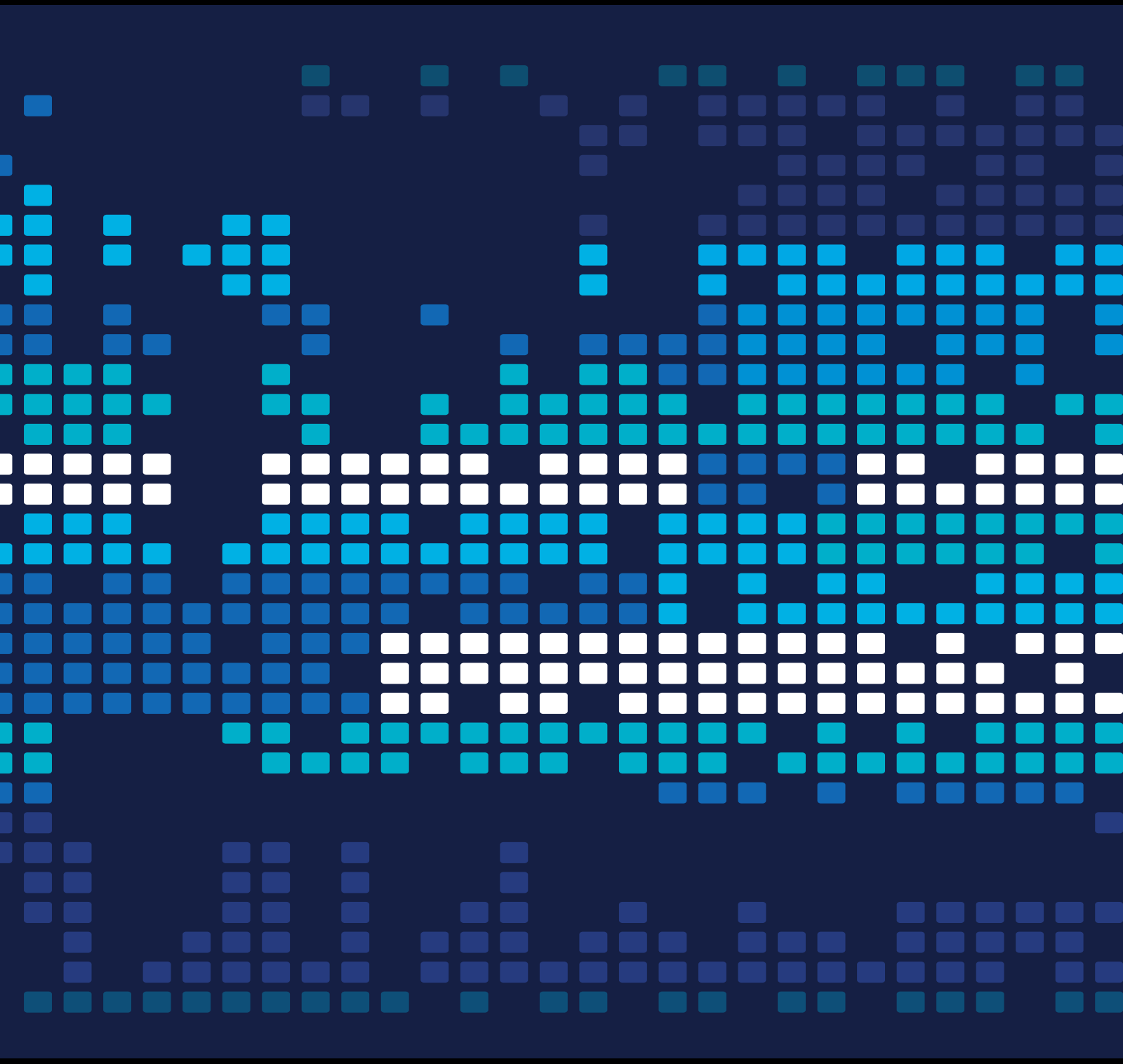
Scientific Programming

Software Engineering for CSE

Guest Editors: Jeffrey C. Carver, Neil Chue Hong, and Selim Ciraci



Software Engineering for CSE

Scientific Programming

Software Engineering for CSE

Guest Editors: Jeffrey C. Carver, Neil Chue Hong, and Selim Ciraci

Editorial Board

Contents

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 591562, 2 pages
http://dx.doi.org/10.1155/2015/591562

Editorial

Software Engineering for CSE

Jeffrey C. Carver,[1] Neil Chue Hong,[2] and Selim Ciraci[3]

[1] *University of Alabama, Tuscaloosa, AL, USA*
[2] *Software Sustainability Institute, University of Edinburgh, Edinburgh, UK*
[3] *Microsoft, Redmond, WA, USA*

Correspondence should be addressed to Jeffrey C. Carver; carver@cs.ua.edu

Received 11 March 2015; Accepted 11 March 2015

1. Workshop Overview

This special issue contains extensions of the best papers from the First International Workshops on Software Engineering for High Performance Computing in Computational Science & Engineering (SE-HPCCSE 2013), which was held during the SC'13 conference. For full details about the workshop (and others in this series), please visit the workshop website http://SE4Science.org/workshops, where the interested reader can find an overview of the workshop, the schedule, and links to the published proceedings. The goal of the workshop was to bring together researchers from different domains (i.e., computational science, software engineering, and high-performance computing) to present their work and discuss important issues related to the intersection of these fields. Because the format of the workshop allows for short paper presentations along with ample time for small group discussion, this workshop provides a unique venue where researchers from different backgrounds can meet and interact in a more informal setting. This editorial first briefly describes the interesting results of the group discussions. Then, it provides a brief overview of the three papers included in the special issue.

2. Summary of Workshop Discussion

During the workshop, the attendees focused their discussion on a number of topics. Here we summarize the discussion on two of the most interesting. The first topic is the use of *design patterns* in high-performance scientific software. In general, attendees thought that using appropriate design patterns in the right situation was beneficial. There was some discussion about where scientific developers could find a good catalog of existing patterns and where new patterns could be published. One concern was whether design patterns are applicable in situations where there is a large base of existing code. The belief was that it is not worth the effort to tear down and rewrite such code, just to take advantage of a design pattern.

The second topic was the use of *unit testing*. Recent programming languages and development environments provide unit testing frameworks, which makes it easier for developers to adopt this approach. The attendees thought that unit testing is necessary to ensure the correctness of high-performance scientific software systems. However, while support for unit testing is present in many languages, the attendees agreed that the languages (such as Fortran) and frameworks typically used for developing high-performance scientific software do not have adequate support for unit testing. The general consensus among the attendees was that developers need to be introduced to widely used software testing tools, such as memory leak detectors and stress testing frameworks.

At the end of the workshop, we conducted a larger group discussion on the path forward in this field. During this discussion, the attendees highlighted a number of barriers that hamper further advancement. Some of these barriers include the following:

(i) How to deal with reproducibility: is it really necessary to have bitwise reproducibility?

(ii) Still a lot of tooling that is not available for the languages commonly in use.

3. Summary of Included Papers

This special issue includes three papers. The paper "Extracting UML Class Diagrams from Object-Oriented Fortran: ForUML" describes a tool that automatically extracts UML class diagrams from Fortran source code. The paper "High-Performance Design Patterns for Modern Fortran" describes and evaluates a coarray MPI implementation of some patterns that support asynchronous evaluation of expressions comprised of parallel operations on distributed data. The paper "Test-driven coarray parallelization of a legacy Fortran application" describes 17 code modernization steps used to refactor and parallelize a legacy Fortran program and evaluates the performance of the resulting code.

Acknowledgments

We would like to thank the authors and the reviewers for making the workshop and this special issue possible. Jeffrey C. Carver would like to acknowledge partial support from NSF-1243887. Neil Chue Hong was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) Grant EP/H043160/1 for the UK Software Sustainability Institute.

Jeffrey C. Carver
Neil Chue Hong
Selim Ciraci

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

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

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

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

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.

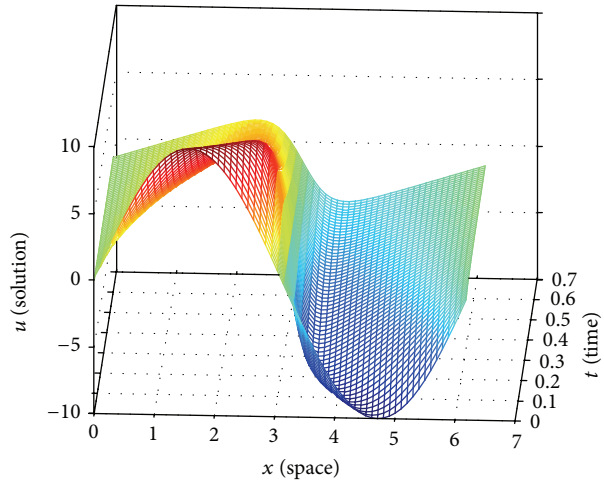


Figure 1: Unsteady, 1D Burgers equation solution values (vertical axis) over space (horizontal axis) and time (oblique axis). 1D Burgers equation solution surface: red (highest) and blue (lowest) relative to the $u = 0$ plane.

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 = [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.

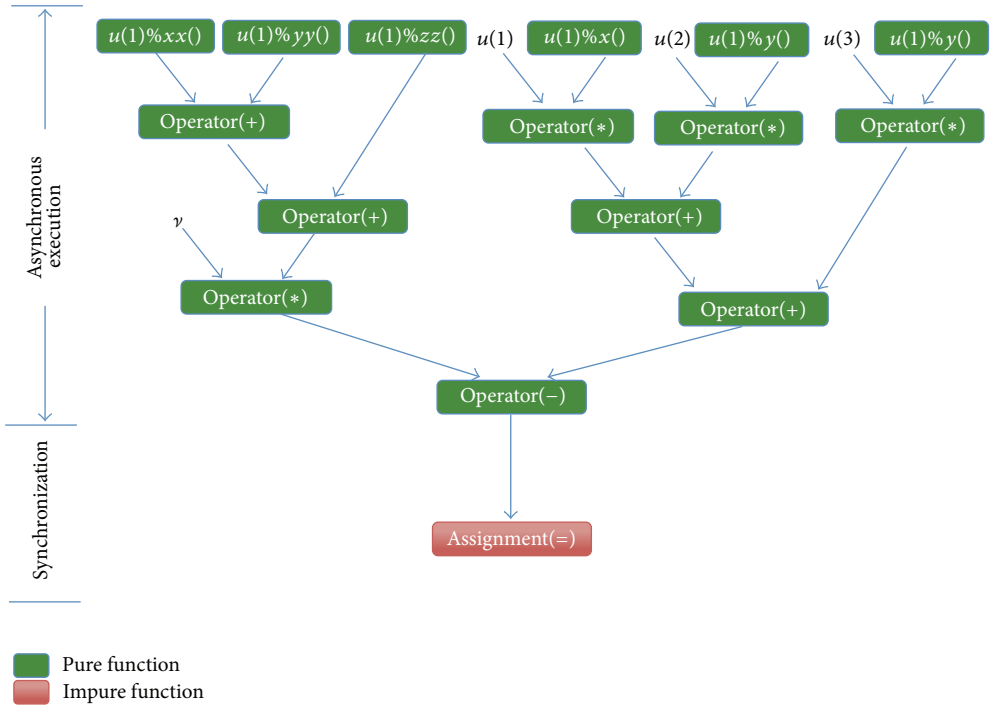


Figure 2: Calling sequence for evaluating the RHS of (2) and assigning the result.

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).

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.

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.

```
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

