A Non-intrusive Technique for Interfacing Legacy Fortran Codes with Modern C++ Runtime Systems

Zachary D. Byerly*§, Hartmut Kaiser*§, Steven Brus†, Andreas Schaefer‡§
*Center for Computation and Technology, Louisiana State University, Baton Rouge, LA, U.S.A.
†Department of Civil Engineering and Geological Sciences, University of Notre Dame, Notre Dame, IN, U.S.A.
‡Computer Science 3, Computer Architectures, Friedrich-Alexander-University, Erlangen, Germany
§ The STE||AR Group (http://www.stellar-group.org)
zbyerly@cct.lsu.edu, hkaiser@cct.lsu.edu, sbrus@nd.edu, andreas.schaefer@fau.de

***Abstract*—Many HPC applications developed over the past two decades have used Fortran and MPI-based parallelization. As the size of today's HPC resources continues to increase, these codes struggle to efficiently utilize the million-way parallelism of these platforms. Rewriting these codes from scratch to leverage modern programming paradigms would be time-consuming and error-prone. We evaluate a robust approach for interfacing with next-generation C++-based libraries and drivers. We have successfully used this technique to modify the Fortran code DGSWEM (Discontinuous Galerkin Shallow Water Equation Model), allowing it to take advantage of the new parallel runtime system HPX. Our goal was to make as few modifications to the DGSWEM Fortran source code as possible, thereby minimizing the chances of introducing bugs and reducing the amount of re-verification that needed to be done.**

I. Introduction

Many scientific codes have been developed, tested, and verified in Fortran. Some of these codes, despite being parallelized, struggle to scale well on current extreme-scale, heterogeneous architectures.

We were faced with the task of optimizing an existing Fortran code (DGSWEM, a discontinuous Galerkin shallow water equations model) [1]. DGSWEM currently uses pure MPI-1.0 for parallelization. In strong-scaling scenarios with multi-core CPUs this leads to a low computational load per process, compared to hybrid OpenMP/MPI codes. We intended to interface DGSWEM with HPX [2], a next-generation parallel runtime system. Utilizing HPX will allow us to leverage the shared intra-node address space while simultaneously enhance overlap of calculation and communication. HPX also supports transparent migration of data between nodes to equalize the computational load.

In between DGSWEM and HPX, we will use LibGeoDecomp (LGD) [3], an auto-parallelizing library for petascale computer simulations. LGD has proven to scale to tens of thousands of nodes and millions of processes on supercomputers ranging from Tokyo Tech's Tsubame 2.5 and JSC's JUQUEEN to ORNL's Titan. Relevant features in this context are dynamic load balancing, accelerator offloading and parallel IO. However, HPX and LGD are both written in C++, so in order to use them for DGSWEM, we must find a method for calling Fortran subroutines from the C++ code in a threadsafe manner. Figure 1 shows the software stack used for the DGSWEM/LGD/HPX code.

In this paper we describe a non-intrusive, easy to implement, thread-safe interface between Fortran subroutines and a C++ driver.

II. Comparisons to Alternative Methods

Before discussing our approach, we will briefly discuss alternate methods and why we chose not to employ them. This is not to meant to be a comprehensive list of methods for interfacing C++ and Fortran codes.

A. Rewriting Fortran Source Code in C/C++

Rewriting a complicated code is time consuming and error prone. Even when performance barriers prevent a domain scientist from achieving certain research goals, they may prioritize their available time into other improvements of the code, e.g. increasing model accuracy. Completely rewriting a code also has the potential to introduce bugs, which may be very subtle and are not easily detected, depending on the nature of the changes made to the code in translation. Additionally, domain scientists who are accustomed to programming in Fortran may not want to learn how to program in a new language.

One variation on this method that we considered was to perform an automatic source-to-source translation at compile time. The Fortran source would remain untouched (except perhaps for some annotations to facilitate the automatic translation), and when the C++ version of the code was compiled, the source could would be automatically translated to C or C++.

Examples for such source-to-source systems are `f2c` [4] or the C/C++ backends of the LLVM (Low-Level Virtual Machine) compiler framework. `f2c` does not appear to be in active maintenance and it was unable to process DGSWEM's source code.

We tried the C and C++ backend in LLVM with DGSWEM's Fortran source. The first obstacle we hit was that LLVM does not have a Fortran frontend. This could be negotiated by using Dragonegg, a GCC plug-in which can generate LLVM IR (Intermediate Representation) when

running inside of `gfortran`. The C backend of LLVM, which was supposed to generate C code from LLVM IR, was removed from LLVM's code base in 2014. The newer C++ backend generates code that interpretes the abstract syntax tree given in the IR. That code has hardly any structural resemblance with the original code, hence interfacing with any C++ library becomes even more difficult.

