# High-performance Design Patterns in Modern Fortran

Damian W. I. Rouson[1], Karla Morris[2], Magne Haveraaen[3], Sameer Shende[4], and Jim Xia[5]

1. Center for Computational Earth and Environmental Sciences, Stanford University, Stanford, California, USA.
2. Combustion Research Facility, Sandia National Laboratories, Livermore, California, USA
3. Bergen Language Design Laboratory, University of Bergen, Bergen, Norway.
4. Performance Research Laboratory, University of Oregon, Eugene, Oregon, USA
5. IBM Canada Lab, Markham, Ontario, Canada.

## Abstract

High-performance computing (HPC) hardware is experiencing a flood of new technologies that promise higher performance and better energy utilization. Continuously adapting HPC codes for multicore processors and manycore accelerators while also adapting to new user requirements, however, drains human resources and prevents utilization of the more cost-effective hardware.

We present two design patterns in Fortran 2008 addressing these problems. Fortran's new coarray distributed data structures and the language's side-effect-free, pure function capability provide the scaffolding on which we implement our patterns. These features empower compilers to organize parallel computations with efficient communication. Our patterns support asynchronous evaluation of expressions comprised of parallel operations on distributed data. We demonstrate the patterns on tensor calculus expressions common in computational physics. A partial differential equation (PDE) solver illustrates the resulting performance profile.

## Introduction

Most useful software evolves over time under various selective pressures,including

1. Changes in the mathematical models employed.
2. The release of new HPC hardware.

This poster aims to demonstrate

1. A domain-specific, object-oriented software design pattern that eases the construction of tensor partial differential equation (PDE) solvers.
2. A general design pattern that supports the first pattern with asynchronous expression evaluation.
3. A language-specific pattern that supports the first two with platform-agnostic, parallel programming using Fortran 2008 coarrays [1].

Portable HPC software must execute efficiently on multicore processors, many-core accelerators, and heterogeneous combinations thereof. Fortran provides such portability by defining a partitioned global address space (PGAS) without referencing a particular parallel architecture. Fortran 2008 compilers may replicate a program across a set of *images* and map those images to the parallel architecture of the compiler's choice. The Intel compiler, for example, maps an image to a Message Passing Interface (MPI) process; whereas the Cray compiler uses a proprietary communication library that outperforms MPI on Cray computers.

For example, a coarray definition of the form

```
real, allocatable :: a(:,:,:)[:]
```

establishes that the program will index into the variable "a" along three dimensions and one codimension so

```
a(1,1,1)[1] = a(1,1,1)[2]
```

copies the first element of image 2 to the first element of image 1.

## Design Patterns

### Coordinate-Free Programming

One way to plan for software evolution involves designing around variation points, which are locations at which a program easily accommodates change. Some likely variation points in a PDE solver include the coordinate system, the numerical discretization, and the algorithms for solving the discrete equations. *Coordinate-free programming* (CFP), a structural design pattern, defines a layered set of mathematical abstractions at the scalar field level, the tensor level, the PDE solver level, and the parallelism level [2]. CFP provides for variation points at each level in the software stack.

To illustrate CFP, we start with the three-dimensional (3D) form of a PDE originally proposed in one dimension by Burgers [3]:

$$\vec{u}_{,t} = \nu\nabla^2\vec{u} - \vec{u}\cdot\nabla\vec{u} \qquad (1)$$

where a comma preceding a subscript denotes partial differentiation with respect to the subscripted variable. CFP maps the variables and operators in eq. (1) to analogous objects and user-defined operators in Fortran syntax. Such a mapping might result in program lines of the form

```
1 class(tensor) :: u_t, u
2 real, parameter :: nu=1.
3 u_t = nu*(.laplacian.u) - (u.dot.(.grad.u))
```

where we depict Fortran keywords in boldface. Line 1 declares `u` and `u_t` to be instances of a tensor class. Line 2 defines `nu` to be a real constant. Line 3 evaluates the right-hand side (RHS) of equation (1) using Fortran's facility for user-defined operators, which the language requires to be bracketed by periods (e.g., `.grad.`).

In three-dimensional (3D) space, the vector equation (1) resolves into three component equations. The first component equation in Cartesian coordinates, for example, takes the form

$$u_{1,t} = \nu(u_{1,xx} + u_{1,yy} + u_{1,zz}) - (u_1u_{1,x} + u_2u_{1,x} + u_3u_{1,x}) \qquad (2)$$

Figure 1 shows the calling sequence for evaluating the RHS of equation (2).

### Functional Programming

The growing gap between processor speeds and communication bandwidth necessitates avoiding interprocessor coordination. Synchronization is one form of coordination.

*Functional programming* composes programs from side-effect-free procedures. Side effects include input/output, modifying arguments, halting execution, modifying non-local data, and synchronizing with other images. Fortran provides a mechanism for communicating to the compiler that a procedure is free of side effects by giving the procedure the **`pure`** attribute. Fortran also explicitly proscribes one common side effect: arguments inside user-defined operators must have the **`intent(in)`** attribute, which precludes modifying operands. Fortran thereby encourages functional expression evaluation in the context of CFP.

Each image is free to execute each function in a purely functional calling sequence asynchronously. In Figure 1, each image executes 15 functions before encountering a synchronization point.

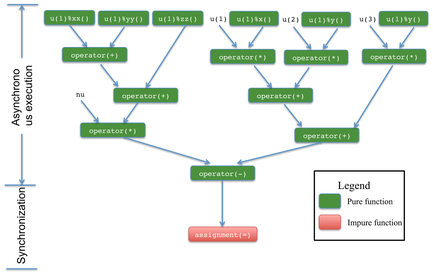


Figure 1. Calling sequence for asynchronous evaluation of the RHS of equation (2).

### Compute Globally, Return Locally

We distribute and communicate `tensor` data via a coarray component. We access data through generic, type-bound operators (e.g., "+") that we associate with parallel, type-bound procedures (e.g., "`add`"):

```
1 type tensor
2   real, allocatable :: global_data(:,:,:)[:]
3 contains
4   procedure :: add
5   generic :: operator(+)=>add
6   procedure :: copy
7   generic :: assignment(=)=>copy
8 end type
```

This poses a dilemma: user-defined operators are functions, but Fortran prohibits function results containing coarrays for performance-related reasons. Fortran compilers allocate the left-hand-side (LHS) of an array assignment to match the size of the RHS array, and memory allocations are implicit synchronization points.

In *Compute Globally, Return Locally*, a behavioral design pattern, each function in an expression accepts arguments containing coarrays. After performing any global communication necessary to compute its local partition of the result, each function stores its partition in a local object with a regular array component:

```
1 type local_tensor
2   real, allocatable :: local_data(:,:,:)
3 contains
4   procedure :: add
5   generic :: operator(+)=>add
6 end type
```

Ultimately, a generic, type-bound assignment associated with a type-bound copy procedure (lines 6-7 at left) transfers the local partitions' data into the global coarray and synchronizes with all images to make the newly assigned coarray data available to subsequent program lines globally.

## Results

We have investigated the feasibility of our approach using a published, one-dimensional (1D) coarray solver [4]. We modified the solver to ensure explicitly pure expression evaluation. Burgers originally proposed the 1D form as a simplified proxy for the Navier-Stokes equations (NSE) in studies of fluid turbulence:

$$u_{1,t} = \nu u_{1,xx} - u_1u_{1,x} \qquad (3)$$

Equation (3) retains the diffusive nature of the NSE in the first RHS term. It retains the nonlinear character of the NSE in the second RHS term. Equation (3) has also found useful application in several branches of physics, and it has the nice property of yielding to exact solution despite its nonlinearity [5].

Figure 2 shows the solution starting from an initial condition of *u(x,t=0)=10 sin(x)* with periodic boundary conditions on the domain [0,2π]. As time progresses, the nonlinear term steepens the initial wave while the diffusive term dampens it.

Figure 3 depicts the execution time profile of the dominant procedures as compiled with the Intel Fortran compiler and profiled by the Tuning and Analysis Utilities (TAU) package [6]. In constructing Figure 3, we emulated larger, multimensional problems by running with $128^3$ grid points on each of 256 cores. The results demonstrate nearly uniform load balancing. Other than the main program (red), the local-to-global field assignment occupies the largest inclusive share of the runtime. Most of that procedure's time lies in its synchronizations.

Figure 4 demonstrates 87% parallel efficiency for weak scaling of the solver for equation (3) up to 16,384 cores on a Cray XE6 supercomputer with one image per processor core.

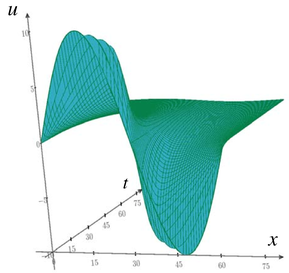


Figure 2. Unsteady, 1D Burgers equation solution values (vertical axis) over space (horizontal axis) and time (oblique

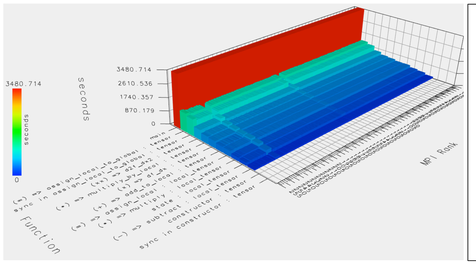


Figure 3. Runtime work distribution on all images. Each operator is shown in parenthesis, followed consecutively by the name of the type-bound procedure implementing the operator and the name of the corresponding module. The two points of synchronization are indicated by the word "sync" followed by the name of the type-bound procedure invoking the synchronization.

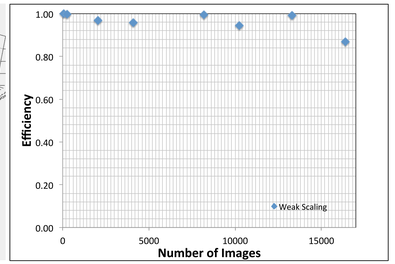


Figure 4. Parallel efficiency of the equation (3) solver in weak scaling versus the number of coarray images witih one image per core.

## Conclusions

As a feasibility study, we implemented each of the aforementioned techniques in a code that solves the one-dimensional Burgers equation where we use subscripts to indicate partial differentiation for x and t, space and time coordinates, respectively. Here we have outlined how to use domain specific scientific software patterns in the PDE domain. CFP provides abstractions aligned with the PDE variation points, resulting in flexibility and easy evolution of code. The functional expression style enhances readability of the code by its close resemblance to the mathematical notation. The CGRL behavioural pattern enables efficient use of Fortran coarrays with the functional expression evaluation. A profiled analysis of our application shows good load balancing, using the coarray enabled Fortran compilers from Intel and Cray. Performance analysis with the Cray compiler exhibited 87% parallel efficiency in weak scaling up to 16,384 cores.

## References

[1] M. Metcalf, J. Reid, and M. Cohen. *Modern Fortran Explained.* Oxford University Press, 2011.
[2] M. Haveraaen and H. A. Friis. Coordinate-free numerics: all your variation points for free? *International Journal of Computational Science and Engineering*, 4(4):223–230, 2009.
[3] J. M. Burgers. A mathematical model illustrating the theory of turbulence. *Adv. Appl. Mech.* (1), 25–27, 1948.
[4] D. W. I. Rouson, J. Xia, and X. Xu. *Scientific Software Design: The Object-Oriented Way.* Cambridge University Press, 2011.
[5] C. Canuto, M. Y. Hussaini, A. Quateroni, and T. A. Zang. *Spectral Methods: Fundamentals in Single Domains*, 2006.
[6] S. Shende and A. D. Malony, TAU Parallel Performance System, International Journal of High Performance Computing Applications, Vol. 20(2), pp. 287-311, Summer 2006.
[7] M. Haveraaen, K. Morris, and D. Rouson. High-performance design patterns for modern Fortran. In *SE-HPCSSE13*, November 22, 2013, Denver, CO, USA. ACM, 2013. http://dx.doi.org/10.1145/2532352.2532358.