Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 942059, 14 pages
http://dx.doi.org/10.1155/2015/942059

Hindawi

Research Article

High-Performance Design Patterns for Modern Fortran

Magne Haveraaen,[1] Karla Morris,[2] Damian Rouson,[3] Hari Radhakrishnan,[4] and Clayton Carson[3]

[1] *Department of Informatics, University of Bergen, 5020 Bergen, Norway*
[2] *Sandia National Laboratories, Livermore, CA 94550, USA*
[3] *Stanford University, Stanford, CA 94305, USA*
[4] *EXA High Performance Computing, 1087 Nicosia, Cyprus*

Correspondence should be addressed to Karla Morris; knmorri@sandia.gov

Received 8 April 2014; Accepted 5 August 2014

Academic Editor: Jeffrey C. Carver

Copyright © 2015 Magne Haveraaen et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper presents ideas for using coordinate-free numerics in modern Fortran to achieve code flexibility in the partial differential equation (PDE) domain. 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 its side-effect-free, pure procedure capability provide the scaffolding on which we implement HPC 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. We implemented these patterns using coarrays and the message passing interface (MPI). We compared the codes' complexity and performance. The MPI code is much more complex and depends on external libraries. The MPI code on Cray hardware using the Cray compiler is 1.5–2 times faster than the coarray code on the same hardware. The Intel compiler implements coarrays atop Intel's MPI library with the result apparently being 2–2.5 times slower than manually coded MPI despite exhibiting nearly linear scaling efficiency. As compilers mature and further improvements to coarrays comes in Fortran 2015, we expect this performance gap to narrow.

1. Introduction

1.1. Motivation and Background. The 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, and so forth.

One way to plan for software evolution involves designing 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 [1]. The explicit use of such abstractions is not common in HPC software, possibly 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 [2] 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.

1.2. Related Work. CFP was first implemented in the context of seismic wave simulation [3] by Haveraaen et al. and Grant et al. [4] presented CFP for computational fluid dynamics applications. These abstractions were implemented in C++, relying on the language's template mechanism to achieve multiple levels of reuse. Rouson et al. [5] developed a "grid-free" representation of fluid dynamics, implementing continuous but coordinate-specific abstractions in Fortran 95, independently using similar abstractions to Diffpack [6]. While both C++ and Fortran 95 offered capabilities for overloading each language's intrinsic operators, neither allowed defining new, user-defined operators to represent the differential calculus operators, for example, those that appear in coordinate-free PDE representations. Likewise, neither language provided a scalable, parallel programming model.

Gamma et al. [7] first introduced the concept of patterns in the context of object-oriented software design. While they presented general design patterns, they suggested that it would be useful for subsequent authors to publish domain-specific patterns. Gardner et al. [8] published the first text summarizing object-oriented design patterns in the context of scientific programming. They employed Java to demonstrate the Gamma et al. general patterns in the context of a waveform analyzer for fusion energy experiments. Rouson et al. [9] published the first text on patterns for scientific programming in Fortran and C++, including several Gamma et al. patterns along with domain-specific and language-specific patterns. The Rouson et al. text included an early version of the PDE solver in the current paper, although no compilers at the time of their publication offered enough coverage of the newest Fortran features to compile their version of the solver.

The work of Cann [10] inspired much of our thinking on the utility of functional programming in parallelizing scientific applications. The current paper examines the complexity and performance of PDE solvers that support a functional programming style with either of two parallel programming models: coarray Fortran (CAF) and the message passing interface (MPI). CAF became part of Fortran in its 2008 standard. We refer the reader to the text by Metcalf et al. [2] for a summary of the CAF features of Fortran 2008 and to the text by Pacheco [11] for descriptions of the MPI features employed in the current paper.

1.3. Objectives and Outline. The current paper expands upon the first four author's workshop paper [12] on the CAF PDE solver by including comparisons to an analogous MPI solver first developed by the fifth author. We show how modern Fortran supports the CFP domain with the language's provision for user-defined operators and its efficient hardware-independent, parallel programming model. We use the PDE of Burgers [13] as our running theme.

Section 2 introduces the theme problem and explains CFP. Section 3 presents the features of modern Fortran used by the Burgers solver. Section 4 presents programming patterns useful in this setting, and Section 5 shows excerpts of code written according to our recommendations. Section 6 presents measurements of the approach's efficiency. Section 7 summarizes our conclusions.

2. Coordinate-Free Programming

Coordinate-free programming (CFP) is a structural design pattern for PDEs [3]. It is the result of domain engineering of the PDE domain. Domain engineering seeks finding the concepts central to a domain and then presenting these as reusable software components [14]. CFP defines a layered set of mathematical abstractions at the ring 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 [1], 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 [13]:

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

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

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 (1) using Fortran's facility for user-defined operators, in 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, and so forth. Yet the steps are mathematically and computationally precise.

Traditionally, the numerical scientist would expand (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

$$\begin{aligned} u_{1,t} &= \nu \left(u_{1,xx} + u_{1,yy} + u_{1,zz} \right) \\ &\quad - \left(u_1 u_{1,x} + u_2 u_{1,x} + u_3 u_{1,x} \right). \end{aligned} \tag{2}$$

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, (1) reduces to

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

Burgers originally proposed the 1D form as a simplified proxy for the Navier-Stokes equations (NSE) in studies of

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

Listing 1

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 [15].

