Anais do Congresso Brasileiro de Iniciação Científica
(https://periodicos.unisanta.br/COB)

ISSN 2317-1316

SWORD: UMA LINGUAGEM DE PROGRAMAÇÃO COM PARALELISMO REPRODUTÍVEL VERIFICADO PELO COMPILADOR

SWORD: A PROGRAMMING LANGUAGE WITH COMPILER-CHECKED REPRODUCIBLE PARALLELISM

Pedro Tessaro Augusto, Alexandre dos Santos Mignon

Universidade Presbiteriana Mackenzie, Faculdade de Computação e Informática, Curso de Ciência da Computação
E-mail para contato: ptssar22@gmail.com

RESUMO – Somas paralelas em ponto flutuante costumam mudar de resultado com o número de threads e com a máquina. Este trabalho apresenta a Sword, uma linguagem de programação compilada em que o compilador detecta corridas de dados e verifica que o resultado de funções marcadas como `det` *depende apenas dos seus argumentos. A garantia foi avaliada contra C com OpenMP em cinco plataformas, com uma soma de dez milhões de valores, uma simulação de Monte Carlo e funções elementares. Em 40 execuções da soma, o C produziu 13 resultados diferentes e a Sword, um. A soma simples da Sword custou o mesmo que a do C; a soma exata, cerca de nove vezes mais nos dados mais difíceis.*

Palavras-chave: linguagens de programação; determinismo; ponto flutuante; concorrência; OpenMP.

ABSTRACT – Parallel floating-point sums often change their result with the number of threads and with the machine. This work presents Sword, a compiled programming language whose compiler detects data races and checks that the result of functions marked `det` *depends only on their arguments. The guarantee was evaluated against C with OpenMP on five platforms, using a* `sum` *of ten million values, a Monte Carlo simulation and elementary functions. Over 40 runs of the sum, C produced 13 different results and Sword produced one. Sword's plain sum cost the same as C's; its exact sum cost about nine times more on the hardest data.*

Keywords: programming languages; determinism; floating point; concurrency; OpenMP.

ISSN 2317-1316

1 INTRODUÇÃO

Em computação paralela, somar os mesmos números pode dar resultados diferentes. A soma em ponto flutuante não é associativa: cada operação arredonda, e arredondar em outra ordem produz outro valor (Goldberg, 1991). Em precisão dupla, (0,1 + 0,2) + 0,3 resulta em 0,6000000000000001, enquanto 0,1 + (0,2 + 0,3) resulta em 0,6. Quando um laço é dividido entre *threads*, cada uma soma uma parte e as partes são combinadas ao final. Mudar o número de *threads* muda a divisão em partes, e mudar de máquina pode mudar a ordem em que as partes são combinadas. Além disso, funções como seno, cosseno e exponencial vêm da biblioteca matemática de cada sistema, e bibliotecas diferentes podem devolver valores ligeiramente diferentes para o mesmo argumento. O resultado deixa de ser reprodutível, o que dificulta depurar um programa, comparar execuções em máquinas diferentes e escrever testes que verifiquem a saída (Collange *et al.*, 2015).

Este trabalho apresenta a Sword, uma linguagem de programação compilada, com gerenciamento explícito de memória, cujo compilador detecta corridas de dados antes de o programa ser executado (Tessaro, 2026a). Uma corrida de dados ocorre quando duas tarefas acessam a mesma memória, pelo menos uma delas escreve e nada impede que os dois acessos aconteçam ao mesmo tempo (Savage *et al.*, 1997). As mesmas regras que tornam essa verificação possível permitem ao compilador verificar outra propriedade: que o resultado de uma função paralela depende apenas dos seus argumentos, e não do número de *threads*, da ordem de execução ou da configuração da máquina.

As contribuições deste trabalho são:

1. a apresentação da linguagem, do seu modelo de concorrência e da garantia `det`, verificada pelo compilador;
2. a descrição das peças da biblioteca padrão que tornam a garantia útil para cálculo numérico;
3. uma avaliação experimental da garantia contra C com OpenMP, em cinco plataformas, medindo também o seu custo.

A Seção 2 discute trabalhos relacionados; a Seção 3 apresenta a linguagem; a Seção 4 descreve os experimentos; a Seção 5 apresenta e discute os resultados; e a Seção 6 apresenta a conclusão do trabalho.

