Fortran DO CONCURRENT Evaluation in Multi-core for NAS-PB Conjugate Gradient and a Porous Media Application

Gabriel Dineck Tremarin[1], Anna Victória Gonçalves Marciano[1], Claudio Schepke[1], Adriano Vogel[2,3]

[1] Laboratory of Advanced Studies in Computation
Federal University of Pampa (UNIPAMPA), Alegrete - RS, Brazil.

[2]JKU/Dynatrace Co-Innovation Lab, LIT CPS Lab,
Johannes Kepler University Linz, Austria

[3]Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC),
Sociedade Educacional Três de Maio (SETREM)

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

Abstract. *High-performance computing exploits the hardware resources available to accelerate the applications' executions, whereas achieving such an exploitation of hardware resources demands software programming. Hence, several parallel programming interfaces (PPIs) are used for sequential programs to call thread resources and parallelism routines. There are explicit PPIs (e.g., Pthreads and TBB) or implicit (e.g., OpenMP and OpenACC). Another approach is parallel programming languages like the Fortran 2008 specification, which natively provides the DO CONCURRENT resource. However, DO CONCURRENT's evaluation is still limited. In this paper, we explore and compare the native parallelism of FORTRAN with the directives provided by the OpenMP and OpenACC PPIs in the NAS-PB CG benchmark and a porous media application. The results show that the DO CONCURRENT provides parallel CPU code with numerical compatibility for scientific applications. Moreover, DO CONCURRENT achieves in multi-cores a performance comparable to and even slightly better than other PPIs, such as OpenMP. Our work also contributes with a method to use DO CONCURRENT.*

1. Introduction

Usually, the concurrent execution of instructions is through a parallel programming interface (PPI). A PPI can be a programming language extension, library, framework, or domain-specific language that offers resources for thread creation and instantiation. There are still other alternatives if we include distributed execution, such as Application Programming Interface (API), Communicating Sequential Processes (CSP), and Partitioned Global Address Space (PGAS). Parallel programming tools benefit different kinds of applications by allowing the generation of concurrent code through processes, threads, accelerators, or vector instructions [Vetter 2013, Kirk and Wen-Mei 2016].

On the other hand, the idea of having a specific language for concurrent programming for any class of applications was unsuccessful [Ozen 2018]. For example, the tentative decision to define High-Performance Fortran extension (HPF [Koelbel et al. 1993]) as

a parallel programming language standard ended up. Most programmers preferred to move towards an OpenMP-oriented programming approach instead of using HPF as a parallel programming language [Kennedy et al. 2007]. Moreover, there are FORTRAN specifications that incorporate some of HPF's ideas.

Considering the Fortran 2008 standard and the increase of locality specifiers in 2018 [Reid 2018], it is possible to create concurrency over loops using the DO CONCURRENT language reserved words. DO CONCURRENT differs from the parallelism provided by precompiled directives. Approaches such as OpenMP and OpenACC rely on directives (pragmas) embedded in the source code to introduce thread routines during compilation. In contrast, DO CONCURRENT enables loop parallelism without requiring any modifications to the source code.

In this context, **this article examines the impact of the DO CONCURRENT clause in Fortran on parallelism, comparing it to the parallelism achieved through compilation directives in PPIs. The study uses the NAS Parallel Benchmark [Löff et al. 2021] and a porous media application [da Silva et al. 2022] as workloads.**. We evaluate the performance of DO CONCURRENT, OpenMP parallel do, and OpenACC parallel loop for Multi-core and GPU architectures, including the need to change the source code and the impact on runtime. The main goal is to measure whether the abstractions provided by DO CONCURRENT are a suitable alternative to replace directives and also maintain the same performance [Vogel et al. 2021]. The **contributions** of this paper are: (i) An assessment of native Fortran parallelism and parallel directives approaches. (ii) A characterization of the necessary steps and the modifications in the code to accelerate necessary for the different approaches covered.

The remainder of the paper's organization is as follows. Section 2 presents the related work. Section 3 describes the DO CONCURRENT command. The description of the methodology of the experiments is in Section 4. The development of the parallelization is in Section 5. Section 6 shows the performance results for the different parallel implementation approaches. Finally, Section 7 presents the conclusion and future work.

2. Related Work

The DO CONCURRENT feature appeared in the 2008 Fortran specification, which was expanded in the 2018 specification. However, there appears to be little discussion in the literature on the performance impact of using such a feature. The reason may also be associated with the fact that compiler providers do not implement the entire specification of a language immediately [Ozen and Lopez 2020].

For example, in Stulajter et al. 2022, an evaluation using the most recent available versions of the Pgfortran, Ifort, and Gfortran compilers shows that only the first could generate CPU and GPU binaries using the DO CONCURRENT and its specific compiler flags. The experiments use DIFFUSE as a mini-app of flux-evolution code to generate observation-based model boundary conditions tests. DIFFUSE integrates the spherical surface heat equation considering a logically rectangular non-uniform grid, the discretization of the operator with a second-order central finite-difference scheme, and time integration with second-order Legendre polynomial extended stability Runge-Kutta scheme. The parallelization of the original application uses OpenACC and OpenMP for GPU and CPU, respectively. The performance of the DO CONCURRENT is comparable to the original

parallel code for both CPUs and GPUs, replacing directives without losing performance. The work demonstrated that the DO CONCURRENT allows cleaner-looking code and adds robustness. It contains nontrivial features for standard Fortran parallelism to handle reductions, atomics, CUDA-aware MPI, and local stack arrays.

Hammond et al. 2022 evaluates the Fortran DO CONCURRENT for CPU and GPU using the BabelStream benchmark. BabelStream is a rewrite of the traditional C++ standard memory bandwidth STREAM benchmark [McCalpin 2007] to modern Fortran code. The authors compare this implementation using the DO CONCURRENT command against CUDA and variants of OpenACC and OpenMP. They also consider loop- and array-based reference and evaluate the code on AArch64 and x86_64 CPUs and AMD, Intel, and NVIDIA GPU platforms.

We conduct our experiments based on Hammond et al. 2022 because it also intends to evaluate the DO CONCURRENT against traditional PPIs based on directives. However, the STREAM benchmark has only four small kernels (COPY, SCALE, SUM, and TRIAD), where operations are done directly in vector operations. In our work, we select a classical benchmark and a representative numerical program, a porous media application, to evaluate the impact of the DO CONCURRENT operations. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. A porous media application is a kind of Computational Fluid Dynamics (CFD) problem, one of the scenarios that mostly demands HPC [Versteeg and Malalasekera 2007]. Our approach is more connected with Stulajter et al. 2022 related work but also considers nontrivial features for standard Fortran parallelism to handle, such as reductions, local private variables, and stack arrays.

3. Parallel Programming: DO CONCURRENT

The addition of parallelism features in a standard language allows for accelerating GPU and CPU parallel programming. When the compiler is tasked with generating concurrency, the initial implementations of a specification may be incomplete. This is often because new features require thorough evaluation and refinement. The full support of this resource probably will stay in the future, increasing the portability of the code in terms of vendor support and duration of support. ISO-standard languages, like Fortran 2018, have a proven track record for stability and portability. The code will also look cleaner by excluding the lines where parallel directives appear.

DO CONCURRENT is an initial way to add concurrency support to code development. In this case, it is unnecessary to learn about directives. On the other hand, programmers need to think about parallelism when implementing the loop. It enables the parallel execution of loops in programs, making the usage of accelerated GPU and CPU parallel programming possible. One of the main advantages of DO CONCURRENT is its syntax, which is similar to the traditional loop syntax making parallelism exploitation easier. Furthermore, this structured integration into the language's syntax makes parallel code more readable and enhances its simplicity concerning external resources such as OpenMP and OpenACC.

The Fortran 2018 standard implemented *locality specifiers* to the DO CONCURRENT construct [ISO Central Secretary 2018]. The locality of a variable that appears in a DO CONCURRENT construct is LOCAL, LOCAL_INIT, SHARED, or unspecified. A

construct or statement entity of a construct or statement within the DO CONCURRENT construct has a SHARED locality if it has the SAVE attribute. If it does not have the SAVE attribute, it is a different entity in each iteration, similar to the LOCAL locality.

A variable that has LOCAL or LOCAL_INIT locality is a construct entity with the same type, type parameters, and rank as the variable with the same name in the innermost executable construct or scoping unit that includes the DO CONCURRENT construct. The outside variable is inaccessible by that name within the construct. The construct entity has the ASYNCHRONOUS, CONTIGUOUS, POINTER, TARGET, or VOLATILE attribute if, and only if, the outside variable has that attribute. It does not have the BIND, INTENT, PROTECTED, SAVE, or VALUE attributes, even if the variable outside has that attribute.

