High-Performance Design Patterns for Modern Fortran

Magne Haveraaen
Dept. of Informatics
University of Bergen
Bergen, Norway
bldl.ii.uib.no

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

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

ABSTRACT

High-performance software, as all software, continuously evolves. Besides the normal changes in user requirements, e.g., the wish to solve a variation of a scientific problem, such software is also challenged by a flood of new technologies that promise higher performance and better energy utilization. Continuously adapting HPC codes for multicore processors and many-core accelerators while also adapting to new user requirements, however, drains human resources and prevents utilization of more cost-effective hardware.

Here we present some ideas for dealing with software variability in the PDE domain, namely the use of coordinate-free numerics for achieving flexibility. We also show how Fortran, over the last few decades, has changed to become a language well suited for state-of-the-art software development.

Fortran's new coarray distributed data structure, the language's class mechanism and side-effect-free, pure function capability provide the scaffolding on which we implement high performance software. These features empower compilers to organize parallel computations with efficient communication. We present some programming patterns that support asynchronous evaluation of expressions comprised of parallel operations on distributed data.

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

SE-HPCSSE13 November 22, 2013, Denver, CO, USA

ACM 978-1-4503-2499-1/13/11...$15.00.
http://dx.doi.org/10.1145/2532352.2532358

1. INTRODUCTION

Most useful software evolves over time. One force driving the evolution of high-performance computing (HPC) software applications derives from the ever-evolving ecosystem of HPC hardware. A second force stems from the need to adapt to new user requirements, where for HPC software the users often are the software development teams themselves. New requirements may come from a better understanding of the scientific domain, yielding changes in the mathematical formulation of a problem, changes in the numerical methods, changes in the problem to be solved, etc.

One way to plan for software evolution involves designing for variation points, areas where a program is expected to accommodate change. In a HPC domain like computational physics, partial differential equation (PDE) solvers are important. Some likely variation points for PDE solvers include the formulation of the PDE itself, like different simplifications depending on what phenomena is studied, the coordinate system and dimensions, the numerical discretization, and the hardware parallelism. The approach of coordinate-free programming (CFP) handles these variation points naturally through domain-specific abstractions [6]. The explicitg use of such abstractions are not common in HPC software, probably due to the historical development of the field.

Fortran has held, and still holds, a dominant position in HPC software. Traditionally, the language supported loops for traversing large data arrays, and had few abstraction mechanisms beyond the procedure. The focus was on efficiency and providing a simple data model that the compiler could map to efficient code.

In the past few decades, Fortran has evolved significantly [8], and now supports class abstraction, object-oriented programming (OOP), pure functions, and a coarray model for parallel programming in shared or distributed memory and running on multicore processors and some many-core accelerators.

This paper shows how domain abstractions like CFP can be accommodated within modern Fortran. It further explores how the new language features support efficient hardware-independent programming. This provides a setting that meets the challenges outlined above. We use Burgers equation as our running example.

The next section introduces the running problem and explains some of the ideas in CFP. The features of modern Fortran used by the solver are introduced in section 3. Section 4 presents programming patterns useful in this setting, and section 5 shows excerpts of code written according to our recommendations. Measurements of the approach's efficiency are presented in section 6, followed by our conclusions.

2. COORDINATE-FREE PROGRAMMING

Coordinate-free programming (CFP) is a structural design pattern for PDEs [7]. It is the result of domain engineering of the PDE

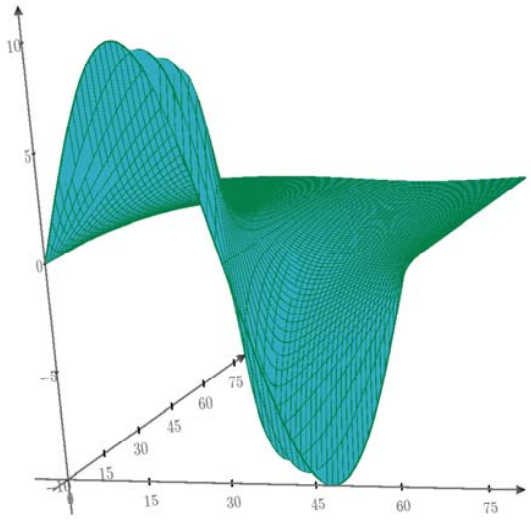

Figure 1: Unsteady, 1D Burgers equation solution values (vertical axis) over space (horizontal axis) and time (oblique axis).

domain. Domain engineering seeks to find the concepts central to a domain, then present these as reusable software components [1]. CFP defines a layered set of mathematical abstractions at the scalar field level (spatial discretization), the tensor level (coordinate systems), and the PDE solver level (time integration and PDE formulation). It also provides abstractions at the mesh level, encompassing abstraction over parallel computations. These layers correspond to the variation points of PDE solvers [6], both at the user level and for the ever changing parallel architecture level.