ISSN 2317-1316

2 TRABALHOS RELACIONADOS

A Sword tem a sintaxe e o modelo de tarefas próximos aos do Go (Donovan e Kernighan, 2015) e o modelo de memória do Zig (Zig Software Foundation, 2026): não há coletor de lixo, e uma função só aloca memória se receber um alocador. O Go detecta corridas de dados em tempo de execução, quando o programa roda com o detector de corridas, e apenas aquelas que de fato ocorrem nesse teste. Na Sword, elas são detectadas em tempo de compilação, antes de o programa ser executado. O Rust (Matsakis e Klock, 2014) oferece garantias mais fortes que as da Sword, como a ausência de uso de memória já liberada, ao custo de um sistema de tipos mais elaborado; a Sword garante a ausência de corridas apenas dentro do escopo em que as tarefas são criadas.

Em C e C++, o paralelismo de dados costuma ser escrito com OpenMP (OpenMP Architecture Review Board, 2021) ou com a biblioteca oneTBB. A oneTBB oferece `parallel_deterministic_reduce`, que executa o mesmo conjunto de divisões e junções do intervalo qualquer que seja o número de *threads*, e por isso dá o mesmo resultado para a mesma entrada, desde que as *threads* arredondem igual (Intel Corporation, 2023; UXL Foundation, 2022). A promessa, porém, vale para aquela chamada: nada impede que o resto da função use um contador atômico ou leia o relógio, e nada avisa quando isso acontece.

O trabalho mais próximo da garantia de determinismo da Sword é o Deterministic Parallel Java (DPJ), que usa um sistema de tipos e efeitos para garantir determinismo em Java (Bocchino *et al.*, 2009). No DPJ, o programador divide o *heap* em regiões nomeadas e escreve, em cada método, um resumo dos efeitos, isto é, das regiões que ele lê e escreve; o compilador verifica, de forma modular, que os acessos feitos em paralelo comutam ou estão sincronizados. Na Sword não há regiões nem resumos de efeito: a verificação parte do escopo léxico das tarefas e das partições, e a análise de determinismo considera o programa inteiro, o que dispensa as anotações ao custo de não ser modular. Kahn (1974) mostrou que processos ligados por canais com um remetente e um receptor calculam o mesmo resultado em qualquer ordem de execução. Para a parte numérica, há algoritmos de soma reprodutível (Demmel e Nguyen, 2013), o acumulador exato de Kulisch (Kulisch, 2013) e geradores de números aleatórios baseados em contador (Salmon *et al.*, 2011).

Nenhuma dessas técnicas é nova isoladamente, e cada uma resolve uma parte do problema. A redução determinística da oneTBB fixa a ordem de uma chamada, mas não

ISSN 2317-1316

considera o resto da função; o DPJ verifica o determinismo, mas pede que o programador anote regiões e efeitos; a soma reprodutível, o acumulador exato e o Philox são bibliotecas que o programador precisa escolher e usar corretamente, sem que nada confira se usou. A Sword junta essas partes. Uma única palavra, `det`, pede ao compilador que verifique a função inteira, incluindo tudo o que ela chama, e a mensagem de erro aponta a linha que quebra a garantia, mesmo quando essa linha está em uma função de outro pacote. As peças numéricas fazem parte da biblioteca padrão e também são `det`, de modo que uma função verificada pode usá-las sem perder a garantia.