We believe that creating a custom source-to-source translation system which is robust enough to be used in an automated fashion would be difficult to develop, and create an additional maintenance burden.

B. Passing Variables as Subroutine Arguments

Our original plan, which we began to implement, was to refactor the original Fortran code such that all variables, rather than being stored in Fortran modules, were passed directly to the subroutines that required them. This method would have been thread safe, and all the variables passed back and forth between subroutines would have explicitly shown which variables were being used and changed in each subroutine. However, this was a time-consuming and error-prone process, and would have ultimately required that the C++ code have a list of every persistent variable used in the Fortran code. This means that virtually any change to the Fortran code would require a change in the C++ code as well. This would increase the likelyhood that the MPI-based and LGD/HPX codes would diverge.

C. Shadow-object Interface

Grey et. al. [5] describe a method for interfacing C++ and Fortran data structures in a manner that would allow both languages to have full access to all members of the data structure. While this certainly has its benefits, the added complication of recreating these data structures on the C++ side of the code does not seem to be worth the extra effort. Additionally, this method restricts the list of parameters to 1D arrays, which is not sufficient for DGSWEM, which contains many arrays of higher dimension. Again, maintaining two separate descriptions of all data structures is tedious and error-prone.

III. Overview

In a nutshell, we convert all Fortran modules, which are collections of quasi-global variables, to user-defined datatypes. Modules are problematic when used with multiple threads as each thread will access the same set of variables.

Fortran 95 allows the creation of user-defined data types. These structures can contain components of different types, including allocatable arrays. We use these structures to store all of the state data from the simulation. We then use the `ISO_C_BINDING` [6] to get the C address of the instance of the user-defined data type. This memory address is then passed to the C++ program. All of the Fortran source code must store any persistent variables in these data structures. When calling a Fortran subroutine from the C++ code, these addresses are passed as arguments. This gives the Fortran code access to all

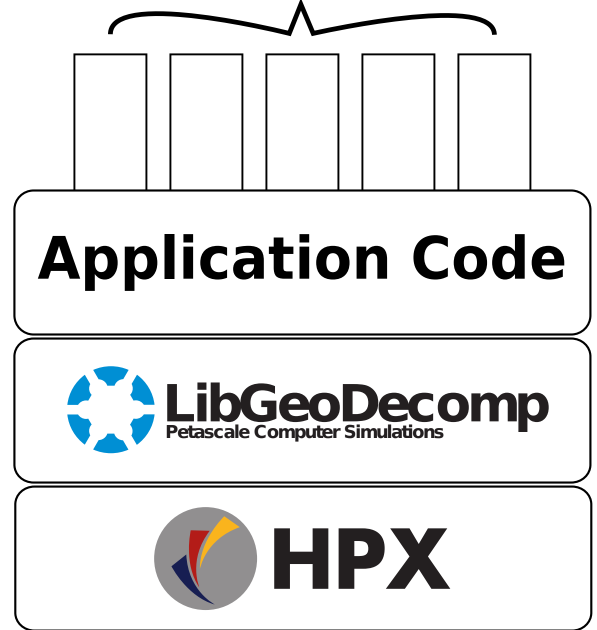

Fig. 1. Software Stack of the DGSWEM/LGD/HPX application

of the members of the data structure, while the C++ code does not need to know the details of the structure.

IV. Implementation

The first step, if not done already, is to place important variables inside a user-defined type inside of a module. Many instances of this user-defined type can then be created, each corresponding to a different subsection of the computational problem space. In the case of DGSWEM, these subsections are subregions of an unstructured mesh representing a geographical area. In our example code, these are pieces of an array, along with a integer `id`, and a parameter `ARRAY_SIZE`:

```
module global

  integer, parameter :: ARRAY_SIZE = 10

  type global_type
     integer :: id
     integer, allocatable, dimension(:)
         :: var
  end type global_type

end module global
```

Note that the parameter cannot be inside of the `global_type`, but because it is a read-only value, this does not pose any thread-safety problems.

