ORNL/TM-2017/350

ForTrilinos Design Document

Mitchell Young (ORNL)
Seth Johnson (ORNL)
Andrey Prokopenko (ORNL)
Benjamin Collins (ORNL)
Katherine Evans (ORNL)
Mike Heroux (project PI) (SNL)

August 24, 2017

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2017/350

COMPUTER SCIENCE AND ENGINEERING DIVSION

FORTRILINOS DESIGN DOCUMENT

Mitchell Young (ORNL)
Seth Johnson (ORNL)
Andrey Prokopenko (ORNL)
Benjamin Collins (ORNL)
Katherine Evans (ORNL)
Mike Heroux (project PI) (SNL)

Date Published: August 24, 2017

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

With the development of a Fortran Interface to Trilinos, ForTrilinos, modelers using modern Fortran will be able to provide their codes the capability to use solvers and other capabilities on exascale machines via a straightforward infrastructure that accesses Trilinos. This document outlines what Fortrilinos does and explains briefly how it works. We show it provides a general access to packages via an entry point and uses an xml file from fortran code. With the first release, ForTrilinos will enable Teuchos to take xml parameter lists from Fortran code and set up data structures. It will provide access to linear solvers and eigensolvers. Several examples are provided to illustrate the capabilities in practice. We explain what the user should have already with their code and what Trilinos provides and returns to the Fortran code. We provide information about the build process for ForTrilinos, with a practical example. In future releases, nonlinear solvers, time iteration, advanced preconditioning techniques, and inversion of control (IoC), to enable callbacks to Fortran routines, will be available.

1. INTRODUCTION

The ForTrilinos project [Prokopenko et al., 2017] aims to provide Fortran developers with access to the Trilinos [Heroux et al., 2005] scientific library. This design document provides information about the plans to develop ForTrilinos to be maximally useful for Fortran codes that would like to access Trilinos.

Trilinos functionality will be exposed gradually, as our tooling becomes more sophisticated. First, popular high-level functionality will be provided, giving access to the Trilinos linear solvers, eigen solvers, and non-linear solvers. These will be implemented with a simplified interface, hiding much of the internal complexity of the Trilinos ecosystem from the end user. These are discussed in Section 4. Following the simplified interfaces, we will target Trilinos packages individually. Important user-facing objects and interfaces from these packages will be wrapped in Fortran derived types, giving end users more flexibly and fine-grained control. This functionality is still under development, and no specific interface is proposed, but some discussion is provided in Section 5.

The goals of ForTrilinos will be achieved through the development of Fortran modules that "wrap" Trilinos code by adapting the interface and calling the Trilinos code under the hood. These wrappers will rely heavily upon the C interoperability (`ISO_C_BINDING`) features introduced in the Fortran 2003 standard. Being written in C++, the Trilinos interfaces must also be adapted using an intermediate C interface that can be targeted by the Fortran wrapper code.

Throughout the process of generating the wrappers, our goal is to satisfy the following objectives:

- **Familiarity**

  Expose Trilinos functionality using an idiomatic Fortran interface. Existing Fortran users should find using ForTrilinos familiar.

- **Performance**

  Due to the nature of interfacing between Fortran and C++ code, there will necessarily be some impact on performance. However, performance impacts should be minimized as much as possible.

The initial version of ForTrilinos [Morris et al., 2012] wraps components of Trilinos in Fortran via some automated tooling, but is still largely manual. The manual aspect requires ongoing maintenance and provides little economy of scale, so the introduction of new Trilinos packages is time-consuming.

In response, the next generation of ForTrilinos leverages SWIG (Simplified Wrapper and Interface Generator) to maximize the automatic generation of ForTrilinos wrapper code. SWIG is designed to automate the process of exposing C++ code to foreign languages, such as Perl or Python. Applied to Fortran wrappers, SWIG minimizes future development and maintenance costs, and enables straightforward expansion to more Trilinos packages. In many cases, updates to Trilinos APIs will only involve re-running the SWIG tool to generate new wrapper code.

