Scientific Programming 20 (2012) 275–292
DOI 10.3233/SPR-2012-0353
IOS Press

Exploring capabilities within ForTrilinos by solving the 3D Burgers equation

Karla Morris [a,*], Damian W.I. Rouson [a], M. Nicole Lemaster [a] and Salvatore Filippone [b]
[a] *Sandia National Laboratories, Livermore, CA, USA*
[b] *Università di Roma "Tor Vergata", Roma, Italy*

Abstract. We present the first three-dimensional, partial differential equation solver to be built atop the recently released, open-source ForTrilinos package (http://trilinos.sandia.gov/packages/fortrilinos). ForTrilinos currently provides portable, object-oriented Fortran 2003 interfaces to the C++ packages Epetra, AztecOO and Pliris in the Trilinos library and framework [*ACM Trans. Math. Softw.* **31**(3) (2005), 397–423]. Epetra provides distributed matrix and vector storage and basic linear algebra calculations. Pliris provides direct solvers for dense linear systems. AztecOO provides iterative sparse linear solvers. We demonstrate how to build a parallel application that encapsulates the Message Passing Interface (MPI) without requiring the user to make direct calls to MPI except for startup and shutdown. The presented example demonstrates the level of effort required to set up a high-order, finite-difference solution on a Cartesian grid. The example employs an abstract data type (ADT) calculus [*Sci. Program.* **16**(4) (2008), 329–339] that empowers programmers to write serial code that lower-level abstractions resolve into distributed-memory, parallel implementations. The ADT calculus uses compilable Fortran constructs that resemble the mathematical formulation of the partial differential equation of interest.

Keywords: ForTrilinos, Trilinos, Fortran 2003/2008, object oriented programming

1. Introduction

The story of modern, scientific programming is the story of modern, mainstream programming: performance gains derive primarily from increasing levels of parallel execution. Likewise, the story of modern, scientific programming languages resembles the globalization of modern society: mixed-language environments increasingly predominate. The ForTrilinos software package sits at the intersection of these two trends and was first released in the August 2010 Trilinos open-source library and framework (release 10.4.1). ForTrilinos brings to the Fortran community the object-oriented (OO), native interfaces to C++ packages in the Trilinos library and framework.

The current article presents the first attempt to construct a three-dimensional (3D) partial differential equation (PDE) solver based on ForTrilinos. We present a case study that involves solving a 3D generalization of the classical one-dimensional (1D) PDE of Burgers [4]. As one of the few nonlinear PDEs with known analytical solutions, the Burgers equation (in both its 1D and 3D forms) often plays a role in theoretical investigations as a proxy for the more complicated 3D Navier–Stokes equations (NSE). The Burgers equation retains the unsteady, advective, and diffusive nature of the NSE without the many additional complications associated with pressure gradients. The Burgers equation also provides qualitative insights into phenomena ranging from charge density waves, vortex lines in high-temperature superconductors, and the large-scale structure of the universe. See [5] and [6] for surveys of applications for the Burgers equation and see [2] and [3] for exact analytical solutions of the 1D and 3D Burgers equation, respectively.

The goal of the current effort was to investigate the level of programming effort required to build a representative scientific software application atop ForTrilinos and provide a detailed guide for future users on ways to exploit the available capabilities. Our study employs one of the compact finite difference schemes developed by Lele [9]. In these schemes, the finite-difference approximations to the nodal derivatives result from inverting a linear system of equations with the nodal function values as the right-hand side (RHS).

[*]Corresponding author: Karla V. Morris, PhD, Combustion Research Facility, Sandia National Laboratories, 7011 East Avenue, MS 9055, Livermore, CA 94550, USA. Tel.: +1 925 294 3287; E-mail: knmorri@sandia.gov.

These schemes' compact stencil and high-order accuracy have made them popular for computing complex fluid flows in geometries amenable to finite difference approximations [12]. In the current context, the solution of the linear systems required by compact finite difference schemes provides a useful showcase for the linear solver interfaces ForTrilinos provides.

In the interest of brevity and to avoid repetition, the remainder of this article assumes familiarity with Fortran 2003. We refer readers who are familiar with Fortran 90/95 but unfamiliar with OO programming (OOP) in Fortran 2003 to the scientific software design text by Rouson et al. [16]. We refer readers who are unfamiliar with Fortran 90/95 – including the rationale for using modules and derived types – to the modern Fortran text by Metcalf et al. [10]. Section 2 of the current article presents the mathematical methodology employed in the current demonstration application and the software design methodology employed in ForTrilinos. Section 3 presents the resulting demonstration application and the recommended approach for accessing the requisite ForTrilinos capabilities. Section 4 discusses the results. Section 5 concludes and presents the path forward.

2. Methodology

2.1. Mathematical model

The 3D Burgers equation describes the nonlinear advection and diffusion of a vector quantity:

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u} \cdot \nabla \mathbf{u} + \nu \nabla^2 \mathbf{u}, \tag{1}$$

where $\mathbf{u}$ is a vector field with components u, v and w and ν is a constant parameter. The analogy to the 3D NSE lies in viewing $\mathbf{u}$ as a fluid velocity vector field with ν playing the role of the fluid's kinematic viscosity. There are some limits to this viewpoint: arriving at a NSE viscous term of the same form as the corresponding term in Eq. (1) requires applying an incompressibility constraint to the velocity field and enforcing that constraint via a pressure gradient term, but Eq. (1) contains no pressure gradient and its solution therefore does not satisfy incompressibility. Due to similar reservations, [17], for example, refer to the medium in their Burgers-equation study of turbulence in a self-gravitating medium as a "sticky dust" rather than as "viscous matter". Nonetheless, the Burgers equation's formal similarity to the incompressible NSE and the existence of exact solutions to the Burgers equation have inspired considerable activity around mining the Burgers equation for insights into fluid behavior.

From a software demonstration standpoint, the Burgers equation offers two primary benefits. First, it models some of the basic numerical properties of many of the more complicated governing equations of interest to engineering and science. Second, the equation itself (most often in the 1D form) has been used to model phenomena such as gas dynamics, traffic flow, flood waves in rivers, chemical reactions, shock and sound waves and many others.

We solve Eq. (1) in a 3D cube with periodic boundary conditions in each direction, approximating all spatial derivatives with a sixth-order-accurate finite difference scheme from [9]. The chosen family of finite difference schemes have the general form

$$\mathbf{A}\mathbf{f}' = \mathbf{B}\mathbf{f}, \tag{2}$$

$$\mathbf{C}\mathbf{f}'' = \mathbf{D}\mathbf{f}, \tag{3}$$

where $\mathbf{f}$, $\mathbf{f}'$ and $\mathbf{f}''$ are nodal solution values and first and second derivatives, respectively, and where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are all sparse coefficient matrices. The Appendix provides the stencils that generate the coefficient matrices. The tridiagonal matrix structure of the corresponding 1D problem generates a band-diagonal system in 3D.

We employ a second-order Runge–Kutta algorithm to advance the governing equation in time. The temporally discrete form of Eq. (1) is thus

$$\mathbf{u}^{n+1/2} = \mathbf{u}^n + [\mathbf{N}(\mathbf{u}^n) + \mathbf{L}(\mathbf{u}^n)]\frac{\Delta t}{2}, \tag{4}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^n + [\mathbf{N}(\mathbf{u}^{n+1/2}) + \mathbf{L}(\mathbf{u}^{n+1/2})]\Delta t, \tag{5}$$

where $\mathbf{L}$ is a discrete linear operator that approximates the diffusive term in (1) and $\mathbf{N}$ is the discrete nonlinear operator that approximates the advective term. We calculate the time step Δt based on the stability requirements of the linear term, which is more restrictive asymptotically than the stability restriction on the nonlinear term (the Courant condition) in the context of an explicit time advancement algorithm.

2.2. Software methodology

The solver presented here demonstrates several ForTrilinos capabilities and, in doing so, showcases several object-oriented features of Fortran 2003. The chosen example is intended to inform users interested in creating and operating on ForTrilinos objects within a fully objected-oriented application. The software design philosophy employed herein closely resembles that of the ForTrilinos package.

A ForTrilinos-enabled application leverages a deep software stack. This stack resolves issues related to language interoperability, portability and memory management in ways that remove such concerns from the end user. The top layer corresponds to user code of the kind presented in the current paper. The bottom layer comprises the Trilinos package of interest. Typically the Trilinos layer does all of the heavy lifting in terms of crunching and communicating numbers in parallel across a distributed-memory platform.

The middle layers comprise the ForTrilinos and CTrilinos packages. CTrilinos wraps Trilinos functionality and flattens its data structures for the sake of language interoperability. CTrilinos exports C function prototypes to which ForTrilinos links via the C-interoperability constructs that Fortran 2003 provides. This approach guarantees portability via standards-compliance. ForTrilinos in turn wraps CTrilinos in native, extensible derived types that mirror the underlying Trilinos C++ class hierarchies. The ForTrilinos layer also provides a novel reference-counting scheme that automates dynamic memory management, resolving a dilemma that results when objects in one language shadow those in a second language: How does the second language know when to destroy an object for which there no longer exists a handle in the first language? In more concrete terms, how does C++ determine when to destroy objects that the Fortran driver code no longer needs. Morris et al. [11] and Rouson et al. [15] provide more detail on the design and implementation of the ForTrilinos reference-counting architecture.

The current demonstration solver leverages the recent ForTrilinos release in Trilinos 10.8. This is the first ForTrilinos release capable of running on commodity x86 Intel and compatible processors using the Numerical Algorithms Group (NAG) Fortran compiler running under Linux and Mac OS X. Previous ForTrilinos releases required the IBM XL Fortran compiler running under IBM's proprietary AIX operating system on IBM's POWER-architecture processors.

3. Results

3.1. Available capabilities

ForTrilinos currently provides OO Fortran interfaces to the Trilinos Epetra, AztecOO and Pliris packages. The Epetra package hosts classes that support parallel and serial basic linear algebra functionality. Epetra classes provide the foundation for all Trilinos packages. AztecOO is the OO interface to the Aztec package, providing massively scalable iterative solvers for sparse linear systems. AztecOO accepts external preconditioners. The Pliris package supports a parallel, OO interface to the solution of dense linear systems, employing LU factorization on double precision data.

Additionally, ForTrilinos contains the procedural infrastructure for accessing Amesos, Galeri and Ifpack from Fortran. OO Fortran interfaces for these latter packages will be constructed upon user request. Amesos is a direct sparse solver package that provides a common interface to functionality in popular numerical libraries including LAPACK, UMFPACK (version 4.4), TAUCS (version 2.2), PARDISO (version 1.2.3 outdated), SuperLU (version 4.1), SuperLU_DIST (version 2.5), DSCPACK (version 1.0), SCALAPACK (version 1.7), and MUMPS (version 4.7.3 experimental support for version 4.9). Galeri provides an interface to generate `Epetra_Map` objects to describe the partitioning of data across distributed memory, `Epetra_CrsMatrix` objects that encapsulate sparse matrices in compressed row-wise format, and `Epetra_VbrMatrix` objects that encapsulate sparse matrices in a block row-wise format. Galeri provides a functionality very close to MATLAB's `gallery()` function for creating several well-known finite difference and finite element matrices. The Ifpack package provides OO algebraic preconditioners for iterative solvers. AztecOO objects can use these preconditioners.

In the current demonstration application, Epetra and AztecOO objects satisfy all linear algebra requirements within the solver. Sections 3.3–3.5 detail the demonstration application software design and implementation.

3.2. Accessing ForTrilinos

The ForTrilinos package is not built by default; thus it must be explicitly enabled in the Trilinos configuration system. For the details of this, we refer the reader to the sample configuration scripts and documentation contained in the Trilinos software distribution.

The OO Fortran interfaces are developed for each class within a package: each module encapsulates a derived type and its type-bound procedures (TBPs). The module name comprises the corresponding C++ class name prefixed by a "F". The derived type

```
program main
  use mpi, only : mpi_init,mpi_finalize,mpi_comm_world
  use FEpetra_MpiComm, only : Epetra_MpiComm
  use FEpetra_Map,     only : Epetra_Map
  use iso_c_binding,   only : c_int
  implicit none
  type(Epetra_MpiComm) :: comm
  type(Epetra_Map)     :: map,map_copy
  integer :: ierr
  ! MPI startup
  call mpi_init(ierr)
  ! Object construction
  comm = Epetra_MpiComm(mpi_comm_world)
  map = Epetra_Map( &
    Num_GlobalElements=512_c_int,IndexBase=1_c_int,comm=comm )

  map_copy = Epetra_Map(map)
  ! Object destruction
  call map_copy%force_finalize
  call map%force_finalize
  call comm%force_finalize
  ! MPI shutdown
  call mpi_finalize(ierr)
end program
```

Fig. 1. ForTrilinos object construction (lines 13–16) and explicit destruction (lines 19–21).

name matches the C++ class name. For example, the `FEpetra_Map` module contains the derived type `Epetra_Map`. The accepted practice for users to access a derived type and its functionality is via use association at the beginning of a programming unit (e.g., a main program, module, or module procedure). Lines 3–4 in Fig. 1 demonstrate use association of `Epetra_MpiComm` and `Epetra_Map` objects that encapsulate a Message Passing Interface (MPI) communicator and a description of the data distribution, respectively. We recommend use of the `only` clauses in lines 3–4 for access control, to document what is being imported into the programming unit in question, and to reduce compile-time overhead associated with module entities that are not used outside the module. In addition to providing access to the type, the depicted `use` statement style enables access to all of the type's public TBPs.

In keeping with the object-oriented programming (OOP) philosophy of hiding information, ForTrilinos derived types contain no public data components. A ForTrilinos object or user-defined object containing a ForTrilinos object must be explicitly constructed before use. For this purpose, each ForTrilinos derived type provides constructor functions, including a copy constructor. All constructors overload the name of the language-intrinsic structure constructor, which in turn overloads the derived type's name as is the common idiom across most OO languages. Figure 1 shows code for constructing and destroying three distinct objects in a main program. Lines 2–5 use-associate several derived types and procedures within the `main` program. Lines 7–8 declare three Epetra objects. Line 9 declares an integer argument required by MPI. Line 11 starts MPI. Lines 13 constructs an `Epetra\_MpiComm` that encapsulates a MPI communicator. Lines 14–16 construct an `Epetra\_Map` that aggregates the communicator into a description of the data distribution. Line 17 invokes a copy constructor that returns a deep copy of the passed object.

The construction process creates a ForTrilinos Fortran object that shadows an underlying Trilinos C++ object. The ForTrilinos object stores meta-data about the Trilinos object and delegates to Trilinos the construction, distribution, and subsequent manipulation of data. When a user invokes a ForTrilinos construc-

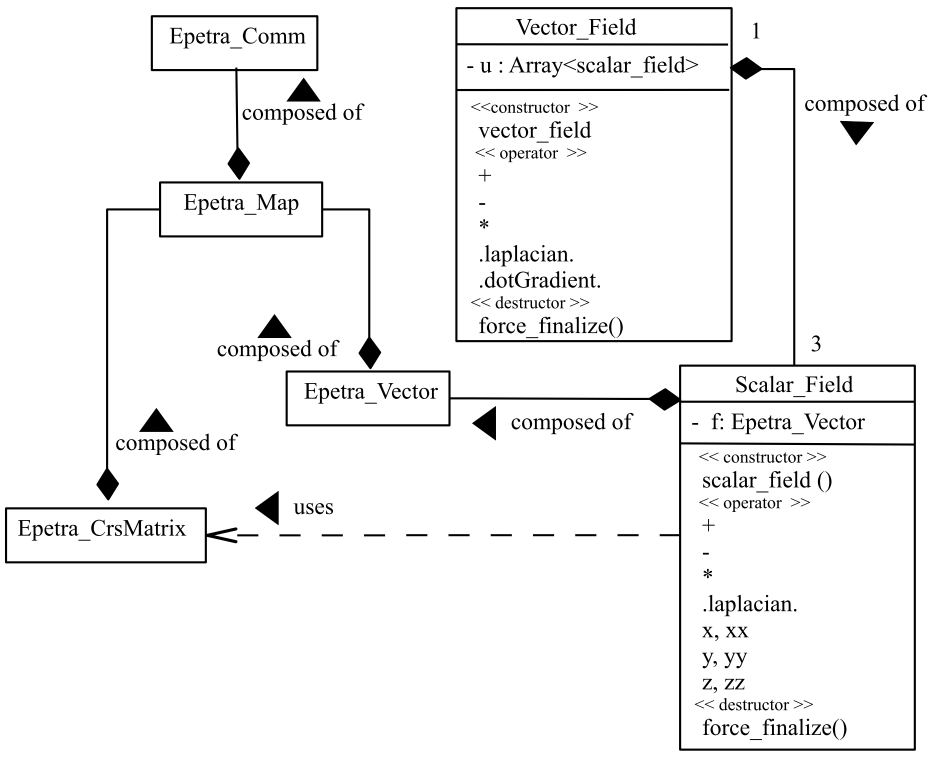


Fig. 2. UML class diagram for a demonstration 3D Burgers equation application build atop ForTrilinos. UML does not contain an Array type. To represent the Vector_Field u component, we adopt the array notation proposed by Rouson et al. [16].

tor, the constructor registers the new object for automated memory management, including the object's ultimate destruction to maintain consistency between the shadow-object meta-data and the underlying object it shadows. The ForTrilinos derived type implementation also includes TBPs that can force object destruction upon user request. Such explicit destruction is never necessary except for objects declared in the main program, in which case conscientious programming practice dictates destroying objects before the program terminates. Lines 19–21 of Fig. 1 destroy all objects declared in the main program. For objects declared in subprograms, the ForTrilinos/CTrilinos infrastructure leverages modern Fortran's type finalization construct to automate object destruction.

3.3. Class structure and behavior

The demonstration application uses the ForTrilinos classes[1] and relationships shown in the Unified Modeling Language (UML) class diagram of Fig. 2. The 3D vector field equation of Burgers naturally lends itself to representation via the depicted `vector_field` abstraction and the mathematical operators it supports. As shown, one `vector_field` object is composed of three `scalar_field` components. A `scalar_field` has an `Epetra_Vector` component. An `Epetra_Vector` has an `Epetra_Map` component. An `Epetra_Map` object has an `Epetra_Comm` component. A `scalar_field` also uses an `Epetra_CrsMatrix` object. We summarize next the roles of each member of this class hierarchy.

The `Epetra_Comm` is an abstract class that publishes abstract interfaces for a "communicator" that facilitates interactions within an abstract model of a communication subsystem. Concrete implementations of these interfaces are provided for two such subsystems: `Epetra_MpiComm` for distributed-memory parallel subsystems and `Epetra_SerialComm` (not shown) for serial subsystems. The `Epetra_Map` holds a description of the data layout on the machine; in

[1]Modern Fortran, as embodied in the 2003 and 2008 standards, allows for declaring extensible derived types with the "class" keyword. All derived types intended for use by ForTrilinos users are extensible. We therefore adopt a convention of treating "derived type" as synonymous with "class".

particular, it describes how an index space maps onto a parallel machine. The Epetra_Map has an Epetra_Comm component that details the available processes constituting the (abstract) parallel machine being employed. An Epetra_Vector stores distributed, 1D arrays, basing their layout on its Epetra_Map component. A Epetra_CrsMatrix stores a compressed, sparse matrix, the rows of which are laid out in memory following the distribution specified by the Epetra_Map that also describes the Epetra_Vector layout.

For purposes of the demonstration application, the scalar_field class abstracts 3D scalar functions sampled on a uniform, Cartesian grid.[2] Figure 2 shows the scalar_field object's private Epetra_Vector component, f, and its public TBPs. The f component holds 3D nodal values unrolled onto a 1D index space and distributed across the computing platform. The scalar_field module provides a like-named constructor and a collection of generic operators (Laplacian, $+$, $*$, $-$, etc.) implemented as TBPs. These TBPs provide arithmetic and differential calculus functionality. The same module provides a force_finalize destruction procedure that should only be necessary for objects declared in the main program. The destruction of scalar_field objects local to subprograms happens at the direction of the compiler, and the destruction of the corresponding Epetra_Vector components happens at the direction of the aforementioned ForTrilinos reference-counting architecture.

Each element of the vector_field component array, u, models a component of the Burgers equation solution vector **u**. The vector_field class defines generic operators implemented as TBPs providing discrete approximations to the differential and arithmetic operators in Eqs (4)–(5). The operator implementations invoke generic operators supported by the scalar_field class.

The use of an abstract data type (ADT) calculus of the kind defined by the vector_field and scalar_field generic operators has been described elsewhere [13,16]. Many other possibilities exist for decomposing the 3D Burgers equation into a set of classes. The composition of classes presented above allows for demonstrating the use of ForTrilinos objects in the challenging setting of aggregating them into a multilevel class hierarchy and then constructing, manipulating, and destructing the aggregate objects repeatedly across deeply nested call trees. The lessons learned in doing so should prove instructive in other use cases with these properties.

[2]Although our grid choice facilitates a simple demonstration, most Trilinos use cases involve unstructured grids.

3.4. Implementation

The full implementation of the 3D Burgers solver will be made available in the examples directory of an upcoming ForTrilinos release. This section presents several excerpts from a current, working implementation. The emphasis here is twofold: demonstrating the construction of objects that aggregate ForTrilinos objects and demonstrating the TBPs invoked on those objects to accomplish tasks that prove useful in the demonstration application. Additionally, we include a few suggestions on style and discuss the support ForTrilinos provides for encouraging safety.

3.4.1. Scalar field construction

Figure 3 excerpts a portion of the scalar_field module. Lines 2–8 import ForTrilinos types, their public TBPs, and other module procedures. Line 13 hides all module entities by default. Line 14 exposes scalar_field, its public TBPs, and constructors. The scalar_field type specification in lines 15–27 aggregates an Epetra_Vector and a logical flag signaling a successful construction. The type specification excerpts some of the TBPs from the actual implementation: isConstructed returns the construction flag; df_dx approximates a derivative; and total and laplacian implement the generic addition operator (+) and the Laplacian differential operator (.laplacian.), respectively.

The constructor at line 49 takes as arguments a procedure pointer and a communicator and uses the procedure associated with the pointer to initialize the constructed object's Epetra_Vector component. The class keyword at line 52 declares the passed communicator to be polymorphic. This declaration allows the passed argument to be of type Epetra_Comm or any type that extends Epetra_Comm.

At line 64–65, the constructor uses the passed communicator to create an Epetra_Map: a simple linear map in the current example. At line 74, the constructor uses the resulting map to construct the Epetra_Vector component, constructing the component first with default zero-initialization (line 74) and then invoking the ReplaceGlobalValues TBP at lines 75–76 to replace the component's entries with the values generated by the passed initialization procedure.

```
module scalar_field_module
  use ForTrilinos_assertion_utility ,only : error_message,assert
  use FEpetra_Comm     ,only: Epetra_Comm
  use FEpetra_Map      ,only: Epetra_Map
  use FEpetra_Vector   ,only: Epetra_Vector
  use FEpetra_CrsMatrix,only: Epetra_CrsMatrix
  use FAztecOO         ,only: AztecOO
  use ForTrilinos_error,only: error
  use field_module  ,only : initial_field  ! Procedure pointer
  use iso_c_binding ,only : c_double,c_int ! Interoperable kinds
  use globals       ,only : nspace,nx,ny,nz,dx
  implicit none
  private ! Hide everything by default
  public :: scalar_field ! Expose type, constructor, TBPs
  type   :: scalar_field
    private
    type(Epetra_Vector) :: f
    logical :: constructed=.false.
  contains
    procedure :: isConstructed !Check for successful construction
    procedure :: total         !Object summation operator
    procedure :: laplacian     !Object differential operator
    generic   :: operator(+) => total
    generic   :: operator(.laplacian.) => laplacian
    procedure :: x => df_dx    ! 1st derivative w.r.t. x
    ! (additional type-bound procedures not shown)
  end type
  ! Module variables
  type(Epetra_Map), allocatable :: map
  real(c_double), dimension(nx) :: x_node
  integer(c_int), save :: NumMy_xy_planes, &
                          my_first_xy_plane,my_last_xy_plane
  ! Module constants
  integer(c_int) ,parameter :: IndexBase=1
  integer(c_int) ,parameter :: NumGlobalElements = nx*ny*nz
  integer(c_int) ,parameter :: NumGlobal_xy_planes = nz, &
                               Num_xy_points_per_plane = nx*ny
  interface scalar_field
    procedure new_scalar_field
  end interface

contains

  logical function isConstructed(this)
    class(scalar_field) ,intent(in) :: this
    isConstructed = this%constructed
  end function
```

Fig. 3. Scalar_field module excerpt: use-association of module dependencies, type specification, and constructor implementation.

```
  function new_scalar_field(initial,comm) result(this)
    type(scalar_field)                          :: this
    procedure(initial_field) ,pointer           :: initial
    class(Epetra_Comm)       ,intent(in)        :: comm
    real(c_double) ,dimension(:) ,allocatable :: f_v
    integer(c_int) :: i,j,k,NumMyElements
    ! Requires (preconditions):
    call assert(mod(NumGlobal_xy_planes,comm%NumProc())==0 &
      ,error_message('new_scalar_field: number of processes not &
                     divisible by number of xy planes.'))
    ! Define local variables:
    NumMy_xy_planes = NumGlobal_xy_planes/comm%NumProc()
    NumMyElements = NumMy_xy_planes*Num_xy_points_per_plane
    ! Define module variables:
    forall(i=1:nx) x_node(i) =(i-1)*dx
    if (.not. allocated(map)) map = &
      Epetra_Map(NumGlobalElements,NumMyElements,IndexBase,comm)
    ! Define derived type components:
    ! Initialize and spread 3D field values along linear 1D array
    allocate(f_v(NumMyElements))
    my_first_xy_plane = comm%MyPID()*NumMy_xy_planes + 1
    my_last_xy_plane = (comm%MyPID()+1)*NumMy_xy_planes
    forall(i=1:nx,j=1:ny,k=my_first_xy_plane:my_last_xy_plane) &
      f_v( i + (j-1)*ny + (k-my_first_xy_plane)*ny*nz) = &
         initial(x_node(i),x_node(j),x_node(k))
    this%f=Epetra_Vector(map,zero_initial=.true.)
    call this%f%ReplaceGlobalValues(NumMyElements,f_v, &
                                    map%MyGlobalElements())
    this%constructed=.true.
    ! Ensures (postcondition):
    call assert(this%isConstructed() &
      ,error_message('new_scalar_field: construction failed.'))
  end function

  ! (additional type-bound procedure implementations not shown)

end module
```

Fig. 3. (Continued.)

3.4.2. *ForTrilinos assertion utility*

Lines 56–58 and 79–80 demonstrate software design philosophy often termed "programming by contract". This approach requires a subprogram to satisfy a contract with the calling code. The contract specifies preconditions required to be true before a subprogram executes and postconditions the subprogram ensures will be true after it executes.

The `ForTrilinos_assertion_utility` module provides several routines that help enforce contracts. Modeled after the C/C++ `assert` intrinsic, the calls to `assert` subroutine at lines 56 and 79 halt execution and print the passed error message to standard error whenever a `logical` expression passed as the first argument evaluates to false. Another ForTrilinos assertion routine accepts an array of `logical` expressions and a corresponding array of error messages. Another assertion routine halts execution when any element of an integer array is not identical to all other elements.

```
function total(lhs,rhs)
  class(scalar_field) ,intent(in) :: lhs,rhs
  type(scalar_field) :: total
  real(c_double), parameter :: one=1._c_double,zero=0._c_double

  ! Requires (preconditions):
  call assert(lhs%isConstructed(),error_message( &
  'scalar_field%total(): unconstructed left-hand side.'))
  call assert(rhs%isConstructed(),error_message( &
  'scalar_field%total(): unconstructed right-hand side.'))

  total%f=Epetra_Vector(map,zero_initial=.true.)
  call total%f%Update(one,lhs%f,one,rhs%f,zero )
  total%constructed = .true.

  ! Ensures (postconditions):
  call assert(total%isConstructed(),error_message( &
 'scalar_field%total(): unconstructed function result.'))
end function
```

Fig. 4. Scalar_field addition operator implementation.

In C/C++, one can turn all assertion-checking off at compile-time to eliminate associated runtime penalties in production code. Akin [1] described an approach that can likewise eliminate runtime penalties with most Fortran compilers: define a global constant and use it to conditionally toggle the execution of certain code on or off (debugging code in Akin's case). With this approach, the beginning of line 56 might instead read

```
if (assertions) call
  assert(mod(NumGlobal_xy_planes,
             comm% ...
```

and elsewhere might be a declaration of the form

```
logical, parameter :: assertions =
  .false.
```

in which case most compilers would eliminate the call to `assert` during an optimization step known as "dead-code-removal". Recompiling with `assertions` set to `.true.` would turn assertion-checking on. For a simple alternative solution, one could wrap `assert` with preprocessor macros, in which case, the code associated with the utility could be removed automatically during a preprocessing stage when not needed.

3.4.3. *Scalar field arithmetic*

The requisite `scalar_field` arithmetic operators invoke TBPs on the private `Epetra_Vector` components. As an example, the `total` function in Fig. 4 sums two scalar fields. This function's preconditions verify that the `lhs` and `rhs` operands were successfully constructed. The postcondition verifies the construction status of the return argument.

Polymorphic declarations of each operand as a `class` at line 2 allows each actual argument to be a `scalar_field` or any type that extends `scalar_field`. The `total` function result is likewise a `scalar_field` computed by invoking the `Update` TBP of its `Epetra_Vector` component `f`. `Update` evaluates expressions of the form `a*lhs%f+b*rhs%f+c*total%f` and assigns the result to `total%f`. In the case shown at line 13 in Fig. 4, `a` and `b` are unity, `c` is zero, and each constant is declared to be of kind `c_double` as required to match the C-interoperable hardware representation ForTrilinos requires for all TBP arguments. The entire operation executes in parallel in distributed memory consistent with the data distribution specified by the `Epetra\_Map`.

3.4.4. *Scalar field differential calculus*

All `scalar_field` differential operators involve the same class interactions. We focus here on the first partial derivative with respect to x shown in Fig. 5. Following the form of the differentiation described by Eqs (2) and (3), the current implementation computes derivatives in three steps:

```
function df_dx(this)
  type(scalar_field) :: df_dx
  class(scalar_field) ,intent(in) :: this
  type(Epetra_CrsMatrix), allocatable :: A
  type(Epetra_Vector) :: b
  type(AztecOO) :: Solver
  type(error) :: err

  ! Requires (preconditions):
  call assert(this%isConstructed(),error_message(&
       'scalar_field%x(): unconstructed argument.'))

  associate( x => df_dx%f )
    ! Create matrix A and vectors x, b
    if (.not.allocated(A)) A=Matrix_diff_x(coef_1st)
    x=Epetra_Vector(A%RowMap())
    call x%Random()  ! initial guess
    b=RHS_diff_x(this,coef_1st,1)  ! coef_1st module variable:
    ! Solve Ax=b for x             ! definition not shown.
    Solver=AztecOO(A,x,b)          ! See Appendix A.1.1.
    call Solver%iterate(A,x,b,MaximumIter,tolerance,err)
    call assert( [err%error_code()==0_c_int] , &
       [error_message('Solver%iterate: failed')] )
  end associate
  df_dx%constructed = .true.

  ! Ensures (postconditions):
  call assert(df_dx%isConstructed(),error_message( &
        'scalar_field%x(): differentiation failed.'))
end function
```

Fig. 5. Scalar field first-derivative operator implementation.

(1) *Constructing the LHS matrix* **A**: The x-derivative TBP delegates this step to the private module procedure `Matrix_diff_x` at line 15 in Fig. 5. The latter procedure returns an `Epetra_CrsMatrix` holding a compressed representation of the sparse coefficient matrix described in the Appendix. The `Epetra_CrsMatrix` constructor at line 26 in the `Matrix_diff_x` function in Fig. 6 takes three arguments: (1) an enumerated value `FT_Epetra_DataAccess_E_t` set to designate a copy (the alternative being a view); (2) an `Epetra_Map` encapsulating the description of the row distribution in memory of the `Epetra_CrsMatrix` and corresponding to the distribution of the `Epetra_Vector` component; and (3) an array expressing the number of nonzero elements of each local row in the matrix. Finally, lines 63 and 69 insert the appropriate coefficients into the constructed object via the `InsertGlobalValues` TBP.

(2) *Constructing the RHS matrix–vector product* **Bf**: The x-derivative TBP delegates this step to the private module procedure `RHS_diff_x` at line 18 in Fig. 5. The latter procedure extracts a copy of its argument's nodal values local to the given process at line 12 in Fig. 7, constructs an `Epetra_Vector` to hold the matrix–vector product at line 14, performs the requisite multiplications locally, and then replaces the corresponding global values in the function result at line 41. This is essentially a matrix-free calculation of the matrix–vector product. (An alternative approach implemented elsewhere for the z derivative constructs the matrix `B` and then in-

```
function Matrix_diff_x(coef)
 use ForTrilinos_enum_wrappers,only:FT_Epetra_DataAccess_E_Copy
  real(c_double),dimension(:), intent(in) :: coef
  type(Epetra_CrsMatrix) :: Matrix_diff_x
  type(error)            :: err
  integer(c_int),dimension(:),allocatable :: MyGlobalElements
  integer(c_int),dimension(:),allocatable :: NumNz
  integer(c_int) :: MyGlobalElements_diagonal(1)
  integer(c_int) :: NumMyElements,i
  integer(c_int) :: indices(4), NumEntries
  real(c_double) :: values(4), one=1.0
  integer(c_int),parameter :: diagonal=1

 ! Get updated list and number of local equations from Map
  NumMyElements = map%NumMyElements()
  MyGlobalElements = map%MyGlobalElements()

 ! Create an integer vector NumNz that is used to build
 ! the Epetra Matrix NumNz(i) is the number of non-zero
 ! elements for the ith global equation on this processor
  allocate(NumNz(NumMyElements))
  NumNz = 5

 ! Create an Epetra_Matrix
 associate( A=>Matrix_diff_x)
   A = Epetra_CrsMatrix(FT_Epetra_DataAccess_E_Copy,map,NumNz)

   ! Add rows one at a time
   ! Need some vectors to help
   values(1:4)=coef(1:4)
   do i=1,NumMyElements
     if (mod(MyGlobalElements(i)-1,nx)==0) then
      indices(1) = MyGlobalElements(i)+2
      indices(2) = MyGlobalElements(i)+1
      indices(3) = MyGlobalElements(i)+nx-1
      indices(4) = MyGlobalElements(i)+nx-2
      NumEntries = 4
    else if(mod(MyGlobalElements(i),nx)==0) then
      indices(1) = MyGlobalElements(i)-nx+2
      indices(2) = MyGlobalElements(i)-nx+1
      indices(3) = MyGlobalElements(i)-1
      indices(4) = MyGlobalElements(i)-2
      NumEntries = 4
    else if (mod(MyGlobalElements(i)-2,nx)==0) then
      indices(1) = MyGlobalElements(i)+2
      indices(2) = MyGlobalElements(i)+1
      indices(3) = MyGlobalElements(i)-1
```

Fig. 6. First-derivative LHS matrix construction.

```
          indices(4) = MyGlobalElements(i)+nx-2
          NumEntries = 4
        else if(mod(MyGlobalElements(i)+1,nx)==0) then
          indices(1) = MyGlobalElements(i)-nx+2
          indices(2) = MyGlobalElements(i)+1
          indices(3) = MyGlobalElements(i)-1
          indices(4) = MyGlobalElements(i)-2
          NumEntries = 4
        else
          indices(1) = MyGlobalElements(i)+2
          indices(2) = MyGlobalElements(i)+1
          indices(3) = MyGlobalElements(i)-1
          indices(4) = MyGlobalElements(i)-2
          NumEntries = 4
        end if
        call A%InsertGlobalValues(MyGlobalElements(i),  &
                   NumEntries,values,indices,err)
        call assert( [err%error_code()==0_c_int] , &
           [error_message('A%InsertGlobalValues: failed')] )
        !Put in the diagonal entry
        MyGlobalElements_diagonal=MyGlobalElements(i)
        call A%InsertGlobalValues(MyGlobalElements(i),  &
                diagonal,[one],MyGlobalElements_diagonal,err)
        call assert( [err%error_code()==0_c_int] , &
            [error_message('A%InsertGlobalValues: failed')] )
      end do

      !Finish up
      call A%FillComplete(.true.,err)
    end associate
  end function
```

Fig. 6. (Continued.)

vokes that matrix's Multiply_Vector TBP to form the product. The latter approach requires interprocess communication but delegates the orchestration of all such communication to Trilinos.)

(3) *Solving the linear system* $\mathbf{Af}' = \mathbf{Bf}$: At line 20 in Fig. 5, the x-derivative TBP constructs an AztecOO solver object from the aforementioned sparse LHS matrix A, the RHS vector b (holding Bf), and the solution vector f'. Line 21 invokes the solver's iterate TBP to compute the solution of the linear system.

3.4.5. *Vector field construction, arithmetic and differential calculus*

As described in Fig. 2, a vector_field aggregates three scalar_field components. The full 3D Burgers equation solver constructs vector_field objects by constructing each corresponding scalar_field component. Each vector_field arithmetic operator invokes the corresponding arithmetic operator of each scalar_field component. The vector_field differential operators .laplacian. and .dotGradiant. delegate all associated derivative calculations to the scalar_field differential calculus TBPs.

Because the vector_field class has no direct interaction with ForTrilinos, we omit most details of its class implementation. We provide one arithmetic operator implementation corresponding to the addition operator in Fig. 8 to illustrate the complicated class relationships currently possible in working with ForTrili-

```
type(Epetra_Vector) function RHS_diff_x(this,coef,order)
  class(scalar_field), intent(in) :: this
  real(c_double), intent(in), dimension(:) :: coef
  integer(c_int), intent(in) :: order
  integer(c_int) :: i,NumMyElements
  integer(c_int),dimension(:),allocatable :: MyGlobalElements
  real(c_double),dimension(:),allocatable :: Bf_v,f_v

  MyGlobalElements=map%MyGlobalElements()
  NumMyElements=map%NumMyElements()

  f_v=this%f%ExtractCopy()
  associate( Bf=>RHS_diff_x )
    Bf=Epetra_Vector(map)
    allocate(Bf_v(Bf%MyLength()))
    select case (order)
    case(1)  ! calculate RHS for 1st derivative
      do i=1,NumMyElements
        if (mod(MyGlobalElements(i)-1,nx)==0) then
          Bf_v(i) = (coef(6)/(4.0*dx))*(f_v(i+2)-f_v(i+nx-2))+&
                     (coef(7)/(2.0*dx))*(f_v(i+1)-f_v(i+nx-1))
        else if(mod(MyGlobalElements(i),nx)==0) then
          Bf_v(i) = (coef(6)/(4.0*dx))*(f_v(i-nx+2)-f_v(i-2))+&
                     (coef(7)/(2.0*dx))*(f_v(i-nx+1)-f_v(i-1))
        else if (mod(MyGlobalElements(i)-2,nx)==0) then
          Bf_v(i) = (coef(6)/(4.0*dx))*(f_v(i+2)-f_v(i+nx-2))+&
                     (coef(7)/(2.0*dx))*(f_v(i+1)-f_v(i-1))
        else if(mod(MyGlobalElements(i)+1,nx)==0) then
          Bf_v(i) = (coef(6)/(4.0*dx))*(f_v(i-nx+2)-f_v(i-2))+&
                     (coef(7)/(2.0*dx))*(f_v(i+1)-f_v(i-1))
        else
          Bf_v(i) = (coef(6)/(4.0*dx))*(f_v(i+2)-f_v(i-2))+ &
                   (coef(7)/(2.0*dx))*(f_v(i+1)-f_v(i-1))
        end if
      enddo
    case(2)  ! RHS for 2nd derivative
      ! (Implementation not shown)
    case default
      stop 'RHS_diff_x: invalid derivative order.'
    endif
    call Bf%ReplaceGlobalValues(NumMyElements,Bf_v &,MyGlobalElements)
   end associate
end function
```

Fig. 7. First-derivative RHS matrix–vector product calculation.

nos: the code in that figure enables user-defined operators to operate on a composite object with components that themselves are composite objects that aggregate `Epetra_Vector` components.

3.5. Application driver

Figure 9 shows the demonstration application main driver. Much of the explanation of the figure follows

```
type(vector_field) function total(lhs,rhs)
  class(vector_field) ,intent(in)  :: lhs, rhs

  ! Requires (preconditions):
  call assert(lhs%isConstructed(),error_message(&
   'vector_field%total(): unconstructed left-hand side.'))
  call assert(rhs%isConstructed(),error_message(&
   'vector_field%total(): unconstructed right-hand side.'))

  total = vector_field()
  total%u(1)=lhs%u(1)+rhs%u(1)
  total%u(2)=lhs%u(2)+rhs%u(2)
  total%u(3)=lhs%u(3)+rhs%u(3)
  total%constructed = .true.

  ! Ensures (postconditions):
  call assert(total%isConstructed(),error_message(&
    'vector_field%total(): unconstructed right-hand side.'))
end function
```

Fig. 8. Implementation of addition operator in the vector_field module.

```
program main
#include "ForTrilinos_config.h"
#ifdef HAVE_MPI
  use mpi
  use FEpetra_MpiComm, only: Epetra_MpiComm
#else
  use FEpetra_SerialComm, only: Epetra_SerialComm
#endif
  use iso_c_binding, only: c_int,c_double
  use vector_field_module, only: vector_field
  use field_module, only: initial_field
  use initializer, only: u_initial,v_initial,w_initial,zero
  implicit none
#ifdef HAVE_MPI
  type(Epetra_MpiComm)    :: comm
#else
  type(Epetra_SerialComm) :: comm
#endif
  type(vector_field)                :: u,N_u,u_half
  procedure(initial_field) ,pointer :: initial_u,initial_v, &
                                       initial_w,initial
  real(c_double) :: dt,half=0.5,t=0.,t_final=0.1,nu=1.
  integer(c_int) :: ierr
```

Fig. 9. Main program for the 3D Burgers solver application.

```
 ! Initializing MPI
#ifdef HAVE_MPI
  call MPI_INIT(ierr)
  comm = Epetra_MpiComm(MPI_COMM_WORLD)
#else
  comm = Epetra_SerialComm()
#endif
 ! Vector field construction
  initial_u => u_initial
  initial_v => v_initial
  initial_w => w_initial
  u = vector_field(initial_u,initial_v,initial_w,comm)
  initial => zero
  N_u = vector_field(initial,initial,initial,comm)
  u_half = vector_field(initial,initial,initial,comm)
  ! Time advance vector field
  do while (t<=t_final)  !2nd-order Runge-Kutta:
    dt = u%runge_kutta_stable_step(2,nu)
    N_u = u.dotGradient.u
    ! first substep
    u_half = u + ( (((.laplacian.u)*nu) - N_u)*(half*dt) )
    N_u = u_half.dotGradient.u_half
    ! second substep
    u  = u + ( (((.laplacian.u_half)*nu) - N_u)*dt )
    t = t + dt
  end do
  call u%output()
 ! Final memory cleanup before finalization
  call N_u%force_finalize
  call u_half%force_finalize
  call u%force_finalize
  call comm%force_finalize
 ! Finalizing MPI
#ifdef HAVE_MPI
  call MPI_FINALIZE(ierr)
#endif
end program
```

Fig. 9. (Continued.)

code discussions earlier in the current article. One feature not used earlier, however, is the ability to toggle between serial and parallel builds by setting configuration flags. The C pre-processor directives beginning at lines 3, 14, 25 and 57 provide this flexibility. Line 2 sets up the build system to pass configuration flags such as the `HAVE_MPI` variable that determines the type of build. This line is needed only when the source code needs access to configuration flags.

Not shown is the abstract interface `initial_field` employed at line 20. It provides an interface of a function that returns a real value given three real values. The time advancement of the 3D Burgers equation occurs in lines 40–49. Also not shown is the `output` TBP invoked on the `vector_field` state. Other details of `main` mirror examples of code earlier in this article.

4. Discussion

The main driver presented in Fig. 9 demonstrates several principles related to our goals for ForTrilinos.

The most obvious is the goal of maintaining the OOP philosophy of the larger Trilinos project while working within idioms that feel natural to Fortran programmers. An example is the use of user-defined structure constructors that overload the name of the class of objects they construct. Fortran 90/95 had intrinsic structure constructors provided by the language and these likewise overload the class name so taking advantage of the Fortran 2003 user-defined structure constructor capability seems natural. Furthermore, naming constructors after the class matches the practice in most other OOP languages. We believe common idioms have expressive power.

On the other hand, there are ways in which we have made subtle departures from common Fortran practice. An important one arises in the need for the copy constructor at line 17 in Fig. 1. If instead a user were to execute the language's intrinsic assignment in the form `map_copy = map`, then `map_copy` would hold a reference (via pointer association) to `map` rather than holding a separate copy. This behavior stems from the underlying reference-counting scheme cited in Section 2.2. The fact that this behavior would seem unnatural to most Fortran programmers led to our advice in Section 3.2 that all objects must be constructed by a ForTrilinos constructor before use – one specific implication being that the copy constructor is generally preferable to the intrinsic assignment in terms of engendering behavior that would likely seem more intuitive to most Fortran programmers.

Because most Fortran programmers have only recently gained access to compilers that support the OOP constructs of Fortran 2003, the current authors have had to develop some new idioms along the way. We have summarized many of them in publications on modern Fortran program design and construction for numerical applications [7,13,14,16].

The demonstration application driver illustrates other principle aims of ForTrilinos. One is the desire to enable scientific programmers to write code using what appear to be serial semantics but to support those semantics via parallel method invocations on distributed-memory data structures. Two by-products of this approach appear in the minimal number of MPI calls required to put together an application and the minimal number of changes that must be made to switch between serial and parallel versions of an application. Only two MPI calls appear in the approximately 1500 lines of source code comprising the high-order, finite-difference, 3D Burgers equation demonstration solver: `MPI_init` for startup and `MPI_finalize` for shutdown. Likewise, the application has only 18 lines associated with C pre-processor conditionals required to switch between serial and parallel builds.

Closely related to the encapsulation of most MPI calls is the encapsulation and hiding of the distributed data structures on which those calls operate. Characteristic of OOP, this encapsulation and information-hiding positively impacts application program construction and debugging. For example, should a ForTrilinos user need to print an object's data during a debugging exercise, doing so does not require that the end user understand the underlying data structure. In particular, the user need not know how the data is distributed. Trilinos offers various input/output (I/O) methods a user can invoke to export the data in various formats. Section 5.2 describes plans for providing access to these methods in ForTrilinos.

5. Conclusions and future work

We have demonstrated the level of effort required to build atop ForTrilinos in one common use case: high-order numerical solution of the 3D vector partial differential equation of Burgers. The chosen class structure illustrates the complex class hierarchies one can build atop ForTrilinos with modern Fortran compilers: a `vector_field` class aggregates three instances of a `scalar_field` class, which in turn aggregates an `Epetra_Vector` and uses an `Epetra_CrsMatrix`. The `Epetra_Vector` aggregates an `Epetra_Map`, which in turn aggregates an `Epetra_Comm`.

The vector field abstraction publishes several arithmetic and differential operators. The main driver program manipulates expressions composing these operators. These expressions and most remaining main program code use syntax that is identical whether the code runs in serial or parallel mode. The highest-level expressions ultimately resolve to lower-level ForTrilinos type-bound procedures invoked on distributed-memory objects manipulated in parallel.

Just over a year from its initial release, ForTrilinos remains relatively early in its development. An important ongoing effort involves expanding the ForTrilinos coverage of Trilinos C++ packages. The following subsections describe other near-term priorities. We offer these to solicit user input on next steps.

5.1. Matrix-free calculations and performance tuning

Although we used the current demonstration application to highlight matrix construction, a more sophisticated use of ForTrilinos might employ the Trilinos `Epetra_Operator` class to build a matrix-free solver. In the current application, the predetermined regularity of the sparseness patterns in the LHS **A** matrix could facilitate direct operation on the iterative solution vectors employed in and generated by AztecOO. In many cases where we currently use a concrete matrix class such as `Epetra_CrsMatrix`, the relevant Trilinos methods manipulate the underlying object via its abstract `Epetra_Object` parent class. A user could therefore extend the `Epetra_Operator` abstract class by providing a concrete class that implements the `Epetra_Operator` deferred bindings.

Matrix-free computation could offer significant performance benefits associated with reduced storage requirements. Although we have verified the correctness of the demonstration application running in parallel on eight nodes in a dual-socket, four-core CPU configuration totaling 64 cores, detailed profiling and performance-tuning of the demonstration application and the underlying ForTrilinos interfaces will be the subject of future research. We plan to explore the matrix-free approach as part of our performance-tuning effort.

5.2. Interface simplification and file input/output (I/O)

While the primary role of ForTrilinos is to wrap Trilinos C++ packages, we also aim to add value wherever possible rather than to merely translate the C++ idioms into Fortran. One simple way of doing so is to exploit Fortran's rich array facilities to simplify argument passing. Whereas several of the C++ methods receive an array as well as the array bounds as separate arguments, Fortran arrays carry along such information. An upcoming version of ForTrilinos will publish interfaces that query array arguments for their bounds in lieu of asking Fortran programmers to pass redundant and therefore potentially error-prone bounds information.

A more significant way to add value will involve native Fortran I/O. File I/O, if not implemented and supported properly, can produce large load imbalances when one processor is doing the I/O and the others are idling. This can be mitigated to some extent by using ROMIO extensions to MPI, and using the collective MPI read and write routines. However, implementing this requires invasive knowledge of the internal structure of the derived types and a good grasp of the MPI collective I/O routines. In keeping with the Trilinos design philosophy, we plan to remove these two hurdles with two strategies: wrapping the EpetraExt I/O subroutines and writing native ForTrilinos I/O subroutines.

The EpetraExt package provides access to ASCII files written in the MatrixMarket format[3] and binary files written in the HDF5 format. Subroutines are provided to convert an `Epetra_Map` to and from these formats. ForTrilinos will provide wrappers to these subroutines for transparent access for the users.

Finally, ForTrilinos programmers often print data in ASCII format for debugging or testing purposes. In a likely ForTrilinos use case, this information will be spread across processors and therefore require specialized knowledge to correctly access. To make this access straightforward, ForTrilinos will provide a few native subroutines for writing the required data to formatted ASCII files.

Acknowledgements

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. The authors thank the Office of Naval Research Sense and Response Logistics program for their support. The authors also thank Hari Radharkrishnan for guidance on the parallel le I/O plan for ForTrilinos. The second author thanks Prof. Stavros Kassinos for hosting a visit to the University of Cyprus, where most of the second author's contributions to this paper took place.

Appendix: Spatial derivative approximation

A.1. First derivative

Lele [9] defined the following first-derivative stencil:

$$\begin{aligned}\beta f'_{i-2} + \alpha f'_{i-1} + f'_i + \alpha f'_{i+1} + \beta f'_{i+2} \\ = c\frac{f_{i-3} - f_{i-3}}{6h} + b\frac{f_{i-2} - f_{i-2}}{4h} \\ + a\frac{f_{i-1} - f_{i-1}}{2h}.\end{aligned} \tag{6}$$

[3]http://math.nist.gov/MatrixMarket/reports/MMformat.ps.

This leads to a pentadiagonal LHS coefficient matrix in 1D and a block-diagonal matrix in higher dimensions. For the sixth-order accuracy used in our demonstration solver, the parameters in Eq. (6) are $\beta = 0$, $\alpha = \frac{1}{3}$, $a = \frac{14}{9}$, $b = \frac{1}{9}$ and $c = 0$.

A.2. Second derivative

The corresponding second-derivative approximation is

$$\begin{aligned}\beta f''_{i-2} + \alpha f''_{i-1} + f''_i + \alpha f''_{i+1} + \beta f''_{i+2} \\ = c\frac{f_{i-3} - 2f_i + f_{i-3}}{9h^2} + b\frac{f_{i-2} - 2f_i + f_{i-2}}{4h^2} \\ + a\frac{f_{i-1} - 2f_i + f_{i-1}}{h^2}. \qquad (7)\end{aligned}$$

For the sixth-order accuracy used in our demonstration application, the parameters in Eq. (7) are $\beta = 0$, $\alpha = \frac{2}{11}$, $a = \frac{12}{11}$, $b = \frac{2}{11}$ and $c = 0$.

References

[1] E. Akin, *Object-Oriented Programming via Fortran 90/95*, Cambridge Univ. Press, Cambridge, 2003.

[2] E.R. Benton and G.W. Platzman, A table of solutions of the one-dimensional burgers equation, *Quart. Appl. Math.* **30** (1972), 195–212.

[3] J.G. Blom and J.G. Verwer, Vlugr3: a vectorizable adaptive grid solver for PDEs in 3D, part I: algorithmic aspects and new applications, *Appl. Numer. Math.* **16** (1994), 129–156.

[4] J.M. Burgers, A mathematical model illustrating the theory of turbulence, *Adv. Appl. Mech.* **1** (1948), 171–199.

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

[6] J. Davoudi, A.A. Masoudi, M.R.R. Tabar, A.R. Rastegar and F. Shahbazi, Three-dimensional forced burgers turbulence supplemented with a continuity equation, *Phys. Rev. E* **63** (2001), 056308.

[7] S. Filippone and A. Buttari, Object-oriented techniques for sparse matrix computations in Fortran 2003, *ACM Trans. Math. Softw.* **38**(4) (2012), to appear.

[8] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams and K.S. Stanley, An overview of the Trilinos project, *ACM Trans. Math. Softw.* **31**(3) (2005), 397–423.

[9] S.K. Lele, Compact finite difference schemes with spectral-like resolution, *J. Comp. Phys.* **103** (1992), 16–42.

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

[11] K. Morris, D. Rouson and J. Xia, On the object-oriented design of reference-counted shadow objects, in: *Proc. 4th International Workshop on Software Engineering for Computational Science and Engineering*, ACM Press, New York, NY, 2011, pp. 19–27.

[12] C. Rosales and C. Meneveau, Linear forcing in numerical simulations of isotropic turbulence: physical space implementations and convergence properties, *Phys. Fluids* **17**(9) (2005), 095106.

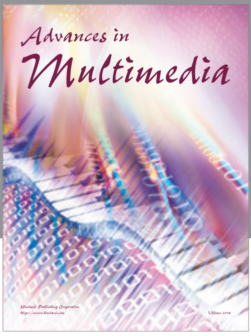
[13] D.W.I. Rouson, Towards analysis-driven scientific software architecture: the case for abstract data type calculus, *Sci. Program.* **16**(4) (2008), 329–339.

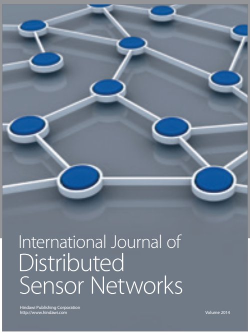
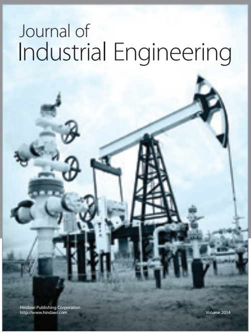
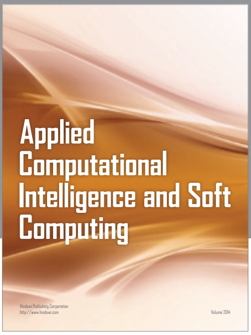
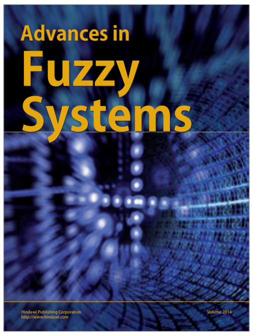
[14] D.W.I. Rouson, H. Adalsteinsson and J. Xia, Design patterns for multiphysics modeling in Fortran 2003 and C++, *ACM Trans. Math Soft.* **37**(1) (2010), 1–30.

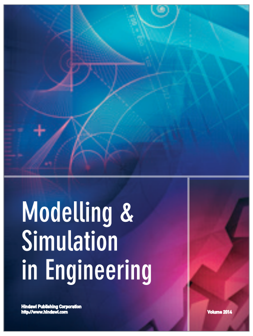
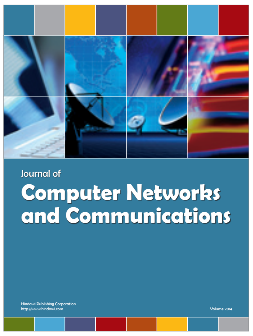
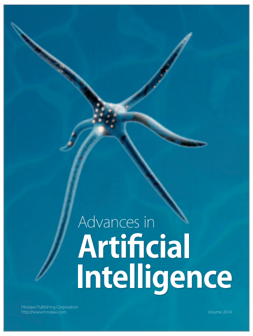
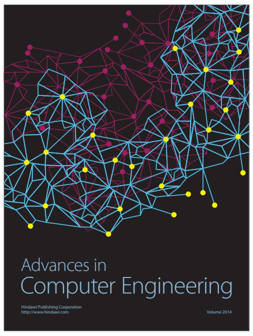
[15] D.W.I. Rouson, K. Morris and J. Xia, Managing C++ objects with modern Fortran in the driver's seat: this is not your parents' Fortran, *Comput. Sci. Eng.* **13** (2012), 46–54.

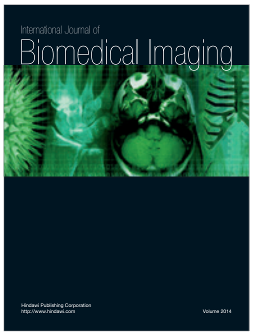
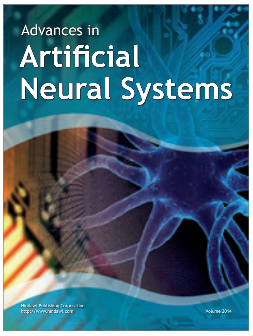
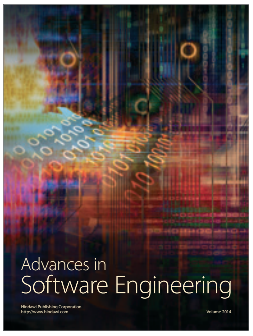
[16] D.W.I. Rouson, J. Xia and X. Xu, *Scientific Software Design: The Object-Oriented Way*, Cambridge Univ. Press. Cambridge, 2011.

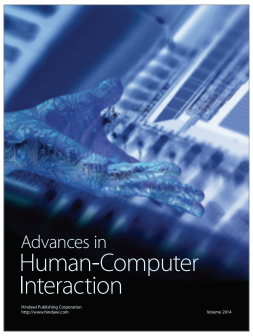
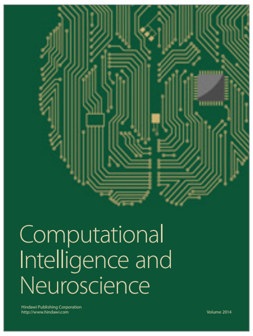
[17] S.F. Shandarin and Y.B. Zeldovich, The large-scale structure of the universe: turbulence, intermittency, structures in a sef-gravitating medium, *Rev. Modern Phys.* **61**(1) (1989), 185–220.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems


Modelling & Simulation in Engineering

Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering