e aplica estratégias definidas de reinício em caso de falha. O módulo `Supervisor` permite especificar como e quando cada processo deve ser reiniciado, por exemplo: reinício individual (`:one_for_one`), reinício em cascata (`:one_for_all`) ou reinício proporcional (`:rest_for_one`).

Essa arquitetura permite que aplicações sejam estruturadas em componentes pequenos, isolados e recuperáveis, nos quais falhas locais não comprometem o funcionamento global. A tolerância a falhas em Elixir, portanto, não é um recurso adicional, mas um princípio de design central no desenvolvimento de sistemas concorrentes confiáveis.

3.7.4. Agentes, Tasks e GenServers

Além do modelo explícito baseado em `spawn`, envio de mensagens e `receive`, Elixir oferece abstrações de mais alto nível para encapsular lógica concorrente de forma estruturada. Entre elas, destacam-se os módulos `Agent`, `Task` e `GenServer`, todos construídos sobre os princípios fundamentais de processos isolados e troca de mensagens.

O módulo `Agent` é usado para encapsular e gerenciar estado de maneira simples. Ele fornece uma interface funcional para atualizar e consultar valores mantidos por um processo. Essa abordagem permite compartilhamento controlado de estado sem recor-

rer a variáveis globais ou memória compartilhada. Um agente é iniciado com a função `Agent.start_link`, que cria um processo que mantém o estado interno:

```
{:ok, pid} = Agent.start_link(fn -> 0 end)
Agent.update(pid, fn state -> state + 1 end)
Agent.get(pid, fn state -> state end)
```

O módulo `Task` fornece uma forma conveniente de executar funções assíncronas e aguardar seu resultado. É útil em situações onde se deseja disparar uma computação paralela de curta duração. A função `Task.async` cria um processo e retorna uma referência, enquanto `Task.await` bloqueia até a resposta estar disponível:

```
task = Task.async(fn -> realizar_calculo() end)
resultado = Task.await(task)
```

Por fim, o módulo `GenServer` (Generic Server) é uma abstração poderosa para construir servidores de processos com ciclo de vida bem definido, estado interno persistente e suporte integrado a chamadas síncronas e assíncronas. Um `GenServer` implementa funções de callback como `handle_call`, `handle_cast` e `handle_info` para lidar com diferentes tipos de mensagens. Ele é amplamente usado em aplicações robustas e é compatível com o sistema de supervisão padrão de Elixir.

3.7.5. Fluxo Paralelo de Dados

O módulo `Flow`, parte do ecossistema Elixir, oferece uma abstração funcional de alto nível para processamento concorrente e paralelo de coleções de dados. Construído sobre a biblioteca `GenStage`, o `Flow` simplifica a criação de pipelines paralelos, delegando à infraestrutura subjacente a coordenação de demanda, o balanceamento de carga e a divisão do trabalho entre múltiplos processos.

A estrutura de um fluxo é composta por três etapas principais: entrada (*source*), transformação (*transform*) e agregação ou coleta (*sink*). A entrada pode ser uma coleção enumerável, como listas ou streams, e o fluxo é iniciado com `Flow.from_enumerable/1`. A seguir, podem ser encadeadas operações como `map/2`, `filter/2`, `flat_map/2` e `reduce/3`, cada uma aplicada em paralelo por diferentes instâncias de processo. A distribuição do trabalho entre essas instâncias é controlada pela função `Flow.partition/2`, que permite, por exemplo, especificar o número de stages (processos consumidores), chaves de particionamento e estratégias de roteamento.

Por exemplo, a função `Flow.partition(stages: n)` cria `n` stages consumidores, entre os quais os dados são distribuídos com base em um particionador hash ou definido pelo usuário. Cada um desses stages executa em seu próprio processo, permitindo o uso eficiente de múltiplos núcleos da CPU.

O processamento é executado de forma preguiçosa, ou seja, apenas quando uma função terminal como `Enum.to_list/1`, `Enum.reduce/3` ou `Enum.each/2` for chamada. Durante a execução, o `Flow` garante consistência entre os estágios, aplica controle de demanda para evitar sobrecarga (backpressure) e permite modelar sistemas complexos, como pipelines de produtores e consumidores, de forma declarativa.

3.8. Estudos de Caso

Esta seção apresenta dois estudos de caso, a implementação do problema do produtor/consumidor e a implementação do cálculo da *n*-ésima posição da série de Fibonacci. Neste texto, apenas parte do código é apresentada para ilustrar o uso das ferramentas. Para os estudos de caso, são apresentadas diferentes implementações em C++, Rust, Go e Elixir, sem, contudo, esgotar as possibilidades de construção de soluções aos problemas nessas linguagens. Como nota, cabe informar que as implementações foram concebidas com intuito de oferecer um material didático, e não houve preocupações com questões de desempenho.

3.8.1. Reprodutibilidade

Os exemplos apresentados este texto encontram-se disponíveis, completos, no repositório criado para complementar o material deste texto `https://github.com/GersonCavalheiro/JAI2025`. O repositório conta também com outras implementações, dos mesmos casos de estudo, e de outros. Para facilitar o uso das ferramentas de programação trabalhadas neste texto, o repositório também conta com um Dockerfile, para construção de uma imagem contendo os recursos necessários.

3.8.2. Produtor/Consumidor

Neste algoritmo, produtores e consumidores compartilham uma área de dados comuns, uma lista compartilhada, onde itens produzidos pelos produtores são enfileirados e consumidos pelos consumidores. Os parâmetros de lançamento deste programa são o número de produtores e consumidores a serem instanciados e o número de itens a serem produzidos pelos produtores individualmente. Os itens produzidos são os primeiros números primos. Cada produtor ao completar sua cota de produção, finaliza. A função `main` é responsável por sinalizar os consumidores do termino do programa introduzindo o valor -1 na fila compartilhada de itens.

3.8.2.1. Produtor/Consumidor em C++

Em C++ foram desenvolvidas três versões de programas implementando o algoritmo Produtor-Consumidor. Todas as versões, em comum, o acesso à seção seção crítica é controlado por mutex, tendo como apoio um segundo mecanismo de sincronização: variável de condição, promessas e futuros, e cancelamento cooperativo.

Na versão empregando variável de condição na sua solução, cada produtor, ao finalizar a produção de um número primo, obtém ingresso a uma seção crítica adquirindo um mutex com o recurso `std::unique_lock`, podendo então o inserir na fila compartilhada protegida de itens produzidos. Após a inserção, os consumidores são sinalizados

da presença de um novo item na lista com uma notificação na variável de condição com `cv.notify_one()` que tem sinaliza algum eventual consumidor que esteja suspenso em espera para realizar um consumo. Cada consumidor, por sua vez, aguarda itens que exista pelo menos um item na fila para ser consumido. Após consumir um item, verifica se é o valor sentinela (número negativo) para encerrar sua execução. O valor sentinela é reenfileirado para acordar outros consumidores restantes. Essa implementação se caracteriza por implementar uma estratégia convencional e robusta e utilizando um mecanismo nativo de espera/bloqueio, garantindo eficiência. No entanto, a programação do algoritmo desta forma exigiu que fossem introduzidos mecanismos para coordenar explicitamente a notificação da produção de itens e acesso em regime de exclusão mútua à lista compartilhada e introduzir explicitamente sinalização da condição de parada. O Código 11 apresenta, de forma simplificada, a estrutura da implementação desta versão do problema.

```cpp
std::mutex mtx;
std::condition_variable cv;
std::queue<int> buffer;

void produtor(int id, int total) {
  int count = 0, num = 2;
  while (count < total) {
    if (is_prime(num)) {
      std::unique_lock<std::mutex> lock(mtx);
      buffer.push(num);
      cv.notify_one();
      count++;
    }
    num++;
  }
}

void consumidor(int id) {
  while (true) {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return !buffer.empty(); });
    int val = buffer.front(); buffer.pop();
    if (val < 0) { buffer.push(val); break; }
    ... // processa item
  }
}
```

Código 11: Produtor-Consumidor em C++ utilizando variável de condição.

Na versão baseada em promessas e futuros, Código 12, a coordenação da finalização entre produtores e consumidores é realizada por meio de objetos `std::promise` e `std::future`, enquanto o acesso à fila compartilhada ainda é protegido por `std::mutex`. Cada produtor, ao gerar um número primo, obtém acesso exclusivo à lista de itens produzidos utilizando `std::lock_guard`, inserindo o item na fila de forma segura. Ao concluir sua participação, o produtor sinaliza seu término invocando `set_value()` em seu objeto `promise`, permitindo que o thread principal detecte que aquela produção foi encerrada por meio da `future` correspondente. Já os consumidores executam de forma contínua, consultando a fila em busca de itens. Caso a fila esteja vazia, aguardam brevemente com `sleep_for()`, realizando uma espera ativa leve. Ao consumir um item, verificam se é o valor de término, o que indica que devem encerrar sua execução. O valor de indicação de término é reenfileirado para garantir que os demais consumidores também recebam o sinal de parada. Essa implementação evita o uso de variáveis de condição, adotando uma abordagem mais explícita de sincronização da finalização dos produtores com base em futuros. Por outro lado, impõe o custo de uma espera ativa no lado do consumidor e exige o uso adicional de objetos de sincronização dedicados ao encerramento do fluxo de produção.

Na terceira versão implementada empregando cancelamento cooperativo, Código 13, é adotado o uso da classe `std::jthread`. Cada thread produtor gera números

```cpp
std::mutex mtx;
std::queue<int> buffer;

void produtor(int id, int total, std::promise<void> prom) {
  int count = 0, num = 2;
  while (count < total) {
    if (is_prime(num)) {
      {
        std::lock_guard<std::mutex> lock(mtx);
        buffer.push(num);
      }
       count++;
    }
    num++;
  }
  prom.set_value(); // sinaliza fim da produção
}
```

```cpp
void consumidor(int id) {
  while (true) {
    int val = -2;
    {
      std::lock_guard<std::mutex> lock(mtx);
      if (!buffer.empty())
        { val = buffer.front(); buffer.pop(); }
    }
    if (val == -1) break;
    if (val >= 0)   ... // processa item
     else std::this_thread::sleep_for(...);
  }
}
```

Código 12: Produtor-Consumidor em C++ utilizando promessas e futuros.

primos enquanto o token associado não sinalizar uma solicitação de parada. A inserção dos itens na fila compartilhada é feita dentro de uma seção crítica protegida por `std::mutex`, utilizando `std::unique_lock`, seguida de uma notificação aos consumidores com `cv.notify_one()`. Cada thread consumidor, por sua vez, aguarda a chegada de novos itens utilizando uma variável de condição combinada com um predicado que leva em consideração tanto a presença de itens na fila quanto o estado do `stop_token`. Quando a fila estiver vazia e o cancelamento tiver sido solicitado, o consumidor encerra sua execução de forma segura. A sinalização de parada é realizada pelo programa principal após um intervalo predeterminado, utilizando o método `request_stop()` oferecido por `std::jthread`. Essa abordagem destaca-se por integrar de forma coesa o controle de cancelamento ao ciclo de vida das threads, promovendo uma finalização ordenada e segura sem a necessidade de valores sentinela ou mecanismos adicionais de sinalização explícita. Em contrapartida, exige que cada thread verifique periodicamente o estado do token de parada, o que introduz um aspecto de programação cooperativa ao controle de execução.

```cpp
std::mutex mtx;
std::condition_variable cv;
std::queue<int> buffer;

void produtor(std::stop_token st, int id, int total) {
  int count = 0, num = 2;
  while (count < total && !st.stop_requested()) {
    if (is_prime(num)) {
      {
        std::unique_lock<std::mutex> lock(mtx);
        buffer.push(num);
      }
      cv.notify_one();
      count++;
    }
    num++;
  }
}
```

```cpp
void consumidor(std::stop_token st, int id) {
  while (!st.stop_requested()) {
    int val = -1;
    {
      std::unique_lock<std::mutex> lock(mtx);
      cv.wait(lock, [&] { return !buffer.empty()
                      || st.stop_requested(); });
      if (st.stop_requested()) continue;
      val = buffer.front(); buffer.pop();
    }
    ... // processa item
    }
}
```

Código 13: Produtor-Consumidor em C++ utilizando cancelamento cooperativo.

3.8.2.2. Produtor/Consumidor em Rust

Em Rust foram desenvolvida duas versões para o problema. A primeira utiliza uma estrutura de dados (fila) compartilhada entre produtores e consumidores, protegida por mutex, enquanto a segunda utiliza canais assíncronos para comunicação. Desta forma, a primeira versão implmenta uma solução baseada em condição associada ao estado da fila compartilhada, enquanto a segunda emprega uma abordagem baseada em comunicação explícita entre threads.

Na versão empregando uma fila compartilhada, Código 14, a seção crítica acessando o dado compartilhado é protegida por `Mutex` e sincronizada com `Condvar`. Os produtores sinalizam disponibilidade de novos elementos por meio de chamadas a `notify_one()` sobre a variável de condição. Já os threads consumidores aguardam até que haja itens disponíveis na fila (`Condvar::wait`) para suspender sua execução de maneira eficiente até serem notificados. Ao detectar um valor especial (`u32::MAX`) inserido após o término da produção, os consumidores encerram sua execução.

```rust
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

fn main() {
    let buffer = Arc::new((Mutex::new(Vec::new()), Condvar::new()));
    let buffer_p = Arc::clone(&buffer);
    let produtor = thread::spawn(move || {
        for i in 1..=5 {
            let (lock, cvar) = &*buffer_p;
            let mut buf = lock.lock().unwrap();
            buf.push(i);
            println!("Produtor gerou {}", i);
            cvar.notify_one();
        }
        let (lock, cvar) = &*buffer_p;
        let mut buf = lock.lock().unwrap();
        buf.push(u32::MAX);
        cvar.notify_one();
    });

    let buffer_c = Arc::clone(&buffer);
    let consumidor = thread::spawn(move || {
        loop {
            let (lock, cvar) = &*buffer_c;
            let mut buf = lock.lock().unwrap();
            while buf.is_empty() {
                buf = cvar.wait(buf).unwrap();
            }
            let val = buf.remove(0);
            if val == u32::MAX {
                break;
            }
            ... // processa item
        }
    });

    produtor.join().unwrap();
    consumidor.join().unwrap();
}
```

Código 14: Produtor-Consumidor em Rust utilizando variável de condição.

Na segunda versão implementada em Rust, os itens produzidos são enviados aos consumidores via um canal de comunicação (`std::sync::mpsc::channel`), Código 15. Os consumidores compartilham um único receptor protegido com `Arc<Mutex<Receiver<Item»>`, garantindo que apenas um thread por vez tenha acesso à operação de recebimento. Ao tentar consumir, cada thread adquire o mutex, realiza a chamada a `recv` e, se obtiver sucesso, imprime o item recebido. Quando todos os produtores finalizam, o canal é fechado, e os consumidores detectam o encerramento por meio de um erro retornado por `recv`.

3.8.2.3. Produtor/Consumidor em Go

Na linguagem Go foram realizadas duas implementações do problema do produtor-consumidor, a primeira mais próxima a concepção tradicional da solução, utilizando seções críticas, controladas por mutex, para acesso a dados compartilhados. Já a segunda

```rust
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
struct Item {
  produtor_id: usize,
  valor: usize,
}
fn main() {
  let (tx, rx) = mpsc::channel();
  let rx = Arc::new(Mutex::new(rx));
  let mut handles = vec![];
  for id in 1..=2 {
    let tx = tx.clone();
    let h = thread::spawn(move || {
      for i in 0..3 {
        let valor = 100 * id + i;
        tx.send(Item { produtor_id: id, valor }).unwrap();
      }
    });
    handles.push(h);
  }
  drop(tx);
  for id in 1..=2 {
    let rx = Arc::clone(&rx);
    let h = thread::spawn(move || {
        loop {
          let item = {
              let rx = rx.lock().unwrap();
              rx.recv()
          };
          match item {
            Ok(Item { produtor_id, valor }) => {
              ... // processa item
            }
            Err(_) => {
              break;
            }
          }
        }
      });
      handles.push(h);
  }
  for h in handles {
    h.join().unwrap();
  }
}
```

Código 15: Produtor-Consumidor em Rust utilizando canais de comunicação.

implementação apresenta uma abordagem idiomática mais característica à linguagem, utilizando exclusivamente canais de comunicação.

Na primeira implementação em Go, Código 16, a coordenação entre os threads produtores e consumidores combina o uso de canais de comunicação com seções críticas protegidas por `sync.Mutex`. Os threads produtores, após produzirem cada item, acessam uma variável global para registrar o último número testado. Esse acesso é protegido por uma região crítica, garantindo que apenas um thread por vez atualize o contador global e escreva no vetor compartilhado. Após preencher esse vetor com todos os primos esperados, o canal de comunicação é utilizado para que os consumidores solicitem os valores gerados. Os threads consumidores, por sua vez, leem da lista de primos também dentro de seções críticas protegidas por `sync.Mutex`, incrementando um contador global que determina quando o consumo deve ser encerrado. Essa abordagem adota um modelo híbrido: os dados são transmitidos por canal, mas a sincronização de acesso ao estado global (como contadores e o vetor de primos) é realizada por exclusão mútua.

A segunda versão implementada em Go, Código 17, emprega canais para comunicação e encerramento automático da execução. Cada thread produtor gera números primos (função `ehPrimo`), e os envia por meio de um canal do tipo `Item`, que associa o valor gerado à identificação do produtor. Os threads consumidores leem continuamente do canal compartilhado, processando cada item recebido com uma pequena simulação de carga. O encerramento da execução dos consumidores ocorre de forma natural com o fechamento do canal, acionado automaticamente após a finalização de todos os produtores, monitorada por um `sync.WaitGroup`. Essa estratégia elimina a necessidade de valores sentinela ou verificação de condições globais, simplificando o controle do ciclo de vida das goroutines. A sincronização entre threads ocorre implicitamente por meio do canal, que garante consistência e exclusividade no acesso aos dados trafegados, dispensando o uso de seções críticas.

```go
func isPrime(n int) bool { ... }
func producer(id, n int, tx chan int,
wg *sync.WaitGroup,
produced *int, mu *sync.Mutex) {
defer wg.Done()
num := 2
for {
    mu.Lock()
    if *produced >= n {
        mu.Unlock()
        break
    }
    mu.Unlock()

    if isPrime(num) {
        mu.Lock()
        if *produced < n {
            tx <- num
            *produced++
        }
        mu.Unlock()
    }
    num++
  }
}
```

```go
func consumer(id, n int, rx chan int,
              wg *sync.WaitGroup,
              consumed *int, mu *sync.Mutex) {
  defer wg.Done()
  for {
    mu.Lock()
    if *consumed >= n { mu.Unlock() break }
    mu.Unlock()
    item := <-rx
    mu.Lock()
    *consumed++
    mu.Unlock()
    ... // processa item
  }
}
func main() {
  tx := make(chan int, BUFFER_SIZE)
  var produced, consumed int
  var mu sync.Mutex
  var wg sync.WaitGroup
  for i := 0; i < numProdutores; i++ {
    wg.Add(1)
    go producer(i, n, tx, &wg, &produced, &mu)
  }
  for i := 0; i < numConsumidores; i++ {
    wg.Add(1)
    go consumer(i, n, tx, &wg, &consumed, &mu)
  }
  wg.Wait()
}
```

Código 16: Produtor-Consumidor em Go empregando seção crítica.

3.8.2.4. Produtor/Consumidor em Elixir

Em Elixir form realizadas implementações de três programas para o problema produtor-consumidor. Todas versões empregam canais de comunicação (`send` e `receive`), e fazem uso do paralelismo oferecido pela criação de processos leves com `spawn`.

A versão com canal único centraliza a comunicação em um único processo intermediário que gerencia uma fila de mensagens compartilhada, exigindo lógica de repetição e controle de disponibilidade explícitos. A segunda versão utiliza um processo distribuidor dedicado à entrega das mensagens aos consumidores, adotando uma estratégia de balanceamento em *round-robin*. Já a terceira versão aproveita a biblioteca `Flow`, que abstrai a distribuição e o particionamento dos dados de forma automática, integrando o paralelismo com uma semântica funcional e pipelines concorrentes.

A primeira versão implementada, Código 18, utiliza um processo dedicado como canal de comunicação entre produtores e consumidores. Cada thread produtor envia mensagens do tipo `{:put, valor}` sobre esse canal, o qual mantém internamente uma fila de itens produzidos. Os consumidores, solicitam novos itens com mensagens `{:get, pid}`, e recebem a resposta diretamente por meio de uma nova mensagem enviada ao seu próprio identificador de processo. Essa abordagem exige que cada consumidor implemente lógica de nova tentativa em caso de fila vazia, utilizando `receive` com `after` para repetição.

A segunda versão em Go, 19, substitui a fila centralizada empregada na versão anterior por um processo distribuidor, eliminando a necessidade de polling nos consu-

```go
func producer(id int, n int, tx chan int, wg *sync.WaitGroup) {
  defer wg.Done()
  num := 2
  enviados := 0
  for enviados < n {
    if isPrime(num) {
      tx <- num
      enviados++
    }
    num++
  }
}
func consumer(id int, rx chan int, wg *sync.WaitGroup) {
  defer wg.Done()
  for item := range rx {
    fmt.Printf("Consumidor %d consumiu: %d\n", id, item)
  }
}
func main() {
  tx := make(chan int, 10)
  var wg sync.WaitGroup
  wg.Add(2)
  go producer(1, 3, tx, &wg)
  go producer(2, 3, tx, &wg)
  go func() {
    wg.Wait()
    close(tx)
  }()
  wg.Add(1)
  go consumer(1, tx, &wg)
  wg.Wait()
}
```

Código 17: Produtor-Consumidor em Go empregando canais de comunicação.

```elixir
defmodule ProdutorConsumidor do
  canal_loop(items) do
    {:put, item}
    {:get, pid}
      - {:item, valor}
      - {:retry}
    {:shutdown}
  end
  canal_pid = spawn(fn -> canal_loop([]) end)
  Process.register(canal_pid, :canal)
  main(args) do
    gerar_primos(n)
    canal_pid = spawn(...)
    Process.register(canal_pid, :canal)
    for i <- 1..nc, do: spawn(...)
    Enum.chunk_every(primos)
    Enum.each(..., fn -> spawn(...) end)
  end
  produtor(id, primos) doA
    Enum.each(primos)
    send(:canal, {:put, p})
    :timer.sleep(:rand.uniform)
  end
  consumidor(id, timeout \\ 1000) do
    send(:canal, {:get, self()})
    receive do
      {:item, valor}  -> ... # processa item
      {:retry}
    after
      timeout -> consumidor(...)
    end
  end
end
```

Código 18: Produtor-Consumidor em Elixir canal único comunicação.

midores e reduzindo o acoplamento entre os produtores e consumidores. O processo distribuidor assume a responsabilidade de receber os itens por um canal de comunicação e os distribuir, em ordem cíclica, aos consumidores.

A terceira versão implementada, Código 20, emprega a biblioteca `Flow` para organizar os produtores como fontes de dados e os consumidores como estágios particionados de um fluxo concorrente. O fluxo é automaticamente particionado com `Flow.partition`, e cada etapa aplica transformações sobre os dados recebidos. O modelo favorece concisão, paralelismo implícito e integração com o paradigma funcional próprio à linguagem, dispensando a implementação manual de lógica de troca de mensagens ou de controle de ciclo de vida dos processos.

3.8.3. Cálculo de Fibonacci

O cálculo da n-ésima posição na série de Fibonacci é dado por $Fibonacci(n)$, quando $n > 1$ é dado por $Fibonacci(n-1) + Fibonacci(n-2)$ e para $n = 0$ ou $n = 1$, $Fibonacci(n) = n$. Este problema é frequentemente abordado no estudo de ferramentas para programação

```
defmodule ProdutorConsumidor do
  distribuidor(nc) do
    consumidores = for i <- 1..nc, do: spawn(...)
    loop(consumidores)
  end
  loop(consumidores) do
    receive do
      {:item, valor} ->
        send(dest, {:item, valor})
        loop(resto ++ [dest])
    end
  end
  main(args) do
    buffer_pid = spawn(fn -> distribuidor(nc) end)
    produtores = for i <- 1..np, do: spawn(...)
    Enum.each(produtores, &Process.monitor/1)
    wait_for_produtores(np)
    for _ <- 1..nc, do: send(buffer_pid, {:item, -1})
  end
  wait_for_produtores(n) do
    receive do
      {:DOWN, _, :process, _, _} ->
                    wait_for_produtores(n - 1)
    end
  end
  produtor(id, primos, buffer_pid) do
    Enum.each(primos)
    send(buffer_pid, {:item, p})
  end
  consumidor(id) do
    receive do
      {:item, valor} -> ... # processa item
    end
  end
```

Código 19: Produtor-Consumidor em Elixir com canal distribuidor.

```
defmodule ProdutorConsumidorFlow do
  main(args) do
    fluxo =
      Flow.from_enumerable(1..n_prod)
      |> Flow.flat_map(fn produtor_id ->
        Enum.map(primos, fn p ->
          {produtor_id, p}
        end) ++ [{:fim_produtor, produtor_id}]
      end)
      |> Flow.partition(stages: n_cons)
      |> Flow.map(fn
        {:fim_produtor, _} -> :ignore
        {_, item} -> consumidor = phash(item, n_cons)
                     ... # processa item
      end)
    fluxo
    |> Enum.reduce(%{}, fn
         :ignore, acc -> acc
         consumidor, acc ->
             Map.update(acc, consumidor, 1, &(&1 + 1))
       end)
    Enum.each(1..n_cons, fn id ->
      ...
    end)
  end
end
```

Código 20: Produtor-Consumidor em Elixir com fluxo concorrente.

concorrente, paralela ou distribuída devido ao elevado grau de concorrência recursiva que o problema expõe e, também, ao desbalanceamento regular da carga (o custo para calcular $Fibonacci(n-1)$ é maior que o custo para calcular $Fibonacci(n-2)$). Nas implementações realizadas, o programa recebe como parâmetro o valor n que corresponde à posição da série a ser calculada. Deve ser observado que, diferente do exemplo Produtor/Consumidor, o algoritmo de Fibonacci, a princípio, não necessita compartilhamento de memória, sendo a comunicação entre as unidades de trabalho naturalmente modeladas em parâmetros de entrada e retorno de e para funções.

3.8.4. Fibonacci em C++

Nesta linguagem são apresentadas duas implementações. A primeira utiliza criação assíncrona de tarefas e a segunda o mecanismo de cancelamento coletivo com apoio de variável de condição.

As duas implementações realizadas adotam estratégias bastante distintas quanto ao controle e à criação de threads. A versão baseada em `std::async` delega à biblioteca padrão a responsabilidade pela criação e agendamento dos threads, utilizando

`std::future` para recuperar os resultados das computações paralelas. Essa abordagem oferece simplicidade de uso e semântica direta, mas limita o controle sobre o número total de threads ativos e pode provocar sobrecarga do sistema para valores maiores de entrada, já que não há reutilização de recursos ou coordenação centralizada. Em contraste, a implementação baseada em `std::jthread` organiza a execução concorrente por meio de uma fila de tarefas compartilhada e um conjunto fixo de threads gerenciados manualmente, evitando a criação explosiva de `threads` recursivas. Essa solução adota uma arquitetura de *thread pool*, utilizando sincronização explícita com `std::mutex`, `std::condition_variable` e contadores atômicos, o que permite um uso mais eficiente dos recursos e maior previsibilidade no escalonamento das tarefas.

A primeira implementação, apresentada no Código 21, utiliza `std::async` com a política `std::launch::async`, indicando lançamento da execução assíncrona imediata de uma função em um novo thread. Cada chamada recursiva gera duas novas chamadas concorrentes, encapsuladas como `futures`. A sincronização se dá pela invocação de `get()` sobre cada `future`, o que bloqueia a execução até que o valor esteja disponível.

```cpp
unsigned long long fib(int n) {
  if (n <= 1) return n;
  auto f1 = std::async(std::launch::async, fib, n - 1);
  auto f2 = std::async(std::launch::async, fib, n - 2);
  return f1.get() + f2.get();
}

int main() {
  int n;
  fib(n);
}
```

Código 21: Fibonacci em C++ com tarefas assíncronas.

A segunda versão implementada, Código 22, emprega um conjunto fixo de threads com instanciadas com `std::jthread`, cada thread responsável por processar tarefas enfileiradas em uma estrutura compartilhada. O envio de novas tarefas à fila é feito com o método `enqueue()`, e as tarefas internas realizam chamadas recursivas agendadas, que ao completarem sinalizam a continuidade da computação por meio de funções de retorno (`callbacks`) e contadores atômicos.

3.8.4.1. Fibonacci em Rust

Em Rust, o algoritmo de Fibonacci teve duas implementações. A primeira com criação explícita recursiva de threads. A segunda foi implementada utilizando um pool de threads, suportada por uma *crate* (biblioteca) não oficial da linguagem amplamente aceita pela comunidade (rayon).

A primeira versão desenvolvida, código simplificado em Código 23, é implementada utilizando os recursos básicos de Rust, criando explicitamente threads com `std::thread::spawn`. A sincronização é feita pela invocação ao `join` sobre o thread criado, obtendo o resultado do cálculo do thread.

A segunda versão implementada, Código 24, adota o modelo de composição de tarefas, responsáveis pelo cálculo, submetidas a um pool de execução, suportado pela crate rayon.

```
class ThreadPool {
  std::vector<std::jthread> threads;
  std::queue<std::function<void()>> tasks;
  std::mutex mtx; std::condition_variable cv;
  bool done = false;
  thread_loop(stop_token st) {
    while (!st.stop_requested()) {
      wait_for_task();
      execute_task();
    }
  }
  enqueue(task) {
    lock_guard(mtx); tasks.push(task); cv.notify_one();
  }
  ~ThreadPool() {
    lock_guard(mtx); done = true; cv.notify_all();
  }
}
```

```
fib_parallel(pool, n, callback) {
  shared left, right, pending = 2;

  done = [=]() {
    if (--pending == 0) callback(left + right);
  };
  pool.enqueue([=]() {
    fib_parallel(..., [=](res) {
      left = res;
      done();
    });
  });
  pool.enqueue([=]() {
    fib_parallel(..., [=](res) {
      right = res;
      done();
    });
  });
}
main() {
  int n;
  ThreadPool pool(hardware_concurrency());
  resultado = 0; done = false;
  fib_parallel(pool, n, [&](res) {
    resultado = res; done = true; cv.notify_one();
  });
  wait_until(done);
  print resultado;
```

Código 22: Fibonacci em C++ com tarefas assíncronas.

```
use std::thread;
fn fib(n: u32) -> u64 {
  if n <= 1 {
    return n as u64;
  }
  let t1 = thread::spawn(move || fib(n - 1));
  let t2 = thread::spawn(move || fib(n - 2));
  t1.join().unwrap() + t2.join().unwrap()
}
```

```
fn main() {
  n
  let res = fib(n);
}
```

Código 23: Fibonacci em Rust com criação recursiva de threads.

3.8.4.2. Fibonacci em Go

Em Go foram implementadas duas versões para o cálculo de Fibonacci. A primeira utilizando uma área de memória compartilhada, a segunda exclusivamente canais.

A primeira implementação apresentada, Código 25, adota uma abordagem baseada no uso de `sync.WaitGroup` para coordenar a conclusão das goroutines e acesso atômico no acesso à variável compartilhada que acumula o resultado. Esta variável compartilhada é instanciada na função `main`, e seu endereço é passado recursivamente a todas as goroutines criadas. Cada chamada recursiva, ao concluir seu cálculo, atualiza essa variável de forma segura, usando o mutex para evitar condições de corrida. O WaitGroup garante que todas as chamadas concorrentes terminem antes que o programa principal continue, assegurando a consistência do valor final.

A segunda implementação, Código 26, é mais idiomática para a linguagem Go. Ela também utiliza uma estratégia recursiva, mas utiliza exclusivamente canais para comunicação entre as goroutines, instanciadas para os cálculos para execução assíncrona,

```
fn fib(n) {
  if n <= 1 { return n; }
  (a, b) = rayon::join(
    || fib(n - 1, threshold),
    || fib(n - 2, threshold)
  )
  return a + b
}
```

```
main() {
  n
  resultado = fib(n)
}
```

Código 24: Fibonacci em Rust com pool de threads.

```
var mu sync.Mutex
func fib(n, res *uint64, wg *sync.WaitGroup) {
  defer wg.Done()
  if n <= 1 {
    mu.Lock()
    *res += uint64(n)
    mu.Unlock()
    return
  }
  var r1, r2 uint64
  var wgInner sync.WaitGroup
  wgInner.Add(2)
  go fib(n-1, &r1, &wgInner)
  go fib(n-2, &r2, &wgInner)
  wgInner.Wait()
  mu.Lock()
  *res += r1 + r2
  mu.Unlock()
}
```

```
func main() {
  var res uint64
  var wg sync.WaitGroup
  wg.Add(1)
  fib(n, &res, &wg)
  wg.Wait()
}
```

Código 25: Fibonacci em Go com WaitGroup e Mutex.

mais especificamente, do envio do retorno do dado calculado pela goroutine. A função principal inicia a computação em uma goroutine e aguardando o resultado final por meio do canal principal.

```
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
}
```

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

Código 26: Fibonacci em Go com sincronização via canais.

3.8.4.3. Fibonacci em Elixir

Em Elixir foram produzidas duas versões para cálculo de Fibonacci. A primeira verão explora os mecanismos básicos do modelo de concorrência de Elixir, processos leves e

troca de mensagens. A segunda abstrai o uso direto de processos e mensagens, utilizando tarefas assíncronas.

A primeira implementação, Código 27, explora recursos oferecidos pelo modelo básico de concorrência da linguagem. São criados dois processos leves com `spawn`, um para cada chamada recursiva, e a comunicação entre esses processos é realizada por meio de troca de mensagens `send`/`receive`.

```
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 27: Fibonacci em Elixir com troca de mensagens explícitas.

A segunda implementação, Código 28, utiliza o módulo `Task`, de forma a abstrair a criação de processos e a troca de mensagens, lançando a execução de tarefas assíncronas responsáveis pelo cálculo. Cada chamada recursiva é efetivada com uma chamada à `Task.async`, sendo os resultados são coletados com `Task.await`.

```
defmodule Fib do
  def compute(n) when n <= 1, do: n
  def compute(n) do
    task1 = Task.async(fn -> compute(n - 1) end)
    task2 = Task.async(fn -> compute(n - 2) end)
    res1 = Task.await(task1)
    res2 = Task.await(task2)
    res1 + res2
  end
defmodule Main do
  def main(_) do
    n =
    result = Fib.compute(n)
  end
end
Main.main([])
```

Código 28: Fibonacci em Elixir com tarefas assíncronas.

3.9. Considerações Finais

Este material apresentou uma visão comparativa dos principais recursos de programação concorrente oferecidos pelas linguagens C++, Rust, Go e Elixir. Cada uma dessas linguagens adota modelos distintos para expressar a concorrência. Enquanto C++ privilegia o controle explícito por meio de threads nativos e mecanismos como `std::mutex` e `std::atomic`, Rust promove segurança por construção, com verificações em tempo de compilação que restringem o compartilhamento inseguro de dados. A linguagem Go posiciona-se com um modelo de programação que explora canais de comunicação entre `goroutines` integradas ao seu *runtime*, enquanto Elixir estrutura a concorrência com

base no modelo de atores, promovendo isolamento entre processos leves e troca assíncrona de mensagens.

As diferenças entre essas abordagens não se limitam à sintaxe ou às bibliotecas disponíveis, mas refletem decisões fundamentais de projeto, moldando a forma como os programas concorrentes são estruturados, sincronizados e executados. Nesse sentido, o objetivo deste texto não foi eleger uma linguagem ou modelo superior, mas oferecer subsídios para que o leitor compreenda essas escolhas e suas implicações práticas e conceituais. A identificação da linguagem mais adequada para um determinado problema será, em grande parte, resultado da experiência prática acumulada pelo programador, combinada com o conhecimento das características de cada modelo de execução.

O conteúdo aqui apresentado deve ser entendido como um ponto de partida. Em nenhum momento buscou-se extrair o máximo potencial de cada linguagem ou esgotar suas possibilidades. Cabe ao leitor, motivado por interesse e curiosidade, aprofundar-se por meio das referências citadas, da documentação oficial e das comunidades ativas que sustentam o desenvolvimento dessas ferramentas, tendo sempre em mente que a construção de competência em programação multithread requer prática constante, análise crítica e familiaridade com os padrões idiomáticos de cada ambiente.

A programação concorrente permanece como uma das áreas mais desafiadoras e dinâmicas da Computação. Com a crescente adoção de arquiteturas paralelas nos mais diversos campos de aplicação, o domínio das abstrações, modelos e ferramentas aqui discutidos torna-se cada vez mais essencial. Recomenda-se que o aprofundamento seja feito em uma ou algumas poucas linguagens, favorecendo o domínio técnico, mas sempre com atenção à diversidade de opções que surgem e evoluem continuamente. Espera-se que este material contribua para esse processo de aprendizado e fomente investigações futuras.

Referências

[Amdahl 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485.

[Andrews 1999] Andrews, G. R. (1999). *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition.

[Boehm 2005] Boehm, H.-J. (2005). Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6).

[Burns and Wellings 1998] Burns, A. and Wellings, A. (1998). *Concurrency in Ada (2nd ed.)*. Cambridge University Press, USA.

[Cavaleiro and Du Bois 2014] Cavaleiro, G. G. H. and Du Bois, A. R. (2014). Ferramentas modernas para programação multicore. In Salgado, A. C., Lóscio, B. F., Alchieri, E., and Barreto, P. S., editors, *Atualizações em Informática 2014*, volume 1, pages 41–83. SBC, Porto Alegre.

[Cavaleiro and Santos 2007] Cavaleiro, 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.

[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. ditora e Gráfica Universitária - PREC UFPel, Pelotas.

[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.

[Culler et al. 1999] Culler, D. E., Singh, J., and Gupta, A. (1999). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann.

[Gupta et al. 2021] Gupta, M., Bhargava, L., and Indu, S. (2021). Mapping techniques in multicore processors: current and future trends. *The Journal of Supercomputing*, 77(8):9308–9363.

[Herlihy et al. 2020] Herlihy, M., Shavit, N., Luchangco, V., and Spear, M. (2020). *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2nd edition.

[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.

[Marx et al. 2018] Marx, B., Tate, B., and Valim, J. (2018). *Adopting Elixir: From Concept to Production*. Pragmatic Bookshelf, Dallas, TX.

[McCool et al. 2012] McCool, M., Robison, A. D., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers, an imprint of Elsevier.

[Pacheco 2022] Pacheco, P. (2022). *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2st edition.

[Pilla et al. 2009] Pilla, M. L., Santos, R. R., and Cavaleiro, G. G. H. (2009). Introdução á programação para arquiteturas multicore. In Dorneles, R. V. and Stein, B., editors, *IX Escola Regional de Processamento de Alto Desempenho*, pages 71–202. SBC, Porto Alegre.

[Rauber and Rünger 2010] Rauber, T. and Rünger, G. (2010). *Parallel Programming for Multicore and Cluster Systems*. Springer.

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

[Silva et al. 2022] Silva, G., Bianchini, C., and Costa, E. (2022). *Programação Paralela e Distribuída com MPI, OpenMP e OpenACC para computação de alto desempenho*.

[Skillicorn and Talia 1998] Skillicorn, D. B. and Talia, D. (1998). Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169.

[Sutter and Larus 2005] Sutter, H. and Larus, J. (2005). Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry. *Queue*, 3(7):54–62.

[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.

[Thomas 2022] Thomas, D. (2022). *Programming Elixir 1.6: Functional |> Concurrent |> Pragmatic |> Fun*. Pragmatic Bookshelf, Raleigh, North Carolina, 7 edition.

[Troutwine 2018] Troutwine, B. L. (2018). *Hands-On Concurrency with Rust: Confidently build memory-safe, parallel, and efficient software in Rust*. Packt, Birmingham.

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