NAS PARALLEL BENCHMARK: NPB-GO

IGOR YUJI ISHIHARA SAKUMA[1]; GERSON GERALDO HOMRICH CAVALHEIRO[2]

[1]Universidade Federal de Pelotas – iyisakuma@inf.ufpel.edu.br
[2]Universidade Federal de Pelotas – gerson.cavalheiro@inf.ufpel.edu.br

1. INTRODUÇÃO

Em 1965, Gordon Moore, cofundador da Intel, observou que o número de transistores em um chip dobraria a cada 18 a 24 meses, resultando em crescimento exponencial do poder de processamento dos computadores (PERSON, 1965). Entretanto, com a proximidade dos limites físicos dessa tendência, o aumento do desempenho passou a depender de arquiteturas com múltiplos núcleos, capazes de explorar o paralelismo.

Nesse contexto, a Computação de Alto Desempenho (HPC) tornou-se essencial para resolver problemas científicos e de engenharia em larga escala, incluindo aplicações em IA e Big Data. Seu avanço depende não apenas de supercomputadores, mas também de métricas e ferramentas para avaliar o desempenho de arquiteturas e linguagens.

Entre essas ferramentas, vários benchmarks são adotados pela comunidade para uniformizar a avaliação de sistemas computacionais. Dentre estes, destaca-se o NAS Parallel Benchmark (NPB), desenvolvido pela NASA em 1991 (BAILEY et al., 1991), amplamente utilizado para medir e comparar o desempenho de sistemas paralelos. O NPB reúne cinco kernels (EP, CG, FT, IS e MG) e três aplicações (BT, SP e LU) que representam diferentes padrões de comunicação e carga computacional (JIN; FRUMKIN; YAN, 1999).

Nos últimos anos, a linguagem Go, criada pelo Google em 2009 (GRIESEMER; PIKE; THOMPSON, 2012), tem ganhado espaço por seu modelo de concorrência leve, baseado em goroutines e canais, que alia simplicidade de uso a eficiência (GO TEAM, 2025). Diferentemente de linguagens que utilizam threads como principal unidade de execução paralela, Go adota uma filosofia inspirada no modelo Communicating Sequential Processes (CSP), proposto por Hoare em 1978. Nesse modelo, sistemas concorrentes são construídos a partir de processos independentes que se comunicam exclusivamente por meio de canais, sem compartilhamento direto de memória.

Go transpõe esse paradigma ao introduzir as goroutines, unidades leves de execução concorrente, e os canais como mecanismos de troca de mensagens, eliminando a necessidade de sincronização explícita por bloqueios em muitos cenários. Essa adoção prática do CSP distingue Go de outras linguagens modernas e representa um diferencial significativo para aplicações concorrentes e distribuídas.

Somado a isso, recursos como o gerenciamento automático de memória via garbage collector e o modelo explícito de tratamento de erros tornam o desenvolvimento em Go mais seguro, reduzindo falhas comuns em ambientes de paralelismo intenso (GRIESEMER; PIKE; THOMPSON, 2012)

Embora existam estudos comparando Go a outras linguagens em microbenchmarks (MCLAUGHLIN, 2016; HIGHTOWER, 2014; KUMAR, 2017; SINGH, 2017), não há implementações completas dos principais kernels do NPB

em Go para comparações sistemáticas. Essa lacuna motivou este trabalho, que visa implementar e validar versões serial e paralela dos cinco kernels principais do NPB em Go, explorando o modelo de concorrência baseado em CSP e avaliando sua viabilidade na Computação de Alto Desempenho.

2. METODOLOGIA

O trabalho está sendo desenvolvido por meio de uma abordagem iterativa e incremental, estruturada em diferentes etapas. Inicialmente, será realizada uma revisão das especificações do NAS Parallel Benchmarks (NPB) e análise das implementações de referência existentes em linguagens como C, Fortran e Rust. Em seguida, proceder-se-á à coleta de bibliografia por meio da metodologia snowballing (bola de neve), tanto retrospectiva quanto prospectiva, a partir das implementações de referência e das produções anteriores do grupo de pesquisa no qual este estudo se insere.

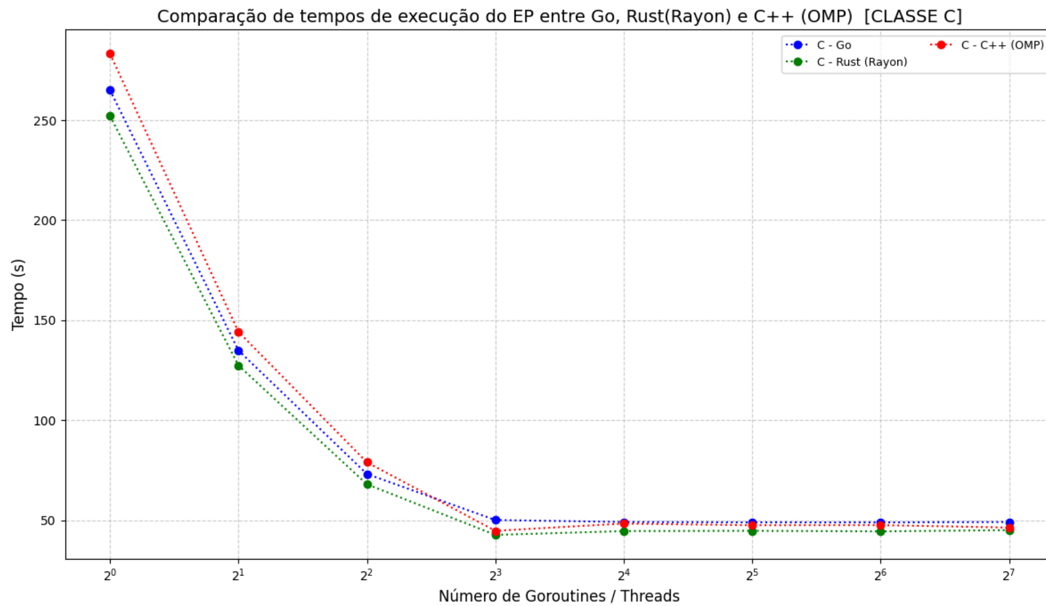
Na sequência, serão implementadas as versões seriais dos cinco kernels principais do NPB em Go, com validação de resultados. Posteriormente, será conduzida a implementação das versões paralelas, explorando o modelo de concorrência nativo da linguagem, baseado em goroutines e canais. A etapa de validação consistirá na verificação da correção das implementações, mediante comparação com as saídas das versões oficiais, bem como na análise de desempenho, empregando métodos estatísticos apropriados.

As ferramentas e tecnologias que serão utilizadas incluem: a linguagem Go em sua versão estável atual, o sistema de controle de versão Git e ambientes de execução com suporte a múltiplos núcleos. Ressalta-se, por fim, que este trabalho conta com o apoio do Grupo de Modelagem de Aplicação Paralela (GMAP) da Pontifícia Universidade Católica do Rio Grande do Sul, que fornecerá suporte metodológico e experimental.

3. RESULTADOS E DISCUSSÃO

Os experimentos foram realizados em uma plataforma multicore equipada com um processador Intel® Core™ i5-1135G7 de 11ª geração, com frequência de 2,40 GHz e 8 núcleos lógicos (4 núcleos físicos com Hyper-Threading). Em relação à memória, o sistema possui 32 GB de RAM, 80 KB de cache L1 por núcleo, 5 MiB de cache L2 e 8 MB de cache L3 compartilhado. A GPU integrada é a Intel® Xe Graphics (TGL GT2). A capacidade de armazenamento é de 256,1 GB. O sistema operacional é o Ubuntu 22.04.5 LTS (64 bits).

Até o momento, foram desenvolvidas duas versões do kernel EP: uma serial, utilizada como baseline, e uma concorrente em Go, que distribui o conjunto de dados entre goroutines de acordo com o número de threads lógicas, utilizando sync.WaitGroup para sincronização. O kernel Embarrassingly Parallel (EP), da suíte NAS Parallel Benchmarks (NPB), avalia o desempenho de sistemas paralelos em tarefas independentes com mínima comunicação, como geração de números pseudoaleatórios e tabulação de desvios gaussianos em métodos Monte Carlo. Utilizou-se a classe C, representando maior complexidade e escala.