A significant portion of the new ForTrilinos effort involves the development of Fortran support in SWIG, which is being developed from scratch as part of this project. Although this is time intensive to accomplish, as support for more C++ features is added to the SWIG Fortran module, it will become easier to wrap more Trilinos capability, and the set of supported features can expand. To illustrate its utility, a simplified Trilinos interface within ForTrilinos has been provided to show basic functionality:

- Interaction with Teuchos parameter lists, both through a direct interface and using XML files.
- A simplified interface to the Stratimikos package for posing and solving sparse linear systems.

A similar simplified interfaces to access the Anasazi package ($Ax = \lambda x$ and $Ax = \lambda Bx$) and non-linear solver capabilities is also presented.

Near term developments beyond this basic functionality include direct wrappers for individual Trilinos packages. Some of these features introduce practical concerns which will require creative solutions. Chief among them is memory management of the C++ objects that are referenced by the Fortran proxy objects, which are discussed more in Section 3.1. Native Trilinos relies heavily on the scoping rules of C++ to automatically handle safely destroying objects and freeing the resources associated with them. Reference counting is often used to permit different scopes to share references to objects, freeing them when no references to that object remain. The corresponding Fortran proxy objects will need to interact reliably with this mechanism to avoid memory leaks and other issues. This is an issue that received much attention as part of the previous incarnation of ForTrilinos[Rouson and Morris, 2012], and we hope to incorporate some of their findings in our SWIG-based approach.

Facilities for inversion of control (IoC), allowing the Trilinos C++ code to call Fortran functions, will be needed to implement general non-linear solvers and preconditioners.

2. USING FORTRILINOS

2.1 OBTAINING AND BUILDING FORTRILINOS

ForTrilinos is an external Trilinos package, so it is built in the source tree together with regular Trilinos packages but it is not distributed with the main Trilinos repository.

To obtain and build ForTrilinos, clone the main Trilinos repository into a directory, here referred to as `$TRILINOS_DIR`:

```
git clone https://github.com/trilinos/Trilinos.git $TRILINOS_DIR
```

Into a separate directory, `$FORTRILINOS_DIR`, clone the ForTrilinos repository:

```
git clone https://github.com/trilinos/ForTrilinos.git $FORTRILINOS_DIR
```

Now create a symlink to ForTrilinos in the Trilinos packages directory:

```
cd $TRILINOS_DIR/packages
ln -s $FORTRLILINOS_DIR .
```

ForTrilinos only has explicit dependencies on Fortran and SWIG. However, as it wraps other Trilinos packages, it inherits their dependencies. To build and use the ForTrilinos code itself, a Fortran 2003 compliant Fortran compiler is required.*

Trilinos and ForTrilinos can now be configured and built. Here is an example of a configuration script for Trilinos to build ForTrilinos and its dependencies:

```
#!/usr/bin/env/bash

EXTRA_ARGS=$@

ARGS=(
    -D CMAKE_BUILD_TYPE=Debug

    -D BUILD_SHARED_LIBS=ON

    ### Install path ###
    -D CMAKE_INSTALL_PREFIX=${TRILINOS_INSTALL_DIR}

    ### COMPILERS AND FLAGS ###
    -D Trilinos_ENABLE_Fortran=ON
    -D CMAKE_CXX_FLAGS="-Wall -Wpedantic"

    ### TPLs ###
    -D TPL_ENABLE_MPI=ON
    -D TPL_ENABLE_BLAS=ON
    -D TPL_ENABLE_LAPACK=ON

    ### ETI ###
    -D Trilinos_ENABLE_EXPLICIT_INSTANTIATION=ON

    ### PACKAGES CONFIGURATION ###
    -D Trilinos_ENABLE_ALL_PACKAGES=OFF
    -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=OFF
```

*Recent versions of Cray, Intel, IBM, PGI and GNU fully support Fortran 2003. NAG as of version 6.1 lacks support for derived type input/output.

```
    -D Trilinos_ENABLE_TESTS=OFF
    -D Trilinos_ENABLE_EXAMPLES=OFF

    -D Trilinos_ENABLE_Amesos=ON
    -D Trilinos_ENABLE_AztecOO=ON
    -D Trilinos_ENABLE_Belos=ON
    -D Trilinos_ENABLE_Epetra=ON
    -D Trilinos_ENABLE_EpetraExt=ON
    -D Trilinos_ENABLE_Ifpack=ON
    -D Trilinos_ENABLE_Ifpack2=ON
    -D Trilinos_ENABLE_Stratimikos=ON
    -D Trilinos_ENABLE_Tpetra=ON
    -D Trilinos_ENABLE_Thyra=ON

    ### FORTRILINOS ###
    -D Trilinos_ENABLE_CTrilinos=ON
    -D Trilinos_ENABLE_ForTrilinos=ON
        -D ForTrilinos_ENABLE_EXAMPLES=ON
        -D ForTrilinos_ENABLE_TESTS=ON
    )
cmake "${ARGS[@]}" $EXTRA_ARGS $TRILINOS_DIR
```