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 semiopen domain $[0, 2\pi)$. As time progresses, the nonlinear term steepens the initial wave while the diffusive term dampens it.

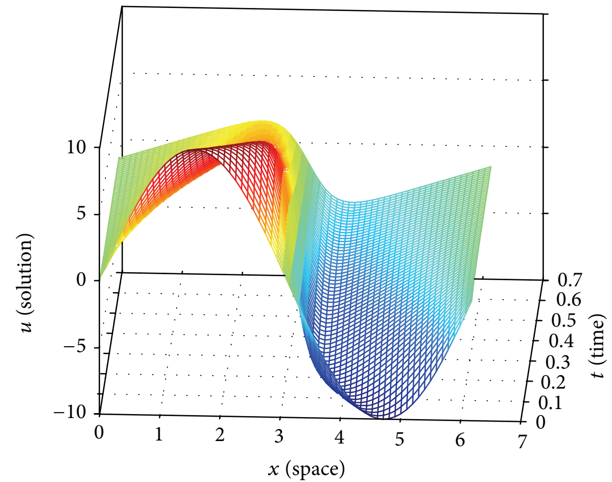


Figure 1: Unsteady, 1D Burgers equation solution values (vertical axis) over space (horizontal axis) and time (oblique axis). 1D Burgers equation solution surface: red (highest) and blue (lowest) relative to the $u = 0$ plane.

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. Examples of side effects include input/output, modifying arguments, halting execution, modifying nonlocal data, and synchronizing parallel processes.

There have been longstanding calls for employing functional programming as part of the solution to programming parallel computers [10]. 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. (Released versions of two free compilers also provide limited support for coarrays: g95 supports coarrays in what is otherwise essentially Fortran 95 and GNU Fortran (gfortran) supports the coarray syntax but runs coarray code as sequential code. Additionally, gfortran's prerelease development branch supports parallel execution of coarray code with communication handled by an external library (OpenCoarrays: http://www.opencoarrays.org) [16]. Ultimately, all compilers must support coarrays to conform to the Fortran standard.)

3.1. Array Language. Since the Fortran 90 standard, the language has introduced a rich set of array features. This set also applies to coarrays in the 2008 standard as we demonstrate in Section 3.4. Fortran 90 contained operations to apply the built-in intrinsic operators, such as + and $*$, to corresponding elements of similarly shaped arrays, that is, 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 invocations are independent of each other and therefore can be executed concurrently. Operations for manipulating arrays also exist, for example, 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, and C, perform the elemental functions +, **sin**, and $*$ on the corresponding elements from each of the arrays, and pad the result with 5 numbers:

$$X = [\mathbf{sin}(A + B) * C, 0., 1., 2., 3., 4., 5.];$$
$$X = X(1:5).$$

In the second line, only the 5 first elements are retained. Thus, for arrays $A = [0., 0.5708]$, $B = [0.5235988, 1.]$, and $C = [3, 5]$, 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, for example, . dot . as used above.

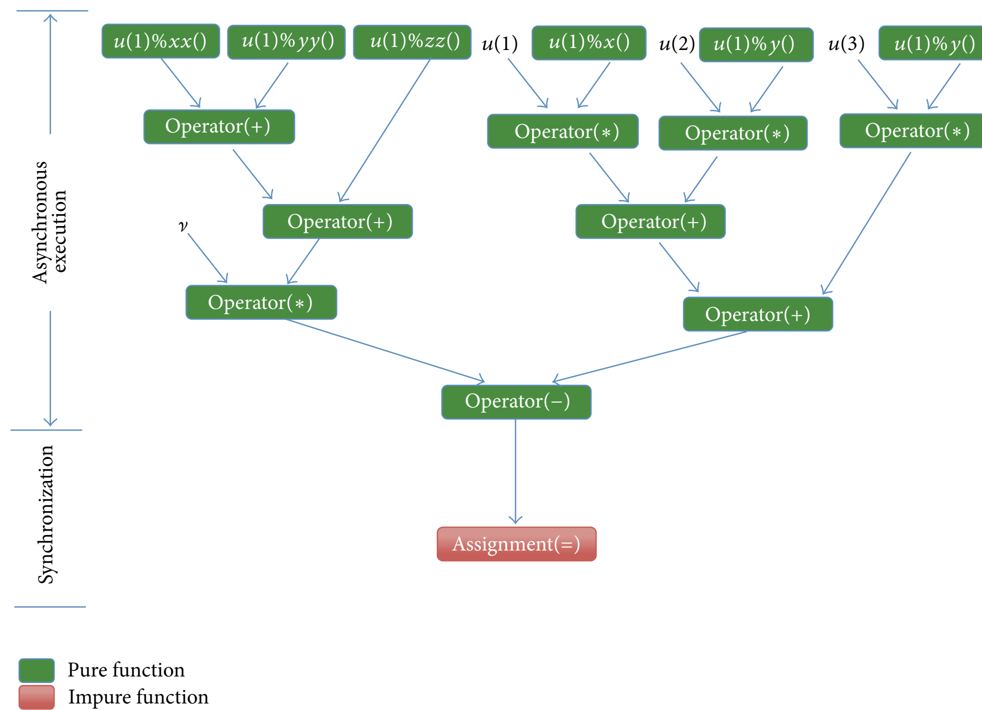


Figure 2: Calling sequence for evaluating the RHS of (2) and assigning the result.

The Fortran class abstractions allow us to implement the CFP domain abstractions, such as ring 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, one implementation to work across a range of floating-point precisions. Fortran procedures can also be written to be generic in rank, which allows one implementation to work across a range of array ranks. Fortran procedures cannnot yet be generic in type, although there is less need for this compared to 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 both reading and modifying the argument. A stricter form is to declare a function as "**pure**," for example, indicating that the procedure harbors no side effects.

Purely functional programming composes programs from side-effect-free procedures and assignments. This facilitates numerous optimizations, including guaranteeing that invocations of such procedures can safely execute asynchronously on separate partitions of the program data. Figure 2 shows the calling sequence for evaluating the RHS of (2) and assigning the result. Expressions in independent subtrees can be executed independently of each other, allowing 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. Mappings to accelerators have also been announced.

For example, a coarray definition of the form given in Listing 2 establishes that the program will index into the variable "a" along three dimensions (in parenthesis) and one codimension (in square brackets), so Listing 3 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 less

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

Listing 2

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

Listing 3

than 3 images, the assignment does not take place. The size of the normal dimensions is decided by the programmer. The run-time environment and compiler decide the codimension. A reference to the array without the codimension index, for example, 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 being 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 $i + 1$, the computation of $u * u$ at images $i - 2$, $i - 1$, and i and i, $i + 1$, and $i + 2$, respectively, 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 nonlocal 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 effectively from a software development, a 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, for example, 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 [17] 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 are universally useful throughout the project. The object class itself is declared **abstract** to preclude constructing any actual objects of type object.

The object class in Listing 4 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 object class. If a problem occurs, the data can be set accordingly and passed on through the pure expressions until it ultimately is acted upon in a procedure where such side-effects are allowed, for example, in an input/output (I/O) statement or an assignment.

The object class in Listing 4 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 user_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 the object class cannot be a superclass of classes containing coarrays because the compiler needs to know if a variable has a coarray component or not. 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.

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

Listing 4

In CGRL, each nonlocal 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 (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 excerpts with error-forwarding boiler plate, we first show code without this and then show how the code will look with this feature in Section 5.4.

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 (see Listing 5).

```
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
```

Listing 5

The procedure declarations list the procedures that 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 in Listing 6.

```
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
```

Listing 6

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. Listing 7 is the declaration of a data structure distributed across the images.

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 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 (see Listing 8).

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. These include a user-defined assignment implicitly invoked in the penultimate

```
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
```

Listing 7

line of the df_dx function. 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 nonlocal data. Although a synchronization must take place before the nonlocal access, the requisite synchronization occurs in a prior assignment or object initialization procedure. Hence, the full expression evaluation generated by the RHS of (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 (see Listing 9).

The rhs%state () function invocation 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 (see Listing 10).

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. Some situations might also necessitate a synchronization at the beginning of the assignment procedure: to prevent modifying data that another image might be accessing. Our chosen 2ndorder Runge Kutta time advancement algorithm did not require this additional synchronization because no RHS expressions contained nonlocal operations on the data structure appearing on the LHS.

5.3. MPI Data Structure. Developing applications using MPI necessitates depending on a library defined outside any programming language standard. This often results in procedural programming patterns instead of functional or object-oriented programming patterns. To make a fair comparison, we will employ a MPI data structure that uses the array data structure shown in Section 5.1. In the MPI version, the 1D grid was partitioned across the cores using a periodic Cartesian communicator, as shown in the code listing in Listing 11.

Using this communicator allowed us to reorder the processor ranks to make the communication more efficient by placing the neighbouring ranks close to each other. The transfer of data between the cores was done using MPI_SENDRECV, as shown in Listing 12. As in the case of the coarray version, nonlocal data was only required during the computation of the partial derivatives. The MPI version of the first derivative function is shown in Listing 12.

MPI_SENDRECV is a blocking routine which means that the processor will wait for its neighbor to complete communication before proceeding. This works as a de facto synchronization of the data between the neighbours ensuring that the data is current on all the processors. The c_double kind parameter used to declare the real variables in Listing 12 is related to the kind parameter MPI DOUBLE PRECISION in the MPI communication calls. These must be in sync, ensuring that the Fortran data has the same format as that used in MPI calls, viz. double precision real numbers that are compatible with C.

5.4. Error Tracing. The error propagating pattern is illustrated in the code in Listing 13.

The *!* Requires test in Listing 13 checks that the two 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 in Listing 14 shows this for the assign_local_to_global procedure.

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

6. Results

6.1. Pattern Tradeoffs. This paper presents two new patterns: the aforementioned object and the CGRL patterns. The object pattern proved to be lightweight in the sense of requiring 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 encourages having 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.

```
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
   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
```

Listing 8

```
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
```

Listing 9

```
subroutine assign_local_to_global(lhs, rhs)
 class(tensor), intent(inout):: lhs
 class(local_tensor), intent(in):: rhs
 lhs%global_f(:) = rhs%state()
 sync all
end subroutine
```

Listing 10

The CGRL pattern is the linchpin holding together the functional expression evaluation in the face of a performance-related language restriction on 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 by enabling particular compiler optimizations. Fortran requires that user-defined operator to have the "**intent** (**in**)" attribute, which precludes a common side effect: modifying arguments. This goes a long way toward enabling the declaration of the operator as "**pure,**" which allows the compiler to execute multiple instances of the operator asynchronously. One cost of CGRL in the context of the CFP pattern lies in the frequent creation of temporary intermediate values. This is true for most compilers that deal naively with the functional programming style, as 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 a numerically equivalent code that exploits mutable data (modifiable arguments) to avoid temporary intermediate values. Such a comparison is beyond the scope of this paper. More advanced approaches to compiling functional expressions exist, as demonstrated by the Sisal compiler [10]. It aggressively rearranges computations to avoid such memory overhead. Whether this is possible within the framework of current Fortran compilers needs to be investigated.

6.2. Performance. We have investigated the feasibility of our approach using the one-dimensional (1D) form of Burgers equation, (3). We modified the solver from [9] to ensure

```
subroutine mpi_begin
integer:: dims(1), periods(1), reorder
! prevent accidentally starting MPI
! if it has already been initiated
 if (program_status .eq. 0) then
   call MPI_INIT(ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, my_id, ierr)
   ! Create a 1D Cartesian partition
   ! with reordering and periodicity
   dims = num_procs
   reorder = .true.
   periods = .true.
   call MPI_CART_CREATE(MPI_COMM_WORLD, 1, dims, periods, reorder, MPI_COMM_CART, ierr)
   call MPI_COMM_RANK(MPI_COMM_CART, my_id, ierr)
   call MPI_CART_SHIFT(MPI_COMM_CART, 0, 1, left_id, right_id, ierr)
   program_status = 1
  endif
end subroutine
```

Listing 11

```
function df_dx(this)
   implicit none
   class(tensor), intent(in):: this
   type(tensor):: df_dx
   integer(ikind):: i, nx
   real(c_double):: dx, left_image, right_image
   real(c_double), dimension(:), allocatable, save:: local_tensor_data
      nx = local_grid_resolution
      if (.not.allocated(local_tensor_data)) allocate(local_tensor_data(nx))
      dx = 2. * pi/(real(nx, c_double) * num_procs)
      if (num_procs > 1) then
         call MPI_SENDRECV(this%global_f(1), 1,
               MPI_DOUBLE_PRECISION, left_id, 0, right_image, 1,
               MPI_DOUBLE_PRECISION, right_id, 0, MPI_COMM_CART,
               status, ierr)
         call MPI_SENDRECV(this%global_f(nx), 1,
               MPI_DOUBLE_PRECISION, right_id, 0, left_image, 1,
               MPI_DOUBLE_PRECISION, left_id, 0, MPI_COMM_CART,
               status, ierr)
      else
         left_image = this%global_f(nx)
         right_image = this%global_f(1)
      end if
      local_tensor_data(1) = 0.5 * (this%global_f(2) − left_image)/dx
      local_tensor_data(nx) = 0.5 * (right_image − 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%global_f = local_tensor_data
end function
```

Listing 12

```
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
```

Listing 13

```
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
```

Listing 14

```
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
```

Listing 15

explicitly pure expression evaluation. The global barrier synchronization in the code excerpt above was replaced by synchronizing nearest neighbors only (see Listing 15).

Figure 3 depicts the execution time profile of the dominant procedures as measured by the tuning and analysis utilities (TAU) package [18]. In constructing Figure 3, we emulated larger, multidimensional 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 4, 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.

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 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.

We also looked at the strong scaling performance of the MPI and coarray versions by looking at change in execution times for a fixed problem size. The strong scaling efficiency for two different problem sizes is shown in Figures 5(a) and 5(b). We expect linear scaling; that is, the execution time will halve when the number of processors are doubled. However, we see that we obtain superlinear speedup during the initial doubling of the number of processors. This superlinear speedup is caused by the difference in speeds of the cache memory. The large problems cannot fit entirely into the heap, and time is consumed in moving objects from the slower memory to the faster memory. As the problem is divided amongst more and more processors, the problem's memory requirements become smaller, and is able to fit in the faster memory that is closer to the processor. This causes the superlinear speedup. As more processors are added, communication between processors starts to become expensive, and the speedup drops. We observe superlinear speedup for both coarray and MPI versions. However, the much greater speedup seen for the coarray version suggest that its memory requirements are higher than those of the MPI version. (These numbers may be slightly misleading, as the MPI version used dynamically allocated data, while the CAF version used statically allocated data. This may cause the CAF version to use more memory than the MPI version. Fixing this will cause minor changes in the numbers and close

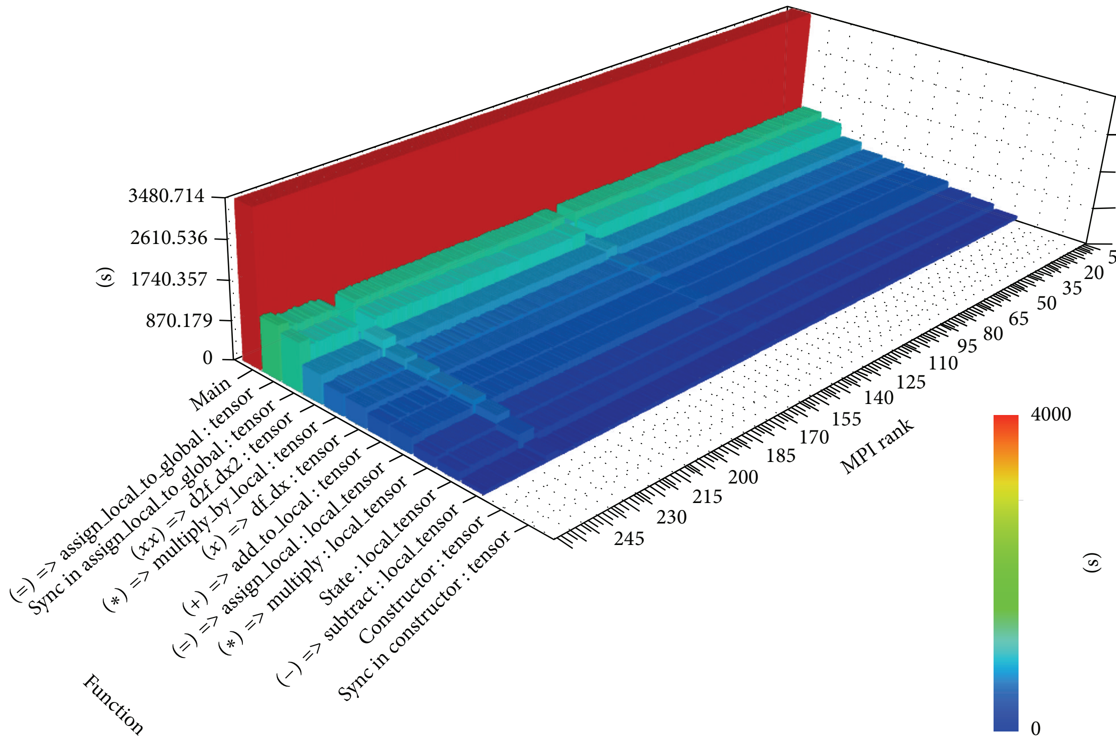
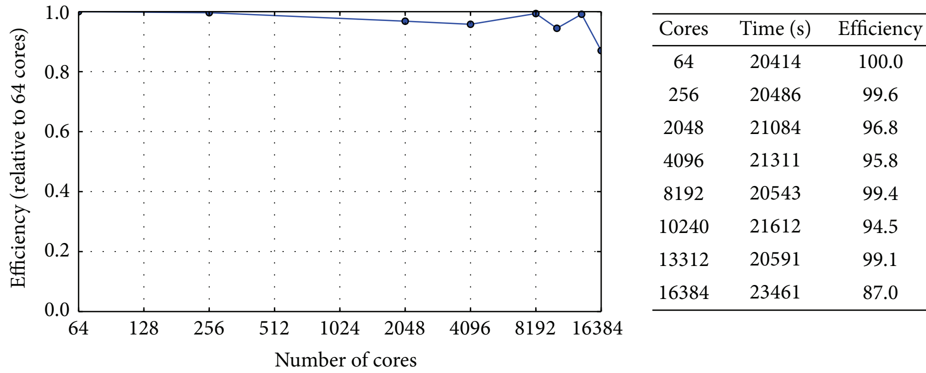


Figure 3: Runtime work distribution on all images. Each operator is shown in parenthesis, followed consecutively by the name of the type-bound procedure implementing the operator and the name of the corresponding module. The two points of synchronization are indicated by the word "sync" followed by the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



Cores	Time (s)	Efficiency
64	20414	100.0
256	20486	99.6
2048	21084	96.8
4096	21311	95.8
8192	20543	99.4
10240	21612	94.5
13312	20591	99.1
16384	23461	87.0

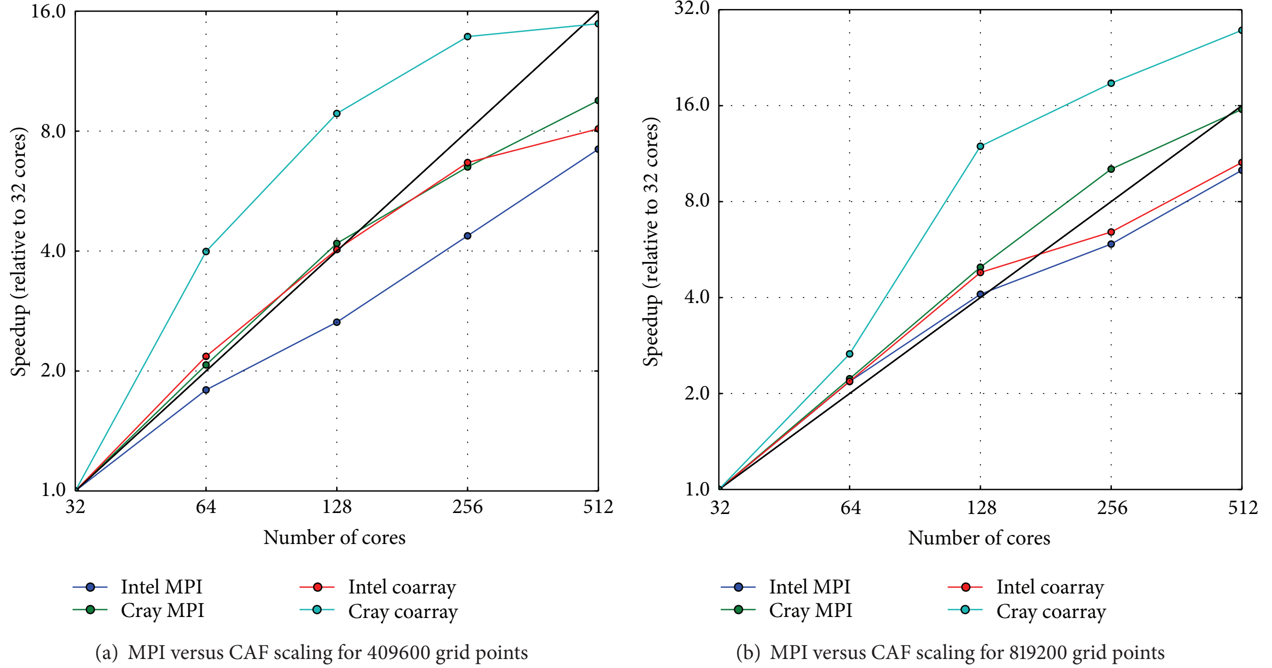
Figure 4: Weak scaling of solver for (3) using the coarray version on Cray.

the ratio between the MPI and CAF efficiency. We will have these numbers available for the revision of this document.)

The raw execution times using the different versions on Intel and Cray platforms are shown in Table 1. We chose a smaller problem for the strong scaling experiments than for the weak scaling experiments because of the limited resources available with the Intel platform. We see that the coarray version is slower than the MPI version on the same platform for the same problem size. Comparing the actual runtimes shown in Table 1 shows that using the Intel compiler, the MPI version is about 2 to 2.5 times faster than the coarray version. For the Cray compiler, the MPI version is about 1.5 to 2 times faster than the coarray version. To understand the difference in runtimes, we analyzed the CAF and MPI versions using TAU and the Intel compiler. Using PAPI [19] with TAU and the Intel compiler to count the floating-point operations, we see that the MPI version is achieving approximately 52.2% of the peak theoretical FLOPS for a problem with 819200 grid points using 256 processors whereas the CAF version is achieving approximately 21% of the peak theoretical FLOPS. The execution times for some of the different functions are shown in Figure 6. We see that the communication routines are taking the longest fraction of the total execution time. However, the coarray syncing is taking significantly longer than the MPI_SENDRECV blocking operation. The Intel coarray implementation is based on its MPI library, and

Table 1: Execution times for the CAF and MPI versions of the Burgers solver for different problem sizes using Intel and Cray compilers.

Cores	409600 grid points				819200 grid points			
	MPI		CAF		MPI		CAF	
	Intel	Cray	Intel	Cray	Intel	Cray	Intel	Cray
32	52.675	59.244	154.649	187.204	128.638	131.048	333.682	512.312
64	29.376	28.598	71.051	46.923	58.980	58.887	152.829	192.396
128	19.864	14.169	38.321	21.137	31.396	26.318	69.612	42.939
256	12.060	9.109	23.183	13.553	21.852	12.953	51.957	27.226
512	7.308	6.204	19.080	12.581	12.818	8.413	31.437	18.577



(a) MPI versus CAF scaling for 409600 grid points

(b) MPI versus CAF scaling for 819200 grid points

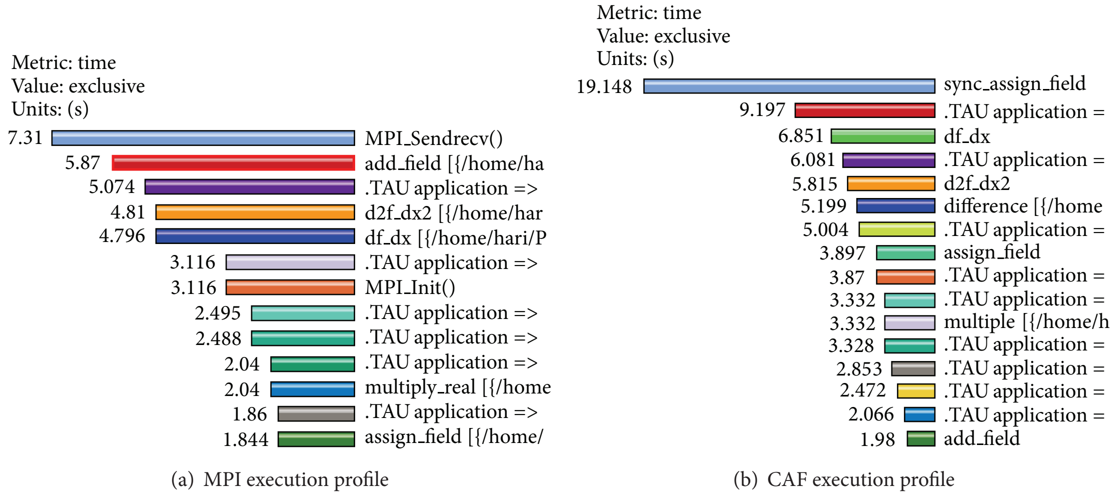
Figure 5: Strong scaling performance of the coarray and MPI versions of the solver for (3) using different platforms. The raw execution times are listed in Table 1.

the overheads of the coarray implementation are responsible for some of the slowdown. The greater maturity of the MPI library compared to CAF also probably plays a role in the superior performance of the MPI implementation. So, we are likely to see the performance gap lessen as compiler implementations of CAF improve over time.

6.3. Complexity. Other than performance considerations, we also wanted to compare the pros and cons of the coarray Fortran (CAF) implementation versus an MPI implementation of the 1D form of the Burgers equation (3) in terms of code complexity and ease of development.

The metrics used to compare the code complexity were lines of code (LOC), use statements, variable declarations, external calls, and function arguments. The results of this comparison may be found in Table 2. As seen in Table 2, the MPI implementation had significantly greater complexity compared to the CAF implementation for all of the metrics which were considered. This has potential consequences in terms of the defect rate of the code. For example, comparing the MPI version with the coarray version listed in Section 5.2, we see that the basic structures of the functions are almost identical. However, the MPI_SENDRECV communication of the local grid data to and from the neighbours is achieved implicitly in the coarray version making the code easier to read. Counterbalancing its greater complexity, the MPI implementation had superior performance compared to the CAF code.

Software development time should also be taken into account when comparing CAF to MPI. Certain metrics of code complexity have been shown to correlate with higher defect rates. For instance, average defect rate has been shown to have a curvilinear relationship with LOC [20]. So, an MPI implementation might drive higher defect density and overall number of defects in a project, contributing to development time and code reliability. Likewise, external calls or fanout has shown positive correlation with defect density, also reducing the relative attractiveness of the MPI implementation [21]. In addition, the dramatically increased number of arguments for function calls, as well as the larger number of functions



(a) MPI execution profile

(b) CAF execution profile

Figure 6: Execution profiles of MPI and CAF versions of Burgers solver.

Table 2: Code complexity of CAF versus MPI.

Metric	CAF	MPI
LOC	238	326
Use statements	3	13
Variables declared	58	97
External calls	0	24
Function arguments	11	79

which are used in the MPI implementation, suggests a higher learning curve for novice parallel programmers compared to CAF.

7. Conclusion

Motivated by the constant changing requirements on HPC software, we have presented coordinate-free programming [1] 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), which make such programming possible. We also looked at implementing coordinate-free programming using MPI and the advantages and disadvantages of the MPI implementation vis-a-vis using only modern Fortran language features.

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

$$u_t = \nu u_{xx} - u u_x. \tag{4}$$

(Subscripts indicate partial differentiation for t and x, time and space coordinates, resp.) 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. Strong scaling studies using MPI and coarray versions of our application show that while the runtimes of the coarray version lag behind the MPI version, the coarray version's scaling efficiency is on par with the MPI version.

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.

Disclosure

This is an extended version of a workshop paper presented at SE-HPCSSE13 in Denver, CO, USA.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

Thanks are due to Jim Xia (IBM Canada Lab) for developing the Burgers 1D solver and Sameer Shende (University of Oregon) for help with TAU. This research is financed in part by the Research Council of Norway. This research was also supported by Sandia National Laboratories, a multiprogram 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 US 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 no. OCI-0960354.

References

[1] M. Haveraaen and H. A. Friis, "Coordinate-free numerics: all your variation points for free?" *International Journal of Computational Science and Engineering*, vol. 4, no. 4, pp. 223–230, 2009.

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

[3] M. Haveraaen, H. A. Friis, and T. A. Johansen, "Formal software engineering for computational modelling," *Nordic Journal of Computing*, vol. 6, no. 3, pp. 241–270, 1999.

[4] P. W. Grant, M. Haveraaen, and M. F. Webster, "Coordinate free programming of computational fluid dynamics problems," *Scientific Programming*, vol. 8, no. 4, pp. 211–230, 2000.

[5] D. W. Rouson, R. Rosenberg, X. Xu, I. Moulitsas, and S. C. Kassinos, "A grid-free abstraction of the Navier-Stokes equations in Fortran 95/2003," *ACM Transactions on Mathematical Software*, vol. 34, no. 1, article 2, 2008.

[6] A. M. Bruaset and H. P. Langtangen, "A comprehensive set of tools for solvingpartial differential equations; Diffpack," in *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito, Eds., pp. 61–90, Birkhäuser, Boston, Mass, USA, 1997.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, 1994.

[8] H. Gardner, G. Manduchi, T. J. Barth et al., *Design Patterns for E-Science*, vol. 4, Springer, New York, NY, USA, 2007.

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

[10] D. C. Cann, "Retire Fortran? A debate rekindled," *Communications of the ACM*, vol. 35, no. 8, pp. 81–89, 1992.

[11] P. S. Pacheco, *Parallel programming with MPI*, Morgan Kaufmann, 1997.

[12] M. Haveraaen, K. Morris, and D. Rouson, "High-performance design patternsfor modern fortran," in *Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pp. 1–8, ACM, 2013.

[13] J. Burgers, "A mathematical model illustrating the theory of turbulence," in *Advances in Applied Mechanics*, R. V. Mises and T. V. Kármán, Eds., vol. 1, pp. 171–199, Elsevier, New York, NY, USA, 1948.

