Traçando um paralelo entre C++ e Rust para programação multithread

Freddy da Paz Ilha[1], **Gerson Geraldo H. Cavalheiro**[1]

[1]Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas (UFPEL)
Caixa Postal 354 – 96.010-610 – Pelotas – RS – Brasil

{freddy.ilha,gerson.cavalheiro}@inf.ufpel.edu.br

Resumo. *O trabalho objetiva comparar recursos multithread em C++ e Rust, linguagens com abordagens distintas no gerenciamento de memória e concorrência. Enquanto C++ oferece grande flexibilidade, responsabilizando o programador pelo controle dos recursos, Rust impõe restrições em tempo de compilação por meio do sistema de ownership e borrow checker. A comparação abrange três eixos principais: threads, ponteiros e comunicação entre threads, analisando como os mecanismos da linguagem afetam o desenvolvimento concorrente. Apoiando-se em trabalhos que indicam desempenho similar entre as linguagens, e propor usando o modelo GQM para estruturar a comparação de expressividade e equivalência de implementação em etapas futuras.*

1. Introdução

C++ é uma linguagem consolidada e muito utilizada na indústria [BV 2026], lançada em 1985. A maturidade da linguagem e a expertise dos seus programadores elevou o padrão de desenvolvimento, de forma a incorporar várias camadas de segurança aos programas. A linguagem também evoluiu incorporando recursos para programação *multithread*, uma necessidade para as linguagens modernas desde o advento das arquiteturas *multicore*.

Rust, por sua vez, lançado em 2015, desenvolveu-se já inserida em um contexto de prevalência de arquiteturas *multicore*. Além disso, o cenário propiciou o pensamento voltado à segurança de código, implicando à verificações adicionais em tempo de compilação, assim, prevenindo que diversos erros comuns aconteçam durante a execução.

O desenvolvimento de aplicações *multithread* em linguagens de sistemas envolve o controle de memória, a sincronização e a comunicação entre *threads*. Em linguagens tradicionais como C++, esses mecanismos oferecem grande flexibilidade, mas exigem cuidado do programador para evitar erros como condições de corrida, ponteiros inválidos e vazamentos de memória. Rust, por sua vez, adota uma abordagem distinta, baseado em *ownership* e verificações em tempo de compilação, com o objetivo de aumentar a segurança do programa na manipulação de memória e no acesso concorrente a dados.

Apesar dessas diferenças, ainda não está evidente até que ponto ambos os modelos permitem implementar soluções *multithread* com níveis semelhantes de expressividade, tampouco quais são as implicações práticas dessas diferenças para o desenvolvimento de software concorrente. A hipótese sugere que ambas as linguagens possuem recursos de programação bastante próximos como ponteiros, threads e outros, por outro lado, diferenciam-se por C++ oferecer esses recursos ao programador, permitindo a escolha

sobre sua utilização. Em Rust, no entanto, alguns recursos são obrigatórios, como o controle de *ownership* e de *lifetimes*, influenciando a forma como programas concorrentes são estruturados.

Neste contexto, este trabalho propõe-se a traçar um paralelo dos recursos para programação *multithread* nas linguagens C++ e Rust. Para isso, será empregado o modelo GQM (*Goal*, *Question*, *Metrics*) utilizado na tese do [Di Domenico 2022], com o objetivo de definir metas, questões e métricas que para analisar e comparar o desenvolvimento de programas concorrentes em ambas as linguagens.

2. Modelo de Programação

O controle de memória consiste em uma prática comum em linguagens de baixo nível, em que faz-se necessário alocar e desalocar memória. Esta tarefa é de responsabilidade do programador, pois, propõe-se a evitar erros no programa, como vazamentos de memória. Uma das práticas desenvolvidas com o objetivo de reduzir o erro humano intitula-se *Resource Acquisition Is Initialization*(RAII), o conceito resume-se em uma abordagem que permite vincular o tempo de vida de um recurso previamente ao uso, garantindo a liberação automática da memoria ao ausentar-se do escopo, removendo, assim, a necessidade do controle manual da memória.

2.1. Threads

A biblioteca padrão de C++ std::thread trouxe maiores possibilidades à linguagem, no entanto, o controle dos recursos está atrelado ao desenvolvedor, uma vez que necessitam ser monitorados durante todo o ciclo de vida das *threads*. Assim, para esta biblioteca, não utiliza-se as vantagens do padrão RAII, ou seja, se porventura o desenvolvedor não executar o *join*/*detach* no fim da *thread*, o programa é terminado com erro.

Em 2020, com o lançamento do padrão C++20, foi introduzida a classe std::jthread, que segue o princípio RAII. Diferentemente de std::thread, seu destrutor invoca automaticamente o método *join*, assegurando a correta finalização da *thread* e evitando erros críticos. Além disso, a biblioteca introduz o mecanismo *stop_token*, que possibilita a parada cooperativa de modo seguro, permitindo interromper tarefas específicas sem causar falhas no programa.

Já em Rust, o módulo std::thread, apresenta uma funcionalidade semelhante a std::thread de C++. Entretanto, existem algumas diferenças fundamentais entre as duas abordagens. Em Rust, não há um mecanismo nativo equivalente ao *Stop Token*. Além disso, ao finalizar uma *thread* sem que o método *join* seja chamado, o *Detach* será executado permitindo a finalização da *thread*, e dessa forma, evitando erros críticos. Mesmo sem *Stop Token*, em Rust é possível empregar diferentes abordagens para controlar a execução de *threads*. O uso de *Channels* permite a comunicação segura entre *thread*, por meio desse processo, uma *thread* pode enviar instruções a outra *thread*, tornando viável a implementação de padrões de parada cooperativa, sem compartilhamento de memória.

2.2. Ponteiros

Outro comparativo relevante refere-se ao uso de ponteiros. Em C++, existem dois tipos amplamente utilizados. O primeiro são os *raw pointers*, que correspondem aos ponteiros comuns da linguagem, nos quais ocorre a alocação e liberação manual de memória por

meio de *new* e *delete*. Considera-se boa prática atribuir nullptr após a liberação do recurso, a fim de evitar ponteiros inválidos. O segundo tipo são os *smart pointers*, disponibilizados na biblioteca padrão *memory* [cppreference.com 2026]. Esses ponteiros encapsulam e automatizam a gestão de memória. Os três principais são: *unique_ptr*, que garante exclusividade de posse, transferindo-a via move e liberando a memória ao sair do escopo; *shared_ptr*, que permite posse compartilhada entre múltiplos ponteiros por meio de contagem de referências; e *weak_ptr*, que atua como ponteiro observador sem participar da contagem, sendo útil para evitar ciclos de referência.

Em Rust, o gerenciamento de memória é baseado no sistema de *Ownership* e no *Borrow Checker*, no qual cada valor possui um único proprietário e é desalocado automaticamente ao sair do escopo, tornando o padrão RAII obrigatório [The Rust Project Developers 2026]. Rust também permite o empréstimo de valores por referências, admitindo múltiplas referências imutáveis ou uma única mutável, nunca simultaneamente [The Rust Programming Language 2026].

Embora esse modelo trate, assim como os *smart pointers* de C++, do controle de tempo de vida dos dados, trata-se de uma abordagem mais abrangente, integrada à linguagem e verificada em tempo de compilação. Dessa forma, Rust previne erros como vazamentos de memória, dupla desalocação e uso de ponteiros após liberação.

2.3. Comunicação

Aborda-se, por fim, a comunicação entre *threads*. Em C++, o padrão mais comum baseia-se no compartilhamento de memória, geralmente por meio de mecanismos de sincronização como mutex. Nesse contexto, o desenvolvedor é responsável por garantir o correto bloqueio e desbloqueio do recurso compartilhado, evitando condições de corrida. A biblioteca padrão oferece *condition_variable*, que permite implementar sistemas de notificação entre *threads*, nos quais uma *thread* pode aguardar até que outra sinalize uma determinada condição. Outro recurso amplamente utilizado são os tipos atômicos (std::atomic), que possibilitam operações *lock-free* e instruções atômicas diretamente suportadas pela CPU. Esse mecanismo é empregado para tipos simples, como inteiros, booleanos e ponteiros, permitindo sincronização eficiente sem a necessidade de bloqueios.

Quando comparado, Rust incentiva o uso de *channels* para a comunicação de *threads*. Nesse padrão, um valor pode ser enviado por um canal, sua posse é transferida e posteriormente ao receptor. Esse mecanismo reduz a necessidade de compartilhamento de memória, contribuindo para evitar condições de corrida. Rust realiza verificações de tipos e de segurança em tempo de compilação, de modo a assegurar que apenas tipos que atendam os requisitos de envio (*Send*) e compartilhamento seguro (*Sync*) possam ser transferidos ou acessados entre *threads*, reduzindo erros comuns em programas concorrentes.

3. Trabalhos relacionados

Neste trabalho, considera-se a existência de diversos estudos previamente publicados que comparam as linguagens C++ e Rust em termos de desempenho. No estudo apresentado por [Ivanov 2022], foram realizados diversos testes envolvendo tarefas comuns, como ordenação e manipulação de dados, nos quais se observou que ambas as linguagens apresentam desempenho aproximado. Em alguns cenários, Rust demonstrou resultados superiores, enquanto em outros C++ apresentou melhor desempenho.

Outro trabalho relevante é apresentado em [Martins et al. 2025], no qual a linguagem Rust foi avaliada utilizando o conjunto de testes NAS *Parallel Benchmarks*, sendo comparada com implementações em C++ e Fortran. Os resultados indicaram que Rust apresentou melhor desempenho em execuções sequenciais, enquanto demonstrou desempenho inferior em ambientes paralelos.

O modelo GQM, proposto por Basili, estrutura a medição de software em três níveis hierárquicos, garantindo que apenas dados relevantes aos objetivos definidos sejam coletados [Di Domenico 2022]. No primeiro nível, as metas definem o objeto e o propósito da análise, por exemplo: avaliar o impacto do sistema de *Ownership* na segurança de programas concorrentes; no segundo nível, as perguntas especificam como verificar se a meta foi atingida, por exemplo: quais classes de erros de concorrência são prevenidos?; por fim, no último nível, as métricas definem as medições que respondem uma ou mais perguntas, por exemplo: número de erros de sincronização capturadas em tempo de compilação contra em tempo de execução.

4. Conclusão

A análise bibliográfica realizada indica que C++ e Rust apresentam capacidade computacional semelhante, oferecendo recursos equivalentes para programação *multithread*, ainda que com abordagens distintas. No entanto, a maior parte dos estudos existentes concentra-se na avaliação de desempenho, havendo menor ênfase na análise da expressividade e das implicações práticas dessas abordagens no desenvolvimento. Nesse contexto, utilizando o modelo GQM este trabalho irá avaliar as características relacionadas à expressividade e à forma como os mecanismos de concorrência influenciam o desenvolvimento a fim de estabelecer uma equivalência de implementação de software concorrente.

Referências

BV, T. S. (2026). The c++ programming language – tiobe index. `https://www.tiobe.com/tiobe-index/cplusplus/`. Accessed: 2026-03-04.

cppreference.com (2026). std::memory header - cppreference.com. `https://en.cppreference.com/w/cpp/header/memory.html`. Accessed: 2026-03-04.

Di Domenico, D. (2022). *A Model for Software Measurement Aiming to Guide Evaluations and Comparisons between Programming Tools to Implement GPU Applications*. Doctoral dissertation, Federal University of Pelotas, Pelotas, Brazil. Accessed: 2026-03-04.

Ivanov, N. (2022). Is rust c++-fast? benchmarking system languages on everyday routines. Accessed: 2026-03-04.

Martins, E. M., Faé, L. G., Hoffmann, R. B., Bianchessi, L. S., and Griebler, D. (2025). Npb-rust: Nas parallel benchmarks in rust. Accessed: 2026-03-04.

The Rust Programming Language (2026). References and borrowing. `https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html`. Accessed: 2026-03-06.

The Rust Project Developers (2026). What is ownership? `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`. The Rust Programming Language, Chapter 4. Accessed: 2026-03-07.