do–configure.sh

Optionally, an install path may be specified with the `-D CMAKE_INSTALL_PREFIX` command line argument, which is useful on systems where the default install path is not writeable by the user. We assume that the environment variable `$TRILINOS_INSTALL_DIR` is used.

Using such a script, ForTrilinos can be configured, built and installed with:

```
mkdir build && cd build
./do-configure
make
make install
```

For more information on how to configure Trilinos and its dependencies, please refer to Trilinos documentation.

2.2 BUILDING PROGRAMS WITH FORTRILINOS

In many cases, ForTrilinos can simply be built as a library dependency of another project. After building ForTrilinos as described above, Fortran module files and shared object files should be located in the specified install directory, `$TRILINOS_INSTALL_DIR`. If Trilinos was not installed to the default system path, `$TRILINOS_INSTALL_DIR/include` must be specified as a search directory for module files (`-I$TRILINOS_INSTALL_DIR/include` in `gfortran`), and `$TRILINOS_INSTALL_DIR/lib` as a library search directory to the linker (`-I$TRILINOS_INSTALL_DIR/lib`).

3. GENERAL PRINCIPLES AND THE FORTEUCHOS MODULE

3.1 OBJECT MODEL

A goal of the ForTrilinos project is to provide Fortran interfaces to Trilinos that match as closely as possible to the existing interfaces of their C++ counterparts.

In general, this means defining Fortran derived types with type-bound procedures that map to methods of the corresponding C++ classes. We tend to refer to these as "proxy types." On the Fortran side, the proxy types store a pointer to an underlying C++ object, which is in turn typically stored in a reference-counted pointer (RCP) [Bartlett, 2010]. Storing a reference to an object created and allocated on the C++ side is necessary because in most cases, Trilinos classes are not directly interoperable with Fortran.

Since each ForTrilinos proxy object is actually a pointer to a C++ object, it is necessary to explicitly allocate and free the memory associated with that object. Most ForTrilinos types provide `foo%create()` and `foo%release()` type-bound procedures to allocate/construct and destroy/free the underlying C++ object, respectively. The `foo%create()` procedure may accept arguments, which are passed to the object's constructor.

The `Teuchos::RCP` is used under the hood because most of the Trilinos interfaces use this class to pass objects around. Most type-bound procedures on the Fortran objects follow a pattern of passing the arguments and internal pointer to an `extern "C"` function, which dereferences the pointer and calls the corresponding method on the pointed-to C++ class with the provided arguments. Interacting with a ForTrilinos object before `create()`-ing it will likely result in errors, or undefined behavior. Forgetting to `clear()` a ForTrilinos object will usually lead to memory leaks.

3.2 ERROR HANDLING

ForTrilinos defines a Fortran module integer *ierr*. SWIG will catch any exceptions thrown by Trilinos and will set that code as well as a string *serr* with the exception message. The error can be cleared by the fortran module by resetting *ierr* to zero; otherwise, the next time a function is called, the error will be rethrown as "Uncaught exception: msg".

Below is an example of how it is used in a Fortran code:

```
call some_function()
if (ierr /= 0) then
  write(*,*) "Got error ", ierr, ":", trim(serr)
  stop 1
endif
```

3.3 FORTEUCHOS MODULE

Many components of Trilinos rely upon utility code provided by the Teuchos package. The Teuchos package provides a parameter list class, which is used to configure many of the solvers provided by the other Trilinos packages. ForTrilinos exposes the Teuchos parameter list in the ForTeuchos module, which provides two methods for interacting with these parameter lists. A direct interface allows a Fortran

developer to interact with the parameter list programmatically using type-bound procedures on the `ParameterList` derived type. Alternatively, `ParameterLists` may be initialized using an XML file containing relevant parameters.

For example, the Stratimikos package provides a unified interface to many of the linear solvers in Trilinos, and relies heavily upon the parameter list for configuration. Below is an example of an XML file that would configure Stratimikos to use the AztecOO solver and Ifpack preconditioner:

```
<ParameterList>
  <Parameter name="Linear Solver Type" type="string" value="AztecOO"/>
  <ParameterList name="Linear Solver Types">
    <ParameterList name="AztecOO">
      <ParameterList name="Forward Solve">
        <ParameterList name="AztecOO Settings">
          <Parameter name="Aztec Solver" type="string" value="GMRES"/>
          <Parameter name="Convergence Test" type="string" value="r0"/>
          <Parameter name="Size of Krylov Subspace" type="int" value="300"/>
        </ParameterList>
        <Parameter name="Max Iterations" type="int" value="400"/>
        <Parameter name="Tolerance" type="double" value="1e-13"/>
      </ParameterList>
      <Parameter name="Output Every RHS" type="bool" value="1"/>
    </ParameterList>
  </ParameterList>
  <Parameter name="Preconditioner Type" type="string" value="Ifpack"/>
  <ParameterList name="Preconditioner Types">
    <ParameterList name="Ifpack">
      <Parameter name="Prec Type" type="string" value="ILU"/>
      <Parameter name="Overlap" type="int" value="1"/>
      <ParameterList name="Ifpack Settings">
        <Parameter name="fact: level-of-fill" type="int" value="2"/>
      </ParameterList>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

example_plist.xml

The program below demonstrates how to use the above XML populate a ForTeuchos `ParameterList` and interact with it.

```
program import_xml
    use ForTeuchos

    implicit none

    type(ParameterList) :: plist, sublist, sublist2
    logical :: some_bool
    integer :: max_iterations

    call plist%create("base")
    call sublist%create("sublist")

    call load_from_xml(plist, "example_plist.xml")

    call plist%print()

    write(*,*)"done printing base list"
```

```
    call plist%get("Linear Solver Types", sublist)
    call sublist%get("AztecOO", sublist)
    call sublist%get("Forward Solve", sublist)
    call sublist%print()
    call sublist%get("Max Iterations", max_iterations)

    write(*,*)"max iterations:", max_iterations

    call plist%release()

end program
```

4. SIMPLIFIED TRILINOS INTERFACE

The goal of the simplified interface is to expose the high level Trilinos functionality that is easy to understand for Fortran users. Rather than expose the low-level API for many Trilinos structures, the simplified interface provides a small interface to full capability areas of Trilinos through a single entry point. This is done through the use of an intermediate C++ code (hidden from Fortran users and Trilinos) that hides the Trilinos functionality and class interactions.

So far, the implemented simplified interfaces expose only linear solvers (section 4.1). We are also working on eigensolvers interface (section 4.2). We also consider interfaces for the nonlinear solvers (section 4.3).

4.1 LINEAR SOLVERS

The linear solvers interface has been partially implemented, and are operational, at the beta testing level. The Fortran interface consists of a single class with the following API:

- *create()*

  Allocate a new handle. Each handle corresponds to a single linear system. This allocation must be done before the handle can be used. Multiple handles may be present at the same time.

- *release()*

  Free resources associated with a specific handle. Does not affect any other handles. A call to *create()* should always be paired with a corresponding *release()* to avoid memory leaks.

- *init()*, *init(comm)*

  Initialize handle. The second variant takes in an MPI communicator.

- *setup_matrix(row_inds, row_ptrs, col_inds, values)*

  Specify a matrix for the linear system. For a serial run this is the full system matrix. For a parallel run, this is a subset of rows of the full matrix that are allocated to a single processor. We call it local matrix. The matrix is required to be provided in a standard CSR[†] format:

  – *row_inds* (in)

    The global indices of the local matrix rows allocated to this processor. It is an integer array of size *num_rows*.

  – *row_ptrs* (in)

    Row offsets in the local matrix (the first array in CSR). It is an integer array of size *num_rows+1*.

  – *col_inds* (in)

    Global column indices in the local matrix (the second array in CSR). It is an integer array of size *num_nnz* (which must be equal to *row_ptrs(num_rows+1)*).

[†]Compressed Sparse Row, also known as CRS: Compressed Row Storage

- *values* (in)

  Values in the local matrix (the third array in CSR). It is a double array of size *num_nnz*.

- *setup_operator(row_inds, func_ptr)*

  An alternative way (to *setup_matrix*) to specify the system. Instead of providing an explicit matrix, a user instead provides a function pointer that implements matrix-vector multiplication. The current API for it is

```
subroutine matvec(n, x, y) BIND(C)
  use, intrinsic :: ISO_C_BINDING
  integer(c_int), intent(in), value :: n
  real(c_double), dimension(:), intent(in) :: x(*)
  real(c_double), dimension(:), intent(out) :: y(*)
  ...
end subroutine
```

  row_inds are the same as in *setup_matrix*.

- *setup_solver(param_list)*

  Specify parameters for the solver. It provides a full access to linear solver algorithms in Trilinos that provide Stratimikos adapters. These include:

  - Krylov iterative methods

    CG, GMRES, ...

  - Classic preconditioners

    Jacobi, Gauss-Seidel, SOR

  - ILU preconditioners

    Several variants including ILU(k), ILUT

  - Multigrid preconditioners

  For more information about available preconditioners, please consult Ifpack2 [Prokopenko et al., 2016] and MueLu [Prokopenko et al., 2014] user manuals.

  Note that if a user provides an operator through *setup_operator* call, the preconditioner functionality is no longer available as it requires an explicit matrix.

  This function must be called after the matrix or the operator has been setup.

- *solve(rhs, lhs)*

  Solve the linear system.

  - *rhs* (in)

    The local part of the right hand side of the system. It is a double array of size *size* (which must be the same as *num_rows* in *setup_matrix* or *setup_operator*).

- *lhs* (out)

  The local part of the solution side of the system. It is a double array of size *size*.

This function must be called after *setup_solver*. It can be called multiple times to solve multiple systems with the same operator.

Below is an example of the linear solver interface in use (note, it uses a slightly different interface where for all fortran arrays we also provide array sizes; this will not be necessary in the final interface, but this is the way current implementation works). The problem uses the following XML file to populate the parameter list used to configure the problem:

```
<ParameterList>
  <Parameter name="Linear Solver Type" type="string" value="Belos"/>
  <ParameterList name="Linear Solver Types">
    <ParameterList name="Belos">
      <Parameter name="Solver Type" type="string" value="Block GMRES"/>
      <ParameterList name="Solver Types">
        <ParameterList name="Block GMRES">
          <Parameter name="Block Size" type="int" value="1"/>
          <Parameter name="Convergence Tolerance" type="double" value="1e-4"/>
          <Parameter name="Maximum Iterations" type="int" value="20"/>
          <Parameter name="Output Frequency" type="int" value="1"/>
          <Parameter name="Show Maximum Residual Norm Only" type="bool" value="1"/>
        </ParameterList>
      </ParameterList>
    </ParameterList>
  </ParameterList>

  <Parameter name="Preconditioner Type" type="string" value="Ifpack2"/>
</ParameterList>
```

narrative/simple_tridiag.xml

```
program main

#include "ForTrilinosSimpleInterface_config.hpp"


  use ISO_FORTRAN_ENV
  use, intrinsic :: ISO_C_BINDING
  use fortrilinos
  use x
  use forteuchos
#ifdef HAVE_MPI
  use mpi
#endif
  implicit none

  integer(c_int) :: i
  integer(c_int) :: n, nnz;
  integer(c_int) :: my_rank, num_procs

  integer(c_int), dimension(:), allocatable :: row_inds, col_inds, row_ptrs
  real(c_double), dimension(:), allocatable :: lhs, rhs, values

  integer(c_int) :: cur_pos, offset
  real(c_double) :: norm
```

```
  type(ParameterList) :: plist
  type(TrilinosHandle) :: tri_handle

  n = 50
  nnz = 3*n

  my_rank = 0
  num_procs = 1

#ifdef HAVE_MPI
  ! Initialize MPI subsystem
  call MPI_INIT(ierr)
  if (ierr /= 0) then
    write(*,*) "MPI failed to init"
    stop 1
  endif

  call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
  EXPECT_EQ(0, ierr)
#endif

  ! Read in the parameterList
  call plist%create("Stratimikos")
  call load_from_xml(plist, "stratimikos.xml")

  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ":", trim(serr)
    stop 1
  endif

  ! -----------------------------------------------------------------
  ! Construct tri-diagonal matrix, and rhs
  allocate(row_inds(n))
  allocate(row_ptrs(n+1))
  allocate(col_inds(nnz))
  allocate(values(nnz))
  row_ptrs(1) = 0
  cur_pos = 1
  offset  = n * my_rank
  do i = 1, n
    if (i .ne. 1 .or. my_rank > 0) then
      col_inds(cur_pos) = offset + i-1
      values  (cur_pos) = -1.0
      cur_pos = cur_pos + 1
    end if
    col_inds(cur_pos) = offset + i
    values  (cur_pos) = 2.0
    cur_pos = cur_pos + 1
    if (i .ne. n .or. my_rank .ne. num_procs-1) then
      col_inds(cur_pos) = offset + i+1
      values  (cur_pos) = -1.0
      cur_pos = cur_pos + 1
    end if
    row_ptrs(i+1) = cur_pos-1;

    row_inds(i) = offset + i
  end do
  nnz = cur_pos-1
```

```
  allocate(lhs(n))
  do i = 1, n
    lhs(i) = 0.0
  end do

  ! The solution lhs(i) = i-1
  allocate(rhs(n))
  if (my_rank > 0) then
    rhs(1) = 0.0
  else
    rhs(1) = -1.0
  end if
  if (my_rank .ne. num_procs-1) then
    rhs(n) = 0.0
  else
    rhs(n) = offset+n
  end if
  do i = 2, n-1
    rhs(i) = 0.0
  end do


  call tri_handle%create()
  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ": ", trim(serr)
    stop 1
  endif

  ! ------------------------------------------------------------------
  ! Setup ForTrilinos handle and solve
  ! ------------------------------------------------------------------
  ! Step 1: initialize a handle to the simplified linear solver interface
#ifdef HAVE_MPI
  call tri_handle%init(MPI_COMM_WORLD)
#else
  call tri_handle%init()
#endif
  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ": ", trim(serr)
    stop 1
  endif

  ! Step 2: setup the problem
  call tri_handle%setup_matrix(n, row_inds, row_ptrs, nnz, col_inds, values)
  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ": ", trim(serr)
    stop 1
  endif

  ! Step 3: setup the solver
  call tri_handle%setup_solver(plist)
  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ": ", trim(serr)
    stop 1
  endif

  ! Step 4: solve the system
  call tri_handle%solve(n, rhs, lhs)
```

```
  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ": ", trim(serr)
    stop 1
  endif

  ! Check the solution
  norm = 0.0
  do i = 1, n
    norm = norm + (lhs(i) - (offset+i-1))*(lhs(i) - (offset+i-1));
  end do
  norm = sqrt(norm);

  ! Step 5: clean up
  call tri_handle%finalize()
  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ": ", trim(serr)
    stop 1
  endif


! ------------------------------------------------------------------
  call plist%release()
  call tri_handle%release()
  if (ierr /= 0) then
    write(*,*) "Got error ", ierr, ":", trim(serr)
    stop 1
  endif

#ifdef HAVE_MPI
  ! Finalize MPI must be called after releasing all handles
  call MPI_FINALIZE(ierr)
  EXPECT_EQ(0, ierr)
#endif

end program
```

narrative/simple_tridiag_example.f90

Current API questions/concerns:

- No API to check the status of the solve. Did it converge?
- No way to get back the number of iterations. Do we need it for the simple API?
- All the input is controlled through the parameter list. Do we want to provide an easier way to set some common parameters, like convergence tolerance?
- How many different interfaces should we have, and how complicated?
- do we separate all the different interfaces or have one that works for all?

4.2 EIGENSOLVERS

Eigensolvers, by their nature, are very similar to linear systems. There are two main differences. First, instead of solving a linear system by providing a right hand side and producing a solution, one instead asks for a number of eigenvalues and associated eigenvectors to be produced. Second, in the case of generalized

eigenvalue problems, the problem is represented by two matrices, rather than just one. As such, the eigensolver interface is very similar to the linear solvers. The eigensolver interface drives the Anasazi Trilinos package, which can solve typical eigenvalue problems ($A\mathbf{x} = \lambda\mathbf{x}$) and generalized eigenvalue problems ($A\mathbf{x} = \lambda B\mathbf{x}$). Anasazi implements a handful of algorithms, each with different benefits and constraints; refer to the Anasazi documentation to determine which algorithm best suits a given problem. The desired solver must be selected using the parameter list passed to the `setup_solver()` function, along with relevant parameters to configure the selected solver.

- *create()*

  Allocate a new handle. Each handle corresponds to a single eigenvalue problem. This allocation must be done before the handle can be used. Multiple handles may be present at the same time.

- *release()*

  Free resources associated with a specific handle. Does not affect any other handles. A call to *create()* should always be paired with a corresponding *release()* to avoid memory leaks.

- *init()*, *init(comm)*

  Initialize handle. The second variant takes in an MPI communicator.

- *setup_matrix(row_inds, row_ptrs, col_inds, values)*

  Specify a matrix (A in $A\mathbf{x} = \lambda\mathbf{x}$ or $A\mathbf{x} = \lambda B\mathbf{x}$) for the eigenvalue problem.

  Refer to `setup_matrix()` from Section 4.1 for more details.

- *setup_operator(row_inds, func_ptr)* An alternative way to represent the system. Refer to `setup_operator` in Section 4.1 for more details.

- *setup_matrix_rhs(row_inds, row_ptrs, col_inds, values)*

  Specify a matrix for the right-hand side of a generalized eigenvalue problem (B in $A\mathbf{x} = \lambda B\mathbf{x}$).

- *setup_operator_rhs(row_inds, func_ptr)*

  An alternative way to specify the RHS system discussed in `setup_matrix_rhs`.

- *setup_solver(param_list)*

  Specify parameters for the solver. The supplied parameter list should contain the following parameters:

  - "Solver Type": The solver algorithm to use. This can be one of the solvers implemented in the Anasazi package. For example "Block Krylov-Schur" or "Generalized Davidson".
  - "NumEV": An integer specifying the number of eigenvalues to seek. These will be ordered based on the "which" parameter specified on the solver-specific parameter list.
  - "[Solver Type]": A parameter sublist containing appropriate settings for the selected solver type. The name of the sublist should match the name of the solver selected. Refer to the Anasazi documentation for the list of relevant parameters for each solver (e.g. Block Davidson).
  - "Preconditioner Type": Optional. A preconditioner type to use. This can be any of those offered by a Trilinos package, such as "Ifpack" or "ML."

- "[Preconditioner Type]": Optional. A parameter sublist containing appropriate settings for the selected preconditioner, if any.

- *solve(evals, evecs)*, *solve(evals, evecs, eguess)*

  Solve the eigenvalue problem, returning all requested eigen pairs. The second variant allows an initial guess to be supplied for each requested eigenvector

  - *evals* (out)

    A pointer to the requested eigenvalues. The passed pointer will be associated with data allocated by ForTrilinos.

  - *evecs* (out)

    A two-dimensional pointer to the requested eigenvectors. Individual eigenvectors may be obtained by `evecs(:, i)`. The passed pointer will be assiociated with data allocated by ForTrilinos.

  - *eguess* (in)

    A `pointer(local_size, nev)` to an array of the local part of the initial guess to use. The pointer should be sized such that `local_size` is the size of the local part of the eigenvectors and `nev` is the number of eigen pairs that are being solved for.

  This function must be called after *setup_solver*.

4.3 NONLINEAR SOLVERS

We would also like to provide a simplified nonlinear solver interface. The main difficulty in designing such an interface is how to pass a nonlinear operator to Trilinos. This will require a proper implementation of the Inversion-of-Control (IoC). Although we have plans to implement wrappers within ForTrilinos soon, this document does not address this strategy. A follow-on design document with ths scope is planned.

5. DIRECT TRILINOS WRAPPERS

The simplified interface sacrifices much of the versatility of the Trilinos library to provide an easier-to-use approach for common use cases. For more advanced applications, it is useful to have direct access to the Trilinos packages, and the classes that they comprise. Once the SWIG Fortran capabilities evolve to a point where it becomes possible, selected Trilinos classes will be exposed to Fortran developers directly. Due to fundamental differences between Fortran and C++, it will not be possible or desirable to achieve a one-to-one correspondence between the C++ classes and wrapped Fortran derived types.

At this point it is too early to propose specific interfaces for individual Trilinos packages. However, we seek to establish package priorities to guide our efforts going forward. Below is a short list of Trilinos packages listed in order of general utility and overall priority. User interest in any package not on this list would make excellent feedback.

1. Teuchos (partially implemented)

   Teuchos is a support package in Trilinos. It provides a lot of auxiliary structures used throughout such as parameter lists. A subset of this functionality will be exposed to Fortran users.

2. Epetra

   Epetra is the older version of Tpetra. It is in maintenance mode, and no more development is being done. However, it is still heavily used by many people. It is unclear if it is worth for ForTrilinos to provide an interface to it if we provide an interface to Tpetra.

3. Tpetra

   Tpetra implements distributed linear algebra objects such as matrices and vectors. Our plan is to expose a significant part of this functionality, helping Fortran users to assemble matrices and vectors in parallel, and allowing access to certain communication patterns.

4. Stratimikos

   Stratimikos provides a single interface to many of the linear solvers and preconditioners provided by Trilinos. This package is already used under the hood for the simplified linear solver interface described in Section 4.1, though wrapping it directly may prove useful for some users.

5. Anasazi

   Anasazi provides eigenvalue solvers for large, distributed matrices. It is already being used internally by the simplified eigensolver interface described in Section 4.2.

6. Kokkos

   Kokkos is the X in the MPI+X programming paradigm, and is heavily used by Tpetra underneath. It is also very suitable to be used on its own to provide on-node performance. ForTrilinos may benefit from exposing some of its functionality, however it may pose a significant challenge due do the advanced concepts used in the package (heavy template metaprogramming, C++11 requirement). ForTrilinos may provide a simplified functionality that wraps specific instances of *Kokkos::View* for certain template parameters.

7. Other packages:

   - Intrepid2 (discretization)

- Zoltan2 (repartitioning)
- SEACAS (I/O)

6. CONCLUSIONS

The ForTrilinos project aims to expose the power of the Trilinos ecosystem to Fortran developers in a manner that should be very accessible and familiar. This will be achieved by using tools developed as part of the SWIG system, reducing the cost of development and continued maintenance of the interfaces.

Simplified interfaces have been proposed to provide access to fundamental Trilinos features. The developers welcome input from the community, and all comments and suggestions regarding the proposed interfaces are welcome.

ForTrilinos development is ongoing. As the SWIG tools mature, more fine-grained interfaces to the underlying Trilinos packages will be proposed, as well as facilities to enable advanced features such as inversion of control, which is a necessary component for many non-linear and general-purpose solvers.

Updated versions of this document will be made available, as user comments are addressed and more functionality is added.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This manuscript has been authored by UT-Battelle, LLC and used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, both of which are supported by the Office of Science of the U.S. Department of Energy under Contract No.DE-AC05-00OR22725. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

R. A. Bartlett. Teuchos::RCP beginner's guide. Technical Report SAND2004-3268, Sandia National Labs, 2010.

M. A. Heroux, R. A. Bartlett, V. E. Howe, 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, and A. Williams. An overview of the Trilinos project. *ACM Trans. Math. Soft.*, 31(3):397–423, 2005.

K. Morris, M. N. L. D.W.I. Rouson, and S. Filippone. Exploring capabilities within ForTrilinos by solving the 3D burgers equation. *Sci. Programming*, 20(3):275–292, 2012.

A. Prokopenko, J. J. Hu, T. A. Wiesner, C. M. Siefert, and R. S. Tuminaro. MueLu User's Guide 1.0. Technical Report SAND2014-18874, Sandia National Labs, 2014.

A. Prokopenko, C. M. Siefert, J. J. Hu, M. Hoemmen, and A. Klinvex. Ifpack2 User's Guide 1.0. Technical Report SAND2016-5338, Sandia National Labs, 2016.

A. Prokopenko, S. Johnson, M. Young, K. Evans, M. Heroux, and B. Collins. Fortrilinos code base. https://github.com/Trilinos/ForTrilinos, 2017.

D. Rouson and K. Morris. This isn't your parents' Fortran: Managing C++ objects with modern Fortran. *Computing in Science and Engineering*, 14:46–54, 2012. doi: 10.1109/MCSE.2012.33.