Comparação das linguagens Rust, Go e OpenMP na Implementação de Algoritmos Concorrentes e Paralelos

Lucas Braatz Araujo[1], **André Rauber Du Bois**[1], **Gerson Geraldo H. Cavalheiro**[1]

[1] Centro de Desenvolvimento Tecnológico – Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas (UFPel) – Pelotas, RS – Brasil

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

Resumo. *Este artigo apresenta um projeto que realiza um estudo comparativo entre Rust, Go e OpenMP no contexto de programação paralela e concorrente. A pesquisa analisa a expressividade e o esforço de programação na implementação de algoritmos clássicos de concorrência. Utiliza-se a metodologia GQM (Goal Question Metric), que orienta a avaliação com base em objetivos, questões e métricas. Resultados preliminares indicam que Rust oferece segurança e controle, Go se destaca pela simplicidade e OpenMP pela facilidade na paralelização incremental. Essas conclusões podem auxiliar na escolha da linguagem conforme requisitos de desempenho, segurança e implementação.*

1. Introdução

A crescente demanda por aplicações que exigem alta eficiência e desempenho tem impulsionado a exploração de novas abordagens e linguagens de programação. A programação paralela e concorrente, que visam otimizar o uso do hardware para aumentar a capacidade de processamento, tornaram-se essenciais. As técnicas iniciais, como o multiprocessamento e o paralelismo de tarefas, trouxeram avanços significativos, mas também novos desafios, como a sincronização [Pankratius et al. 2009].

Nesse contexto, o OpenMP facilitou a programação paralela em sistemas de memória compartilhada, destacando-se por sua simplicidade e eficiência [Dagum and Menon 1998]. Mais recentemente, novas linguagens como Rust e Go surgiram, oferecendo abordagens inovadoras para a concorrência e o paralelismo. Rust, por exemplo, é reconhecida por seu sistema de tipos rigoroso e verificações em tempo de compilação, que ajudam a prevenir problemas comuns na programação concorrente [Matsakis and Klock 2014]. Go, por outro lado, tem como um dos principais objetivos de design melhorar as linguagens de programação multithread tradicionais e tornar a programação simultânea mais fácil e menos propensa a erros [Lee et al. 2020].

Neste projeto é realizado um estudo comparativo entre Rust, Go e OpenMP no contexto da programação paralela e concorrente. Estas linguagens foram selecionadas pela sua relevância e diferença de abordagens, visando contribuir para a literatura existente e atualizar as práticas de ensino da programação concorrente.

2. Metodologia

A metodologia deste estudo envolveu uma série de etapas para atingir os objetivos específicos. Primeiramente, foi realizada uma pesquisa bibliográfica para identificar os aspectos

relevantes das linguagens de programação que seriam avaliados. Os aspectos selecionados foram expressividade, esforço de programação e eficiência.

Foram selecionadas três linguagens de programação distintas para explorar suas abordagens únicas à concorrência e paralelismo:

- **OpenMP**: Amplamente utilizado como um modelo de programação paralela para sistemas de memória compartilhada [Trobec et al. 2018].
- **Rust**: Escolhida por sua reputação de oferecer segurança de memória e alto desempenho [Matsakis and Klock 2014].
- **Go (Golang)**: Reconhecida por sua eficiente implementação de goroutines e simplicidade no gerenciamento de concorrência [Togashi and Klyuev 2014].

Após a definição das linguagens, foram selecionados algoritmos clássicos de programação concorrente e paralela para serem implementados em cada uma delas. Os algoritmos escolhidos foram: Jantar dos Filósofos, Produtor-Consumidor, Leitores-Escritores, Barbeiro Dorminhoco e Fumantes de Cigarro.

Em seguida, foi elaborado um modelo de medição de software baseado no método Goal Question Metric (GQM) [Basili et al. 1994]. O modelo GQM é uma abordagem estruturada que organiza o processo de medição em três níveis hierárquicos: objetivos, questões e métricas. O quadro na Figura 1 apresenta o modelo GQM elaborado.

GO1: Avaliar a expressividade da linguagem em fornecer abstrações 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 em um paradigma clássico?* • 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 CPR: Proporção de comentários no código. • M-2.2.3 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 elaborado para comparação entre implementações.

Com o modelo GQM definido, foram implementados os algoritmos clássicos em cada uma das linguagens selecionadas. Os experimentos foram guiados pelo modelo GQM, e os resultados foram utilizados para explorar as métricas obtidas e responder às questões propostas no modelo.

3. Resultados Obtidos

Na atual etapa do projeto estão sendo implementados algoritmos clássicos de concorrência e paralelismo em Rust, Go e OpenMP. Os resultados preliminares são exemplificados com o uso das linguagens na solução do problema do Produtor-Consumidor, permitindo

uma discussão detalhada das diferenças entre suas abordagens. Os códigos obtidos estão disponíveis em repositório público[1].

A análise desses trechos revela como Rust utiliza `Mutexes` e `Condvars` para gerenciar a concorrência, Go emprega `goroutines` e `channels`, e OpenMP se baseia em diretivas de compilador para paralelizar seções de código.

- **Rust**: Oferece segurança de memória e controle preciso sobre a sincronização, evitando corrida de dados em tempo de compilação. No entanto, o código pode ser mais complexo devido à necessidade de gerenciar explicitamente as travas e as condições.
- **Go**: Simplifica a concorrência com goroutines e canais, tornando o código mais leve e fácil de entender. A sincronização é implícita nas operações de envio e recebimento nos canais.
- **OpenMP**: Permite paralelizar seções de código em C++, facilitando a integração com aplicações existentes. No entanto, o gerenciamento da sincronização (com seções críticas) precisa ser feito manualmente, o que pode levar a erros.

As métricas definidas no modelo GQM obtidas para a implementação deste problema nas diferentes linguagens são apresentadas na Tabela 1.

Tabela 1. Métricas GQM para o Algoritmo Produtor-Consumidor.

Métrica	Rust	OpenMP	Go
GO1: Expressividade			
M-1.1.1 NLA	1	1	1
M-1.1.2 NCA	29	5	17
M-1.1.3 TCL	Arc, Mutex, Condvar, spawn, join, wait, notify_one, lock, unwrap, drop	parallel, critical, flush	go, Mutex, Lock, Unlock, Cond, newCond, Wait, Signal, Waitroup, Add, Done, Wait
M-1.2.1 SUA	10	4	10
M-1.2.2 SSE	Alta	Alta	Alta
M-1.3.1 CMI	Threads + Mutex/Condvar	Threads com OpenMP	Goroutines + Mutex/Cond
M-1.3.2 CMF	Flexível	Restrito	Flexível
GO2: Esforço			
M-2.1.1 LOC	81	65	80
M-2.1.2 LOC-P	29	7	17
M-2.2.1 CC	3	3	3
M-2.2.2 CPR	12%	25%	14%
M-2.2.3 ACU	Baixo	Médio	Baixo
M-2.3.1 NOPC	29	7	17
M-2.3.2 PELU	1	1	8

No que se refere à expressividade (GO1), Rust e Go oferecem abstrações de alto nível como threads e goroutines, além de canais (em Go com sync.Cond), enquanto OpenMP utiliza diretivas #pragma para paralelizar regiões de código, sendo menos flexível. Embora a contagem de NLA seja a mesma para as três, Rust se destaca com o maior número de ferramentas de controle (TCL). Em termos de sincronização (Q2), Rust e Go utilizam Mutex e Condvar para um bom controle, enquanto OpenMP depende de seções críticas, menos eficientes em alguns casos. No modelo de concorrência (Q3), Rust e OpenMP utilizam threads, enquanto Go usa goroutines, mais leves e gerenciadas pelo runtime. Rust e Go oferecem maior flexibilidade, enquanto OpenMP é mais restrito. Em

[1] `https://github.com/araujoblucas/ufpel-classical-concurrency-algorithms`.

termos de insights, Rust garante maior segurança contra condições de corrida devido ao seu modelo de posse, Go apresenta um modelo de concorrência mais leve e eficiente em determinadas situações, e OpenMP é mais adequado para paralelizar código existente, mas com menos flexibilidade.

No aspecto de esforço (GO2), OpenMP exige menos linhas de código (LOC e LOC-P) do que Rust e Go, devido ao uso de diretivas que alteram o comportamento do código sequencial. Rust e Go apresentam um número semelhante de LOC, indicando uma complexidade similar. Em termos de legibilidade (Q2), todas as linguagens têm a mesma complexidade ciclomática (CC). Quanto a abstrações claras (ACU), Rust e Go são considerados com ACU baixo, devido à necessidade de gerenciar threads e sincronização manualmente, enquanto OpenMP tem ACU médio, oferecendo abstrações mais altas, mas com menos flexibilidade. Em relação ao esforço no paralelismo e concorrência (Q3), OpenMP requer menos operações de paralelismo e concorrência, enquanto Go depende mais de funções externas. Insights revelam que OpenMP é eficiente para prototipagem e paralelização de código sequencial, mas pode precisar de mais explicações. Rust oferece segurança, porém com maior esforço, enquanto Go equilibra desempenho e facilidade de uso com um modelo de concorrência leve e canais.

4. Conclusão

Este estudo comparativo entre Rust, Go e OpenMP na programação concorrente e paralela revelou as características distintas, pontos fortes e pontos fracos de cada linguagem. Rust se destacou por sua segurança de memória e controle preciso sobre a concorrência, Go por sua simplicidade e eficiência no gerenciamento de goroutines, e OpenMP por sua facilidade de uso em sistemas de memória compartilhada.

Os resultados obtidos fornecem insights valiosos para desenvolvedores que buscam escolher a linguagem mais adequada para seus projetos de programação concorrente e paralela, considerando os requisitos de desempenho, segurança e facilidade de uso.

Referências

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

Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55.

Lee, K., Song, H., Kim, J., and Ryu, S. (2020). Understanding real-world concurrency bugs in go. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30.

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

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.

Togashi, N. and Klyuev, V. (2014). Concurrency in go and java: Performance analysis. In *2014 4th IEEE Inter. Conf. on Information Science and Technology*.

Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). Programming multi-core and shared memory multiprocessors using openmp. In *Introduction to Parallel Computing*, Undergraduate Topics in Computer Science, pages 75–108. Springer, Cham.