To see how this works, consider the coordinate-free generalization of the Burgers equation [2]:

$$\frac{\partial \vec{u}}{\partial t} = \nu \nabla^2 \vec{u} - \vec{u} \cdot \nabla \vec{u}. \quad (1)$$

CFP maps each of the variables and operators in equation (1) to software objects and operators. In Fortran syntax, such a mapping of equation (1) might result in program lines of the following form.

```
class(tensor) :: u_t, u
real :: nu=1.0
u_t = nu*(.laplacian.u) - (u.dot.(.grad.u))
```

Fortran keywords are depicted in boldface. The first line declares that u and u_t are (distributed) objects in the tensor class. The second line defines the parameter value corresponding to ν. The third line evaluates the right-hand side of equation (1) using Fortran's facility for user-defined operators, which the language requires to be bracketed by periods: laplacian (.laplacian.), dot product (.dot.), and gradient (.grad.). The mathematical formulation, and the corresponding program code, both are independent of dimensions, choice of coordinate system, discretisation method, etc. Yet the steps are mathematically, and computationally, precise.

Traditionally, the numerical scientist would expand Equation (1) into its coordinate form. Deciding that we want to solve the 3D problem, the vector equation resolves into three component equations. The first component equation in Cartesian coordinates, for example, becomes

$$u_{1,t} = \nu(u_{1,xx} + u_{1,yy} + u_{1,zz}) - (u_1 u_{1,x} + u_2 u_{1,x} + u_3 u_{1,x}). \quad (2)$$

Here subscripted commas denote partial differentiation with respect to the subscripted variable preceded by the comma, for instance, $u_{1,t} \equiv \partial u_1 / \partial t$. Similar equations must be given for $u_{2,t}$ and $u_{3,t}$.

For one-dimensional (1D) data equation (1) reduces to

$$u_{1,t} = \nu u_{1,xx} - u_1 u_{1,x}. \quad (3)$$

Burgers originally proposed the 1D form as a simplified proxy for the Navier-Stokes equations (NSE) in studies of fluid turbulence. Equation (3) retains the diffusive nature of the NSE in the first right-hand-side (RHS) term and the nonlinear character of the NSE in the second RHS term. This equation has also found useful applications in several branches of physics. It has the nice property of yielding an exact solution despite its nonlinearity [4].

Figure 1 shows the solution values (vertical axis) as a function of space (horizontal axis) and time (oblique axis) starting from an initial condition of $u(x, t = 0) = 10 \sin(x)$ with periodic boundary conditions on the semi-open domain $[0, 2\pi)$. As time progresses, the nonlinear term steepens the initial wave while the diffusive term dampens it.

3. MODERN FORTRAN

Fortran has always been a language with a focus on high efficiency for numerical computations on array data sets. Over the past 10-15 years, it has picked up features from mainstream programming, such as class abstractions, but also catered to its prime users by developing a rich set of high-level array operations. Controlling the flow of information allows for a purely functional style of expressions — that is expressions that rely solely upon functions that have no side effects. Side effects influence the global state of the computer beyond the function's local variables. Eamples of side effects include input/output, modifying arguments, halting execution, modifying non-local data, and synchronizing parallel processes.

There have been longstanding calls for employing functional programming as part of the solution to programming parallel computers [3]. The Fortran 2008 standard also includes a parallel programming model based primarily upon the coarray distributed data structure. The advent of support for Fortran 2008 coarrays in the Cray and Intel compilers makes the time ripe to explore synergies between Fortran's explicit support for functional expressions and coarray parallel programming[1].

3.1 Array Language

Since the Fortran 90 standard, the language has introduced a rich array feature set. This set also applies to coarrays, see section 3.4, in the 2008 standard. Fortran 90 contained operations to apply the built-in intrinsic operators, such as + and ∗, to corresponding elements of similarly shaped arrays, i.e., mapping them on the elements of the array. Modern Fortran also allows the mapping of user defined procedures on arrays. Such procedures have to be declared "**elemental**," which ensures that for every element of the array, the

[1]Two open-source compilers also provide limited support for coarrays: the g95 project supports coarrays in what is otherwise essentially Fortran 95 and the GNU Fortran (gfortran) project supports the coarray syntax but runs coarray code as sequential code. Ultimately, all compilers must support coarrays to maintain compliance with the Fortran standard.

invocations are independent of each other, and therefore can be executed concurrently. Operations for manipulating arrays also exist, e.g., slicing out a smaller array from a larger one, requesting upper and lower range of an array, and summing or multiplying all elements of an array.

