Implementação e Avaliação de Desempenho da Linguagem Rust no NAS *Embarassingly Parallel Benchmark*

Bernardo B. Zomer [1], **Renato B. Hoffmann** [1], **Dalvan Griebler** [1]

[1] Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

(bernardo.barzotto,renato.hoffmann)@edu.pucrs.br, dalvan.griebler@pucrs.br

Resumo. *Rust é uma linguagem multiparadigmática de alto desempenho que garante segurança de memória.* NAS Parallel Benchmarks *engloba aplicações paralelas de computação de dinâmica de fluidos, possuindo versões em Fortran e C++. Neste trabalho, a aplicação EP foi convertida para Rust e paralelizada com as bibliotecas Rayon e Rust SSP. Na avaliação de desempenho, Rust demonstrou a melhor escalabilidade no paralelismo quando foi usado Rust SSP.*

1. Introdução

O paralelismo é necessário para aproveitar toda a capacidade do hardware moderno [Löff et al. 2021]. Todavia, utilizá-lo não é uma tarefa trivial, especialmente quando somado a outros desafios da computação como a segurança de memória. Dentro deste contexto, é possível usar a linguagem de programação Rust, que é multiparadigmática e tem foco em alto desempenho, ao mesmo tempo que garante segurança de memória e de concorrência em tempo de compilação.

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. Nos anos posteriores à sua criação, o NPB ganhou grande reconhecimento como indicador padrão do desempenho de supercomputadores [Bailey and et. al. 1995].

Rust já é usada no desenvolvimento de aplicações complexas a nível de produção, mas ainda não existem muitos estudos avaliando sua eficácia em aplicações numéricas de alta complexidade. Para abordar esta questão, este trabalho desenvolveu uma versão da aplicação *Embarassingly Parallel* (EP) do NPB para a linguagem, com uma implementação sequencial e três paralelas. Sendo assim, as contribuições científicas deste artigo incluem o desenvolvimento deste programa assim como a análise e comparação de desempenho com C++ e Fortran, incluindo versões paralelas com OpenMP e OneTBB.

Em seguida, o processo de implementação dos programas avaliados foi explicado na Seção 2. Posteriormente, os resultados obtidos foram exibidos na Seção 3; por fim, a Seção 4 comenta trabalhos relacionados e a Seção 5 apresenta as considerações finais.

2. Desenvolvimento

Dentre os *kernels* disponíveis no NPB, foi escolhido o EP. Esta aplicação resolve um problema típico de muitas simulações do modelo probabilístico Monte Carlo. A conversão do código sequencial para Rust foi facilitada pelas garantias da linguagem e não acarretou em complexidade extra. Uma única instância *unsafe* foi necessária já que o programa original não verificava os limites de um vetor.

A partir da versão sequencial, foram usadas as bibliotecas de paralelismo Rayon e Rust SSP. Destas duas, a primeira é muito popular no ecossistema e a segunda provém da academia e já demonstrou desempenho superior [Pieper et al. 2021]. A estratégia de paralelismo envolveu a divisão de trabalho igualmente entre os elementos paralelos e requer uma sincronização ao final de todas suas operações, o que constitui um típico *MapReduce* [Bailey and et. al. 1991]. A Figura 1 mostra um pseudocódigo das duas versões. Rayon exigiu poucas mudanças na estrutura do código com o uso de iteradores paralelos, enquanto Rust SSP requereu uma re-estruturação do código para compor a sintaxe de *pipeline* estruturado.

```
let res = (1..end).into_par_iter()
    .map(|k| k as i32 + K_OFFSET)
    .map(|mut kk| {
        /* Encontrar a semente inicial */
        /* Computar números pseudoaleatórios */
        /* Computar desvios gaussianos */
    })
    .reduce( /* Reduzir */ );
```

(a) Rayon

```
let pipeline = pipeline![
    parallel!( /* Encontrar a semente inicial */ ),
    parallel!( /* Computar números pseudoaleatórios */ ),
    parallel!( /* Computar desvios gaussianos */ ),
    collect!() ];
/* Iniciar o pipeline */
(1..end).into_iter()
    .map(|k| k as i32 + K_OFFSET)
    .for_each(|kk| {
        pipeline.post(kk).unwrap();
    });
let res = pipeline.collect().iter()
    .fold(  /* Reduzir */  );
```

(b) Rust SSP

Figura 1. Comparação de sintaxe entre Rayon e Rust SSP

É possível separar a aplicação EP em três estágios independentes de um *pipeline* e mais dois estágios sequenciais. Esta versão não foi desenvolvida com as outras interfaces de paralelismo porque não oferecem uma sintaxe de *pipeline* estruturada como Rust SSP. Posteriormente, uma versão com apenas um estágio paralelo colapsando todos os outros estágios foi desenvolvida a fins de comparação com as outras implementações.