```
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

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

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

```
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

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

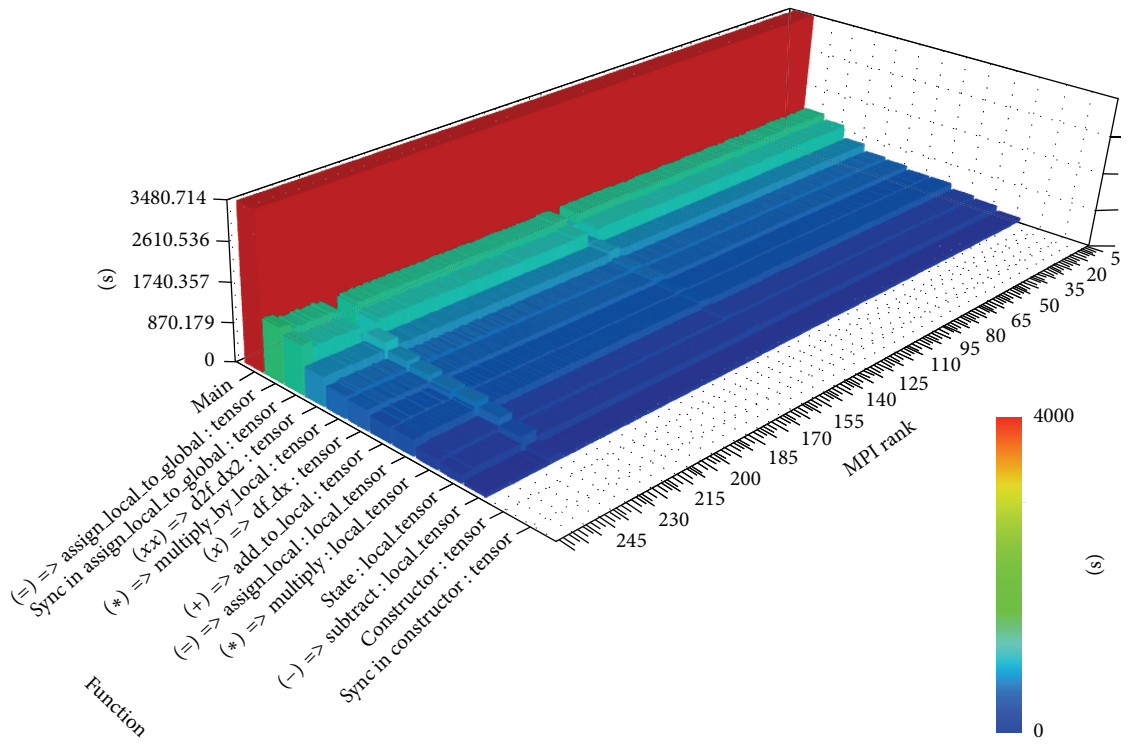
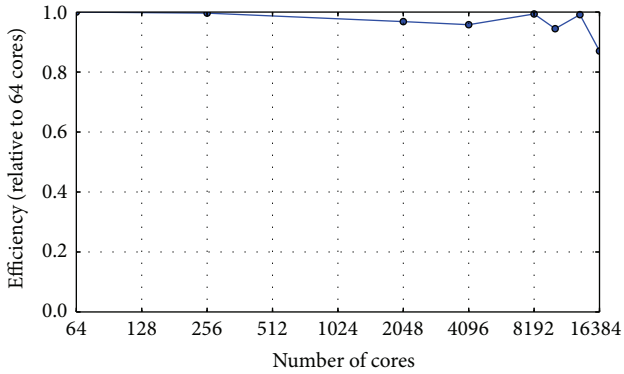


FIGURE 3: Runtime work distribution on all images. Each operator is shown in parenthesis, followed consecutively by the name of the type-bound procedure implementing the operator and the name of the corresponding module. The two points of synchronization are indicated by the word "sync" followed by the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

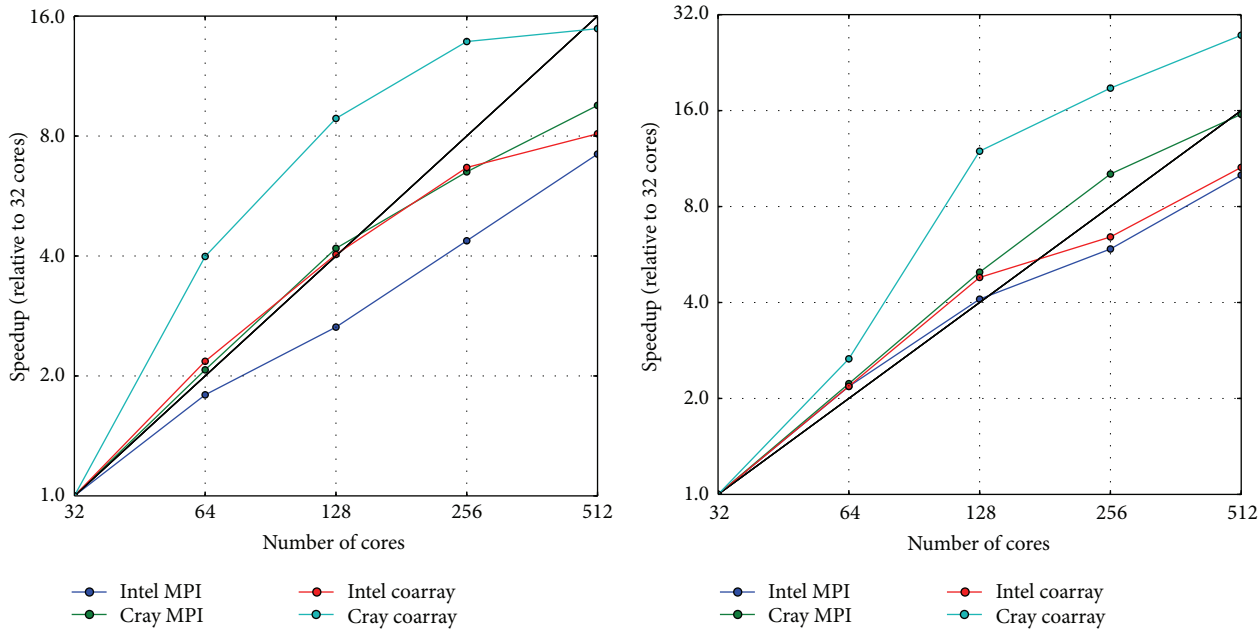
FIGURE 4: Weak scaling of solver for (3) using the coarray version on Cray.

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

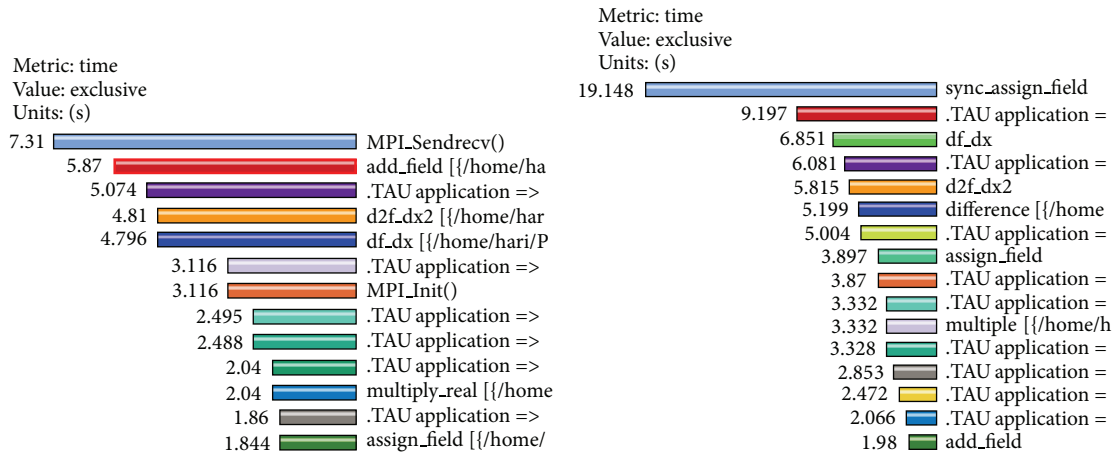
Figure 5: Strong scaling performance of the coarray and MPI versions of the solver for (3) using different platforms. The raw execution times are listed in Table 1.

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:

$$u_t = \nu u_{xx} - u u_x. \quad (4)$$

(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.

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 904983, 12 pages
http://dx.doi.org/10.1155/2015/904983

Research Article

Using Coarrays to Parallelize Legacy Fortran Applications: Strategy and Case Study

Hari Radhakrishnan,[1] Damian W. I. Rouson,[2] Karla Morris,[3] Sameer Shende,[4] and Stavros C. Kassinos[5]

[1]*EXA High Performance Computing, 1087 Nicosia, Cyprus*
[2]*Stanford University, Stanford, CA 94305, USA*
[3]*Sandia National Laboratories, Livermore, CA 94550, USA*
[4]*University of Oregon, Eugene, OR 97403, USA*
[5]*Computational Sciences Laboratory (UCY-CompSci), University of Cyprus, 1678 Nicosia, Cyprus*

Correspondence should be addressed to Damian W. I. Rouson; damian@rouson.net

Received 8 April 2014; Accepted 5 August 2014

Academic Editor: Jeffrey C. Carver

This paper summarizes a strategy for parallelizing a legacy Fortran 77 program using the object-oriented (OO) and coarray features that entered Fortran in the 2003 and 2008 standards, respectively. OO programming (OOP) facilitates the construction of an extensible suite of model-verification and performance tests that drive the development. Coarray parallel programming facilitates a rapid evolution from a serial application to a parallel application capable of running on multicore processors and many-core accelerators in shared and distributed memory. We delineate 17 code modernization steps used to refactor and parallelize the program and study the resulting performance. Our initial studies were done using the Intel Fortran compiler on a 32-core shared memory server. Scaling behavior was very poor, and profile analysis using TAU showed that the bottleneck in the performance was due to our implementation of a collective, sequential summation procedure. We were able to improve the scalability and achieve nearly linear speedup by replacing the sequential summation with a parallel, binary tree algorithm. We also tested the Cray compiler, which provides its own collective summation procedure. Intel provides no collective reductions. With Cray, the program shows linear speedup even in distributed-memory execution. We anticipate similar results with other compilers once they support the new collective procedures proposed for Fortran 2015.

1. Introduction

Background. Legacy software is old software that serves a useful purpose. In high-performance computing (HPC), a code becomes "old" when it no longer effectively exploits current hardware. With the proliferation of multicore processors and many-core accelerators, one might reasonably label any serial code as "legacy software." The software that has proved its utility over many years, however, typically has earned the trust of its user community.

Any successful strategy for modernizing legacy codes must honor that trust. This paper presents two strategies for parallelizing a legacy Fortran code while bolstering trust in the result: (1) a test-driven approach that verifies the numerical results and the performance relative to the original code and (2) an evolutionary approach that leaves much of the original code intact while offering a clear path to execution on multicore and many-core architectures in shared and distributed memory.

The literature on modernizing legacy Fortran codes focuses on programmability issues such as increasing type safety and modularization while reducing data dependancies via encapsulation and information hiding. Achee and Carver [1] examined object extraction, which involves identifying candidate objects by analyzing the data flow in Fortran 77 code. They define a cohesion metric that they use to group

global variables and parameters. They then extracted methods from the source code. In a 1500-line code, for example, they extract 26 candidate objects.

Norton and Decyk [2], on the other hand, focused on wrapping legacy Fortran with more modern interfaces. They then wrap the modernized interfaces inside an object/abstraction layer. They outline a step-by-step process that ensures standards compliance, eliminates undesirable features, creates interfaces, adds new capabilities, and then groups related abstractions into classes and components. Examples of undesirable features include **common** blocks, which potentially facilitate global data-sharing and aliasing of variable names and types. In Fortran, giving procedures explicit interfaces facilitates compiler checks on argument type, kind, and rank. New capabilities they introduced included dynamic memory allocation.

Greenough and Worth [3] surveyed tools that enhance software quality by helping to detect errors and to highlight poor practices. The appendices of their report provide extensive summaries of the tools available from eight vendors with a very wide range of capabilities. A sample of these capabilities includes memory leak detection, automatic vectorization and parallelization, dependency analysis, call-graph generation, and static (compile-time) as well as dynamic (run-time) correctness checking.

Each of the aforementioned studies explored how to update codes to the Fortran 90/95 standards. None of the studies explored subsequent standards and most did not emphasize performance improvement as a main goal. One recent study, however, applied automated code transformations in preparation for possible shared-memory, loop-level parallelization with OpenMP [4]. We are aware of no published studies on employing the Fortran 2008 coarray parallel programming to refactor a serial Fortran 77 application. Such a refactoring for parallelization purposes is the central aim of the current paper.

Case Study: PRM. Most commercial software models for turbulent flow in engineering devices solve the Reynolds-averaged Navier-Stokes (RANS) partial differential equations. Deriving these equations involves decomposing the fluid velocity field, $\mathbf{u}$, into a mean part, $\overline{\mathbf{u}}$, and a fluctuating part, $\mathbf{u}'$:

$$\mathbf{u} \equiv \overline{\mathbf{u}} + \mathbf{u}'. \tag{1}$$

Substituting (1) into a momentum balance and then averaging over an ensemble of turbulent flows yield the following RANS equation:

$$\rho \overline{u_j} \frac{\partial \overline{u_i}}{\partial x_j} = \rho \overline{f_i} + \frac{\partial}{\partial x_j} \left[-\overline{p}\delta_{ij} + \mu \left(\frac{\partial \overline{u_i}}{\partial x_j} + \frac{\partial \overline{u_j}}{\partial x_i} \right) - \rho \overline{u_i' u_j'} \right], \tag{2}$$

where μ is the fluid's dynamic viscosity; ρ is the fluid's density; t is the time coordinate; u_i and u_j are the ith and jth cartesian components of $\mathbf{u}$; and x_i and x_j are the ith and jth cartesian components of the spatial coordinate $\mathbf{x}$.

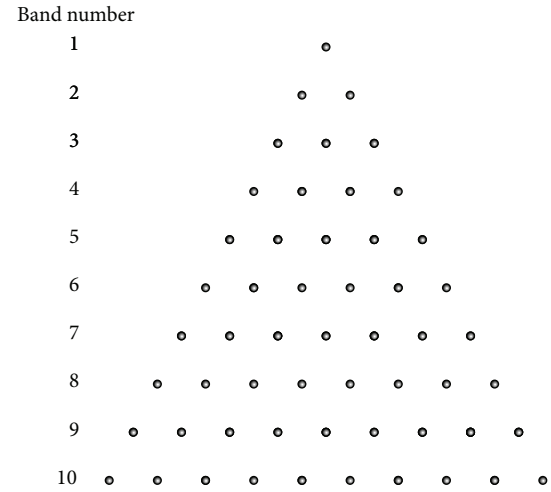


Figure 1: Distribution of particles in bands in one octant.

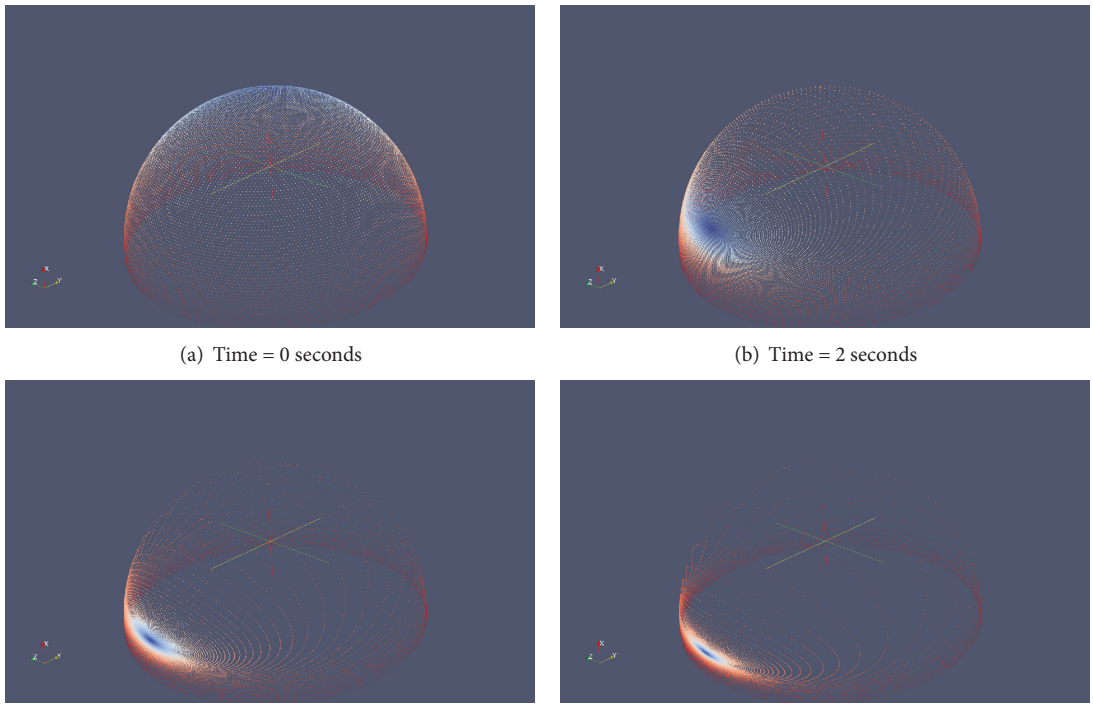
The term $-\rho \overline{u_i' u_j'}$ in (2) is called the Reynolds stress tensor. Its presence poses the chief difficulty at the heart of Reynolds-averaged turbulence modeling; closing the RANS equations requires postulating relations between the Reynolds stress and other terms appearing in the RANS equations, typically the velocity gradient $\partial \overline{u_j}/\partial x_i$ and scalars representing the turbulence scale. Doing so in the most common ways works well for predicting turbulent flows in which the statistics of $\mathbf{u}'$ stay in near-equilibrium with the flow deformations applied via gradients in $\overline{\mathbf{u}}$. Traditional RANS models work less well for flows undergoing deformations so rapid that the fluctuating field responds solely to the deformation without time for the nonlinear interactions with itself that are the hallmark of fluid turbulence. The Particle Representation Model (PRM) [5, 6] addresses this shortcoming. Given sufficient computing resources, a software implementation of the PRM can exactly predict the response of the fluctuating velocity field to rapid deformations.

A proprietary in-house software implementation of the PRM was developed initially at Stanford University, and development continued at the University of Cyprus. The PRM uses a set of hypothetical particles over a unit hemisphere surface. The particles are distributed on each octant of the hemisphere in bands, as shown in Figure 1 for ten bands. The total number of particles is given by

$$\begin{aligned} N_{\text{particles}} &= \underbrace{4}_{\text{Number of octants in hemisphere}} \\ &\quad \times \underbrace{\frac{N_{\text{bands}} \times (N_{\text{bands}} + 1)}{2}}_{\text{Number of particles in one octant}} \\ &= 2 \times N_{\text{bands}} \times (N_{\text{bands}} + 1). \end{aligned} \tag{3}$$

So, the computational time scales quadratically with the number of bands used.

Each particle has a set of assigned properties that describe the characteristics of an idealized flow. Assigned particle properties include vector quantities such as velocity and



(a) Time = 0 seconds

(b) Time = 2 seconds

(c) Time = 4 seconds

(d) Time = 6 seconds

Figure 2: Results of a PRM computation. The particles are colored based on their initial location. The applied flow condition, shear flow along the y-direction, causes the uniformly distributed particles to aggregate along that axis.

orientation as well as scalar quantities such as pressure. Thus, each particle can be thought of as representing the dynamics of a hypothetical one-dimensional (1D), one-component (1C) flow. Tracking a sufficiently large number of particles and then averaging the properties of all the particles (as shown in Figure 2), that is, all the possible flows considered, yield a representation of the 3D behavior in an actual flowing fluid.

Historically, a key disadvantage of the PRM has been costly execution times because a very large number of particles are needed to accurately capture the physics of the flow. Parallelization can reduce this cost significantly. Previous attempts to develop a parallel implementation of the PRM using MPI were abandoned because the development, validation, and verification times did not justify the gains. Coarrays allowed us to parallelize the software with minimal invasiveness and the OO test suite facilitated a continuous build-and-test cycle that reduced the development time.

2. Methodology

2.1. Modernization Strategy. Test-Driven Development (TDD) grew out of the Extreme Programming movement of the 1990s, although the basic concepts date as far back as the NASA space program in the 1960s. TDD iterates quickly toward software solutions by first writing tests that specify what the working software must do and then writing only a sufficient amount of application code in order to pass the test. In the current context, TDD serves the purpose of ensuring that our refactoring exercise preserves the expected results for representative production runs.

Table 1 lists 17 steps employed in refactoring and parallelizing the serial implementation of the PRM. They have been broken down into groups that addressed various facets of the refactoring process. The open-source CTest framework that is part of CMake was used for building the tests. Our first step, therefore, was to construct a CMake infrastructure that we used for automated building and testing and to set up a code repository for version control and coordination.

The next six steps address Fortran 77 features that have been declared obsolete in more recent standards or have been deprecated in the Fortran literature. We did not replace **continue** statements with **end do** statements as these did not affect the functionality of the code.

The next two steps were crucial in setting up the build testing infrastructure. We automated the initialization by replacing the keyboard inputs with default values. The next step was to construct extensible tests based on these default values, which are described in Section 3.

The next three steps expose optimization opportunities to the compiler. One exploits Fortran's array syntax. Two exploit Fortran's facility for explicitly declaring a procedure to be "**pure**," that is, free of side effects, including input/output, modifying arguments, halting execution, or modifying nonlocal state. Other steps address type safety and memory management.

Array syntax gives the compiler a high-level view of operations on arrays in ways the compiler can exploit with various optimizations, including vectorization. The ability to communicate functional purity to compilers also enables numerous compiler optimizations, including parallelism.

Table 1: Modernization steps: horizontal lines indicate partial ordering.

Step	Details
1	Set up automated builds via CMake[1] and version control via Git[2].
2	Convert fixed- to free-source format via "convert.f90" by Metcalf[3].
3	Replace **goto** with **do while** for main loop termination.
4	Enforce type/kind/rank consistency of arguments and return values by wrapping all procedures in a **module**.
5	Eliminate implicit typing.
6	Replace **data** statements with **parameter** statements.
7	Replace write-access to **common** blocks with module variables.
8	Replace keyboard input with default initializations.
9	Set up automated, extensible tests for accuracy and performance via OOP and CTest[1].
10	Make all procedures outside of the main program **pure.**
11	Eliminate actual/dummy array shape inconsistencies by passing array subsections to assumed-shape arrays.
12	Replace static memory allocation with dynamic allocation.
13	Replace loops with array assignments.
14	Expose greater parallelism by unrolling the nested loops in the particle set-up.
15	Balance the work distribution by spreading particles across images during set-up.
16	Exploit a Fortran 2015 collective procedure to gather statistics.
17	Study and tune performance with TAU[4].

[1]http://www.cmake.org/.
[2]http://git-scm.com/.
[3]ftp://ftp.numerical.rl.ac.uk/pub/MandR/convert.f90.
[4]http://tau.uoregon.edu/.

The final steps directly address parallelism and optimization. One unrolls a loop to provide for more fine-grained data distribution. The other exploits the co_sum intrinsic collective procedure that is expected to be part of Fortran 2015 and is already supported by the Cray Fortran compiler. (With the Intel compiler, we write our own co_sum procedure.) The final step involves performance analysis using the Tuning and Analysis Utilities [7].

3. Extensible OO Test Suite

At every step, we ran a suite of accuracy tests to verify that the results of a representative simulation did not deviate from the serial code's results by more than 50 parts per million (ppm). We also ran a performance test to ensure that the single-image runtime of the parallel code did not exceed the serial code's runtime by more than 20%. (We allowed for some increase with the expectation that significant speedup would result from running multiple images.)

Our accuracy tests examine tensor statistics that are calculated using the PRM. In order to establish a uniform protocol for running tests, we defined an abstract base tensor class as shown in Listing 1.

The base class provided the bindings for comparing tensor statistics, displaying test results to the user, and exception handling. Specific tests take the form of three child classes, reynolds_stress, dimensionality, and circulicity, that extend the tensor class and thereby inherit a responsibility to implement the tensor's deferred bindings compute_results and expected_results. The class diagram is shown in Figure 3. The tests then take the form

```
if (.not. stess_tensor%verify_result (when)) &
    error stop 'Test_failed.'
```

where stress_tensor is an instance of one of the three child classes shown in Figure 3 that extend tensor; "when" is an integer time stamp; **error stop** halts all images and prints the shown string to standard error; and verify_result is the **pure** function shown in Listing 1 that invokes the two aforementioned deferred bindings to compare the computed results to the expected results.

4. Coarray Parallelization

Modern HPC software must be executed on multicore processors or many-core accelerators in shared or distributed memory. Fortran provides for such flexibility by defining a partitioned global address space (PGAS) without referencing how to map coarray code onto a particular architecture. Coarray Fortran is based on the Single Program Multiple Data (SPMD) model, and each replication of the program is called an image [8]. Fortran 2008 compilers map these images to an underlying transport network of the compiler's choice. For example, the Intel compiler uses MPI for the transport network whereas the Cray compiler uses a dedicated transport layer.

A coarray declaration of the form

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

facilitates indexing into the variable "a" along three regular dimensions and one codimension so

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

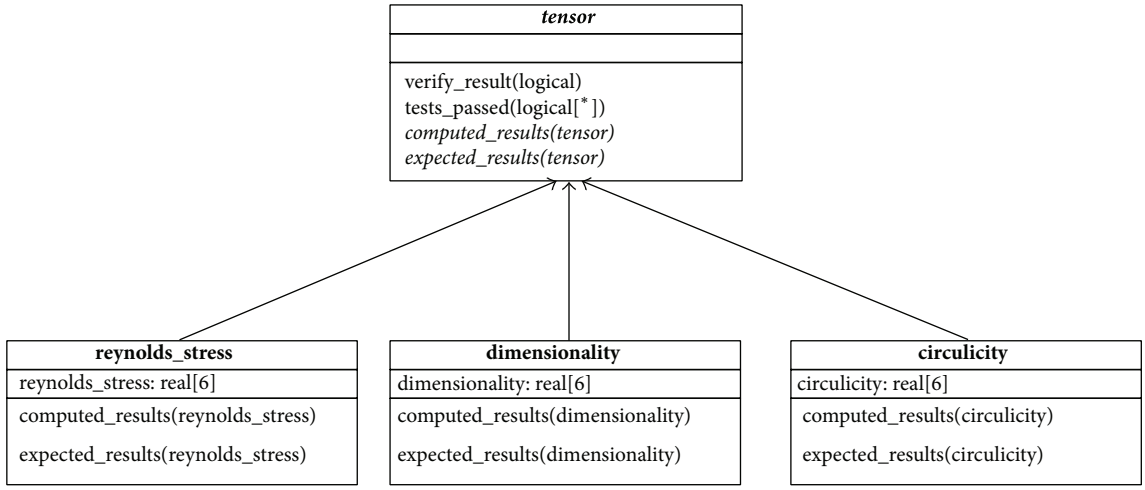


Figure 3: Class diagram of the testing framework. The deferred bindings are shown in italics, and the abstract class is shown in bold italics.

```
module abstract_tensor_class
  type, abstract :: tensor
  contains
     procedure(return_computed_results), deferred :: &
         computed_results
     procedure(return_expected_results), deferred :: &
         expected_results
     procedure :: verify_result
  end type
  abstract interface
    pure function return_computed_results(this) &
      result(computed_values)
      import :: tensor
      class(tensor), intent(in) :: this
      real, allocatable :: computed_values(:)
    end function
    ! return_expected_results interface omitted
  end abstract interface
 contains
  pure function verify_result(this) &
    result(all_tests_passed)
    class(tensor), intent(in) :: this
    logical :: all_tests_passed
    all_tests_passed = all(tests_passed( &
    this%computed_results(), this%expected_results()))
  end function
end module
```

Listing 1: Base tensor class.

copies the first element of image 2 to the first element of whatever image executes this line. The ability to omit the coindex on the left-hand side (LHS) played a pivotal role in refactoring the serial code with minimal work; although we added codimensions to existing variables' declarations, subsequent accesses to those variables remained unmodified except where communication across images is desired. When necessary, adding coindices facilitated the construction of collective procedures to compute statistics.

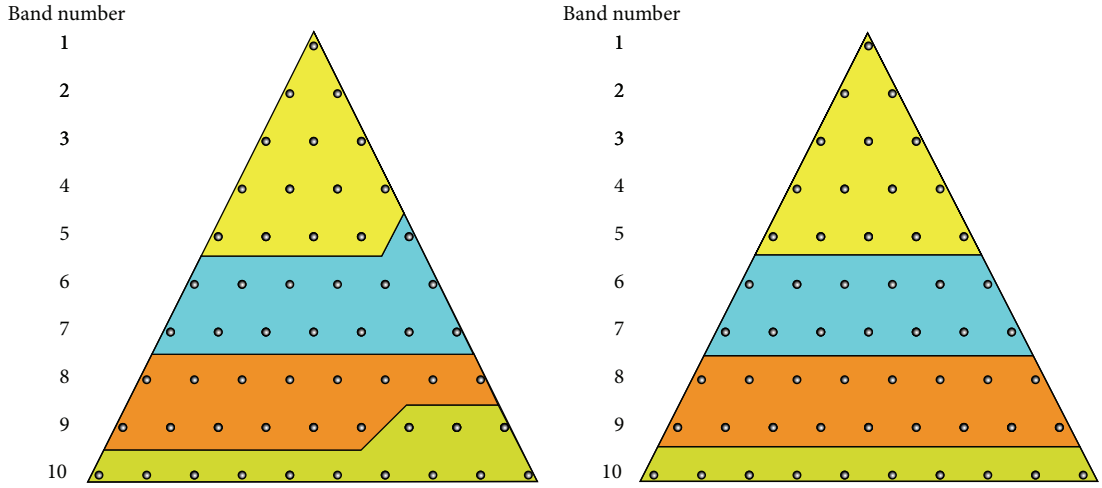
```
l = 0 ! Global particle number
do k = 1, nb ! Loop over the bands
    do m = 1, k ! Loop over the particles in band
        ! First octant
        l = l + 1
        ! Do some computations
        ! Second octant
        l = l + 1
        ! Do some computations
        ! Third octant
        l = l + 1
        ! Do some computations
        ! Fourth octant
        l = l + 1
        ! Do some computations
    end do
end do
```

Listing 2: Legacy particle loop.

In the legacy version, the computations of the particle properties were done using two nested loops, as shown in Listing 2.

Distributing the particles across the images and executing the computations inside these loops can speed up the execution time. This can be achieved in two ways.

Method 1 works with the particles directly, splitting them as evenly as possible across all the images, allowing image boundaries to occur in the middle of a band. This distribution is shown in Figure 4(a). To achieve this distribution, the two nested do loops are replaced by one loop over the particles, and the indices for the two original loops are computed from the global particle number, as shown in Listing 3. However in this case, the code becomes complex and sensitive to precision.



(a) Partitioning of the particles to achieve even distribution of particles

(b) Partitioning of the bands to achieve nearly even distribution of particles

Figure 4: Two different partitioning schemes were tried for load balancing.

```
! Loop over the particles
do l = my_first_particle, my_last_particle, 4
    k = nint(sqrt(real(l) * 0.5))
    m = (l - (1 + 2 * k * (k - 1) - 4))/4
        ! First octant
        ! Do some computations
        ! Second octant
        ! Do some computations
        ! Third octant
        ! Do some computations
        ! Fourth octant
        ! Do some computations
end do
```

Listing 3: Parallel loop by splitting particles.

Method 2 works with the bands, splitting them across the images to make the particle distribution as even as possible. This partitioning is shown in Figure 4(b). Method 2, as shown in Listing 4, requires fewer changes to the original code shown in Listing 2 but is suboptimal in load balancing.

```
! Loop over the bands
do k = my_first_band, my_last_band
    ! Global number
    ! of last particle in (k - 1) band
    l = k ** 2 + (k - 1) ** 2 - 1
    ! Loop over the particles in band
    do m = 1, k
        ! First octant
        l = l + 1
        ! Do some computations
        ! Second octant
        l = l + 1
        ! Do some computations
        ! Third octant
        l = l + 1
        ! Do some computations
        ! Fourth octant
        l = l + 1
        ! Do some computations
    end do
end do
```

Listing 4: Parallel loop by splitting bands.

5. Results

5.1. Source Code Impact. We applied our strategy to two serial software implementations of the PRM. For one version, the resulting code was 10% longer than the original: 639 lines versus 580 lines with no test suite. In the second version, the code expanded 40% from 903 lines to 1260 lines, not including new input/output (I/O) code and the test code described in Section 3. The test and I/O code occupied additional 569 lines.

5.2. Ease of Use: Coarrays versus MPI. The ability to drop the coindex from the notation, as explained in Section 4, was a big help in parallelizing the program without making significant changes to the source code. A lot of the bookkeeping is handled behind the scenes by the compiler making it possible to make the parallelization more abstract but also easier to follow. For example, Listing 5 shows the MPI calls necessary to gather the local arrays into a global array on all the processors.

The equivalent calls using the coarray syntax is the listing shown in Listing 6.

Reducing the complexity of the code also reduces the chances of bugs in the code. In the legacy code, the arrays

```
integer :: my_rank, num_procs
integer, allocatable, dimension(:) :: &
    my_first, my_last, counts, displs
call mpi_comm_size(MPI_COMM_WORLD, num_procs, ierr)
call mpi_comm_rank(MPI_COMM_WORLD, my_rank, ierr)
allocate(my_first(num_procs), my_last(num_procs), &
    counts(num_procs), displs(num_procs))
my_first(my_rank + 1) = lbound(sn, 2)
my_last(my_rank + 1) = ubound(sn, 2)
call mpi_allgather(MPI_IN_PLACE, 1, MPI_INTEGER, &
  my_first, 1, MPI_INTEGER, MPI_COMM_WORLD, ierr)
call mpi_allgather(MPI_IN_PLACE, 1, MPI_INTEGER, &
  my_last, 1, MPI_INTEGER, MPI_COMM_WORLD, ierr)
do i = 1, num_procs
     displs(i) = my_first(i) − 1
     counts(i) = my_last(i) − my_first(i) + 1
end do
call mpi_allgatherv(sn, 5 ∗ counts(my_rank + 1), &
  MPI_DOUBLE_PRECISION, sn_global, 5 ∗ counts, &
  5 ∗ displs, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
call mpi_allgatherv(cr, 5 ∗ counts(my_rank + 1), &
  MPI_DOUBLE_PRECISION, cr_global, 5 ∗ counts, &
  5 ∗ displs, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
```

Listing 5: Using MPI_ALLGATHER to collect local arrays into a global array.

```
integer :: my_first[∗], my_last[∗]
my_first = lbound(sn, 2)
my_last = ubound(sn, 2)
do l = 1, num_images()
    cr_global(:, my_first[l]:my_last[l]) = cr(:,:)[l]
    sn_global(:, my_first[l]:my_last[l]) = sn(:,:)[l]
end do
```

Listing 6: Coarray method of gathering arrays.

sn and *cr* carried the information about the state of the particles. By using the coarray syntax and dropping the coindex, we were able to reuse all the original algorithms that implemented the core logic of the PRM. This made it significantly easier to ensure that the refactoring did not alter the results of the model. The main changes were to add codimensions to the *sn* and *cr* declarations and update them when needed, as shown in Listing 6.

5.3. Scalability. We intend for PRM to serve as an alternative to turbulence models used in routine engineering design of fluid devices. There is no significant difference in the PRM results when more than 1024 bands (approximately 2.1 million particles) are used to represent the flow state so this was chosen as the upper limit of the size of our data set. Most engineers and designers run simulations on desktop computers. As such, the upper bound on what is commonly available is roughly 32 to 48 cores on two or four central processing units (CPUs) plus additional cores on one or more accelerators. We also looked at the scaling performance of parallel implementation of the PRM using Cray hardware and Fortran compiler which has excellent support for distributed-memory execution of coarray programs.

Figure 5 shows the speedup obtained for 200 and 400 bands with the Intel Fortran compiler using the two particle-distribution schemes described in the Coarray Parallelization section. The runs were done using up to 32 cores on the "fat" nodes of ACISS (http://aciss-computing.uoregon.edu/). Each node has four Intel X7560 2.27 GHz 8-core CPUs and 384 GB of DDR3 memory. We see that the speedup was very poor when the number of processors was increased.

We used TAU [7] to profile the parallel runs to understand the bottlenecks during execution. Figure 6 shows the TAU plot for the runtime share for the dominant procedures using different number of images. Figure 7 shows the runtimes for the different functions on the different images. The heights of the columns show the runtime for different functions on the individual cores. There is no significant difference in the heights of the columns proving that the load balancing is very good across the images. We achieved this by mainly using the one-sided communication protocols of CAF as shown in Listing 6 and restricting the sync statements to the collective procedures as shown in Listings 7 and 8. Looking at the runtimes in Figure 6, we identified the chief bottlenecks to be the two collective co_sum procedures which sum values across a coarray by sequentially polling each image for its portion of the coarray. The time required for this procedure

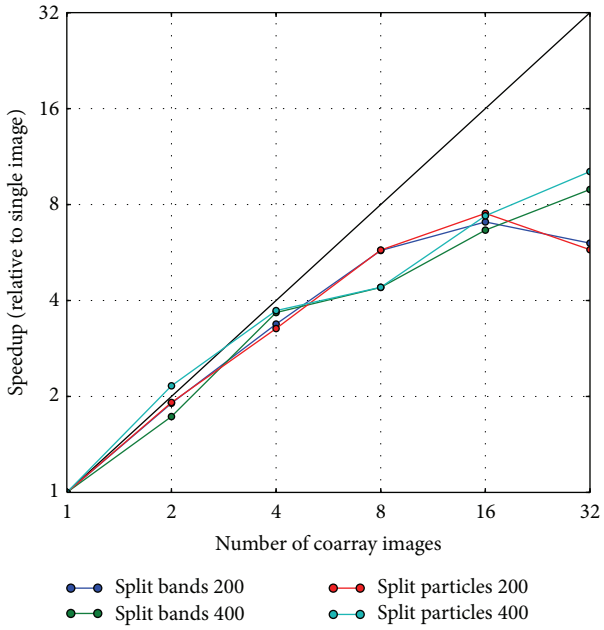


Figure 5: Speedup obtained with sequential co_sum implementation using multiple images on a single server.

```
subroutine vector_co_sum_parallel(vector)
  real(rkind), intent(inout) :: vector(:)[*]
  real(rkind), allocatable :: temp(:)
  integer image, step
  allocate (temp, mold = vector)
  step = 2
  do while (step/2 <= num_images())
    sync all
    if (this_image() + step/2 <= num_images()) then
      temp = vector + vector[this_image() + step/2]
    else
      temp = vector
    end if
    sync all
    vector = temp
    step = step * 2
  end do
  sync all
  if (this_image()/ = 1) vector = vector[1]
  sync all
end subroutine
```

Listing 8: Optimized collective sum routine.