Setting up the C++ program to call a Fortran subroutine requires using a name mangling procedure. We put this into

a file fname.h and included it in the C++ code. Fortran subroutines are then declared like so:

```
extern"C" {
  void FNAME(init_fort)(int *n, void**
      global);
  void FNAME(print_fort)(void** global);
  void FNAME(term_fort)(void** global);
}
```

To create an instance of one of these data structures, we call init_fort from the C++ code, passing it an integer ID:

```
FNAME(init_fort)(&ids[i],&domains[i]);
```

This calls a wrapper function:

```
subroutine init_fort(n, global_c_ptr)
  use, intrinsic :: iso_c_binding
  use global
  implicit none

  integer :: n
  type (C_PTR), intent(out) ::
      global_c_ptr
  type (global_type), pointer :: g
  allocate(g)

  call init(n, g)

  global_c_ptr = C_LOC(g)
end subroutine init_fort
```

Notice that the global_c_ptr is being passed out of the subroutine. g is a pointer to an instance of a global_type which is allocated in this subroutine. id and g are passed to the Fortran subroutine init, below. After it is initialized, the C_LOC function retrieves the c memory address of the instance g and passes this back to the c++ main program. The init subroutine allocates the var array and assigns it some values:

```
subroutine init(id, g)
  use global
  implicit none

  !incoming variable
  integer :: id

  !loop variable
  integer :: i

  type(global_type) :: g
  g%id = id

  ! allocate array
  allocate(g%var(ARRAY_SIZE))

  ! initialize arrays
  do i=1,ARRAY_SIZE
     g%var(i) = g%id+i
  end do
end subroutine init
```

Notice that each variable that is a member of the g object is accessed by prepending the variable name with g%.

When the init_fort function returns to the C++ code, the C++ code has a memory address to the g instance of the global_type Fortran data structure. This memory address can be passed around inside the C++ code without the the C++ code need to know any details of the global_type data structure. In our example code, we create many instances of these structures, each unique, and each with a unique associated pointer. In order to perform an operation using the data stored in a particular global_type instance, we simply pass the corresponding pointer associated with that particular instance to a Fortran subroutine.

For example, if we want to print the values of the var array (the C++ code doesn't know what's in the var array), we can pass the pointer to another Fortran subroutine wrapper:

```
subroutine print_fort(global_c_ptr)
  use, intrinsic :: iso_c_binding
  use global
  implicit none

  type (C_PTR) :: global_c_ptr
  type (global_type), pointer :: g

  call C_F_POINTER(global_c_ptr,g)

  call print(g)

end subroutine print_fort
```

This wrapper calls C_F_POINTER to associate the Fortran pointer g with the structure located at global_c_ptr. We can then call print:

```
subroutine print(g)
  use global
  implicit none

  !loop variable
  integer :: i

  type(global_type) :: g

  print*, "domain id:",g%id

  do i=1,ARRAY_SIZE
     print*, i, g%var(i)
  enddo

end subroutine print
```

For the sake of completeness, here is the C++ code:

```
#include <iostream>
#include "fname.h"
#include <vector>

extern"C" {
  void FNAME(init_fort)(int *n, void**
      global);
  void FNAME(print_fort)(void** global);
  void FNAME(term_fort)(void** global);
}

int main(
          int argc
          , char* argv[]
          )
{
  int n_domains = 10;

  std::vector<void *> domains;
  std::vector<int> ids;

  // Create a vector of null pointers
  for(int i=0; i<n_domains; i++){
    void *domain = NULL;
    domains.push_back(domain);
    ids.push_back(i);
  }

  std::cout << "Initializing domains... "
      ;
  for (int i=0; i<domains.size(); i++){
      FNAME(init_fort)(&ids[i],&domains[i
          ]);
  }
  std::cout << "done." << std::endl;

  std::cout << "Printing domains... " <<
      std::endl;
s   for (int i=0; i<domains.size(); i++){
      FNAME(print_fort)(&domains[i]);
  }
  std::cout << "done." << std::endl;



  std::cout << "Terminating domains... "
      << std::endl;
  for (int i=0; i<domains.size(); i++){
      FNAME(term_fort)(&domains[i]);
  }
  std::cout << "done." << std::endl;

  return 0;
}
```

This simple example code can be compiled with gfortran and g++.

V. Advantages & Limitations

Because the C++ code only sees an opaque memory block, there is no need to recreate the complicated data structure in the C++ code. This is also a limitation, but one with an easy workaround. In order to extract any data from the user-defined-type, we can create a simple Fortran subroutine where we pass the pointer to the data structure, package the information into one variable or an array, and pass this back to the C++ code. In our case these marshaling functions were basically already present in the Fortran source, as they were used to pack data for the original MPI parallelization.

Because we pass all of the state variables to the Fortran subroutines as pointers to structures, we can call multiple instances of the Fortran subroutines on different data structures simultaneously without complications normally associated with multi-threading.

We have taken this method and applied it to a large pre-existing code, DGSWEM. This code already had most of its variables in modules, so these modules were left intact, but the non-parameter variables put inside of a user-defined type inside of the module. It was a laborious task to then add the `g%` (or equivalent) to the start of each variable contained in one of the modules, but this process could be automated better in the future.

Running the code in a multi-threaded environment presented some additional issues. For example, we were able to reuse the input/output routines already present in the code. However, the Fortran file unit numbers would by default be the same for each instance, and these needed to be modified such that each subdomain had its own unique file number for a particular input or output file.

The nature of this code meant that it was necessary to implement boundary exchange between neighboring subdomains. This was accomplished by passing the pointers to a specially written Fortran subroutine which packaged the boundary values and sent them back to the C++ code as an array of doubles. This array of doubles was then passed, along with the pointer to the neighboring subdomain, to a second subroutine, which placed the boundary values in their appropriate place. This method works even when the two subdomains don't reside in the same shared memory space.

The DGSWEM code, modified using the procedure described above, and coupled with the LGD and HPX libraries has been tested and verified running on multiple threads and multiple localities (compute nodes). One current limitation is that we have no facility for packaging up an entire subdomain and moving it from one compute node to another, which presents challenges for dynamic load balancing.

Perhaps the most important advantage of this method was that once all of the changes were made, the original pure Fortran MPI-based functionality was left intact. This means that developers of the code could choose to use either the new C++ based LGD/HPX or the original Fortran-MPI parallelization methods. Any changes to the code that were purely local

(did not require extra communication between neighboring subdomains) could be done in this overlapping code base and wouldn't require any additional changes to be compatible with either of the parallelization methods.

A comparison between a Fortran-only and a C++ code which both do the same operations have nearly identical execution time and memory footprint. Any overheads associated with using this method are small, especially compared to the potential performance gains associated that some codes will see when using more modern parallelization libraries.

Overall, the advantages of this system greatly outweigh the disadvantages. Speed of implementation, the retention of most of the original source code (including input and output routines), the flexibility of the application developer to use the new or old parallelization scheme, and the relatively simple C++ code required to drive the Fortran subroutines are all very desirable features to have in a code migration.

VI. Conclusions

The method we have presented shows promise as a relatively painless way for legacy code users and developers to take advantage of great leaps in parallel computing. Leaving large amounts of the original code intact means that the introduction of subtle bugs is unlikely. Verification and validation of the code using the new parallelization methods is much easier. Even for large, complicated codes, the time it takes to perform the necessary transformations is short compared to any of the alternatives, and has relatively few disadvantages. As computing resources become larger and more heterogeneous, use of flexible parallel runtime systems like HPX will become more and more important. Providing methods for domain scientists to painlessly transform their codes to leverage libraries like LibGeoDecomp or the HPX runtime will greatly increase the utilization of these resources.

VII. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Number 1339782 (STORM). We would also like to thank the anonymous reviewers for their helpful comments.

References

[1] C. Dawson, E. J. Kubatko, J. J. Westerink, C. Trahan, C. Mirabito, C. Michoski, and N. Panda, "Discontinuous galerkin methods for modeling hurricane storm surge," *Advances in Water Resources*, vol. 34, no. 9, pp. 1165–1176, 2011.

[2] H. Kaiser, T. Heller, A. Berge, and B. Adelstein-Lelbach, "HPX V0.9.10: A general purpose C++ runtime system for parallel and distributed applications of any scale," 2015, http://github.com/STEllAR-GROUP/hpx. [Online]. Available: http://dx.doi.org/10.5281/zenodo.16302

[3] A. Schäfer, T. Heller, and D. Fey, "A portable petascale framework for efficient particle methods with custom interactions," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 83:83–83:89. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642787

[4] S. I. Feldman, "A fortran to c converter," *SIGPLAN Fortran Forum*, vol. 9, no. 2, pp. 21–22, Oct. 1990. [Online]. Available: http://doi.acm.org/10.1145/101363.101366

[5] M. Gray, R. Roberts, and T. Evans, "Shadow-object interface between fortran 95 and c++," *Computing in Science Engineering*, vol. 1, no. 2, pp. 63–70, Mar 1999.

[6] "GNU Fortran Compiler Manual."