UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Daniel Roberto de Aguiar

AVALIAÇÃO DO POTENCIAL DO USO DA LINGUAGEM DE PROGRAMAÇÃO RUST NA REIMPLEMENTAÇÃO DO MÓDULO DE TRIANGULAÇÃO DO MOHID

Florianópolis, Santa Catarina – Brasil
2025

Daniel Roberto de Aguiar

AVALIAÇÃO DO POTENCIAL DO USO DA LINGUAGEM DE PROGRAMAÇÃO RUST NA REIMPLEMENTAÇÃO DO MÓDULO DE TRIANGULAÇÃO DO MOHID

Trabalho de Conclusão de Curso submetido ao curso de Bacharelado em Sistemas de Informação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientador(a): Eduardo Camilo Inacio, Dr.

Florianópolis, Santa Catarina – Brasil
2025

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.
Arquivo compilado às 09:25h do dia 9 de dezembro de 2025.

Daniel Roberto de Aguiar
Avaliação do Potencial do Uso da Linguagem de Programação Rust na Reimplementação do Módulo de Triangulação do MOHID / Daniel Roberto de Aguiar; Orientador(a), Eduardo Camilo Inacio, Dr. – Florianópolis, Santa Catarina – Brasil, 27 de novembro de 2025.
63 p.

Trabalho de Conclusão de Curso – Universidade Federal de Santa Catarina, Departamento de Informática e Estatística (INE) do Centro Tecnológico (CTC), Bacharelado em Sistemas de Informação.

Inclui referências

1. Rust, 2. Fortran, 3. Benchmark, 4. Mohid, I. Eduardo Camilo Inacio, Dr. II. Bacharelado em Sistemas de Informação III. Avaliação do Potencial do Uso da Linguagem de Programação Rust na Reimplementação do Módulo de Triangulação do MOHID

CDU 02:141:005.7

Daniel Roberto de Aguiar

AVALIAÇÃO DO POTENCIAL DO USO DA LINGUAGEM DE PROGRAMAÇÃO RUST NA REIMPLEMENTAÇÃO DO MÓDULO DE TRIANGULAÇÃO DO MOHID

Este(a) Trabalho de Conclusão de Curso foi julgado adequado(a) para obtenção do Título de Bacharel em Sistemas de Informação, e foi aprovado em sua forma final pelo Bacharelado em Sistemas de Informação do Departamento de Informática e Estatística (INE) do Centro Tecnológico (CTC) da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 27 de novembro de 2025.

____

Álvaro Junio Pereira Franco, Dr.
Coordenador(a) do Bacharelado em Sistemas de Informação

Banca Examinadora:

____

Eduardo Camilo Inacio, Dr.
Orientador(a)
Universidade Federal de Santa Catarina – UFSC

____

Prof. Marcio Bastos Castro, Dr.
Universidade Federal de Santa Catarina – UFSC

____

Prof. Odorico Machado Mendizabal, Dr.
Universidade Federal de Santa Catarina – UFSC

RESUMO

O presente trabalho tem como objetivo avaliar o potencial da linguagem de programação Rust no desenvolvimento de aplicações científicas de alto desempenho, comparando-a com a linguagem Fortran, tradicionalmente utilizada nesse contexto. Para isso, foi analisado o módulo de triangulação do software de modelagem hidrodinâmica MOHID, originalmente implementado em Fortran. O algoritmo responsável pela construção da triangulação tridimensional foi isolado do código original e reimplementado em Rust em duas versões: uma sequencial e outra paralela, com suporte a múltiplas threads. As execuções experimentais foram realizadas em ambiente Ubuntu Linux, com o uso da ferramenta perf para obtenção de métricas detalhadas de desempenho, como tempo de execução, ciclos de CPU, *cache misses*, *branch misses* e *page faults*. Os resultados demonstraram que a linguagem Rust apresentou melhor desempenho geral em relação ao Fortran, com menor tempo de processamento e melhor aproveitamento de *cache*, embora tenha exibido um número maior de *page faults*. A versão paralela em Rust obteve ganhos de desempenho a partir de 2 threads, com aumento do *speedup* até aproximadamente seis threads, seguido de estabilização, e eficiência paralela decrescente conforme o número de threads aumentava. Conclui-se que a linguagem Rust apresenta elevado potencial para o desenvolvimento de aplicações científicas de alto desempenho, combinando eficiência, segurança e controle de memória, consolidando-se como uma alternativa moderna ao Fortran.

Palavras-chaves: Rust. Fortran. Benchmark. Mohid.

ABSTRACT

This work aims to evaluate the potential of the Rust programming language in the development of high-performance scientific applications, comparing it with the Fortran language, practices used in this context. To this end, the triangulation module of the hydrodynamic modeling software MOHID, originally implemented in Fortran, was developed. The algorithm responsible for constructing the three-dimensional triangulation was isolated from the original code and reimplemented in Rust in two versions: one sequential and the other parallel, with support for multiple threads. Experimental runs were performed in the Ubuntu Linux environment, using the perf tool to obtain performance details such as execution time, CPU cycles, cache misses, branch misses, and page failures. The results demonstrated that the Rust language showed better overall performance compared to Fortran, with shorter processing times and better cache utilization, although it presented a higher number of page failures. The parallel version in Rust achieved significant performance gains, with increased speed up to approximately six threads, followed by stabilization, and decreasing efficiency in parallel as the number of threads increased. It is concluded that the Rust language presents high potential for the development of high-performance scientific applications, combining efficiency, security, and memory control, establishing itself as a modern alternative to Fortran.

Keywords: Rust. Fortran. Benchmark. Mohid.

LISTA DE FIGURAS

Figura 1 – Tempo médio por etapa do algoritmo ConstructTriangulationXYZ . 25
Figura 2 – Tempo médio de execução das funções do algoritmo ConstructTriangulationXYZ em Rust sequencial (escalas independentes) . . . 26
Figura 3 – Tempo médio por etapa da função ConstructDirectedEdges . . . 27
Figura 4 – Tempo médio de execução do processamento de dados (escalas independentes) 33
Figura 5 – Distribuição de quantidade de ciclos por linguagem em implementação sequencial 33
Figura 6 – Distribuição de Referências de *Cache* por linguagem em implementação sequencial 34
Figura 7 – Distribuição de *Cache Misses* por linguagem em implementação sequencial 34
Figura 8 – Distribuição da Quantidade de *Branches* por linguagem em implementação sequencial 35
Figura 9 – Distribuição de *Branch Misses* por linguagem em implementação sequencial 35
Figura 10 – Distribuição de *Page Faults* por linguagem em implementação sequencial 36
Figura 11 – Distribuição do tempo de execução por implementação sequencial em Rust 37
Figura 12 – Distribuição de Quantidade de ciclos por implementação sequencial em Rust 38
Figura 13 – Distribuição de *Cache Misses* por implementação 38
Figura 14 – Distribuição de Referências de *Cache* por implementação 39
Figura 15 – Distribuição de *Branch Misses* por implementação 40
Figura 16 – Distribuição da Quantidade de *Branches* por implementação . . . 40
Figura 17 – Distribuição de faltas de página por implementação 41
Figura 18 – Distribuição do tempo de execução por número de *threads* 42
Figura 19 – Tempo médio de execução das funções por número de *threads* . 42
Figura 20 – Distribuição de Quantidade de ciclos por número de *threads* . . . 43
Figura 21 – Distribuição de *Cache Misses* por número de *threads* 44
Figura 22 – Distribuição de Referências de *Cache* por número de *threads* . . 44
Figura 23 – Distribuição de *Branch Misses* por número de *threads* 45
Figura 24 – Distribuição da Quantidade de *Branches* por número de *threads* . 46
Figura 25 – Distribuição de *Page Faults* por número de *threads* 46
Figura 26 – *Speedup* em função do número de *threads* 47
Figura 27 – Eficiência paralela em função do número de *threads* 48

LISTA DE TABELAS

Tabela 1 – Tipos numéricos do Rust 9
Tabela 2 – Comparação de abstrações de programação paralela em Rust . . 17
Tabela 3 – Configuração do computador 30

SUMÁRIO

1 INTRODUÇÃO . . . 6
1.1 OBJETIVOS . . . 8
1.1.1 Objetivo Geral . . . 8
1.1.2 Objetivos Específicos . . . 8
2 FUNDAMENTAÇÃO TEÓRICA . . . 9
2.1 LINGUAGEM DE PROGRAMAÇÃO RUST . . . 9
2.2 LINGUAGEM DE PROGRAMAÇÃO FORTRAN . . . 11
2.2.1 gfortran . . . 12
2.3 *BENCHMARK* . . . 13
2.4 COMPUTAÇÃO PARALELA . . . 15
2.4.1 Programação Paralela em Rust . . . 16
2.5 MOHID . . . 17
3 TRABALHOS CORRELATOS . . . 19
4 METODOLOGIA . . . 23
4.1 ANÁLISE DO CÓDIGO-FONTE E SELEÇÃO DA SUBROTINA . . . 23
4.2 ESTRUTURA E FUNCIONAMENTO DA SUBROTINA ORIGINAL . . . 24
4.3 REIMPLEMENTAÇÃO EM RUST . . . 25
4.3.1 Implementação Paralela em Rust . . . 28
4.4 VARIAÇÕES DOS EXPERIMENTOS . . . 28
4.5 CONFIGURAÇÃO DOS EXPERIMENTOS . . . 29
4.6 COLETA E TRATAMENTO DOS DADOS . . . 30
5 RESULTADOS . . . 32
5.1 COMPARAÇÃO DAS IMPLEMETAÇÕES SEQUENCIAIS . . . 32
5.2 COMPARAÇÃO ENTRE A IMPLEMENTAÇÃO SEQUENCIAL EM RUST E A VERSÃO PARALELA EXECUTADA EM 1 THREAD . . . 36
5.3 COMPARAÇÃO DA IMPLEMENTAÇÃO PARALELA EM RUST POR NÚMERO DE THREADS . . . 41
6 CONCLUSÕES . . . 49
REFERÊNCIAS . . . 51
APÊNDICE A – ARTIGO . . . 56

1 INTRODUÇÃO

O avanço contínuo das tecnologias digitais tem transformado profundamente a forma como dados são gerados, processados e analisados. Essa evolução impõe aos desenvolvedores o desafio constante de acompanhar novas linguagens, paradigmas e ferramentas, de modo a atender às crescentes demandas por desempenho, segurança e escalabilidade nas aplicações modernas. No contexto das ciências exatas e da modelagem computacional, essa necessidade torna-se ainda mais evidente, especialmente diante do volume e da complexidade dos dados envolvidos em simulações e análises espaciais (JACOBY *et al.*, 2019). O processamento de dados geográficos, em particular, caracteriza-se por seu alto custo computacional, exigindo algoritmos otimizados e linguagens capazes de explorar eficientemente os recursos de hardware disponíveis, como múltiplos núcleos de processamento e hierarquias de memória (JACOBY *et al.*, 2019).

Desde a década de 1950, a linguagem Fortran (*FORmula TRANslator*) consolidou-se como uma das principais ferramentas no desenvolvimento de modelos numéricos voltados à simulação e análise de fenômenos físicos, sendo amplamente empregada em áreas como meteorologia, oceanografia, hidrodinâmica e modelagem ambiental (IBM, 2025a). Sua ênfase em cálculos numéricos e manipulação eficiente de *arrays* fez dela a linguagem dominante em aplicações científicas e de engenharia por várias décadas, contribuindo para a criação de sistemas complexos como o *Regional Atmospheric Modeling System* (RAMS) (RAMS DEVELOPMENT TEAM, 2025) *Weather Research and Forecasting Model* (WRF) (WRF DEVELOPMENT TEAM, 2025) e o MOHID *Water Modelling System* (MOHID WATER MODELLING SYSTEM TEAM, 2025a). No entanto, apesar de sua maturidade e desempenho comprovado, o Fortran apresenta limitações frente aos paradigmas de programação modernos, como a orientação a objetos, o gerenciamento seguro de memória e o suporte nativo à concorrência (GUEZ, 2025). Além disso, a vasta base de código legado acumulada ao longo de décadas, muitas vezes escrita em versões antigas da linguagem, torna a atualização, refatoração e integração com tecnologias contemporâneas um processo complexo e dispendioso. Essas características tornam especialmente desafiadora a modernização de grandes sistemas científicos consolidados, entre eles o MOHID.

O MOHID *Water Modelling System* é um sistema de modelagem hidrodinâmica e ambiental iniciado na década de 1980 no Instituto Superior Técnico (IST) e atualmente desenvolvido pelo grupo MARETEC (MOHID WATER MODELLING SYSTEM TEAM, 2025b). Projetado com arquitetura modular, o software integra modelos acopláveis para hidrodinâmica tridimensional, transporte de sedimentos, qualidade da água, processos biogeoquímicos, ondas e interação rio-oceano (MOHID WATER MODELLING SYSTEM TEAM, 2025e). Seu núcleo computacional, escrito predominantemente em

Fortran, segue um conjunto de padrões estruturais e decisões históricas que, embora tenham garantido robustez e desempenho por décadas, hoje impõem dificuldades para manutenção, integração com novas tecnologias e exploração de paradigmas modernos de paralelismo e segurança de memória.

As linguagens de programação evoluem continuamente, e o cenário atual mostra que a adoção de tecnologias emergentes tornou-se um fator crítico para a competitividade e segurança dos softwares. Nesse contexto, Rust vem ganhando destaque significativo: segundo a reportagem de Taft (2024), Rust é a linguagem que mais cresce, corroborando com a pesquisa da StackOverflow (2024), que aponta Rust como a linguagem mais admirada pela comunidade de desenvolvedores, com cerca de 83% de aprovação. Esse reconhecimento reflete não apenas a robustez técnica da linguagem, mas também a confiança de uma base crescente de desenvolvedores que veem em Rust uma alternativa moderna às linguagens tradicionais (TAFT, 2024). Adicionalmente, em fevereiro de 2024, o relatório técnico do Office of the National Cyber Director recomendou a adoção de linguagens com segurança de memória para reduzir vulnerabilidades, citando expressamente Rust como uma das escolhas estratégicas para sistemas críticos (ONCD, 2024). Esse conjunto de fatores tem impulsionado gigantes da tecnologia como Meta (FOSTER, 2025), Microsoft (COURIOL, 2025) e Google (CLABURN, 2024), a migrarem bases de código legadas, frequentemente escritas em C ou C++, para Rust, motivadas por ganhos em segurança, desempenho e manutenção, o que fortalece a relevância de estudos que investiguem empiricamente o uso de Rust em domínios de alto desempenho.

Apesar do crescente interesse e adoção da linguagem Rust por parte da indústria de software, ainda são limitados os estudos acadêmicos que avaliam empiricamente seu desempenho em comparação com linguagens consolidadas no domínio científico, como Fortran e C++. A maior parte das análises disponíveis concentra-se em aspectos qualitativos, como segurança de memória ou legibilidade do código, com aspectos quantitativos limitando-se ao tempo de execução (IONITA; MANOLE; SLUSANSCHI, 2020; COSTANZO *et al.*, 2021; MARTINS *et al.*, 2025), sem considerar métricas de hardware mais detalhadas, como utilização de cache, ocorrência de page faults e comportamento de branches. Essa limitação é particularmente relevante considerando que aplicações científicas de alto desempenho dependem fortemente da eficiência no uso dos recursos computacionais. Assim, torna-se essencial a realização de estudos comparativos que examinem, sob condições controladas, o comportamento de Rust frente a linguagens tradicionais amplamente utilizadas nesse contexto, contribuindo para uma compreensão mais abrangente sobre seu potencial em ambientes científicos e de engenharia.

1.1 OBJETIVOS

A partir da contextualização do problema, foram definidos os objetivos do presente trabalho.

1.1.1 Objetivo Geral

Avaliar o potencial da linguagem de programação Rust para desenvolvimento de aplicações científicas de alto desempenho.

1.1.2 Objetivos Específicos

- Analisar algoritmos sequenciais, porém paralelizáveis, utilizados no software MOHID;
- Desenvolver algoritmos sequenciais, porém paralelizáveis, utilizados no software MOHID na linguagem de programação Rust;
- Desenvolver versões paralelas de algoritmos sequenciais utilizados no software MOHID na linguagem de programação Rust;
- Realizar benchmark entre os algoritmos originais do software e suas versões geradas utilizando como métricas: tempo de execução, uso de memória e uso de CPU.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 LINGUAGEM DE PROGRAMAÇÃO RUST

Rust é uma linguagem de programação mantida pela Mozilla Research, que combina segurança de memória, alto desempenho, simultaneidade confiável e recursos de alto nível voltados à produtividade (PIEPER *et al.*, 2021). Segundo Pieper *et al.* (2021), sua principal característica é o sistema de posse e movimento, que permite o gerenciamento seguro da memória sem recorrer a um coletor de lixo. Esse mecanismo é semelhante ao padrão RAII (*Resource Acquisition Is Initialization*), utilizado por desenvolvedores C++, no qual os recursos são desalocados automaticamente ao final de seu escopo.

De acordo com Blandy, Orendorff e Tindall (2023), a linguagem é projetada em torno de seu sistema de tipos, o que possibilita a criação de código de alto desempenho ao permitir que o desenvolvedor escolha representações de dados adequadas a cada situação, equilibrando simplicidade e custo. Ainda segundo os autores, a robustez desse sistema de tipos e a flexibilidade conferida pelo uso de *traits* e tipos genéricos são fundamentais para as garantias de segurança de memória e de concorrência fornecidas pela linguagem. A Tabela 1 apresenta os tipos numéricos disponíveis em Rust, onde o termo "palavra de máquina" refere-se ao tamanho do endereço na arquitetura em que o código é executado, geralmente 32 ou 64 bits (BLANDY; ORENDORFF; TINDALL, 2023).

Tabela 1 – Tipos numéricos do Rust.

Tamanho (bits)	Número inteiro sem sinal	Número inteiro com sinal	Ponto flutuante
8	u8	i8	
16	u16	i16	
32	u32	i32	f32
64	u64	i64	f64
128	u128	i128	
Palavra de máquina	usize	isize	

Fonte: (BLANDY; ORENDORFF; TINDALL, 2023)

Segundo Blandy, Orendorff e Tindall (2023), há dois principais paradigmas de gerenciamento de memória: o modelo "*Safety First*", que utiliza coletores de lixo para liberar automaticamente objetos inacessíveis, como ocorre em Java, e o modelo "*Control First*", no qual o programador é responsável por alocar e desalocar a memória manualmente, sendo C e C++ os principais representantes desse paradigma. O problema desse último modelo é que o uso incorreto de ponteiros é uma das principais causas de vulnerabilidades de segurança relatadas historicamente. Rust busca conciliar segurança e desempenho, restringindo o uso de ponteiros de modo a permitir que o compilador verifique, em tempo de compilação, se o programa está livre de erros

de memória. Essas mesmas regras também fundamentam o suporte da linguagem à programação concorrente segura (BLANDY; ORENDORFF; TINDALL, 2023).

Uma dessas regras é o conceito de posse (*ownership*), incorporado diretamente à linguagem e reforçado por verificações estáticas. Cada valor em Rust possui um único proprietário, responsável por determinar seu tempo de vida. Quando o proprietário é liberado, o valor associado também é desalocado. Essa abordagem facilita a análise de tempo de vida e oferece ao desenvolvedor o controle explícito sobre os recursos, algo essencial em linguagens de sistemas. Outro conceito é o de movimento (*move semantics*) onde, ao atribuir um valor a uma variável, passá-lo como argumento ou retorná-lo de uma função, o valor não é copiado, a posse é transferida para o destino, e a origem torna-se inválida. Isso torna a atribuição eficiente, eliminando a necessidade de contagem de referências ou coletor de lixo (BLANDY; ORENDORFF; TINDALL, 2023).

Conforme destacam Pieper *et al.* (2021), há historicamente uma divisão entre linguagens voltadas à segurança, que utilizam sistemas de tipos rigorosos para evitar erros em tempo de compilação, como Java, e linguagens voltadas ao controle e ao desempenho, como C e C++. Rust foi concebida para reduzir essa distância, oferecendo um sistema de tipos robusto e garantias de segurança de memória, sem abrir mão do desempenho típico de linguagens de baixo nível.

Esses ganhos são proporcionados principalmente pelo rustc, o compilador oficial da linguagem, implementado em Rust e baseado no backend LLVM. Similar a compiladores front-end clássicos como o Clang, o rustc traduz o código-fonte para a Representação Intermediária LLVM (LLVM IR), mas introduz etapas e representações intermediárias adicionais para lidar com os mecanismos exclusivos da linguagem (LIU *et al.*, 2025). O processo de compilação ocorre em múltiplos estágios, iniciando pela análise sintática e semântica, na qual a Árvore de Sintaxe Abstrata (AST) é convertida para a Representação Intermediária de Alto Nível (HIR), que abstrai detalhes sintáticos e facilita a inferência e a verificação de tipos, além da resolução de traits (RUST PROJECT DEVELOPERS, 2025). Essa fase é especialmente complexa devido à flexibilidade do sistema de traits e à presença de tempos de vida (*lifetimes*) em tipos e referências (LIU *et al.*, 2025).

Em seguida, a HIR é convertida para a Representação Intermediária de Nível Médio (MIR), uma forma de código orientada ao fluxo de controle, essencial para garantir as regras de posse, verificação de empréstimos (*borrow checking*) e semântica de movimento (RUST PROJECT DEVELOPERS, 2025). Antes de gerar a LLVM IR, o rustc aplica uma série de otimizações específicas de Rust sobre a MIR, visando maximizar a eficiência do código gerado sem comprometer as garantias de segurança de memória e de concorrência (LIU *et al.*, 2025).

2.2 LINGUAGEM DE PROGRAMAÇÃO FORTRAN

A linguagem Fortran (do inglês *FORmula TRANslator*) foi desenvolvida originalmente pela IBM, sob liderança de John W. Backus, tendo sido concebida por volta de 1954 para o computador IBM 704 e lançada comercialmente em 1957 (ENCYCLOPAEDIA BRITANNICA, 2025). O principal objetivo era permitir que cientistas e engenheiros pudessem expressar fórmulas matemáticas e cálculos numéricos diretamente em alto nível, em lugar de lidarem com linguagem de máquina ou de montagem (*assembly*), de modo a aumentar a produtividade e reduzir custos com programação de baixo nível (IBM, 2025a).

Fortran pode ser considerada a primeira linguagem de programação de alto nível realmente bem-sucedida e padronizada para uso científico e de engenharia (LINDEMANN; DAHLBLOM, 2025). Uma das razões para seu êxito foi que, desde o início, a equipe de Backus preocupou-se com a geração de código eficiente: o compilador de Fortran produzia código comparável ao de mão (assembly) em muitos casos, o que garantiu aceitação na comunidade científica tradicionalmente exigente com o desempenho (IBM, 2025a).