```
subroutine vector_co_sum_serial(vector)
  real(rkind), intent(inout) :: vector(:)[*]
  integer image
  sync all
  if (this_image() == 1) then
    do image = 2, num_images()
      vector(:)[1] = vector(:)[1] + vector(:)[image]
    end do
  end if
  sync all
  if (this_image()/ = 1) vector(:) = vector(:)[1]
  sync all
end subroutine
```

Listing 7: Unoptimized collective sum routine.

is $O(N_{\text{images}})$. The unoptimized co_sum routine for adding a vector across all images is shown in Listing 7. There is an equivalent subroutine for summing a matrix also.

Designing an optimal co_sum algorithm is a platform-dependent exercise best left to compilers. The Fortran standards committee is working on a co_sum intrinsic procedure that will likely become part of Fortran 2015. But to improve the parallel performance of the program, we rewrote the collective co_sum procedures using a binomial tree algorithm that is $O(\log N_{\text{images}})$ in time. The optimized version of the co_sum version is shown in Listing 8.

The speedup obtained with the optimized co_sum routine is shown in Figure 8. We see that the scaling performance of the program becomes nearly linear with the implementation of the optimized co_sum routine. We also see that the scaling efficiency increases when the problem size is increased. This indicates that the poor scaling at smaller problem sizes is due to communication and synchronization [9].

The TAU profile analysis of the runs using different number of images is shown in Figure 9. While there is a small increase in the co_sum computation time when increasing the number of images, it is significantly lower than increase in time for the unoptimized version.

To fully understand the impact of the co_sum routines, we also benchmarked the program using the Cray compiler and hardware. Cray has native support for the co_sum directive in the compiler. Cray also uses its own communication library on Cray hardware instead of building on top of MPI as is done by the Intel compiler. As we can see in Figure 10, the parallel code showed very good strong scaling on the Cray hardware up to 128 images for the problem sizes that we tested.

We also looked at the TAU profiles of the parallel code on the Cray hardware, shown in Figure 11. The profile analysis shows that the time is spent mainly in the time advancement loop when the native co_sum implementation is used.

We hope that, with the development and implementation of intrinsic co_sum routines as part of the 2015 Fortran standard, the Intel compiler will also improve its strong scaling performance with larger number of images. Table 2 shows the raw runtimes for the different runs using 128 bands whose TAU profiles have been shown in Figures 6, 9, and 11. The runtimes for one to four images are very close but they quickly diverge as we increase the number of images due to the impact of the collective procedures.

Table 3 shows the weak scaling performance of the program using the optimized co_sum procedures using the Intel compiler. The number of particles as shown in Figure 1 scales as the square of the number of bands. Therefore, when

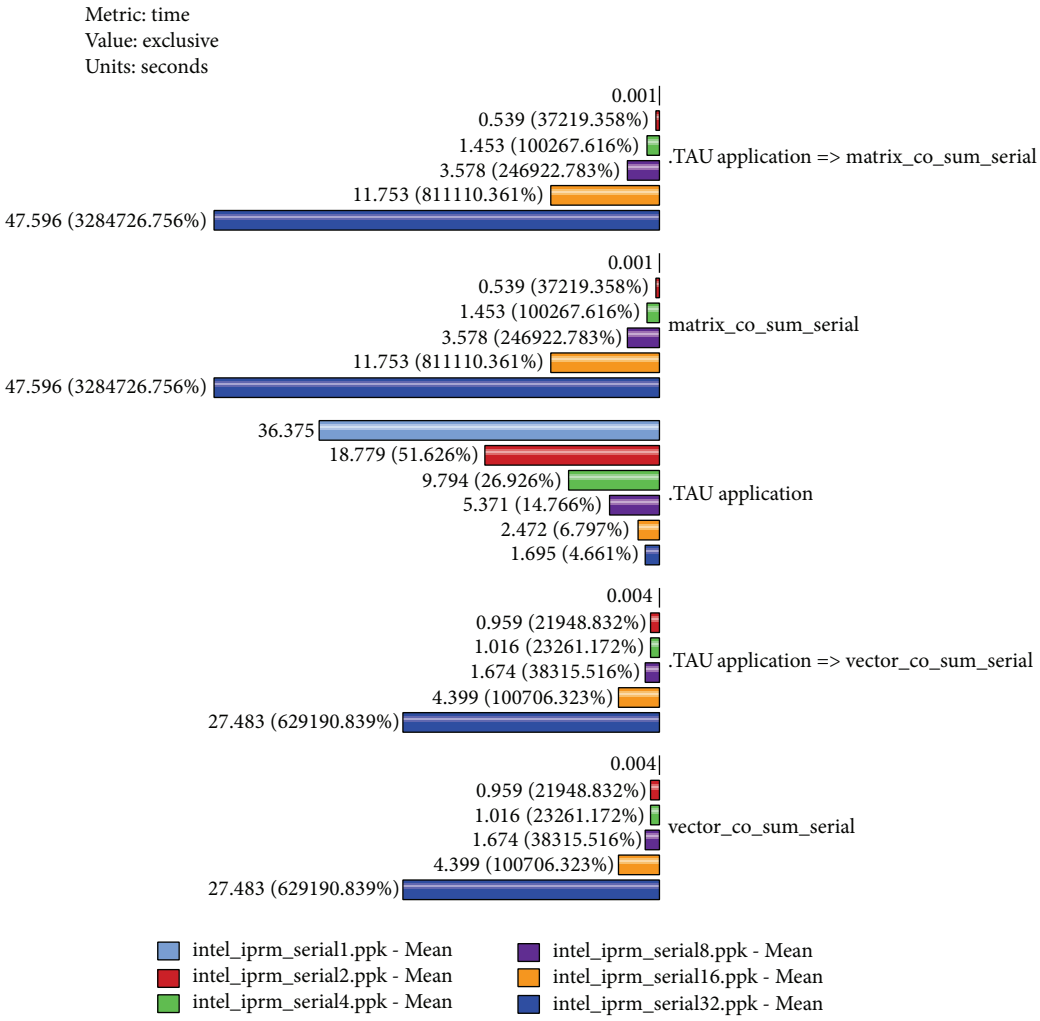


Figure 6: TAU profiling analysis of function runtimes when using the unoptimized co_sum routines with 1, 2, 4, 8, 16, and 32 images. The *.TAU application* is the main program wrapped by TAU for profiling, and *.TAU application =>* refers to functions wrapped by TAU. This notation is also seen in Figures 7 and 9.

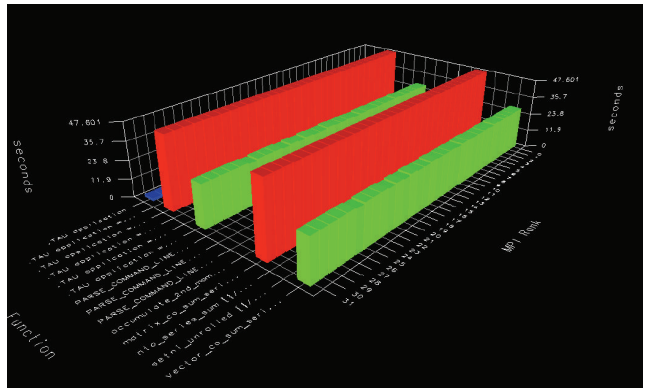


Figure 7: TAU analysis of load balancing and bottlenecks for the parallel code using 32 images.

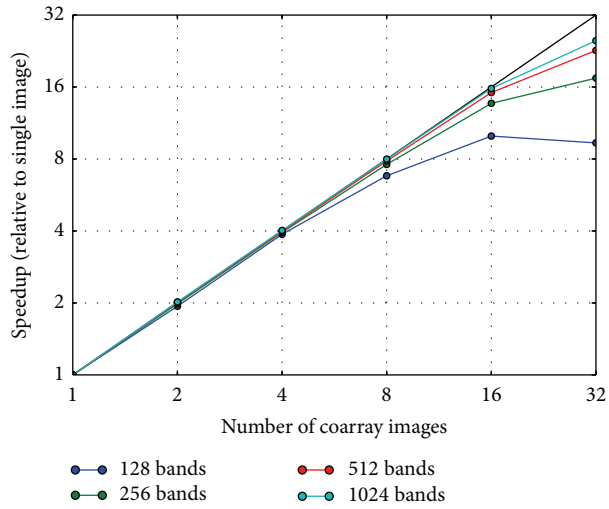


Figure 8: Speedup obtained with parallel co_sum implementation using multiple images on a single server.

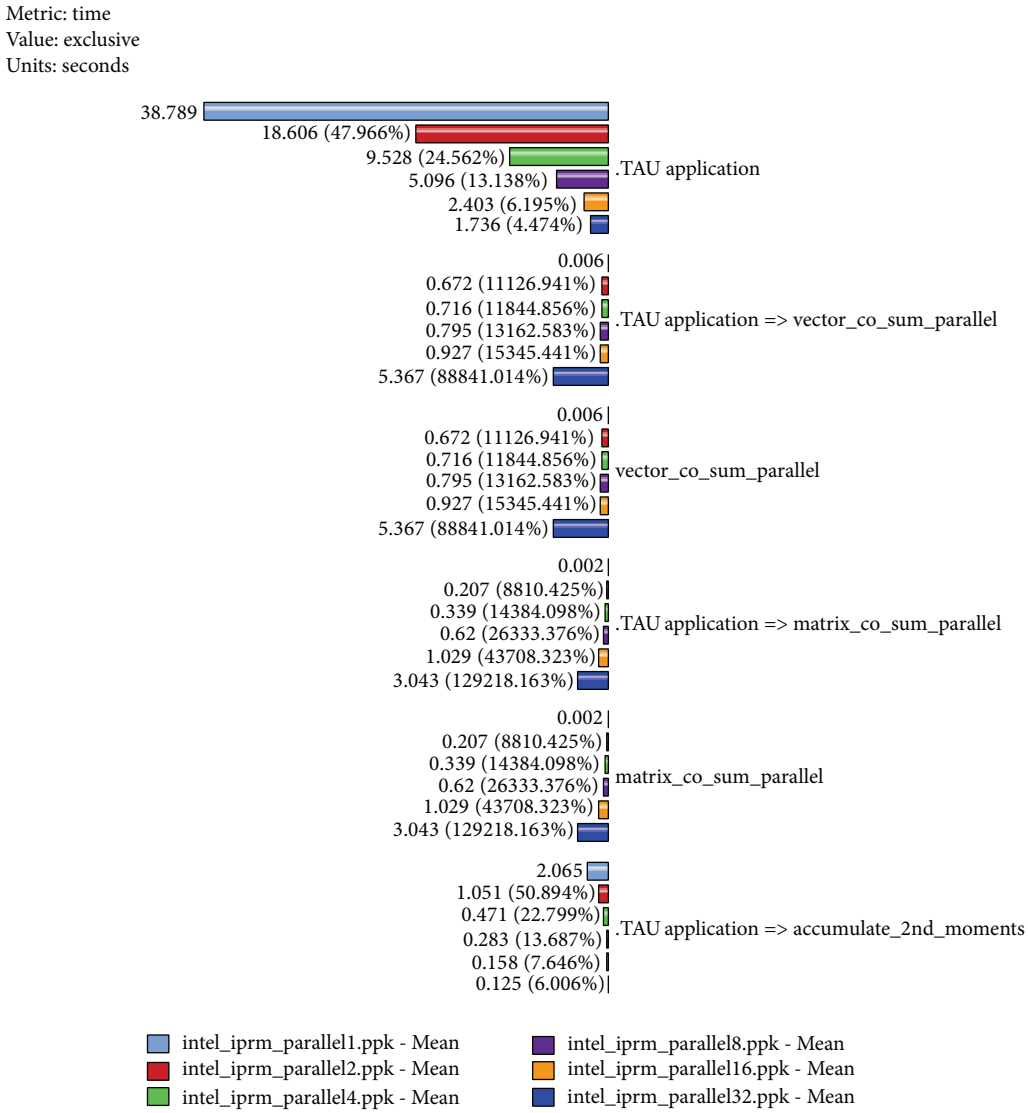


Figure 9: TAU profiling analysis of function runtimes when using the optimized co_sum routines with 1, 2, 4, 8, 16, and 32 images.

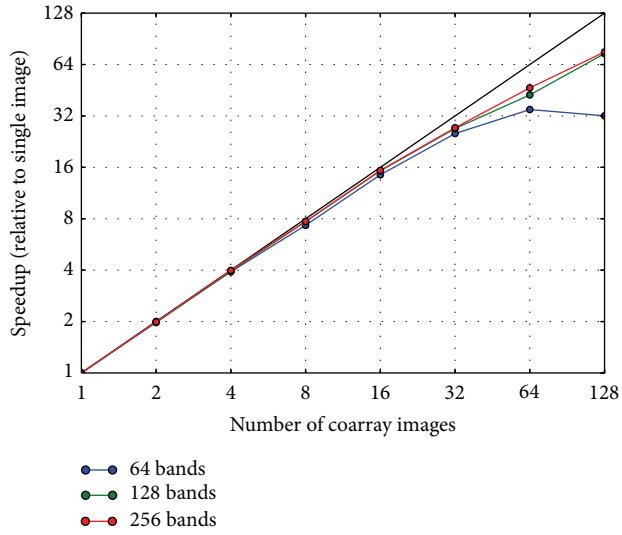


Figure 10: Speedup obtained with parallel co_sum implementation using multiple images on a distributed-memory Cray cluster.

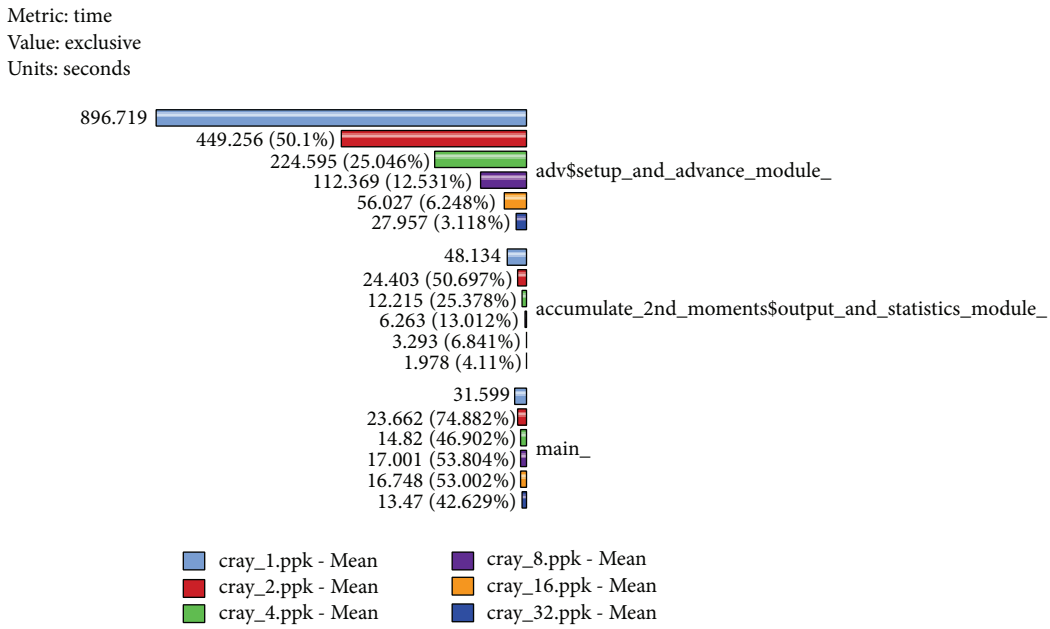


Figure 11: TAU profiling analysis of function runtimes when using the Cray native co_sum routines with 1, 2, 4, 8, 16, and 32 images.

doubling the number of bands, the number of processors must be quadrupled to have the same execution time. The scaling efficiency for the larger problem drops because of memory requirements; the objects fit in the heap and must be swapped out as needed, increasing the execution time.

6. Conclusions and Future Work

We demonstrated a strategy for parallelizing legacy Fortran 77 codes using Fortran 2008 coarrays. The strategy starts with constructing extensible tests using Fortran's OOP features. The tests check for regressions in accuracy and performance. In the PRM case study, our strategy expanded two Fortran 77 codes by 10% and 40%, exclusive of the test and I/O infrastructure. The most significant code revision involved unrolling two nested loops that distribute particles across images. The resulting parallel code achieves even load balancing but poor scaling. TAU identified the chief bottleneck as a sequential summation scheme.

Based on these preliminary results, we rewrote our co_sum procedure, and the speedup showed marked improvement. We also benchmarked the native co_sum implementation available in the Cray compiler. Our results show that the natively supported collective procedures show the best scaling performance even when using distributed memory. We hope that future native support for collective

Table 2: Runtime in seconds for parallel using 128 bands, and different collective sum routines.

	Number of Images					
	1	2	4	8	16	32
Intel Serial co_sum	35.55	19.80	11.69	9.73	18.71	66.82
Intel Parallel co_sum	37.30	19.33	10.00	6.17	4.62	5.41
Cray Native co_sum	46.71	23.68	11.88	6.06	3.06	1.73

Table 3: Weak scaling performance of coarray version.

Number of images	Number of bands	Number of particles	Particles per image	Time in seconds	Runtime per particle	Efficiency
1	128	33024	33024	44.279	1.34×10^{-3}	1.000
4	256	131584	32896	44.953	1.37×10^{-3}	0.978
16	512	525312	32832	49.400	1.50×10^{-3}	0.893
2	256	131584	65792	101.03	1.54×10^{-3}	1.000
8	512	525312	65664	102.11	1.56×10^{-3}	0.987
32	1024	2099200	65600	129.75	1.98×10^{-3}	0.777

procedures in Fortran 2015 by all the compilers will bring such performance to all platforms.

Conflict of Interests

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

Acknowledgments

The initial code refactoring was performed at the University of Cyprus with funding from the European Commission Marie Curie ToK-DEV grant (Contract MTKD-CT-2004-014199). Part of this work was also supported by the Cyprus Research Promotion Foundation's Framework Programme for Research, Technological Development and Innovation 2009-2010 (ΔΕΣΜΗ 2009-2010) under Grant ΤΠΕ/ΠΛΗΡΟ/0609(ΒΕ)/11. 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 hardware 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. 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. Portions of the Sandia contribution to this work were funded by the New Mexico Small Business Administration and the Office of Naval Research.

References

[1] B. L. Achee and D. L. Carver, "Creating object-oriented designs from legacy FORTRAN code," *Journal of Systems and Software*, vol. 39, no. 2, pp. 179–194, 1997.

[2] C. D. Norton and V. K. Decyk, "Modernizing Fortran 77 legacy codes," *NASA Tech Briefs*, vol. 27, no. 9, p. 72, 2003.

[3] C. Greenough and D. J. Worth, "The transformation of legacy software: some tools and processes," Tech. Rep. TR-2004-012, Council for the Central Laboratory of the Research Councils, Rutherford Appleton Laboratories, Oxfordshire, UK, 2004.

[4] F. G. Tinetti and M. Méndez, "Fortran Legacy software: source code update and possible parallelisation issues," *ACM SIGPLAN Fortran Forum*, vol. 31, no. 1, pp. 5–22, 2012.

[5] S. C. Kassinos and W. C. Reynolds, "A particle representation model for the deformation of homogeneous turbulence," in *Annual Research Briefs*, pp. 31–61, Center for Turbulence Research, Stanford University, Stanford, Calif, USA, 1996.

[6] S. C. Kassinos and E. Akylas, "Advances in particle representation modeling of homogeneous turbulence. from the linear PRM version to the interacting viscoelastic IPRM," in *New Approaches in Modeling Multiphase Flows and Dispersion in Turbulence, Fractal Methods and Synthetic Turbulence*, F. Nicolleau, C. Cambon, J.-M. Redondo, J. Vassilicos, M. Reeks, and A. Nowakowski, Eds., vol. 18 of *ERCOFTAC Series*, pp. 81–101, Springer, Dordrecht, The Netherlands, 2012.

[7] 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.

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

[9] H. Radhakrishnan, D. W. I. Rouson, K. Morris, S. Shende, and S. C. Kassinos, "Test-driven coarray parallelization of a legacy Fortran application," in *Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pp. 33–40, ACM, November 2013.

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 421816, 15 pages
http://dx.doi.org/10.1155/2015/421816

Research Article

Extracting UML Class Diagrams from Object-Oriented Fortran: ForUML

Aziz Nanthaamornphong,[1] Jeffrey Carver,[2] Karla Morris,[3] and Salvatore Filippone[4]

[1]*Department of Information and Communication Technology, Prince of Songkla University, Phuket Campus, Phuket 83120, Thailand*
[2]*Department of Computer Science, University of Alabama, Tuscaloosa, AL 35487, USA*
[3]*Sandia National Laboratories, 7011 East Avenue, Livermore, CA 94550-9610, USA*
[4]*Department of Civil and Computer Engineering, University of Rome 'Tor Vergata', Roma 00173, Italy*

Correspondence should be addressed to Aziz Nanthaamornphong; aziz.nantha@gmail.com

Received 10 April 2014; Accepted 20 June 2014

Academic Editor: Selim Ciraci

Many scientists who implement computational science and engineering software have adopted the object-oriented (OO) Fortran paradigm. One of the challenges faced by OO Fortran developers is the inability to obtain high level software design descriptions of existing applications. Knowledge of the overall software design is not only valuable in the absence of documentation, it can also serve to assist developers with accomplishing different tasks during the software development process, especially maintenance and refactoring. The software engineering community commonly uses reverse engineering techniques to deal with this challenge. A number of reverse engineering-based tools have been proposed, but few of them can be applied to OO Fortran applications. In this paper, we propose a software tool to extract unified modeling language (UML) class diagrams from Fortran code. The UML class diagram facilitates the developers' ability to examine the entities and their relationships in the software system. The extracted diagrams enhance software maintenance and evolution. The experiments carried out to evaluate the proposed tool show its accuracy and a few of the limitations.

1. Introduction

Computational research has been referred to as the third pillar of scientific and engineering research, along with experimental and theoretical research [1]. Computational science and engineering (CSE) researchers develop software to simulate natural phenomena that cannot be studied experimentally or to process large amounts of data. CSE software has a large impact on society as it is used by researchers to study critical problems in a number of important application domains, including weather forecasting, astrophysics, construction of new physical materials, and cancer research [2]. For example, US capabilities in science and engineering are frequently called upon to address urgent challenges in national and homeland security, economic competitiveness, health care, and environmental protection [3]. Recently the software engineering (SE) community has become more interested in the development of software for CSE research.

In this critical type of software, Fortran is still a very widely used programming language [4]. Due to the growing complexity of the problems being addressed through CSE, the procedural programming style available in a language like Fortran 77 is no longer sufficient. Many developers have applied the object-oriented programming (OOP) paradigm to effectively implement the complex data structures required by CSE software. In the case of Fortran developers, this OOP paradigm was first emulated following an object-based approach in Fortran 90/95 [5–7]. By including full support for OOP constructs, the Fortran 2003 language standard influenced the advent of several CSE packages [8–12].

One of the greatest challenges faced by CSE developers is the ability to effectively maintain their software over its

generally long lifetime [13]. This challenge implies high development and maintenance costs during a software system's lifetime. The difficulty of the maintenance process is affected by at least three factors. First, most CSE developers are not formally trained in SE. Second, some existing SE tools are difficult to use in CSE development. In general, CSE developers request tools to accommodate documentation, correctness testing, and aid in design software for testability. Unfortunately, most SE tools were not designed to be used in the context of CSE development. Third, CSE software often lacks the formal documentation necessary to help developers understand its complex design. This lack of documentation presents an even larger software maintenance challenge. The objective of this work is to provide tool support for automatically extracting UML class diagrams from OO Fortran code.