An entity has the same bounds as the outside variable if it is not a pointer. At the beginning of the execution of each iteration, if a variable has unspecified locality and if in an iteration it is referenced, it shall either be previously defined during that iteration or shall not be defined or become undefined during any other iteration. If it is defined or comes undefined by more than one iteration, it becomes when the loop terminates.

The programmer is responsible for ensuring the safe parallelization of loops, as the pgf90 (Nvfortran) may parallelize the loop even if there are data dependencies [Ozen and Lopez 2020]. For instance, incorrect results can be caused by addresses of inappropriate data dependencies.

4. Methodology

The methodology consists of the parallel implementation of the Conjugate Gradient of NAS-PB and a Porous Media model. That includes developing PARALLEL DO, OpenMP and OpenACC code versions. Moreover, we evaluate the execution performance of the implementations. We run 30 times for each implemented version to collect the average execution time.

The computational environment used in this work for running the tests is a workstation composed of two Intel Xeon CPU E5-2420 six-core processors (1.90GHz). The operating system is Ubuntu 22.04.1 with GNU/Linux kernel version 5.19.0-1025-oracle.

As pointed out in [Stulajter et al. 2022], currently, only Pgfortran (also currently known as Nvfortran) allows generating CPU and GPU parallelism for the DO CONCURRENT. Hence, we could use only this compiler for the experiments. We compile the code with pgf90 (nvfortran), using the hpc_sdk 23.5 NVidia toolkit, with the additional flag `-O3`, `Minfo=all`, and `-stdpar`. We add the tag `-fopenmp` for OpenMP and the flag `-acc` for OpenACC. We also consider `-mp=multi-core -stdpar=multi-core` for CPU and `-mp=gpu -stdpar=gpu` for GPUs.

We run the Conjugate Gradient with the NAS-PB class C, that is the number of rows of the matrix is 150,000 and 75 iterations. The algorithm was executed 100 times to determine the average execution time.

We run the fluid flow simulation for the tests discretizing the spatial domain into 100×124 mesh elements, that is 100 nodes in the x direction and 124 nodes in the y direction. We define a simulation time of 0.04. The discrete time step (Δt) was 0.01. The maximum number of iterations used for convergence in each discrete time was 20,000.

4.1. Conjugate Gradient - NAS-PB

The Conjugate Gradient algorithm is an iterative method used to solve large systems of linear equations, particularly those arising from discretized partial differential equations [Saad 2003]. It is especially effective for sparse, symmetric, and positive-definite matrices. The algorithm iteratively refines an estimate of the solution, making it suitable for problems where direct methods would be overly computationally expensive [Shewchuk 1994]. By utilizing gradients and conjugate directions, it converges faster than simpler methods like the gradient descent. Algorithm 1 describes the steps of the algorithm.

4.2. Porous Media Application

The coupled open-porous medium problem application is a discrete numerical simulation algorithm to grain drying. Algorithm 2 shows the called routines in the main iterative step, where external loops are time-dependent. Namely, a step depends on the previous step and can not be parallel, which is a characteristic of this domain of applications. Hence, the exploitation of parallelism occurred in invoked routines. In a previous experiment, considering sequential execution, `solve_U()` and `solve_V()` each one represents around 40% of the execution time of the application.

Figure 1-A shows the original sequence of subroutine calls for the application in the iterative step. Figure 1-B presents the new code structure for the analyzed part of the code. We rewrite the code to simplify the operations and to easily express the loop parallelism, maintaining the same numerical results. In this new structure, the `Solve_U` and `Solve_V` routines call three times `ResU` or `ResV`, `UpwindU()` or `UpwindV()`, and its respective boundary condition subroutine (`bcUV()`). `UpwindU()` and `UpwindV()` treat distinct operations of the four border side elements. `Solve_P` incorporates the `ResP()` operations and only calls the boundary conditions `bcP()`. `Solve_Z` incorporates all short subroutines `ResZ()` and boundary conditions `BcZ()`.

After the program modification, we identify 32 parallel loops are the main iterative step. We apply 10 parallel loop operations on `solve_U`, 10 for `solve_V`, 6 for `solve_P`, and 6 for `solve_Z`. We parallelize the loops using OpenMP, OpenACC, and the DO CONCURRENT approaches. We only do not parallelize the boundary operation loops due to the simplicity of the computations, that is, only attribution operations.

Algorithm 1 Steps of the Conjugate Gradient algorithm

1. *Compute* $r_0 := b - Ax_0, p_0 := r_0$.
2. *For* $j = 0, 1, \ldots$, *until convergence Do:*
3. $\quad \alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4. $\quad x_{j+1} := x_j + \alpha_j p_j$
5. $\quad r_{j+1} := r_j - \alpha_j Ap_j$
6. $\quad \beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7. $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
8. *EndDo*

Algorithm 2 The main iterative step of the porous media problem

```
1  DO WHILE (time .LT. final_time)
2      time = time + dt
3      DO WHILE(itc.LT.itc_max)
4          CALL solve_U()
5          CALL solve_V()
6          CALL solve_P()
7          CALL solve_Z()
8          CALL convergence()
9          ... !data updates
10     ENDDO
11     ... !data updates
12 ENDDO
```

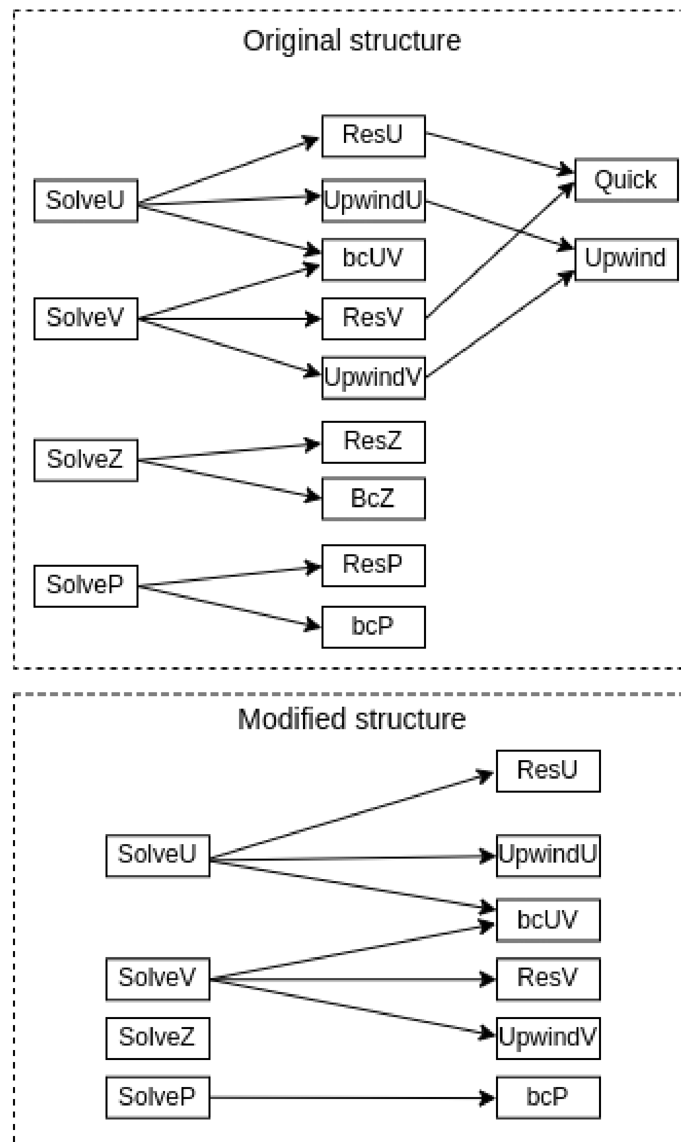

Figure 1. Code optimizations

Algorithm 3 A code region using `!$omp parallel do` directive

```
1  ...
2  !$omp parallel do private(i,j)
3  DO i=2,imax-1
4    DO j=2,jmax-1
5      res_p(i,j) = dtau * RP(i,j) * c2
6      pn(i,j) = 1.0d0 / 3.0d0 * p(i,j) +
7      2.0d0 / 3.0d0 * (pi(i,j) + res_p(i,j))
8    ENDDO
9  ENDDO
10 !$omp end parallel do
11 ...
```

5. Implementation

5.1. OpenMP Implementation

OpenMP is a parallel programming API for C/C++ and Fortran languages, widely used for shared memory parallelism, and it has its components, such as libraries, environment variables, and directives [OpenMP 2023]. Although it is necessary to indicate parallelism through directives, the popularity of OpenMP lies in the fact that the creation of concurrent executions is implicit [Chapman et al. 1998]. The programmer can even define the number of threads and how the distribution of data computation will be among the tasks. But usually, the vast majority adopts only loop parallelism without worrying much about the instantiated number of threads or the choice of parameters for processing distribution.

In our work, we first create parallel threads using the `!$omp parallel` directive, associated with `do` to split the computation. We set private variables for restricted data at each thread. It was also possible to generate binary code to CPU and GPU by setting flags to the compiler.

For the "OpenMP parallel do" approach, it was necessary to identify the inner loops of the application and include the appropriate pragma. Algorithm 3 shows an example of a parallel loop applied in an update of the pressure variable p for the porous media application.

5.2. OpenACC Implementation

OpenACC is an API based on directives for developing parallel applications on heterogeneous architectures, available for C/C++ and FORTRAN [Chandrasekaran and Juckeland 2017]. Those directives specify loops and code blocks that offload from the CPU to an attached accelerator [OpenACC 2023].

In our application, data is copied to the GPU from the CPU before the beginning of the iterative step using `!$acc enter data copy in (VARIABLES)`. Held data is in the GPU during the execution of the routines. The copy of the results to the CPU is for post-processing at the end of the iterative step using `!$acc exit data copy out (VARIABLES)`. The use of `!$acc data present(VARIABLES)` directive for each paralleled routine indicates the variables to be used and previously copied.

Algorithm 4 A code region using `!$acc parallel do` directive

```
1  !$acc data present (res_p, pn, p, pi c2)
2  ...
3  !$acc parallel do collapse(2) private(i,j)
4  DO i=2,imax-1
5    DO j=2,jmax-1
6      res_p(i,j) = dtau * RP(i,j) * c2
7      pn(i,j) = 1.0d0 / 3.0d0 * p(i,j) + 2.0d0 / 3.0d0 * (pi(i,
          j) + res_p(i,j))
8    ENDDO
9  ENDDO
10 !$acc end parallel
11 ...
12 !$acc end data
```

Algorithm 5 A code region using `DO CONCURRENT`

```
1  ...
2  DO CONCURRENT(j=2:jmax-1)
3        DO i=2,imax-1
4          res\_p(i,j) = dtau * RP(i,j) * c2
5          pn(i,j) = 1.0d0 / 3.0d0 * p(i,j) + 2.0d0 / 3.0d0 * (
              pi(i,j) + res\_p(i,j))
6        ENDDO
7  ENDDO
8  ...
```

The directive `!$acc parallel` defines the parallel execution. The composition of most routines is nested loops. So, we adopt the directive `!$acc do collapse(2)` because the mesh is bi-dimensional for the porous media application. Figure 4 shows a code snPPIet of our OpenACC implementation for the pressure update.

For the OpenACC implementations, it was necessary to indicate the routines that go run on GPU. For that, it is necessary to indicate `!$acc routine (name)` in the caller routine, where the name is the called routine.

5.3. DO CONCURRENT Implementation

In the DO CONCURRENT implementation, we change each one of the external `DO` command to the concurrent instruction. So, we trust that the compiler assigns all local variables correctly. Algorithm 5 shows a code snPPIet using `DO CONCURRENT`.

6. Experimental Results

6.1. Conjugate Gradient Results

Table 1 shows the execution time results. Surprisingly, the results indicated no significant difference between the two parallelization approaches. Both implementations exhibited similar numerical results, which demonstrates precise and equivalent calculations. Despite

Table 1. Conjugate Gradient Results

Implementation	Standard Deviation	Average Execution Time(s)
OpenMP	1.471	36.144
Do Concurrent	1.404	36.096

Table 2. Porous Media Application

Implementation	Standard Deviation	Average Execution Time(s)
OpenMP	7,77	116,203
Do Concurrent	3,53	105,798
OpenACC	6,56	111,301

minor variations in execution time, the overall outcomes were consistent across iterations. The standard deviations of execution times were 1.404 and 1.471 seconds, respectively. The lower standard deviation of Do Concurrent indicates slightly more consistent execution times.

6.2. Porous Media Application

Table 2 presents the average execution time in seconds of all versions for the mesh size configuration 100×124. The figure show the versions: OpenMP `parallel do`, DO CONCURRENT, and OpenACC `parallel loop`. We also calculate the standard deviation for these average times. We evaluate and guarantee numerical compatibility. That is, all codes need to produce identical values of results. In some cases, the values have a little difference of 10^{-10} concerning the sequential version.

The parallel OpenMP and DO CONCURRENT implementation running on the CPU provides execution time reduction. The sequential execution has a execution time of $605, 66s$. All experiments could use the maximum cores available(24). Both approaches obtain similar results, but DO CONCURRENT is slightly better. The speedup was respectively 5.21 and 5.51 for OpenMP and DO CONCURRENT. These results are compatible with the values found in related works ([Silva et al. 2022]).