Ao se tornar padrão (pela ANSI em 1966 e posteriormente pela ISO) e portável entre diferentes máquinas, Fortran permitiu a migração de códigos intensivos em cálculo para diferentes arquiteturas, o que foi um fator decisivo para sua adoção em centros de computação de alto desempenho (HPC) (ENCYCLOPAEDIA BRITANNICA, 2025).

Historicamente, Fortran dominou áreas como engenharia, modelagem numérica, dinâmica de fluidos, previsão meteorológica e simulações físicas por várias décadas, até que linguagens como C e C++ começaram a dividir esse espaço (MUSEU INFORMÀTICA, 2025). Entretanto, a linguagem continuou evoluindo, passando por sucessivas revisões que introduziram novos paradigmas e recursos, sem comprometer a compatibilidade com códigos legados, fator essencial em ambientes científicos, onde décadas de desenvolvimento acumulado permanecem em uso (GUEZ, 2025).

Alguns marcos importantes incluem:

- O lançamento original em 1957 (Fortran I) para IBM 704 (IBM, 2025a).
- Padronização ANSI X3.9-1966 (Fortran 66), posteriormente Fortran 77 em 1978 (GUEZ, 2025).
- A revisão Fortran 90 (1991) que introduziu módulos, programação estruturada, alocação dinâmica, recursão, e outras facilidades modernas.
- Versões subsequentes: Fortran 95, 2003, 2008, 2018 (e atualmente Fortran 2023) com recursos como tabelas hash, programação orientada a objetos, coarrays para paralelismo, interoperabilidade com C, entre outros (GUEZ, 2025).

Fortran possui características que a tornaram especialmente adequada para aplicações científicas de alto desempenho como sua forte ênfase em operações numéricas e vetoriais, com suporte eficiente a arrays e matrizes (FORTRAN-LANG COMMUNITY, 2025), compiladores altamente otimizados que permitem geração de código com desempenho próximo ao *assembly* (IBM, 2025a), compatibilidade com códigos legados, permitindo a manutenção e evolução de grandes sistemas científicos escritos originalmente em Fortran (MUSEU INFORMÀTICA, 2025) e suporte a paralelismo e arquiteturas modernas por meio de recursos como coarrays, teams, e collective subroutines que permitem explorar sistemas multi-núcleo, distribuídos ou em nuvem (FORTRAN-LANG COMMUNITY, 2025)

2.2.1 gfortran

O *GNU Fortran* (gfortran) faz parte da *GNU Compiler Collection* (GCC) e é o sucessor do compilador g77, apresentando uma implementação completamente nova. O gfortran implementa integralmente os padrões Fortran 77, 90 e 95, a maior parte dos padrões 2003 e 2008, e diversos recursos do padrão 2018, além de oferecer extensões adicionais, como suporte às interfaces de paralelismo OpenMP e OpenACC (THE GFORTRAN TEAM, 2025). Trata-se de uma das implementações livremente disponíveis mais difundidas da linguagem Fortran, amplamente adotada em ambientes acadêmicos e de código aberto (THE GFORTRAN TEAM, 2025).

A GCC é uma coleção de *front-ends* para diferentes linguagens de programação, responsáveis por traduzir o código-fonte para uma representação intermediária independente chamada *GENERIC*. Essa representação é então processada por um *middle-end* comum, responsável por aplicar otimizações, e posteriormente enviada a um dos *back-ends*, que geram o código de máquina otimizado para diversas arquiteturas e sistemas operacionais (THE GFORTRAN TEAM, 2025).

No caso da linguagem Fortran, o gfortran reconhece arquivos com extensões como .f, .for, .ftn, .f90, .f95, .f03 e .f08. Cada arquivo é processado pelo front-end do compilador (f951), que realiza a análise sintática e semântica do código, convertendo-o para a representação intermediária. Em seguida, o montador (*assembler*) e o ligador (*linker*) são invocados para gerar o executável final (THE GFORTRAN TEAM, 2025).

O comando gfortran herda todas as opções do GCC, possibilitando o uso de uma ampla variedade de *flags* que permitem personalizar o processo de compilação, adequando-o a diferentes objetivos, como otimização de desempenho, depuração (*debug*), análise de desempenho, ou interoperabilidade entre linguagens (THE GFORTRAN TEAM, 2025).

2.3 *BENCHMARK*

Os *benchmarks* são avaliações comparativas utilizadas para medir o desempenho de sistemas, componentes ou algoritmos, caracterizando-se pelo uso de uma metodologia estruturada que aplica cargas de trabalho definidas ou representativas a um sistema, mede variáveis de desempenho e permite inferir a eficiência, escalabilidade ou custo-benefício da implementação ou arquitetura em estudo. A comparação é feita com valores de referência, sendo essa uma das principais estratégias utilizadas no campo da Computação Evolutiva (EC) (BARTZ-BEIELSTEIN *et al.*, 2020).

O objetivo central dos *benchmarks* é obter valores quantitativos que permitam comparar alternativas, como diferentes linguagens, algoritmos ou configurações, e avaliar se um sistema atende aos requisitos de desempenho estabelecidos. No contexto da HPC, a medição adequada de desempenho é fundamental para a tomada de decisões de projeto, alocação de recursos, escolha de linguagens e otimizações de execução (MICROSOFT IGNITE, 2025).

Segundo Bartz-Beielstein *et al.* (2020), três aspectos principais devem ser considerados em todo estudo de *benchmark*:

- as medidas de desempenho;
- o problema; e
- o algoritmo.

Ainda, conforme Bartz-Beielstein *et al.* (2020), estudos realizados em meados de 1994 indicavam que o estudo empírico de algoritmos era um campo relativamente imaturo, e pouca evolução foi observada desde então. Entre as razões apontadas para essa situação insatisfatória no domínio da EC está a ausência de consenso sobre uma metodologia geral para a realização de *benchmarks*, diferentemente de áreas como o Projeto Estatístico de Experimentos ou a Mineração de Dados, que já dispõem de metodologias consolidadas. Essa carência decorre, em parte, da alta complexidade envolvida na elaboração de um estudo de *benchmark* robusto, especialmente devido às considerações estatísticas exigidas. Outra causa relevante é a falta de diretrizes práticas, o que faz com que cada pesquisador estabeleça suas próprias metodologias para casos específicos (BARTZ-BEIELSTEIN *et al.*, 2020).

As motivações para a realização de estudos comparativos entre algoritmos de otimização são diversas. Além dos objetivos científicos, *benchmarks* também podem servir como meio de popularização de abordagens algorítmicas ou de problemas específicos. Bartz-Beielstein *et al.* (2020) resumem os principais objetivos científicos desses estudos em cinco categorias:

- Visualização e avaliação básica de algoritmos e problemas;

- Análise de sensibilidade no projeto de algoritmos e nas características do problema;
- Treinamento e extrapolação de desempenho;
- Objetivos orientados à teoria; e
- *Benchmarking* aplicado ao desenvolvimento de algoritmos.

Em geral, esses objetivos buscam aperfeiçoar a implementação prática dos algoritmos, a partir de uma melhor compreensão da interação entre escolhas de projeto algorítmico e as características das instâncias dos problemas. Além disso, o *benchmarking* atua como um elo entre a comunidade científica e os usuários de heurísticas de otimização, e entre as correntes teóricas e empíricas dentro da própria comunidade científica (BARTZ-BEIELSTEIN *et al.*, 2020).

A importância dos *benchmarks* decorre de diversos fatores. Sem dados quantitativos de desempenho, qualquer alegação de "melhor desempenho" ou "maior rapidez" torna-se pouco confiável, especialmente em contextos científicos, nos quais escalabilidade e consumo de recursos podem variar significativamente. Os *benchmarks* também permitem identificar gargalos e limitações reais de sistemas ou implementações, como tempo de CPU ocioso, latências de memória ou contenção de cache.

Métricas precisas possibilitam avaliar melhorias e demonstrar ganhos de desempenho de forma empiricamente fundamentada. Além disso, em estudos que buscam reprodutibilidade científica, o uso de protocolos de *benchmark* padronizados assegura credibilidade, transparência e comparabilidade entre resultados.

As principais métricas de desempenho utilizadas em estudos de *benchmark* incluem:

- Tempo de execução: intervalo real entre o início e o término de uma tarefa.
- *Throughput* / taxa de operações: número de operações realizadas por unidade de tempo.
- Utilização de recursos: instruções por ciclo (IPC), ciclos de CPU por tarefa, ou percentual de uso de núcleos/*threads*.
- Latência e largura de banda de memória: tempo de acesso e taxa de transferência entre diferentes níveis de memória (cache, RAM, armazenamento).
- Taxas de falhas de cache, *branch misses* e *stalls*: eventos que penalizam o *pipeline* do processador e reduzem o desempenho.
- Escalabilidade: comportamento do desempenho ao variar o número de *threads*, tamanho dos dados ou número de nós de processamento, métrica essencial

em aplicações paralelas ou distribuídas, frequentemente medida em termos de *speed-up* ou eficiência.

A ferramenta perf é um componente fundamental do ecossistema Linux para a medição precisa de eventos de hardware e software, sendo amplamente utilizada em *benchmarks* de desempenho. O utilitário (também conhecido como perf_events) permite acessar contadores de desempenho do processador, *tracepoints* do *kernel*, sondas dinâmicas (kprobes/uprobes) e outras métricas de baixo nível, indispensáveis para avaliar o comportamento interno de algoritmos e sistemas (PERFWIKI, 2025).

2.4 COMPUTAÇÃO PARALELA

A computação paralela pode ser conceituada como o uso simultâneo de múltiplos recursos de processamento, como *núcleos* de CPU, várias CPUs, nós em cluster ou ainda aceleradores (GPUs), para executar diferentes partes de uma tarefa ou diferentes tarefas ao mesmo tempo, com o objetivo de aumentar o desempenho (reduzir tempo de execução) e/ou permitir a solução de problemas de maior escala (IBM, 2025b).

Em termos mais diretos, em vez de executar uma sequência de operações de forma estritamente serial (uma após a outra), a computação paralela procura "dividir" o problema em subproblemas ou em fluxos de execução que podem progredir simultaneamente, de modo a acelerar a resolução global ou permitir a resolução de instâncias maiores do problema.

Segundo Jacoby *et al.* (2019), a utilização de ferramentas que acelerem o processamento de dados se torna essencial para lidar com dados geoespaciais, tendo em vista a grande quantidade gerada em todo o mundo, sejam imagens de satélite ou dados alfanuméricos oriundos de diferentes sensores, o que eleva vertiginosamente o custo computacional e o tempo de processamento ao se trabalhar afim de gerar informações sobre um determinado tema.

Porém, Pieper *et al.* (2021) destaca que a computação paralela não é uma atividade trivial, tendo em vista que programadores devem lidar com conceitos complexos e propensos a erros, como criação e gerenciamento de *threads*, sincronização, mecanismos de comunicação, balanceamento de carga, dependências de dados, acesso a dados críticos ou mutuamente exclusivos, etc. Tais fatos se tornam ainda mais relevantes quando se considera que o principal paradigma de programação continua sendo o sequencial. Ainda segundo os autores, o comportamento não determinístico de programas paralelos os torna mais difíceis de raciocinar e depurar se comparados a programas sequenciais, havendo ainda a questão de que desenvolvedores de aplicativos não se especializam em técnicas de programação paralela, geralmente buscando soluções algorítmicas para resolução de problemas. Outro aspecto fundamental é a existência de um limite teórico de ganho de desempenho em aplicações paralelas,

definido pela Lei de Amdahl, a qual estabelece que o acréscimo de desempenho de um programa paralelo é limitado pela fração de processamento serial em relação ao tempo total de execução do programa em um único processador, dessa forma, é possível derivar um limite superior de aceleração (*speedup*) conforme o número de processadores aumenta (SHI, 1996).

Blandy, Orendorff e Tindall (2023) elencam que as principais abordagens de paralelismo que desenvolvedores de sistemas geralmente utilizam são:

- *Thread* de *background*: uma única thread trabalhadora, que periodicamente executa instruções;
- *Pools* de trabalhadores(*workers*): conjunto de threads de propósito geral que se comunicam com os clientes por meio de filas de tarefas;
- *Pipelines*: os dados fluem de uma *thread* para a próxima, com cada thread fazendo um pouco do trabalho;
- Paralelismo de dados: nesta abordagem supõe-se que todo o computador fará principalmente uma única grande computação, sendo esta dividida em n partes e executada por n *threads* na esperança de colocar todos os n núcleos da máquina para funcionar de uma só vez;
- Um mar de objetos sincronizados: várias *threads* têm acesso aos mesmos dados e os problemas de concorrência são evitados utilizando esquemas de bloqueio *ad hoc* baseados em primitivas de baixo nível como mutexes;
- Operações inteiras atômicas: vários núcleos se comunicam passando instruções por meio de campos do tamanho de uma palavra de máquina.

2.4.1 Programação Paralela em Rust

A linguagem de programação Rust descata-se por permitir a elaboração de programas concorrentes rápidos e seguros, possibilitando a utilização de diferentes abordagens visando obter-se a melhor solução para cada problema computacional, enquanto programas escritos em outras linguagens que utilizem bem *threads* estão cheios de regras tácitas. Em Rust tais regras são especificadas no código e aplicadas pelo compilador (BLANDY; ORENDORFF; TINDALL, 2023).

Porém, Pieper *et al.* (2021) apontam de que a biblioteca padrão de Rust não fornece nenhuma abstração de programação paralela, trazendo apenas recursos básicos de baixo nível como *threads* de sistema operacional e bloqueio de chamadas de sistema, no entanto, ainda segundo os autores, a comunidade segue muito ativa e desenvolveu novas implementações e abstrações paralelas eficientes, sendo as principais Rayon e Tokio, ambas baseando-se em uma abordagem de paralelismo *ad-hoc*. A Tabela 2

Tabela 2 – Comparação de abstrações de programação paralela em Rust.

Suporte à implementação	**Tokio**	**Rayon**
Paralelismo de fluxograma	Sim	Possível
Paralelismo de Dados	Possível	Sim
Paralelismo de tarefas	Sim	Não
Paralelismo de fluxo de dados	Possível	Não
Paralelismo estruturado	Não	Possível
Estágios com estados	Sim	Não

Fonte: Adaptado de (PIEPER *et al.*, 2021)

apresenta uma comparação entre as bibliotecas.

Tokio destaca-se como uma biblioteca para programação assíncrona, sendo inteiramente baseada em *futures*, que representam unidades abstratas de trabalho que podem ser concluídas no futuro (PIEPER *et al.*, 2021).

A biblioteca Rayon destaca-se como uma solução voltada ao paralelismo de dados em Rust, baseada no uso de iteradores paralelos que estendem os iteradores sequenciais da biblioteca padrão. Suas funções internas seguem o paradigma de padrões paralelos, como map e reduce, o que a torna uma das ferramentas mais próximas do conceito de paralelismo estruturado na linguagem (PIEPER *et al.*, 2021). Para garantir alto desempenho, o Rayon implementa uma estratégia de balanceamento dinâmico de carga por meio do algoritmo de roubo de trabalho (*work-stealing*), no qual as *threads* gerenciam tarefas de forma cooperativa, com cada *thread* recebendo uma tarefa iterativa e a dividindo em duas partes, uma é executada imediatamente, enquanto a outra é adiada e colocada em uma fila de tarefas, deste modo *threads* ociosas, por vezes, procuram continuamente por tarefas disponíveis nessas filas, podendo "roubar" parte do trabalho de outras *threads*, o que resulta em melhor aproveitamento dos recursos de processamento e redução de ociosidade (MARTINS *et al.*, 2025).

2.5 MOHID

O MOHID Water Modelling System (MOHID) é um sistema integrado de modelagem de águas desenvolvido no centro de tecnologia MARETEC do Instituto Superior Técnico, Universidade de Lisboa (Portugal) (MOHID WATER MODELLING SYSTEM TEAM, 2025b). A origem do modelo remonta meados da década de 1980, quando iniciou como um modelo bidimensional de marés escrito em Fortran 77, com o nome derivado da abreviação portuguesa "MOdelo HIDrodinâmico" (MOHID WATER MODELLING SYSTEM TEAM, 2025c). Ao longo das décadas seguintes, o projeto evoluiu para uma plataforma modular que inclui os módulos "Water" (dinâmica de águas superficiais e costeiras), "Land" (hidrologia de bacias) e "Soil" (escoamento em aquíferos), permitindo simular processos físicos, químicos e ecológicos em ambientes aquáticos e terrestres interligados (CHAMBEL-LEITÃO *et al.*, 2007).

Em termos de funcionalidades, o MOHID permite a discretização em três dimensões de volumes de água, acoplamento entre colunas de água, sedimentos e rios, transporte de sedimentos e de massa dissolvida, modelagem de qualidade da água, e interação entre domínio hidrodinâmico, hidrológico e de solo. O manual do módulo hidrodinâmico apresenta-se como uma obra técnica de referência, detalhando discretização horizontal e vertical, condições de contorno, fluxos e forças, o que reforça a robustez e maturidade do sistema (MOHID WATER MODELLING SYSTEM TEAM, 2025e). Sua ampla base de usuários está evidenciada por mais de 2.000 utilizadores registrados mundialmente (MOHID WATER MODELLING SYSTEM TEAM, 2025d), participando de ambientes de pesquisa e operacionais, o que atesta o caráter científico e aplicado do sistema.

Quanto à implementação, embora a versão original tenha sido escrita em Fortran 77, tornou-se evidente ao longo do tempo que limitações técnicas dessa versão, especialmente no que se refere à modularização, manutenção e reutilização de código, exigiram uma reestruturação. O sistema foi então migrado para ANSI Fortran 95 (ou Fortran 90/95), aproveitando as funcionalidades introduzidas, como tipos derivados e módulos, e possibilitando uma abordagem mais modular e escalável (CHAMBEL-LEITÃO *et al.*, 2007). Ademais, o MOHID emprega uma filosofia de “orientação a objetos” aplicada aos recursos da linguagem Fortran, organizando o código em múltiplos módulos e tipos compostos que simulam classes e métodos, promovendo reaproveitamento e legibilidade mesmo num ambiente predominantemente procedural (MOHID WATER MODELLING SYSTEM TEAM, 2025e).

3 TRABALHOS CORRELATOS

A literatura recente aponta diversos esforços voltados ao desenvolvimento de soluções otimizadas por meio de paralelismo com o uso da linguagem de programação Rust.

Sydow *et al.* (2020) apresentam uma nova otimização guiada por perfil para processamento de fluxo paralelo, tendo como objetivo equilibrar sistematicamente o ganho de desempenho que pode ser obtido da paralelização com a sobrecarga de comunicação. A ideia chave da abordagem para otimização guiada por perfil é, primeiramente, estimar rigorosamente as informações de desempenho para um determinado conjunto de tarefas e possíveis paralelizações em Rust para, na sequencia, explorar essas informações para equilibrar sistematicamente o ganho de desempenho que pode ser obtido da paralelização com custo de simultaneidade.

Para explorar todo o potencial do paralelismo, os autores adotaram os padrões “Passagem de Mensagens” (*Message Passing*), para a comunicação entre tarefas e para desacoplar suas execuções, “Fazenda de Tarefas” (*Task Farm Pattern*), para atingir o paralelismo de dados bifurcando um fluxo de mensagens para fornecer uma fazenda de instâncias de processamento paralelo para a mesma tarefa, e “*Chunking*”, ou “Processamento de Trens” (*Train Processing*), para lidar com a sobrecarga de comunicação introduzida pela passagem de mensagens e o roteamento da fazenda de tarefas (SYDOW *et al.*, 2020).

Com isso os autores conseguiram desenvolver uma extensão de um modelo de programação gráfica proposto anteriormente para aplicativos de processamento de fluxo paralelo baseados em Rust com um gerador de código paralelizante que utiliza de uma abordagem de otimização que combina criação de perfil com análise estática para estimar rigorosamente o desempenho de vários graus de paralelização e executa processamento de trem ideal, bem como alocação de tarefas gananciosas (SYDOW *et al.*, 2020).

Ionita, Manole e Slusanschi (2020) apresentam uma comparação de desempenho de várias implementações paralelas em um sistema multinúcleo com recursos de GPU e também em um FPGA incorporado dentro de um SoC em um aplicativo que implementa a propagação de sinais sem fio em um ambiente bidimensional, considerando reflexões e atenuação de sinal.

Os autores utilizam um modelo de perda de caminho livre para atenuação de potência de sinal, realizando algumas correções com o intuito de compensar fatores como perda de potência, fato de proporcionalidade, características no ambiente, entre outros (IONITA; MANOLE; SLUSANSCHI, 2020).

Quanto a implementação, inicialmente os autores realizaram de forma serial, mas organizado de forma a ser paralelizável, com o intuito de facilitar a sua compreensão

nesta fase inicial de programação, baseado nas equações do modelo apresentadas. Durante o processo de análise do código serial obtiveram-se alguns apontamentos que resultaram em um algoritmo que resulta em uma simulação que é executada em paralelo no espaço, mas serial no tempo (IONITA; MANOLE; SLUSANSCHI, 2020).

O algoritmo foi implementado com vários paradigmas de programação paralela, sendo eles OpenMP, TBB, Rust e CUDA, de modo que o modelo de propagação sem amortização foi implementado em OpenMP, TBB e Rust, enquanto o com amortização foi implementado em TBB, CUDA e HLS (IONITA; MANOLE; SLUSANSCHI, 2020).

Ao comparar o desempenho, os autores observaram que na implementação com amortização, CUDA obteve melhor desempenho, porém apresenta maior complexidade de uso devido a necessidade de entendimento do paradigma GP-GPU, assim como a existência de uma GPU. Quanto a implementação sem amortização, OpenMP obteve maior desempenho, porém Rust ganha quanto a segurança, tendo em vista o fato de que *deadlocks* e condições de corrida são eliminados pelas regras de tipagem fortes e pelo sistema de propriedade exclusivo, além de apresentar menor quantidade de bugs durante a configuração dos experimentos (IONITA; MANOLE; SLUSANSCHI, 2020).

HoseinyFarahabady *et al.* (2019) propõem uma técnica para alocação de recursos da CPU com o objetivo de melhorar a latência do tempo de resposta em aplicativos com modelos de programação de fluxo de dados oportuno com diferentes níveis de qualidade. Para isso sua solução prevê o desempenho esperado da plataforma subjacente usando uma abordagem *on-line* baseada na teoria de filas, ajustando as correções necessárias na alocação da CPU visando atingir o melhor desempenho.

No estudo os autores consideraram uma classe de problema de alocação dinâmica de CPU onde se receberia um conjunto de aplicativos TDF, cada um com um atraso máximo tolerável predefinido para realizar sua computação sobre dados de entrada. Cada aplicativo TDF recebe vários fluxos de dados de *streaming* ao longo do tempo, cada um com uma taxa que é desconhecida para o provedor de serviços e pode mudar aleatoriamente ao longo do tempo, com isso os autores desenvolveram um algoritmo de Controle Preditivo de Modelo (MPC) de baixo *overhead* para o problema de alocação dinâmica de CPU de aplicativos TDF distintos, cada um com seu próprio nível de desempenho predefinido (HOSEINYFARAHABADY *et al.*, 2019).

