Revisitando Clássicos da Concorrência: Implementação e Avaliação em OpenMP, Rust e Go

Lucas Braatz Araújo[1], **Daniel Di Domenico**[2], **André Rauber Du Bois**[1], **Gerson Geraldo H. Cavalheiro**[1]

[1]Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas
R. Gomes Carneiro, 01 – Pelotas – RS/Brasil – 96010-610

[2]Instituto Federal do Paraná
Foz do Iguaçu, PR/Brasil

{lbaraujo,dubois,gersonc}@inf.ufpel.edu.br, daniel.domenico@ifpr.edu.br

Abstract. *This paper presents a study comparing the expressiveness of OpenMP, Rust, and Go in the implementation of five classical concurrent programming problems. The solutions, based on idiomatic approaches, were evaluated in terms of implementation effort, safety, and performance, using the Goal Question Metric (GQM) method. The results highlight the strengths and limitations of each language in different synchronization and resource-sharing contexts, providing insights for selecting appropriate tools in the development of multithreaded systems.*

Resumo. *Este artigo apresenta um estudo em que foram comparadas a expressividade de OpenMP, Rust e Go na implementação de cinco problemas clássicos de programação concorrente. As soluções, baseadas em abordagens idiomáticas, foram avaliadas quanto ao esforço de implementação, segurança e desempenho, utilizando o método Goal Question Metric (GQM). Os resultados revelam os pontos fortes e limitações de cada linguagem em diferentes contextos de sincronização e compartilhamento de recursos, oferecendo subsídios para a escolha de ferramentas no desenvolvimento de sistemas multithread.*

1. Introdução

A evolução tecnológica tem ampliado a demanda por aplicações com alta eficiência e desempenho. A crescente complexidade dos sistemas computacionais exige novas abordagens e linguagens de programação que ofereçam soluções robustas para o desenvolvimento de software. Nesse contexto, a popularização de arquiteturas multicore impulsionou o surgimento de linguagens com abstrações voltadas à exploração do paralelismo.

A programação concorrente e paralela não é apenas útil para otimizar o uso do hardware disponível, mas também essencial em aplicações cuja natureza exige múltiplas tarefas simultâneas. Técnicas como multiprocessamento e paralelismo de tarefas viabilizaram melhorias significativas no desempenho de programas, embora introduzam desafios, como a sincronização e o compartilhamento seguro de recursos entre múltiplos fluxos de execução. Problemas como condições de corrida e *deadlocks* são comuns e demandam mecanismos que garantam a consistência dos dados e a execução correta dos programas [Garzaran 2009].

Algoritmos clássicos são amplamente utilizados para ilustrar os desafios da programação concorrente [Hazlett 2019, Foley 2021, Salguero 2018], além de serem ferramentas pedagógicas fundamentais. Dentre eles, citam-se Produtor-Consumidor, Leitores-Escritores, Jantar dos Filósofos, Barbeiro Dorminhoco e Problema dos Fumantes. Se distanciando das abordagens em que tais problemas são solucionados com o uso de ferramentas consolidadas, como Pthread e Java, este trabalho foca na comparação das implementações obtidas com OpenMP [van der Pas 2007], Rust [Klock 2014] e Go [Tu et al. 2019], três interfaces/linguagens contemporâneas para programação concorrente e paralela, quanto à sua capacidade de expressar esses problemas.

O objetivo deste trabalho é realizar uma análise comparativa da expressividade da interface OpenMP e das linguagens Rust e Go na implementação de problemas clássicos de programação concorrente. A avaliação será conduzida com base no método Goal Question Metric (GQM) [Rombach 1994], permitindo uma investigação sobre aspectos como esforço de implementação, segurança e desempenho das soluções adotadas.

A principal contribuição deste estudo consiste na análise empírica dessas interfaces/linguagens no contexto da programação concorrente, fornecendo evidências sobre suas vantagens e limitações. Para isso, são implementadas versões dos problemas clássicos Produtor-Consumidor, Leitores-Escritores, Jantar dos Filósofos, Barbeiro Dorminhoco e Problema dos Fumantes, permitindo uma comparação direta entre as abordagens utilizadas em OpenMP, Rust e Go. Além disso, os resultados obtidos contribuem para a literatura existente ao oferecer uma visão atualizada sobre o uso dessas tecnologias na resolução de problemas de concorrência. Essa investigação pode servir de base para futuras pesquisas, auxiliando desenvolvedores e acadêmicos na escolha de ferramentas adequadas para programação paralela e concorrente.

Reprodutibilidade e informações adicionais: os códigos-fonte desenvolvidos para este estudo, assim como os dados completos das aferições realizadas, encontram-se disponíveis em repositório público[1] para fins de auditoria e reprodutibilidade.

O restante deste texto está organizado como segue. A Seção 2 detalha a API OpenMP e as linguagens Rust e Go. Em seguida, a Seção 3 descreve os problemas clássicos de concorrência utilizados neste trabalho. A análise comparativa da expressividade entre as linguagens baseadas nos estudos de caso é o foco da Seção 4. A Seção 5 aborda as limitações a serem consideradas para este estudo. Os trabalhos relacionados são listados na Seção 6. Por fim, a Seção 7 apresenta a conclusão e os trabalhos futuros.

2. Ferramentas de Programação

Esta seção apresenta os conceitos fundamentais, em particular seus mecanismos de concorrência e gerenciamento de memória, das linguagens OpenMP, Rust e Go.

2.1. OpenMP

OpenMP[2] é uma API amplamente adotada para programação paralela em memória compartilhada, especialmente em C, C++ e Fortran. Seu modelo *fork-join* permite paralelismo

[1]Repositório Git para reprodutibilidade e informações adicionais: `https://github.com/GersonCavalheiro/yvyrupa_code`

[2]OpenMP: `https://www.openmp.org`

incremental, no qual diretivas como `#pragma omp parallel`, `for` e `sections` possibilitam a criação e sincronização de threads. Além disso, OpenMP oferece suporte a atributos de memória como `shared` e `private`, com modificadores `firstprivate` e `lastprivate` para inicialização personalizada. A consistência de memória é gerenciada com memória local a cada thread, sendo sincronizada com a diretiva `flush`.

Para controle de concorrência, OpenMP dispõe de regiões `critical`, diretivas `atomic` e barreiras explícitas. A ausência de suporte a exceções paralelas é compensada por mecanismos que asseguram exclusão mútua e consistência. A API também permite otimizações envolvendo controle de afinidade de threads, balanceamento de carga com `schedule` e redução do overhead de sincronização. O suporte da API é amplo, com compatibilidade com compiladores (GCC, Clang, ICC) e ferramentas como Intel VTune e ARM MAP, sendo aplicável também em arquiteturas emergentes.

2.2. Rust

Rust[3] é uma linguagem de sistemas desenvolvida pela Mozilla Research, cujo design prioriza segurança, desempenho e concorrência. Seu sistema de tipos é estático com inferência. Já o gerenciamento de memória é feito sem *garbage collector* por meio de um modelo de propriedade verificada em tempo de compilação [II 2014]. Com isso, valores têm um único dono e são desalocados automaticamente ao fim do escopo. O modelo da Rust distingue claramente entre alocação em pilha e *heap*, prevenindo vazamentos e acessos inválidos. Por fim, seu compilador aplica otimizações agressivas e suporta abstrações de custo zero [Zhang et al. 2023].

Para concorrência, Rust oferece `Arc` e `Mutex` para compartilhamento seguro e sincronização entre threads, com garantias em tempo de compilação contra *data races* [Qin et al. 2020]. Abstrações como `Condvar` permitem coordenação eficiente entre threads. Já para exceções, Rust possui um tratamento de erros que distingue casos recuperáveis (`Result<T, E>`) e irrecuperáveis (`panic!`), com *pattern matching* para manuseio claro de falhas.

Outra característica da linguagem Rust é que seu ecossistema é gerenciado pelo Cargo, com suporte a testes, benchmarks e documentação integrados. O uso crescente da linguagem em sistemas operacionais, redes e aplicações embarcadas destaca sua maturidade, com integração a outras linguagens via FFI (*Foreign Function Interface*).

2.3. Go

Go[4], criada pelo Google, combina simplicidade sintática com recursos eficientes de concorrência, sendo amplamente empregada em redes, serviços em nuvem e sistemas distribuídos. O modelo de execução é baseado em *goroutines*, que são funções concorrentes gerenciadas pelo *runtime* da linguagem. Canais (*channels*) são utilizados como primitivas para comunicação e sincronização entre *goroutines* [Liu et al. 2021]. A linguagem oferece recursos como retorno múltiplo de valores, `defer` para liberação de recursos e interfaces estruturais, além de primitivas adicionais de sincronização do pacote `sync`, incluindo `Mutex`, `WaitGroup` e `Cond`.

[3]The Rust Programming Language, `https://doc.rust-lang.org/book/`.

[4]Go: `https://go.dev`

O sistema de tipos de Go é estático com inferência. Arrays são fixos e passados por valor, enquanto `slices`, `maps` e canais são alocados com `make`. O tratamento de erros é feito com o tipo `error`, sendo `panic` e `recover` reservados a exceções críticas. O *runtime* de Go inclui coletor de lixo moderno e escalonador leve de goroutines. O compilador gera código nativo e permite análises com ferramentas como `pprof`. O comando `go` centraliza formatação, testes e dependências, e o repositório `pkg.go.dev` oferece vasto acervo de bibliotecas, com suporte a depuração e construção de sistemas distribuídos com gRPC.

3. Problemas Clássicos de Concorrência e Suas Implementações

Os algoritmos clássicos de sincronização são amplamente utilizados para ilustrar os desafios do compartilhamento de recursos entre múltiplas threads ou processos. Neste trabalho, são considerados cinco problemas tradicionais: Jantar dos Filósofos, Produtor-Consumidor, Leitores-Escritores, Barbeiro Dorminhoco e Fumantes de Cigarro. Suas regras e restrições são descritas a seguir, juntamente com as abordagens de implementação em OpenMP, Rust e Go.

3.1. Jantar dos Filósofos

Neste problema, cada filósofo alterna entre pensar e comer, precisando de dois garfos adjacentes para comer. O objetivo é evitar deadlocks e starvation, garantindo justiça na alocação de recursos [Hoare 1981].

OpenMP representa cada filósofo com uma thread e usa operações atômicas para garantir a aquisição segura dos garfos, evitando o uso de mutexes explícitos. A criação e sincronização das threads ocorrem com diretivas `parallel` e barreiras implícitas. Já *Rust* utiliza `Mutex` e `Condvar` para controlar o acesso aos garfos compartilhados. Os garfos são protegidos por `Arc<Mutex<bool>>`, e os filósofos aguardam de forma eficiente até que ambos estejam disponíveis. *Go*, por sua vez, modela cada filósofo como uma *goroutine*, utilizando `sync.Mutex` para os garfos. A tentativa de aquisição usa `TryLock()`, e em caso de falha, o primeiro garfo é liberado e a tentativa é repetida após um atraso aleatório.

3.2. Produtor-Consumidor

O programa modela a comunicação entre processos que compartilham um buffer limitado. O produtor insere itens e o consumidor os retira, exigindo sincronização para evitar inconsistências [Bos 2014].

A implementação *OpenMP* usa uma thread produtora e várias consumidoras com exclusão mútua via `critical`, e visibilidade garantida com `flush`. O acesso ao buffer é direto, típico do modelo de memória compartilhada. A solução com *Rust* emprega `Arc<Mutex<T>>` para o buffer, com uso de `Condvar` para coordenação eficiente entre produtores e consumidores. A segurança contra *data races* é garantida em tempo de compilação. O código em *Go* utiliza goroutines com `sync.Mutex` e `sync.Cond`. Os produtores e consumidores aguardam condições favoráveis com `Wait()`, sendo notificados por `Signal()` ou `Broadcast()`.

3.3. Leitores-Escritores

Este algoritmo permite múltiplos leitores simultâneos, mas exige exclusividade para escritores. Esse padrão é comum em bancos de dados e sistemas de arquivos [Galvin 2018].

A versão com *OpenMP* sincroniza o acesso com diretivas `atomic` e `critical`, usando *busy-waiting* para monitorar o estado do recurso. A desenvolvida com *Rust* adota `Arc<Mutex<T>>` e `Condvar`, priorizando escritores para evitar starvation. O controle é seguro e eficiente, mas mais verboso. Já a versão em *Go* emprega `sync.Mutex` e `sync.Cond` com *goroutines*. A implementação também prioriza escritores, com uso de `WaitGroup` para sincronizar o término das *goroutines*.

3.4. Barbeiro Dorminhoco

Este problema clássico modela um barbeiro que atende clientes em uma barbearia com número limitado de cadeiras. O desafio é sincronizar adequadamente o fluxo de clientes e evitar ociosidade [Galvin 2018].

Com *OpenMP*, divide-se a execução entre barbeiro e clientes com `parallel sections` e `critical`, utilizando *busy-waiting* para verificar disponibilidade. O programa *Rust* sincroniza o estado da barbearia com `Arc<Mutex<T>>` e `Condvar`, permitindo que o barbeiro durma e seja acordado por clientes com notificações eficientes. Por fim, a aplicação em *Go* utiliza `sync.Mutex` e `sync.Cond` para que o barbeiro aguarde com `Wait()` e seja acordado com `Signal()`, onde canais e contadores controlam o fluxo e a ocupação das cadeiras.

3.5. Fumantes de Cigarro

A implementação apresenta um cenário onde três fumantes aguardam dois ingredientes fornecidos por um agente para produzir um cigarro. A sincronização deve evitar acesso simultâneo à mesa e garantir justiça no acesso aos ingredientes [Downey 2008].

OpenMP usa `parallel sections` e `critical` para dividir a execução entre agente e fumantes. Os fumantes verificam periodicamente os ingredientes, o que pode gerar espera ativa. O programa *Rust* usa canais (via `crossbeam`) para comunicação entre threads, com segurança garantida em tempo de compilação. A gestão dos canais e threads é mais explícita e requer tratamento cuidadoso. Já *Go* utiliza canais dedicados para cada fumante, permitindo comunicação eficiente e sem bloqueios explícitos. Assim, o agente envia pares de ingredientes, e os fumantes reagem ao canal correspondente.

3.6. Desafios da Concorrência

Cada algoritmo lida com os desafios de **deadlock** (processos entram em espera mútua indefinida por recursos), **starvation** (um processo nunca obtém os recursos necessários) e **livelock** (processos continuam executando sem progresso efetivo), conforme mostrado na Tabela 1. Esses problemas comprometem o progresso das threads e o acesso justo a recursos compartilhados.

As estratégias adotadas por cada linguagem para resolver esses problemas refletem seus paradigmas: OpenMP privilegia desempenho e simplicidade com memória compartilhada; Rust enfatiza segurança com verificação em tempo de compilação; Go valoriza leveza e produtividade com goroutines e canais. A questão que se impõe é se essas linguagens modernas realmente oferecem vantagens concretas na resolução desses problemas clássicos. O restante do texto busca contribuir com essa análise.

Tabela 1. Principais desafios enfrentados nos algoritmos clássicos

Algoritmo	Deadlock	Starvation	Livelock
Jantar dos Filósofos	Sim	Sim	Sim
Produtor-Consumidor	Não	Sim	Não
Leitores-Escritores	Sim	Sim	Sim
Barbeiro Dorminhoco	Não	Sim	Não
Fumantes de Cigarros	Sim	Sim	Sim

4. Avaliação dos Estudos de Caso

A avaliação da expressividade de OpenMP, Rust e Go para implementação dos problemas clássicos de concorrência foi realizada utilizando o método GQM (Goal, Question, Metric) [Rombach 1994]. Este método permite definir metas a serem analisadas, questões para avaliar o atingimento de metas e métricas objetivas para mensuração.

4.1. Modelo GQM Empregado

O modelo GQM proposto possui dois objetivos, a partir dos quais foram concebidas as questões para as quais métricas foram empregadas a fim de respondê-las. O modelo proposto encontra-se na Figura 1.

GO1: Avaliar a expressividade da linguagem em fornecer abstrações úteis para implementar paralelismo e concorrência.

Q1: A linguagem oferece abstrações úteis para concorrência e paralelismo?
- M-1.1.1 NLA: Quantidade de bibliotecas ou funções nativas usadas para paralelismo ou concorrência.
- M-1.1.2 NCA: Presença de abstrações nativas (e.g., async, fork, join).
- M-1.1.3 TCL: Ferramentas específicas fornecidas pela linguagem.

Q2: Quais mecanismos de controle e sincronização são suportados pela linguagem?
- M-1.2.1 SUA: Uso de mutexes, canais, ou outras abstrações de sincronização.
- M-1.2.2 SSE: Simplicidade no uso dessas abstrações.

Q3: A linguagem impõe um modelo específico de concorrência?
- M-1.3.1 CMI: Identificação do modelo (e.g., threads, processos, eventos).
- M-1.3.2 CMF: Flexibilidade na escolha de diferentes paradigmas de concorrência.

GO2: Quantificar o Esforço necessário para desenvolver os algoritmos clássicos.

Q1: Qual é a quantidade de linhas de código necessária para implementar a funcionalidade desejada utilizando concorrência e paralelismo?
- M-2.1.1 LOC: Número total de linhas de código.
- M-2.1.2 LOC-P: Número total de linhas de código com instruções de paralelismo e concorrência.

Q2: O código resultante é legível e fácil de entender?
- M-2.2.1 CC: Complexidade ciclomática.
- M-2.2.2 ACU: Uso de abstrações claras.

Q3: Quanto do esforço é direcionado ao paralelismo e concorrência?
- M-2.3.1 NOPC: Número de operações relacionadas a paralelismo e concorrência.
- M-2.3.2 PELU: Número de bibliotecas ou funções externas específicas usadas.

Figura 1. Modelo GQM concebido para os estudos de caso.

As métricas qualitativas M-1.3.2 (CMF) e M-2.2.3 (ACU) foram classificadas de acordo com níveis interpretativos. Para CMF, utilizamos os valores Restrito, Média e Flexível, representando respectivamente a rigidez ou liberdade que a linguagem ou ferramenta oferece na escolha do modelo de concorrência. Já para ACU, os valores Baixo, Médio e Alto refletem o grau de clareza e transparência das abstrações utilizadas no código, sendo Alto o nível mais claro e fácil de compreender, e Baixo o menos claro.

A métrica M-2.2.1, complexidade ciclomática (CC), foi estimada de forma simples somando-se +1 para cada estrutura de controle condicional ou de repetição presente

no código. Entre essas estruturas estão: *if*, *else if*, *for*, *while*, *do while*, *case* (em *switch*, cada *case* conta como um caminho independente), e *catch*. Operadores lógicos compostos como *&&* e || também podem ser considerados, pois introduzem múltiplos ramos de decisão, dependendo do nível de granularidade desejado na análise. Em linguagens como Rust e Go, estruturas como *match* e *select* também devem ser consideradas, pois cada ramo representa um caminho lógico distinto.

Nas subseções a seguir, as métricas adotadas foram agrupadas para facilitar a análise comparativa, reduzindo redundâncias e destacando padrões entre as abordagens. Os dados consolidados encontram-se na Tabela 2. As informações completas sobre os dados coletados encontram-se no repositório referenciado na Introdução deste texto.

Tabela 2. Métricas GQM Consolidadas (Expressividade e Esforço).

Expressividade				**Esforço**			
Métrica	**OpenMP**	**Rust**	**Go**	**Métrica**	**OpenMP**	**Rust**	**Go**
NLA (abstração)	Baixo	Alto	Médio	LOC (linhas de código)	Baixa	Alta	Média
NCA (complexidade)	Baixa	Alta	Média	LOC-P (código concorrente)	Baixa	Alta	Média
SSE (segurança)	Alta	Alta	Média-Alta	CC (complexidade)	Baixa	Alta	Média
CMF (flexibilidade)	Restrita	Alta	Alta	ACU (acessibilidade)	Alta	Média	Média

4.2. Avaliação da Expressividade

Com base nos dados apresentados na Tabela 2, observa-se uma variação significativa na expressividade entre as linguagens avaliadas.

No problema dos Filósofos Jantando, a implementação em Rust apresenta alta complexidade e exige um controle detalhado de recursos compartilhados para evitar interbloqueios. Em contraste, Go oferece uma solução mais intuitiva e com complexidade média, graças ao uso de *goroutines* e `sync.Mutex`. Já OpenMP adota a abordagem mais simples, com baixa complexidade, mas à custa de baixo nível de abstração e flexibilidade restrita, o que limita o controle sobre o estado global.

No problema do Produtor-Consumidor, Rust utiliza *Mutex* e *Condvar*, mantendo alta segurança mesmo com uma complexidade elevada. Go, por sua vez, emprega *channels*, o que resulta em um nível de abstração médio e boa flexibilidade (CMF: Alta), favorecendo a clareza do código. OpenMP, ainda que com baixa complexidade, revela-se limitado em termos de flexibilidade devido ao modelo baseado em diretivas.

Para o problema de Leitores e Escritores, Rust mantém sua ênfase na segurança com o uso de `RwLock`, reforçando seu alto nível de abstração. Go busca um equilíbrio entre controle e simplicidade, utilizando `sync.RWMutex`, o que resulta em expressividade média nas dimensões avaliadas. OpenMP, novamente, mostra-se restrito em flexibilidade e com baixa abstração, não oferecendo suporte nativo para controle refinado de acesso.

No problema do Barbeiro Dorminhoco, Rust alcança alta flexibilidade ao combinar canais assíncronos com *Mutex*, porém com alta complexidade. Go favorece a clareza da implementação ao empregar *channels* e *select*, com complexidade e abstração em níveis médios. OpenMP, mais uma vez, enfrenta limitações, com baixa expressividade geral, dada sua restrição estrutural e ausência de suporte direto à espera ativa.

Por fim, no problema dos Fumantes de Cigarro, Rust mantém o padrão de segurança elevado, utilizando *Condvar* para sincronização fina, o que implica em alta com-

plexidade. Go, mais conciso, oferece uma solução baseada em *channels*, que combina boa flexibilidade e complexidade moderada. OpenMP, com suas limitações em expressividade, demanda abordagens alternativas que tornam a implementação menos direta.

4.3. Esforço de Implementação

Com base nas métricas consolidadas de esforço apresentadas na Tabela 2, nota-se que Rust demanda, de modo geral, maior esforço de implementação, enquanto Go oferece um bom equilíbrio entre concisão e controle. OpenMP, por sua vez, reduz o esforço com menor detalhamento na gestão da concorrência.

No problema dos Filósofos, Rust requer alta quantidade de código para lidar com os estados dos recursos e evitar impasses, além de apresentar alta complexidade de controle. Go oferece uma solução mais direta, com esforço médio em termos de linhas de código e complexidade. OpenMP, com seu modelo baseado em diretivas, reduz o esforço com baixo volume de código e baixa complexidade, embora limite o controle fino da concorrência.

No problema do Produtor-Consumidor, Rust impõe um esforço elevado devido à necessidade de uso explícito de *Mutex* e *Condvar*, o que se reflete tanto em muitas linhas de código concorrente quanto em complexidade elevada. Go, por meio de *channels*, permite uma implementação mais enxuta, com complexidade e esforço médios. OpenMP apresenta baixo esforço em número de linhas, embora com acessibilidade alta às diretivas de paralelismo, facilitando a escrita.

Para o problema de Leitores e Escritores, Rust torna a exposição da concorrência mais clara, porém com alto esforço em código e controle. Go oferece esforço moderado, mantendo a clareza da implementação. OpenMP resulta na solução com menor esforço geral, mas com limitações no controle refinado dos acessos simultâneos, refletindo sua acessibilidade alta mas baixa flexibilidade.

No problema do Barbeiro Dorminhoco, Rust exige mais esforço ao empregar canais e *Mutex* de forma articulada, com alta complexidade e alto volume de código concorrente. Go simplifica esse processo com *channels* e *select*, proporcionando esforço médio. OpenMP, sem suporte direto para espera ativa, requer soluções alternativas, mas ainda assim mantém baixo volume de código, embora com menor naturalidade.

Por fim, no problema dos Fumantes de Cigarro, Rust impõe alto esforço de programação na coordenação de múltiplos agentes via *Condvar*. Go permite uma abordagem mais enxuta e intuitiva com *channels*, apresentando esforço médio. OpenMP, voltado para paralelismo de dados, mostra-se menos adequado, exigindo adaptações que comprometem sua acessibilidade e tornam a implementação menos natural.

4.4. Discussão Geral

A análise das métricas permite compreender como as linguagens avaliadas equilibram expressividade e esforço de programação.

OpenMP apresenta baixo esforço de implementação, evidenciado por métricas como LOC, LOC-P e CC em níveis baixos e ACU alta. Isso reflete um modelo acessível e direto, ideal para aplicações com paralelismo simples. No entanto, essa facilidade vem

à custa da baixa expressividade, com NLA e CMF reduzidos, o que limita a capacidade de modelar padrões de concorrência mais sofisticados.

Rust, por outro lado, oferece alta expressividade, especialmente nas dimensões de segurança (SSE: Alta), abstração (NLA: Alto) e flexibilidade (CMF: Alta). Essa expressividade, porém, está fortemente associada ao maior esforço, como indicam os altos valores de LOC, LOC-P e CC. A necessidade de controle explícito da concorrência, embora poderosa, exige mais atenção do programador e maior detalhamento na implementação.

Go posiciona-se como uma solução intermediária, com expressividade moderada a alta (NLA: Médio, SSE: Média-Alta, CMF: Alta) e esforço também moderado (LOC e CC: Médios). As construções nativas como canais e select permitem representar padrões concorrentes com clareza, mantendo o código relativamente simples. Isso torna Go adequada para aplicações que requerem bom controle da concorrência com um custo cognitivo mais baixo que o de Rust.

O estudo permitiu inferir que há uma relação inversa entre esforço e acessibilidade, e uma relação direta entre expressividade e complexidade, pelo menos nos casos de estudo relatados. Observou-se também que aplicações com maior complexidade de coordenação acentuam as diferenças entre as linguagens, favorecendo Rust e Go em termos de expressividade. Por outro lado, aplicações mais simples tendem a suavizar essas diferenças, tornando OpenMP uma opção mais competitiva em cenários menos exigentes.

5. Limitações do Trabalho

Este estudo comparou a expressividade e o esforço de programação em OpenMP, Rust e Go por meio de implementações de problemas clássicos da programação concorrente. No entanto, algumas limitações devem ser consideradas.

Cada problema foi implementado com uma única estratégia idiomática por linguagem: diretivas de paralelismo em memória compartilhada com OpenMP; `Arc`, `Mutex` e `Condvar` em Rust; e goroutines, canais e `sync`. Embora os recursos selecionados representem práticas comuns de programação em cada ferramenta, outras abordagens de implementação, como uso de outros recursos providos pelas ferramentas ou mesmo de bibliotecas externas ou ainda do modelos baseados em atores na linguagem Go, poderiam influenciar os resultados quanto à clareza, esforço e desempenho.

Além disso, o foco foi qualitativo, sem incluir métricas quantitativas como tempo de execução ou uso de memória, o que limita a aplicação dos resultados a cenários de produção. Também cabe notar que, por questões de espaço neste artigo, as informações sobre as métricas obtidas pela aplicação do modelo GQM desenvolvido foram apresentadas em termos qualitativos, na Tabela 2. O leitor interessado pode acessar no repositório disponibilizado para reprodutibilidade do experimento os dados numéricos completos.

6. Trabalhos Relacionados

Como trabalhos relacionados, esta seção aborda estudos que aplicaram o método GQM para avaliações de software. Além disso, são listados trabalhos que utilizaram Go e Rust para implementar aplicações concorrentes e paralelas. Trabalhos que empregaram OpenMP não foram considerados, visto que esta ferramenta é o padrão *de facto* [Cavalheiro 2020, Kasielke et al. 2019] para implementações visando explorar plataformas *multicore*, sendo amplamente utilizada em estudos científicos.

O método GQM foi empregado na definição de modelos de avaliação de software baseados em objetivos, questões e métricas por alguns estudos. [Youssef et al. 2021] propuseram um modelo GQM para recomendar padrões de projeto para implementações de software. [Effendi 2018] focaram em desenvolver um modelo GQM para gerar indicadores visando melhorar a qualidade do desenvolvimento de software. Já [Hernandes et al. 2012] definiram um modelo GQM para validar a usabilidade e facilidade de uso de uma ferramenta para auxiliar no processo de revisão sistemática de literatura. Além disso, o método GQM já foi utilizado para avaliar ferramentas de programação com recursos de concorrência e paralelismo. [D. Jardim et al. 2021] aplicaram um modelo GQM a fim de comparar programas que fizeram uso de memória transacional com OpenMP. Por fim, [Di Domenico 2022] propôs um modelo GQM para avaliar expressividade, esforço e desempenho de interfaces de programação para explorar GPUs.

Diversos estudos têm investigado a aplicabilidade das linguagens Go e Rust no contexto da programação concorrente e paralela. O trabalho de [Tipirneni 2022] apresenta uma análise geral sobre o uso de recursos de concorrência em Go, enquanto [Abhinav et al. 2020] realizaram uma comparação entre Go e Java, considerando tanto aspectos de desempenho quanto mecanismos de concorrência. Especificamente no contexto distribuído, [Pantoja 2019] propuseram uma biblioteca para a execução distribuída de programas escritos em Go. Em relação à linguagem Rust, há um volume crescente de trabalhos explorando suas capacidades para concorrência e paralelismo. Uma versão da suíte NPB implementada em Rust foi apresentada recentemente por [Martins et al. 2025], e estudos como os de [Pieper et al. 2021] e [Besozzi 2024] investigam bibliotecas e abstrações voltadas a esse domínio. Além disso, comparações de desempenho entre Rust e outras ferramentas têm sido conduzidas por diversos autores [Lewis et al. 2025, Zhang et al. 2023, Costanzo et al. 2021], sendo que [Costanzo et al. 2021] também examinou o esforço de programação envolvido.

7. Conclusão

Este estudo analisou a expressividade e o esforço de implementação de OpenMP, Rust e Go na programação concorrente, utilizando problemas clássicos como base de avaliação. A abordagem adotada, baseada no método Goal Question Metric (GQM), permitiu uma investigação estruturada sobre os desafios e benefícios dessas linguagens na modelagem de algoritmos paralelos e concorrentes.

Os resultados evidenciaram que Rust oferece o maior nível de segurança e controle sobre a concorrência, prevenindo condições de corrida por meio de seu sistema de posse e verificação de empréstimo. No entanto, esse rigor implica um esforço de implementação mais elevado, tornando o código mais complexo. OpenMP, por sua vez, simplifica a paralelização com diretivas `pragma`, reduzindo a quantidade de código necessária, mas limitando a flexibilidade da sincronização. Go apresentou uma solução intermediária, combinando um modelo acessível de concorrência baseado em *goroutines* e canais com uma implementação relativamente concisa e expressiva.

Além de contribuir para a compreensão do impacto dessas linguagens na programação concorrente, este trabalho reforça a importância de se considerar fatores como segurança, complexidade e flexibilidade ao escolher uma ferramenta para desenvolvimento paralelo. Os achados também se alinham às tendências recentes na adoção de lingua-

gens que priorizam segurança de memória e concorrência segura, como recomendado por instituições como a NSA.

Como continuidade desta pesquisa, sugere-se uma análise mais aprofundada do desempenho das implementações, explorando métricas como tempo de execução e consumo de recursos. Além disso, a inclusão de outras linguagens, como C++ e Java, poderia ampliar a comparação, fornecendo um panorama mais abrangente sobre as alternativas disponíveis para programação concorrente e paralela.

Referências

Abhinav et al. (2020). Concurrency analysis of go and java. In *5th Int. Conf. on Computing, Communication and Security*, pages 1–6.

Adams, J. C., Koning, E. R., and Hazlett, C. D. (2019). Visualizing classic synchronization problems: Dining philosophers, producers-consumers, and readers-writers. In *Proc. of the 50th ACM Technical Symposium on Computer Science Education*, page 934–940, New York. ACM.

Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, Cambridge.

Basili, V. R., Caldiera, G., and Rombach, D. H. (1994). *The Goal Question Metric Approach*, volume I. John Wiley & Sons.

Besozzi, V. (2024). Ppl: Structured parallel programming meets Rust. In *32nd Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing*, pages 78–87.

Qin et al. (2020). Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proc. of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 492–506. ACM.

Hoare, C. A. R. (1981). Communicating sequential processes. In *Proc. of the Int. Colloq. on Formalization of Programming Concepts*, pages 413–425. Springer.

Capel, M. I., Tomeu, A. J., and Salguero, A. G. (2018). A set of patterns for concurrent and parallel programming teaching. In *Euro-Par 2017: Parallel Processing Workshops*, pages 203–215, Cham. Springer International Publishing.

Costanzo et al. (2021). Performance vs programming effort between Rust and C on multicore architectures: Case study in n-body. In *XLVII Latin American Computing Conference*, pages 1–10.

D. Jardim et al. (2021). An extension for transactional memory in OpenMP. In *25th Brazilian Symposium on Prog. Languages*, page 58–65, New York. ACM.

Di Domenico, D. (2022). *A Model for Software Measurement Aiming to Guide Evaluations and Comparisons between Programming Tools to Implement GPU Applications*. PhD thesis, Federal University of Pelotas.

Di Domenico, D. and Cavalheiro, G. G. H. (2020). JAMPI: A C++ parallel programming interface allowing the implementation of custom and generic scheduling mechanisms. In *32nd Int. Symp. Comput. Archit. High Perform. Comput.*, pages 273–280.

Downey, A. B. (2008). *The Little Book of Semaphores*. Green Tea Press, Needham.

Martins et al. (2025). NPB-Rust: NAS Parallel Benchmarks in Rust.

Hernandes et al. (2012). Using gqm and tam to evaluate start – a tool that supports systematic review. *CLEI Electronic Journal*, 15.

Kasielke et al. (2019). Exploring loop scheduling enhancements in OpenMP: An llvm case study. In *18^{th} Int. Symp. on Parallel and Distributed Computing*, pages 131–138.

Lewis et al. (2025). Parallel n-body performance comparison: Julia, Rust, and more. In *Paral. and Distribut. Processing Techniques*, pages 20–31. Springer Nature.

Matsakis, N. D. and Klock, F. S. (2014). The Rust Language. In *Proc. of the 2014 ACM Annual Conf. on High Integrity Language Technology (HILT '14)*, pages 103–104.

Matsakis, N. D. and II, F. S. K. (2014). The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104.

Setiawan, R., Rasjid, Z. E., and Effendi, A. (2018). Design metric indicator to improve quality software development (study case: Student desk portal). *Procedia Computer Science*, 135:616–623. The 3rd Int. Conf. on Comp. Science and Comput. Intelligence.

Pieper et al. (2021). High-level and efficient structured stream parallelism for rust on multi-cores. *Journal of Computer Languages*, 65:101054.

Silberschatz, A., Gagne, G., and Galvin, P. B. (2018). *Operating System Concepts*. Wiley, Hoboken, NJ, USA, 10th edition.

Tanenbaum, A. S. and Bos, H. (2014). *Modern Operating Systems*. Pearson, Upper Saddle River, NJ, 4 edition.

Tipirneni, D. S. (2022). An empirical study of concurrent feature usage in go. Master's thesis, East Carolina University.

Tu et al. (2019). Understanding real-world concurrency bugs in go. In *Proc. of the 24 Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, page 865–878, New York. ACM.

Pankratius, V., Adl-Tabatabai, A.-R., and Garzaran, M. J. (2009). Software engineering for multicore systems: An experience report. *IEEE Software*, 26(6):20–29.

Whitney, J., Gifford, C., and Pantoja, M. (2019). Distributed execution of communicating sequential process-style concurrency: Golang case study. *J. Supercomput.*, 75(3):1396–1409.

Yakes, A. and Foley, S. S. (2021). A tool for visualizing classic concurrency problems. In *Proc. of the 52nd ACM Technical Symposium on Computer Science Education*, page 1375, New York. ACM.

Youssef et al. (2021). GQM-based tree model for automatic recommendation of design pattern category. In *Proc. of the 9^{th} Int. Conf. on Software and Information Engineering*, page 126–130, New York. ACM.

Zhang et al. (2023). Towards understanding the runtime performance of rust. In *Proc. of the 37th IEEE/ACM Int. Conf. on Automated Software Engineering*. ACM.

Liu et al. (2021). Automatically detecting and fixing concurrency bugs in go software systems. In *Proc. of the 26th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 875–888.