Analisando Paralelismo de Dados em Rust Usando o Método do Gradiente Conjugado

Lucas S. Bianchessi[1], **Leonardo G. Faé**[1], **Renato B. Hoffmann**[1], **Dalvan Griebler**[1]

[1]Politecnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

l.bianchessi@edu.pucrs.br, leonardo.fae@edu.pucrs.br, renatowbhf@gmail.com

Resumo. *Em meio ao ambiente da computação de alto desempenho, a linguagem Rust vem se tornando cada vez mais popular, prometendo segurança, desempenho e um ambiente de desenvolvimento moderno. Afim de analisar a viabilidade e eficiência do Rust, foi utilizado método do gradiente conjugado do NPB benchmarks. Os resultados demonstraram resultados paralelos comparáveis ao C++, e perda de desempenho na versão sequencial.*

1. Introdução

É estimado que entre 49 e 88 por cento dos bugs em software são causados por *memory unsafety* [Sudwoj 2020]. Rust tem um grande foco na área de segurança ao apresentar o sistema de *posse*. Esse sistema tem custo quase zero, já que as verificações são realizadas em tempo de compilação. Sendo assim, programas Rust diminuem significativamente a preocupação referente a segurança de memoria e concorrência.

O *NAS Parallel Benchmarks* (NPB) foi originalmente desenvolvido em Fortran pela divisão de supercomputação avançada da NASA com o intuito de mensurar, objetivamente, o desempenho de computadores altamente paralelos. O NPB, contém 8 aplicações matemáticas, sendo algumas delas o método do gradiente conjugado, transformada rápida de Fourier, e também 3 aplicações de simulação de dinâmica de fluidos. Devido à complexidade e importância das aplicações contidas, o NPB ganhou popularidade, e eventualmente foi convertido para C++ [Griebler et al. 2018, Löff et al. 2021].

A Rust já é utilizada no desenvolvimento de aplicações em ambientes de produção, devido as garantias de segurança e velocidade de aplicações. No entanto, não existem muitos estudos avaliando sua eficácia em aplicações numéricas de alta complexidade. Para abordar essa questão, este estudo desenvolveu uma versão da aplicação "Gradiente Conjugado" (CG) do NPB para a linguagem Rust, com implementações sequencial e paralela. As contribuições científicas deste artigo incluem o desenvolvimento deste programa e a análise e comparação de seu desempenho com C++.

Em seguida, a Seção 2 fala sobre alguns trabalhos relacionados e a Seção 3 descreve questões sobre a implementação da CG em Rust. Posteriormente, os resultados obtidos foram exibidos na Seção 4; por fim, a Seção 5 apresenta as considerações finais.

2. Trabalhos Relacionados

A linguagem de programação Rust apesar de ser nova, já foi comprovada como apta a integrar o cenário de computação de alto desempenho, apesar que atualmente não é muito

adequado par uso em GPUs, reforçado em [Sudwoj 2020]. Diferentemente do atual artigo, que apesar de servir como reforço para a ideia da Rust como linguagem de alto desempenho, toma como foco a capacidade de paralelização da lingua. Em [Ivanov 2022] implementa alguns diferentes tipos de *sorts* e os compara entre diversas linguagens, porém não focou muito na parte de sua implementação para supercomputadores. Ambos os estudos concluíram que a performance das aplicações em Rust, atingiu, em sua maioria, performances muito semelhantes ás suas versões equivalentes em línguas já mais estabelecidas.

É importante ressaltar que este presente artigo é a continuação do artigo [Zomer et al. 2023], no qual foi a aplicação embarrassingly parallel(EP) foi transformado em Rust, porém não foi utilizado a biblioteca Rust SSP para paralelização pois se foi julgado indevida para o uso em dada aplicação.

3. Método do Gradiente Conjugado em Rust

O Benchmark do NPB escolhido para realizar os testes foi o conjugado gradiente, que realiza a conjugação do gradiente de autovetores de uma matriz. A conversão do código foi uma conversão de sintaxe entre as linguagens, sendo que não faltou nenhuma ferramenta no Rust para implementar a aplicação. Dentre as mudanças de sintaxe mais notáveis, foi a alocação de vetores sem o uso de um equivalente ao `malloc` do C++.

Para o paralelismo, foi utilizada a biblioteca Rayon, que facilita a programação paralela e concorrente em Rust. A principal abstração fornecida por Rayon é o conceito de iteradores paralelos. Eles permitem transformar um laço de repetição sequencial em uma operação paralela com relativamente pouco esforço, análogo ao OpenMP *parallel for*. Entretanto, para garantir segurança, todos os dados dos laços são considerados automaticamente imutáveis, desde que não especificado o contrário, assim custando desempenho. Para extrair dados dos laços Rayon, geralmente é usada uma operação de acumulação, redução, ou coleta de um resultado de retorno.

A conversão dos laços de repetição paralelos foi realizada com iteradores paralelos, devido a forma como a biblioteca de paralelismo Rayon foi implementada. Outra questão da versão Rayon é foram as matrizes, que na versão em C++ foram alocadas de forma contígua. Em Rust, foi utilizado um cast para tratar as matrizes como um *array* de *arrays*, foram substituídas por vetores indexados de acordo com a operação *x1 * x0 + y1*,sendo x0 o tamanho da dimensão x, x1 e y1 os índices desejados. Dessa forma, é possível transformar as matrizes em um único iterador paralelo.

4. Experimentos

Os experimentos foram realizados em uma máquina com AMD Ryzen 5 5600X (6 cores, 12 *threads*) e 12 GB de RAM. O código Rust foi compilado com a versão 1.75.0 do rustc, por meio do perfil *release* do Cargo e 1.8 do Rayon. Para C++, foi usado usado g++ 9.4.0 com a opção `-O3` de compilação, além de `-std=c++14` e `-mcmodel=medium`, no caso do C++. Os resultados foram obtidos a partir da média aritmética de dez execuções.

O gráfico (Figura-1) representa a relação entre o tempo e o numero de *threads* utilizadas na execução da aplicação, que variou entre "zero" (aplicação em sequencial) e doze *threads*. Existem duas conclusões notáveis: primeiro, a versão sequencial C++ tem

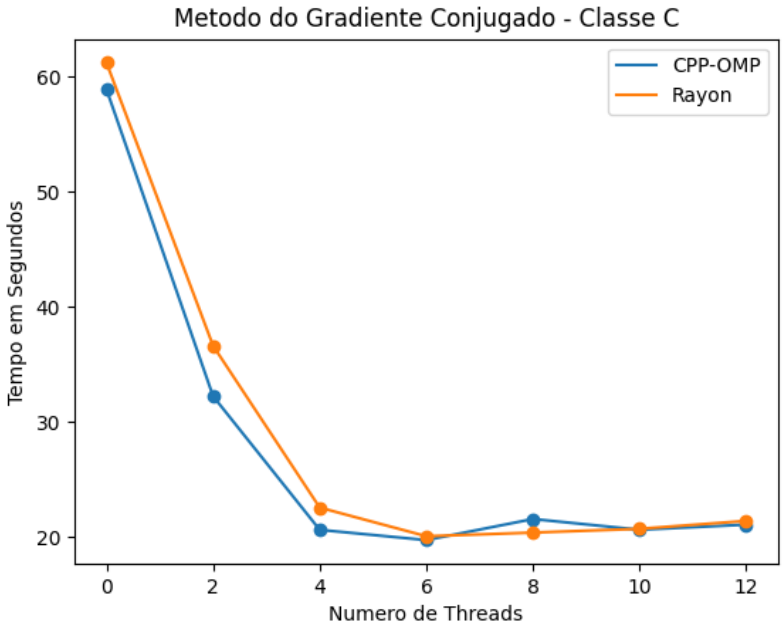

Figure 1. (Figura-1) - Método do Gradiente Conjugado. C++ vs Rust.

desempenho melhor do que Rust e, segundo, a versão Rust tem desempenho comparável à C++ com um número de trabalhadores paralelos maior. A maior discrepância entre as duas aplicações foi a execução com duas *threads* sendo o pico do teste inteiro com uma diferença de em torno de 12%, e a segunda maior discrepância foi com quatro *threads* chegando a 10% por cento de diferença no tempo de execução à favor do C++. Já na versão com oito *threads*, Rust foi 5% mais rápido. As outras execuções não demonstraram diferenças de porcentagem maiores do que 1%.

Apesar das maiores discrepâncias apresentadas a versão em C++ e Rust tem seus respectivos *runtimes* extremamente semelhantes, com uma diferença percentual irrelevante. Um dos principais fatores que explica a diferença de desempenho é a verificação de limites para todos os acessos a vetores em Rust. Isso foi comprovado através da mudança de 2 trechos de código crítico. Adicionando código *Unsafe*, que remove a verificação de limites, foi possível melhorar o desempenho sequecial Rust em aproximadamente 10%, resultado esse que já está representado no gráfico.

Outro ponto importante de desempenho em operações vetoriais Rust é a repetição inclusiva. Ao remover a sintaxe da repetição inclusiva `for i in 1..=n`, e substituir por uma repetição não-inclusiva equivalente `for i in 1..n-1`. Dessa forma, foi possível melhorar o desempenho do Rust sequencial em aproximadamente 20%, resultado esse que também já está representado no gráfico. A explicação é que o compilador Rust gera um código com instruções a mais que são executadas à cado laço para garantir que não ocorrem erros de memória.

5. Conclusão

Neste artigo foi testada a eficiência da linguagem Rust em uma aplicação do método do gradiente conjugado, que pertence ao NAS Parallel Benchmarks(NPB)

[Bailey et al. 1995]. Os resultados demonstraram que o desempenho da aplicação paralela usando Rust, com instancias *Unsafe*, teve um desempenho semelhante a versão em C++. Apesar da utilização de 2 instâncias de código *unsafe*, a aplicação se manteve próxima das boas praticas da linguagem. Como trabalhos futuros, será realizada a implementação de outras aplicações do NPB em Rust, com o intuito de completar o conjunto experimental e revelar mais questões de desempenho da linguagem Rust.

References

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M. (1995). The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center.

Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018). Efficient NAS Benchmark Kernels with C++ Parallel Programming. In *26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, PDP'18, pages 733–740, Cambridge, UK. IEEE.

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

Löff, J., Griebler, D., Mencagli, G., Araujo, G., Torquati, M., Danelutto, M., and Fernandes, L. G. (2021). The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757.

Sudwoj, M. (2020). Rust programming language in the high-performance computing environment. B.S. thesis, ETH Zurich.

Zomer, B., Hoffmann, R., and Griebler, D. (2023). Implementação e Avaliação de Desempenho da Linguagem Rust no NAS Embarassingly Parallel Benchmark. In *Anais da XXIII Escola Regional de Alto Desempenho da Região Sul*, pages 53–56, Porto Alegre, Brazil. Sociedade Brasileira de Computação.