Programação Concorrente em Rust e Java: Uma análise comparativa de segurança, produção e performance

Lucas Borgomani Rezzaghi[1], André Marcos Silva[1]

[1]Centro Universitário Campo Limpo Paulista (UNIFACCAMP)
Jardim América – CEP 13231-230, Campo Limpo Paulista – SP – Brasil

lbrezzaghi@gmail.com, andre@faccamp.br

Abstract. *One of the main problems of a concurrent program in Java is when two objects share the same value in memory, one object reading and another writing at the same time. In Rust, this problem is solved with the ownership system, which ensures that each value has an owner, releasing the value only when the owner's lifecycle comes to an end. In this way, Rust guarantees and manages memory at compile time. In this article, we will make a comparative study between Rust and Java in the concurrent and performance aspects, concluding which language provides the most secure memory management.*

Resumo. *Um dos principais problemas de um programa concorrente em Java é quando dois objetos compartilham o mesmo valor na memória, um objeto lendo e outro escrevendo ao mesmo tempo. Em Rust, esse problema é resolvido com o sistema de ownership, que faz com que cada valor tenha um dono, liberando o valor apenas quando o ciclo de vida do dono chega ao fim. Desse modo, Rust assegura e gerencia a memória em tempo de compilação. Este artigo irá fazer um estudo prático comparativo entre Rust e Java no aspecto concorrente e performático, concluindo qual linguagem garante o gerenciamento de memória mais performático.*

1. Introdução

Em diversas aplicações, múltiplas atividades ocorrem simultaneamente. Muitas delas podem sofrer bloqueios intermitentes, razão pela qual as *threads* foram criadas. Apesar de *threads* não garantirem necessariamente um aumento de performance em todas as situações, quando há intensa atividade de *input/output* de dados e grande carga computacional, elas possibilitam a intercalação dessas atividades, acelerando o desempenho da aplicação. Por exemplo, numa aplicação de edição de texto, é viável dividir as tarefas entre várias *threads*: uma aguarda a entrada do usuário via teclado, enquanto outra renderiza o texto e o exibe na tela (Tanenbaum, 2014).

No entanto, essa divisão de tarefas entre *threads* pode gerar complicações. Por compartilharem memória, quando algumas *threads* têm a capacidade de ler e escrever entre si, podem acabar sobrescrevendo o mesmo dado compartilhado. Isso resulta em um comportamento inesperado chamado de *Race condition* (Tanenbaum, 2014), apesar de o programa continuar executando normalmente, o que pode dificultar a depuração do código.

Essa anomalia tende a ocorrer sobretudo em linguagens que permitem o uso de *aliasing*. No contexto de programação, duas variáveis, também conhecidas como *l-values*, são consideradas aliases quando, em algum momento da execução do

programa, se referem ao mesmo local de memória (Ramalingam, 1994); por exemplo, quando duas diferentes variáveis do tipo ponteiro possuem ou recebem um mesmo valor literal, em tempo de execução, durante uma chamada de função com passagem de parâmetro por referência. A linguagem Java é particularmente suscetível a tais anomalias em razão do uso de *aliasing* e mutabilidade, que permitem que dois *l-values* diferentes alterem o mesmo valor, tal como descrito no problema de *race condition*.

Por outro lado, a linguagem Rust atenua esses problemas de *aliasing* e mutabilidade com seu principal mecanismo, o *ownership* (Troutwine, 2018). O conceito de *ownership*, uma das principais estratégias de segurança do Rust, permite que cada valor tenha apenas um proprietário, e que este valor seja liberado quando a vida útil do proprietário termina. O Rust aprimora essa regra fundamental com um conjunto de normas que mantêm a segurança da memória e das *threads*. Por exemplo, a propriedade pode ser emprestada ou transferida, e vários *aliases* podem ler um valor. Essas regras de segurança proíbem, essencialmente, a combinação de *aliasing* e mutabilidade. O Rust realiza a verificação dessas regras de segurança durante a compilação, atingindo um desempenho em tempo de execução que é compatível com linguagens consideradas inseguras, Contudo, ainda proporciona garantias de segurança significativamente mais robustas (Klabnik, 2022).

Este artigo, apresenta uma análise comparativa entre algoritmos concorrentes escritos em Java e Rust. Foco será examinar a corretude de ambos os exemplos, ou seja, avaliar qual dos códigos demonstra maior previsibilidade e determinismo. Também, são consideradas a performance e a segurança de memória desses códigos em execução, utilizando os mesmos algoritmos e reportaremos o tempo total necessário para a conclusão da execução em ambas linguagens.

2. Objetivos e Metodologia

Serão simulados ambientes para permitir comparações analíticas tanto no nível de produção quanto no de execução entre as linguagens Rust e Java, utilizando um algoritmo de *merge sort multithread* para ordenar *arrays* pequenos (1000 elementos), médios (100000 elementos) e grandes (1000000 elementos). O algoritmo será baseado, principalmente, em Cormen (2009). Neste caso, corretude e segurança se referem ao determinismo e previsibilidade do código antes da execução, enquanto performance diz respeito ao tempo total necessário para executar todo o programa.

Para efetuar a comparação de performance, utilizaremos as bibliotecas nativas de tempo, próprias, de cada linguagem. Já para analisar a corretude, examinaremos a tipagem do código e o número de erros que são detectados previamente pelo compilador antes da execução do programa. Quanto à segurança, analisaremos quais linguagens são mais aptas a ter possíveis problemas relacionados ao compartilhamento de recursos entre *threads* durante a execução.

3. Desenvolvimento

3.1. Medição Efetiva e Seleção de Métricas

A técnica de avaliação para realização da comparação proposta é baseada na Medição Efetiva (Cancin, 2001), na qual um ambiente real deve ser base para coleta de dados observáveis também reais. Quanto às métricas de avaliação, necessárias para definição e

concretização do ambiente e dos dados de análise (Jain, 1991), são vetores para a comparação neste estudo: (a) *tempo* de resposta de execução, (b) *uso de recursos de máquina* e (c) *complexidade de desenvolvimento*; que são observados ao longo de uma proposta diversificada de plataforma, a fim de estabelecer fatores (ou níveis) livre de vícios e mais justos de comparação (Jain, 1991). São quatro (4) os fatores desenhados para observação deste projeto (Tabela 1). Estas fatores vão desde um nível mais baixo, considerando uma linguagem de alto nível de programação até um nível mais alto de abstração de sistema, diretamente conectado ao usuário, que seria uma camada de usuário, ou seja, aplicações clientes.

Tabela 1. Fatores (ou Níveis) de execução para coleta de dados observáveis

NÍVEL	DESCRIÇÃO	OBSERVAÇÃO	TOOLS/APOIO
1 - execução	ambiente de execução (usuário)	tempo de resposta	LoadRunner (proposta)
2 - Devops	ambiente de execução (desenvolvimento)	complexidade de desenvolvimento	Eclipse e VSCode
3 - S.O.	ambiente de execução (gerenciamento de recursos)	performance, gerenciamento de recursos de máquina	HWMonitor, HW Info (proposta)
4 - unitário	código de programação e bibliotecas nativas de apoio	performance	bibliotecas e extensões de depuração (*debug*)

Durante a fase de coleta de dados para comparação, os diferentes fatores (Tabela 1) permitem que as informações de reprodução sejam cruzadas, permitindo uma maior consistência das conclusões. Para melhor refinar os critérios de variância dos dados, os níveis de execução serão triplicados em três diferentes plataformas operacionais: *Linux (Ubuntu 20.04 LTS), Windows (Professional 10, 64 bits) e Apple (MacOS Ventura 13.4.1)*; todos em instância de sistema operacional dedicado e local, rodando em ambiente de *hardware* nas especificações de um AMD Ryzen 5 1600, e de 16Gb de RAM, e o MacOS na especificação de um chip M1 com 8Gb de RAM.

3.2. Carga de Trabalho

Para efetuar a comparação entre os algoritmos, conforme Cancin (2001), utilizaremos o Visual Studio Code como editor de texto para a linguagem Rust, enquanto o Eclipse será usado para a linguagem Java. Para preparação do *benchmark* de teste, ambas linguagens utilizarão suas respectivas bibliotecas padrão de tempo, `std::time` para Rust e `java.time` para Java.

O algoritmo base construído para execução dos testes que serão piloto para as análises comparativas, é composto basicamente por duas funções bem definidas que manipulam uma estrutura de dados nativa de cada linguagem (github.com/rezzaghi/).

- `merge_sort(lista)`: Opera utilizando a estratégia de dividir para conquistar, decompõe recursivamente o vetor em duas metades, esquerda e direita, até que cada subconjunto contenha apenas um elemento. após a decomposição, as metades são progressivamente recombinadas em uma sequência ordenada, concluindo a ordenação.
- `merge(lista_a, lista_b)`: Combina dois vetores ordenados em um único vetor ordenado. Compara iterativamente os primeiros elementos de ambos os vetores, remove o menor e o adiciona ao vetor resultado até que ambos os vetores estejam vazios.

Na preparação do ambiente de teste, é empregada a estratégia de *work stealing* para paralelizar o algoritmo de forma eficaz. Em Rust, esta estratégia é implementada através do método `join()` da biblioteca Rayon, enquanto em Java, é aplicada por meio da classe `ForkJoinPool`. Tanto em Rust quanto em Java, a estratégia de *work stealing* opera da mesma forma. Quando um processo está carente de trabalho, ele "rouba" *threads* de outros processos, dessa forma, maximizando o aproveitamento de recursos disponíveis, e aumentando significativamente a performance (Blumofe, 1999).

3.3. Simulações Iniciais

Utilizando os métodos de depuração nativos de cada linguagem durante a execução, os seguintes resultados (Tabela 2) foram obtidos em uma sessão de simulações baseadas em coleta de dados por bibliotecas de testes unitários na camada de código fonte.

Tabela 2. Simulações baseadas em testes unitários no nível de código fonte

	10000 ($10,10^3$)		100000 ($10,10^4$)		1000000 ($10,10^5$)	
Plataforma OS	**Rust**	**Java**	**Rust**	**Java**	**Rust**	**Java**
Windows	9 ms	96 ms	85 ms	759 ms	772 ms	47938 ms
Linux	11 ms	53 ms	78 ms	643 ms	678 ms	49605 ms
MacOS	10 ms	38 ms	57 ms	410 ms	600 ms	38914 ms

4. Conclusão

Os resultados desta etapa do projeto foram focados na descrição da construção da estrutura de comparação e definição dos critérios de análise. Como atividade preliminar de validação desta estrutura, foram realizadas algumas das simulações do processo, antecipando um dos níveis de execução, no nível de código unitário. Nesta simulação, a Análise Comparativa ficou evidente que, com uma base de código mais sucinta, Rust demonstrou um resultado menos suscetível a falhas e com performance relativamente favorável em relação a Java. Devido ao seu sistema de *Ownership*, Rust detecta erros durante o tempo de compilação evitando a necessidade de um *garbage collector*, como ocorre em Java, devido a este fato, rust tem uma performance relativamente maior em todos os dados testados. Essa diferença de abordagens ressalta a capacidade do Rust em oferecer segurança e eficiência sem comprometer o desempenho, tornando-se uma opção mais valiosa. Estes resultados parciais, estão servindo como orientação para definição e possibilidades das futuras etapas do projeto. Atualmente está em curso a preparação dos laboratórios para simulações das execuções seguindo as estratégias já definidas; entre elas, as execuções monitoradas por ferramentas de apoio nos níveis de S.O., com HWMonitor, e no nível do usuário, com LoadRunner.

Referência Bibliográfica

Blumofe, R. Leiserson, C. (1999) "Scheduling multithreaded computations by work stealing". Journal of the ACM (JACM), v. 46, n. 5, p. 720-748, setembro, 1999.

Cancin, R. L. (2001). “Avaliação de desempenho de algoritmo de escalonamento de tempo real para o ambiente do multicomputador Crux”, Dissertação de Mestrado, UFSC, Florianópolis, agosto de 2001.

Cormen, T., Leiserson, C. E. Rivest, R. L., e Stein, C. (2009). “Introduction to algorithms”, Ed. The MIT Press; ed. 3a., setembro, 1999.

Jain, R. (1991) "The Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling". New York, US, Ed. John Wiley & Sons, Inc,, janeiro de 1991.

Klabnik, S. e Nichols, C. (2022). “The Rust Programming Language”, Ed. Starch Press, ed. 2th, dezembro de 2022.

Ramalingam, G. (1994). “The undecidability of aliasing”, ACM Transactions on Programming Languages and Systems, v. 16 Issue 5., pp. 1467–1471; setembro de 1994.

Tanenbaum, A. (2014). “Modern Operating Systems”, Prentice Hall; ed. 4a. março de 2014.

Troutwine, B. (2018). “Hands-On Concurrency with Rust: Confidently build memory-safe, parallel, and efficient software in Rust”; Ed. Packt Publishing; maio de 2018.