This implies that in many cases it is no longer necessary to express an algorithm by explicitly looping over its elements. Rather a few operations on entire arrays are sufficient to express a large computation. For instance, the following array expressions, given an allocatable real array X, will in the first line take 1-rank arrays A, B, C, perform the elemental functions +, sin, and ∗ on the corresponding elements from each of the arrays, and pad the result with 5 numbers.

```
X = [sin(A+B)*C,0.,1.,2.,3.,4.,5.]
X = X(1:5)
```

In the second line, only the 5 first elements are retained. Thus for arrays A=[0.,0.5708], B=[0.5235988,1.], C=[3,5], the result is an array X =[1.5,5.,0.,1.,2.] .

3.2 Class Abstraction

Class abstractions allow us to associate a set of procedures with a private data structure. This is the basic abstraction mechanism of a programming language, allowing users to extend it with libraries for domain-specific abstractions. The Fortran notation is somewhat verbose compared to other languages, but gives great freedom in defining operator names for functions, both using standard symbols and introducing new named operators, e.g., .dot. as used above.

The Fortran class abstractions allow us to implement the CFP domain abstractions, such as scalar and tensor fields. Note that Fortran has very limited generic facilities. Fortran variables have three intrinsic properties: type, kind and rank. Fortran procedures can be written to be generic in kind, which allows, for example, for one implementation to work across a range of floating-point precisions. Fortran procedures can also be written to be generic in rank, which allows for one implementation to work acros a range of array ranks. Fortran procedures cannnot yet be generic in type, although there is less need for this than in languages where changing precision implies changing type. In Fortran, changing precision only implies changing kind, not type.

3.3 Functional Programming

A compiler can do better optimizations if it knows more about the intent of the code. A core philosophy of Fortran is to enable programmers to communicate properties of a program to a compiler without mandating specific compiler optimizations. In Fortran, each argument to a procedure can be given an attribute, **intent**, which describes how the procedure will use the argument data. The attribute "**in**" stands for just reading the argument, whereas "**out**" stands for just creating data for it, and "**inout**" allows for both reading and modifying the argument. A stricter form is to declare a function as "**pure**," i.e., indicating that the procedure harbors no side effects.

Purely functional programming composes programs from side-effect-free procedures and assignments. This allows the compiler to analyze each expression to find independent invocations of functions, and possibly execute them asynchronously. Figure 2 shows the calling sequence for evaluating the RHS of Equation (2) and assigning the result. Expressions in independent subtrees can be executed independently of each other, allowing for concurrency.

When developing abstractions like CFP, the procedures needed can be implemented as subroutines that modify one or more arguments, or as pure functions. Using pure functions makes the abstractions more mathematical and eases reasoning about the code.

3.4 Coarrays

Of particular interest in HPC are variation points at the parallelism level. Portable HPC software must allow for efficient execution on multicore processors, many-core accelerators, and heterogeneous combinations thereof. Fortran 2008 provides such portability by defining a partitioned global address space (PGAS), the *coarray*. This provides a single-program, multiple-data (SPMD) programming style that makes no reference to a particular parallel architecture. Fortran compilers may replicate a program across a set of *images*, which need to communicate when one image reaches out to a nonlocal part of the coarray. Images and communications are mapped 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[2]. Mappings to accelerators have also been announced.

For example, a coarray definition of the form

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

establishes that the program will index into the variable "a" along three dimensions (in parenthesis) and one codimension (in square brackets), so

```
if (this_image()==3) then
  a(1,1,1)[1] = a(1,1,1)[2]
end if
```

lets image 3, as given by the **this_image**() function, copy the first element of image 2 to the first element of image 1. If there are fewer than 3 images, the assignment does no take place. The size of the normal dimensions are decided by the programmer. The runtime environment and compiler decide the codimension. A reference to the array without the codimension index, e.g., a (1,1,1) , denotes the local element on the image that executes the statement. Equivalently, the expression "a (1,1,1)[**this_image**()]" makes the reference to the executing image explicit.

A dilemma arises when writing parallel operations on the aforementioned tensor object by encapsulating a coarray inside it: Fortran prohibits function results that contain coarrays. Performance concerns motivate this prohibition: in an expression, function results become input arguments to other functions. For coarray return values to be safe, each such result would have to be synchronized across images, causing severe scalability and efficiency problems. The growing gap between processor speeds and communication bandwidth necessitates avoiding interprocessor coordination.