To address this objective, we developed and evaluated the *ForUML* tool. *ForUML* uses a reverse engineering approach to transform Fortran source code into UML models. To ensure flexibility, our solution uses a Fortran parser that does not depend on any specific Fortran compiler and generates output in the XML Metadata Interchange (XMI) format. The tool then displays the results of the analysis (the UML class diagram) using the *ArgoUML* (http://argouml.tigris.org/) modeling tool. We evaluated the accuracy of *ForUML* using five CSE software packages that use object-oriented features from the Fortran 95, 2003, and 2008 compiler standards. This paper extends the workshop paper [14] by providing more background information and more details on the transformation process in ForUML. Additionally, this paper includes a discussion of the audience feedback during the *Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering* (SE-HPCCSE'13).

The contributions of this paper are as follows:

(i) the ForUML tool that will help CSE developers extract UML design diagrams from OO Fortran code to enable them make good decisions about software development and maintenance tasks;

(ii) description of the transformation process used to develop ForUML, which may help other tool authors create tools for the CSE community;

(iii) the results of the evaluation and our experiences using ForUML on real CSE projects to highlight its benefits and limitations;

(iv) workshop feedback that should help SE develop practices and tools that are suitable for use in the CSE domain.

The rest of this paper is organized as follows. Section 2 provides the background concepts related to this work. Section 3 presents ForUML. Section 4 describes the evaluation and our experiences with ForUML. Section 5 discusses the evaluation results and limitations of ForUML. Finally, Section 6 draws conclusions and presents future work.

2. Related Work

This section first describes important CSE characteristics that impact the development of tool support. Next, it presents two important concepts used in the development of ForUML, reverse engineering and OO Fortran. Finally, because one of the benefits of using ForUML is the ability to recognize and maintain design patterns, the last subsection provides some background on design patterns.

2.1. Important CSE Characteristics. This section highlights three characteristics of CSE software development that differentiate it from traditional software development. First, CSE developers typically have a strong background in the theoretical science but often do not have formal training about SE techniques that have proved successful in other software areas. More specifically, because the complexity of the problems addressed by CSE generally requires a domain expert (e.g., a Ph.D. in physics or biology) to even understand the problem, and that domain expert generally must learn how to develop software [15]. Wilson [16] stated that one of the reasons why scientists tend to be less effective programmers is that they do not have the time to learn yet another programming language and software tool. Furthermore, the CSE culture, including most funding agencies, tends to view software as the means to a new scientific discovery rather than as a CSE instrument that must be carefully engineered, maintained, and extended to enable novel science.

Second, some SE tools are difficult to use in a CSE development environment [17]. CSE applications are generally developed with software tools that are crude compared to those used today in the commercial sector. Researchers and scientists seek easy-to-use software that enables analysis of complex data and visualization of complicated interactions. Consequently, CSE developers often have trouble identifying and using the most appropriate SE techniques for their work, in particular as it relates to reverse engineering tasks. Scientists interested in scientific research cannot spend most of their time understanding and using complex software tools. The limited interoperability of the tools and their complexity are major obstructions to their adaptation by the CSE community. For example, Storey noted that CSE developers who lack formal SE training need help with program comprehension when they are developing complex applications [18]. To address this problem, the SE community must develop tools that satisfy the needs of CSE developers. These tools must allow the developers to easily perform important reverse engineering tasks. More specifically, a visualization-based tool is appropriate for program comprehension in complex object-oriented applications [19].

Third, CSE software typically lacks adequate development-oriented documentation [20]. In fact, documentation for CSE software often exists only in the form of subroutine library documentation. This documentation is usually quite clear and sufficient for library users, who treat the library as a black box, but not sufficient for developers who need to understand the library in enough detail to maintain it. The lack of design documentation in particular leads to multiple problems. Newcomers to a project must invest a lot of effort to understand the code. There is an increased risk of failure when developers of related systems cannot correctly understand how to interact with the subject system.

In addition, the lack of documentation makes refactoring and maintenance difficult and error prone. CSE software typically evolves over many years and involves multiple developers [21], as functionality and capabilities are added or extended [22]. The developers need to be able to determine whether the evolved software deviates from the original design intent. To ease this process, developers need tools that help them identify changes that affect the design and determine whether those changes have undesired effects on design integrity. The availability of appropriate design documentation can reduce the likelihood of poor choices during the maintenance process.

2.2. Reverse Engineering. Reverse engineering is a method that transforms source code into a model [23]. ForUML builds upon and expands some existing reverse engineering work. The Dagstuhl middle metamodel(DMM) is a schema for describing the static structure of source code [24]. DMM supports reverse engineering by representing models extracted from source code written in most common OOP languages. We applied the idea of DMM to OO Fortran.

The transformation process in ForUML is based on the XMI format, which provides a standard method of mapping an object model into XML. XMI is an open standard that allows developers and software vendors to create, read, manage, and generate XMI tools. Transforming the model (Fortran code) to XMI requires use of the model driven architecture (MDA) technology [25], a modeling standard developed by the object management group (OMG) [26]. MDA aims to increase productivity and reuse by using separation of concerns and abstraction. A platform independent model (PIM) is an abstract model that contains the information to drive one or more platform specific models (PSMs), including source code, data definition language (DDL), XML, and other outputs specific to the target platform. MDA defines transformations that map from PIMs to PSMs.

The basic idea of using an XMI file to maintain the metadata for UML diagrams was drawn from four reverse engineering tools. Alalfi et al. developed two tools that use XMI to maintain the metadata for the UML diagrams: a tool that generates UML sequence diagrams for web application code [27] and a tool to create UML-entity relationship diagrams for the structured query language (SQL) [28]. Similarly, Korshunova et al. [29] developed *CPP2XMI* to extract various UML diagrams from C++ source code. CPP2XMI generates an XMI document that describes the UML diagram, which is then displayed graphically by DOT (part of the Graphviz framework) [30]. Duffy and Malloy [31] created *libthorin*, a tool to convert C++ source code into UML diagrams. Prior to converting an XMI document into a UML diagram, *libthorin* requires developers to use a third party compiler to compile code into the DWARF (http://www.dwarfstd.org/), which is a debugging file format used to support source level debugging. In terms of Fortran, DWARF only supports Fortran 90, which does not include all object-oriented features. This limitation may cause compatibility problems with different Fortran compilers. Conversely, ForUML is compiler independent and able to generate UML for all types of OO Fortran code.

Doxygen is a documentation tool that can use Fortran code to generate either a simple, textual representation with procedural interface information or a graphical representation. The only OOP class relationship Doxygen supports is inheritance. With respect to our goals, Doxygen has two primary limitations. First, it does not support all OOP features within Fortran (e.g., type-bound procedures and components). Second, the diagrams generated by Doxygen only include class names and class relationships but do not contain other important information typically included in UML class diagrams (e.g., methods, properties). Our work expands upon Doxygen by adding support for OO Fortran and by generating UML diagrams that include all relevant information about the included classes (e.g., properties, methods, and signatures).

There are a number of available tools (both open source and commercial) that claim to transform OO code into UML diagrams (e.g., Altova UModel, Enterprise Architect, StarUML, and ArgoUML). However, in terms of our work, these tools do not support OO Fortran. Although they cannot directly create UML diagrams from OO Fortran code, most of these tools are able to import the metadata describing UML diagrams (i.e., the XMI file) and generate the corresponding UML diagrams. ForUML takes advantages of this feature to display the UML diagrams described by the XMI files it generates from OO Fortran code.

This previous work has contributed significantly to the reverse engineering tools of traditional software. ForUML specifically offers a method to reverse engineering code implemented with modern Fortran, including features in the Fortran 2008 standard. Moreover, the tool was deliberately designed to support important features of Fortran, such as coarrays, procedure overloading, and operator overloading.

2.3. Object-Oriented Fortran. Fortran is an imperative programming language. Traditionally, Fortran code has been developed through a procedural programming approach that emphasizes the procedures and subroutines in a program rather than the data. A number of studies discuss approaches for expressing OOP principles in Fortran 90/95. For example, Decyk described how to express the concepts of data encapsulation, function overloading, classes, objects, and inheritance in Fortran 90 [6, 7, 32]. Moreover, several authors have described the use and syntax of OO features in Fortran 2003 [33–35]. Table 1 presents important Fortran-specific terms along with their OOP equivalent and some examples of Fortran keywords.

The Fortran 2003 compiler standard added support for OOP, including the following OOP principles: dynamic and static polymorphism, inheritance, data abstraction, and encapsulation. Currently, a number of Fortran compiler vendors support all (or almost all) of the OOP features included in the Fortran 2003 standard. These compilers include [36]

(i) NAG (http://www.nag.com/);

(ii) GNU Fortran (http://gcc.gnu.org/fortran/);

(iii) IBM XL Fortran (http://www-142.ibm.com/software/products/us/en/fortcompfami/);

Table 1: Object-oriented Fortran terms (adapted from [12]).

Fortran	OOP equivalent	Fortran keywords
Module	Package	Module
Derived type	Abstract data type (ADT)	Type
Component	Attribute	—
Type-bound procedure	Method	Procedure
Parent type	Parent class	—
Extend type	Child class	Extends
Intrinsic type	Primitive type	For example, real, integer

(iv) Cray (http://www.nersc.gov/users/software/compilers/cray-compilers/);

(v) Intel Fortran (https://software.intel.com/en-us/fortran-compilers).

Fortran 2003 supports procedure overriding where developers can specify a type-bound procedure in a child type that has the same binding name as a type-bound procedure in the parent type. Fortran 2003 also supports user-defined constructors that can be implemented by overloading the intrinsic constructors provided by the compiler. The user-defined constructor is created by defining a generic interface with the same name as the derived type.

Algorithm 1 illustrates a snippet of Fortran 2003 code in which the parent type `shape_` (Line 2) is extended by the type `circle` (Line 7). At runtime the compiler invokes the type-bound procedure `add` (Line 18) whenever an operator "+" (with the specified argument type) is used in the client code. This behavior conforms to polymorphism, which allows a type or procedure to take many object or procedural forms.

Data abstraction is the separation between the interface and implementation of the program. It allows developers to provide essential information about the program to the outside world. In Fortran, the `private` and `public` keywords control access to members of the type. Members defined with `public` are accessible to any part of the program. Conversely, members defined with `private` are not accessible to code outside the module in which the type is defined. In the example, the component `radius` (Line 11) cannot be accessed directly by other programs. Rather, the caller must invoke the type-bound procedure `set_radius` (Line 13).

With the increase in parallel computing, the CSE community needs to utilize the full processing power of all available resources. Fortran 2008 improves the performance for a parallel processing feature by introducing the Coarray model [37]. The Coarray extension allows developers to express data distribution by specifying the relationship between memory images/cores. The syntax of the Coarray is very much like normal Fortran array syntax, except with square brackets instead of parentheses. For example, the statement `integer:: m[*]` (Line 4) declares `m` to be an integer that is sharable across images. Fortran uses normal rounded brackets () to point to data in local memory. Although using Coarray requires the additional syntax, the coarray has been designed to be easy to implement and to provide the compiler scope both to apply its optimizations within each image and among images.

```
(1) module example
(2)    type shape_
(3)      real :: area
(4)      integer :: m[*]
(5)    end type
(6)    !  Inheritance
(7)    type, extends (shape_) :: circle
(8)      !  Data abstraction
(9)      private
(10)     !  Encapsulation
(11)     real :: radius
(12)   contains
(13)     procedure :: set_radius
(14)     procedure :: add
(15)     procedure  :: area
(16)     !  Polymorphism
(17)     generic :: total => area
(18)     generic :: operator(+) => add
(19)   end type
(20)   !  Overloads intrinsic constructor
(21)   interface circle
(22)     module procedure new_circle
(23)   end interface
(24)   !  ...
(25) end module
```

Algorithm 1: Samples code snippet of OOP constructs supported by Fortran 2003.

2.4. Design Patterns. A design pattern is a generic solution to a common software design problem that can be reused in similar situations. Design patterns are made of the best practices drawn from various sources, such as building software applications, developer experiences, and empirical studies. Generally, we can classify the design patterns of the software into classical and novel design patterns. The 23 classical design patterns were introduced by the "Gang of Four" (GoF) [38]. Subsequently, software developers and researchers have proposed a number of novel design patterns targeted at particular domains, for example, parallel programming [39, 40].

In general, a design pattern includes a section known as *intent*. Intent is "a short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issues or problem does it address?" [38]. For example, the intent of the template method pattern requires that developers define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. When using design patterns, developers have to understand the intent of each design pattern to determine

whether the design pattern could provide a good solution to a given problem.

Several researchers have proposed design patterns for computational software implemented with Fortran. For example, Weidmann [41] implemented design patterns to enable sparse matrix computations on NVIDIA GPUs. They then evaluated the benefits of the implementation and reported that the design patterns provided a high level of maintainability and performance. Rouson et al. [12] proposed three new design patterns, called multiphysics design patterns, to implement the differential equations, which are integrated into multiphysics and numerical software. These new design patterns include the semidiscrete, surrogate and template class patterns. Markus demonstrated how some well-known design patterns could be implemented in Fortran 90, 95, and 2003 [42, 43]. Similarly, Decyk et al. [4] proposed the factory pattern in Fortran 95 based on CSE software. These researchers presented the proposed pattern implementation in their particle-in-cell (PIC) methods [44] in plasma simulation software. Decyk and Gardner [45] also described a way to implement the strategy, template, abstract factory, and facade patterns in Fortran 90/95.

3. ForUML

This section describes the rationale and benefits of developing ForUML and details the transformation process used by ForUML.

3.1. The Need for ForUML. The CSE characteristics described in Section 2.1 indicate that CSE developers could benefit from a tool that creates system documentation with little effort. The SE community typically uses reverse engineering to address this problem.

Although there are a number of reverse engineering tools [46] (see Section 2.2), those tools that can be applied to OO Fortran do not provide the full set of documentation required by developers. Therefore, we identified the need for a tool that automatically reverses engineers OO Fortran code into the necessary UML design documentation.

This work is primarily targeted at CSE developers who develop OO Fortran. The ForUML tool will provide the following benefits to the CSE community.

(1) The extracted UML class diagrams should support software maintenance and evolution and help maintainers ensure that the original design intentions are satisfied.

(2) The developers can use the UML diagrams to illustrate software design concepts to their team members. In addition, UML diagrams can help developers visually examine relationships among objects to identify code smells [47] in software being developed.

(3) Because SE tools generally improve productivity, ForUML can reduce the training time and learning curve required for applying SE practices in CSE software development. For instance, ForUML will help developers perform refactoring activities by allowing them to evaluate the results of refactoring using the UML diagrams rather than inspecting the code manually.

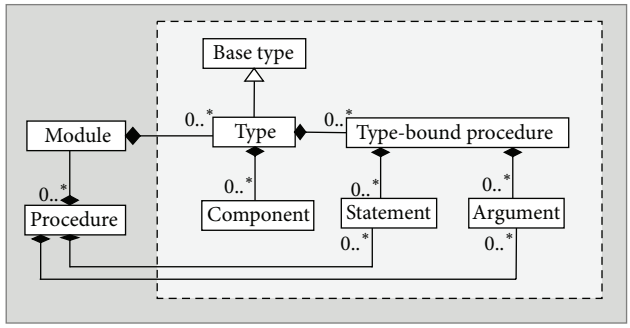


Figure 1: The Fortran model.

Since Fortran 2003 provides all of the concepts of OOP, tools like ForUML can help to place Fortran and other OOP program languages on equal levels.

3.2. Transformation Process. The primary goal of ForUML is to reverse engineering UML class diagrams from Fortran code. By extracting a set of source files, it builds a collection of objects associated with syntactic entities and relations. Object-based features were first introduced in the Fortran 90 language standard. Accordingly, ForUML supports all versions of Fortran 90 and later, which encompasses most platforms and compiler vendors. We implemented ForUML using Java Platform SE6 so that it could run on any client computing systems.

The UML object diagram in Figure 1 expresses the model of the Fortran language. The module object corresponds to Fortran modules, that is, containers holding type and procedure objects. The type-bound procedure and component objects are modeled with a composition association to instances of type. Both the procedure and type-bound procedure objects are composed of argument and statement objects. The generalization relation with base type object leads to the parents in the inheritance hierarchy. When generating the class diagram in ForUML, we consider only the objects inside the dashed-line box that separates object-oriented entities from the module-related entities.

Figure 2 provides an overview of the transformation process embodied in ForUML, comprising the following steps: parsing, extraction, generating, and importing. The following subsections discuss each step in more detail.

3.2.1. Parsing. The Fortran code is parsed by the Open Fortran Parser (OFP) (http://fortran-parser.sourceforge.net/). OFP provides ANTLR-based parsing tools [48], including Fortran grammars and libraries for performing translation actions. ANTLR is a parser generator that can parse language specifications in an EBNF-like syntax, a notation for formally describing programming language syntax, and generate the library to parse the specified language. ANTLR distinguishes three compilation phases: lexical analysis, parsing, and tree walking.

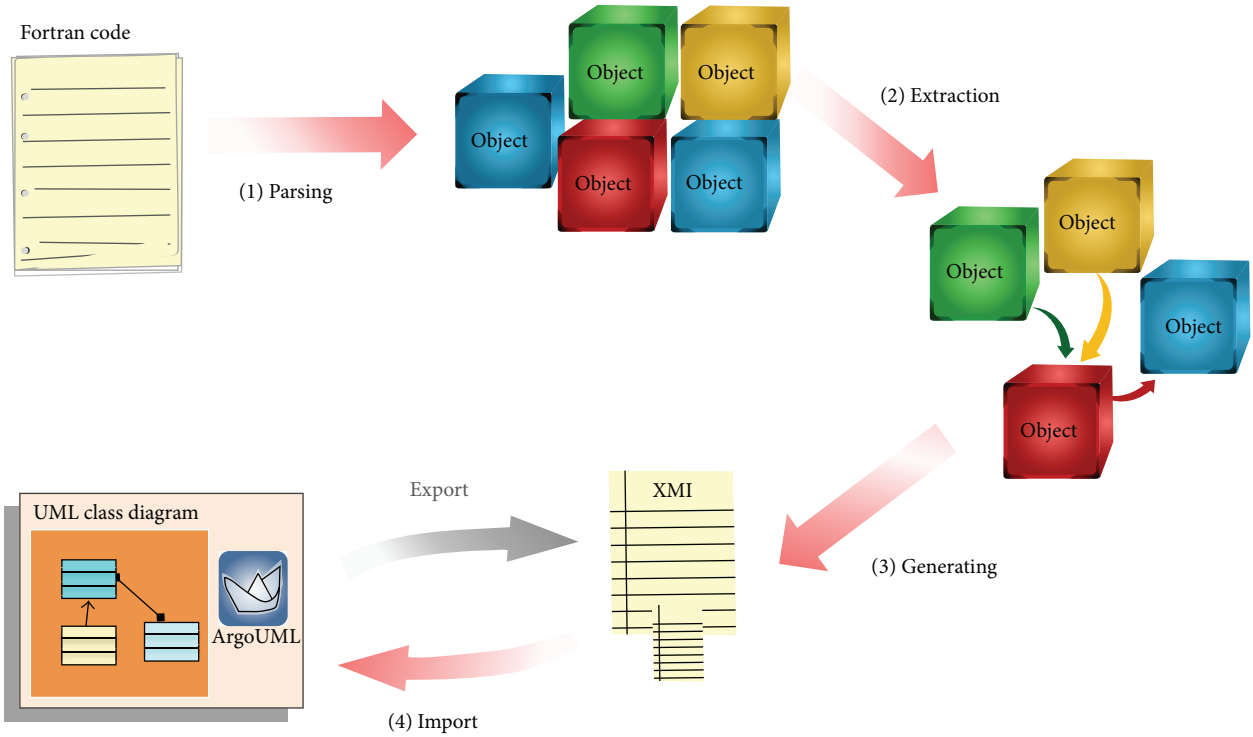


Figure 2: The transformation process.

We have customized the ANTLR libraries to translate particular AST nodes (i.e., type, component, and type-bound procedure) into objects. These AST nodes are only the basic elements of UML class diagrams. In fact, a UML class diagram includes classes, attributes, methods, and relations. The parsing actions include two steps. The first step verifies the syntax in the source file and eliminates source files that have syntax problems. It also eliminates source files that do not contain any instances of type and module. For example, ForUML will eliminate modules that contain only subroutines or functions. After this step, ForUML reports the results to the user via a GUI. In the second step, the parser manipulates all AST nodes, relying on the model described earlier. Note that ForUML only manipulates the selected input source files. Any associated type objects that exist in files not selected by the user are not included in the class diagram.

3.2.2. Extraction. During the extraction process, ForUML excerpts the objects and identifies their relationships. ForUML determines the type of each extracted relationship and maps each relationship to a specific relationship's type object. Based on the example code in Algorithm 1, the type `circle` inherits the type `shape`. As a consequence, the extraction process will create a generalization object. ForUML supports two relationship types: composition and generalization.

(i) Composition represents the whole-part relationship. The lifetime of the part classifier depends on the lifetime of the whole classifier. In other words, a composition describes a relationship in which one class is composed of many other classes. In our case, the composition association will be produced when a type object refers to another type object in the component. The association refers to a type not provided by the user and as a result it does not appear in the class diagram. In the UML class diagram, a composition relationship appears as a solid line with a filled diamond at the association end that is connected to the whole class.

(ii) Generalization represents an *is-a* relationship between a general object and its derived specific objects, commonly known as an inheritance relation. Similar to the composition association, the generalization association is not shown in the class diagram if the source file of the base type is not provided by the user. This relationship is represented by a solid line with a hollow unfilled arrowhead that points from the child class to the parent class.

3.2.3. Generating. We developed the XMI generator module to convert the extracted objects into XMI notation based on our defined rules for mapping the extracted objects to the proper XMI notation. The rules for mapping the extracted objects and XMI document are specified in Table 2. In addition to these rules, we developed new stereotype notations for the constructor, coarray constructs, type-bound procedure overloading, and operator overloading, such as ≪`Constructor`≫, ≪`Coarray`≫, ≪`Overloading`≫, and ≪`Overloading of +`≫.

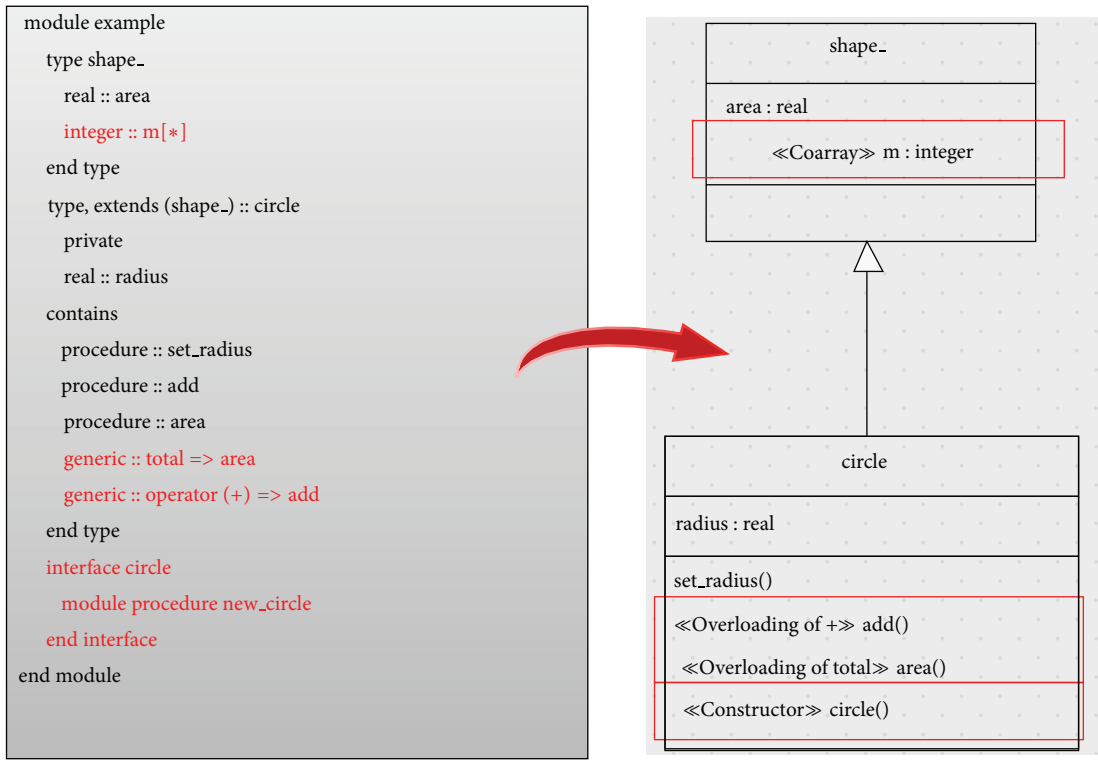


Figure 3: Sample code snippet of Fortran supported by ForUML.

Table 2: Fortran to XMI conversion rules.

Fortran	XMI elements
Derived type	UML: class
Type-bound Procedure	UML: operation
Dummy argument	UML: parameter
Component	UML: attribute
Intrinsic type	UML: DataType
Parent type	UML: Generalization.parent
Extended type	UML: Generalization.child
Composite	UML: association (the aggregation property as "composite")

Figure 3 provides the sample Fortran code without procedure implementation and its generated class diagram including stereotypes.

3.2.4. Importing. To visually represent the extracted information as a UML class diagram, we import the XMI document into a UML modeling tool. We decided to include a UML modeling tool directly in ForUML to prevent the user from having to install or use a second application for visualization. We chose to include ArgoUML as the UML visualization tool in the current version of ForUML. We had to modify the ArgoUML code to allow it to automatically import the XMI document. Of course, if a user would prefer to use a different modeling tool, he or she can manually import the generated XMI file into any tool that supports the XMI format.

After importing the XMI file, ArgoUML's default view of the class diagram does not show any entities in the editing pane. Like the WYSIWYG ("what you see is what you get") concept, the user needs to drag the target entity from a hierarchical view to the editing pane. To help with this problem, we added features so that ArgoUML will show all entities in the editing pane immediately after successfully importing the XMI document. Note that the XMI document does not specify how to present the elements graphically, so ArgoUML automatically adjusts the diagram when rendering the graphics. Each graphical tool may have its own method for generating the graphical layout of diagrams. The key reasons why we chose to integrate ArgoUML into ForUML are that (1) it has seamless integration properties as an open source and Java implementation; (2) it has sufficient documentation; and (3) it provides sufficient basic functions required by the users (e.g., export graphics, import/export XMI, zooming).

ForUML provides a Java-based user interface for executing the command. To create a UML class diagram, the user performs these steps.

(1) Select the Fortran source code

(2) Select the location to save the output.

(3) Open the UML diagram.

Figures 4–7 show screenshots from the ForUML tool. Figure 4 presents the graphical user interface (GUI) of

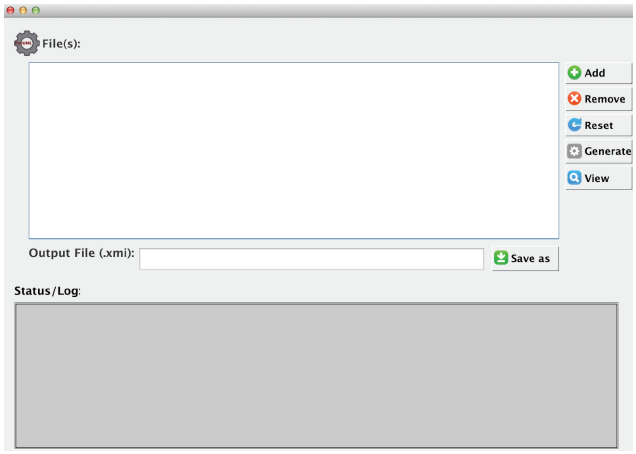


Figure 4: A graphical user interface of ForUML.

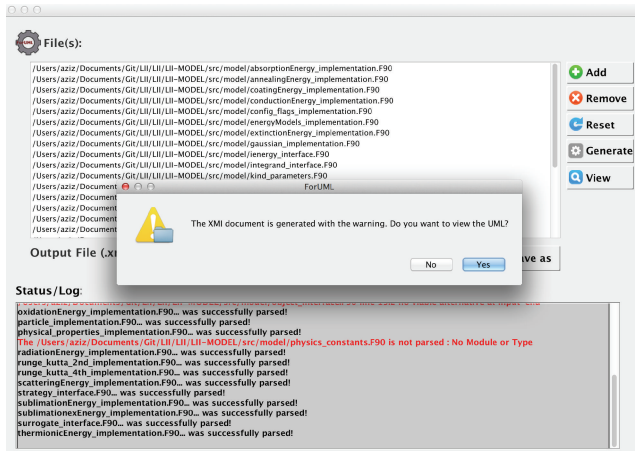


Figure 6: Generating the XMI.

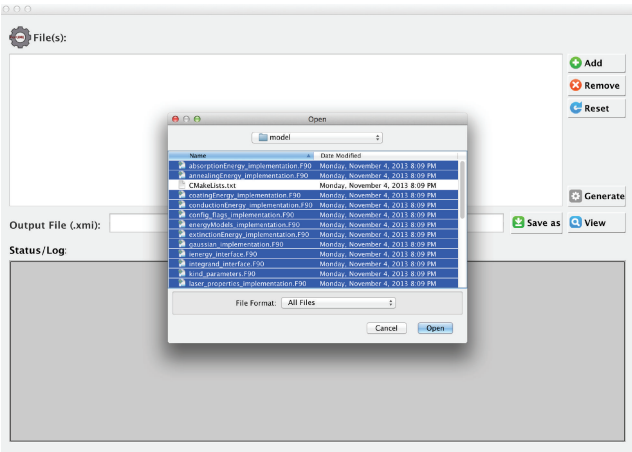


Figure 5: Selection of the Fortran code.

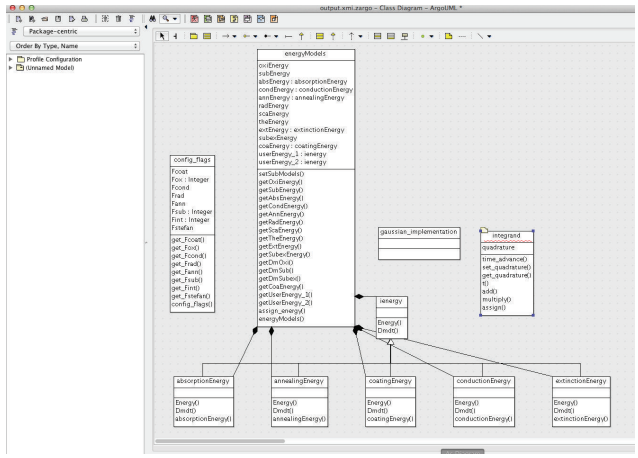


Figure 7: Viewing the UML class diagram.

ForUML. Figure 5 illustrates how a user can select multiple Fortran source files for input to ForUML. The *Add* button opens a new window to select target file(s). Users can remove the selected file(s) by selecting the *Remove* button. The *Reset* button clears all selected files. After selecting the source files, the user chooses the location to save the generated XMI document (.xmi file). The *Generate* button activates the transformation process. During the process, the user can see whether each given source file is successfully parsed or not (Figure 6). Once the XMI document is successfully generated, the user can view the class diagram by selecting the *View* button. Figure 7 illustrates the UML class diagram that is automatically represented in the editing pane with the ArgoUML. ArgoUML allows users to refine the diagram and then decide to either save the project or export the XMI document, which contains all the modified information.

4. Evaluation

To assess the effectiveness of ForUML, we conducted some small experiments to gather data about its accuracy in extracting UML constructs from code. This section also provides some lessons learned from the studies and feedback from the SEC-HPC'13 workshop audience.

4.1. Controlled Experiment. The following subsections provide the details of a controlled experiment to evaluate ForUML. The accompanying website (http://aziz.students.cs.ua.edu/foruml-eval.htm) contains all of the class diagrams. The website also provides the ForUML executable (source code is not available yet) for download.

4.1.1. Experimental Design. We evaluated the *accuracy* of ForUML on five OO Fortran software packages by adopting the definitions of *recall* and *precision* defined by Tonella and Potrich [49].

(i) *Recall* measures the percentage of the various objects, that is, type, components, type-bound procedure, and associations, in the source code correctly identified by ForUML.

(ii) *Precision* measures the percentage of the objects identified by ForUML that are correct when compared with the source code.

We performed the evaluations as follows.

(1) We manually inspected the source code to document the number of relevant objects in each package. Note that we performed this step multiple times to ensure that the numbers were not biased by human error.

(2) We ran ForUML on each software package and documented the number of relevant objects included in the generated class diagram.

(3) To compute *recall*, we compared the number of objects manually identified in the source code (Step 1) with the number identified by ForUML (Step 2).

(4) To compute *precision*, we determined whether there were any objects produced by ForUML (Step 2) that we did manually observe in the code (Step 1).

(5) We investigated whether the generated class diagrams could present the design pattern classes existing in the subject systems.

4.1.2. Subject Systems. The five software packages we used in the experiments were (1) ForTrilinos (http://trilinos.sandia.gov/packages/fortrilinos/); (2) CLiiME; (3) PSBLAS (http://www.ce.uniroma2.it/psblas/); (4) MLD2P4 (http://www.mld2p4.it/); and (5) MPFlows. We selected these software packages because they were intentionally developed for use in the CSE environment. Two of the software packages (CLiiME and MPFlows) are not yet publicly available. A description of each software package follows.

(1) ForTrilinos: ForTrilinos consists of an OO Fortran interface to expand the use of Trilinos (http://trilinos.sandia.gov/) into communities that predominantly write Fortran. Trilinos is a collection of parallel numerical solver libraries for the solution of CSE applications in the HPC environment. To provide portability, ForTrilinos extensively exploits the Fortran 2003 standard's support for interoperability with C. ForTrilinos includes 4 subpackages (epetra, aztecoo, amesos, and fortrilinos), 36 files, and 36 modules.

(2) *CLiiME*: community laser induced incandescence modeling environment (CLiiME) is a dynamic simulation model that predicts the temporal response of laser-induced incandescence from carbonaceous particles. CLiiME is implemented with Fortran 2003. It contains 2 subpackages (model and utilities), 30 files, and 29 modules. Additionally, this application contains three design patterns, including factory method, strategy, and surrogate.

(3) *PSBLAS*: PSBLAS 3.0 is a library for parallel sparse matrix computations, mostly dealing with the iterative solution of sparse linear system via a distributed memory paradigm. The library assumes a data distribution consistent with a domain decomposition approach, where all variables and equations related to a given portion of the computation domain are assigned to a process; the data distribution can be specified in multiple ways allowing easy interfacing with many graph partitioning procedures. The library design also provides data management tools allowing easy interfacing with data assembly procedures typical of finite elements and finite volumes discretization. Researchers have used versions of the library in various application domains, mostly in fluid dynamics and structural analysis, where it has been successfully used to solve linear system with millions of unknowns arising in complex simulations. The PSBLAS library version 3.0 is implemented with Fortran 2003. PSBLAS contains 10 subpackages (prec, psblas, util, impl, krylov, tools, serial, internals, comm, and modules), 476 files, and 135 modules.

(4) *MLD2P4*: multi-level domain decomposition parallel preconditioners package based on PSBLAS (MLD2P4 version 1.2) is a package of parallel algebraic multilevel preconditioners. This package provides a variety of high-performance preconditioners for the Krylov methods of PSBLAS. A preconditioner is an operator capable of reducing the number of iterations needed to achieve convergence to the solution of a linear system; multilevel preconditioners are very powerful tools especially suited for problems derived from elliptic PDEs. This package is implemented with object-based Fortran 95. The MLD2P4 contains only one package (miprec), 117 files and 9 modules.

(5) *MPFlows*: multiphase flows (MPFlows) is a package developed for computational modeling of spray applications. MPFlows is implemented with Fortran 2003/2008. The use of coarrays within this application enables scalable CSE software package that works without requiring the use of external parallel libraries. MPFlows contains 2 subpackages (spray and utilities), 12 files, and 12 modules. Note that this package contains two design patterns, including strategy and surrogate.

4.1.3. Analysis. Table 3 shows the results of experiments. Each cell represents the recall as a ratio between extracted data and actual data. The results show that the recall reaches 100% for all subpackages. Overall, there was only one error in *precision* in the *ForTrilinos* subpackage of *ForTrilinos*. Our analysis of the code identified a conditional preprocessor statement (specified by the `#if` statement) as the source of the problem. ForUML currently does not handle preprocessor directives. During the experiments, only 6 files were not parsed (0.89% of all files). The notification messages informed the users which files were not processed and specifically why each file could not be processed. Based on code inspection, we found four files that do not conform to the Fortran model described earlier (Figure 1). Those files do not have the `module` keyword that is the starting point for the transformation process. Other files exceptions were due to ambiguous syntax; for example, Fortran keywords were used as part of a procedure name (e.g., `print`, `allocate`). Table 3 only shows the results for packages that have the

Table 3: Evaluation of ForUML: recall (extracted data/actual data).

Packages	Subpackages	Type	Procedure	Component	Inheritance	Composition
ForTrilinos	Epetra	16/16	304/304	17/17	12/12	2/2
	Aztecoo	1/1	12/12	1/1	0/0	0/0
	Amesos	1/1	7/7	1/1	0/0	0/0
	ForTrilinos	48/48	11/11	139/139	4/4	4/4
CLiiME	Model	23/23	167/167	61/61	32/32	32/32
PSBLAS	Modules	50/50	1309/1309	160/160	34/34	28/28
	prec	20/20	208/208	28/28	24/24	12/12
MLDP4	miprec	11/11	0/0	67/66	0/0	10/10
MPFlows	Spray	10/10	55/55	29/29	2/2	3/3
	Overall	180/180 (100%)	2073/2073 (100%)	503/503 (100%)	108/108 (100%)	91/91 (100%)

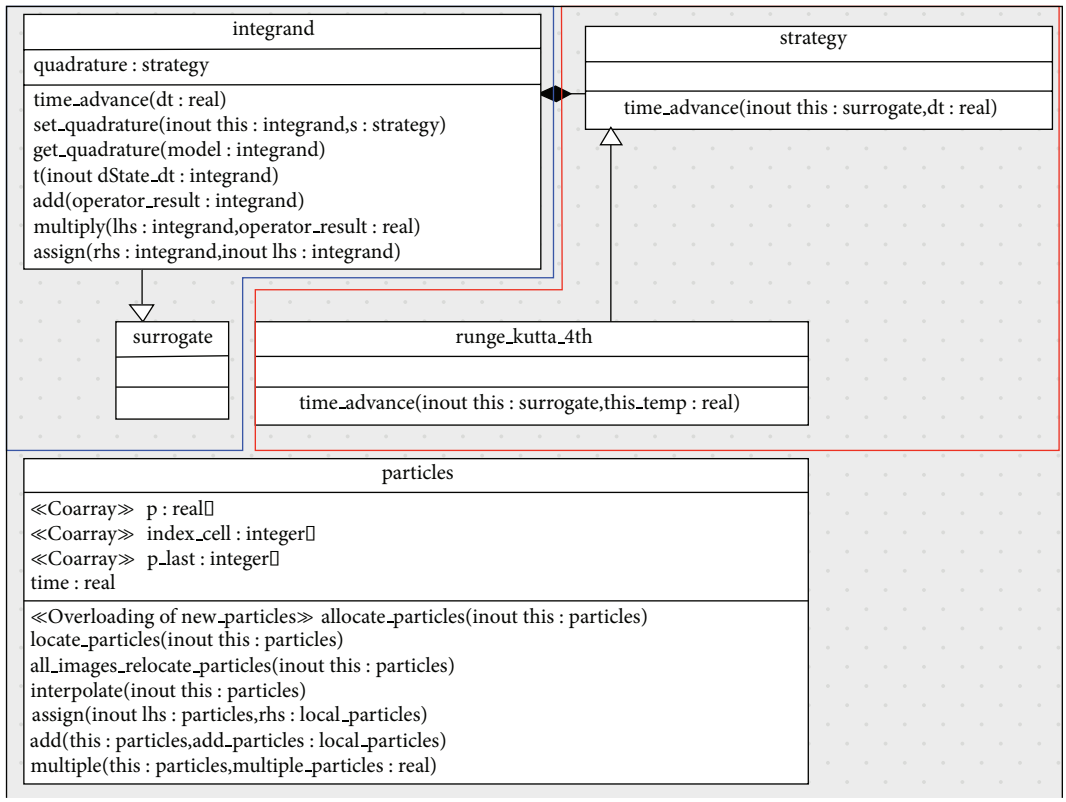


Figure 8: The class diagram (partial): MPFlows.

type construct. We only evaluated the correctness of ForUML current capabilities.

Figure 8 provides an example of an excerpt from a class diagram derived from MPFlows. This diagram consists of the implementation of two design patterns, including strategy (inside the red box) and surrogate (inside the blue box) patterns. In the strategy pattern, an interface class `strategy` defines only the time integration method, deferring to subclasses the implementation of the actual quadrature schemes. The concrete strategy class (derived class) `runge_kutta_4th` provides the algorithm that presents a part of the `time_advance` method declared by the `strategy` interface. Next, the surrogate pattern is very similar in concept to an ATM. An ATM holds a surrogate database for bank information that exists in another place. The bank's customer can perform transactions through the ATM and circumvent a visit to the bank. The implementation of the surrogate pattern introduces the `surrogate` abstract class (virtual class in C++). The class `integrand` has a component of class `strategy` meaning that

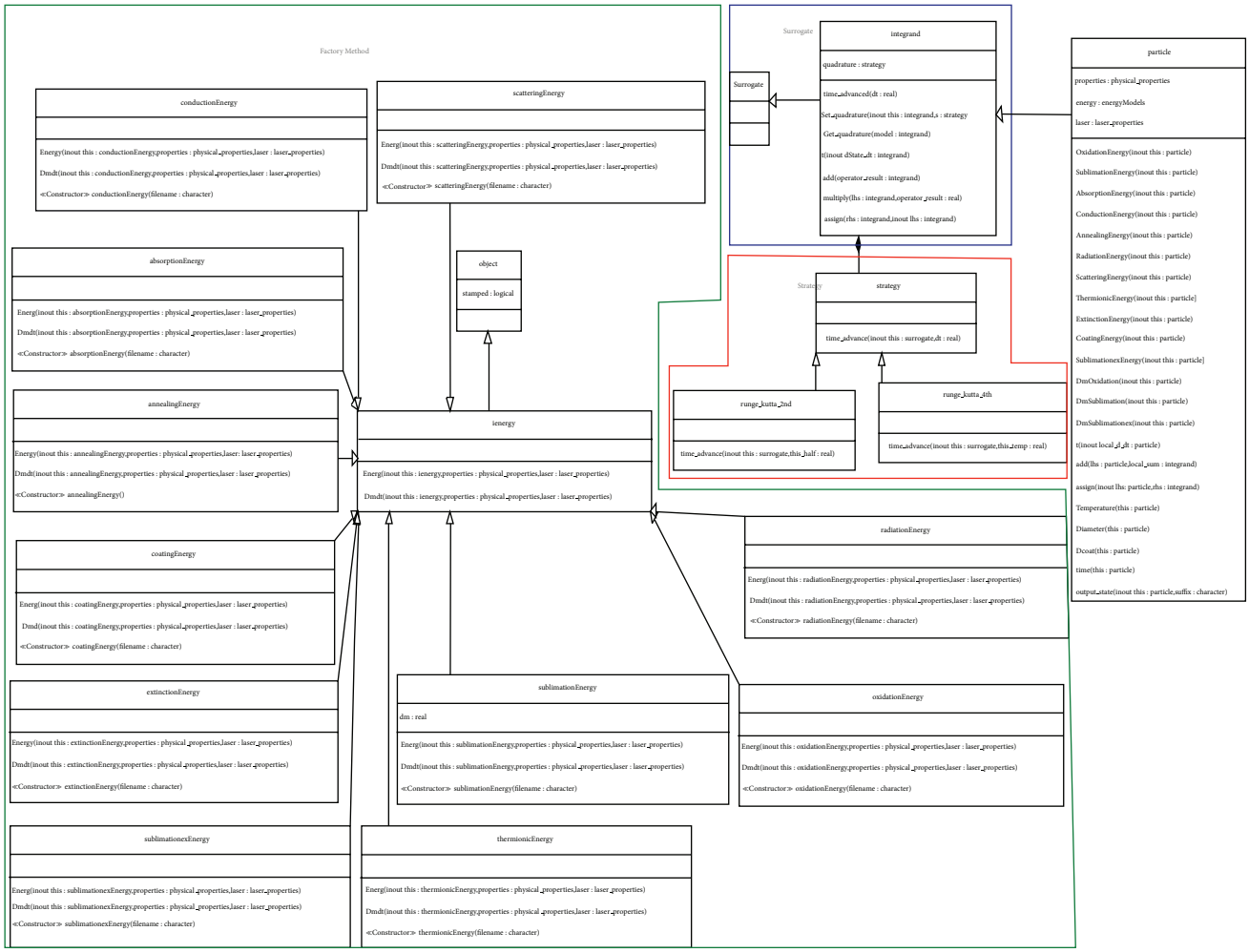


Figure 9: The class diagram (partial): CLiiME.

the `surrogate` allows us to pass an `integrand` child class dummy argument to the type-bound procedures implemented in `runge_kutta_4th`. The class `particle` contains components and type-bound procedures computing the energy of the particle. In Fortran, each dummy argument has three possible intent attributes including IN, OUT, and INOUT. Therefore, each parameter, which is passed to the operation in the diagram, needs to be specified with a specific intent. In the class diagram, the keyword IN is omitted because ArgoUML assumes that a parameter has the IN by default.

4.2. Experience. The following subsections describe our experiences using ForUML on a real CSE project and discuss feedback on ForUML we received during the SE-HPCCSE'13 workshop.

4.2.1. CLiiME Project. ForUML played a significant role in the development of the CLiiME package [50]. The developers used ForUML to validate the design after each code refactoring process. The developers compared the class diagram produced by ForUML with the originally agreed upon design. After comparison, they determined instances in which the code implementation deviated from the design. Instead of inspecting the source code manually, the developers were able to make the comparison/decision with less effort. Also, the developers were able to use the extracted UML diagrams to identify code smells, places where the code might induce some defects in the future. For instance, we inspected the UML class diagrams and identified places where classes had too many type-bound procedure or procedures with too many arguments, all of which we corrected during the refactoring process.

This project also deployed three design patterns. Figure 9 presents the UML class diagram of the CLiiME project, including the strategy (inside the red box), surrogate (inside the blue box), and factory method (inside the green box) patterns. The factory method pattern indicates encapsulating the subclass selection (∗Energy class) and object construction processes into one class (ienergy). We used ForUML to confirm the correct implementation of those three design patterns rather than reviewing the source code. In addition to helping CLiiME developers, ForUML also influenced the

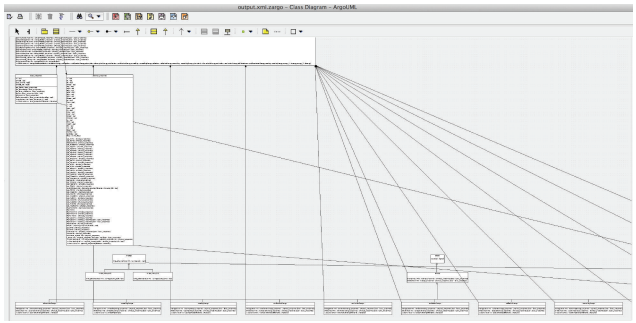


Figure 10: Example of larger classes.

development of PSBLAS version 3.1, by allowing a comprehensive and unitary view of the project.

The UML diagram must be properly arranged to foment design comprehension. A large class diagram that contains several classes and relationships requires additional effort from users' as they try to assimilate all the information. Unfortunately, the built-in function layout in ArgoUML does not refine the layout in diagrams that contain numerous elements. Although ArgoUML provides the ability to zoom in or zoom out, large diagrams can still be difficult to view. Figure 10 shows an example of a UML class diagram generated by ForUML that includes large classes. These problems can be addressed by dividing the collection of classes into smaller packages, which should improve the diagram's understandability. Another option is to provide different settings for the information included in the class diagrams, allowing a user to create diagrams with the level of detail required for a particular task. This option can ease the development and/or maintenance process by eliminating irrelevant information.

4.2.2. SEC-HPC'13 Workshop. In addition to our own experiences, we can make some observations based on the discussions during the SE-HPCCSE'13 workshop regarding the use of UML in CSE applications. UML helps partition the coding workloads in large projects. For larger projects, especially libraries, it is a matter of dwelling on the "use cases" and designing an interface perhaps with UML. Then feature coding tasks can be distributed to other developers. In contrast, CSE has been reluctant to adopt object-oriented design, whereas in other standard mathematics, linear algebra design bears some similarly to OOP considering larger mathematical structures as objects. Many audiences believed that better SE practices, including adoption ForUML could lead to a better adaptation of codes to multiple architectures. However, one reason for the lack of advance SE in CSE is that CSE developers try to use UML for everything. The audience suggested that other domain specific languages (DSLs) could be useful targets for generating information from legacy code. Further, during the workshop's discussion, there were some questions that inspired us to study the impact of ForUML on the CSE community. We believe that we can find answers to these questions by conducting human-based studies of ForUML. Below is a list of questions that arose during the workshop.

(i) Is UML really useful for CSE developers?

(ii) Can ForUML and UML support larger application sizes and multiple developers?

(iii) Many graphical design models serve multiple purposes. Some users can convey a high-level design for discussion, and others want to display the low-level of design. In the context of CSE software development, does UML serve all these needs well?

(iv) Which aspects of the CSE application should be documented in the UML?

5. Discussion

Based on the experimental results, ForUML provided quite precise outputs. ForUML was able to automatically transform the source code into correct UML diagrams. To illustrate the contributions of ForUML, Table 4 compares ForUML with other visualization-based tools [18] that have features to support program comprehension tasks. Based on this table, one of the unique contributions of ForUML is its ability to reverse engineered OO Fortran code. ForUML integrates the capabilities of ArgoUML to visually display the class diagram.

We believe that ForUML can be used by three types of people during the software development process, especially for CSE software.

(1) Stakeholders or customers: ForUML generates documentation that describes the high-level structure of the software. This documentation should make communication between developers and the stakeholders or customers more efficient.

(2) Developers: ForUML helps developers extract design diagrams from their code. Developers might need to validate whether the code under development conforms to the original design. Similarly, when developers refactor the code, they need to ensure that the refactoring does not break exiting functionality or decompose the architecture.

(3) Maintainers: they need a document that provides adequate design information to enable them to make good decisions. In particular, maintainers who are familiar with other OOP languages can understand a system implemented with OO Fortran.

However, ForUML has a few limitations that must be addressed in the future as follows.

(1) Provide more relationships: two other relationships that we frequently found in the Fortran applications are as follows.

(i) Dependency: in practice, dependency is most commonly used between elements (e.g., packages, folders) that contain other elements located in different packages. The relationship

Table 4: A brief comparison between UML tools.

Features	Rose enterprise [53]	Doxygen	Libthorin	ForUML + ArgoUML	Rigi [54]
Visualization	UML	Graph	UML	UML	Graph
Reverse eng. (Fortran)	No	No	Ver.90	Yes	No
Hide/show detail	Yes	No	Yes	No	No
Inheritance	Yes	No	Yes	Yes	No
Layout	A/M	A	A	A/M	A

Note: automatically adjusted (A) and manually adjusted (M).

is represented by a dashed line with an arrow pointing toward a class that is an argument in a procedure that is bound to another class.

(ii) Realization: it refers to the links between either the interface or abstract and its implementing classes. A dashed line is connected to an open triangle for a type that extends an abstract type.

Note that although the current version of ForUML does not support these relation types, the users can edit the relationships in the ArgroUML after importing the XMI document.

(2) Incorporation of other UML visualization tools: currently, ForUML integrates ArgoUML as the CASE tool. We plan to build different interfaces to integrate with other UML tools, so users can select their tool of preference. Although many UML CASE tools support the use of XMI documents, there are several XMI versions defined by object management group (OMG) and different tools support different versions. We also plan to develop a plugin for Photran (http://www.eclipse.org/photran/), to allow users to automatically generate UML diagrams within the IDE.

(3) Generate UML sequence diagram: a single diagram does not sufficiently describe the entire software system. Sequence diagrams are widely used to represent the interactive behavior of the subject system [51]. To create UML sequence diagrams, we would have to augment the ForUML extractor to build the necessary relationships among objects necessary for the generator to create the corresponding XMI code.

6. Conclusion and Future Work

This paper presents and evaluates the ForUML tool that can be used for extracting UML class diagram from Fortran code. Fortran is one of the predominant programming languages used in the CSE software domain. ForUML generates a visual representation of software implemented in OO Fortran in the same way as is done in other, more traditional OO languages. Software developers and practitioners can use ForUML to improve the program comprehension process. ForUML will help CSE developers adopt better SE approaches for the development of their software. Similarly, software engineers who are not familiar with scientific principles may be able to understand a CSE software system just based on information in the generated UML class diagrams. Currently, ForUML can produce an XMI document that describes the UML class diagrams. The tool supports the inheritance and composition relationships that are the most common relationships found in software systems. The tool integrates ArgoUML, an open source UML modeling tool to allow users to view and modify the UML diagrams without installing a separate UML modeling tool.

We have run ForUML on five CSE software packages to generate class diagrams. The experimental results showed that ForUML generates highly accurate UML class diagrams from Fortran code. Based on the UML class diagrams generated by ForUML, we identified a few limitations of its capabilities. To augment the results of experiments, we have created a website that contains all of the diagrams generated by ForUML along with a video demonstrating the use of ForUML. We plan to add more diagrams to the website as we run ForUML on additional software packages. We believe that ForUML conforms to Chikofsky and Cross II [52] objectives of reverse engineering, which are identified as follows: (1) to identify the system's component and their relationships and (2) to represent the system in another form or at a higher level of abstraction.

In the future, we plan to address the limitations we have identified. We also plan to conduct human-based studies to evaluate the effectiveness and usability of ForUML by other members of the CSE software developer community. To encourage wider adoption and use of ForUML, we are investigating the possibility of releasing it as open source software. This direction can help us to get more feedback about the usability and correctness of the tool. Demonstrating that ForUML is a realistic tool for large-scale computational software will make it an even more valuable contribution to both the SE and CSE communities.

Conflict of Interests

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

Acknowledgments

The authors gratefully thank Dr. Damian W. I. Rouson, at Stanford University, and Dr. Hope A. Michelsen, member of the Combustion Chemistry Department at Sandia National Laboratories, for their useful comments and helpful discussions which were extremely valuable.

References

[1] National Science Foundation, *Cyberinfrastructure for 21st Century Science and Engineering Advanced Computing Infrastructure (Vision and Strategies Plan)*, 2012, http://www.nsf.gov/pubs/2012/nsf12051/nsf12051.pdf.

[2] J. C. Carver, "Software engineering for computational science and engineering," *Computing in Science and Engineering*, vol. 14, no. 2, Article ID 6159198, pp. 8–11, 2011.

[3] J. H. Marburget, "Report of the high-end computing revitalization task force (hecrtf)," Tech. Rep., National Coordination Office for Information Technology Research and Development, 2004.

[4] V. K. Decyk, C. D. Norton, and H. J. Gardner, "Why fortran?" *Computing in Science and Engineering*, vol. 9, no. 4, Article ID 4263269, pp. 68–71, 2007.

[5] E. Akin, *Object-Oriented Programming via Fortran 90/95*, Cambridge University Press, Cambridge, UK, 2003.

[6] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "Expressing object-oriented concepts in Fortran 90," *ACM SIGPLAN Fortran Forum*, vol. 16, no. 1, pp. 13–18, 1997.

[7] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to support inheritance and run-time polymorphism in Fortran 90," *Computer Physics Communications*, vol. 115, no. 1, pp. 9–17, 1998.

[8] D. Barbieri, V. Cardellini, S. Filippone, and D. Rouson, "Design patterns for scientific computations on sparse matrices," in *Proceedings of the International Conference on Parallel Processing (Euro-Par '11)*, vol. 7155 of *Lecture Notes in Computer Science*, pp. 367–376, Springer, Berlin, Germany, 2012.

[9] S. Filippone and A. Buttari, "Object-oriented techniques for sparse matrix computations in Fortran 2003," *ACM Transactions on Mathematical Software*, vol. 38, no. 4, article 23, 2012.

[10] K. Morris, D. W. I. Rouson, M. N. Lemaster, and S. Filippone, "Exploring capabilities within ForTrilinos by solving the 3D Burgers equation," *Scientific Programming*, vol. 20, no. 3, pp. 275–292, 2012.

[11] D. W. Rouson, J. Xia, and X. Xu, "Object construction and destruction design patterns in fortran 2003," *Procedia Computer Science*, vol. 1, no. 1, pp. 1495–1504, 2003.

[12] D. W. I. Rouson, H. Adalsteinsson, and J. Xia, "Design patterns for multiphysics modeling in Fortran 2003 and C++," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, article 3, 2010.

[13] Z. Merali, "Computational science: ...Error," *Nature*, vol. 467, no. 7317, pp. 775–777, 2010.

[14] A. Nanthaamornphong, K. Morris, and S. Filippone, "Extracting uml class diagrams from object-oriented fortran: Foruml," in *Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE '13)*, pp. 9–16, Denver, Colo, USA, November 2013.

[15] J. C. Carver, "Report: the second international workshop on software engineering for CSE," *Computing in Science and Engineering*, vol. 11, no. 6, Article ID 5337640, pp. 14–19, 2009.

[16] G. V. Wilson, "What should computer scientists teach to physical scientists and engineers?" *IEEE Computational Science & Engineering*, vol. 3, no. 2, pp. 46–55, 1996.

[17] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: a series of case studies," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pp. 550–559, Minneapolis, Minn, USA, May 2007.

[18] M.-A. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[19] M. J. Pacione, "Software visualisation for object-oriented program comprehension," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 63–65, May 2004.

[20] J. Segal, "Professional end user developers and software development knowledge," Tech. Rep., Open University, England, UK, 2004.

[21] M. T. Sletholt, J. E. Hannay, D. Pfahl, and H. P. Langtangen, "What do we know about scientific software development's agile practices?" *Computing in Science and Engineering*, vol. 14, no. 2, Article ID 6081842, pp. 24–36, 2012.

[22] R. N. Britcher, "Re-engineering software: a case study," *IBM Systems Journal*, vol. 29, no. 4, pp. 551–567, 1990.

[23] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison Wesley Longman, Boston, Mass, USA, 1999.

[24] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder, "The dagstuhl middle metamodel: a schema for reverse engineering," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 7–18, 2004.

[25] OMG, OMG Model Driven Architecture (MDA), 1997, http://www.omg.org/mda/.

[26] Object Management Group (OMG), 1997, http://www.omg.org.

[27] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Automated reverse engineering of UML sequence diagrams for dynamic web applications," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*, pp. 287–294, Denver, Colo, USA, April 2009.

[28] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "SQL2XMI: reverse engineering of UML-ER diagrams from relational database schemas," in *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pp. 187–191, Antwerp, Belgium, October 2008.

[29] E. Korshunova, M. Petkovic, M. van den Brand, and M. Mousavi, "CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pp. 297–298, Benevento, Italy, October 2006.

[30] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo, "A technique for drawing directed graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–230, 1993.

[31] E. B. Duffy and B. A. Malloy, "A language and platform-independent approach for reverse engineering," in *Proceedings of the 3rd ACIS International Conference on Software Engineering Research, Management and Applications (SERA '05)*, pp. 415–422, Pleasant, Mich, USA, August 2005.

[32] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to express C++ concepts in Fortran 90," *Scientific Programming*, vol. 6, no. 4, pp. 363–390, 1997.

[33] W. S. Brainerd, *Guide to Fortran 2003 Programming*, Springer, 1st edition, 2009.

[34] M. Metcalf, J. Reid, and M. Cohen, *Modern Fortran Explained*, Oxford University Press, New York, NY, USA, 4th edition, 2011.

[35] D. Rouson, J. Xia, and X. Xu, *Scientific Software Design: The Object-Oriented Way*, Cambridge University Press, New York, NY, USA, 1st edition, 2011.

[36] I. D. Chivers and J. Sleightholme, "Compiler support for the Fortran 2003 and 2008 Standards Revision 11," *ACM SIGPLAN Fortran Forum*, vol. 31, no. 3, pp. 17–28, 2012.

[37] J. Reid, "Coarrays in the next fortran standard," *SIGPLAN Fortran Forum*, vol. 29, no. 2, pp. 10–27, 2010.

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Longman Publishing, Boston, Mass, USA, 1995.

[39] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, Addison-Wesley Professional, 1st edition, 2004.

[40] J. L. Ortega-Arjona, *Patterns for Parallel Software Design*, John Wiley & Sons, 1st edition, 2010.

[41] M. Weidmann, "Design and performance improvement of a real-world, object-oriented C++ solver with STL," in *Scientific Computing in Object-Oriented Parallel Environments*, Y. Ishikawa, R. Oldehoeft, J. Reynders, and M. Tholburn, Eds., vol. 1343 of *Lecture Notes in Computer Science*, pp. 25–32, Springer, Berlin, Germany, 1997.

[42] A. Markus, "Design patterns and Fortran 90/95," *ACM SIGPLAN Fortran Forum*, vol. 25, no. 1, pp. 13–29, 2006.

[43] A. Markus, "Design patterns and Fortran 2003," *ACM SIGPLAN Fortran Forum*, vol. 27, no. 3, pp. 2–15, 2008.

[44] H. Neunzert, A. Klar, and J. Struckmeier, "Particle methods: theory and applications," Tech. Rep. 95-113, Fachbereich Mathematik, Universitat Kaiserslautern, Kaiserslautern, Germany, 1995.

[45] V. K. Decyk and H. J. Gardner, "Object-oriented design patterns in Fortran 90/95: mazev1, mazev2 and mazev3," *Computer Physics Communications*, vol. 178, no. 8, pp. 611–620, 2008.

[46] H. A. Muller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, pp. 47–60, Limerick, Ireland, June 2000.

[47] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman, Boston, Mass, USA, 1999.

[48] T. J. Parr and R. W. Quong, "ANTLR: a predicated-LL(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[49] P. Tonella and A. Potrich, "Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)*, pp. 376–385, Florence, Italy, November 2001.

[50] A. Nanthaamornphong, K. Morris, D. W. I. Rouson, and H. A. Michelsen, "A case study: agile development in the community laser-induced incandescence modeling environment (CLiiME)," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, pp. 9–18, San Francisco, Calif, USA, May 2013.

[51] L. C. Briand, Y. Labiche, and Y. Miao, "Towards the reverse engineering of UML sequence diagrams," in *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 57–66, Victoria, Canada, November 2003.

[52] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[53] IBM, *Rational Rose Enterprise*, 2013, http://www-03.ibm.com/software/products/en/enterprise/.

[54] Department of Computer Science University of Victoria, Rigi, 2001, http://www.rigi.cs.uvic.ca/rigi/blurb/rigi-blurb.html.