Programação Multithread: Modelos e Abstrações em Linguagens Contemporâneas

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

45º Congresso da Sociedade Brasileira de Computação

45ª Jornadas de Atualização em Informática

Maceió/AL, Brasil, 23 de julho de 2025.

Momento Cultural

- **Origem:** Maceió nasceu de um engenho de açúcar no século XVII, crescendo em torno de um porto natural.
- **Nome:** De origem tupi, "Maceió" significa "água que brota do chão", em referência aos olhos d'água da região.
- **Capitania:** A região era parte da capitania de Pernambuco até 1817, quando Alagoas tornou-se capitania autônoma.
- **Domínio holandês:** Alagoas foi invadida pelos holandeses durante as Guerras do Brasil (1630-1645).
- **Quilombo dos Palmares:** Maior símbolo de resistência negra à escravidão, ativo por quase um século no território alagoano.
- **Confederação do Equador (1824):** Movimento republicano que teve repercussão em Alagoas.

Momento Cultural (Cont.)

- **Graciliano Ramos (1892-1953):**
  - Um dos maiores escritores do modernismo brasileiro.
  - Autor de *Vidas Secas*, nascido em Quebrangulo, viveu em Palmeira dos Índios e Maceió.
- **Lêdo Ivo (1924-2012):**
  - Poeta, romancista e jornalista; membro da Academia Brasileira de Letras.
  - Sua obra aborda o mar, a cidade e as contradições do país.
- **Hermeto Pascoal (1936-):**
  - Multi-instrumentista e compositor experimental, nascido em Lagoa da Canoa.
  - Referência mundial no jazz e na música instrumental brasileira.

E, claro, tem Djavan, CRB e CSA.

Sumário

Objetivo do Material

Este material visa apoiar o estudo da programação concorrente e multithread utilizando ferramentas contemporâneas. A proposta combina abordagem teórica e exemplos práticos, explorando fundamentos, modelos e estratégias de multithreading em C++, Rust, Go e Elixir.

Organização do Conteúdo

- Fundamentos da programação multithread.
- Estratégias de implementação e terminologia.
- Análise das linguagens:
  - C++: threads nativos e mutexes.
  - Rust: segurança de memória e sincronização segura.
  - Go: goroutines e canais.
  - Elixir: modelo de atores e tolerância a falhas.
- Casos de estudo e considerações finais.

Fundamentos

Multicores e Software Concorrente

- Processadores multicore estão em todos os dispositivos.
- Desempenho não é automático: o software precisa explorar o paralelismo.
- Programação concorrente tornou-se central no desenvolvimento moderno.

Motivações para o Multithreading

- Superar os limites físicos de frequência dos processadores.
- Exigir maior desempenho em tempo de execução.
- Responder a múltiplas tarefas simultaneamente.

Concorrência e Paralelismo

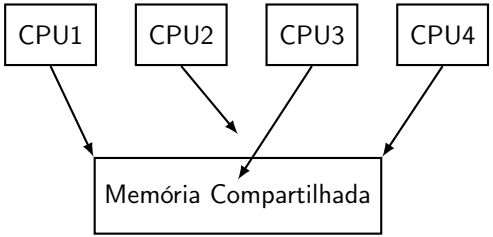
- **Concorrência**: tarefas em progresso (interleaving ou simultâneo).
- **Paralelismo**: execução simultânea de múltiplas tarefas.
- Todo programa paralelo é concorrente, mas o inverso não é verdade.

Aplicações Concorrentes

- Sistemas embarcados: controle de sensores e atuadores.
- Servidores web: múltiplas conexões simultâneas.
- Computação científica: redução de tempo por paralelismo.

Arquiteturas Multiprocessadas

- **UMA**: acesso uniforme à memória.
- **NUMA**: acesso não uniforme; favorece escalabilidade.



Processos e Threads

- Processo: espaço de endereçamento independente.
- Thread: fluxo leve que compartilha memória com outros do mesmo processo.
- Mais baratos de criar e agendar.

Pthreads e Threads Nativos

- `Pthreads`: padrão suportado nativamente em sistemas modernos.
- Base de muitas linguagens e bibliotecas de concorrência.
- Threads são manipulados via identificadores explícitos.

Exemplo Pthreads

```c
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
  printf("Olá do thread!\n");
  return NULL;
}

int main() {
  pthread_t t;
  pthread_create(&t, NULL, hello, NULL);
  pthread_join(t, NULL);
  return 0;
}
```

Suporte ao Multithreading no Hardware

- Instruções atômicas: `compare-and-swap`, `test-and-set`, ...
- Coerência de cache e barreiras de memória.
- Fundamentais para sincronização eficiente.

Relembrando o modelo de von Neumann

- **Instrução: a unidade de trabalho**
  Cada instrução executa uma operação específica – é o tijolo básico da computação sequencial.
- **Memória: o canal de comunicação**
  O efeito colateral da execução de uma instrução é a **escrita de seu resultado** na memória (RAM, registrador ou flag).
- **Sequência: a base da sincronização**
  A ordem em que as instruções aparecem no código **impõe a sincronização**: uma instrução só executa após o término da anterior, garantindo que os dados estejam disponíveis.

E na programação concorrente/paralela?

Ainda que a **concorrência a nível de instrução** seja atualmente explorada com considerável sucesso por compiladores e pelo hardware – com apoio de técnicas de *escalonamento de instruções* – em níveis de granularidade mais grossa (blocos, funções, métodos), o papel do **programador** torna-se determinante.

Isso ocorre porque não existe uma abstração universal para a programação concorrente/paralela.

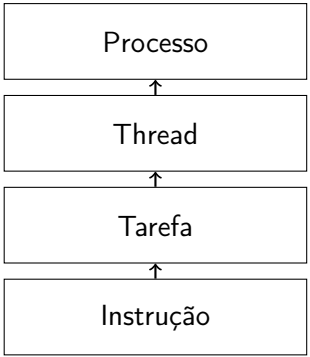
As ferramentas de programação devem oferecer mecanismos suficientes para:

- especificação e manipulação de **unidades de trabalho**,
- promoção da **comunicação**,
- e garantia da **sincronização**.

Unidades de Trabalho

Em diferentes níveis:

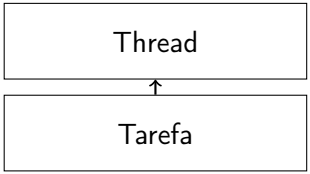
- **Processo**: entidades independentes, comunicando e sincronizando explicitamente via mecanismos oferecidos pelo sistema operacional ou por ferramentas de programação, com ou sem memória compartilhada.
- **Thread**: fluxo leve de controle; threads compartilham o espaço de memória e os recursos de um mesmo processo.
- **Tarefa**: unidade lógica de trabalho, geralmente agendada e executada no contexto de threads.
- **Instrução**: unidade mínima de trabalho no nível da CPU.



Unidades de Trabalho

O que interessa neste curso:

- **Thread**: fluxo leve de controle; threads compartilham o espaço de memória e os recursos de um mesmo processo.
- **Tarefa**: unidade lógica de trabalho, geralmente agendada e executada no contexto de threads.



Comunicação

- **Memória compartilhada**
  - Leitura/escrita em variáveis em um mesmo espaço de endereçamento.
  - Inclui uso de buffers, estruturas globais e pilha.
- **Troca de mensagens**
  - Envio e recebimento explícito de dados entre unidades concorrentes.
  - Base de modelos como canais e **modelo de atores**.
- **Futuros e promessas**
  - Comunicação indireta via valores que serão disponibilizados no futuro.

Mecanismos de Sincronização

- **Exclusão mútua e controle de fluxo**
  - Garantem acesso coordenado a regiões críticas e ordenação da execução.
  - Incluem travas (mutexes), semáforos, barreiras e sinais de condição.
- **Canais**
  - Mecanismo de troca de mensagens entre unidades concorrentes.
  - A sincronização está embutida: o envio e o recebimento só ocorrem quando ambas as partes estão prontas.
- **Futuros e promessas**
  - Representam valores que ainda não estão disponíveis, mas que serão produzidos por outra unidade de execução.
  - A sincronização ocorre no momento da obtenção do resultado, que pode bloquear até o valor estar pronto.

Exclusão mútua: exemplo com pthread_mutex_t

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* tarefa(void* arg) {
    pthread_mutex_lock(&lock);

    contador_global++;

    pthread_mutex_unlock(&lock);
    return NULL;
}
```

O mutex garante que apenas um thread execute a região crítica por vez.

Comunicação por troca de mensagens: exemplo com MPI

Sender (Processo 0):

```
int valor = 313;
MPI_Send(&valor, 1, MPI_INT,
         1, 0, MPI_COMM_WORLD);
```

Receiver (Processo 1):

```
int recebido;
MPI_Recv(&recebido, 1, MPI_INT,
         0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

A sincronização está embutida: a comunicação só ocorre quando ambas as partes estão prontas.

Comunicação por valor futuro: *future/promise*

Chamador:

```
promessa = criar_promessa()
lançar_tarefa(execução, promessa)
...
valor = aguardar(promessa)
usar(valor)
```

Executor:

```
função execução(promessa):
    resultado = computar()
    cumprir_promessa(promessa,
                     resultado)
```

O valor é produzido de forma assíncrona e lido futuramente, com sincronização implícita.

Granularidade das Unidades de Trabalho

- **Granularidade fina**: *tarefas* ou threads executam blocos muito pequenos de trabalho; isso aumenta a sobrecarga de gerenciamento (criação, agendamento, sincronização).
- **Granularidade grossa**: tarefas ou *threads* executam blocos grandes de trabalho; isso reduz a sobrecarga, mas pode limitar o aproveitamento do paralelismo.
- O desempenho paralelo depende do equilíbrio entre essas escolhas; é preciso ajustar a granularidade para maximizar o ganho, considerando os limites impostos pela parte sequencial do programa (Lei de Amdahl).

Granularidade das Seções Críticas

- **Granularidade grossa**: um único *mutex* protege uma região crítica ampla, envolvendo vários recursos ou operações.
- **Granularidade fina**: diferentes *mutexes* são usados para proteger recursos ou operações distintos.
- Granularidade fina permite maior paralelismo, mas aumenta a complexidade do código e os riscos de erros, como impasses.

Granularidade na Comunicação e Sincronização

- **Canais**: comunicações frequentes com pequenos dados aumentam o custo por interação; comunicações mais espaçadas com blocos maiores podem reduzir a sobrecarga.
- **Futures e promessas**: esperar por resultados muito pequenos ou de curta duração pode não compensar o custo da coordenação.
- A escolha da granularidade afeta o número de interações, a latência e a eficiência geral da aplicação.

Comparando Granularidades

Granularidade das Unidades de Trabalho	**Granularidade na Comunicação e Sincronização**
Define quanto trabalho cada thread ou tarefa executa antes de terminar ou sincronizar.	Define a frequência e o escopo das interações e dos controles de acesso entre unidades concorrentes.
A escolha afeta o aproveitamento do paralelismo e a sobrecarga de criação/agendamento.	A escolha afeta a latência, o custo de sincronização e o grau de contenção.
Exemplo: dividir um loop em 4 ou 100 tarefas.	Exemplo: usar um mutex único para vários dados (granularidade grossa) ou um mutex por recurso (fina); comunicar a cada iteração ou apenas ao final.
Envolve abstrações como tarefas e threads.	Envolve mutexes, barreiras, canais, futures/promises.

As duas formas de granularidade impactam o desempenho e devem ser ajustadas em conjunto.

Problemas Clássicos na Programação Concorrente

- **Condição de corrida**: resultado incorreto surge quando múltiplas unidades acessam e modificam um dado compartilhado sem coordenação.
  - Ocorre tipicamente em memória compartilhada onde os mecanismos de exclusão mútua não foram utilizados ou foram utilizados incorretamente.
- **Deadlock**: espera circular entre unidades que aguardam recursos uns dos outros.
  - Pode ocorrer com múltiplos mutexes, canais cruzados ou dependências encadeadas entre promessas.
- **Starvation**: um thread ou tarefa nunca recebe oportunidade de executar.
  - Pode ocorrer por prioridades desbalanceadas ou recursos constantemente ocupados por outros.

Panorama das Ferramentas

- `Pthreads`: API de baixo nível baseada em threads nativos.
- Ada: tarefas e *rendezvous*.
- OpenMP: diretivas para paralelismo estruturado.

Modelos Contemporâneos

C++

- Threads nativos com `std::thread`.
- Sincronização com `mutex`, `lock`, condição.
- Comunicação por memória compartilhada.
- Assíncrono com `std::async`, `future`.
- Cancelamento cooperativo com `std::jthread`.

Go

- Goroutines leves controladas pelo runtime.
- `sync.Mutex`, `RWMutex`.
- Comunicação via canais (`chan`).
- Coordenação com `context`, `WaitGroup`.
- Otimização com `sync.Pool`.

Rust

- Threads com `thread::spawn`, ownership garantido.
- `Mutex`, `RwLock`, `Arc`.
- Comunicação com canais (`mpsc`, `crossbeam`).
- `async/await`, executores como Tokio.
- Paralelismo de dados com `rayon`.

Elixir

- Processos leves (atores) com `spawn`.
- Isolamento e supervisores para resiliência.
- Comunicação por mensagens assíncronas.
- Abstrações: `Task`, `Agent`, `GenServer`.
- Reatividade com `receive/after` e `timeout`.

Implementações e Modelos

Implementações e Modelos

Para comparar diferentes ferramentas de programação concorrente, é necessário:

- Uniformizar a nomenclatura e as abstrações adotadas;
- Identificar os modelos de implementação subjacentes (1×1, N×M);
- Compreender como as ferramentas aplicam e denominam abstrações como *threads*, *tarefas*, *canais* e *atores*; e,
- Reconhecer os paradigmas de programação favorecidos por cada linguagem/

A análise foca em C++, Rust, Go e Elixir, destacando os mecanismos oferecidos para criação, comunicação e sincronização entre múltiplas unidades de execução.

Modelos de Implementação

- **1×1 (thread-sistema)**: cada unidade de concorrência corresponde a um thread do sistema operacional.
  - Ex: Pthreads, C++, Rust.
- **N×M (tarefas sobre threads)**: múltiplas tarefas de usuário são mapeadas sobre um número menor de threads do sistema.
  - Ex: OpenMP, Go, Elixir.
- O modelo N×1 (obsoleto) foi abandonado com o fim das plataformas monoprocessadas.

Comparativo: Modelos 1×1 e N×M

Aspecto	Modelo 1×1	Modelo N×M
Unidade	Thread	Tarefa
Custo de criação	Maior (chamada ao SO)	Menor (espaço de usuário)
Escalonamento	Sistema operacional	Ambiente de execução
Escalabilidade	Limitada	Alta (milhares de tarefas)
Exemplos	Pthreads, C++, Rust	Go, Elixir, OpenMP

Modelos de Programação Concorrente

- **Memória compartilhada**: threads acessam dados em espaço comum.
  - Ex: `thread + lock`, paralelismo de dados, paralelismo de tarefas.
- **Troca de mensagens**: tarefas se comunicam por envio e recepção explícita.
  - Ex: canais (Go, Rust), modelo de atores (Elixir).
- **Abordagem funcional**: pipelines de dados e composição de funções puras.
  - Ex: `Flow` (Elixir), `Parallel STL` (C++), `rayon` (Rust).

Panorama das Linguagens

Aspecto	C++	Rust
Unidade	Thread nativa	Thread nativa / Tarefa assíncrona
Comunicação	Memória compartilhada	Memória com propriedade
Sincronização	Mutex, locks	Mutex + verificação de tipo
Mapeamento	Manual	Automático
	Go	**Elixir**
Unidade	Goroutine	Processo leve
Comunicação	Canais / memória	Troca de mensagens
Sincronização	Canais, mutex opcional	Mailbox + isolamento
Mapeamento	Automático	Supervisionado

C++

Evolução do Suporte a Threads em C++

- C++ **não** apresenta uma camada sobre Pthreads
  - O modelo de thread faz parte da linguagem.
  - Ortogonalidade com o sistema de tipos: nenhuma necessidade de `void*`, como ocorre em Pthreads.
- Suporte nativo a programação concorrente desde o padrão C++11.
  - `std::thread` integra o modelo de objetos da linguagem.
- C++14, C++17 e C++20 ampliaram o suporte
  - Sincronização, tarefas, algoritmos paralelos.

RAII: Gerenciamento Seguro de Recursos

- RAII significa *Resource Acquisition Is Initialization*.
- É um padrão de projeto fundamental em C++.
- Um recurso é adquirido no construtor e liberado no destrutor.
- Isso garante que o recurso seja corretamente liberado:
  - Mesmo se ocorrer uma exceção.
  - Mesmo se houver retorno antecipado.
- **Aplica-se a memória, arquivos, conexões etc.**
- Não introduz overhead significativo, pois o objeto RAII encapsula o controle do recurso sem adicionar lógica extra em tempo de execução.

RAII: Exemplo Básico

```
class Arquivo {
  std::ofstream f;
public:
  Arquivo(const std::string& nome) : f(nome) {
    std::cout << "Arquivo aberto.\n";
  }
  ~Arquivo() {
    f.close();
    std::cout << "Arquivo fechado.\n";
  }
};
void exemplo() {
  Arquivo a("dados.txt");
  // Uso do arquivo...
  if (erro()) return; // o destrutor ainda será chamado
}
```

- O arquivo é fechado automaticamente ao final do escopo.
- O programador não precisa se lembrar de fechar.

RAII em Programação Concorrente

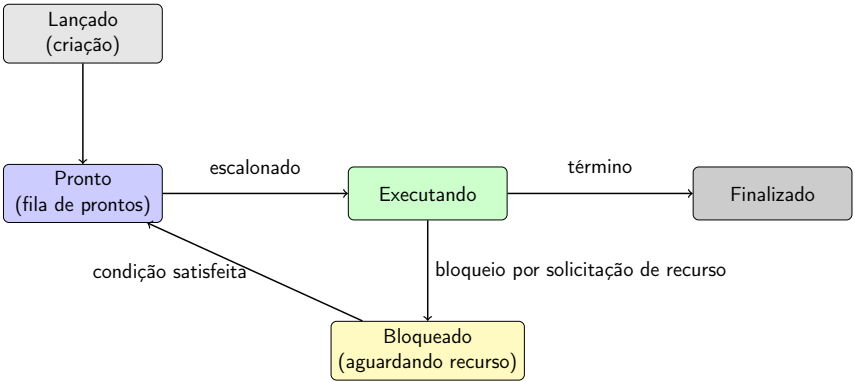
- O padrão RAII é fundamental para o uso seguro de threads e sincronizadores.
- Garante que mutexes sejam liberados mesmo em caso de exceções.
- Evita erros como deadlocks e uso de threads não finalizados.
- **Aplicações típicas em concorrência:**
  - `std::thread` (evita `std::terminate()`).
  - `std::lock_guard`, `std::unique_lock`.
  - `std::scoped_lock`, `std::shared_lock`.
- O RAII reduz significativamente a complexidade de programas concorrentes robustos.

Modelo Básico de Criação de Threads

- `std::thread` recebe um objeto invocável: função, functor ou lambda.
- O thread inicia automaticamente na construção do objeto.

```
void foo() { std::cout << "Função foo.\n"; }
class Functor {
public:
  void operator()() { std::cout << "Functor.\n"; }
};
int main() {
  Functor f;
  std::thread t1(foo);  // O invocável é uma função
  std::thread t2(f);    // O invocável é um o método operator() do objeto
  std::thread t3([]() { // O invocável é uma função lambda
                        std::cout << "Lambda.\n";
                      });
  ...
  t1.join(); t2.join(); t3.join();
}
```

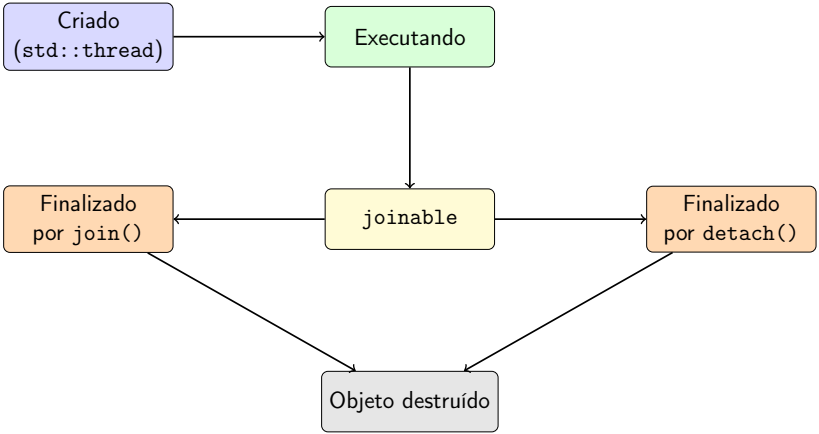
Ciclo de Escalonamento de Threads



Controle e Ciclo de Vida dos Threads

- O objeto `std::thread` representa um thread em execução.
- Um thread pode ser:
  - **Sincronizado** com `join()` (bloqueia até o fim),
  - **Dissociado** com `detach()` (execução autônoma).
- Cada thread pode ser sincronizado apenas uma vez com `join()`.
- Se o objeto for destruído sem `join()` ou `detach()`, o programa é encerrado com `std::terminate()`.
- RAII recomenda: sempre finalize o thread antes da destruição do objeto.

Ciclo de Vida de um Thread em C++



Compartilhamento de Dados entre Threads

- Em C++, threads compartilham variáveis no mesmo espaço de memória.
- Isso permite comunicação eficiente (por acessos em leitura e escrita em variáveis compartilhadas), mas requer atenção:
  - Leitura e escrita simultâneas podem causar resultados imprevisíveis.
  - Condições de corrida são difíceis de reproduzir e depurar.
- C++ oferece mecanismos distintos para proteger o acesso a dados:
  - **Exclusão mútua** com `std::mutex` – controle explícito de regiões críticas.
  - **Operações atômicas** com `std::atomic` – proteção de uma única *palavra* de dados.
- Ambos os mecanismos mapeiam diretamente para instruções eficientes das arquiteturas modernas.

Operações Atômicas com `std::atomic`

- `std::atomic` atua sobre tipos de dados representados por uma única palavra de máquina, como inteiros ou ponteiros.

```
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() << "\n";
else
  std::cout << "Falha, valor = " << valor.load()
            << ". Esperado: " << esperado << "\n";
```

- Indivisibilidade evita condições de corrida.
- Adequado para contadores, flags, algoritmos *lock-free*.

Exclusão Mútua com `std::mutex` (sem RAII)

- A exclusão mútua permite proteger regiões críticas contra acessos concorrentes.
- O controle explícito exige chamadas a `lock()` e `unlock()`.
- Erros como esquecer o `unlock()` podem causar deadlocks.

```
std::mutex m;
int dado = 0;

void operacao() {
  m.lock();
  dado++; // Região crítica
  m.unlock();
}
```

Exclusão Mútua com RAII (`std::lock_guard`)

- RAII automatiza o controle do mutex: bloqueia no construtor e libera no destrutor.
- Elimina o risco de esquecer o `unlock()`, mesmo com exceções ou retornos antecipados.
- `std::lock_guard` é ideal para escopos simples.

```
std::mutex m;
int dado = 0;

void operacao() {
  std::lock_guard<std::mutex> guard(m);
  dado++; // Região crítica protegida
} // m é desbloqueado automaticamente aqui
```

Leitura e Escrita Concorrentes com shared_mutex

- std::shared_mutex permite:
  - Múltiplas threads leitoras simultâneas.
  - Escrita exclusiva: apenas um thread pode escrever.
- O tipo de acesso é indicado pela trava usada:
  - std::shared_lock para leitura.
  - std::unique_lock para escrita.
- Útil quando há muito mais leitura que escrita.

```
std::shared_mutex smutex;
void leitor() {
  std::shared_lock lock(smutex); // leitura simultânea
}
void escritor() {
  std::unique_lock lock(smutex); // escrita exclusiva
}
```

Travamentos Combinados e Reentrância

- `std::scoped_lock` permite travar múltiplos mutexes de forma segura:
  - Evita deadlocks causados por ordens diferentes de aquisição.
  - Ideal para proteger múltiplos recursos simultaneamente.
- `std::recursive_mutex` permite que um mesmo thread adquira o mesmo mutex várias vezes:
  - Útil em chamadas reentrantes ou funções recursivas.
  - Deve ser usado com cautela para evitar inconsistências lógicas.

```
std::mutex a, b;
void f() {
  std::scoped_lock lock(a, b); // travamento múltiplo
  ...
}

std::recursive_mutex r;
void rec() {
  r.lock();
  rec(); // chamada reentrante
  r.unlock();
}
```

Observações sobre `std::async`

- Esconde o controle direto de threads, promovendo estilo mais declarativo.
- Políticas de execução afetam o comportamento:
  - `launch::async`: executa em um novo thread.
  - `launch::deferred`: executa apenas quando `get()` é chamado.
- **Implementações variam:**
  - GCC/Clang: criam threads dedicados (sem thread pool).
  - MSVC: usa thread pool para eficiência.
- Sincronização com `get()` (bloqueante) ou `wait()`.

Tarefas Assíncronas em C++

- Executar tarefas em background pode ser feito sem gerenciar diretamente threads.
- `std::async` lança a execução de uma função e retorna imediatamente um objeto `std::future`.
- O `std::future` representa o resultado da computação que será produzido no futuro.
- A chamada a `get()` em `future` sincroniza com a conclusão da tarefa e recupera o valor.
- **Comparando com** `std::thread`**:**
  - `std::thread` requer controle manual de ciclo de vida (join/detach).
  - `std::async` automatiza o gerenciamento e fornece sincronização segura via `future`.
  - Segue um modelo orientado a tarefas, mais expressivo e robusto.

Programação Assíncrona com `std::async` e `std::future`

```cpp
int resposta() {
  calcule_7_5_milhoes_de_anos();
  return 313;
}

int main() {
  std::future<int> f = std::async(std::launch::async, resposta);
  int resultado = f.get(); // bloqueia até que a tarefa termine
  std::cout << "Resultado: " << resultado << "\n";
}
```

- O retorno de `std::async` é sempre um `std::future`.
- O `future` encapsula o valor de retorno e o ponto de sincronização.
- A função associada é executada em segundo plano, e a sincronização ocorre via `get()`.

Algoritmos Paralelos na STL

- Introduzidos no C++17, com suporte expandido no C++20.
- Permitem aplicar algoritmos padrão (ex: `for_each`, `reduce`, `sort`, ...) com paralelismo automático.
- Baseados em políticas de execução que guiam o paralelismo:
  - `std::execution::seq` – execução sequencial.
  - `std::execution::par` – execução paralela.
  - `std::execution::par_unseq` – paralela + vetorização.
- O programador não precisa controlar threads nem sincronização.

Exemplos com Algoritmos Paralelos

```cpp
// Eleva cada elemento ao quadrado
std::for_each(std::execution::par, v.begin(), v.end(), [](int& x) {
  x *= x;
});

// Encontra o maior elemento
int maior = std::reduce(std::execution::par, v.begin(), v.end(), INT_MIN,
                        [](int a, int b) { return std::max(a, b); });
```

- Os algoritmos executam operações em paralelo sem controle manual de threads.
- `std::for_each`: ideal para transformações locais sem efeitos colaterais.
- `std::reduce`: alternativa paralela a `accumulate`, com associatividade exigida.
- O uso da política `par_unseq` permite vetorização adicional (dependente do compilador).

Considerações sobre os Algoritmos Paralelos

- A ordem de execução entre as iterações **não é garantida**.
- Recomendado o uso com funções puras ou operações independentes.
- A biblioteca padrão delega o paralelismo à infraestrutura interna:
  - Pode usar criação direta de threads, thread pools, ou bibliotecas como TBB.
- Abordagem declarativa que favorece expressividade e manutenção.

Variáveis de Condição (`std::condition_variable`)

- Permite que um thread espere até que uma condição seja satisfeita.
- Sempre utilizada em conjunto com um `std::mutex` e `std::unique_lock`.
- Libera o mutex durante a espera e o readquire ao acordar.
- A condição deve ser sempre verificada dentro de um laço, devido a possíveis sinais espúrios.
- Útil em padrões como produtor/consumidor, buffers de dados e sistemas reativos.

```
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);
```

Barreiras e Travamentos em C++20

- `std::barrier` permite sincronização cíclica entre múltiplos threads:
  - Todos os threads precisam atingir o ponto de barreira.
  - Só então todos são liberados para continuar.
  - Reutilizável em múltiplas fases.
- `std::latch` permite sincronização de ponto único:
  - Espera até que um número fixo de threads sinalize sua chegada.
  - Não pode ser reutilizado após liberado.
  - Útil para inicialização coletiva ou etapas únicas de preparação.
- Esses mecanismos substituem usos complexos de contadores e mutexes.

Motivação para `std::jthread`

- Introduzido no C++20, `std::jthread` oferece:
  - Gerenciamento automático do ciclo de vida do thread.
  - Cancelamento cooperativo por meio de `stop_token`.
  - Maior segurança e simplicidade na escrita de código concorrente.
- Evita erros comuns com `std::thread`:
  - Esquecimento de `join()` ou `detach()`.
  - Interrupção desordenada de execução.
- Mantém compatibilidade com código moderno baseado em RAII e permite controle explícito do cancelamento.

Cancelamento Cooperativo com `std::jthread`

```cpp
void tarefa(std::stop_token st, int i) { // st é injetado pelo jthread
  while (!st.stop_requested()) {
    ...
  }
  std::cout << "Thread " << i << " cancelada.\n";
}

int main() {
  std::jthread t1(tarefa, 1);
  std::jthread t2(tarefa, 2);
  ...
  t1.request_stop();
  t2.request_stop();
}
```

- O `stop_token` permite verificação periódica do pedido de cancelamento.
- `std::jthread` chama `join()` automaticamente ao sair de escopo.
- Abordagem segura, sem necessidade de controle manual do ciclo de vida.

Sincronização com `std::latch` e `std::jthread`

```cpp
std::latch inicio(3); // espera por 3 threads

void tarefa(std::stop_token, int id) {
  std::cout << "Thread " << id << " preparada\n";
  inicio.arrive_and_wait(); // espera até todas estarem prontas
  std::cout << "Thread " << id << " iniciou execução\n";
}

int main() {
  std::jthread t1(tarefa, 1);
  std::jthread t2(tarefa, 2);
  std::jthread t3(tarefa, 3);
}
```

- Os três threads se sincronizam por meio de `latch`.
- A execução principal de cada thread só começa após todos estarem prontos.
- Combinação segura e moderna de sincronização e cancelamento.

Rust

Rust: Visão Geral

- Primeira versão estável em 2015.
  - Linguagem moderna voltada ao desenvolvimento de sistemas seguros e eficientes.
- Segurança garantida em tempo de compilação.
  - Baseada nos conceitos de propriedade (*ownership*) e regras de empréstimo (*borrowing*).
- Suporte à programação multithread:
  - Biblioteca padrão: `thread`, `Mutex`, tipos atômicos e canais.
  - Bibliotecas de terceiros amplamente adotadas, como `Tokio`, com suporte a `async/await`.
- `Cargo`: ferramenta unificada para gerenciamento de pacotes, builds e dependências.

RAII (alternativa em C++, regra em Rust)

- Assim como em C++, Rust adota o princípio de RAII.
- Recursos (memória, arquivos, locks, etc.) são liberados automaticamente ao final do escopo.
- A liberação é implementada por meio da trait `Drop`.
- O uso de ponteiros inteligentes é desnecessário: todas as variáveis possuem dono único.
- `JoinHandle` também é regido por RAII – se não for `join()`-ado, ele será descartado.
- RAII torna o código concorrente mais seguro, evitando vazamentos e condições de corrida associadas à desalocação manual.

```rust
use std::fs::File;
use std::thread;

fn main() {
    let f = File::create("saida.txt").unwrap();
    writeln!(&f, "Arquivo será fechado ao final.").unwrap();

    let handle = thread::spawn(|| {
        println!("Thread em execução.");
    });

    handle.join().unwrap(); // sincronização explícita
} // f é fechado automaticamente aqui
```

Modelos de Concorrência em Rust

- Mecanismos principais:
  - **Threads nativos**
    - Execução paralela com `thread::spawn`, útil em tarefas de CPU intensivas.
  - **Tarefas assíncronas**
    - Concorrência cooperativa para I/O, escalável com runtimes como `Tokio` utilizando `async/await`.
  - **Canais de comunicação**
    - Transferência segura de dados entre threads ou tarefas (`mpsc::channel`).
- Todos os mecanismos integram-se ao sistema de propriedade (*ownership*) e empréstimo (*borrowing*).
- A segurança de memória é verificada em tempo de compilação, sem uso de coletor de lixo.

Threads nativas com `thread::spawn`

- Cada thread é criada com `thread::spawn`, que recebe um invocável que implemente a trait `FnOnce`.
- Isso inclui funções livres, closures e métodos em structs.
- Invocáveis que capturam variáveis do ambiente devem usar `move`, transferindo a posse para o novo fluxo.
- O retorno é uma `JoinHandle`, que permite sincronização por meio de `join()`.
- O método `join()` consome a `JoinHandle` e retorna um `Result<T>`.
- Cada thread pode ser sincronizada no máximo uma vez.
- Não há suporte a `detach`, como em C++.

Modelo básico de criação de threads

- `thread::spawn` aceita qualquer invocável que implemente `FnOnce`.
- Três formas de definir o código do thread:
  - Função livre.
  - Método em uma struct (como um functor).
  - Closure (lambda).
- A execução do thread inicia imediatamente após `spawn`.
- Cada `JoinHandle` deve ser sincronizada com `join()`.

```
use std::thread;
fn foo() {
    println!("Função foo.");
}
struct Functor;
impl Functor {
    fn call(&self) {
        println!("Método de objeto.");
    }
}
fn main() {
    let f = Functor;
    let t1 = thread::spawn(foo);      // função
    let t2 = thread::spawn(move ||    // método
                           f.call()); // em objeto
    let t3 = thread::spawn(||         // closure
                            println!("Closure."));
    t1.join().unwrap();
    t2.join().unwrap();
    t3.join().unwrap();
}
```

Programação Assíncrona com Tokio

A biblioteca `Tokio` é o runtime assíncrono mais amplamente adotado pela comunidade Rust. Embora seja uma biblioteca de terceiros, é considerada padrão de fato para tarefas assíncronas na linguagem.

- Funções assíncronas (`async fn`) produzem valores que implementam a trait `Future`.
- A execução desses valores exige um *runtime* que coordene o avanço das tarefas.
  - O runtime é um sistema de agendamento cooperativo, baseado em filas de tarefas e poll de eventos.
  - Ele executa múltiplas tarefas assíncronas em poucos threads nativos, com baixo custo de troca de contexto.
- Com `Tokio`, tarefas são lançadas com `task::spawn`, retornando um `JoinHandle`.
- O operador `.await` suspende a tarefa até que o resultado esteja pronto, sem bloquear o thread.
- A anotação `#[tokio::main]` inicializa automaticamente o runtime na função principal.

Exemplo: Tarefa com Tokio

- `task::spawn` lança uma tarefa assíncrona e retorna um `JoinHandle<T>`, onde `T` é o tipo retornado pela tarefa.
- O corpo da tarefa é um bloco `async`, que retorna um valor.
- A chamada `.await` sobre o `JoinHandle` aguarda a conclusão da tarefa e recupera o resultado.
- Esse padrão permite delegar trabalho assíncrono e sincronizar explicitamente com os resultados.

```rust
use tokio::task;

#[tokio::main]
async fn main() {
    let t = task::spawn(async {
        println!("Executando tarefa em paralelo");
        313 // valor retornado
    });

    let res = t.await.unwrap();
    println!("Resultado da tarefa: {}", res);
}
```

Exemplo: Função que Retorna um `Future`

- Uma função marcada como `async` não executa imediatamente, mas retorna um objeto que implementa a trait `Future`.
- Esse objeto encapsula a lógica assíncrona e pode ser aguardado posteriormente com `.await`.
- O tipo exato do `Future` é inferido pelo compilador, mas pode ser inspecionado com ferramentas como `debug-type`.
- Esse modelo permite compor tarefas assíncronas como valores de primeira classe.

```rust
use std::future::Future;

async fn tarefa() -> u32 {
    println!("Executando tarefa assíncrona");
    7
}

fn retorna_future() -> impl Future<Output = u32> {
    tarefa() // retorna o Future, não executa
}

#[tokio::main]
async fn main() {
    let futuro = retorna_future();
    let resultado = futuro.await;
    println!("Resultado: {}", resultado);
}
```

Canais de comunicação em Rust

- Rust fornece canais de comunicação baseados no modelo `mpsc` (múltiplos produtores, um consumidor).
- Um canal é criado com `mpsc::channel()`, que retorna um par `(Sender<T>, Receiver<T>)`.
- O `Sender` pode ser clonado, permitindo que múltiplos threads ou tarefas enviem mensagens ao mesmo `Receiver`.
- O `Receiver` oferece métodos como `recv()` (bloqueante) e `try_recv()` (não bloqueante).
- O mecanismo permite comunicação segura entre fluxos de execução, sem necessidade de bloqueios explícitos.
- Os canais são *thread-safe* por construção e compatíveis com as regras de posse e movimento de dados do Rust.

Exemplo: uso de canal, produtor único

```rust
use std::sync::mpsc;          // Importa o módulo de canais (multi-produtor, único consumidor)
use std::thread;              // Importa suporte a threads

fn main() {
   // Cria um canal, retornando um par (transmissor, receptor)
   let (tx, rx) = mpsc::channel();

   // Cria um thread que enviará uma mensagem para o canal
   let t = thread::spawn(move || {
     let mensagem = String::from("Olá do thread!");
     tx.send(mensagem).unwrap(); // Envia a mensagem pelo canal
   });

   // Bloqueia até receber uma mensagem do canal
   let recebido = rx.recv().unwrap();
   println!("Mensagem recebida: {}", recebido);

   // Aguarda o término do thread
   t.join().unwrap();
}
```

Exemplo: uso de canal, múltiplos produtores

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
fn main() {
    let (tx, rx) = mpsc::channel();
    // Clona o transmissor para permitir múltiplos produtores
    let tx1 = tx.clone();
    let t1 = thread::spawn(move || {
        tx.send(String::from("Mensagem do thread 1")).unwrap();
    });
    let t2 = thread::spawn(move || {
        tx1.send(String::from("Mensagem do thread 2")).unwrap();
    });
    // Recebe duas mensagens do mesmo canal
    for _ in 0..2 {
        let msg = rx.recv().unwrap();
        println!("Recebido: {}", msg);
    }
    ...
    t1.join().unwrap();
    t2.join().unwrap();
}
```

Referência compartilhada com `Arc<T>`

- `Arc<T>` (Atomic Reference Counted) permite compartilhar dados imutáveis entre múltiplos threads.
- Internamente, mantém uma contagem de referências atualizada de forma atômica.
- O dado original só é desalocado quando todas as referências forem descartadas.
- Diferente de `Rc<T>`, que não é seguro para múltiplos threads.
- Uso típico: `Arc<Mutex<T>>` para dados estruturados e `Arc<AtomicT>` para dados escalares (inteiro, float...) com acesso concorrente.
- O método `Arc::clone` incrementa a contagem atômica e retorna uma nova referência segura.

Compartilhamento seguro de dados mutáveis

- Para permitir mutação segura de dados compartilhados com `Arc<T>`, é necessário controle de concorrência.
- `Mutex<T>` e `RwLock<T>` oferecem exclusão mútua e leitura concorrente.
- Tipos atômicos como `AtomicBool`, `AtomicUsize` etc. permitem acesso concorrente sem bloqueios.
- `Arc<Mutex<T>>` e `Arc<AtomicT>` são padrões comuns para sincronização segura entre threads.
- Operações atômicas são ideais para dados escalares e algoritmos sem bloqueio (*lock-free*).

`Arc<T>` fornece apenas acesso imutável a `T`; mutabilidade segura requer que `T` implemente *mutabilidade interior*, como em `Mutex<T>` ou `AtomicT`.

Exemplo: variável atômica com múltiplos threads

- `AtomicI32` permite operações atômicas sobre inteiros de 32 bits.
- Pode ser usado com `Arc` para ser compartilhado entre threads.
- Métodos como `fetch_add` e `fetch_sub` são seguros e não bloqueantes.
- Ideal para contadores, flags e controle leve de estado.
- Evita o uso de `Mutex` quando a atomicidade for suficiente.

```rust
use std::sync::{Arc, atomic::{AtomicI32, Ordering}};
use std::thread;
fn main() {
    let counter = Arc::new(AtomicI32::new(0));
    let inc = {
        let c = Arc::clone(&counter);
        thread::spawn(move || {
            for _ in 0..1000 {
                c.fetch_add(1, Ordering::SeqCst);
            }
        })
    };
    let dec = {
        let c = Arc::clone(&counter);
        thread::spawn(move || {
            for _ in 0..1000 {
                c.fetch_sub(1, Ordering::SeqCst);
            }
        })
    };
    inc.join().unwrap();
    dec.join().unwrap();
    println!("Valor final: {}", counter.load(Ordering::SeqCst));
}
```

Exemplo: uso de Mutex com Arc

```rust
use std::sync::{Arc, Mutex};
use std::thread;
fn main() {
  // Cria contador protegido por Mutex, compartilhado via Arc
  let counter = Arc::new(Mutex::new(0));
  let mut handles = vec![];
  for _ in 0..10 {
    // Clona a referência para o thread
    let counter = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        // Obtém o lock do Mutex e incrementa o contador
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
  }
  // Aguarda todos os threads terminarem
  for handle in handles {
      handle.join().unwrap();
  }
  // Imprime valor final do contador protegido
  println!("Contador: {}", *counter.lock().unwrap());
}
```

Para não dizer que não foi visto rayon

Rust com `rayon`

O exemplo abaixo utiliza o `par_iter()` da biblioteca `rayon` para aplicar paralelismo de dados de forma segura e automática. Está sendo comparado com C++ apenas para caracterizar funcionalidades equivalentes.

```rust
use rayon::prelude::*;

fn main() {
    let v: Vec<u32> = (1..=100_000).collect();

    let soma_quadrados: u32 = v.par_iter()
        .map(|x| x * x)
        .sum();

    println!("Soma dos quadrados: {}", soma_quadrados);
}
```

C++ com Parallel STL

Utiliza `std::execution::par` com `transform_reduce` para obter o mesmo resultado com paralelismo de dados declarativo.

```cpp
#include <execution>
#include <vector>
#include <numeric>
#include <iostream>

int main() {
    std::vector<int> v(100000);
    std::iota(v.begin(), v.end(), 1);

    auto result = std::transform_reduce(
        std::execution::par, v.begin(), v.end(), 0u,
        std::plus<>(), [](int x) { return x * x; });

    std::cout << "Soma dos quadrados: "
              << result << '\n';
}
```

Ferramentas e padrões avançados não abordados

- Estruturas adicionais de sincronização: `Barrier`, `Condvar`, semáforos e filas concorrentes.
- `actix`: modelo de atores com troca de mensagens assíncronas.
- Várias outras bibliotecas para programação concorrente e assíncrona: `tokio`, `crossbeam`, `async-std`, entre outras.

Go

Go: Concorrência com Goroutines e Canais

- Criada no final da década de 2000 com foco em eficiência e concorrência.
- Concorrência baseada em **goroutines**, agendadas sobre threads sistema mantidos pelo runtime.
- Comunicação entre goroutines é feita por **canais**, evitando seções críticas.
- Modelo inspirado em CSP, favorecendo programação concorrente estruturada.
- Biblioteca padrão inclui ferramentas para sincronização, timers, `sync.Pool`, etc.

Modelo Básico

O modelo de concorrência de Go estimula a decomposição de programas em tarefas cooperativas que trocam mensagens, reduzindo a necessidade de memória compartilhada.
$\Longrightarrow$ Inspirado na notação CSP (Communicating Sequential Processes)

Goroutines

- Unidades de execução concorrentes leves, criadas pelo programa e delegadas ao runtime para execução.
- São escalonadas dinamicamente sobre threads sistema mantidos pelo runtime.
  - O escalonador do runtime decide qual thread sistema executará cada goroutine.
  - Pode ocorrer preempção, migração entre threads e retomada após bloqueios.
- Comunicam-se por meio de **canais**, que fornecem troca segura de dados e atuam como mecanismo de sincronização.
  - O envio e o recebimento em canais são operações bloqueantes por padrão, promovendo sincronismo entre remetente e receptor.
  - Isso permite coordenar o avanço de goroutines sem o uso de exclusão mútua explícita.

Escalonador de Goroutines

- Goroutines são agendadas por um escalonador cooperativo mantido pelo runtime.
- Utiliza múltiplas filas e distribui goroutines sobre threads nativos.
- Mapeamento $M \times N$ promove boa utilização dos cores sem sobrecarga.
- O gerenciamento de threads sistema é responsabilidade do runtime; não é necessário – nem usual – que o programador os controle diretamente.

Criação de Goroutines: função e lambda

- Uma goroutine pode executar:
  - Uma **função nomeada**, como `f`.
  - Uma **função anônima** (lambda).
  - Um **método** de uma estrutura (não mostrado aqui).
- O prefixo go cria e agenda a goroutine para execução concorrente.
- O runtime gerencia o escalonamento sobre threads sistema.
- `sync.WaitGroup` garante que `main` aguarde a finalização.

```
package main
import (
  "fmt"
  "sync"
)
func f(nome string, wg *sync.WaitGroup) {
  defer wg.Done()
  fmt.Println("Executando:", nome)
}
func main() {
  var wg sync.WaitGroup
  wg.Add(2)
  go f("goroutine 1", &wg) // função nomeada
  go func() {              // função anônima
    defer wg.Done()
    fmt.Println("Executando: goroutine 2")
  }()
  wg.Wait()
  fmt.Println("Função main retornou.")
}
```

Canais: Comunicação entre Goroutines

- Canais permitem a comunicação de dados bidirecional (por padrão) entre goroutines forma segura e sincronizada.
- Criados com `make`, com o tipo aceito para a mensagem declarado.
- A operação de envio bloqueia até que haja recebedor (e vice-versa).
- O envio e recebimento são operações sincronizadas: um bloqueia até que o outro esteja pronto.

```
ch := make(chan int)

go func() {
  ch <- 313 // bloqueia até a recepção
}()

valor := <-ch
fmt.Println(valor)
```

Leitura com `for-range` em Canais

- Usado para ler todos os valores enviados por um canal.
- A leitura termina quando o canal é fechado com `close()`.
- Evita bloqueios e facilita leitura contínua de dados.

```
ch := make(chan int)

go func() {
  for i := 0; i < 3; i++ {
    ch <- i
  }
  close(ch)
}()

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

Canais Unidirecionais

- Uma função pode ser declarada com canal de leitura ou de escrita[a].
- Isso limita o uso do canal à operação esperada, reforçando segurança.
- Útil para modularizar e isolar responsabilidades.

[a] As setas para leitura (`<-chan`) e escrita (`chan<-`) fazem parte da definição do tipo. O hífen deve aparecer junto a `chan`, como em `chan<-` (e não separado, como `chan <-`).

```
func produtor(out chan<- int) {
  out <- 313
}

func consumidor(in <-chan int) {
  fmt.Println(<-in)
}

func main() {
  ch := make(chan int)
  go produtor(ch)
  consumidor(ch)
}
```

select: Comunicação Concorrente

- O comando `select` permite aguardar múltiplas operações de canal simultaneamente.
- Executa o primeiro caso disponível — a escolha é não determinística entre os canais prontos.
- Pode ser usado para implementar timeouts, cancelamento ou multiplexação.
- Caso nenhum canal esteja pronto, o `select` bloqueia até que algum esteja.

Exemplo: timeout após 2 segundos

```
select {
case msg1 := <-canal1:
  fmt.Println("Recebido de canal1:", msg1
case msg2 := <-canal2:
  fmt.Println("Recebido de canal2:", msg2
case <-time.After(2 * time.Second):
  fmt.Println("Timeout")
}
```

Canais com e sem Buffer

- Os exemplos anteriores usaram **canais sem buffer**, que exigem que remetente e receptor estejam prontos simultaneamente.
- Canais **com buffer** armazenam dados temporariamente até um limite definido.
- O envio em um canal com buffer **bloqueia apenas quando o buffer está cheio**.
- No exemplo ao lado, a terceira mensagem bloqueia o produtor até que o consumidor libere espaço no buffer.
- Esse mecanismo permite maior desacoplamento entre produtor e consumidor.

```
var ch = make(chan int, 2)
func produtor() {
    ch <- 167
    ch <- 176
    ch <- 617 // bloqueia se buffer cheio
}
func consumidor() {
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
func main() {
    go produtor()
    consumidor()
}
```

Barreiras e Seções Críticas

- Embora a comunicação por canais seja preferida em Go, há situações em que goroutines acessam dados compartilhados diretamente.
- Para garantir exclusão mútua nesse acesso, a linguagem fornece `sync.Mutex` e `sync.RWMutex`.
- O `RWMutex` permite múltiplas leituras concorrentes, mas bloqueia escritas até que todas as leituras terminem.
- Go não possui guardas de mutex como em C++; o padrão idiomático é usar `defer` logo após o `Lock()`, garantindo o desbloqueio seguro mesmo em caso de erro ou retorno antecipado.
- A estrutura `sync.WaitGroup` funciona como uma barreira: permite aguardar a finalização de múltiplas goroutines antes de continuar a execução.

Exclusão Mútua: Mutex e RWMutex

sync.Mutex

- Garante acesso exclusivo à seção crítica.
- Bloqueia todas as outras goroutines até que o mutex seja liberado.
- Usado para proteger variáveis compartilhadas em atualizações concorrentes.
- O uso de `defer` assegura o desbloqueio mesmo em caso de erro.

```
var m sync.Mutex
var contador int

func incrementar() {
  m.Lock()
  defer m.Unlock()
  contador++
}
```

sync.RWMutex

- Permite múltiplas leituras concorrentes, mas leituras exclusivas.
- Caso em que # de leituras >> # de escritas.
- Requer `RLock/RUnlock` para leitura e `Lock/Unlock` para escrita.

```
var rw sync.RWMutex
var dados = make(map[string]string)

func ler(chave string) string {
  rw.RLock()
  defer rw.RUnlock()
  return dados[chave]
}
func escrever(chave, valor string) {
  rw.Lock()
  defer rw.Unlock()
  dados[chave] = valor
}
```

Sincronização com sync.WaitGroup

- `sync.WaitGroup` é utilizado para aguardar a finalização de múltiplas **goroutines**.
- Atua como uma barreira: a goroutine que chama `Wait()` bloqueia até que todas as outras sinalizem término com `Done()`.
- O método `Add(n)` indica quantas goroutines serão aguardadas.
- Deve-se garantir que cada goroutine chame `Done()`, preferencialmente com `defer`.
- Muito útil para manter a função `main` ativa até a conclusão das **goroutines**.

```go
var wg sync.WaitGroup

func tarefa(id int) {
  defer wg.Done()
  fmt.Println("Executando:", id)
}

func main() {
  wg.Add(2) // aguardará 2 goroutines

  go tarefa(1)
  go tarefa(2)

  wg.Wait() // bloqueia até ambas terminarem
  fmt.Println("Todas as goroutines terminaram.")
}
```

Recursos Avançados do Runtime

- Além de goroutines e canais, o runtime de Go fornece mecanismos complementares para controle fino da concorrência estruturada por mensagens.
  - **context** permite coordenar o ciclo de vida de execuções concorrentes, promovendo cancelamento cooperativo.
  - **sync.Pool** oferece reutilização eficiente de objetos temporários para reduzir o custo de alocação.

Controle de Ciclo de Vida: context

- Cancelamento cooperativo com `context.WithCancel`.
- Timeout automático com `context.WithTimeout`.
- O contexto é propagado entre goroutines.
- Evita vazamentos de execução e facilita coordenação.

```
ctx, cancel := context.WithCancel(context
go func() {
  select {
  case <-ctx.Done():
    fmt.Println("Cancelado")
  }
}()
time.Sleep(time.Second)
cancel() // sinaliza cancelamento
```

Otimização com sync.Pool

- Reutiliza objetos temporários para evitar alocações frequentes.
- Reduz pressão sobre o coletor de lixo.
- Ideal para cargas intensas (ex.: buffers, estruturas auxiliares).
- Objetos podem ser descartados a qualquer momento.

```
var pool = sync.Pool{
  New: func() any {
    return make([]byte, 1024)
  },
}

buf := pool.Get().([]byte)
// uso temporário de buf
pool.Put(buf) // retorna ao pool
```

Elixir

Elixir

- Criado por José Valim, lançado em 2012.
- Projetado para sistemas concorrentes, distribuídos e tolerantes a falhas.
- Executado sobre a máquina virtual Erlang (BEAM).
- Adota o modelo de atores: processos leves, isolamento completo, comunicação assíncrona.
- Filosofia "let it crash":
  - Parte do princípio de que falhas são inevitáveis em sistemas concorrentes.
  - Em vez de capturar e tratar todos os erros localmente, processos devem falhar rápido.
  - A recuperação é delegada a supervisores dedicados.
  - Permite sistemas mais simples, modulares e resilientes.
- Favorável à escalabilidade e robustez em aplicações como servidores web e pipelines de dados.

Filosofia "Let it Crash"

- Elixir (e Erlang) adotam um princípio fundamental:
  - **Não** tente capturar e tratar todos os erros localmente.
  - Permita que processos falhem imediatamente ao detectar erro.
  - Deixe que um supervisor reinicie o processo em estado limpo.
- Vantagens:
  - Código mais simples, sem lógica defensiva excessiva.
  - Evita propagação de estado corrompido.
  - Torna o sistema mais robusto e previsível sob carga.
- O sistema *assume falhas como normais*, e se organiza para sobreviver a elas.

O que é um Processo em Elixir

- Um *processo* em Elixir não deve ser confundido com:
  - Um processo do sistema operacional.
  - Um *thread* tradicional de linguagens como C++ ou Java.
- Cada processo Elixir:
  - Possui seu próprio heap, pilha e mailbox.
  - É completamente isolado dos demais – não há compartilhamento de memória.
  - Tem um descritor leve gerenciado pela máquina virtual BEAM.
- A execução dos processos é feita por **threads nativas** do SO, chamadas **schedulers**.
  - Os processos são multiplexados nesses schedulers por agendamento preemptivo cooperativo.
- Essa estrutura permite a criação de **centenas de milhares de processos concorrentes**, com alta eficiência e robustez.

Tarefas Assíncronas (Task)

- Executam computações concorrentes de curta duração.
- `Task.async` cria um processo leve que executa uma função.
- Retorna uma estrutura que identifica a tarefa em andamento.
- `Task.await` bloqueia até o resultado da tarefa estar disponível.
- É a forma mais simples de utilizar multiprogramação leve em Elixir.

```elixir
# Cria uma tarefa assíncrona
tarefa = Task.async(fn -> 2 * 21 end)

# Aguarda o término e obtém o resultado
resultado = Task.await(tarefa)

IO.puts("Resultado: #{resultado}")
```

Modelo Básico de Concorrência

- Concorrência baseada em processos Erlang leves, isolados e sem memória compartilhada.
- Criados com `spawn`, cada um com seu próprio `pid`.
- Cada processo executa uma função de forma independente.
- Milhares de processos podem ser criados com baixo custo.
- O identificador do processo (`pid`) permite comunicação posterior.

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

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

Envio e Recepção de Mensagens

- Processos em Elixir trocam mensagens assíncronas por meio de suas **mailboxes**.
- Envio com o operador `pid <- mensagem`, equivalente a `send(pid, mensagem)`.
- Recepção com `receive`, que examina a mailbox e aplica **pattern matching**.
- A execução bloqueia até que uma mensagem compatível seja encontrada.
- Cada processo trata suas próprias mensagens — não há interferência externa.

```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()})
```

Timeout em `receive` e Boas Práticas

- Um processo pode aguardar mensagens com `receive`.
- Se nenhuma mensagem correspondente chegar em tempo hábil, executa-se a cláusula `after`.
- Neste exemplo, outro processo envia a mensagem `:ping`.
- O tempo limite evita bloqueio indefinido.

```
spawn(fn ->
  receive do
    :ping -> IO.puts("pong")
  after
    1000 -> IO.puts("Timeout")
  end
end)

# Processo remetente
:timer.sleep(500)
send(self(), :ping)
```

Monitoramento e Tolerância a Falhas

Em Elixir, a possibilidade de criar milhares de processos leves com segurança está diretamente ligada à forma como a linguagem lida com falhas: de maneira isolada, previsível e supervisionada.

- Processos podem ser monitorados com `Process.monitor(pid)` (notificação unidirecional).
- Também podem ser vinculados com `link(pid)` (vínculo bidirecional).
- `Process.flag(:trap_exit, true)` permite tratar falhas como mensagens.
- Supervisores automatizam a reinicialização de processos:
  - Estratégias: `:one_for_one`, `:one_for_all`, `:rest_for_one`.
- Base do modelo de tolerância a falhas do Elixir (*let it crash*).

Agentes (Agent)

Definição técnica e conceitual:

Agent é um processo leve que encapsula um estado interno e o expõe por meio de uma interface funcional (funções que recebem o estado atual e retornam o novo).

- Permite armazenar e atualizar valores de forma segura entre processos.
- Criado com `Agent.start_link/1` ou `/2`.
- Estado acessado com `Agent.get/2`.
- Estado modificado com `Agent.update/2`.
- As funções são puras e operam sobre o estado encapsulado.

Paralelo com objeto:

De forma livre, é possível associar `Agent` ao conceito de **objeto** (da POO) que possui estado interno. Nesse caso, o estado pertence a um processo isolado (o agente), o que permite acesso seguro e concorrente sem bloqueios. Diferencia-se por não possuir métodos próprios, mas sim por receber funções que descrevem as operações a serem aplicadas ao seu estado interno.

Agentes (Agent)

Definição técnica e conceitual:

Agent é um processo leve que encapsula um estado interno e o expõe por meio de uma interface funcional (funções que recebem o estado atual e retornam o novo).

- Permite armazenar e atualizar valores de forma segura entre processos.
- Criado com `Agent.start_link/1` ou `/2`.
- Estado acessado com `Agent.get/2`.
- Estado modificado com `Agent.update/2`.
- As funções são puras e operam sobre o estado encapsulado.

Paralelo com o modelo de ator:

`Agent` é uma forma especializada de **ator** (do modelo de atores): encapsula estado, executa de forma isolada e responde a requisições sequencialmente. A interação ocorre por envio implícito de mensagens, onde funções são recebidas para leitura ou modificação do estado. Assim, comporta-se como um ator que oculta a troca explícita de mensagens sob uma interface

Agent na prática

- O agente é iniciado com valor interno igual a zero.
- A função update recebe uma função que incrementa esse valor.
- A função get recebe uma função que retorna o valor atual.
- O nome da variável cont é livre e representa o valor interno.

```
{:ok, pid} = Agent.start_link(fn -> 0 end)

Agent.update(pid, fn cont ->
  cont + 1
end)

valor = Agent.get(pid, fn cont ->
  cont
end)

IO.puts("Valor atual: #{valor}")
```

Tarefas Assíncronas (Task)

- Executam computações concorrentes de curta duração.
- `Task.async` cria um processo leve que executa uma função.
- Retorna uma estrutura que identifica a tarefa em andamento.
- `Task.await` bloqueia até o resultado da tarefa estar disponível.

```elixir
# Cria uma tarefa assíncrona
tarefa = Task.async(fn -> 2 * 21 end)

# Aguarda o término e obtém o resultado
resultado = Task.await(tarefa)

IO.puts("Resultado: #{resultado}")
```

GenServer

- Abstração robusta para implementar processos com estado interno e comportamento controlado.
- Segue o modelo de ator completo: recebe mensagens e atualiza seu estado de forma isolada.
- Usa funções de callback:
  - `handle_call` para mensagens síncronas (espera resposta).
  - `handle_cast` para mensagens assíncronas (sem resposta).
  - `handle_info` para outras mensagens.
- Integrado ao sistema de supervisão do Elixir.
- Ideal para serviços concorrentes persistentes que mantêm estado.

GenServer na prática

- O módulo `Contador` implementa um processo com estado interno.
- A função `start_link` inicia o processo com valor zero.
- `incrementar` envia uma mensagem assíncrona (`cast`) para somar 1.
- `valor` envia uma mensagem síncrona (`call`) para obter o valor atual.
- O estado é mantido isolado e atualizado pelos callbacks internos.

```
defmodule Contador do
  use GenServer
  def start_link do
    GenServer.start_link(__MODULE__, 0,
                         name: __MODULE__)
  end
  def incrementar do
    GenServer.cast(__MODULE__, :inc)
  end
  def valor do
    GenServer.call(__MODULE__, :valor)
  end
  def init(cont), do: {:ok, cont}
  def handle_cast(:inc, cont),
    do: {:noreply, cont + 1}
  def handle_call(:valor, _de, cont),
    do: {:reply, cont, cont}
end
```

Fluxo Concorrente com `Flow`

- `Flow` permite construir pipelines de processamento de dados com execução concorrente.
- Cada etapa do pipeline (filtro, transformação etc.) pode ser executada em paralelo.
- Os dados fluem por processos leves organizados em estágios.
- O fluxo é criado a partir de uma coleção com `Flow.from_enumerable/1`.
- A paralelização é configurada com `Flow.partition(stages: n)`.
- A execução é **preguiçosa**: o processamento só ocorre quando uma função do módulo `Enum` (como `sum`, `to_list`, etc.) é chamada no final.
- Útil para aplicações que processam grandes volumes de dados de forma eficiente.

Flow na prática

- Um fluxo é iniciado a partir de uma faixa de números.
- O filtro seleciona apenas os pares.
- Cada número par é multiplicado por 2.
- A execução do pipeline só começa com `Enum.sum/1`.
- A paralelização ocorre nos estágios de filtro e mapeamento.

```
alias Experimental.Flow

1..100_000
|> Flow.from_enumerable()
|> Flow.filter(&rem(&1, 2) == 0)
|> Flow.map(&(&1 * 2))
|> Enum.sum()
```

Estudos de Caso

Estudos de Caso

- Dois estudos de caso:
  - Produtor/Consumidor
  - Cálculo da n-ésima posição da série de Fibonacci
- Implementações em C++, Rust, Go e Elixir
- Ênfase didática, não em desempenho

Reprodutibilidade

- Códigos disponíveis em: `https://github.com/GersonCavalheiro/JAI2025`
- Outras implementações e casos adicionais
- Dockerfile disponível para facilitar execução

Produtor/Consumidor (Visão Geral)

- Área de dados compartilhada: fila de itens produzidos
- Parâmetros: número de produtores, consumidores e itens por produtor
- Itens produzidos: números primos
- Valor sentinela (-1 ou `u32::MAX`) indica fim da produção

C++ (Prod/Cons): Variável de Condição

```
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) break;
    // processa item
  }
}
```

C++ (Prod/Cons): Promessas e Futuros

```
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();
}

void consumidor(int id) {
  while (true) {
    int val = -2;
    {
      std::lock_guard<std::mutex> lock(mtx);
      if (!buffer.empty())
        val = buffer.front(); buffer.pop();
```

C++ (Prod/Cons): Cancelamento Cooperativo

```
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++;
  }
}

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(); });
```

Rust (Prod/Cons): Variável de Condição

```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();
  });
```

Rust (Prod/Cons): Variável de Condição (continuação)

```
  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();
}
```

Rust (Prod/Cons): Canais

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
  let (tx, rx) = mpsc::channel();
  let produtores: Vec<_> = (0..2).map(|id| {
    let tx = tx.clone();
    thread::spawn(move || {
      let mut num = 2;
      let mut enviados = 0;
      while enviados < 5 {
        if eh_primo(num) {
          tx.send(num).unwrap();
          enviados += 1;
        }
        num += 1;
      }
    })
  }).collect();
```

Rust (Prod/Cons): Canais (cont.)

```
  drop(tx);
  for val in rx {
    println!("Consumidor recebeu {}", val);
  }

  for p in produtores {
    p.join().unwrap();
  }
}

fn eh_primo(n: u32) -> bool {
  if n < 2 { return false; }
  for i in 2..=((n as f64).sqrt() as u32) {
    if n % i == 0 { return false; }
  }
  true
}
```

Go (Prod/Cons): Goroutines e canais

```go
func produtor(id int, total int, ch chan int) {
  count, num := 0, 2
  for count < total {
    if ehPrimo(num) {
      ch <- num
      count++
    }
    num++
  }
}

func consumidor(ch chan int, done chan bool) {
  for val := range ch {
    fmt.Println("Consumidor recebeu", val)
  }
  done <- true
}
```

Go (Prod/Cons): Goroutines e canais (cont.)

```
func main() {
  ch := make(chan int)
  done := make(chan bool)
  go produtor(0, 5, ch)
  go func() {
    for i := 0; i < 5; i++ {
      fmt.Println("Consumidor recebeu", <-ch)
    }
    close(ch)
    done <- true
  }()
  <-done
}
```

Elixir (Prod/Cons): spawn e send

```elixir
defmodule Produtor do
  def iniciar(consumer_pid, total) do
    spawn(fn -> gerar(2, 0, total, consumer_pid) end)
  end

  defp gerar(_, count, total, _) when count >= total, do: :ok
  defp gerar(n, count, total, pid) do
    if primo?(n) do
      send(pid, {:item, n})
      gerar(n + 1, count + 1, total, pid)
    else
      gerar(n + 1, count, total, pid)
    end
  end
```

Elixir (Prod/Cons): spawn e send (cont.)

```
  defp primo?(n), do:
    n > 1 and Enum.all?(2..:math.sqrt(n) |> trunc, &(rem(n, &1) != 0))
end

defmodule Consumidor do
  def iniciar() do
    spawn(fn -> loop() end)
  end

  defp loop() do
    receive do
      {:item, val} ->
        IO.puts("Consumidor recebeu #{val}")
        loop()
    end
  end
end

pid = Consumidor.iniciar()
Produtor.iniciar(pid, 5)
```

Fibonacci: Descrição Geral

- Objetivo: calcular o valor da posição n da série de Fibonacci.
- Requisitos:
  - Entrada: valor de n
  - Saída: $F(n)$
- Implementações exploram recursos característicos de cada linguagem:
  - C++20: `std::async` e `std::future`
  - Rust: `JoinHandle` e recursão paralela
  - Go: goroutines e canais
  - Elixir: processos leves e `receive`

C++20 – std::async

```
int 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();
}
```

Rust – rayon

```
fn fib(n: u32) -> u32 {
    if n <= 1 {
        n
    } else {
        let (a, b) = rayon::join(|| fib(n - 1), || fib(n - 2));
        a + b
    }
}
```

Go – Goroutines e Canais

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

Elixir – Processos e Mensagens

```
defmodule Fib do
  def start(n, caller) do
    send caller, {:result, fib(n)}
  end

  def fib(0), do: 0
  def fib(1), do: 1
  def fib(n) do
    caller = self()
    spawn(Fib, :start, [n-1, caller])
    spawn(Fib, :start, [n-2, caller])
    receive do {:result, a} ->
      receive do {:result, b} -> a + b end
    end
  end
end
```

Considerações Finais

- Todas as abordagens são didáticas e evidenciam o modelo concorrente de cada linguagem.
- Nem todas são eficientes para grandes valores de entrada.
- Elixir e Go favorecem modelos com passagem de mensagens.
- C++ e Rust oferecem mais controle de baixo nível.

Considerações Finais

Encerramento

- Visão comparativa de C++, Rust, Go e Elixir.
- Cada linguagem adota um modelo distinto de concorrência.
- Objetivo: oferecer subsídios, não eleger vencedores.

Modelos de Concorrência: Contrastes Técnicos

- **C++**: controle explícito com `thread`, `mutex`, `atomic`, ttfuture.
- **Rust**: segurança por construção, verificação de posse em tempo de compilação.
- **Go**: concorrência leve com `goroutines` e canais (*CSP*).
- **Elixir**: modelo de atores com troca assíncrona de mensagens e supervisão.

Modelos de Concorrência: Abstrações e Paradigmas

- Diferenças vão além da sintaxe ou das APIs.
- Moldam estrutura do código, sincronização e estratégia de execução.
- Comparação evidencia decisões de projeto e suas implicações práticas.

Limites e Horizontes

- Conteúdo como ponto de partida, não de exaustão.
- Incentivo à leitura das referências e da documentação oficial.
- A prática é indispensável à construção de competência em programação multithread.

Reflexão Final

- Concorrência é um dos campos mais desafiadores da Computação.
- Adoção crescente de arquiteturas paralelas amplia a relevância do tema.
- Recomenda-se especialização em uma ou poucas linguagens, sem ignorar a diversidade.

Comparando Modelos de Concorrência

- Todas as linguagens analisadas oferecem suporte à concorrência.
- C++20 fornece construções de baixo nível: `thread`, `mutex`, `atomic`, etc.
- Rust, Go e Elixir incorporam modelos mais restritivos ou estruturados.
- A flexibilidade de C++ é ampla, mas exige maior responsabilidade do programador.

O que C++20 oferece (não exaustivo...)

- **Criação e controle**: threads nativos com `std::thread` e cancelamento com `std::jthread`.
- **Sincronização**: `mutex`, `shared_mutex`, variáveis de condição.
- **Assíncrono**: tarefas com `std::async` e `future`.
- **Paralelismo de dados**: algoritmos paralelos da STL.
- Máxima flexibilidade com suporte direto da linguagem e biblioteca padrão.

O que Rust, Go e Elixir trazem que C++20 não oferece, pelo menos não na mesma forma (não exaustivo...)

- **Rust:** garantias de segurança em tempo de compilação (ownership, borrowing).
- **Go:** goroutines leves com agendamento cooperativo no runtime.
- **Elixir:** modelo de atores supervisionado, com isolamento completo.
- **Rust/Elixir:** estilo funcional com imutabilidade.
- **Elixir:** tolerância à falha como parte da arquitetura.

Alguns dos possíveis comentários que podem ser feitos para fechamento, passíveis de discussão e de espaço ao contraditório

- **C++20:** poder e flexibilidade, mas requer grande cuidado.
- **Rust:** segurança forte, excelente para sistemas confiáveis e paralelismo seguro.
- **Go:** leveza, escalabilidade e simplicidade na concorrência.
- **Elixir:** concorrência robusta com foco em tolerância à falha.
- A construção do conhecimento em multithread é contínua e cumulativa.


Obrigado!
https://github.com/GersonCavalheiro/JAI2025