To see the scalability concern, consider implementing the expression $(u * u)_x$ using finite differences with a stencil of width 1 for the partial derivative, with data u spread across images on a coarray. The part of the partial derivative function u_x executing on image i requires access to data from neighboring images $i + 1$ and $i - 1$. The multiplication $u * u$ will be run independently on each image for the part of the coarray residing on that image. Now for $(u*u)_x$ on image i to be correct, the system must ensure that $u*u$ on images $i - 1$, i and $i + 1$ all have finished computing and stored the result in their local parts of the coarray. Likewise for the computation of $(u*u)_x$ at images $i-1$ and $i+1$, then the computation of $u * u$ at images $i - 2$, $i - 1$, i and i, $i + 1$, $i + 2$, respectively,

[2]This fundamental difference limits the utility of comparing the two compilers: the native Cray compiler is always preferable on Cray hardware. Porting to different hardware to study performance with the Intel compiler is beyond the scope of this article.

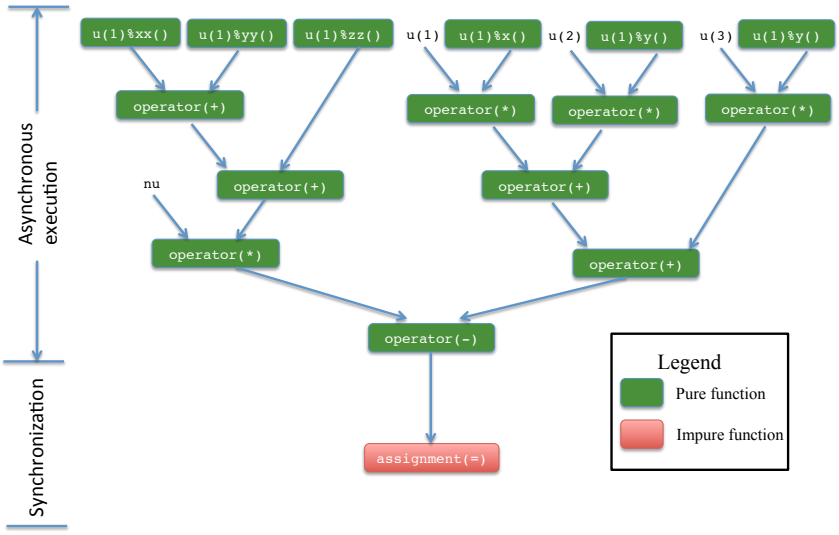

Figure 2: Calling sequence for evaluating the RHS of Equation (2) and assigning the result.

must be ready. Since the order of execution for each image is beyond explicit control, synchronization is needed to ensure correct ordering of computation steps.

Because analyzing whether and when synchronization is needed is beyond the compiler, the options are either synchronizing at return (with a possibly huge performance hit), or not synchronizing at return, risking hard to track data inconsistencies. The aforementioned prohibition precludes these issues, by placing the responsibility for synchronization with the programmer, yet allowing each image to continue processing local data for as long as possible. Consider the call graph in Figure 2. The only function calls requiring access to non-local data are the 6 calls to the partial derivatives on the top row. The remaining 9 function calls only need local data, allowing each image to proceed independently of the others until the assignment statement calls for a synchronization to prepare the displacement function u for the next time-step by assigning to $u_{1,t}$.

4. DESIGN PATTERNS

Programming patterns capture experience in how to express solutions to recurring programming issues in effective ways, effective from a software development, q software evolution, or even a performance perspective. Standard patterns tend to evolve into language constructs, the modern "**while**" statement evolved from a pattern with "**if**" and "**goto**" in early Fortran.

Patterns can also be more domain-specific, e.g., limited to scientific software [9]. Here we will look at patterns for high performance, parallel PDE solvers.

4.1 Object Superclass and Error Tracing

Many object-oriented languages, from the origins in Simula [5] and onwards, have an object class that is the ultimate parent of all classes. Fortran, like C++, does not have a universal base class. For many projects, though, 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. The object class itself is declared **abstract** to preclude constructing any actual objects of type object.

```
type, abstract :: object
  logical :: user_defined=.false.
contains
  procedure :: is_defined
  procedure :: mark_as_defined
end type
```

The object class above represents a solution to the problem of tracing assurances and reporting problems in pure functions. Assertions provide one common mechanism for verifying requirements and assurances. However, assertions halt execution, a prohibited side effect. The solution is to have the possible error information as extra data items in the objet class. If a problem occurs, the data can be set accordingly, and be passed on through the pure expressions until it ultimately is acted upon in a procedure where such side-effects are allowed, e.g., in an input/output (I/O) statement or an assignment.

The object class above allows tracking of the definedness of a variable declared to belong to the object class, or any of its subclasses. Such tracking can be especially useful when dynamically allocated structures are involved. The is_defined function returns the value of the user_defined component. The mark_as_defined subroutine sets the value of is_defined to .true.. Using this systematically in each procedure that defines or uses object data, will allow a trace of the source of uninitialized data.