Foram realizados uma série de experimentos usando fluxos de dados sintéticos em larga escala sobre uma implementação de estrutura de fluxo de dados oportuno escrita em Rust para validar o algoritmo de limite de CPU proposto, os resultados de desempenho foram comparados com os obtidos por três heurísticas empíricas implementadas em muitos pacotes comerciais, sendo *Weighted Round-Robin* (WRR), *Fixed Priority Scheduling* (FPS) e *Class-Based Weighted Fair Queueing* (CFWFQ) (HOSEINYFARAHABADY *et al.*, 2019).

Com isso os autores puderam observar que a escolha da política de alocação de CPU pode afetar significativamente o tempo médio de processamento do algoritmo de

junção, especialmente em cenários com operações iterativas mais complexas e registros de dados mais volumosos. Além disso, a medição da utilização em todo o cluster em todos os núcleos da CPU confirma que a estratégia de controle proposta pode manter o nível de utilização dos núcleos da CPU entre 55% e 90% quando a demanda da carga de trabalho é alta. Deste modo observou-se que o controlador proposto aumenta a utilização média de todos os núcleos da CPU em 28,3% (máximo de 46,9%) em comparação com o melhor resultado de outras heurísticas (HOSEINYFARAHABADY *et al.*, 2019).

Também observou-se nos experimentos que o desequilíbrio da carga de trabalho é altamente relevante para a estratégia de limite de CPU operada, sendo exacerbada em escala conforme o tamanho dos dados recebidos cresce. Porém, ao contrário de outras heurísticas que se esforçam para distribuir o número de tuplas entre os trabalhadores da forma mais uniforme possível, sem considerar as condições de tempo de execução e as cargas acumuladas em cada *buffer*, a solução proposta explora a distorção nos *buffers* intermediários ao diminuir/aumentar dinamicamente o limite de CPU (HOSEINYFARAHABADY *et al.*, 2019).

Pieper *et al.* (2021) apresentam o Rust-SSP, uma abstração de programação paralela de alto nível para a linguagem Rust, voltada a aplicações de processamento de fluxos em arquiteturas *multi-cores* com memória compartilhada.

Os autores destacam que, embora Rust ofereça segurança de memória e concorrência confiável, sua ecossistema ainda carece de ferramentas e bibliotecas que implementem modelos de programação paralela estruturada, amplamente explorados em C++. Assim, o Rust-SSP busca preencher essa lacuna, oferecendo uma interface simples, eficiente e segura, que dispensa o uso de *Unsafe* Rust e suporta estágios com e sem estado (PIEPER *et al.*, 2021).

O estudo também propõe uma nova suíte de *benchmarks* composta por quatro aplicações (Micro-bench, Processamento de Imagem, Compressão Bzip2 e Detecção de Olhos) e três níveis de carga de trabalho (pequena, média e grande), utilizada para comparar o Rust-SSP com as bibliotecas paralelas de Rust Tokio, Rayon, Pipeliner e std::threads. Os experimentos foram conduzidos em um ambiente *multi-core* com 40 *threads* de hardware (PIEPER *et al.*, 2021).

Os resultados demonstraram que o Rust-SSP obteve até 41,1% de desempenho superior em relação às demais soluções, além de demandar o menor número de linhas de código adicionais para habilitar o paralelismo. As interfaces Tokio e std::threads apresentaram boa expressividade, mas menor desempenho em cenários desbalanceados ou com necessidade de reordenação de dados. Já Rayon e Pipeliner apresentaram limitações quanto ao suporte a estágios com estado (PIEPER *et al.*, 2021).

Martins *et al.* (2025) propõem uma versão em Rust da suíte *benchmark NAS Parallel Benchmarks* (NPB), tradicionalmente usada para avaliar hardware e *frameworks* de paralelismo em HPC.

O trabalho apresenta como contribuições principais a implementação da NPB em Rust (denominada NPB-Rust), a análise da expressividade e desempenho dos recursos da linguagem Rust (sequential e paralelo) no contexto de aplicações científicas, e a comparação de desempenho entre Rust, Fortran e C++ em versões sequenciais, bem como comparação de Rust com C++/OpenMP em versões paralelas (MARTINS *et al.*, 2025).

Nos experimentos, a versão sequencial em Rust apresentou desempenho cerca de 1,23% mais lenta que a implementação em Fortran, e 5,59% mais rápida que C++. Já na versão paralela com a biblioteca Rayon, Rust ficou atrás tanto de Fortran quanto de C++/OpenMP em suas medições (MARTINS *et al.*, 2025).

Os autores discutem os desafios do paralelismo em Rust, tais como gerenciamento de dados, sincronização de *threads* e uso de blocos *unsafe*, e apontam que, embora Rust ofereça benefícios de segurança de memória e expressividade, ele ainda enfrenta limitações de maturidade de infraestrutura e de *frameworks* de paralelismo comparados às linguagens consolidadas em HPC (MARTINS *et al.*, 2025).

Conclui-se que a construção de uma suíte de *benchmark* em Rust contribui para a comunidade de HPC ao fornecer uma base para comparações sistemáticas, e que o desempenho de Rust já se aproxima das implementações tradicionais, embora existam fatores de sobrecarga que requerem estudo adicional (MARTINS *et al.*, 2025).

Tais trabalhos sustentam a hipótese de que Rust possui potencial como alternativa para o desenvolvimento de algoritmos voltados à HPC que executam de forma rápida e segura.

4 METODOLOGIA

A metodologia adotada neste trabalho teve como objetivo avaliar o potencial da linguagem de programação Rust para o desenvolvimento de aplicações científicas de alto desempenho, por meio da análise comparativa entre implementações equivalentes de um mesmo algoritmo, originalmente desenvolvido em Fortran e integrante do software de modelagem hidrodinâmica MOHID.

Os códigos desenvolvidos e utilizados neste estudo encontram-se disponíveis no repositório público `https://github.com/Boreias/INE5660`

O processo metodológico foi estruturado em quatro etapas principais:

1. Análise e seleção da subrotina;
2. Isolamento e reestruturação do código para execução em ambiente controlado;
3. Reimplementação em Rust nas versões sequencial e paralela;
4. Execução dos experimentos e análise estatística dos resultados.

Cada uma dessas etapas é detalhada a seguir.

4.1 ANÁLISE DO CÓDIGO-FONTE E SELEÇÃO DA SUBROTINA

Inicialmente, foi realizada uma análise detalhada do código-fonte do software MOHID, desenvolvido majoritariamente na linguagem Fortran (83,7%), com o objetivo de identificar uma subrotina que atendesse aos seguintes critérios:

- Implementação sequencial, mas com potencial para paralelização;
- Baixa dependência em relação a outros módulos do sistema, facilitando seu isolamento;
- Relevância computacional para o funcionamento geral do software.

A subrotina selecionada foi ConstructTriangulationXYZ, pertencente ao módulo de triangulação e utilizada pelo módulo lagrangiano global (ModuleLagrangianGlobal) em sua rotina de construção. Essa subrotina é responsável pela criação das estruturas fundamentais empregadas na geração dos cenários hidrodinâmicos no MOHID. Foi considerada adequada para o estudo por depender apenas do módulo de dados globais (ModuleGlobalData), por sua importância no processo de modelagem e pela presença de múltiplos loops aninhados, que indicam elevado potencial de paralelização.

4.2 ESTRUTURA E FUNCIONAMENTO DA SUBROTINA ORIGINAL

A subrotina ConstructTriangulationXYZ pode ser dividida em duas etapas principais. Na primeira etapa, são lidos três *arrays* de entrada contendo as coordenadas X, Y e Z da área de estudo. Essa etapa envolve o pré-processamento e a atribuição dos valores aos atributos do tipo de dado T_Triangulation, definido no módulo em questão. Devido à presença de dependências internas e de instruções GOTO, inexistentes na linguagem Rust, a transcrição direta dessa parte do código exigiria uma refatoração estrutural significativa, o que poderia comprometer a equivalência entre as implementações e, consequentemente, a validade da comparação de desempenho.

Por esse motivo, optou-se por isolar essa etapa e desenvolver um projeto auxiliar responsável por ler os dados de entrada, identificar as coordenadas de cada array e gerar arquivos intermediários no formato .txt, contendo o estado atual da aplicação até aquele ponto. Esses arquivos serviram como entrada padronizada para as implementações subsequentes, garantindo que todas as versões processassem exatamente os mesmos dados.

A segunda etapa da subrotina é responsável pelos cálculos geométricos e pela construção das estruturas internas da triangulação. Nessa fase, são realizadas operações como o cálculo das coordenadas dos centróides dos triângulos, raios e distâncias, populando o tipo T_Triangulation com objetos dos tipos T_Node, T_Triangle e T_DirectedEdge.

Embora exista dependência entre os resultados das três funções que compõem essa etapa, cada uma delas opera de forma independente sobre os dados pré-processados, o que permite sua implementação paralela sem comprometer a consistência dos resultados.

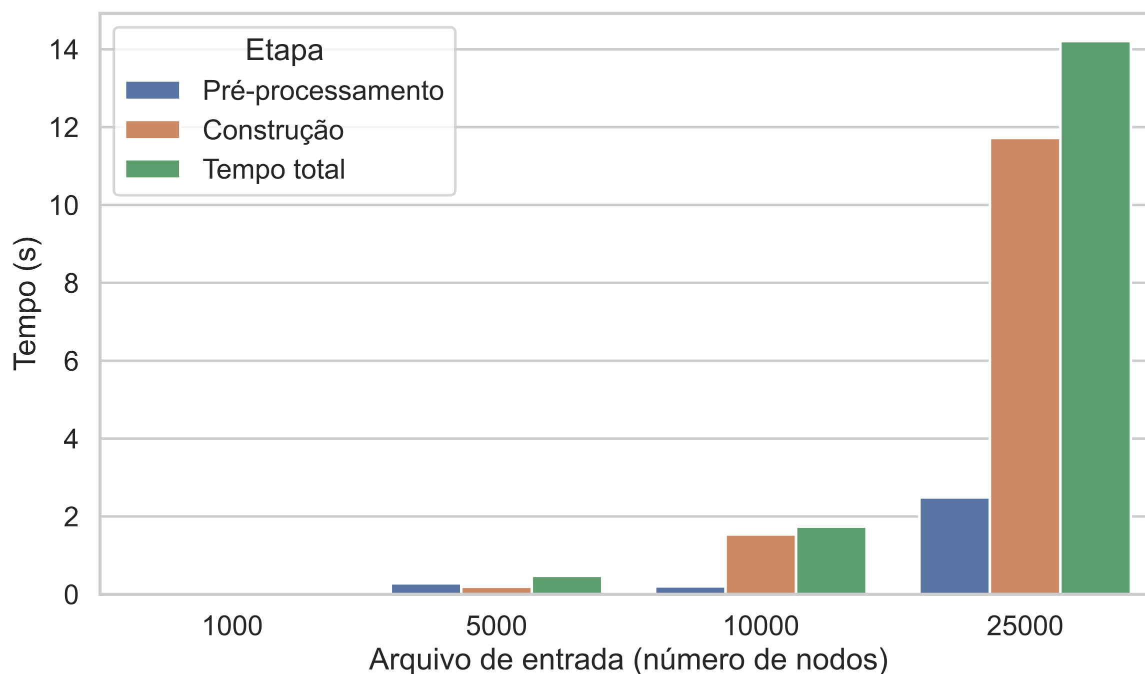
Com o objetivo de verificar a pertinência de processar apenas a segunda etapa da subrotina, foram realizados experimentos para medir o tempo de execução de cada etapa em relação ao tempo total da subrotina. Para esse experimento, foram utilizados arquivos de entrada contendo 1000, 5000, 10000 e 25000 nodos, sendo cada arquivo executado dez vezes. O gráfico da Figura 1 apresenta os resultados obtidos.

Também foi identificado que a função ConstructDirectedEdges é responsável por quase a totalidade do tempo de processamento, conforme observado no gráfico da Figura 2.

A análise do código dessa função evidenciou que ela pode ser subdividida em duas etapas: a primeira é responsável por popular o atributo de T_Triangulation com elementos do tipo T_DirectedEdge, enquanto a segunda realiza a busca pelos elementos recém-criados, visando popular os atributos counter_clock_edge de T_DirectedEdge e first_edge de T_Node, completando o ciclo de construção da triangulação.

Para identificar o impacto de cada etapa no tempo total de processamento da função, foram realizados experimentos utilizando arquivos de entrada com 1000, 5000,

Figura 1 – Tempo médio por etapa do algoritmo ConstructTriangulationXYZ



Fonte: o autor

10000 e 25000 nodos, sendo cada arquivo executado dez vezes. O gráfico da Figura 3 apresenta os resultados obtidos, nos quais se observa que ambas as etapas apresentam tempos de processamento relativamente próximos, embora a etapa destinada à construção dos elementos T_DirectedEdge apresente valores ligeiramente superiores.

4.3 REIMPLEMENTAÇÃO EM RUST

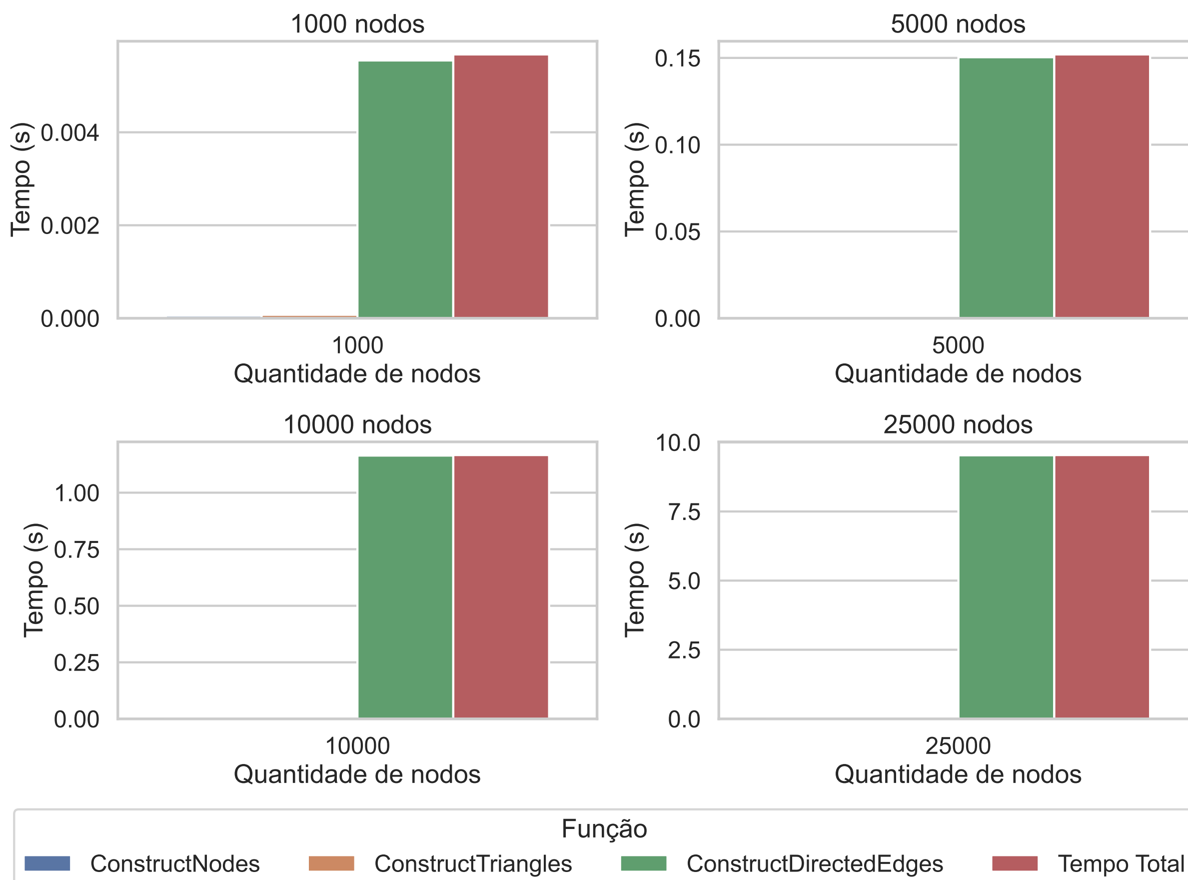
Com base na estrutura descrita, foi desenvolvida uma implementação em Rust dividida em duas versões principais:

1. Versão Sequencial — Implementada de forma fiel à lógica original em Fortran, buscando reproduzir exatamente o comportamento da subrotina ConstructTriangulationXYZ.

2. Versão Paralela — Desenvolvida com o uso da biblioteca Rayon, explorando o paralelismo de dados de forma simples e eficiente. As iterações dos *loops* foram distribuídas entre a quantidade de *threads* recebida como parâmetro, utilizando diferentes estruturas de dados e métodos nativos da linguagem, com o objetivo de obter o melhor desempenho possível.

Devido às características intrínsecas de cada linguagem, foram necessários pequenos ajustes para o correto funcionamento do programa, descritos a seguir:

Figura 2 – Tempo médio de execução das funções do algoritmo ConstructTriangulationXYZ em Rust sequencial (escalas independentes)



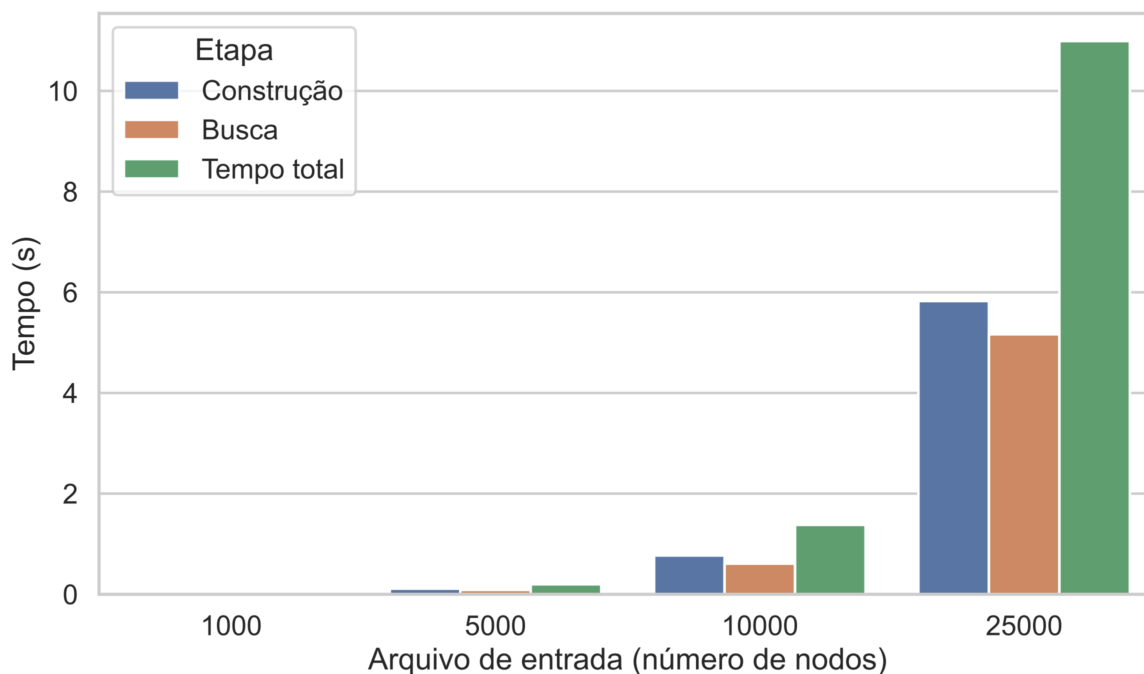
Fonte: o autor

- Valores numéricos: A linguagem Fortran conta com dois tipos principais para representar valores numéricos: integer (para valores inteiros) e real (para valores de ponto flutuante). Já Rust dispõe dos tipos apresentados na Tabela 1. Visando reduzir a quantidade de conversões de tipo (*casting*), foi realizada uma análise dos valores armazenados em cada atributo de T_Triangulation. Como resultado, utilizou-se o tipo usize para valores do conjunto dos naturais, i32 para inteiros e f32 para valores de ponto flutuante (real).

- Arrays e vetores: O código original em Fortran utiliza *arrays* como estrutura de dados para o armazenamento de conjuntos. Entretanto, enquanto em Fortran o tamanho dos *arrays* pode ser definido em tempo de execução, em Rust ele deve ser definido em tempo de compilação. Considerando que o tamanho dos conjuntos é diretamente proporcional à quantidade de nodos presente nos arquivos de entrada, a utilização de *arrays* em Rust torna-se inviável. Dessa forma, optou-se pelo uso de vetores, que armazenam dados de forma semelhante, mas permitem redimensionamento dinâmico em tempo de execução.

- Valores padrão: Em Fortran, ao se utilizar *arrays*, é necessária a alocação de

Figura 3 – Tempo médio por etapa da função ConstructDirectedEdges



Fonte: o autor

espaço na memória; nesse processo, valores aleatórios podem ser armazenados temporariamente e substituídos ao longo do processamento. Já em Rust, ao se utilizar vetores, é necessário definir explicitamente os valores a serem armazenados no momento da alocação. Assim, adotou-se o valor 0 para atributos do tipo usize, 0.0 para valores do tipo f32 e false para valores do tipo booleano.

- Orientação a objetos: O Fortran 90 não possui suporte completo à orientação a objetos, introduzida apenas a partir do padrão Fortran 2003. Assim, o código original faz uso de tipos compostos (*derived types*) para representar estruturas semelhantes a classes, cabendo aos métodos definidos nos módulos a responsabilidade de modificar os valores desses tipos. De forma análoga, embora Rust também não seja uma linguagem orientada a objetos no sentido clássico, ela oferece o tipo de dado *struct* para representar entidades compostas e o bloco *impl* para associar funções e métodos a essas estruturas.

Ambas as versões realizam a leitura dos arquivos intermediários, executam as três funções responsáveis pela construção das estruturas da triangulação e, ao final, geram um arquivo .txt contendo o estado da aplicação. Esse arquivo permite verificar a consistência entre os resultados obtidos em cada implementação.

4.3.1 Implementação Paralela em Rust

A implementação paralela em Rust foi desenvolvida com o objetivo de maximizar o desempenho do processo de construção da triangulação. Para isso, foi adotada a abordagem de paralelismo de dados por meio da biblioteca Rayon, em conjunto com o uso de métodos e estruturas de dados nativas da linguagem. Considerando o impacto significativo da função ConstructDirectedEdges no tempo total de processamento da subrotina ConstructTriangulationXYZ, observado no gráfico da Figura 1, optou-se por apenas paralelizar as iterações das funções ConstructNodes e ConstructTriangles, direcionando esforços para realizar, além da paralelização, uma refatoração da função ConstructDirectedEdges. O gráfico da Figura 3 demonstra um balanceamento adequado do tempo de processamento entre as etapas da função, razão pela qual ambas foram refatoradas.

Na etapa de construção das arestas, o código original em Fortran utiliza *arrays* de T_DirectedEdges, o que implica acesso a regiões não contíguas de memória e pode resultar em *cache misses*, reduzindo o desempenho geral do programa. Além disso, a estrutura contém propriedades que não são calculadas no processo de construção, ocasionando alocação de memória superior à necessária. Na versão paralela, optou-se pela utilização de estruturas temporárias simplificadas, contendo apenas os atributos relevantes ao processo e armazenadas em vetores contínuos, facilitando a otimização pelo compilador. Também foram removidas as chamadas às funções “AlreadyInList” e “AddEdge”, ambas pequenas, porém executadas milhares de vezes dentro do laço principal, o que poderia ocasionar perda de contexto e possíveis *branch misses*; na versão paralela, suas lógicas foram executadas *inline*. A função “AlreadyInList” tem por finalidade verificar se uma determinada aresta já se encontra na lista, porém realiza uma busca linear, percorrendo o *array* de T_DirectedEdges, o que resulta em complexidade O(n) no caso médio. Em Rust, optou-se por utilizar tabelas *hash*, cuja complexidade é O(1) para o caso médio, sendo a mesma abordagem aplicada também à etapa de busca de ConstructDirectedEdges, reduzindo consideravelmente o tempo de execução da função. É importante destacar que o MOHID foi implementado em Fortran 90, versão que não dispõe de estrutura nativa equivalente a tabelas *hash*, as quais foram introduzidas apenas a partir do Fortran 2003.

4.4 VARIAÇÕES DOS EXPERIMENTOS

Para fins de análise de desempenho, foram desenvolvidas duas variações experimentais de cada implementação.

A primeira, denominada Versão Duration, tem como foco a mensuração do tempo de processamento em nível de código-fonte, medindo o tempo de execução individual de cada função (ConstructNodes, ConstructTriangles e ConstructDirectedEdges) e o

tempo total de processamento, desconsiderando as operações de entrada e saída (I/O) relacionadas à leitura e escrita em disco. Os tempos medidos são registrados no arquivo de saída .txt, permitindo a comparação detalhada entre as diferentes versões.

A segunda variação, denominada Versão General, foi executada utilizando a ferramenta perf do sistema operacional Linux, com o objetivo de capturar métricas de desempenho em nível de sistema, incluindo tempo total de execução, utilização de CPU, acessos a cache, número de instruções e ciclos. Diferentemente da versão anterior, esta engloba todo o processo de execução, contemplando as etapas de leitura de arquivos, processamento e gravação dos resultados.

4.5 CONFIGURAÇÃO DOS EXPERIMENTOS

Os experimentos foram conduzidos utilizando sete arquivos de entrada distintos, contendo 1.000, 5.000, 10.000, 25.000, 50.000, 75.000 e 100.000 nodos, representando cenários de complexidade crescente. Cada versão sequencial foi executada dez vezes para cada arquivo de entrada, enquanto as versões paralelas foram executadas dez vezes para cada arquivo e para cada quantidade de *threads* utilizadas, variando de 1 a 20 *threads*, sendo as execuções em 1 *thread* utilizadas para análise de desempenho da refatoração. Foi criado um arquivo .sh automatização da execução dos experimentos, onde foram realizadas iterações por caso de experimento e arquivo de entrada na implementação sequencial, e por caso de experimento, número de *threads* e arquivo de entrada na implementação paralela, visando reduzir os impactos provenientes de cache entre os experimentos. Essa metodologia visou garantir um conjunto robusto de dados para análise estatística, reduzindo o impacto de variações ocasionadas por ruídos do sistema.

A compilação das implementações em Rust foi realizada por meio da instrução “cargo build –release”, enquanto a compilação das implementações em Fortran foi realizada por meio da instrução “gfortran -g -O3 -march=native ModuleGlobalData.F90 TCCModule.F90 TCCMain.F90 -o tcc”

- -g: inclui símbolos de depuração no arquivo binário, possibilitando depuração de código com ferramentas como gdb e perf (STALLMAN; THE GCC DEVELOPER COMMUNITY, 2003);
- -O3: ativa o conjunto mais agressivo de otimizações automáticas visando redução do tempo de processamento, porém pode acarretar em aumento no uso de CPU, registradores e cache (STALLMAN; THE GCC DEVELOPER COMMUNITY, 2003);
- -march=native: gera código otimizado para a microarquitetura do processador (STALLMAN; THE GCC DEVELOPER COMMUNITY, 2003).

As métricas de desempenho foram obtidas por meio da ferramenta perf do sistema operacional Linux, abrangendo os seguintes indicadores:

- Tempo de CPU (“*cpu-clock*”)
- Quantidade de ciclos (“*cycles*”)
- Falha de cache (“*cache-misses*”)
- Referências de cache (“*cache-references*”)
- Previsão de ramifinicação (“*branch-misses*”)
- Ramificações (“*branches*”)
- Faltas de página (“*page-faults*”)

Os experimentos foram executados em um microcomputador cujas características de hardware e software estão descritas na Tabela 3

Tabela 3 – Configuração do computador.

Sistema operacional	Ubuntu 24.04.3 LTS
Memória RAM	31Gb
Arquitetura do Processador	x86_64
Modo(s) operacional da CPU	32-bit, 64-bit
Address sizes	46 bits physical, 48 bits virtual
Ordem dos bytes	Little Endian
CPU(s)	12
Lista de CPU(s) on-line	0-11
Nome do modelo	Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz
Família da CPU	6
Modelo	85
Thread(s) por núcleo	2
Núcleo(s) por soquete	6
Soquete(s)	1
GPU	NVIDIA Corporation GP107GL [Quadro P1000] (rev a1)

Fonte: O autor

4.6 COLETA E TRATAMENTO DOS DADOS

Após a execução dos experimentos, foi realizada a análise dos arquivos gerados para aferição dos resultados. Nessa etapa, observaram-se duas discrepâncias principais entre as diferentes versões, sendo estas consideradas aceitáveis dentro do escopo deste trabalho, uma vez que não comprometem a validade da comparação de desempenho entre as versões, as discrepâncias encontradas foram:

- Valores não calculados: por se tratar de um método construtor, alguns atributos das classes não são calculados nesse estágio, cabendo sua definição a métodos modificadores. Nessas situações, a diferença nos métodos de inicialização resultou em valores distintos entre as implementações — em Fortran foram observados valores aleatórios, enquanto em Rust foram utilizados valores padrão de inicialização.
- Arredondamentos: verificou-se uma variação de até uma unidade para mais ou para menos a partir da sexta casa decimal. Essa discrepância foi atribuída a diferenças nos processos de arredondamento durante a execução dos cálculos.

Na sequência, foi desenvolvido um script em Python responsável por percorrer todos os arquivos gerados pelas diferentes variações de implementação, consolidando os resultados em arquivos CSV específicos para cada caso. Em seguida, elaborou-se um notebook Jupyter, também em Python, destinado à realização do processamento estatístico dos resultados, incluindo:

- Cálculo das médias e dos desvios padrão das métricas analisadas;
- Comparações diretas entre as linguagens e versões;
- Geração de gráficos e visualizações para suporte à interpretação dos resultados.

Essas análises permitiram identificar o comportamento de desempenho das versões em Fortran e Rust, nas modalidades sequencial e paralela, considerando diferentes tamanhos de entrada e quantidades de *threads*. Os resultados obtidos possibilitaram uma avaliação abrangente do potencial da linguagem Rust no contexto de aplicações científicas de alto desempenho.

5 RESULTADOS

5.1 COMPARAÇÃO DAS IMPLEMETAÇÕES SEQUENCIAIS

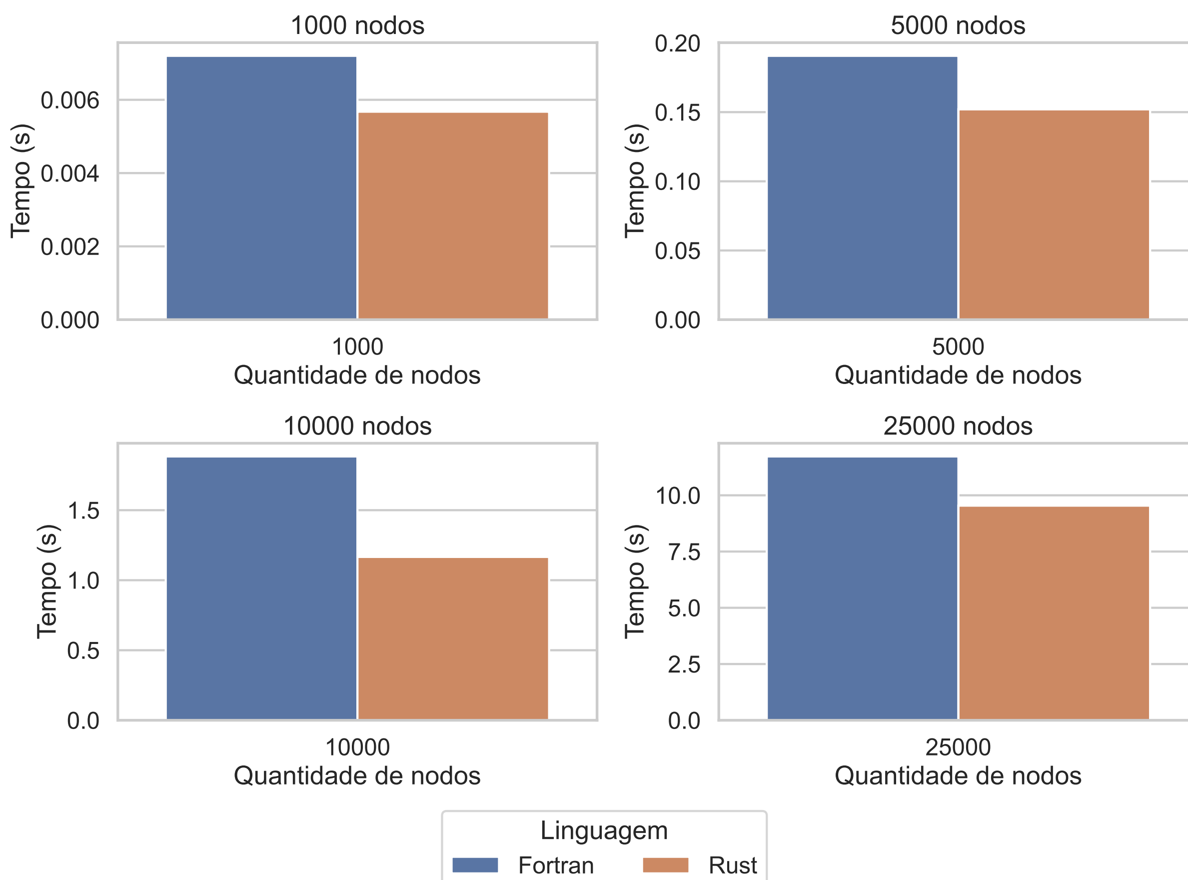
Analisando o gráfico da Figura 4, observa-se que o tempo médio de execução das funções ConstructNodes, ConstructTriangles e ConstructDirectedEdges na linguagem Rust é aproximadamente 20% menor em comparação à implementação em Fortran. Esse resultado é corroborado pelo gráfico da Figura 5, que demonstra um menor número total de ciclos de CPU na versão em Rust, o que pode indicar uma melhor otimização do código compilado, um número reduzido de instruções executadas ou menor latência de acesso à memória. Entretanto, é importante destacar que, devido às diferenças estruturais na forma de manipulação de *arrays* entre as duas linguagens, não é possível afirmar com precisão se essa melhoria decorre exclusivamente da mudança de linguagem, do uso de vetores em substituição aos *arrays* ou da combinação de ambos os fatores.

A análise dos dados obtidos demonstrou que a gerência de cache na linguagem Rust apresentou desempenho superior em comparação à implementação em Fortran, realizando menos acessos ao cache, conforme ilustrado no gráfico da Figura 6, e registrando um menor número de faltas de acesso, como observado no gráfico da Figura 7. Esses resultados indicam um uso mais eficiente da hierarquia de memória pela linguagem Rust, o que possivelmente contribuiu de forma significativa para a redução do tempo de processamento observado na Figura 4.

Em relação à quantidade de ramificações, observou-se uma redução significativa na implementação em Rust, conforme apresentado no gráfico da Figura 8. Esse resultado pode estar diretamente relacionado à necessidade de execução de um maior número de instruções na implementação em Fortran — como o registro de módulos em memória, alocação e liberação de espaço, entre outras operações — ou ainda à melhor otimização do código compilado em Rust. Não se descarta, entretanto, a influência do processo de leitura do arquivo intermediário sobre essa métrica, considerando a menor familiaridade do autor com a linguagem, o que pode ter resultado em um número maior de verificações e, consequentemente, em mais ramificações. Outro fator relevante é a possível reorganização do código realizada pelo compilador Rust, que pode reduzir a quantidade total de ramificações ao aproximá-las no fluxo de execução, porém aumentando sua imprevisibilidade. Essa característica explicaria a maior incidência de branch misses observada na implementação em Rust, conforme ilustrado no gráfico da Figura 9.

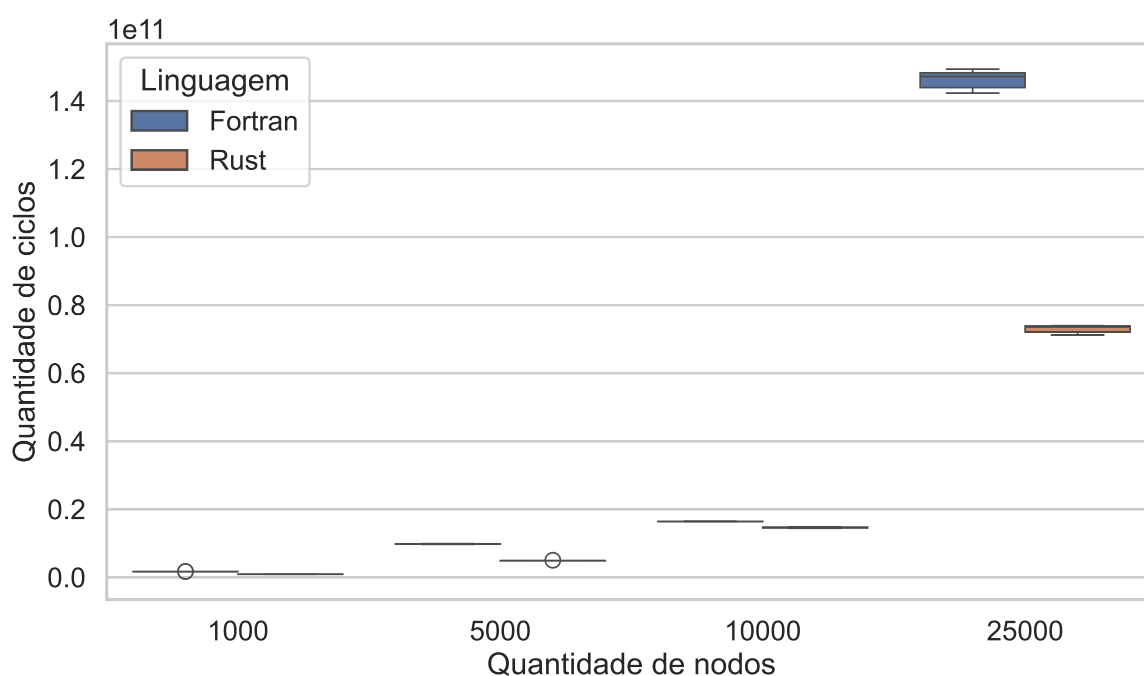
No que se refere à quantidade de faltas de página (*page faults*), os dados apresentados no gráfico da Figura 10 indicam uma menor incidência na implementação em Fortran, o que sugere um uso mais eficiente da memória virtual, com acessos mais localizados e previsíveis aos dados ou menor volume de informações carregadas

Figura 4 – Tempo médio de execução do processamento de dados (escalas independentes)



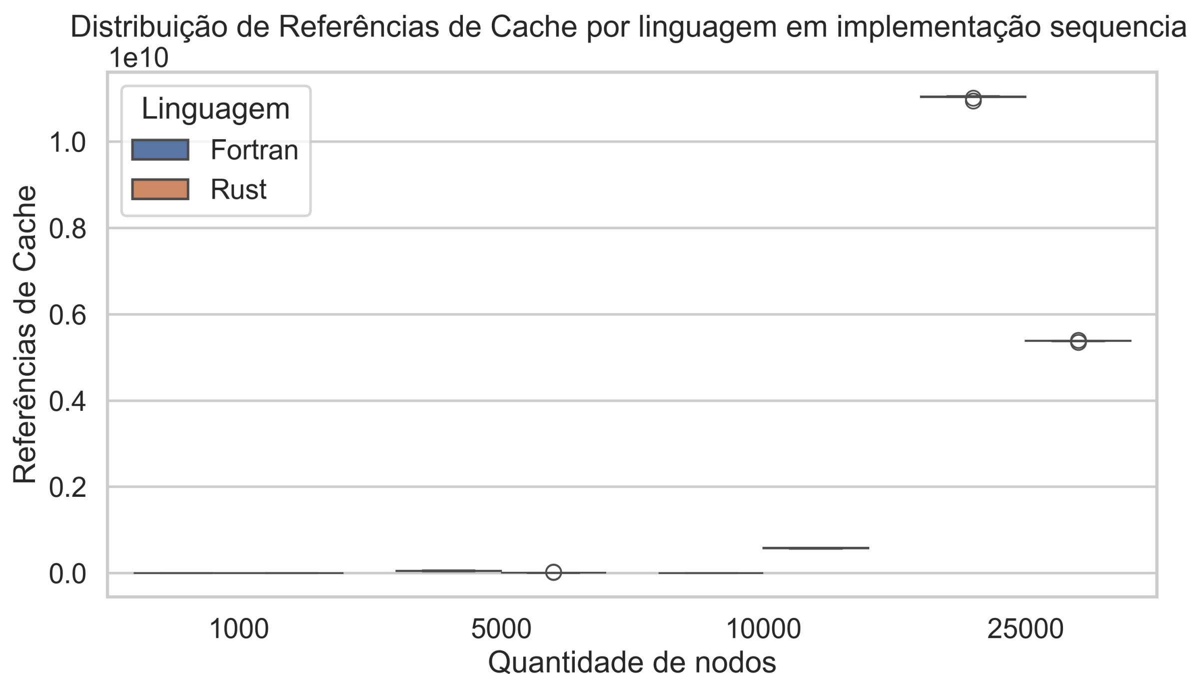
Fonte: o autor

Figura 5 – Distribuição de quantidade de ciclos por linguagem em implementação sequencial



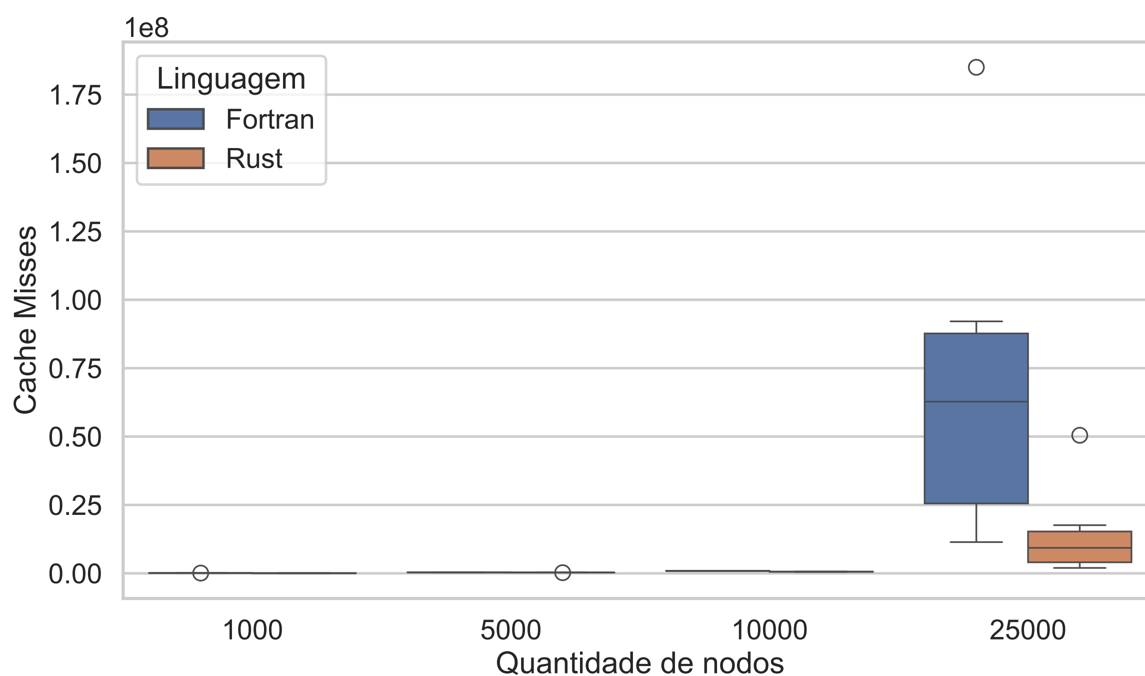
Fonte: o autor

Figura 6 – Distribuição de Referências de *Cache* por linguagem em implementação sequencial



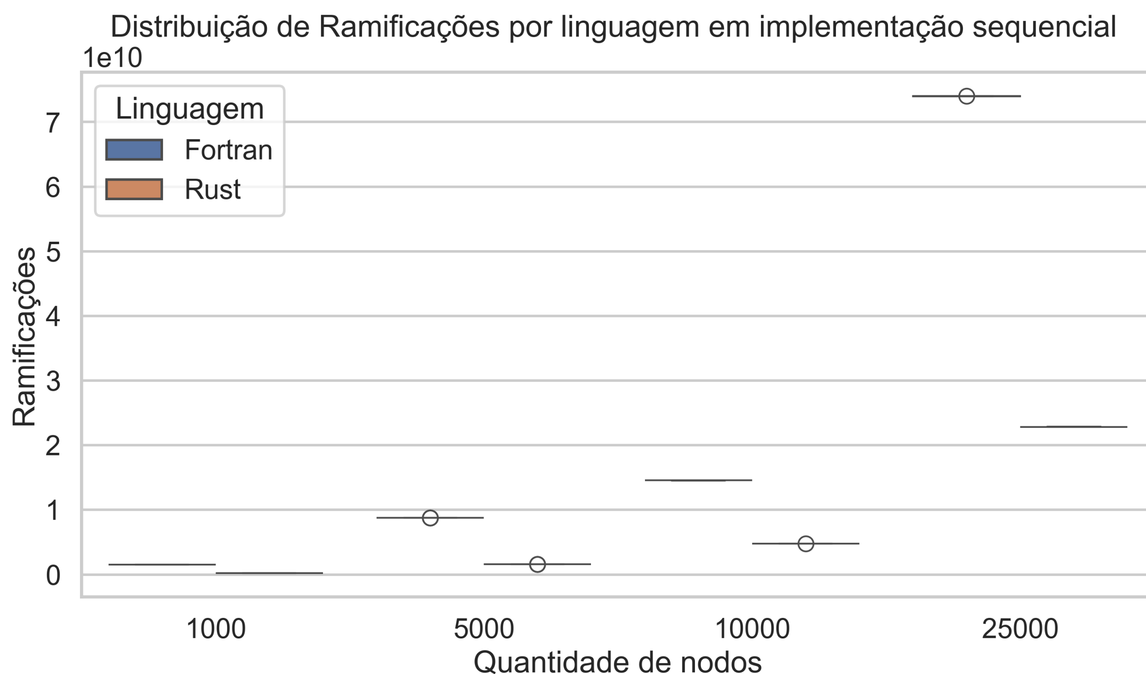
Fonte: o autor

Figura 7 – Distribuição de *Cache Misses* por linguagem em implementação sequencial



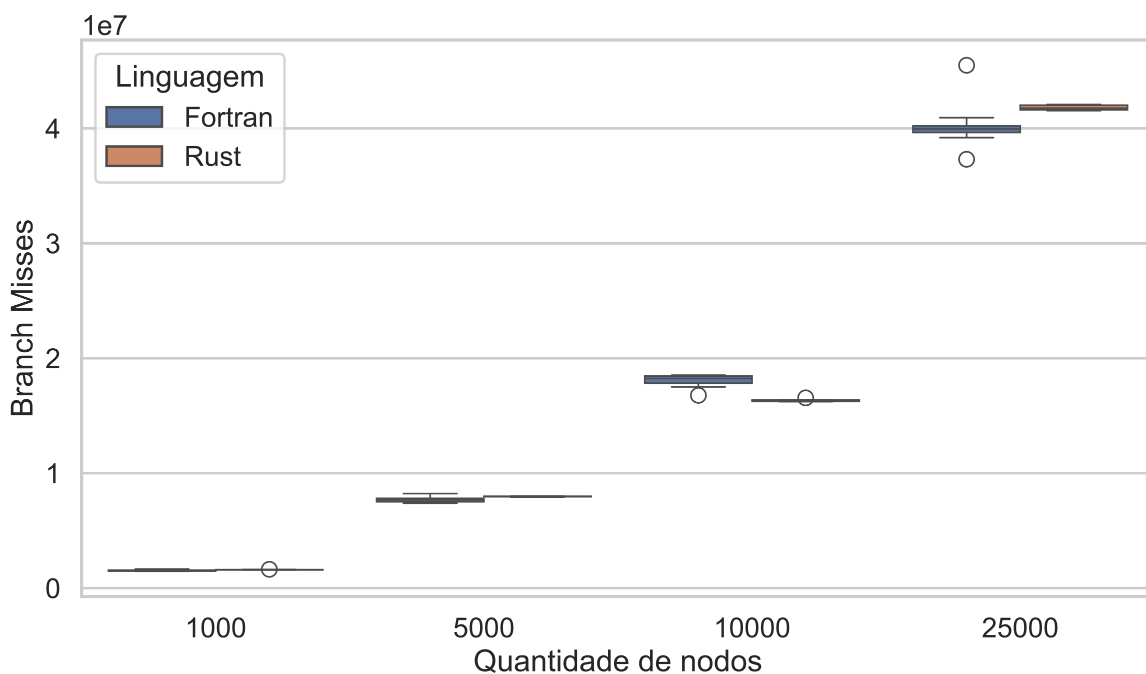
Fonte: o autor

Figura 8 – Distribuição da Quantidade de *Branches* por linguagem em implementação sequencial



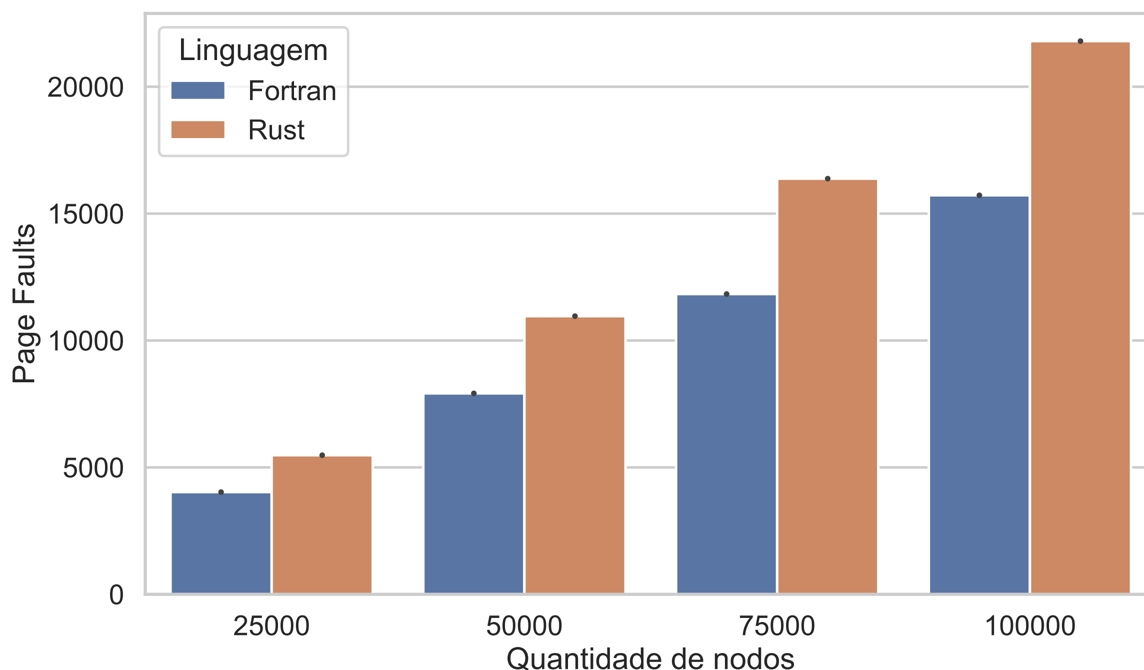
Fonte: o autor

Figura 9 – Distribuição de *Branch Misses* por linguagem em implementação sequencial



Fonte: o autor

Figura 10 – Distribuição de *Page Faults* por linguagem em implementação sequencial



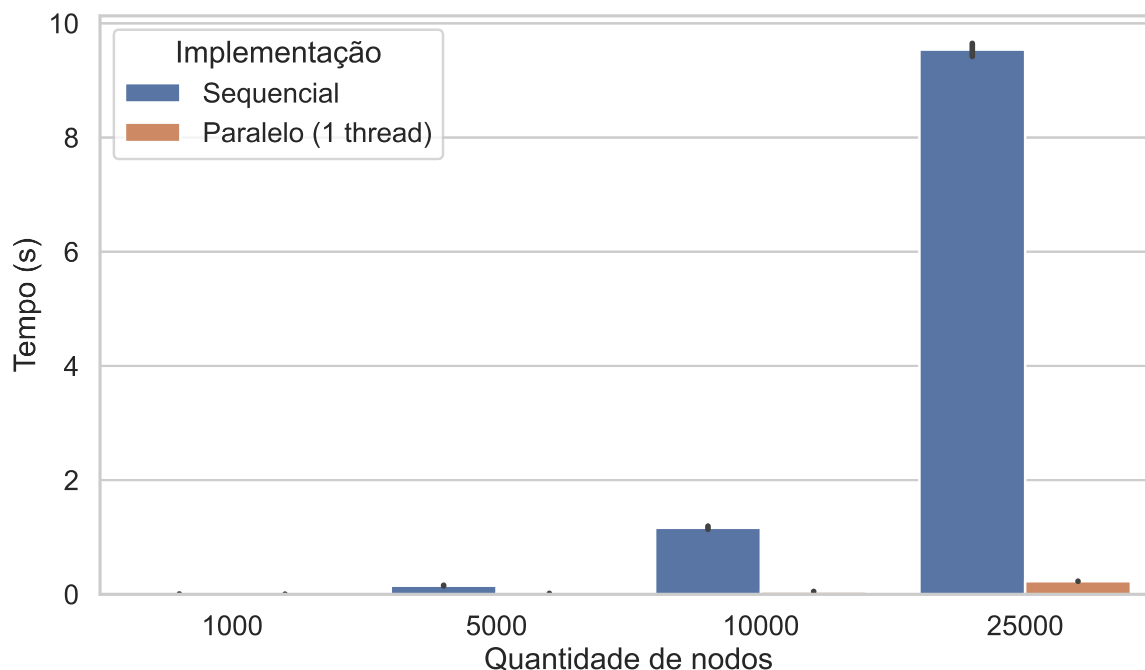
Fonte: o autor

dinamicamente. Um dos fatores que podem ter contribuído positivamente para esse resultado é o uso de *arrays*, uma vez que, em Rust, o tamanho dessa estrutura de dados deve ser definido em tempo de compilação, o que inviabiliza seu uso para armazenar conjuntos de tamanho variável. Assim, tornou-se necessária a utilização de vetores, que, embora permitam a criação em tempo de execução com tamanho específico, apresentam alocação dinâmica, podendo resultar em um número maior de faltas de página.

5.2 COMPARAÇÃO ENTRE A IMPLEMENTAÇÃO SEQUENCIAL EM RUST E A VERSÃO PARALELA EXECUTADA EM 1 *THREAD*

Ao comparar a implementação paralela executada em 1 *thread* com a implementação sequencial, apresentado no gráfico da Figura 11, observa-se que o código refatorado reduz substancialmente o tempo necessário para processar todas as quantidades de nodos avaliadas, mantendo uma vantagem consistente em toda a faixa de experimentação. Essa diferença é corroborada pelo gráfico da Figura 12, onde pode-se observar que a versão sequencial consome um número muito maior de ciclos para realizar o mesmo trabalho, enquanto a versão paralela em 1 *thread* utiliza significativamente menos ciclos, evidenciando uma execução mais eficiente por iteração e por acesso à memória. Assim, fica claro que a refatoração do algoritmo elimina redundâncias, reduz dependências internas e permite que o compilador Rust aplique

Figura 11 – Distribuição do tempo de execução por implementação sequencial em Rust



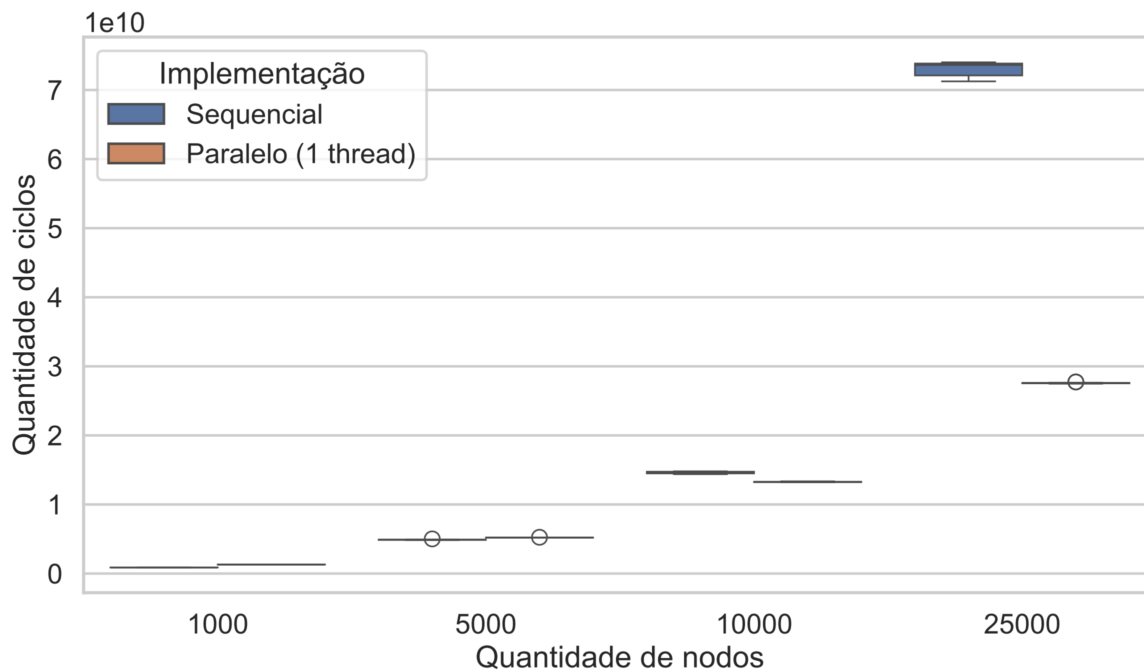
Fonte: o autor

mais otimizações do que o código baseado na implementação original em Fortran 90, resultando em ganhos expressivos de desempenho mesmo em execução *monothread*.

Quanto ao uso de *cache*, o gráfico da Figura 13 mostra que a implementação paralela em 1 *thread* apresenta valores consistentemente menores do que a implementação sequencial em Rust baseada fielmente no Fortran. Isso indica que a reorganização do algoritmo resultou em um padrão de acesso mais amigável à hierarquia de *cache*, informação corroborada pelo gráfico da Figura 14 onde observa-se uma quantidade muito inferior de referências de *cache* na versão paralela executada em 1 *thread*. Logo, o ganho não decorre apenas de maior intensidade de acessos, mas sim de melhor aproveitamento da localidade de dados, o que reduz *cache misses* e diminui o custo de recuperação de informações da memória principal. Esses resultados deixam claro que a refatoração aprimorou substancialmente o comportamento de memória do algoritmo, contribuindo para a redução do tempo final de execução.

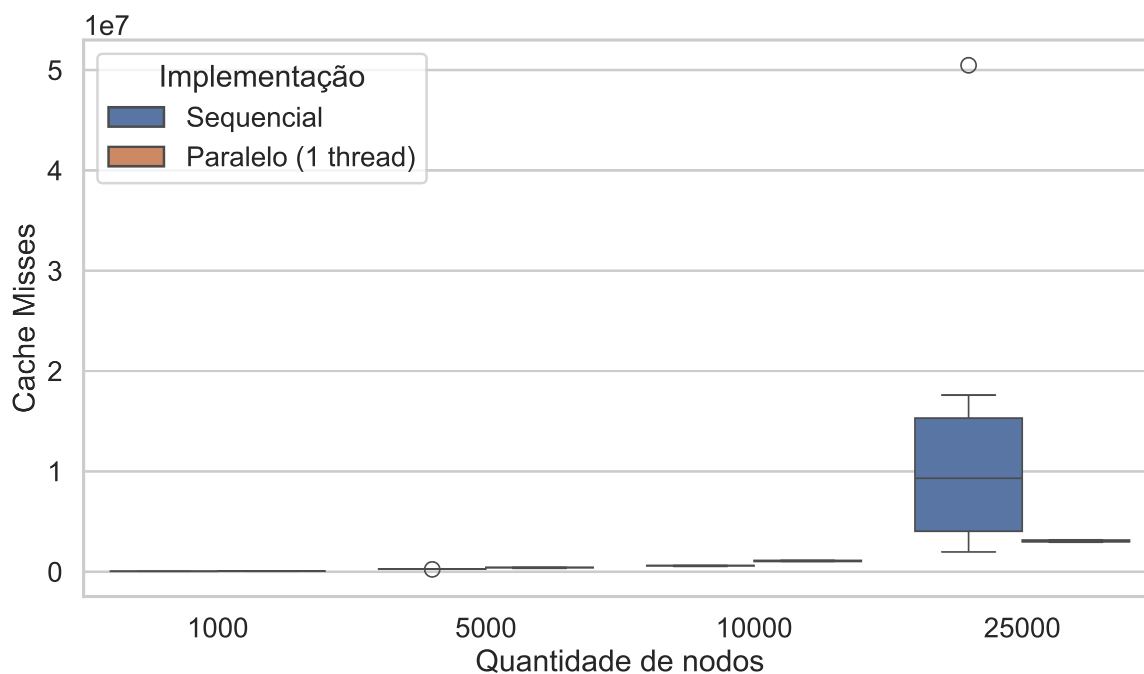
Ao se comparar as *branches*, observou-se um comportamento diferenciado entre as versões analisadas. O gráfico da Figura 15 demonstra que a implementação sequencial apresentou uma quantidade menor de *branch misses* quando comparada à implementação paralela executada em 1 *thread*. Apesar disso, a versão paralela em 1 *thread* demonstrou menor variabilidade nesse indicador, sugerindo um fluxo de execução mais consistente entre diferentes execuções. Já no gráfico da Figura 16, observa-se um comportamento inverso, de modo que a versão paralela apresenta desempenho inferior nas execuções com 1.000 nodos, exibindo maior número de

Figura 12 – Distribuição de Quantidade de ciclos por implementação sequencial em Rust



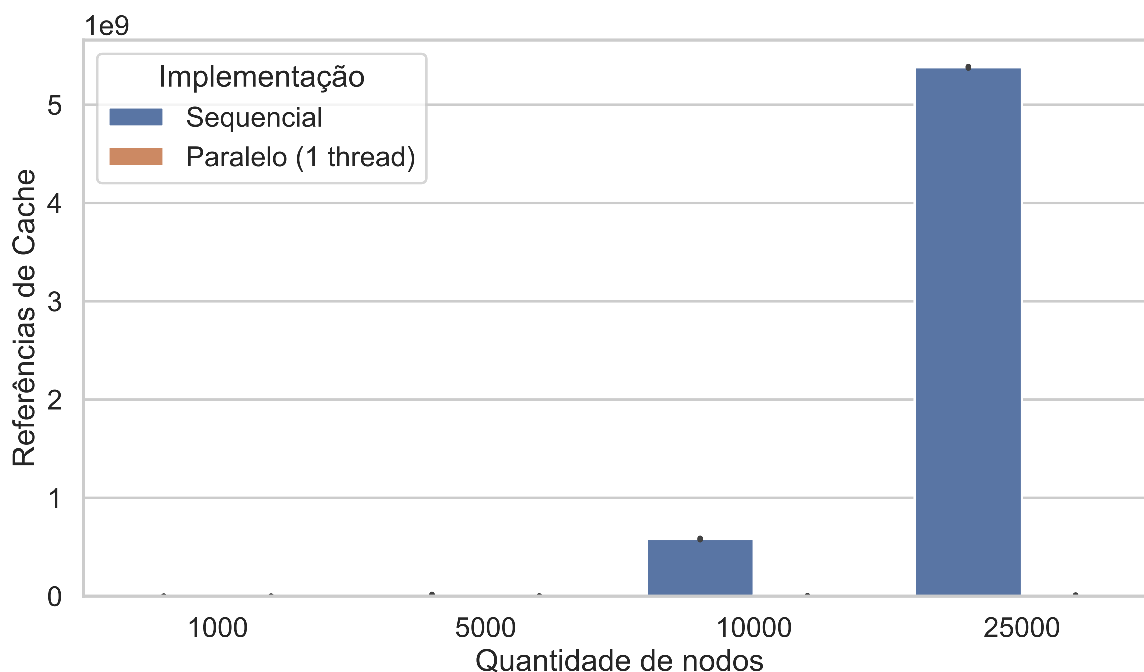
Fonte: o autor

Figura 13 – Distribuição de *Cache Misses* por implementação



Fonte: o autor

Figura 14 – Distribuição de Referências de *Cache* por implementação

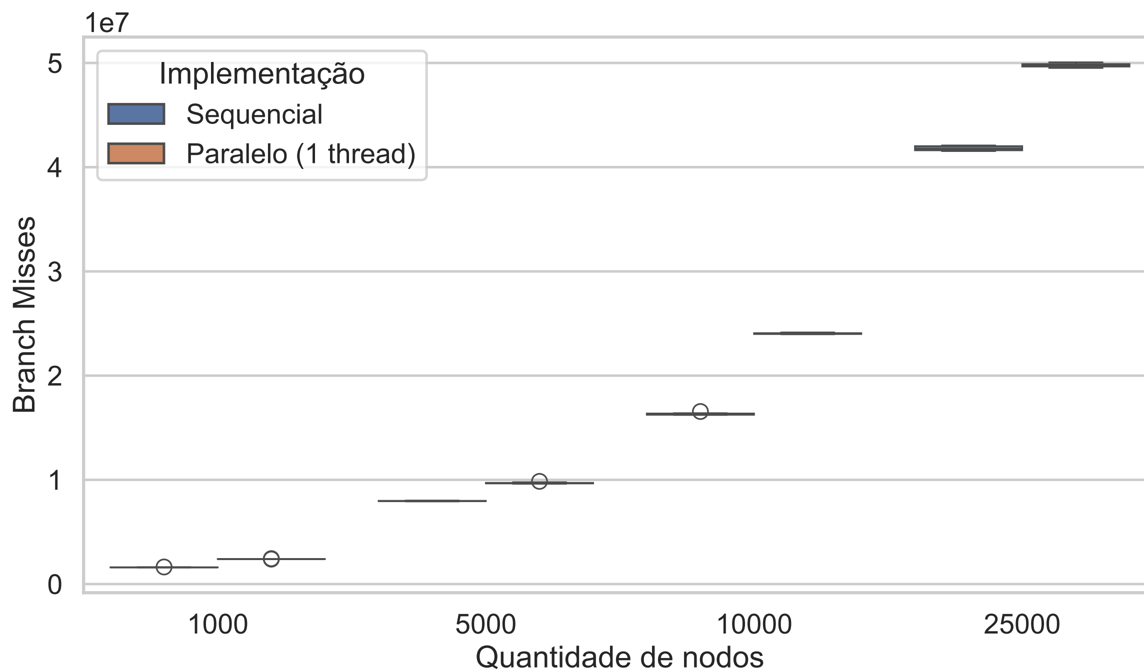


Fonte: o autor

branches do que a versão sequencial no entanto, à medida que o tamanho da malha cresce, a eficiência estrutural da refatoração se torna evidente. Para o caso de 25.000 nodos, a versão paralela em 1 *thread* executa menos de um terço do número total de *branches* registrados na versão sequencial, indicando que o código reorganizado reduz significativamente o número de caminhos condicionais e toma decisões mais previsíveis em cenários de maior carga computacional. Esses resultados reforçam que o ganho estrutural introduzido pela refatoração se manifesta principalmente em problemas de maior escala, quando a complexidade do fluxo interno se torna mais crítica para o desempenho.

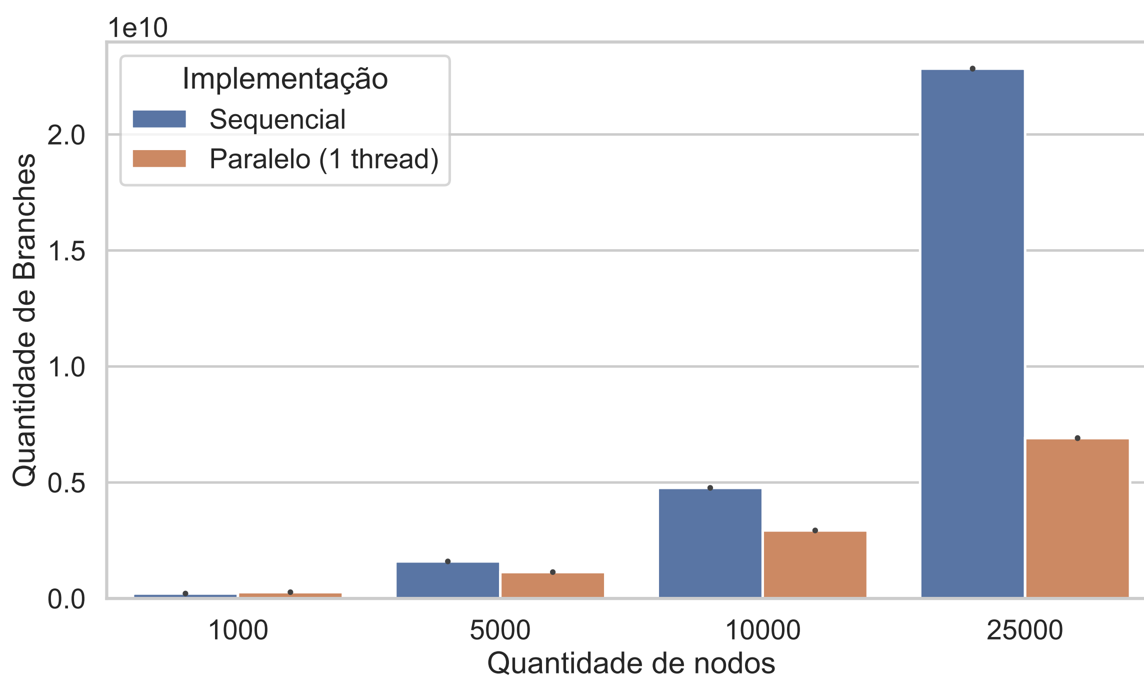
Quanto às *page faults*, o gráfico da Figura 17 revela que a versão sequencial, embora apresente pior desempenho em tempo total, ciclos e eficiência de *cache*, registra pouco menos da metade da quantidade de *page faults* quando comparada à implementação paralela executada em 1 *thread*. Esse resultado sugere que, embora a refatoração tenha melhorado a localidade de referência no nível da *cache*, ela pode ter introduzido padrões de acesso à memória que exigem mais interações com a memória virtual, possivelmente pela reorganização de estruturas internas ou maior quantidade de alocações temporárias. Ainda assim, é importante observar que o impacto desse aumento de *page faults* não se traduz em pior desempenho global, já que o custo agregado desses acessos permanece inferior às melhorias obtidas nos demais níveis da hierarquia de memória.

Figura 15 – Distribuição de *Branch Misses* por implementação



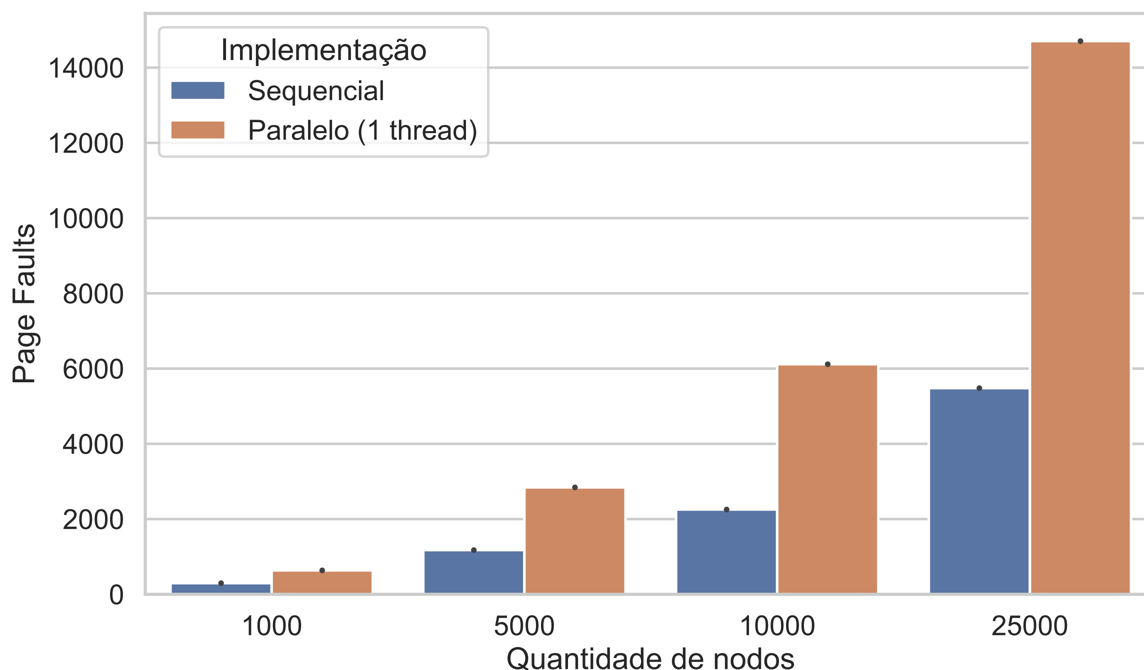
Fonte: o autor

Figura 16 – Distribuição da Quantidade de *Branches* por implementação



Fonte: o autor

Figura 17 – Distribuição de faltas de página por implementação



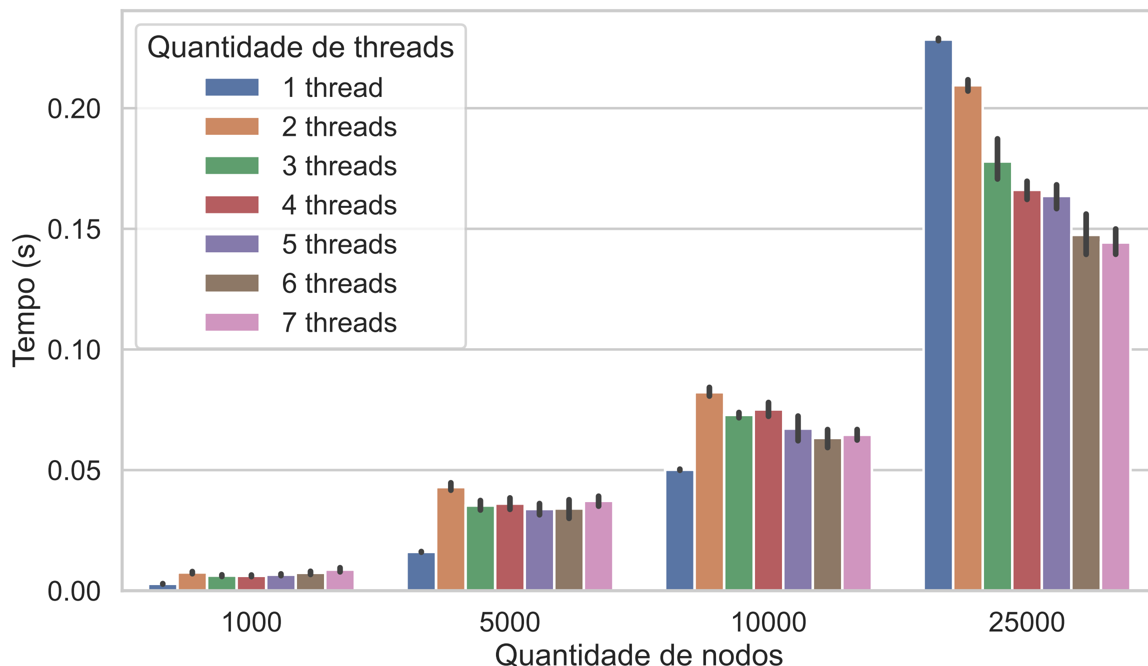
Fonte: o autor

5.3 COMPARAÇÃO DA IMPLEMENTAÇÃO PARALELA EM RUST POR NÚMERO DE *THREADS*

Ao comparar a implementação paralela por número de *threads*, observa-se que o desempenho do tempo de execução varia conforme o tamanho da malha e o número de *threads* utilizados. Conforme observado no gráfico da Figura 18, para os arquivos menores a execução em 1 *thread* apresenta desempenho superior às demais configurações, enquanto a execução em 2 *threads* apresenta, de forma consistente, o pior desempenho desses cenários. Esse comportamento sugere que, para cargas reduzidas, a sobrecarga de criação e coordenação de *threads* supera os benefícios do paralelismo. Entretanto, a partir de 25.000 nodos, a utilização de múltiplas *threads* passa a oferecer ganho significativo, com redução expressiva do tempo total à medida que mais threads são empregadas, estabilizando-se em um platô a partir de 6 *threads*, quando novos incrementos deixam de trazer benefícios substanciais. Esse padrão é coerente com o gráfico da Figura 19, que demonstra ganho progressivo até seis *threads* seguidas por saturação. Em contraste, o gráfico da Figura 20 revela comportamento oposto, onde o número total de ciclos aumenta com o número de *threads* em todos os experimentos, indicando que o paralelismo aumenta o custo computacional bruto, mas compensa esse aumento ao reduzir o tempo de execução, especialmente nos problemas maiores.

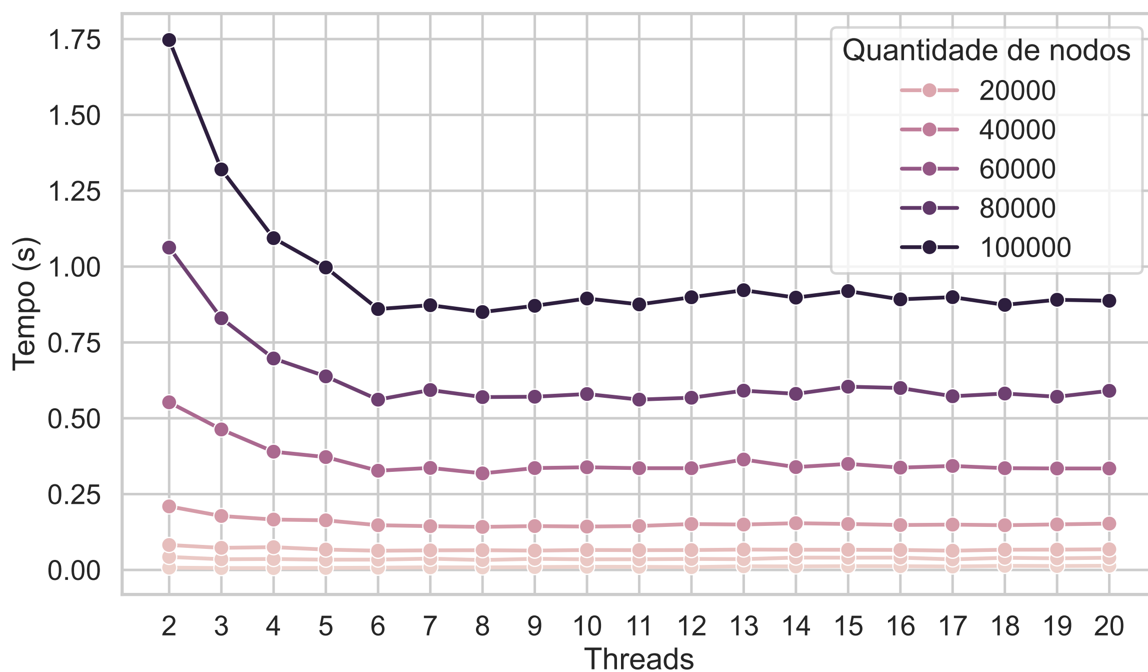
Quanto ao uso de *cache*, os resultados observados no gráfico da Figura 21

Figura 18 – Distribuição do tempo de execução por número de *threads*



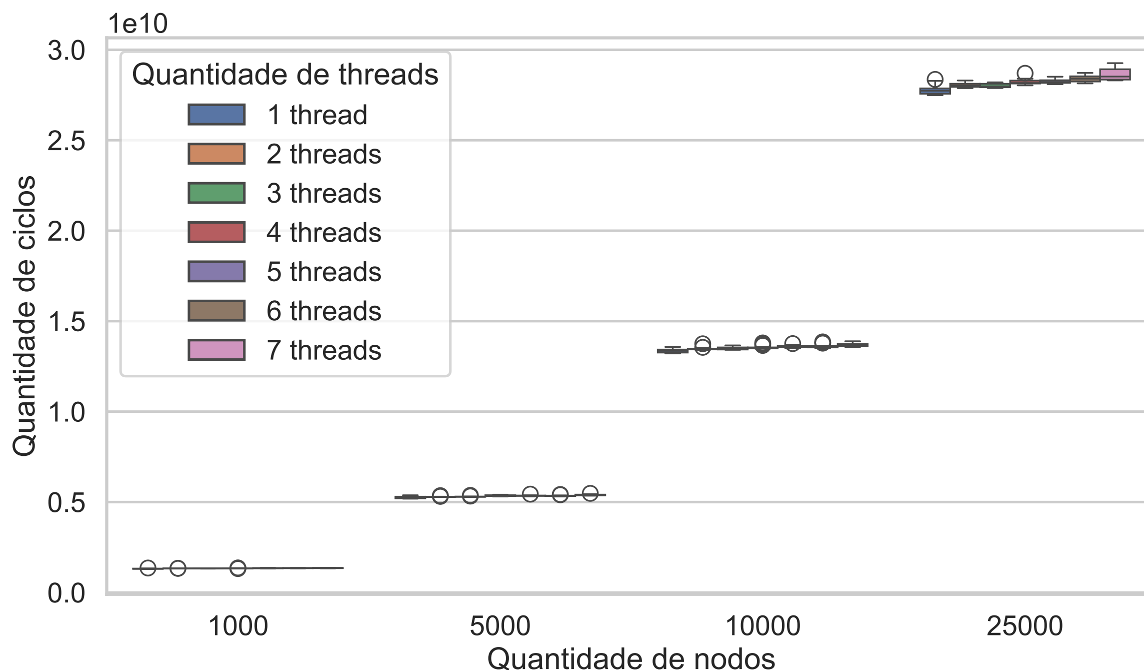
Fonte: o autor

Figura 19 – Tempo médio de execução das funções por número de *threads*



Fonte: o autor

Figura 20 – Distribuição de Quantidade de ciclos por número de *threads*

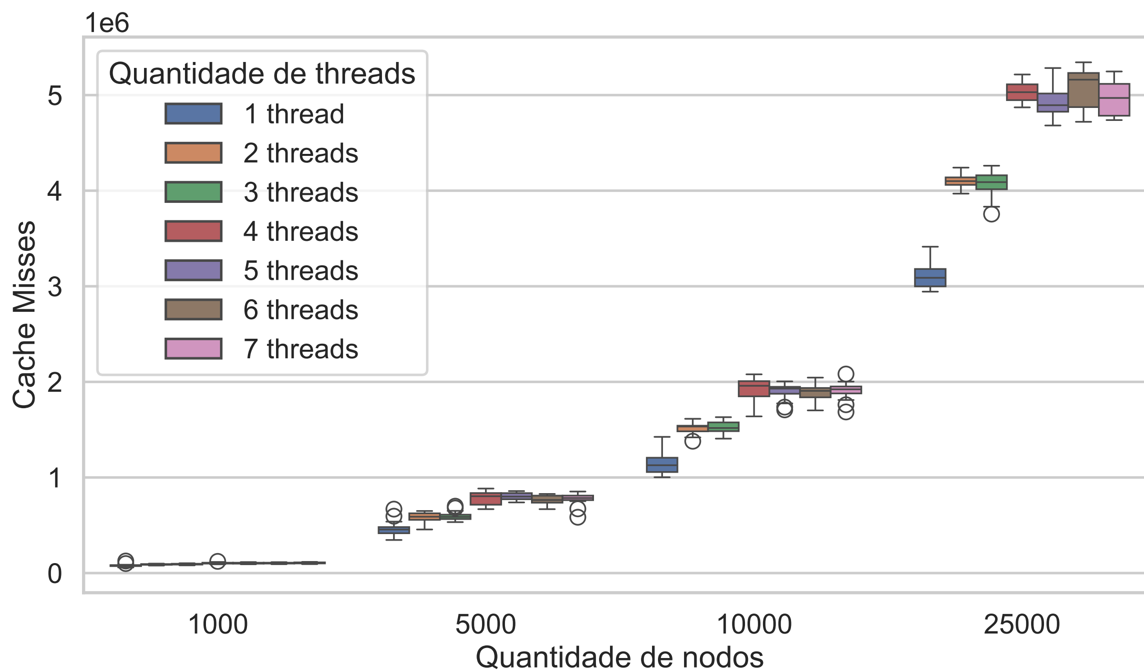


Fonte: o autor

mostram uma distinção clara entre as execuções com diferentes números de *threads*. As execuções em 1 *thread* apresentam o menor número de *cache misses*, refletindo o melhor aproveitamento da localidade de dados e o menor grau de interferência entre fluxos de execução. Em seguida, forma-se um agrupamento consistente dos resultados para as execuções com 2 e 3 *threads*, que exibem maior incidência de *misses*, porém com menor variabilidade do que as execuções com maiores quantidades de *threads*. O terceiro grupo, formado pelas execuções com 4 ou mais *threads*, apresenta o pior desempenho em termos de *misses*, refletindo a crescente competição por linhas de *cache* e maior pressão sobre a hierarquia de memória. Já quanto aos dados apresentados no gráfico da Figura 22, observa-se uma tendência crescente conforme o número de *threads* aumenta, indicando que o paralelismo intensifica o volume total de acessos ao *cache*, tanto pelo maior número de fluxos simultâneos quanto pelo aumento de interferências e invalidações provocadas pela execução paralela. Esse comportamento explica parte do custo adicional observado no gráfico da Figura 20 nas execuções com mais threads.

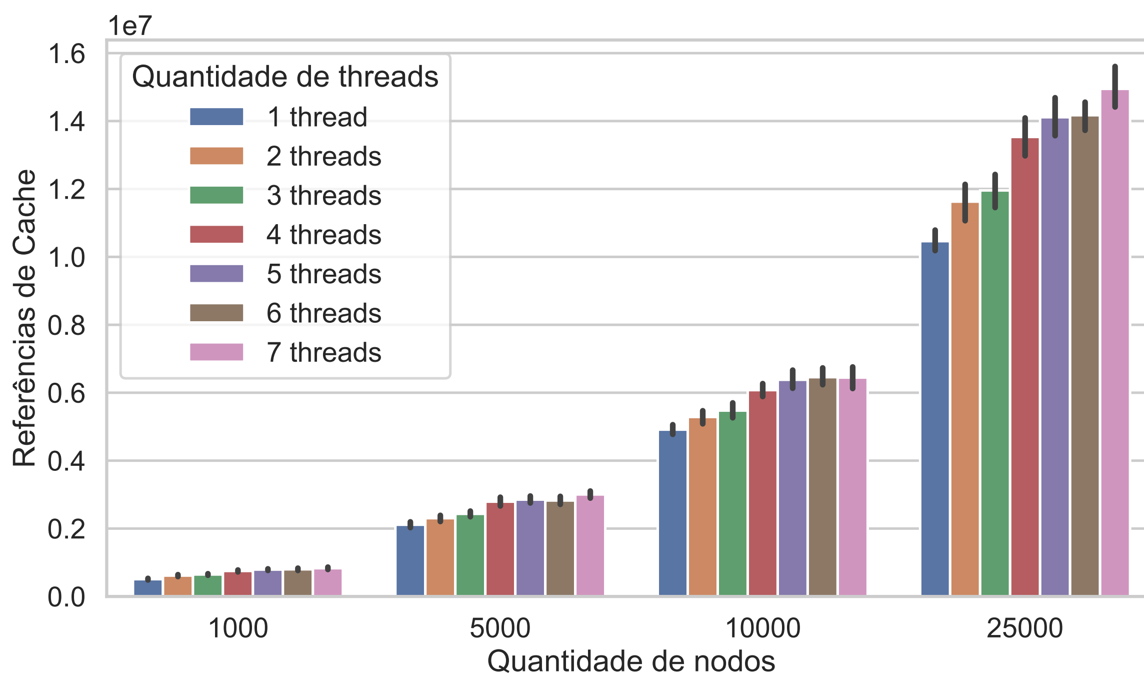
Ao comparar os dados referentes às *branches*, os gráficos das Figura 23 e Figura 24 mostram que, diferentemente das demais métricas analisadas, esses indicadores apresentam valores muito próximos entre todos os números de *threads* estudados. A ausência de variações significativas sugere que o paralelismo não altera de maneira relevante o padrão de fluxo de controle do algoritmo, uma vez que a lógica condicional permanece essencialmente a mesma independentemente do número de *threads*. Esse

Figura 21 – Distribuição de *Cache Misses* por número de *threads*



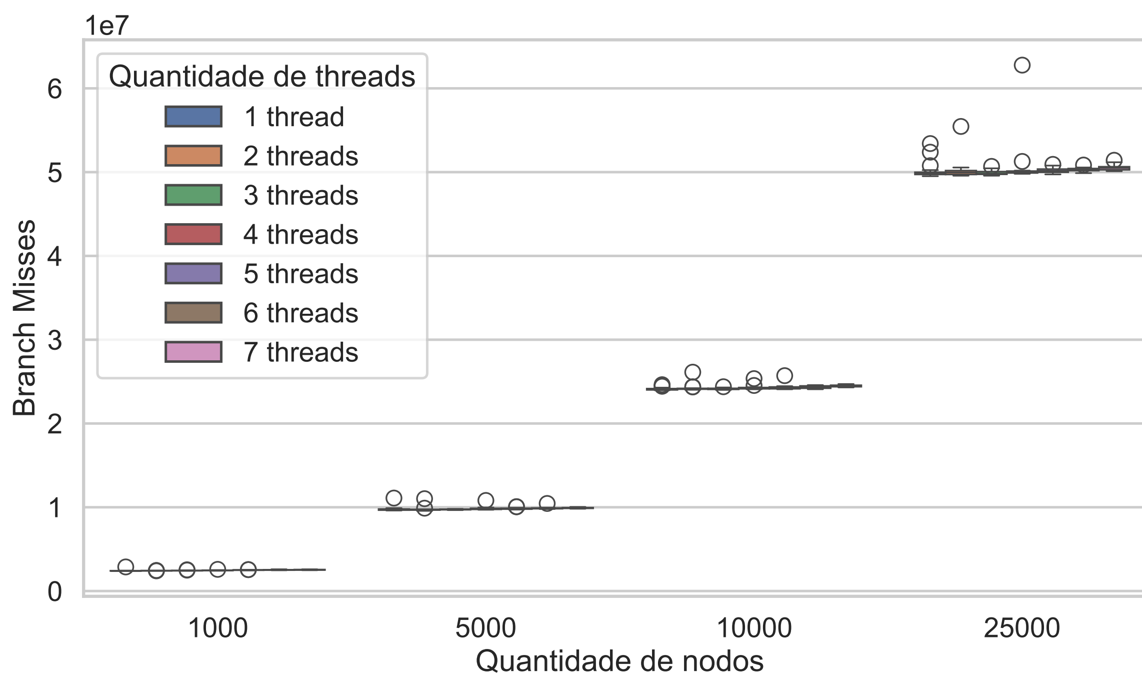
Fonte: o autor

Figura 22 – Distribuição de Referências de *Cache* por número de *threads*



Fonte: o autor

Figura 23 – Distribuição de *Branch Misses* por número de *threads*

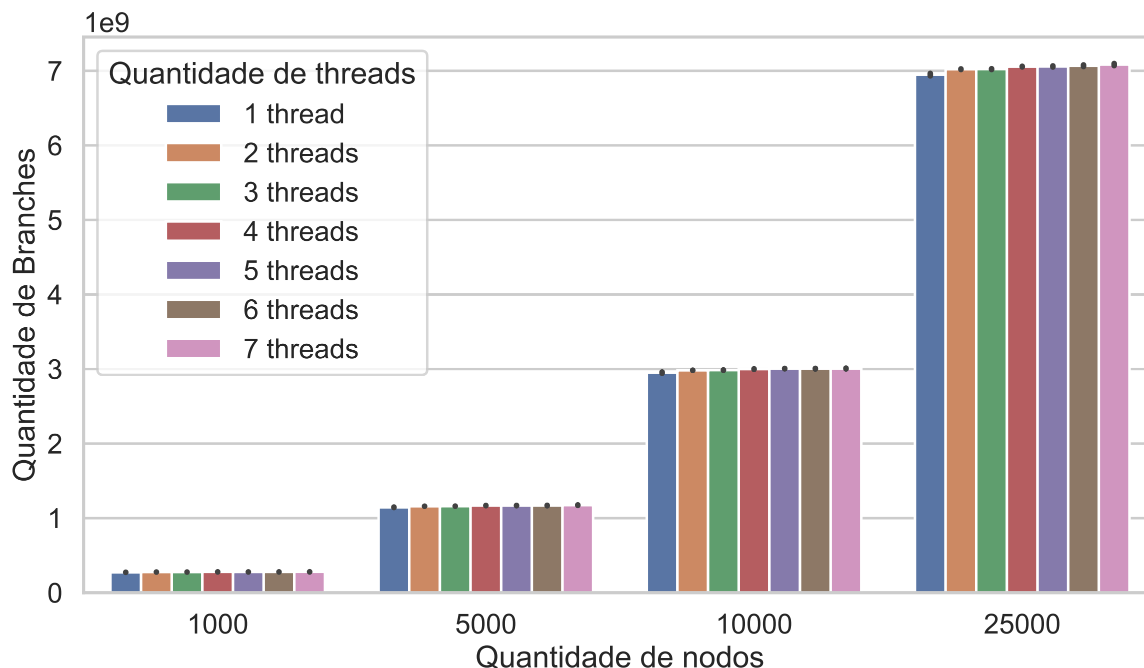


Fonte: o autor

comportamento está associado ao fato de que a refatoração do algoritmo foi responsável por reorganizar o fluxo de forma mais eficiente, mas a distribuição do processamento entre as *threads* não introduz novos caminhos condicionais nem altera a densidade relativa de desvios no código. Assim, o paralelismo afeta principalmente métricas ligadas à memória e à coordenação entre *threads*, enquanto o comportamento de *branches* permanece estável. A uniformidade dos valores de *branch misses* entre diferentes configurações reforça a ideia de que a previsibilidade do fluxo interno não é afetada pelo paralelismo em si.

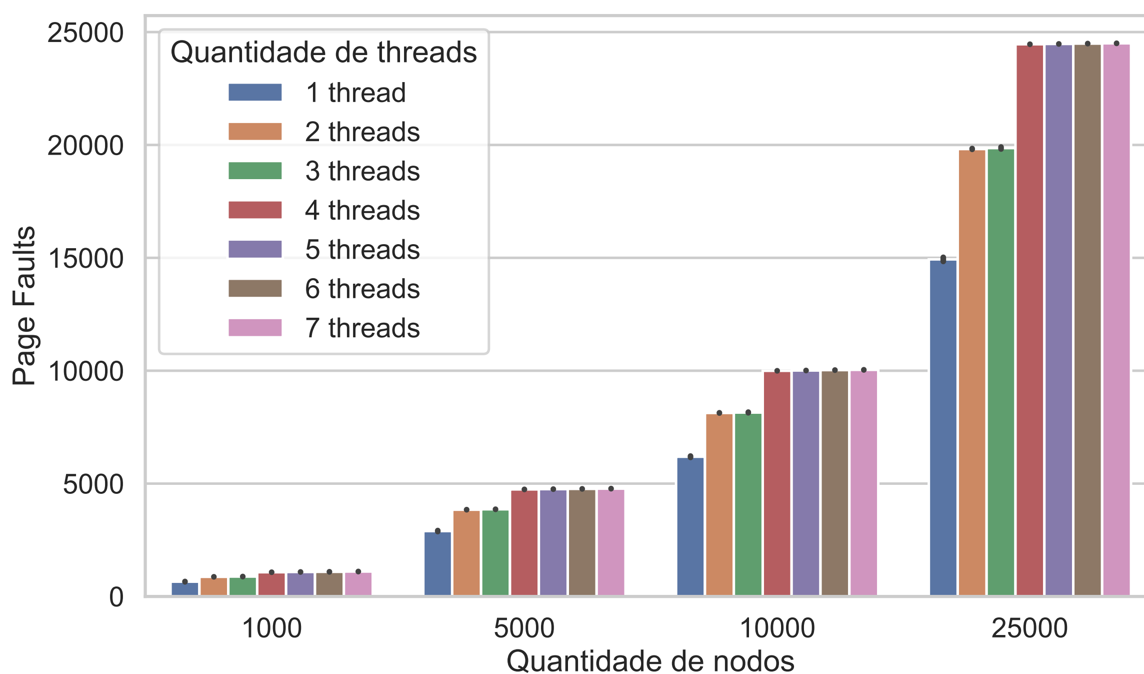
Quanto às *page faults*, o gráfico da Figura 25 apresentou comportamento semelhante ao gráfico da Figura 21, evidenciando três patamares distintos. As execuções em 1 *thread* apresentam o melhor desempenho, com o menor número de *page faults* entre todas as configurações testadas. Em seguida, há um agrupamento das execuções em 2 e 3 *threads*, que exibem números intermediários de *page faults*. O terceiro degrau é composto pelas configurações com 4 ou mais *threads*, que apresentam os maiores valores dessa métrica. Esse comportamento sugere que o aumento no número de *threads* intensifica o consumo de memória virtual, seja por maior dispersão dos acessos, seja por aumento do volume de dados ativos por unidade de tempo, ampliando a probabilidade de acessos que resultam em *page faults*. Assim como nos resultados referentes ao *cache*, os dados mostram que o paralelismo melhora o tempo final de execução para problemas grandes, mas ao custo de maior pressão sobre a memória, refletindo o *trade-off* clássico entre paralelismo e eficiência da hierarquia de memória.

Figura 24 – Distribuição da Quantidade de *Branches* por número de *threads*



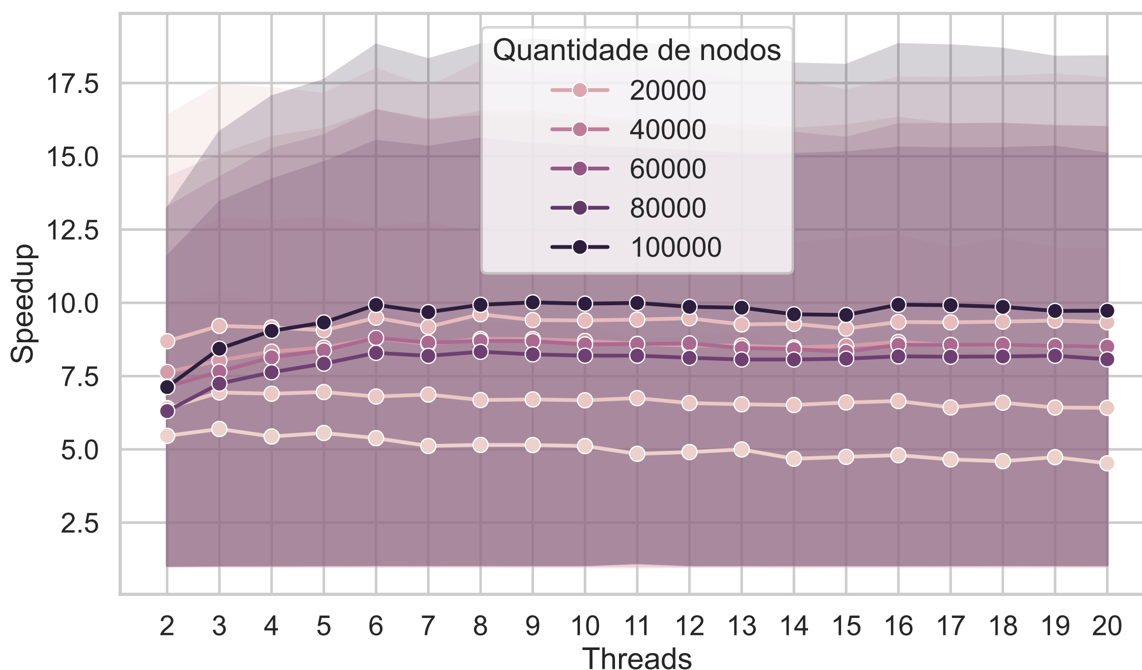
Fonte: o autor

Figura 25 – Distribuição de *Page Faults* por número de *threads*



Fonte: o autor

Figura 26 – *Speedup* em função do número de *threads*

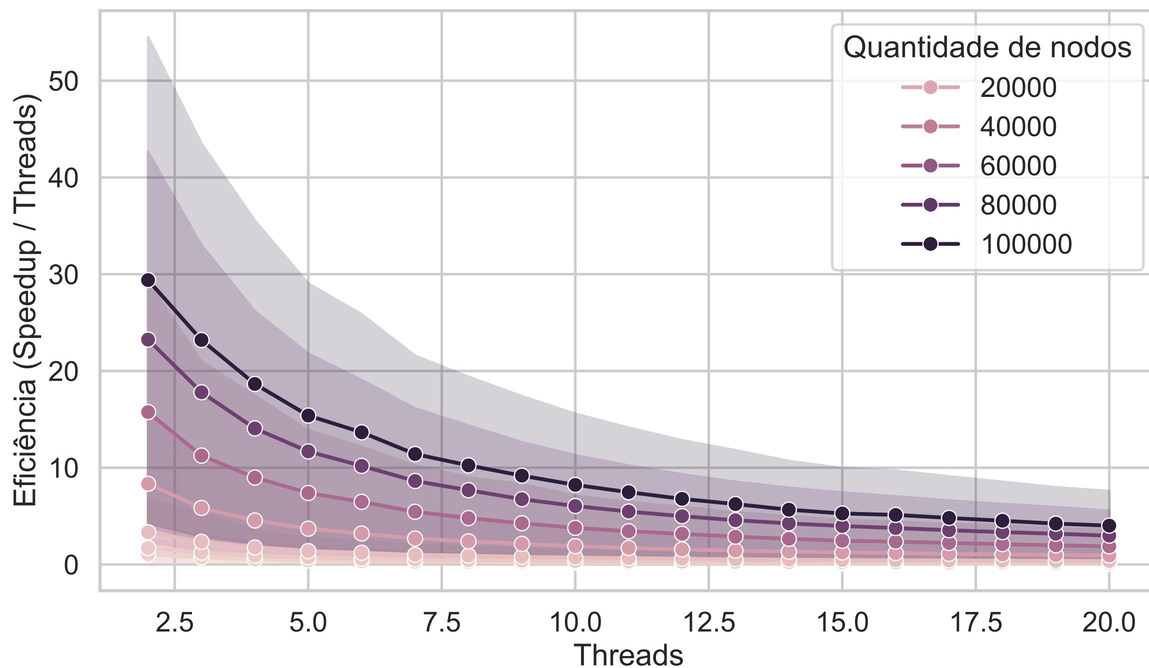


Fonte: o autor

Com base nos tempos de processamento da versão paralela em 1 *thread* e as demais execuções, foi possível calcular o *speedup*, apresentado no gráfico da Figura 26, e a eficiência paralela, ilustrada no gráfico da Figura 27. A partir desses resultados, observa-se um ganho de *speedup* até aproximadamente seis *threads*, a partir do qual o desempenho tende à estabilidade. Esse comportamento pode indicar um limite de paralelização inerente ao problema ou uma questão de granularidade dos dados, em que o custo de gerenciamento de múltiplas *threads* passa a superar os benefícios do paralelismo, justificando a redução da eficiência à medida que o número de *threads* aumenta.

Tais resultados demonstram que, para o conjunto de dados utilizado nos experimentos, a redução significativa no tempo de processamento é proveniente principalmente da refatoração da função ConstructDirectedEdges, que emprega estratégias voltadas à minimização da incidência de *cache misses* e à otimização do processo de busca.

Figura 27 – Eficiência paralela em função do número de *threads*



Fonte: o autor

6 CONCLUSÕES

O presente trabalho teve como objetivo avaliar o potencial da linguagem de programação Rust no desenvolvimento de aplicações HCP, por meio da reimplementação e análise comparativa do algoritmo ConstructTriangulationXYZ do modelo hidrodinâmico MOHID, originalmente desenvolvido em Fortran. Foram desenvolvidas e avaliadas versões sequenciais e paralelas do algoritmo, abrangendo diferentes tamanhos de entrada, com a coleta de métricas detalhadas de desempenho em nível de sistema operacional e de código-fonte.

Os resultados obtidos demonstraram que, na execução sequencial, a implementação em Rust apresentou desempenho superior ao Fortran, com tempos de processamento cerca de 20% menores para o conjunto de dados considerado. Essa diferença foi corroborada por métricas como o menor número de ciclos de CPU e menor incidência de *cache misses*, indicando uma melhor utilização da hierarquia de memória e uma possível otimização mais eficiente do código compilado. Também se observou uma redução significativa no número total de ramificações (*branches*) em Rust, o que sugere um fluxo de execução mais linear e otimizado. Contudo, essa reorganização pode ter aumentado a imprevisibilidade das ramificações, resultando em uma leve elevação na taxa de *branch misses* quando comparada à implementação em Fortran. Também observou-se que Rust apresentou maior número de *page faults* em comparação ao Fortran, o que pode estar relacionado à forma como o sistema de alocação de memória da linguagem distribui os dados em memória virtual. No entanto, o impacto desses comportamentos mostrou-se pouco significativo no tempo total de execução, uma vez que as demais métricas de desempenho mantiveram-se consistentemente favoráveis à implementação em Rust.

Ao se comparar as versões sequencial e paralela executada em 1 *thread*, verificou-se que os principais ganhos de desempenho não estão vinculados ao paralelismo, mas sim às refatorações realizadas. A versão paralela em 1 *thread* apresentou tempos de execução menores, menor quantidade total de ciclos e maior regularidade nas métricas de execução, indicando que a reorganização interna do código, a substituição de estruturas de busca sequenciais por abordagens mais eficientes e a melhoria da localidade de memória foram determinantes para o desempenho superior. Embora a versão paralela em 1 *thread* tenha apresentado maior número de *page faults* e referências de *cache*, esse comportamento reflete o maior volume de dados efetivamente acessado pela nova estrutura, sem resultar em penalidades significativas no tempo final. Assim, a análise mostra que a refatoração do algoritmo foi responsável por uma parcela crítica dos ganhos de desempenho, independentemente da utilização de múltiplas *threads*.

A análise da versão paralela sob diferentes quantidades de *threads* indicou esca-

labilidade do algoritmo até aproximadamente seis *threads*, faixa na qual foram observadas reduções no tempo médio de execução e aumentos consistentes de *speedup*. Para tamanhos de entrada menores, o *overhead* inerente à criação e sincronização de *threads* tornou execuções com duas ou três *threads* menos eficientes que a versão em 1 *thread*, mas à medida que o volume de dados aumentou, as execuções com maior paralelismo passaram a apresentar desempenho significativamente melhor. As métricas de hardware refletiram esse comportamento, de modo que houve um aumento nas referências de *cache* e na quantidade total de ciclos com o crescimento do número de *threads*, ao passo que *cache misses* e *page faults* formaram grupos claros de desempenho (1 thread como melhor cenário, 2-3 *threads* como intermediário e a prtir de 4 *threads* como cenário de maior pressão sobre a memória). Apesar disso, a estabilidade das quantidades de *branches* e *branch misses* mostraram que o paralelismo não alterou substancialmente o fluxo lógico do algoritmo. De forma geral, os resultados confirmam que o algoritmo se beneficia do paralelismo até certo limite, superando o desempenho sequencial para grandes volumes de dados, mas apresentando restrições típicas de aplicações com dependências estruturais e fração serial significativa.

De modo geral, os resultados obtidos indicam que Rust é uma linguagem promissora para o desenvolvimento de aplicações HCP, oferecendo excelente relação entre eficiência, segurança e expressividade. A combinação entre controle de baixo nível, ausência de coletor de lixo e mecanismos de segurança em tempo de compilação mostrou-se capaz de produzir códigos de alto desempenho, comparáveis, e em alguns casos superiores, aos escritos em Fortran.

A linguagem Rust apresenta interoperabilidade nativa com C, e embora essa capacidade tenha sido incorporada ao Fortran apenas a partir do padrão 2003, as versões mais recentes do compilador gfortran também oferecem suporte a essa funcionalidade. Diante disso, como perspectiva futura, recomenda-se a investigação da viabilidade de intercomunicação entre códigos Rust e Fortran por meio da linguagem C como intermediário, possibilitando a integração do código mais otimizado em fluxos de trabalho reais. Além disso, sugere-se a ampliação da análise para outros módulos do software MOHID e para problemas de maior complexidade computacional, bem como a exploração de estratégias de paralelismo híbrido (CPU-GPU) e otimizações voltadas à arquitetura NUMA. Tais investigações poderão aprofundar a compreensão sobre o comportamento da linguagem Rust em cenários de computação científica de larga escala e contribuir para sua consolidação como uma alternativa moderna, eficiente e segura no ecossistema de software científico.

REFERÊNCIAS

BARTZ-BEIELSTEIN, Thomas *et al.* Benchmarking in Optimization: Best Practice and Open Issues, 2020. Acessado em: 09 nov. 2025. DOI: https://doi.org/10.48550/arXiv.2007.03488. Disponível em: <https://arxiv.org/abs/2007.03488>. Citado nas pp. 13, 14.

BLANDY, Jim; ORENDORFF, Jason; TINDALL, Leonora F. S. **Programação em Rust: Desenvolvimento de Sistemas Rápidos e Seguros**. [*S.l.*]: Novatec Editora, 2023. Citado nas pp. 9, 10, 16.

CHAMBEL-LEITÃO, Pedro *et al.* Integration of MOHID Model and Tools with SWAT Model, 2007. Acessado em: 10 nov. 2025. Disponível em: <https://www.researchgate.net/publication/242086241_Integration_of_MOHID_Model_and_Tools_with_SWAT_Model>. Citado nas pp. 17, 18.

CLABURN, Thomas. **Rust developers at Google are twice as productive as C++ teams**. [*S.l.*: *s.n.*], 2024. https://www.theregister.com/2024/03/31/rust$_{google_c}/$. Acessado em: 10 nov. 2025. Citado na p. 7.

COSTANZO, M. *et al.* Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. **2021 XLVII Latin American Computing Conference (CLEI)**, 2021. DOI: 10.1109/CLEI53233.2021.9640225. Citado na p. 7.

COURIOL, Bruno. **Microsoft CTO Details Successes, Challenges, and Commitment to Rust at Rust Nation UK**. [*S.l.*: *s.n.*], 2025. https://www.infoq.com/news/2025/05/microsoft-cto-rust-commitment/. Acessado em: 10 nov. 2025. Citado na p. 7.

ENCYCLOPAEDIA BRITANNICA, The Editors of. **FORTRAN**. [*S.l.*: *s.n.*], 2025. https://www.britannica.com/technology/FORTRAN. Acessado em: 28 out. 2025. Citado na p. 11.

FORTRAN-LANG COMMUNITY. **Fortran**: High-performance parallel programming language. [*S.l.*: *s.n.*], 2025. https://fortran-lang.org. Acessado em: 22 out. 2025. Citado na p. 12.

FOSTER, Matt. **From C to Rust: inside Meta's Developer-Led Messaging Migration**. [*S.l.*: *s.n.*], 2025. https://www.infoq.com/news/2025/07/meta-rust-dx/. Acessado em: 10 nov. 2025. Citado na p. 7.

GUEZ, Lionel. **Fortran course**. [*S.l.*: *s.n.*], 2025. https://web.lmd.jussieu.fr/ lguez/Fortran$_{l}anguage_{s}ite/$. Acessado em: 28 out. 2025. Citado nas pp. 6, 11.

HOSEINYFARAHABADY, M. R. *et al.* Dynamic Control of CPU Cap Allocations in Stream Processing and Data-Flow Platforms. **2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)**, 2019. Citado nas pp. 20, 21.

IBM. **Fortran**. [*S.l.*: *s.n.*], 2025. https://www.ibm.com/history/fortran. Acessado em: 28 out. 2025. Citado nas pp. 6, 11, 12.

IBM. **What is parallel computing?** [*S.l.*: *s.n.*], 2025. https://www.ibm.com/think/topics/parallel-computing. Acessado em: 02 nov. 2025. Citado na p. 15.

IONITA, Dorin Marian; MANOLE, Filip George; SLUSANSCHI, Emil Ioan. Efficient Parallel Simulations of Wireless Signal Wave Propagation. **2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)**, 2020. Citado nas pp. 7, 19, 20.

JACOBY, D. *et al.* Geospatial Computing Collaborations. **2019 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)**, 2019. DOI: `10.1109/PACRIM47961.2019.8985053`. Citado nas pp. 6, 15.

LINDEMANN, Jonas; DAHLBLOM, Ola. **Modern Fortran in Science and Technology**. [*S.l.*: *s.n.*], 2025. https://modern-fortran-in-science-and-technology.readthedocs.io/en/latest/. Acessado em: 22 out. 2025. Citado na p. 11.

LIU, Zixi *et al.* An Empirical Study of Rust-Specific Bugs in the rustc Compiler, 2025. Acessado em: 08 nov. 2025. Disponível em: <https://arxiv.org/html/2503.23985v1>. Citado na p. 10.

MARTINS, Eduardo M. *et al.* NPB-Rust: NAS Parallel Benchmarks in Rust, 2025. Acessado em: 08 nov. 2025. Disponível em: <https://arxiv.org/html/2502.15536v1>. Citado nas pp. 7, 17, 21, 22.

MICROSOFT IGNITE. **High-Performance Computing (HPC) Performance and Benchmarking Overview**. [*S.l.*: *s.n.*], 2025. https://learn.microsoft.com/en-us/azure/high-performance-computing/performance-benchmarking/overview?tabs=processperf. Acessado em: 28 out. 2025. Citado na p. 13.

MOHID WATER MODELLING SYSTEM TEAM. **MOHID - Water Modelling System**. [*S.l.*: *s.n.*], 2025. https://github.com/Mohid-Water-Modelling-System/Mohid. Acessado em: 10 nov. 2025. Citado na p. 6.

MOHID WATER MODELLING SYSTEM TEAM. **MOHID - Water Modelling System**. [*S.l.*: *s.n.*], 2025. https://www.mohid.com/. Acessado em: 10 nov. 2025. Citado nas pp. 6, 17.

MOHID WATER MODELLING SYSTEM TEAM. **MOHID Modelling System**. [*S.l.*: *s.n.*], 2025. https://mohid.wordpress.com/the-model/. Acessado em: 10 nov. 2025. Citado na p. 17.

MOHID WATER MODELLING SYSTEM TEAM. **The MOHID Community**. [*S.l.*: *s.n.*], 2025. https://www.mohid.com/pages/home/community.shtml. Acessado em: 10 nov. 2025. Citado na p. 18.

MOHID WATER MODELLING SYSTEM TEAM. **What is MOHID?** [*S.l.*: *s.n.*], 2025. https://www.mohid.com/pages/home/whatismohid.shtml. Acessado em: 10 nov. 2025. Citado nas pp. 6, 18.

MUSEU INFORMÀTICA. **The FORTRAN Programming Language**. [*S.l.*: *s.n.*], 2025. https://museo.inf.upv.es/en/fortran/. Acessado em: 28 out. 2025. Citado nas pp. 11, 12.

ONCD, Office of the National Cyber Director. **Press Release: Future Software Should Be Memory Safe**. [*S.l.*: *s.n.*], 2024. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf. Acessado em: 10 nov. 2025. Citado na p. 7.

PERFWIKI. **perf: Linux profiling with performance counters**. [*S.l.*: *s.n.*], 2025. https://perfwiki.github.io/main/. Acessado em: 22 out. 2025. Citado na p. 15.

PIEPER, Ricardo *et al.* High-level and efficient structured stream parallelism for rust on multi-cores. **Journal of Computer Languages**, v. 65, n. 101054, 2021. Acessado em: 06 out. 2025. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85112331163&doi=10.1016%2fj.cola.2021.101054&partnerID=40&md5=dbdeeb3f56b672c495bf836fc53fef6c>. Acesso em: 6 out. 2025. Citado nas pp. 9, 10, 15–17, 21.

RAMS DEVELOPMENT TEAM. **RAMS**. [*S.l.*: *s.n.*], 2025. https://github.com/RAMSmodel/RAMS. Acessado em: 10 nov. 2025. Citado na p. 6.

RUST PROJECT DEVELOPERS. **Rust Compiler Development Guide**. [*S.l.*: *s.n.*], 2025. https://rustc-dev-guide.rust-lang.org/. Acessado em: 08 nov. 2025. Disponível em: <https://rustc-dev-guide.rust-lang.org/>. Citado na p. 10.

SHI, Yuan. Reevaluating Amdahl's Law and Gustafson's Law, 1996. Acessado em: 09 nov. 2025. Disponível em: <https://www.researchgate.net/publication/228367369_Reevaluating_Amdahl's_law_and_Gustafson's_law>. Citado na p. 16.

STACKOVERFLOW. **2024 Developer Survey**. [*S.l.*: *s.n.*], 2024. https://survey.stackoverflow.co/2024/technology. Acessado em: 07 jul. 2025. Disponível em: <https://survey.stackoverflow.co/2024/technology>. Acesso em: 7 jun. 2025. Citado na p. 7.

STALLMAN, Richard M.; THE GCC DEVELOPER COMMUNITY. **Using the GNU Compiler Collection**. [*S.l.*: *s.n.*], 2003. https://gcc.gnu.org/onlinedocs/gcc.pdf. Acessado em: 08 nov. 2025. Citado na p. 29.

SYDOW, Stefan *et al.* Towards Profile-Guided Optimization for Safe and Efficient Parallel Stream Processing in Rust. **2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**, 2020. Citado na p. 19.

TAFT, Darryl K. **Rust Growing Fastest, But JavaScript Reigns Supreme**. [*S.l.*: *s.n.*], 2024. https://thenewstack.io/rust-growing-fastest-but-javascript-reigns-supreme/. Acessado em: 10 jun. 2025. Disponível em: <https://thenewstack.io/rust-growing-fastest-but-javascript-reigns-supreme/>. Acesso em: 10 jun. 2025. Citado na p. 7.

THE GFORTRAN TEAM. **Using GNU Fortran**. [*S.l.*: *s.n.*], 2025. https://gcc.gnu.org/onlinedocs/gfortran.pdf. Acessado em: 08 nov. 2025. Citado na p. 12.

WRF DEVELOPMENT TEAM. **Weather Research and Forecasting Model (WRF)**. [*S.l.*: *s.n.*], 2025. https://github.com/wrf-model/WRF. Acessado em: 10 nov. 2025. Citado na p. 6.

Apêndices

APÊNDICE A – AVALIAÇÃO DO POTENCIAL DO USO DA LINGUAGEM DE PROGRAMAÇÃO RUST NA REIMPLEMENTAÇÃO DO MÓDULO DE TRIANGULAÇÃO DO MOHID

Avaliação do Potencial do Uso da Linguagem de Programação Rust na Reimplementação do Módulo de Triangulação do MOHID

Daniel Roberto de Aguiar[1], **Eduardo Camilo Inacio**[1]

[1]Instituto de Informática – Universidade Federal de Santa Catarina (UFSC)
– Florianópolis – SC – Brazil

daniel.aguiar@grad.ufsc.br , eduardo.camilo@ufsc.br

Abstract. *This paper aims to evaluate the potential of the Rust programming language in the development of high-performance scientific applications, comparing it with the Fortran language, practices used in this context. To this end, the triangulation module of the hydrodynamic modeling software MOHID, originally implemented in Fortran, was developed. The algorithm responsible for constructing the three-dimensional triangulation was isolated from the original code and reimplemented in Rust in two versions: one sequential and the other parallel, with support for multiple threads. Experimental runs were performed in the Ubuntu Linux environment, using the perf tool to obtain performance details such as execution time, CPU cycles, cache misses, branch misses, and page failures. The results demonstrated that the Rust language showed better overall performance compared to Fortran, with shorter processing times and better cache utilization, although it presented a higher number of page failures. The parallel version in Rust achieved significant performance gains, with increased speed up to approximately six threads, followed by stabilization, and decreasing efficiency in parallel as the number of threads increased. It is concluded that the Rust language presents high potential for the development of high-performance scientific applications, combining efficiency, security, and memory control, establishing itself as a modern alternative to Fortran.*

Resumo. *O artigo tem como objetivo avaliar o potencial da linguagem de programação Rust no desenvolvimento de aplicações científicas de alto desempenho, comparando-a com a linguagem Fortran, tradicionalmente utilizada nesse contexto. Para isso, foi analisado o módulo de triangulação do software de modelagem hidrodinâmica MOHID, originalmente implementado em Fortran. O algoritmo responsável pela construção da triangulação tridimensional foi isolado do código original e reimplementado em Rust em duas versões: uma sequencial e outra paralela, com suporte a múltiplas threads. As execuções experimentais foram realizadas em ambiente Ubuntu Linux, com o uso da ferramenta perf para obtenção de métricas detalhadas de desempenho, como tempo de execução, ciclos de CPU, cache misses, branch misses e page faults. Os resultados demonstraram que a linguagem Rust apresentou melhor desempenho geral em relação ao Fortran, com menor tempo de processamento e melhor aproveitamento de cache, embora tenha exibido um número maior de page faults. A versão paralela em Rust obteve ganhos expressivos de desempenho a partir de 2 threads, com aumento do speedup até aproximadamente seis threads,*

seguido de estabilização, e eficiência paralela decrescente conforme o número de threads aumentava. Conclui-se que a linguagem Rust apresenta elevado potencial para o desenvolvimento de aplicações científicas de alto desempenho, combinando eficiência, segurança e controle de memória, consolidando-se como uma alternativa moderna ao Fortran.

1. Introdução

O avanço das tecnologias digitais e a crescente complexidade dos sistemas de modelagem numérica têm impulsionado a busca por linguagens de programação capazes de oferecer desempenho, segurança e paralelismo eficientes. Embora o Fortran permaneça amplamente utilizado em aplicações científicas devido à sua tradição no processamento numérico [IBM 2025], suas limitações estruturais e o peso de décadas de código legado dificultam sua evolução em direção a paradigmas modernos de desenvolvimento. No contexto da modelagem hidrodinâmica, sistemas como o MOHID, cujo núcleo foi escrito majoritariamente em Fortran [MOHID Water Modelling System Team 2025], enfrentam desafios relacionados à manutenção, modernização e integração com novas arquiteturas de hardware.

Nesse cenário, a linguagem Rust tem ganhado destaque por combinar alta performance, segurança de memória e mecanismos robustos de concorrência [Pieper et al. 2021]. Apesar de sua popularidade crescente, ainda são escassos os estudos que avaliam empiricamente seu desempenho em algoritmos científicos complexos, particularmente quando comparados a implementações maduras em Fortran. Este trabalho investiga o potencial de Rust para aplicações científicas de alto desempenho por meio de uma reimplementação do algoritmo ConstructTriangulationXYZ do MOHID e da análise detalhada de três componentes internos: ConstructNodes, ConstructTriangles e ConstructDirectedEdges.

2. A Linguagem de Programação Rust

Rust é uma linguagem de programação desenvolvida pela Mozilla Research que busca conciliar segurança de memória, alto desempenho e concorrência confiável, oferecendo abstrações de alto nível voltadas à produtividade [Pieper et al. 2021]. Sua principal inovação é o sistema de posse e movimento, que permite gerenciar recursos de forma automática e segura sem recorrer a coletores de lixo, aproximando-se do modelo RAII utilizado em C++ e garantindo que alocações sejam liberadas ao final de seu escopo [Pieper et al. 2021]. Esse mecanismo é sustentado por um sistema de tipos rigoroso, projetado para permitir representações eficientes de dados e promover verificações estáticas que evitam erros comuns de ponteiros, ao mesmo tempo em que habilita padrões seguros de programação concorrente [Blandy et al. 2023].

A linguagem foi concebida para reduzir a lacuna histórica entre linguagens orientadas à segurança, como Java, e linguagens orientadas ao controle e desempenho, como C e C++ [Pieper et al. 2021]. Para isso, Rust adota regras de *ownership* e *borrow checking*, que garantem a ausência de acessos inválidos à memória, condições de corrida e ciclos de referências. Conceitos como movimento sem cópia (*move semantics*), tempos de vida explícitos e uso extensivo de *traits* contribuem para um modelo expressivo e eficiente, permitindo ao desenvolvedor escrever código próximo ao *assembly* sem abrir mão de garantias formais [Blandy et al. 2023].

As garantias da linguagem são viabilizadas pelo compilador oficial, o rustc, implementado sobre o backend LLVM. Diferentemente de compiladores tradicionais, o rustc utiliza múltiplas representações intermediárias para lidar com os mecanismos exclusivos da linguagem. Após a análise sintática e semântica, a Árvore de Sintaxe Abstrata (AST) é convertida para a Representação Intermediária de Alto Nível (HIR), que facilita a inferência de tipos, resolução de traits e verificação de tempos de vida [Rust Project Developers 2025]. Em seguida, a HIR é transformada na Representação Intermediária de Nível Médio (MIR), uma forma de código mais próxima do fluxo de controle e crucial para a verificação de empréstimos e otimizações específicas de Rust [Liu et al. 2025]. Somente após essas etapas é gerada a LLVM IR, que passa por otimizações adicionais antes da emissão do código final.

Combinando um sistema de tipos expressivo, regras estáticas de segurança e um pipeline de compilação cuidadosamente projetado, Rust oferece um ambiente robusto para o desenvolvimento de software de alto desempenho, como sistemas embarcados, motores de jogo e aplicações científicas. Essas características tornam a linguagem uma candidata promissora para a modernização de códigos tradicionais escritos em C, C++ ou Fortran, especialmente em domínios que exigem eficiência e confiabilidade.

3. Metodologia

A metodologia adotada neste estudo foi estruturada em quatro etapas principais: (i) seleção e compreensão do algoritmo original do MOHID, (ii) reimplementação em Rust em diferentes versões, (iii) execução controlada dos experimentos com coleta de métricas, e (iv) análise comparativa dos resultados. O foco recaiu sobre o módulo de triangulação do MOHID, responsável pela construção da malha espacial a partir de um conjunto de coordenadas, composto pelas rotinas ConstructNodes, ConstructTriangles e ConstructDirectedEdges.

Na primeira etapa, o algoritmo foi analisado diretamente a partir dos arquivos Fortran, preservando sua lógica original para garantir a equivalência funcional das implementações comparadas. A partir dessa análise, foram produzidas duas versões em Rust: uma versão sequencial, que reproduz fielmente a estrutura computacional do código em Fortran, e uma versão paralela refatorada, na qual o código foi reorganizado para melhorar a localidade de referência, reduzir condicionais redundantes e explorar paralelismo via *multithreading*.

Na terceira etapa, foram realizados experimentos controlados utilizando diferentes tamanhos de entrada. Para a versão paralela, foram testadas execuções com 1 a 20 threads, permitindo observar o impacto da concorrência no desempenho. As medições incluíram métricas como tempo total de execução, ciclos de CPU, cache misses, referências de cache, branch misses, quantidade total de branches e page faults. Os resultados foram armazenados em arquivos tabulares e posteriormente processados para geração dos gráficos analíticos.

Por fim, a análise comparativa considerou três eixos principais: (i) comparação entre o código original em Fortran e a versão sequencial em Rust; (ii) comparação entre a versão sequencial em Rust e a versão paralela executada em 1 thread, com o objetivo de mensurar exclusivamente o impacto da refatoração do código; e (iii) análise detalhada do comportamento da versão paralela sob diferentes quantidades de threads, a fim de identifi-

car limites de escalabilidade e padrões de eficiência relacionados à arquitetura multicore. A interpretação dos resultados foi conduzida separadamente para cada grupo de métricas — tempo, ciclos, cache, branches e page faults — a fim de construir uma compreensão abrangente dos efeitos da refatoração e do paralelismo no desempenho do algoritmo.

4. Resultados

4.1. Comparação das implementações sequenciais

A comparação entre o código original em Fortran e a implementação sequencial equivalente em Rust revelou ganhos substanciais de desempenho, conforme observado no gráfico da figura 1. A versão sequencial em Rust, construída de forma a replicar exatamente a lógica do algoritmo Fortran, apresentou tempos de execução menores em todos os tamanhos de entrada avaliados. Esses ganhos também foram observados na redução da quantidade de ciclos de CPU e da latência média por operação, indicando que a organização interna e as otimizações do compilador rustc produziram um binário mais eficiente do que o compilador Fortran utilizado. No entanto, houve menor incidência de *page faults* e *branch misses* na versão Fortran, sugerindo melhor previsibilidade do fluxo de execução e melhor gerenciamento da memória virtual.

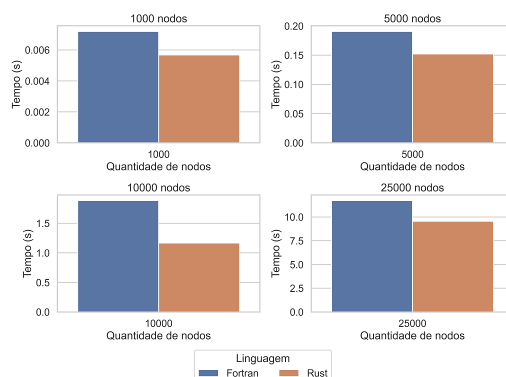


Figura 1. Tempo médio de execução do processamento de dados (escalas independentes)

4.2. Comparação entre a implementação sequencial em Rust e a versão paralela executada em 1 *thread*

Quando comparada à versão sequencial, a versão paralela executada em apenas 1 thread apresentou desempenho superior (Figura 2). As melhorias de refatoração resultaram em uma queda consistente do tempo de execução, bem como na redução da quantidade total de ciclos. Observou-se também maior estabilidade nas métricas de *branch misses*, ainda que o número absoluto tenha sido ligeiramente maior na versão paralela. Em contrapartida, essa versão apresentou maior número de *page faults* e *cache references*, indicando que a reestruturação interna do algoritmo, embora mais eficiente do ponto de vista computacional, ampliou o volume de dados manipulados, aumentando o tráfego entre CPU e memória. Por outro lado, esse comportamento foi acompanhado de uma leve redução nas *cache misses* relativas, sugerindo melhor aproveitamento da hierarquia de memória, ainda que com maior pressão sobre ela.

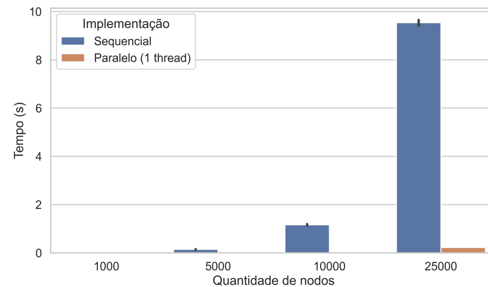


Figura 2. Distribuição do tempo de execução por implementação sequencial em Rust

4.3. Comparação da implementação paralela em Rust por número de *threads*

A análise da versão paralela em múltiplos níveis de concorrência revelou um comportamento claro de escalabilidade até aproximadamente seis *threads*, ponto em que foi observado um platô no tempo médio de execução, conforme observado no gráfico da figura 3. Para entradas menores (até 10.000 nodos), a execução com uma única *thread* foi superior às execuções com mais threads, resultado atribuído ao overhead de criação, sincronização e divisão de tarefas. Porém, para 25.000 nodos, o uso de múltiplas threads proporcionou aceleração significativa, com tempos de execução drasticamente menores nas configurações com maior paralelismo. Quanto às métricas de hardware, observou-se que o número de ciclos aumenta com o número de threads, um padrão típico em aplicações paralelas devido ao custo adicional de sincronização e ao aumento no consumo de recursos de CPU. Da mesma forma, houve aumento monotônico nas *cache references*, refletindo maior paralelismo de memória, enquanto as *cache misses* formaram três grupos: execuções em 1 thread apresentaram o melhor desempenho, execuções em 2–3 threads formaram um segundo grupo intermediário, e execuções com 4 ou mais threads exibiram maiores taxas de *misses*, compatíveis com maior contenção do *cache*. Resultados similares foram observados nos *page faults*, que também se distribuíram em três degraus de desempenho, enquanto o número de *branches* e *branch misses* manteve-se relativamente estável entre as configurações, indicando que o paralelismo não alterou substancialmente o fluxo de controle do algoritmo.

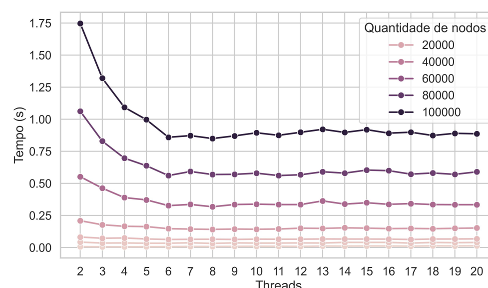


Figura 3. Tempo médio de execução das funções por número de *threads*

5. Conclusões

O estudo avaliou o potencial da linguagem Rust para aplicações de alto desempenho por meio da reimplementação do algoritmo ConstructTriangulationXYZ do sistema MOHID, originalmente escrito em Fortran. Foram desenvolvidas versões sequencial e paralela do algoritmo, e uma ampla bateria de experimentos foi conduzida com diferentes tamanhos de entrada, acompanhada da coleta de métricas detalhadas de desempenho em nível de hardware e execução.

Os resultados demonstraram que a implementação sequencial em Rust apresentou desempenho superior ao código original em Fortran com tempo de processamento cerca de 20% menor. Além da redução no tempo total de execução, observou-se também menor número de ciclos de CPU e menor incidência de *cache misses*, indicando melhor utilização da hierarquia de memória e maior eficiência das otimizações realizadas pelo compilador rustc. Embora Rust tenha apresentado maior número de *page faults* e uma leve elevação na taxa de *branch misses*, tais diferenças não comprometeram o desempenho geral, reforçando a vantagem da linguagem mesmo em execução puramente sequencial.

A comparação entre a versão sequencial e a versão paralela executada em 1 thread evidenciou o impacto da refatoração realizada. A reorganização estrutural do algoritmo reduziu substancialmente o tempo de execução, melhorou o fluxo de controle do código e diminuiu a variabilidade de desempenho. Isso indica que parcela significativa dos ganhos obtidos na versão paralela não está relacionada ao paralelismo em si, mas às melhorias introduzidas na estrutura de dados e na forma como o algoritmo explora a hierarquia de memória.

Por fim, a análise da versão paralela sob diferentes quantidades de *threads* revelou escalabilidade clara até aproximadamente seis *threads*, intervalo no qual foram observados os maiores speedups. A partir desse ponto, tanto o tempo de execução quanto a eficiência paralela passaram a se estabilizar, sugerindo a influência de fatores como fração serial do algoritmo, *overhead* de sincronização e contenção de *cache*. Mesmo assim, as execuções com maior número de *threads* continuaram apresentando desempenho superior para os maiores volumes de dados avaliados. Esses resultados reforçam que Rust é capaz de explorar eficientemente o paralelismo em múltiplos núcleos, mantendo segurança de memória e controle explícito dos recursos.

De maneira geral, os experimentos confirmam que Rust é uma linguagem promissora para aplicações científicas de alto desempenho, oferecendo uma combinação de eficiência, segurança e expressividade que a coloca como alternativa moderna aos códigos tradicionais em Fortran e C/C++. Como trabalhos futuros, recomenda-se investigar a interoperabilidade entre Rust e Fortran por meio de interfaces em C, avaliar o desempenho em outros módulos do MOHID e explorar estratégias híbridas de paralelismo, incluindo arquiteturas NUMA e aceleração por GPU, ampliando o entendimento sobre o papel da linguagem em aplicações científicas de larga escala.

Referências

Blandy, J., Orendorff, J., and Tindall, L. F. S. (2023). *Programação em Rust: Desenvolvimento de Sistemas Rápidos e Seguros*. Novatec Editora.

IBM (2025). Fortran. https://www.ibm.com/history/fortran. Acessado em: 28 out. 2025.

Liu, Z., Feng, Y., Ni, Y., Li, S., Yin, X., Shi, Q., Xu, B., and Su, Z. (2025). An empirical study of rust-specific bugs in the rustc compiler. Acessado em: 08 nov. 2025.

MOHID Water Modelling System Team (2025). Mohid - water modelling system. https://github.com/Mohid-Water-Modelling-System/Mohid. Acessado em: 10 nov. 2025.

Pieper, R., Löff, J., Hoffmann, R. B., Griebler, D., and Fernandes, L. G. (2021). High-level and efficient structured stream parallelism for rust on multi-cores. *Journal of Computer Languages*, 65(101054). Acessado em: 06 out. 2025.

Rust Project Developers (2025). Rust compiler development guide. https://rustc-dev-guide.rust-lang.org/. Acessado em: 08 nov. 2025.