The OpenACC result shows an execution time reduction compared to the CPU Sequential execution. The time is similar to the previous 24 CPU cores OpenMP and DO CONCURRENT execution.

7. Conclusion and Future Works

DO CONCURRENT is a language command provided in FORTRAN ISO 2018 standard, which is an alternative to facilitate the parallelism of an HPC application. DO CONCURRENT is also implemented in Pgfortran (Nvfortran) compiler [Hammond et al. 2022]. DO CONCURRENT provides cleaner code because there is no necessity to put pre-compilation directives.

In our evaluation of DO CONCURRENT, we use a more compound application, with the objective of not changing the source code, except for code writing optimizations. We point steps and code changes to usage the DO CONCURRENT in large applications. We provide parallel approach implementations: parallel OpenMP do, parallel OpenACC

do, and DO CONCURRENT for the CPU of NAS-PB Conjugate Gradient and for an open-porous-medium problem. The results show similar or even better results for the DO CONCURRENT than the OpenMP parallel do directive in CPU. These results demonstrate the potential of DO CONCURRENT to enable coding abstractions for introducing parallelism and also maintain high performance when running the applications.

We intend to provide a better OpenMP target implementation for GPUs in the future. The target approach allows the selection of the parallel environment at compilation time. Another experience includes combining the DO CONCURRENT Fortran language command with OpenACC to provide better speedup.

Acknowledgements

This work was partially funded by FAPERGS: 07/2021 PqG - project Nº 21/2551-0002055-5 and PROBIC program, and CNPq: 10/2023 Universal - Project Nº 407827/2023-4 and PIBIC program.

References

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 control. In *Proc. ECWMF Workshop" Towards Teracomputing-The Use of Parallel Processors in Meteorology*, Austrian. Citeseer, PSU.

da Silva, H. U., Lucca, N., Schepke, C., de Oliveira, D. P., and da Cruz Cristaldo, C. F. (2022). Parallel OpenMP and OpenACC Porous Media Simulation. *The Journal of Supercomputing*.

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.

ISO Central Secretary (2018). Information technology — Programming languages — Fortran — Part 1: Base language. Standard ISO/IEC 1539-1:2018, International Organization for Standardization, Geneva, CH.

Kennedy, K., Koelbel, C., and Zima, H. (2007). The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 7–1–7–22, New York, NY, USA. Association for Computing Machinery.

Kirk, D. B. and Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

Koelbel, C. H., Loveman, D., Schreiber, R. S., Jr., G. L. S., and Zosel, M. (1993). *High Performance Fortran Handbook*. The MIT Press.

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.

McCalpin, J. D. (1991-2007). Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia. A continually updated technical report. http://www.cs.virginia.edu/stream/.

OpenACC (2023). What is OpenACC?

OpenMP (2023). The OpenMP API Specification for Parallel Programming.

Ozen, G. (2018). *Compiler and runtime based parallelization & optimization for GPUs*. PhD thesis, Department of Computer Architecture - DAC Universitat Politècnica de Catalunya - UPC.

Ozen, G. and Lopez, G. (2020). Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK.

Reid, J. (2018). The new features of fortran 2018. *SIGPLAN Fortran Forum*, 37(1):5–43.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition.

Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, USA.

Silva, H. U., Schepke, C., da Cruz Cristaldo, C. F., de Oliveira, D. P., and Lucca, N. (2022). An Efficient Parallel Model for Coupled Open-Porous Medium Problem Applied to Grain Drying Processing. In Gitler, I., Barrios Hernández, C. J., and Meneses, E., editors, *High Performance Computing*, pages 250–264, Cham. Springer International Publishing.

Stulajter, M. M., Caplan, R. M., and Linker, J. A. (2022). Can Fortran's 'do concurrent' Replace Directives for Accelerated Computing? In Bhalachandra, S., Daley, C., and Melesse Vergara, V., editors, *Accelerator Programming Using Directives*, pages 3–21, Cham. Springer International Publishing.

Versteeg, H. K. and Malalasekera, W. (2007). *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Pearson Education.

Vetter, J. S. (2013). *Contemporary High Performance Computing: from Petascale Toward Exascale*. CRC Press.

Vogel, A., Griebler, D., and Fernandes, L. G. (2021). Providing High-level Self-adaptive Abstractions for Stream Parallelism on Multicores. *Software: Practice and Experience*, 51(6):1194–1217.