3 A LINGUAGEM SWORD

3.1 Objetivo e termos da linguagem

A Sword foi projetada para programas concorrentes, como servidores e cálculos paralelos, em que a memória é controlada pelo programador e o compilador detecta corridas de dados. Para que essa verificação seja possível, tudo o que uma tarefa pode alcançar fica visível no ponto em que ela é criada. Uma tarefa é criada com a palavra reservada `spawn`, que recebe uma única chamada de função, e só dentro de um bloco definido pela palavra reservada `scope`, que só termina quando todas as tarefas criadas nele terminam; assim, os dados que a função passa a uma tarefa continuam existindo enquanto a tarefa os utiliza. Variáveis são imutáveis, a menos que declaradas com a palavra reservada `mut`. Dentro de um `scope`, duas tarefas só podem acessar a mesma memória se ambas apenas lerem ou se cada uma receber um pedaço diferente de uma mesma divisão feita com `xs.chunks(n)`; qualquer outro caso é erro de compilação. Para compartilhar memória de propósito, há `atomic[T]`, que só permite operações atômicas (que nenhuma outra tarefa vê pela metade), e `shared[T]`, que guarda um valor acessível só sob um *lock*, um objeto de exclusão mútua que pertence a uma tarefa de cada vez (Herlihy *et al.*, 2020; Savage *et al.*, 1997). O laço `parallel for` divide as iterações entre os núcleos, e a cláusula `reduce` diz como os resultados parciais são combinados. Não há coletor de lixo: uma função que aloca memória recebe um `mem.Allocator`. Por fim, `det` marca uma função cujo resultado deve depender apenas dos seus argumentos, e o compilador verifica essa afirmação.

3.2 Um programa

ISSN 2317-1316

A Figura 1 mostra uma função que soma, em paralelo, um vetor de números em precisão dupla e devolve sempre o mesmo resultado, qualquer que seja o número de *threads*. Ela é marcada com `det`. A variável `sum` é um acumulador exato, que soma sem arredondar e arredonda uma única vez, no fim; o `parallel for` divide as iterações entre os núcleos, cada um acumulando a sua parte, e `num.MergeExact` combina os acumuladores parciais.

Figura 1: Soma paralela determinística com o acumulador exato.

```
det func total(xs []f64) f64 {
    mut sum := num.NewExact()
    parallel for i in 0..xs.len reduce(num.MergeExact: sum) {
        sum.Add(xs[i])
    }
    return sum.Value()
}
```

3.3 Determinismo verificado

Para cada função, o compilador determina se o resultado depende apenas dos argumentos. Marcar uma função com `det` torna isso uma exigência: se a análise encontrar algo que possa mudar o resultado, o programa não compila. O resultado pode passar a depender de outra coisa quando a função usa um `atomic` ou `shared`, uma chamada para fora da linguagem (como o relógio), um endereço lido como número, uma chamada por interface a algo que não é `det` ou um canal com mais de um remetente ou receptor simultâneo. Por outro lado, a garantia se mantém com um `scope` de várias tarefas, um `parallel for` com redução ou tarefas ligadas por canais com um remetente e um receptor cada (Kahn, 1974). A Figura 2 mostra o que acontece quando uma função marcada com `det` não cumpre a exigência. A função `doubled` devolve um novo vetor com cada elemento do vetor recebido multiplicado por dois, e aloca esse vetor com o alocador do sistema. A primeira linha da mensagem diz que `doubled` está marcada como `det`, mas que o resultado pode depender de mais do que os argumentos; as seguintes mostram o caminho até a causa: `doubled` chama `mem.Alloc`, que usa o alocador recebido, e o alocador do sistema chama uma função fora da linguagem, cujo comportamento o compilador não consegue verificar. A mesma função compila se receber uma arena, um alocador que reparte um vetor da própria linguagem. Programas que dependem da ordem de execução, como um servidor com *locks* e contadores, ficam fora de `det`. Para eles, o comando `shield test -sim` executa cada teste em uma única *thread*, com as escolhas do escalonador tiradas de uma semente, e informa a semente que reproduz uma falha.

ISSN 2317-1316

Figura 2: Mensagens do compilador (Shield 0.2.0) para uma função `det` *que aloca com o alocador do sistema; prefixos de arquivo e trechos de código omitidos.*

```
error: 'doubled' is marked det, but what it answers can depend
       on more than its arguments
note: doubled calls mem.Alloc$i64, which is not det
note: mem.Alloc$i64 calls through an interface, and System.Alloc
      is not det
note: System.Alloc calls out of the language
```

Uma redução dá o mesmo resultado com qualquer número de *threads* porque o corte do intervalo depende apenas do seu comprimento, cada pedaço acumula em um espaço próprio e os pedaços são combinados na ordem do corte. Esse resultado é reprodutível, mas não é igual ao da soma sequencial, porque a ordem dos arredondamentos é outra. A biblioteca padrão completa a garantia: `std/rand` gera o i-ésimo número aleatório em função da semente e de i (Philox4x32-10) (Salmon *et al.*, 2011); `std/num` calcula raiz, exponencial, logaritmo, seno e cosseno com os mesmos bits em qualquer máquina, a no máximo uma ULP do valor verdadeiro (do inglês *unit in the last place*, unidade na última casa: a distância entre um número em precisão dupla e o próximo número que o computador consegue representar; Goldberg, 1991), usando só operações que todo processador arredonda igual (IEEE, 2019) e núcleos do fdlibm (Sun Microsystems, 1993); e `num.Exact` soma sem arredondar até o fim (Kulisch, 2013).

3.4 Um exemplo: merge sort paralelo

A Figura 3 apresenta o código em Sword de um *merge sort* recursivo, paralelo e marcado com `det`. Acima de 4096 elementos, a função divide o vetor em duas metades com `s.chunks(half)` e ordena cada uma em uma tarefa; abaixo disso, ordena as duas na mesma tarefa, sem criar novas. Depois que o `scope` termina, `merge` junta as metades: escreve o resultado da fusão no campo `t` de cada elemento e o copia de volta para `v`. Como cada elemento leva consigo o seu espaço de trabalho, uma única divisão entrega a cada tarefa os dados e o espaço de que ela precisa. A versão mais comum, com um vetor auxiliar separado, é recusada, porque o verificador só aceita pedaços de uma mesma divisão.

ISSN 2317-1316

Figura 3: Merge sort recursivo e paralelo.

```
struct Slot {
    v i64
    t i64
}

func merge(mut s []Slot, mid u64) {
    mut i u64 = 0
    mut j u64 = mid
    for k in 0..s.len {
        if j == s.len || (i < mid && s[i].v <= s[j].v) {
            s[k].t = s[i].v
            i += 1
        } else {
            s[k].t = s[j].v
            j += 1
        }
    }
    for k in 0..s.len {
        s[k].v = s[k].t
    }
}

det func mergeSort(mut s []Slot) {
    if s.len < 2 {
        return
    }
    half := (s.len + 1) / 2
    if s.len <= 4096 {
        mergeSort(s[..half])
        mergeSort(s[half..])
    } else {
        scope {
            for part in s.chunks(half) {
                spawn mergeSort(part)
            }
        }
    }
    merge(s, half)
}
```

3.5 O compilador

O compilador da linguagem Sword, chamado Shield, verifica tipos, corridas e determinismo, gera uma representação intermediária de três endereços e, dela, a do LLVM (Lattner e Adve, 2004), que o clang transforma no executável. São cerca de 11 400 linhas de C++ no compilador, 4 500 no *runtime* e 8 100 de Sword na biblioteca padrão.

4 MATERIAIS E MÉTODOS

Quatro experimentos compararam a Sword com C e OpenMP, cada um respondendo a uma pergunta. O E1 soma 10^7 valores em precisão dupla e pergunta se o resultado muda com o número de *threads*, qual o erro e qual o custo. O E1b repete a mesma soma 100 vezes com o

ISSN 2317-1316

número de *threads* fixo e pergunta se o resultado se repete. O E2 estima por Monte Carlo a integral de $\exp(-x^2)$ entre 0 e 1 com 10^7 amostras, juntando sorteio, função elementar e soma, e pergunta se o cálculo inteiro se reproduz. O E3 avalia 10^6 argumentos das funções seno, cosseno, exponencial e logaritmo e pergunta se elas dão os mesmos bits em outra plataforma.

O E1 usa dados *uniform*, em [0, 1), e *mixed*, com sinais misturados e 18 ordens de grandeza, em que os cancelamentos expõem o arredondamento. As entradas são geradas com semente fixa e conferidas por SHA-256 em cada plataforma; essa conferência detectou, durante o trabalho, um gerador que produzia entradas diferentes no GCC em x86-64, que foi corrigido antes dos experimentos finais. A referência para o erro é o math.fsum do Python, corretamente arredondado e independente dos programas testados (Shewchuk, 1997). Na soma, o C foi testado em série, com OpenMP e com OpenMP e soma compensada de Kahan (Higham, 2002); a Sword, com `reduce(+)`, Kahan e soma exata. No Monte Carlo, o C usou um gerador por *thread*, um por índice e o Philox; dois pares de métodos usam as mesmas amostras nas duas linguagens, para que a diferença de custo venha da linguagem e não do gerador.

Os experimentos rodaram nas cinco plataformas apresentadas na Tabela 1, com 1, 2, 3, 4, 6, 8, 10 e 16 *threads*. Os tempos são a mediana de 7 repetições e vêm de um MacBook Pro com o chip Apple M4 (4 núcleos de desempenho e 6 de eficiência), 24 GB de memória e macOS 27.0. As máquinas do GitHub Actions são compartilhadas e foram usadas apenas para comparar os resultados bit a bit, não os tempos. A Sword usada foi a versão 0.2.0, compilada a partir do mesmo código em cada máquina; o código e os resultados dos experimentos estão disponíveis (Tessaro, 2026b).

Tabela 1. Plataformas

Nome	Máquina	Núcleos	Compilador C
macos-arm64	Apple M4 (4 desempenho + 6 eficiência)	10	Apple clang 21 + libomp
linux-arm64-docker	VM Linux no mesmo M4	10	GCC 13 (libgomp)
gh-linux-x86_64	GitHub Actions, AMD EPYC 9V74	4	GCC 13
gh-linux-arm64	GitHub Actions, ARM Neoverse	4	GCC 13
gh-macos-arm64	GitHub Actions, Apple M1 virtual	3	Apple clang 21 + libomp

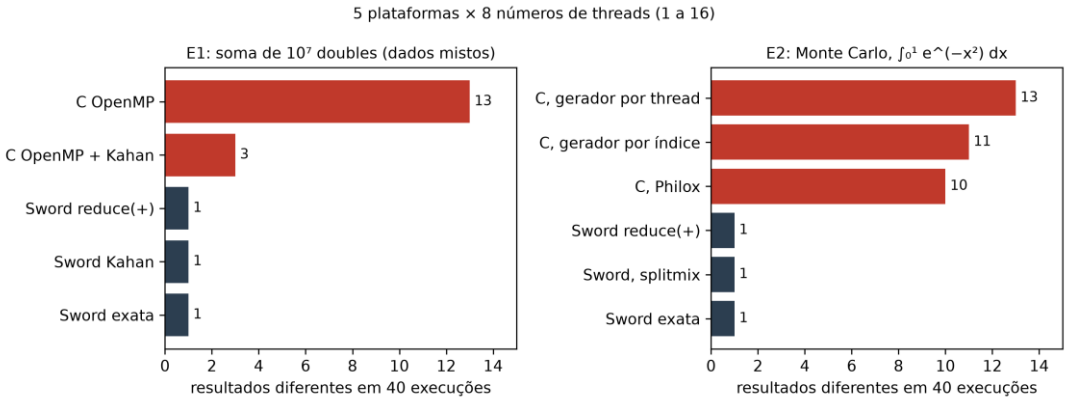
ISSN 2317-1316

5 RESULTADOS E DISCUSSÃO

5.1 Reprodutibilidade

A Figura 4 apresenta quantos resultados diferentes cada método produziu em 40 execuções cada (5 plataformas e 8 contagens de *threads*). Na soma com dados *mixed*, o C com OpenMP produziu 13 resultados diferentes, e o C com OpenMP e soma compensada de Kahan, 3. No Monte Carlo, as três versões em C produziram 13, 11 e 10 resultados. Todos os métodos da Sword produziram um único resultado nos dois experimentos. Na soma, o C serial também deu um único resultado: a variação vem do paralelismo, não da linguagem C. As funções matemáticas mostraram o mesmo padrão: entre Linux e macOS, a biblioteca do sistema deu resultados diferentes para cerca de 4% dos argumentos do seno e do cosseno, enquanto as funções da Sword deram os mesmos bits em todos. A diferença vem das bibliotecas, e não dos processadores, já que duas máquinas Linux com processadores diferentes e a mesma biblioteca concordaram em todos os argumentos.

Figura 4: Resultados diferentes em 40 execuções (5 plataformas e 8 contagens de threads).



5.2 Erro

O erro foi medido em ULPs, comparando cada resultado com a soma exata. Um erro de 30 ULPs significa que, entre o resultado e a soma exata, cabem 30 números que o computador consegue representar. No Apple M4, o erro do C com OpenMP variou de 37 a 243 ULPs conforme o número de *threads*; o da Sword com `reduce(+)` foi 30 em todas as contagens. O erro da soma simples da Sword é menor que o do C serial (243 ULPs), o que é compatível com a soma em pedaços descrita na Seção 3, em que cada parcial acumula menos arredondamento (Higham, 2002). A garantia tem um custo que depende do que se pede. No Apple M4, a soma simples da Sword levou o mesmo tempo que a do C (4,97 ms com uma *thread*), e a soma exata

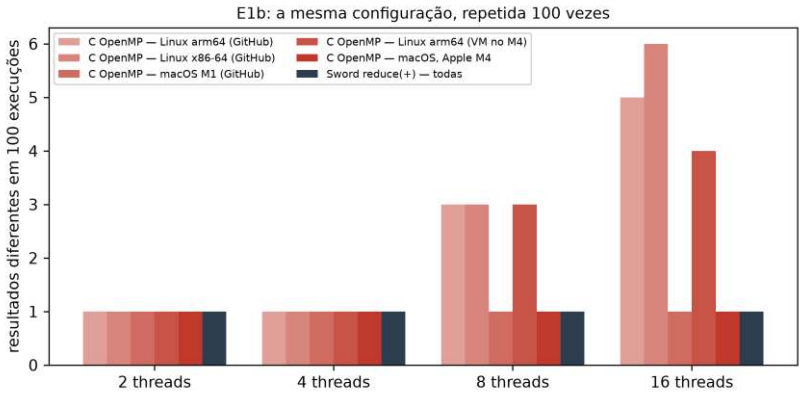
ISSN 2317-1316

cerca de nove vezes mais (44,42 ms). Com dados *uniform*, sem cancelamentos, a diferença cai para cerca de 2,3 vezes. No Monte Carlo, com as mesmas amostras, a Sword levou cerca de 2,7 vezes o tempo do C, a maior parte por causa da função exponencial com os mesmos bits em toda máquina.

5.3 Repetibilidade com o número de *threads* fixo

Fixar o número de *threads* não bastou para que o C repetisse o resultado (Figura 5). Com o OpenMP do GCC, usado nas máquinas Linux, 100 execuções idênticas com 8 ou 16 *threads* deram de 3 a 6 resultados diferentes; com o OpenMP do LLVM, no macOS, e com a Sword, o resultado se repetiu sempre. A especificação do OpenMP permite essa variação, pois não define em que ordem as somas parciais das *threads* são combinadas (OpenMP Architecture Review Board, 2021). No GCC, cada *thread* acrescenta a sua parte ao total na ordem em que termina; quando o OpenMP do LLVM foi configurado para fazer o mesmo, também passou a dar 3 resultados diferentes em 100 execuções.

Figura 5: Resultados diferentes em 100 execuções idênticas, por número de threads.



5.4 Discussão

Os resultados não mostram que C não pode ser reprodutível: geradores por contador, soma reprodutível e uma biblioteca matemática própria resolvem as fontes de variação medidas aqui (Demmel e Nguyen, 2013; Salmon *et al.*, 2011). A diferença é que, em C, manter essa disciplina fica a cargo do programador, e uma única chamada pode desfazê-la sem que o compilador avise; na Sword, o compilador aponta onde a garantia se perde. Reprodutível também não significa correto: a soma simples da Sword erra sempre os mesmos 30 ULPs, e quem precisa do valor correto usa a soma exata, pagando o custo medido na Seção 5.2.

ISSN 2317-1316

6 CONCLUSÃO

Este trabalho apresentou a Sword, uma linguagem de programação em que o compilador detecta corridas de dados e verifica que funções marcadas com `det` dependem apenas dos seus argumentos. Em cinco plataformas, todos os métodos da Sword deram um único resultado para a soma e para o Monte Carlo, enquanto as versões em C com OpenMP chegaram a 13 resultados diferentes em 40 execuções. A garantia não teve custo na soma simples e teve custo mensurável quando se pediu o valor exato ou funções elementares com os mesmos bits em toda máquina. Os próximos passos planejados são um gerenciador de pacotes, na versão 0.3, e, na 0.4, uma simulação de rede e a avaliação de testes executados com escalonamento controlado por semente, como forma de encontrar erros de concorrência. Para uso em computação científica, estão nos planos vetores multidimensionais, instruções SIMD explícitas, execução em GPU e integração com Python, além de avaliar a linguagem em aplicações completas, e não só em núcleos de cálculo. O autor agradece à Universidade Presbiteriana Mackenzie e a Diego Saragoza da Silva, mentor na Apple Developer Academy | Mackenzie, pelo incentivo e pelo direcionamento ao longo deste trabalho.

7 DECLARAÇÃO DE USO DE IA

Declaração de Uso de IA: Durante a elaboração deste trabalho, utilizou-se a ferramenta Claude Code (Anthropic, Claude Opus 5) no desenvolvimento da documentação da linguagem e a ferramenta Claude (Anthropic, Claude Opus 5.5) para a revisão deste documento e para a busca por referências.

8 REFERÊNCIAS

Bocchino RL Jr., Adve VS, Dig D, Adve SV, Heumann S, Komuravelli R, Overbey J, Simmons P, Sung H, Vakilian M. A Type and Effect System for Deterministic Parallel Java. ACM SIGPLAN Notices. 2009;44(10).

Collange S, Defour D, Graillat S, Iakymchuk R. Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures. Parallel Computing. 2015;49.

Demmel J, Nguyen HD. Fast Reproducible Floating-Point Summation. In: Proceedings of the 21st IEEE Symposium on Computer Arithmetic (ARITH). 2013.

Donovan AAA, Kernighan BW. The Go Programming Language. Addison-Wesley; 2015.

Goldberg D. What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Computing Surveys. 1991;23(1).

ISSN 2317-1316

Herlihy M, Shavit N, Luchangco V, Spear M. The Art of Multiprocessor Programming. 2. ed. Cambridge: Morgan Kaufmann; 2020.

Higham NJ. Accuracy and Stability of Numerical Algorithms. 2. ed. Philadelphia: SIAM; 2002.

IEEE. IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019). IEEE. 2019.

Intel Corporation. oneAPI Threading Building Blocks (oneTBB) Developer Guide and API Reference, versão 2021.9: Reduction. 2023. Disponível em: https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021-9/reduction.html. Acesso em: 9 out. 2026.

Kahn G. The Semantics of a Simple Language for Parallel Programming. In: Information Processing 74: Proceedings of the IFIP Congress. North-Holland; 1974.

Kulisch U. Computer Arithmetic and Validity: Theory, Implementation, and Applications. 2. ed. Berlin: De Gruyter; 2013.

Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). 2004.

Matsakis ND, Klock FS II. The Rust Language. ACM SIGAda Ada Letters. 2014;34(3).

OpenMP Architecture Review Board. OpenMP Application Programming Interface, Version 5.2. 2021. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. Acesso em: 9 out. 2026.

Salmon JK, Moraes MA, Dror RO, Shaw DE. Parallel Random Numbers: As Easy as 1, 2, 3. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11). 2011.

Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems. 1997;15(4).
Shewchuk JR. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Discrete & Computational Geometry. 1997;18(3).

Sun Microsystems. FDLIBM: Freely Distributable LIBM. 1993. Disponível em: https://www.netlib.org/fdlibm/. Acesso em: 9 out. 2026.

Tessaro P. Sword: linguagem de programação, versão 0.2.0 "Fold". 2026a. Disponível em: https://github.com/PedroTessaro/sword. Acesso em: 9 out. 2026.

Tessaro P. sword-experiments: reprodutibilidade bit a bit, C com OpenMP contra Sword. 2026b. Disponível em: https://github.com/PedroTessaro/sword-experiments. Acesso em: 9 out. 2026.

ISSN 2317-1316

UXL Foundation. oneAPI Specification, versão 1.3-rev-1: oneTBB, parallel_deterministic_reduce. 2022. Disponível em: https://oneapi-spec.uxlfoundation.org/specifications/oneapi/v1.3-rev-1/elements/onetbb/source/algorithms/functions/parallel_deterministic_reduce_func. Acesso em: 9 out. 2026.

Zig Software Foundation. Zig Language Reference. 2026. Disponível em: https://ziglang.org/documentation/master/. Acesso em: 9 out. 2026.