A análise dos resultados mostra que, para a classe de problema C, a paralelização proporciona ganhos expressivos de desempenho, embora ainda limitados pelo overhead de gerenciamento de threads ou gorrotinas. Rust (Rayon) apresenta, de forma consistente, os melhores speedups, refletindo maior eficiência no balanceamento de carga e menor custo de sincronização. Go mantém desempenho competitivo, mesmo com o overhead introduzido pelo garbage collector, que simplifica o gerenciamento de memória. Já C++ com OpenMP apresenta maior variabilidade nos resultados, sugerindo sensibilidade a fatores de escalonamento e afinidade de threads. No geral, a escalabilidade prática atinge cerca de 5–6 vezes em 8 threads/gorrotinas, com eficiência próxima a 70–75%, em conformidade com a Lei de Amdahl, indicando que ganhos adicionais dependem de otimizações específicas nas implementações paralelas

Por fim, os ganhos estabilizaram por limitações de hardware e custos de sincronização, além do overhead do garbage collector em Go. Esse comportamento evidencia que melhorias adicionais dependem de otimizações específicas e da arquitetura subjacente.

4. CONCLUSÕES

Este trabalho apresentou uma investigação comparativa do desempenho do kernel Embarrassingly Parallel (EP) da suíte NAS Parallel Benchmarks em diferentes linguagens de programação, cada uma utilizando paradigmas distintos de concorrência: goroutines em Go, a biblioteca Rayon em Rust e OpenMP em C++.

A principal inovação deste estudo está na análise experimental do modelo de concorrência do Go em um cenário tradicionalmente dominado por linguagens com suporte nativo ou otimizações consolidadas para HPC (High Performance Computing), como C++ e Rust. Ao implementar e avaliar o EP em Go, foi possível explorar as particularidades do modelo baseado em CSP e goroutines, destacando suas vantagens, limitações e implicações para aplicações paralelas de alta intensidade computacional.

Outra contribuição relevante é a padronização metodológica da comparação, utilizando o mesmo kernel e a classe de problema C, além das mesmas métricas (tempo de execução e Mop/s), garantindo equidade entre as implementações. Essa abordagem fornece uma base sólida para estudos futuros sobre linguagens modernas sob a ótica do paralelismo e da escalabilidade em cenários de alta complexidade, especialmente diante do avanço das arquiteturas multicore.

Apesar do Go ter sido projetado para simplificar a programação concorrente e, em muitos casos, introduzir menos overhead que abordagens baseadas em diretivas, como o OpenMP, seu desempenho ainda não atinge o de Rust, que foi concebida com otimizações específicas para paralelismo multithread e apresenta menor custo de sincronização. Além disso, a ausência de mecanismos nativos como reduce e map-reduce, somada à falta de bibliotecas consolidadas para essas operações, limita a expressividade da linguagem frente a ferramentas maduras como OpenMP e Rayon, impactando produtividade e clareza em padrões paralelos amplamente utilizados. Essa limitação reforça a necessidade de estratégias mais avançadas para que Go se torne competitivo em cenários intensivos de HPC.

Por fim, este trabalho amplia a compreensão sobre o potencial do Go em cenários de computação científica e de alto desempenho, propondo novas perspectivas para seu uso em ambientes HPC, seja como alternativa ou como complemento a linguagens tradicionais.

5. REFERÊNCIAS BIBLIOGRÁFICAS

BAILEY, D. H. et al. The NAS Parallel Benchmarks. **International Journal of Supercomputer Applications**, v. 5, n. 3, p. 63–73, 1991.

COX, R. et al. **The Go Programming Language: Design and Implementation**. Boston: Addison-Wesley, 2022.

GO TEAM. **The Go Programming Language Specification**. Google, 2025. Acessado em 29 Ago. 2025. Disponível em: https://go.dev/ref/spec

GRIESEMER, R.; PIKE, R.; THOMPSON, K. The Go Programming Language. **Communications of the ACM**, v. 55, n. 2, p. 89–95, 2012.

HIGHTOWER, K. Practical Go: Building Scalable Applications. San Francisco: **TechPress**, 2014.

JIN, H.; FRUMKIN, M.; YAN, J. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. **NASA Ames Research Center**, 1999.

KUMAR, A.; SINGH, R. Comparative Study of Go and Java for Concurrent Applications. **International Journal of Computer Applications**, v. 162, n. 4, p. 1–7, 2017.

MCLAUGHLIN, B. Go in Practice: Techniques for Writing Scalable Applications. **New York: Manning Publications**, 2016.

PERSON, G. E. **Cramming More Components onto Integrated Circuits**. Electronics Magazine, v. 38, n. 8, p. 114–117, 1965.