[14] D. Bjørner, *Domain Engineering: Technology Management, Research and Engineering*, vol. 4 of *COE Research Monograph Series*, JAIST, 2009.

[15] C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. Zang, *Spectral Methods: Fundamentals in Single Domains*, Springer, Berlin, Germany, 2006.

[16] A. Fanfarillo, T. Burnus, S. Filippone, V. Cardellini, D. Nagle, and D. W. I. Rouson, "OpenCoarrays: open-source transport layers supporting coarray Fortran compilers," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*, Eugene, Ore, USA, October 2014.

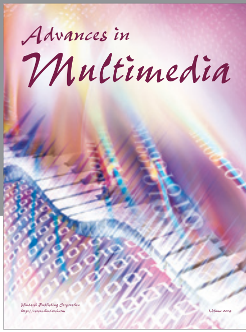
[17] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA 67 Common Base Language*, vol. S-2, Norwegian Computing Center, Oslo, Norway, 1968.

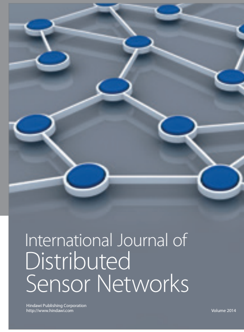
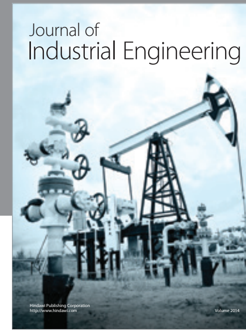
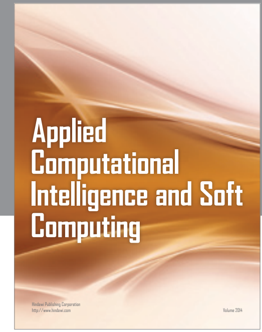
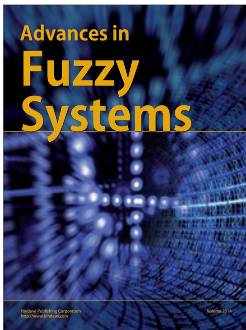
[18] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

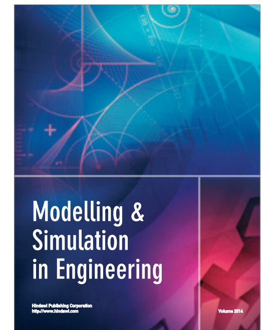
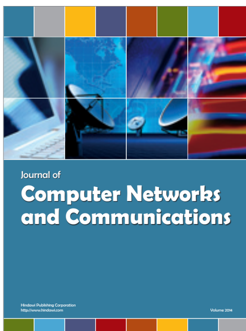
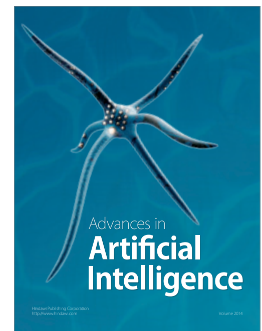
[19] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using PAPI for hardware performance monitoring on linux systems," in *Conference on Linux Clusters: The HPC Revolution*, Linux Clusters Institute, 2001.

[20] C. Withrow, "Error density and size in Ada software," *IEEE Software*, vol. 7, no. 1, pp. 26–30, 1990.

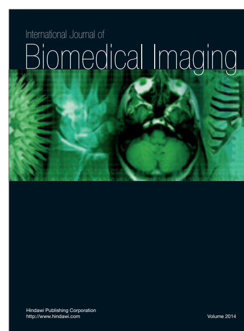
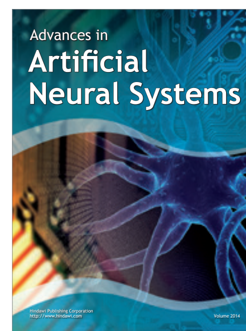
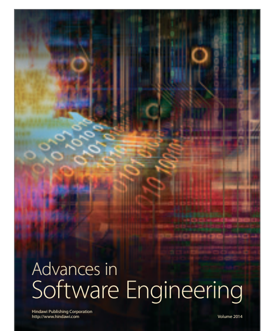
[21] S. H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley, New York, NY, USA, 2nd edition, 2002.

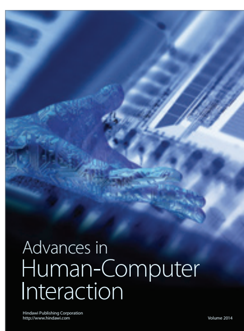
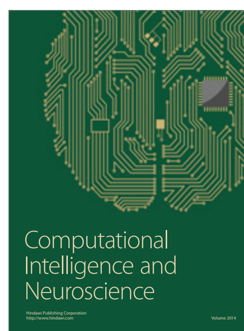

Advances in
Multimedia

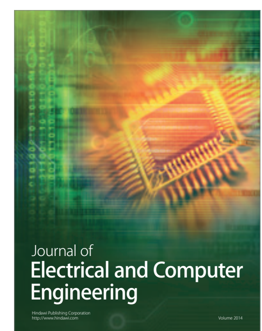
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering