Comparação de Paralelismo com *DO CONCURRENT*, *OpenMP* e *MPI* em Algoritmos do *NAS Parallel Benchmark*[*]

Anna V. G. Marciano[†], Artur dos Santos Antunes, Claudio Schepke

[1]Laboratório de Estudos Avançados em Computação (LEA)
Universidade Federal do Pampa (UNIPAMPA) – Alegrete – RS – Brazil

{annamarciano,arturantures}.aluno@unipampa.edu.br
claudioschepke@unipampa.edu.br

Abstract. *This paper compares native Fortran parallelism, via* `DO CONCURRENT`*, with OpenMP and MPI in three algorithms from the NAS Parallel Benchmark (Conjugate Gradient, Multi-Grid, and Fast Fourier Transform. The results show that* `DO CONCURRENT` *achieves competitive performance in scenarios with regular memory access, such as CG and MG, but is less efficient in FFT, which requires intensive global synchronization. The main contribution is to demonstrate the practical feasibility of* `DO CONCURRENT` *on the CPU, offering more readable and maintainable code without losing significant efficiency.*

Resumo. *Este trabalho compara o paralelismo nativo do Fortran, via* `DO CONCURRENT`*, com OpenMP e MPI em três algoritmos do NAS Parallel Benchmark (Conjugate Gradient, Multi-Grid e Fas-Fourier Transform). Os resultados mostram que* `DO CONCURRENT` *alcança desempenho competitivo em cenários com acesso regular à memória, como CG e MG, mas é menos eficiente em FFT, que exige comunicação global intensiva. A contribuição principal é evidenciar a viabilidade prática do* `DO CONCURRENT` *em CPU, oferecendo código mais legível e de fácil manutenção sem perder eficiência significativa.*

1. Introdução

O paralelismo é fundamental para acelerar simulações e resolver problemas complexos em tempo razoável. Interfaces de Programação Paralela (*APIs*), sejam extensões de linguagens, bibliotecas ou *frameworks*, possibilitam a criação e o gerenciamento de *threads* [Chapman et al. 1998]. Em ambientes distribuídos, alternativas, como *Message Passing Interface* (*MPI*), permitem comunicação entre processos com memória separada [Löff et al. 2021].

Desde *Fortran 2008*, o construtor `DO CONCURRENT` introduziu uma forma padronizada de expressar paralelismo em *loops*, delegando ao compilador a responsabilidade de realizar a paralelização e otimizações, como vetorização e execução concorrente [Reid 2018]. Diferentemente de abordagens como *Open Multi-Processing* (*OpenMP*), que dependem de diretivas para geração explícita de *threads* [Ayguade et al. 2009], `DO CONCURRENT` permite escrever código limpo e portável, facilitando a manutenção e ainda permitindo que o compilador permaneça responsável pelo paralelismo.

[*]Trabalho parcialmente financiado pelo Edital CNPq: Projeto 135963/2023-0.
[†]Bolsista PIBIC/CNPq 2024/2025.

Apesar dos benefícios teóricos do `DO CONCURRENT`, sua adoção na comunidade científica permanece limitada devido a incertezas sobre seu desempenho real em aplicações complexas e, também, sobre o suporte heterogêneo entre compiladores [Tremarin et al. 2024]. A falta de evidências empíricas sobre a competitividade do paralelismo automático comparado às abordagens explícitas tradicionais representa um obstáculo significativo para sua adoção mais ampla. Este trabalho aborda essas incertezas ao demonstrar que, embora existam limitações específicas, o `DO CONCURRENT` pode atingir desempenho competitivo em *benchmarks* de alta complexidade, especialmente quando considerada a melhoria na legibilidade e manutenibilidade do código.

Este artigo investiga o impacto do paralelismo da cláusula `DO CONCURRENT` contra o paralelismo tradicional em três algoritmos do *NAS Parallel Benchmarks* (*Conjugate Gradient*, *Multi-Grid* e *Fast-Fourier Transform*). O trabalho fornece dados empíricos para orientar decisões de implementação em projetos de computação científica e também busca reduzir a lacuna pendente, avaliando o desempenho do `DO CONCURRENT` comparando-o com *OpenMP* e *MPI*.

2. Trabalhos Relacionados

Desde a sua introdução no *Fortran 2008*, `DO CONCURRENT` vem sendo avaliado como alternativa aos modelos tradicionais de paralelismo baseado em diretivas. Estudos recentes indicam que seu desempenho pode se aproximar ou até superar implementações com *OpenMP*, especialmente em arquiteturas com bom suporte à vetorização [Hammond et al. 2022, Stulajter and Smith 2022]

No artigo de Gabriel Tremarin, foi investigado o uso do `DO CONCURRENT` no algoritmo *Conjugate Gradient* do *NPB* em uma aplicação de meio poroso, mostrando que o desempenho é competitivo quando o padrão de acesso à memória é regular [Tremarin et al. 2024]. Outros trabalhos recentes reforçam essa linha de pesquisa:

- Foi analisado a possibilidade de substituir diretivas *OpenMP* por `DO CONCURRENT` em computação acelerada no artigo de Stulajter, identificando vantagens de portabilidade [Stulajter et al. 2021];
- no artigo de Hammond foi realizado o *benchmarking* em CPUs e GPUs com *BabelStream*, mostrando que o ganho de desempenho depende fortemente do compilador [Hammond et al. 2022];
- Maqbool e Lee compararam `DO CONCURRENT` e *OpenMP* em simulações de manufatura aditiva, demonstrando que é possível atingir desempenho equivalente utilizando apenas paralelismo padrão da linguagem, sem dependência de APIs externas [Maqbool and Lee 2025];

No campo das bibliotecas e modelos de paralelismo, destacam-se os trabalhos fundadores do *OpenMP* [Dagum and Menon 1998], *MPI* [Gropp et al. 1996] e *OpenACC* [Chandrasekaran and Juckeland 2017], que permanecem como referências essenciais.

O presente trabalho expande os estudos anteriores ao incluir não apenas o *Conjugate Gradient*, mas também *Multi-Grid* e *Fast-Fourier Transform*, permitindo avaliar o comportamento do `DO CONCURRENT` sob diferentes padrões de comunicação (ponto a ponto, local/global e global) e de uso de memórias.

Tabela 1. Resumo dos *benchmarks NPB* utilizados

Benchmark	**Tipo de acesso**	**Comunicação**	**Uso de memória**
Conjugate Gradient	Irregular	Ponto-a-ponto	Moderado
Multi-Grid	Regular	Local/Global	Intensivo
Fast-Fourier Transform	Regular	Global	Moderado

3. *NAS Parallel Benchmark*

NAS Parallel Benchmarks (*NPB*) é uma coleção de algoritmos destinados a medir o desempenho de supercomputadores paralelos, originados de aplicações de Dinâmica de Fluidos Computacional (DFC). Neste trabalho, avaliamos três algoritmos principais da versão *NPB* 3.4.3:

- ***Conjugate Gradient* (*CG*):** Avalia acesso irregular à memória e comunicação ponto-a-ponto.
- ***Multi-Grid* (*MG*):** Avalia comunicação de curta e longa distância e uso intensivo de memória.
- ***Fast-Fourier Transform* (*FFT*):** Avalia comunicação global entre processos.

A escolha desses algoritmos permite capturar diferentes dimensões do desempenho paralelo: *CG* evidencia acesso irregular à memória, *MG* combina comunicação local e remota, e *FFT* representa operações globais de sincronização e redistribuição de dados. Dessa forma, obtém-se uma visão abrangente da eficiência de paralelismo, uso de memória e comunicação em arquiteturas *HPC* [Löff et al. 2021]. Com base na documentação oficial do *NPB* disponibilizada pela *NASA*, foi elaborada a Tabela 1 pelo autor, que compara os três *benchmarks* quanto ao tipo de acesso, comunicação e uso de memória.

4. Metodologia

Os testes tiveram como objetivo avaliar o **tempo de execução** e a **compatibilidade numérica** das implementações dos *benchmarks* selecionados. O código foi compilado utilizando o compilador *nvfortran* (*HPC SDK* 25.5, *Nvidia*), empregando a *flag -O3* para otimização sequencial e `-fopenmp` para paralelismo *OpenMP*, garantindo otimizações agressivas de instruções, vetorização e uso eficiente de memória.

4.1. Uso da classe C

Cada *benchmark* foi executado utilizando a Classe C do *NPB*. Esta escolha se justifica por representar problemas suficientemente grandes para evidenciar diferenças significativas de desempenho entre os modelos de paralelismo (*OpenMP*, *DO CONCURRENT* e *MPI*), sem sobrecarregar os recursos computacionais da estação de trabalho utilizada. A Tabela 2, elaborada pelos autores com base na documentação oficial do *NPB* disponibilizada pela *NASA*, apresenta a classificação das classes do *NPB*. Classes menores (S, W, A, B) não seriam adequadas para demonstrar as diferenças de *overhead* entre as abordagens, enquanto classes maiores (D, E, F) ultrapassariam os limites de memória e capacidade de processamento disponíveis. Assim, a Classe C oferece um equilíbrio ideal entre complexidade computacional e viabilidade experimental, permitindo análises estatisticamente confiáveis em um ambiente controlado.

Tabela 2. Classes do *NPB*

Classe	Tamanho	Abordagem	Velocidade
S	Muito pequeno	Teste rápido	Muito rápida
W	Pequeno	Teste rápido	Rápida
A	Médio	Realista	Média
B	Médio	Realista	Média
C	**Grande**	**Realista**	**Média/Lenta ✓**
D	Muito grande	Supercomputadores	Lenta
E	Imensa	Supercomputadores	Muito lenta
F	Extrema	Supercomputadores	Extremamente Lenta

4.2. Ambiente de Execução

O ambiente de execução consistiu em um ***Windows Subsystem for Linux*** (*WSL*) sobre ***Windows 11***, com processador ***AMD Ryzen 5 3400G*** de 4 núcleos físicos e 8 *threads* e **16 GB de memória RAM**. Para o paralelismo em memória compartilhada, foram escolhidas **4 *threads***, correspondendo ao número de núcleos físicos do processador. Essa decisão visa evitar que múltiplas *threads* virtuais (*hyper-threading*) compartilhem os mesmos recursos de núcleo, minimizando a contenção de cache L1/L2, e proporciona uma avaliação mais estável e representativa do desempenho do `DO CONCURRENT` em comparação com implementações *OpenMP*, que neste trabalho foi utilizada na versão suportada pelo ***gfortran 13.3.0*** (compatível com *OpenMP 5.1*).

O paralelismo distribuído foi realizado utilizando *MPI* (***Open MPI 4.1.7rc1***), permitindo a execução de processos independentes com comunicação explícita entre eles. A repetição de execuções permite reduzir a variabilidade causada por flutuações do sistema operacional, escalonamento de *threads* e utilização de cache, garantindo a confiabilidade das medições. Para este trabalho, cada *benchmark* foi executado **18 vezes**, permitindo o cálculo de médias e análise estatística robusta dos resultados.

Scripts em *shell* automatizaram o processo de compilação e execução, garantindo consistência entre os testes e evitando erros. Cada execução gerou arquivos de saída com tempos de execução e métricas de desempenho que, em seguida, foram consolidadas por scripts *Python* em arquivos *.json* para análise estatística, geração de gráficos e comparação entre as estratégias de paralelismo adotadas.

5. Implementação

Os *loops* paralelizados com *OpenMP* foram convertidos para `DO CONCURRENT`, usando o especificador `local`, conforme pode ser visto nos Códigos 1 e 2. *Loops* com chamadas de função permaneceram sem alteração, devido a limitações da construção.

Código 1. Exemplo de uso de `private` com *OpenMP*

```
!$omp parallel do private(i,j,k) collapse(2)
do k = 1, d3
  do j = 1, d2
    do i = 1, d1
       u0(i,j,k) =0.d0
    end do
  end do
end do
```

Código 2. Exemplo com `DO CONCURRENT`

```
do concurrent (k = 1:d3) local(i,j)
  do j = 1, d2
    do i = 1, d1
       u0(i,j,k) =0.d0
    end do
  end do
end do
```

Já no contexto do *MPI*, a implementação do paralelismo envolve a decomposição do problema em múltiplos processos independentes, cada qual com seu próprio espaço de memória. A comunicação entre esses processos ocorre de forma explícita por meio de chamadas de envio e recebimento de mensagens, permitindo a execução de programas em ambientes distribuídos. Essa abordagem confere ao *MPI* elevada escalabilidade, sendo especialmente adequada para supercomputadores e *clusters*; entretanto, também aumenta a complexidade da implementação, uma vez que o programador deve gerenciar explicitamente a troca de dados e a sincronização entre processos.

6. Resultados

A Figura 1 apresenta a média dos tempos de execução obtidos para os três *benchmarks* avaliados.

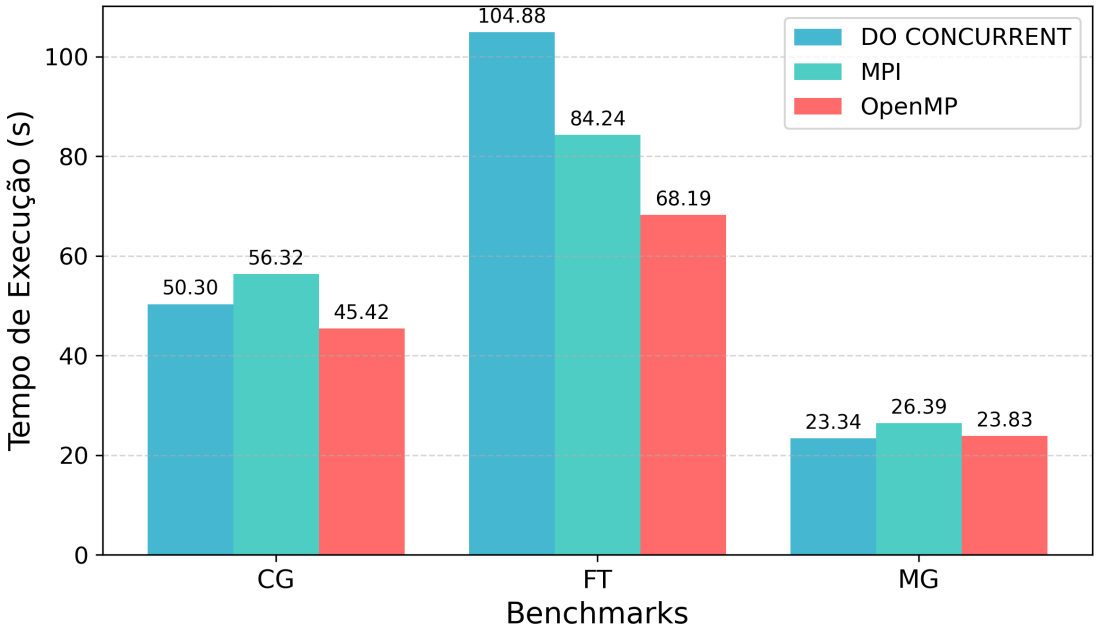

Figura 1. Comparativo de performance dos *benchmarks*.

6.1. *Conjugate Gradient*

O *benchmark Conjugate Gradient* apresenta tempos de execução distintos dependendo do modelo de paralelismo empregado. O uso de *OpenMP* (45,42s) mostrou-se mais eficiente, seguido de `DO CONCURRENT` (50,3s) e *MPI* (56,32s). A diferença observada evidencia que, para este algoritmo, o paralelismo baseado em *threads* compartilhadas (*OpenMP*) beneficia-se melhor do acesso à memória e da sobrecarga reduzida na comunicação entre *threads*, enquanto o *MPI*, que depende da comunicação entre processos distribuídos, apresenta maior latência, impactando negativamente o tempo total de execução.

6.2. *Multi-Grid*

No *benchmark Multi-Grid*, os tempos de execução entre as diferentes formas de paralelismo foram mais próximos. Observa-se que `DO CONCURRENT` atingiu 23,34s, *OMP* 23,83s e *MPI* 26,39s. A pequena variação entre `DO CONCURRENT` e *OMP* sugere que este algoritmo se beneficia de paralelismo em memória compartilhada e da facilidade de expressão do paralelismo pelo compilador, enquanto *MPI* ainda apresenta *overhead* ligeiramente maior devido à comunicação entre processos.

6.3. *Fast-Fourier Transform*

O *benchmark Fast Fourier Transform* apresentou comportamento distinto em relação aos demais. *OMP* apresentou o menor tempo de execução (68,19s), seguido de **MPI** (84,24s) e `DO CONCURRENT` (104,88s). O desempenho superior de *OpenMP* neste caso indica que *FFT* possui muitas operações paralelizáveis em *loops* aninhados e acesso eficiente à memória local, favorecendo modelos de paralelismo de *threads*. O desempenho relativamente inferior do `DO CONCURRENT` sugere que, embora o paralelismo automático

de *loops* seja eficaz, ele não consegue extrair toda a performance possível frente a um controle mais explícito, como o proporcionado por *OpenMP*.

6.4. Análise Completa

A análise consolidada de todos os *benchmarks* evidencia tendências gerais no comportamento dos diferentes modelos de paralelismo. *OpenMP* apresentou o melhor desempenho médio nos três *benchmarks*, seguido de `DO CONCURRENT` e *MPI*, embora com variações específicas dependendo da natureza do algoritmo. Algoritmos com forte dependência de acesso à memória compartilhada, como *CG* e *FFT*, se beneficiam mais de paralelismo em *threads*. Por outro lado, algoritmos como *MG*, que possuem etapas mais balanceadas e comunicação entre subproblemas, apresentam tempos mais homogêneos entre os modelos, com *MPI* mostrando leve desvantagem devido à sobrecarga de comunicação entre processos.

A comparação consolidada confirma que a escolha do modelo de paralelismo deve considerar tanto a arquitetura da máquina quanto o padrão de acesso à memória e comunicação do algoritmo. *OpenMP* é geralmente mais indicado para sistemas com memória compartilhada e grande quantidade de *loops* paralelizáveis, enquanto *MPI* é mais adequado para ambientes distribuídos com baixo grau de comunicação entre processos.

7. Considerações Finais

Este estudo demonstrou que o `DO CONCURRENT` é uma ferramenta viável e competitiva para paralelização em CPU, especialmente quando considerados os benefícios de código mais legível e facilmente manutenível. Os resultados confirmam que, embora existam limitações específicas em cenários como *loops* com chamadas de funções ou algoritmos que exigem comunicação global complexa, `DO CONCURRENT` se aproxima significativamente do desempenho de soluções explícitas como *OpenMP* em *benchmarks* representativos como *CG* e *MG*.

A contribuição principal deste trabalho é fornecer evidências que podem orientar decisões de implementação na comunidade científica. Para a maioria dos casos de paralelização de *loops* simples a complexos, `DO CONCURRENT` oferece uma alternativa atrativa que combina facilidade de implementação com eficiência competitiva.

As limitações identificadas, particularmente no *benchmark FFT*, não diminuem o valor do `DO CONCURRENT`, mas sim definem com clareza os seus domínios de aplicação. Para desenvolvedores que priorizam legibilidade, manutenibilidade e portabilidade do código, `DO CONCURRENT` representa uma opção válida na maioria dos cenários práticos.

Como trabalhos futuros, pretendemos:

- Paralelizar todos os *benchmarks* do *NPB* restantes, como *Scalar Penta-diagonal solver* (*SP*) e *Lower-Upper Gauss-Seidel solver* (*LU*), por exemplo, de forma com que utilizemos diferentes formas de paralelismo;
- Explorar Classes D e E do *NPB* para avaliar como a escalabilidade do `DO CONCURRENT` se comporta em problemas de dimensões maiores;
- Avaliar *benchmarks* adicionais para validar a generalização dos resultados;
- Investigar otimizações de compilador e *flags* de otimização para verificar o quão afeta o desempenho do `DO CONCURRENT`;

- Estudar a possibilidade de implementações híbridas para otimização em diferentes escalas de paralelização, possivelmente combinando *OpenMPI* com `DO CONCURRENT`, visto que a implementação de ambos é semelhante.

Referências

Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418.

Chandrasekaran, S. and Juckeland, G. (2017). *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 1st edition.

Chapman, B., Mehrotra, P., and Zima, H. (1998). Enhancing OpenMP with features for locality contro. In *Proc. ECWMF Workshop: Towards Teracomputing-The Use of Parallel Processors in Meteorology*, Austria. PSU.

Dagum, L. and Menon, R. (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55.

Gropp, W., Lusk, E., and Skjellum, A. (1996). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1st edition.

Hammond, J. R., Deakin, T., Cownie, J., and McIntosh-Smith, S. (2022). Benchmarking Fortran DO CONCURRENT on CPUs and GPUs Using BabelStream. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 82–99.

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

Maqbool, S. and Lee, B.-J. (2025). High performance additive manufacturing phase field simulation: Fortran do concurrent vs openmp. *Computational Materials Science*, 252:113788.

Reid, J. (2018). The New Features of Fortran 2018. *SIGPLAN Fortran Forum*, 37(1):5–43.

Stulajter, J. and Smith, A. (2022). Advances in Parallel Programming Techniques for Modern HPC Systems. *Journal of Parallel and Distributed Computing*, 160:1–15.

Stulajter, M. M., Caplan, R. M., and Linker, J. A. (2021). Can Fortran's 'do concurrent' replace directives for accelerated computing?

Tremarin, G., Marciano, A., Schepke, C., and Vogel, A. (2024). Fortran DO CONCURRENT Evaluation in Multi-core for NAS-PB Conjugate Gradient and a Porous Media Application. In *Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 133–143, Porto Alegre, RS, Brasil. SBC.