A caveat is that due to the compilers' need to know if a variable has a coarray component or not, object cannot be a superclass of classes containing coarrays. We therefore need to declare a corresponding co_object class to be the superclass for classes with coarray components.

4.2 Compute Globally, Return Locally

The behavioural design pattern *Compute Globally, Return Locally* (CGRL) [9] has been suggested as a way to deal with the prohibition on returning coarrays from functions.

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 when synchronization is needed.

5. IMPLEMENTATION EXAMPLE

In this section we implement the functions needed to evaluate Equation (2), as illustrated in Figure 2. We follow the CGRL pattern: the derivation functions take a coarray data structure and return an array data structure, the multiplication then takes a coarray and an array data structure and return an array data structure, and the remaining operators work on array data structures. The assignment then synchronizes after assigning the local arrays to the corresponding component of the coarray.

To avoid cluttering the code extracts with error forwarding boiler plate, we first show code without this, then show how the code will look like with this feature in section 5.3.

5.1 Array Data Structure

First we declare a local_tensor class with only local array data. It is a subclass of object. The ampersand (&) is the Fortran line continuation character and the exclamation mark (!) precedes Fortran comments. The size of the data on each image is set by a global constant, the parameter local_grid_size .

```
  type, extends(object) :: local_tensor
    real :: f(local_grid_size)
  contains
  !...
  procedure :: add
  procedure :: assign_local
  procedure :: state
  procedure :: subtract
  generic   :: operator(+)    => add
  generic   :: operator(-)    => subtract
  generic   :: assignment(=) => assign_local
  !...
 end type
```

The **procedure** declarations list the procedures the class exports. The **generic** declarations introduce the operator symbols as synonyms for the procedure names. The four functions that are of interest to us are implemented below.

```
pure function add(lhs,rhs) result(total)
  class(local_tensor),intent(in) :: lhs,rhs
  type(local_tensor) :: total
  total%f = lhs%f + rhs%f
end function

pure subroutine assign_local (lhs, rhs)
  class(local_tensor),intent(inout) :: lhs
  real, intent(in) :: rhs(:)
  lhs%f = rhs
end subroutine

pure function state (this) result(my_data)
  class(local_tensor), intent(in) :: this
  real :: my_data(local_grid_size)
  my_data = this%f
end function

pure function subtract(lhs,rhs) &
  result(difference)
  class(local_tensor) ,intent(in) :: lhs, rhs
  type(local_tensor)  :: difference
  difference%f = lhs%f - rhs%f
end function
```

These are normal functions on array data. If executed in parallel, each image will have a local instance of the variables, and locally execute each function. Notice how we use the Fortran operators "+" and "−" directly on the array data structures in these computations.

5.2 Coarray Data Structure

The following is the declaration of a data structure distributed across the images.

```
type, extends(co_object) :: tensor
 private
 real, allocatable :: global_f(:)[:]
contains
 !...
 procedure :: assign_local_to_global
 procedure :: multiply_by_local
 procedure :: add_to_local
 procedure :: x              => df_dx
 generic   :: operator(*)    => &
              multiply_by_local
 generic   :: assignment(=) => &
              assign_local_to_global
 !...
end type
```

The coarray declaration allows us to access data on other images.

The partial derivative function takes a coarray data structure as argument and returns an array data structure. The algorithm is a simple finite difference that wraps around on the boundary. The processing differs depending on whether **this_image**() is the first image, an internal image, or the last image **num_images**(). An internal image needs access to data from the next image above or below. The extremal images do a wrap-around for their missing neighbors.

```
function df_dx(this)
 class(tensor), intent(in) :: this
 type(local_tensor) ::  df_dx
 integer :: i,nx, me, east, west
 real :: dx
 real :: local_tensor_data(local_grid_size)
  nx=local_grid_size
  dx=2.*pi/(real(nx)*num_images())

  me = this_image()

  if (me == 1) then
   west = num_images()
   east = merge(1,2,num_images()==1)
  else if (me == num_images()) then
   west = me - 1
   east = 1
  else
   west = me - 1
```