3. Experimentos

Para fins de comparação com a versão desenvolvida em Rust neste trabalho, foram escolhidas implementações do EP em Fortran ([Bailey and et. al. 1991]), de forma sequencial e paralela com OpenMP, e C++ ([Löff et al. 2021]), de forma sequencial e paralela com OpenMP e OneTBB.

Os experimentos foram realizados em uma máquina com dois processadores Intel® Xeon® Silver 4210 a 2.20GHz (20 cores, totalizando 40 threads) e 188 GB de RAM. O código Rust foi compilado com a versão 1.61.0 do rustc, por meio do perfil *release* do Cargo, 1.6.1 do Rayon e 0.1.0 (commit b05a591) do Rust SSP. Para Fortran e C++, foram usados o gfortran e o 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 cinco execuções. Quando representados por barras de erro, os desvios-padrões são pouco visíveis por serem baixos.

As cargas do NPB são expressas por meio de classes de problema. Para este trabalho, os experimentos foram rodados sob as classes S, W, A, B e C. As classes A e C serão o foco para os resultados dos códigos paralelos, visto que as outras resultaram em tendências similares. A classe C é cerca de dezesseis vezes maior do que a A.

Tabela 1. Tempo de execução (s) do EP sequencial sob as classes S a C

Classe	C++	Rust	Fortran
S	2.06 ± 0.00	1.80 ± 0.00	1.78 ± 0.00
W	4.13 ± 0.01	3.60 ± 0.00	3.56 ± 0.00
A	33.09 ± 0.10	28.88 ± 0.14	28.47 ± 0.02
B	132.09 ± 0.03	115.81 ± 0.70	113.84 ± 0.03
C	528.31 ± 0.10	461.15 ± 0.14	455.41 ± 0.09

Os resultados comparando as versões sequenciais das linguagens foram expostos na Tabela 1. A implementação em C++ se mostrou a mais lenta em todos os casos, seguida daquela em Rust. Fortran teve tempo de execução médio 1,16 e 1,01 vezes menor do que C++ e Rust respectivamente. Dado que o código Rust foi baseado diretamente na versão C++, é possível que essa diferença de desempenho seja devido a questões da programabilidade da linguagem e ferramentas de compilação.

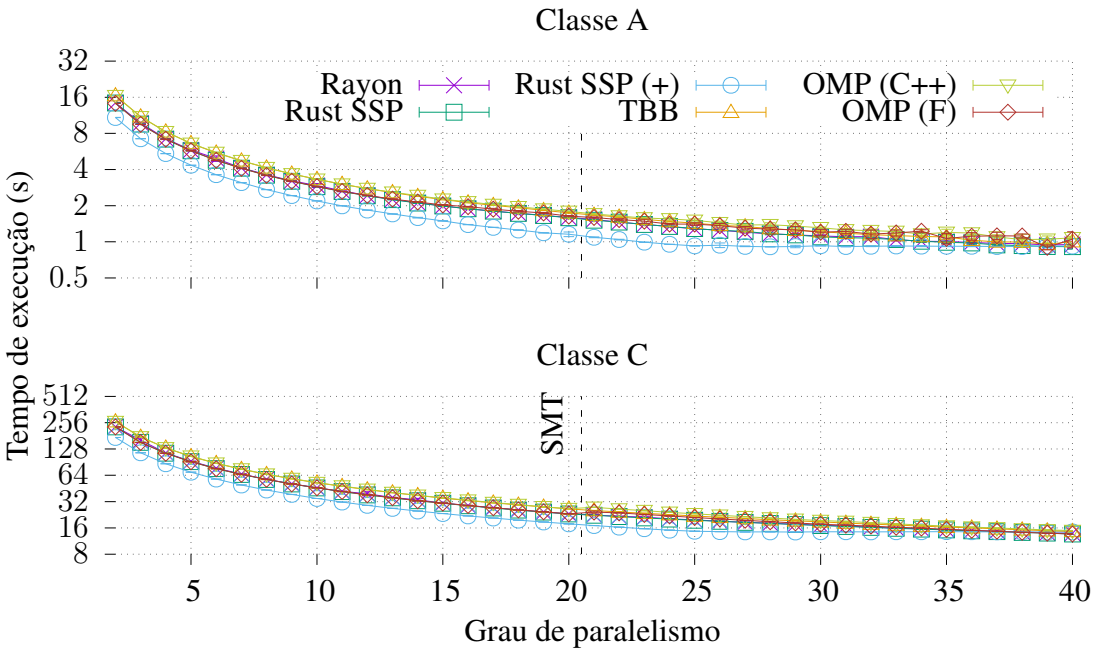

Figura 2. Tempo de execução do EP paralelo sob as classes A e C

Com paralelismo, o melhor desempenho médio foi exibido por Rust SSP quando usado mais de um estágio paralelo, legendado na Figura 2 como `Rust SSP (+)`. Na classe C, esta versão obteve um ganho médio em tempo de execução de 22,71% em relação à alternativa com um único estágio paralelo. Contudo, esta outra versão foi a que alcançou a melhor métrica nesta mesma classe, junto com Fortran, com a marca de 13,62 segundos no último grau de paralelismo; o melhor com `Rust SSP (+)` foi de 14,38 segundos.

Por fim, é importante considerar que o *runtime* utilizado em uma certa execução define o número de *threads* ativas a sua maneira. Ou seja, esta métrica não necessariamente condiz com o grau de paralelismo solicitado pelo usuário. Isto explica o desempenho superior do Rust SSP quando usando mais de um estágio paralelo.

4. Trabalhos Relacionados

No presente, poucas pesquisas acerca da eficácia da linguagem Rust no paralelismo foram realizadas. Em [Pieper et al. 2021], Rust SSP foi introduzido e testes com aplicações de paralelismo de *stream* foram conduzidos. Como neste trabalho, Rust SSP obteve desempenho melhor que Rayon; especificamente, em 30,3%. Em [Bychkov and et. al. 2021], foram efetuados testes de desempenho paralelo com bibliotecas matemáticas, onde também foi concluído que Rust é capaz de entregar resultados comparáveis a C++. Neste caso, o foco era em aplicações para supercomputadores.

Em [Costanzo and et. al. 2021], foi realizado um estudo comparativo entre Rust e C em termos de desempenho e esforço de programação, usando como base o problema dos n-corpos, típico na computação de alto desempenho. Neste trabalho, Rust também apresentou performance comparável a uma linguagem consolidada.

5. Conclusões

Neste trabalho, a aplicação EP do NPB foi convertida para Rust e paralelizada com duas bibliotecas diferentes. Posteriormente, seu desempenho foi avaliado com implementações sequenciais e paralelas das linguagens C++ e Fortran. Os resultados demonstraram que o desempenho da versão sequencial Rust foi superior à C++ e similar à versão Fortran; a alternativa paralela com melhor desempenho médio foi usando mais de um estágio paralelo com Rust SSP. No futuro, é possível converter mais aplicações numéricas do NPB para Rust com o intuito de avaliar outras categorias de problemas computacionais e analisar se os resultados obtidos aqui são consistentes entre diferentes métricas.

Referências

Bailey, D. and et. al. (1991). The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

Bailey, D. and et. al. (1995). The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center.

Bychkov, A. and et. al. (2021). Rust language for supercomputing applications. In *Supercomputing*, pages 391–403, Cham. Springer International Publishing.

Costanzo, M. and et. al. (2021). Performance vs programming effort between rust and c on multicore architectures: Case study in n-body. In *Proceedings - 2021 47th Latin American Computing Conference, CLEI 2021*.

Löff, J., Griebler, D., and et. al. (2021). The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757.

Pieper, R., Löff, J., Hoffmann, R. B., Griebler, D., and et. al. (2021). High-level and Efficient Structured Stream Parallelism for Rust on Multi-cores. *Journal of Computer Languages*.