High-Performance Design Patterns for Modern Fortran

Damian Rouson
CEES, Stanford University
Stanford, California, USA
cees.stanford.edu

Karla Morris
Combustion Research Facility
Sandia National Laboratories
Livermore, California, USA
crf.sandia.gov

Magne Haveraaen
University of Bergen
Bergen, Norway
www.ii.uib.no/~magne/

Jim Xia
IBM Canada Lab
Markham, Ontario, Canada

Sameer Shende
University of Oregon
Eugene, Oregon, USA
ix.cs.uoregon.edu/~sameer/

ABSTRACT

We discuss the use of the Fortran 2008 coarray feature for parallel programming, and how it integrates well with a more abstract, functional programming style.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture—*Patterns*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.1.5 [**Programming Techniques**]: Object-oriented programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design, Performance

Keywords

Scientific Computing, High Performance Computing, Fortran, Coarrays, Coordinate-Free Programming, Compute Globally – Return Locally

1. INTRODUCTION

Most useful software evolves over time. One selective pressure driving the evolution of high-performance computing (HPC) application software derives from the ever-evolving ecosystem of HPC hardware. A second pressure stems from the need to adapt to new user requirements. This poster demonstrates a domain-specific, object-oriented software design pattern that eases the evolution of partial differential equation (PDE) solver software based on vector and tensor calculus expressions. It also presents a design pattern for platform-agnostic, parallel programming using the coarray feature set of Fortran 2008. We also discuss the benefits of combining these patterns to achieve asynchronous expression evaluation.

SC13 Poster session, November 19, 2013, Denver, CO, USA

2. DESIGN PATTERNS

Programming patterns capture experience in how to express solutions to recurring programming issues in effective ways, effective from a development, evolution or even performance perspective. One way to plan for software evolution involves designing around variation points, which then become locations at which a program easily accommodates change.

What constitutes an intrinsic feature in one language might be implemented as a design pattern in another. Java provides an Object class that is the ultimate parent of all classes. Much like Java's Object, it can be useful to define a Fortran Object class that is the ultimate parent of all classes in a project. Such an Object can provide state and functionality that is universally useful throughout the project.

Patterns can also be more domain specific, e.g., scientific software in general [4]. Here we will look at patterns for high performance, parallel PDE solvers. Some likely variation points in a PDE solver include the coordinate system, the numerical discretization, the algorithms for solving the discrete equations, and what kind of parallelism to explore.

2.1 Coordinate-free programming

Coordinate-free programming (CFP) is a structural design pattern for PDEs that 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 naturally provides for variation points at each level in the software stack, aligned with the expectations above.

CFP starts with vector and tensor calculus PDEs such as the three-dimensional (3D) equation of Burgers [1]:

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

where subscripting represents partial differentiation so that $\vec{u}_t \equiv \partial\vec{u}/\partial t$. CFP represents each of the variables and operators in equation (1) to software objects and operators. In Fortran syntax, this might result in program lines of the form (using a functional notation)

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

where the first line declares that `u` and `u_t` are distributed objects in the `tensor` class or any class that extends `tensor`. The second line defines the parameter value corresponding to ν. The third evaluates the right-hand side of equation (1). Periods in the third

line bracket user-defined laplacian, dot product, and gradient operators.

2.2 Fortran 2008 Coarrays

Of particular interest in HPC are variation points at the parallelism level. Portable HPC software must allow for efficient execution on multicore processors, manycore accelerators, and heterogeneous combinations thereof. Fortran provides for such portability by supporting a single-program, multiple-data (SPMD) programming style without making reference to a particular parallel architecture. The Fortran 2008 standard mandates that compilers be able to replicate a program across a set of images, but the standard leaves compilers free to map those images to the parallel architecture of choice. The Intel compiler maps an image to a Message Passing Interface (MPI) process. The Cray compiler uses a proprietary communication library that outperforms MPI on Cray hardware. Mappings to GPUs can also be envisioned.

2.3 Compute Globally, Return Locally

Compute Globally, Return Locally (CGRL), a behavioural design pattern, resolves a dilemma that arises when one embeds coarrays inside the aforementioned `tensor` objects: each user-defined operator is a function, but Fortran prohibits function results that contain coarrays. In general, the return value of a function call is used in an outer expression, e.g., as the input argument of another function call. For coarray return values to be safe, each such return value would have to be synchronized, causing severe scalability and efficiency problems. The aforementioned prohibition precludes these issues.

In CGRL, each operator accepts operands that contain coarrays. The operator performs any global communication required to execute some parallel algorithm. On each image, the operator packages its local partition of the result in an object containing a regular array. Ultimately, when the operator of lowest precedence completes and each image has produced its local partition of the result, a user-defined assignment copies the local partitions into the global coarray and performs any necessary synchronizations to make the result available to subsequent program lines. The asymmetry between the argument and return types forces splitting large expressions into separate statements where such synchronization is needed.

Below is an excerpt from our code showing the use of this pattern. The `co_object` is a project specific superclass, equivalent to `object`, for all objects containing coarrays.

```
type, extends(co_object)
      :: periodic_2nd_order
  private
  real(rkind), allocatable
    :: global_f(:)[:]
contains
  ..
  procedure :: add_field
  generic   :: operator(+)   => add_field
  ..
end type
```

The line with generic allows us to use the + operator symbol in the code. The implementation of the `add_field` procedure has the object with the coarray as the first argument, and a local object with array data as its second argument and return type.

```
function add_field (this, rhs)
  class(periodic_2nd_order), intent(in)
    :: this
  class(field), intent(in) :: rhs
  class(field), allocatable :: add_field
  allocate (add_field)
  add_field = rhs%state()+this%global_f(:)
end function
```

The `rhs%state()` function call returns the local data array from the field, and this is then added to the local component of the coarray, using Fortran's array operator notation.

3. FUNCTIONAL PROGRAMMING

CGRL represents an object-oriented design choice. It is also desirable to embed information in the code that the compiler can exploit for optimizations outside the purview of the designer. An important class of optimizations become available to the compiler when it knows a procedure is free of side effects. The Fortran standard provides for communicating the lack of side effects to the compiler by giving a procedure the `pure` attribute. This communicates that the procedure will not modify its arguments, will not halt execution, will not modify objects declared outside the procedure's scope, and will not conduct input or output. Purely functional programming composes complete programs from pure procedures.

CFP naturally lends itself to purely functional programming in at least one respect: Fortran requires that the operands of user-defined operators have the `intent(in)` attribute, which precludes modifying them. It thus proves very natural to achieve purely functional expression evaluation.

4. DISCUSSION

As a feasibility study, we implemented each of the aforementioned techniques in a code that solves the one-dimensional Burgers equation:

$$u_t = \nu u_{xx} - u u_x, \tag{2}$$

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 evalution.

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 good weak scalability. See [3] for more details.

5. REFERENCES

[1] J. Burgers. A mathematical model illustrating the theory of turbulence. In R. V. Mises and T. V. Kármán, editors, *Advances in Applied Mechanics*, volume 1, pages 171 – 199. Elsevier, 1948.

[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] 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`.

[4] D. W. Rouson, J. Xia, and X. Xu. *Scientific Software Design: The Object-Oriented Way*. Cambridge University Press, 2011.