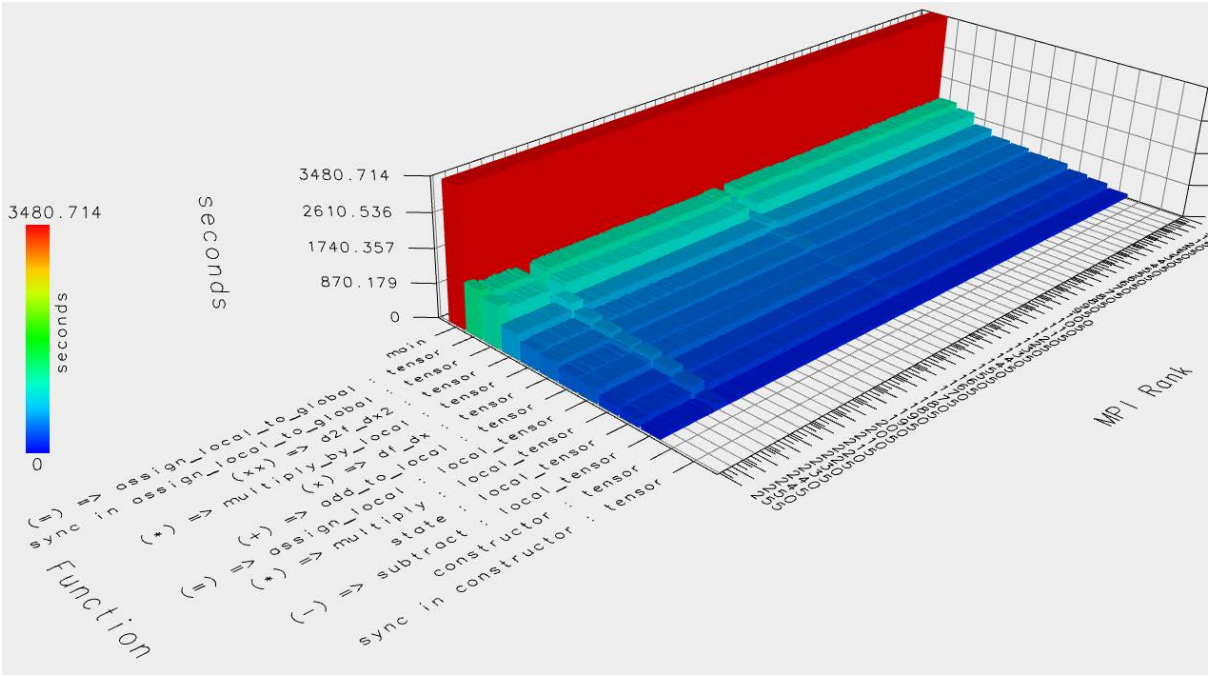

Figure 3: Runtime work distribution on all images. Each operator is shown in parenthesis, followed consecutively by the name of the type-bound procedure implementing the operator and the name of the corresponding module. The two points of synchronization are indicated by the word "sync" followed by the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

```
    east = me + 1
  end if

  local_tensor_data(1) = &
    0.5*(this%global_f(2)- &
    this%global_f(nx)[west])/dx

  local_tensor_data(nx) = &
    0.5*(this%global_f(1)[east]- &
    this%global_f(nx-1))/dx

  do concurrent (i=2:nx-1)
    local_tensor_data(i)=&
      0.5*(this%global_f(i+1)- &
      this%global_f(i-1))/dx
  end do

  df_dx = local_tensor_data
end function
```

In the tensor class, the local_tensor class is opaque, disallowing direct access to its data structure. Only procedures from the interface can be used. Note again how most of the computation is done by using intrinsics on array data. We also make use of the Fortran 2008 capability for expressing the opportunity for concurrently executing loop traversals when no data dependencies exist from one iteration to the next. The "**do concurrent**" construct exposes this concurrency to the compiler.

The partial derivative functions, the single derivative shown here and the second derivative (omitted), are the only procedures needing access to non-local data. Although a synchronization must take place before the non-local access, the requisite synchronization occurs in a prior assignment or object initialization procedure. Hence, the full expression evaluation generated by the RHS of Equation 2 occurs asynchronously, both among the images for the distributed coarray, and at the expression level for the pure functions.

The implementation of the add_to_local procedure has the object with the coarray as the first argument, and a local object with field data as its second argument and return type.

```
function add_to_local(lhs,rhs) result(total)
  class(tensor), intent(in) :: lhs
  type(local_tensor), intent(in) :: rhs
  type(local_tensor) :: total
  total = lhs%state()+rhs%global_f(:)
end function
```

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

Finally, the assignment operation synchronizes when converting the array class local_tensor back to the coarray class tensor.

```
subroutine assign_local_to_global(lhs,rhs)
  class(tensor), intent(inout) :: lhs
  class(local_tensor) ,intent(in) :: rhs
    ! Requires
    call assert(rhs%user_defined())
    ! update global field
    lhs%global_f(:) = rhs%state()
    ! Ensures
    call lhs%mark_as_defined
  sync all
end subroutine
```

After each component of the coarray has been assigned, the global barrier "**sync all**" is called, forcing all images to wait until all of them have completed the assignment. This ensures that all components of the coarray have been updated before any further use of the data takes place.

5.3 Error Tracing

The error propagating pattern is illustrated in the code below.

```
pure function add(lhs,rhs) result(total)
  class(local_tensor),intent(in) :: lhs,rhs
  type(local_tensor) :: total
  ! Requires
  if (lhs%user_defined() .and. &
      rhs%user_defined()) then
    total%f = lhs%f + rhs%f
    ! Ensures
    call total%mark_as_defined
  end if
end function
```

The *! Requires* test checks that both of the arguments to the add function have the definedness attribute set. It then performs the actual computation, and sets the definedness attribute for the return value. In case of an error in the input, the addition does not take place, and the default object value of undefined data gets propagated through this function.

The actual validation of the assurance and reporting of the error takes place in the user-defined assignment or I/O that occurs at the end of evaluation of a purely functional expression. The listing below shows this for the assign_local_to_global procedure.

```
subroutine assign_local_to_global(lhs,rhs)
  class(tensor), intent(inout) :: lhs
  class(local_tensor) ,intent(in) :: rhs
    ! Requires
    call assert(rhs%user_defined())
    ! update global field
    lhs%global_f(:) = rhs%state()
    ! Ensures
    call lhs%mark_as_defined
  sync all
end subroutine
```

More detailed error reporting can be achieved by supplying more meta-data in the object for such reporting purposes.

6. RESULTS

6.1 Pattern tradeoffs

This paper presents two new patterns: the Object and the CGRL patterns. The Object pattern is lightweight using simple boolean conditionals that improve the code robustness with negligible impact on execution time. The Object pattern is, however, heavyweight in terms of source-code impact: the pattern has every class extend the object superclass, and it encourages evaluating these conditionals at the beginning and end of every method. We found the robustness benefit to be worth the source-code cost.

The CGRL patttern is the linchpin holding together the functional expression evaluation with the language restriction forbidding coarray function results. The benefit of CGRL is partly syntactical in that it enables the writing of coordinate-free expressions composed of parallel operations on coarray data structures. CGRL also offers potential performance benefits as operators can be declared "**pure**". This enables compiler optimisations, e.g., allowing the compiler to execute multiple instances of the operator asynchronously. Furthermore, the "**pure**" attribute communicates to the compiler the absence of any other side effects, implying the existence of a one-to-one mapping from arguments to result. Therefore, if the compiler detects multiple invocations operating on the same data, the compiler can store the result once and substitute the result at the location of other invocations of the operator.

One cost of CGRL in the context of the CFP pattern lies in the frequent creation of temporary intermediate values. This is true of any functional programming style: precluding the modification of arguments inherently implies allocating memory on the stack or the heap for each operator result. This implies a greater use of memory. It also implies latencies associated with each memory allocation. Balancing this cost is a reduced need for synchronization and the associated increased opportunities for parallel execution. A detailed evaluation of this tradeoff requires writing an numerically equivalent code that exploits mutable data (modifiable arguments) to avoid temporary intermediate values. Such a comparison is beyond the scope of this paper.

6.2 Performance

We have investigated the feasibility of our approach using the one-dimensional (1D) form of Burgers equation, Equation (3). We modified the solver from [9] to ensure explicitly pure expression evaluation. The global barrier synchronization in the code excerpt above was replaced by synchronizing nearest neighbors only.

```
if (num_images()==1 .or. &
    num_images()==2) then
 sync all
else
 if (this_image()==1) then
  sync images([2,num_images()])
 elseif (this_image()==num_images()) then
  sync images([1,this_image()-1])
 else
  sync images([this_image()-1, &
               this_image()+1])
 endif
endif
```

Figure 3 depicts the execution time profile of the dominant procedures as measured by the Tuning and Analysis Utilities (TAU) package [10]. In constructing Figure 3, we emulated larger, multi-dimensional problems by running with 128^3 grid points on each of the 256 cores. The results demonstrate nearly uniform load balancing. Other than the main program (red), the local-to-global assignment occupies the largest inclusive share of the runtime. Most of that procedure's time lies in its synchronizations.

We also did a larger weak scaling experiment on the Cray. Here we emulate the standard situation where the user exploits the available resources to solve as large a problem as possible. Each core is assigned a fixed data size of 2 097 152 values for 3 000 time-steps, and the total size of the problem solved is then proportional to the number of cores available. The solver shows good weak scaling properties, see Figure 3, where it remains at 87% efficiency for 16 384 cores. We have normalized the plot against 64 cores. The Cray has an architecture of 24 cores per node, so our base measurement takes into account the cost due to off node communication.

Cores	Time	Efficiency
64	20414s	100.0%
256	20486s	99.6%
2 048	21084s	96.8%
4 096	21311s	95.8%
8 192	20543s	99.4%
10 240	21612s	94.5%
13 312	20591s	99.1%
16 384	23461s	87.0%

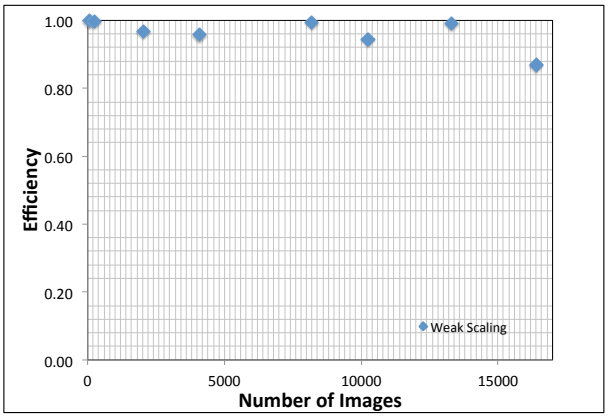

Figure 4: Weak scaling of solver for Equation (3).

Currently we are synchronizing for every time-step, only reaching out for a couple of neighboring values (second derivative) for each synchronization. We may want to trade some synchronization for duplication of computations. The technique is to introduce ghost values in the coarray, duplicating the values at the edge of the neighboring images. These values can then be computed and used locally without the need for communication or synchronization. The optimal number of ghost values depends on the relative speed between computation and synchronization. For example, using 9 ghost values on each side in an image, should reduce the need for synchronization to every 8th time-step, while it increases computation at each core by $\frac{18}{1283} = 1.4\%$. The modification should be local to the tensor class, only affecting the partial derivative (the procedures needing remote data) and assignment (the procedure doing the synchronization) procedures. We leave this as future work.

7. CONCLUSION

Motivated by the constant changing requirements on HPC software, we have presented *coordinate-free programming* [6] as an approach that naturally deals with the relevant variation points, resulting in flexibility and easy evolution of code. We then looked at the modern Fortran language features, such as pure functions and coarrays, and related programming patterns, specifically *compute globally, return locally* (CGRL), that make such programming possible.

As a feasibility study for the approach, we used these techniques in a code that solves the one-dimensional Burgers equation:

$$u_t = \nu u_{xx} - u u_x.$$

(Subscripts indicate partial differentiation for t and x, time and space coordinates, respectively.) 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 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 good weak scalability from 64 to above 16 000 cores.

Future work includes going from this feasibility study to a full coordinate-free implementation in Fortran of the general Burgers equation. This will allow us to study the behaviour of Fortran on such abstractions. We also want to increase the parallel efficiency by introducing ghost cells in the code, seeing how well modern Fortran can deal with the complexities of contemporary hardware architecture.

8. ACKNOWLEDGMENTS

Thanks to Jim Xia (IBM Canada Lab) for developing the Burgers 1D solver and Sameer Shende (University of Oregon) for help with TAU. This research financed in part by the Research Council of Norway. This research was also supported by Sandia National Laboratories a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the National Nuclear Security Administration under contract DE-AC04-94-AL85000. This work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work also used resources from the ACISS cluster at the University of Oregon acquired by a Major Research Instrumentation grant from the National Science Foundation, Office of Cyber Infrastructure, "MRI- R2: Acquisition of an Applied Computational Instrument for Scientific Synthesis (ACISS)," Grant #: OCI-0960354.

9. REFERENCES

[1] D. Bjørner. *Domain Engineering – Technology Management, Research and Engineering*, volume 4 of *COE Research Monograph Series*. JAIST, 2009.

[2] 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.

[3] D. C. Cann. Retire Fortran? A debate rekindled. *Commun. ACM*, 35(8):81–89, 1992.

[4] C. Canuto, M. Hussaini, A. Quateroni, and T. Zang. *Spectral Methods: Fundamentals in Single Domains*. Springer-Verlag, 2006.

[5] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA 67 Common Base Language*, volume S-2. Norwegian Computing Center, Oslo, 1968.

[6] 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.

[7] M. Haveraaen, H. A. Friis, and T. A. Johansen. Formal software engineering for computational modelling. *Nordic J. of Computing*, 6(3):241–270, 1999.

[8] M. Metcalf, J. Reid, and M. Cohen. *Modern Fortran Explained*. Oxford University Press, 2011.

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

[10] S. Shende and A. D. Malony. The tau parallel performance system. *IJHPCA*, 20(2):287–311, 2006. `http://tau.uoregon.edu`.