Multilingual Interfaces for Parallel Coupling in Multiphysics and Multiscale Systems

Everest T. Ong[1], J. Walter Larson[2,3], Boyana Norris[2], Robert L. Jacob[2], Michael Tobis[4], and Michael Steder[4]

[1] Department of Atmospheric and Oceanic Science, University of Wisconsin, Madison, Wisconsin, USA
{eong,larson,norris,jacob}@mcsanl.gov,
{steder,tobis}@gmail.com
[2] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA
[3] ANU Supercomputer Facility, The Australian National University, Canberra ACT 0200, Australia
[4] Department of Geophysical Sciences, University of Chicago, Chicago, IL USA

Abstract. Multiphysics and multiscale simulation systems are emerging as a new grand challenge in computational science, largely because of increased computing power provided by the distributed-memory parallel programming model on commodity clusters. These systems often present a *parallel coupling problem* in their intercomponent data exchanges. Another potential problem in these coupled systems is language interoperability between their various constituent codes. In anticipation of combined parallel coupling/language interoperability challenges, we have created a set of interlanguage bindings for a successful parallel coupling library, the Model Coupling Toolkit. We describe the method used for automatically generating the bindings using the Babel language interoperability tool, and illustrate with short examples how MCT can be used from the C++ and Python languages. We report preliminary performance reports for the MCT interpolation benchmark. We conclude with a discussion of the significance of this work to the rapid prototyping of large parallel coupled systems.

1 Introduction

Multiphysics and multiscale models are emerging or are in active use in many fields, including meteorology and climate, space weather, combustion and reactive flow, fluid-structure interactions, material science, and hydrology. Multiphysics models portray complexity stemming from the mutual interactions among a system's many constituent subsystems. Multiscale models depict complex phenomena originating in interactions between a system's multiple prevalent spatiotemporal scales. In both multiphysics and multiscale models, these interactions are called *couplings*, and thus they are *coupled models*.

Coupled systems are very demanding computer applications and often require high-performance computing for their solutions. Most HPC platforms today are

Y. Shi et al. (Eds.): ICCS 2007, Part I, LNCS 4487, pp. 931–938, 2007.

distributed-memory multiprocessor clusters programmed with a message-passing programming model. This approach creates a new concomitant problem—*the parallel coupling problem* (PCP)[1]. Specifically, implementing couplings between message-passing models that solve their equations of evolution on distributed domains entails the description, transfer, and transformation of *distributed data*. Thus the PCP is an important emerging problem in computational science.

Typical coupled models have purpose-built *ad hoc* solutions to the PCP. For example, numerous coupling packages exist in the climate area alone, including Ocean-Atmosphere-Sea Ice-Surface (OASIS; [2]) coupler, Projet D'Assimilation par Logiciel Multi-methodes (PALM; http://cerfacs.fr/ palm), the Flexible Modeling System (FMS; http://gfdl.noaa.gov/fms/), and the Earth System Modeling Framework (ESMF; http://esmf.ucar.edu). Each of the cited examples has its own custom (and slightly different) solution to the same underlying problem (the PCP), but implemented with numerous domain-specific assumptions (e.g., mesh descriptions based on Arakawa grids from geophysical fluid dynamics). The PCP is sufficiently widespread across many distinct scientific fields that an application-neutral software solution is desirable.

MCT[1] is an open-source package that supports rapid development of parallel coupling interfaces between MPI-based parallel codes. MCT is in active use in numerous applications, most notably as coupling middleware for CCSM[2], and as a prototype coupling layer for the WRF[3]. MCT is highly scalable and performance portable, supporting both vector and commodity microprocessor architectures. Its Fortran-based API is naturally compatible with scientific applications, as Fortran remains the dominant programming language for science. MCT's programming model is minimally invasive, and scientific-programmer-friendly.

Here we report results of our work to broaden the applicability of the MCT programming model from its native Fortran API. Our motivations for extending MCT to other languages are 1) to broaden MCT's applicability from that of a Fortran-based toolkit to a callable framework that allows one to compose parallel coupling mechanisms in multiple programming languages, 2) to allow coupling of separately developed codes implemented in different languages, and 3) to leverage MCT's robust and efficient Fortran compute kernels and build coupling mechanisms in languages better suited to object-oriented programming (OOP). The strategy we have chosen is to create a set of multilingual bindings that can be installed on top of the MCT code base, rather than making them an inextricable part of MCT. This separation of concerns is in keeping with the MCT philosophy of offering a minimally invasive programming model, and will be addressed in some detail in Section 3. We believe that the outcome of the work reported here will enable rapid prototyping of parallel coupled models implemented in different languages, and should be of great interest to both current and would-be multiphysics and multiscale modeling teams.

[1] Model Coupling Toolkit, http://www.mcs.anl.gov/mct
[2] The Community Climate System Model, http://www.ccsm.ucar.edu/
[3] Weather Research and Forecasting Model, http://wrf-model.org

2 The Model Coupling Toolkit

MCT [1,3] is a Fortran toolkit that eases the programming of parallel couplings in MPI-based applications. MCT addresses the parallel data processing part of the PCP, while maximizing developer flexibility in choices regarding parallel coupled model architecture. MCT achieves this by providing a set of Fortran modules designed to emulate object oriented classes and methods[4] and a library of routines that perform parallel data transfer and transformation. These classes and methods amount to programming shortcuts that are used *à la carte* to create custom parallel couplings. MCT supports parallel coupling for both serial and parallel compositions—and combinations thereof—and also supports single and multiple executables.

MCT has nine classes for use as parallel coupling building blocks. Three datatypes constitute the MCT data model, encapsulating storage of multi-field integer- and real-valued data (AttrVect), the grids or spatial discretizations on which the data reside (GeneralGrid), and their associated domain decompositions (GlobalSegMap). MCT's data transfer facility's fundamental class is a lightweight component registry (MCTWorld) containing a directory of all components to be coupled and a process rank translation table supporting intercomponent messaging. One-way parallel data transfer message scheduling is encapsulated in the Router class, and this function for parallel data redistribution is embodied in the Rearranger. Data transformations supported directly by MCT are handled by three additional classes. The Accumulator is a set of time-integration registers for state and flux data. MCT supports regridding of data in terms of sparse linear transformations, with user-supplied transform coefficients stored in coordinate (COO) format by the SparseMatrix class. The SparseMatrixPlus class encapsulates matrix element storage and the necessary communications scheduling for parallel matrix-vector multiplication.

MCT has a library of routines that manipulate MCT datatypes to perform parallel coupling, supporting both blocking and non-blocking parallel data transfer and redistribution. MCT's transformation library routines support 1) parallel linear transforms used for intergrid interpolation; 2) time accumulation of flux and state data; 3) computation of spatial integrals required for flux conservation diagnoses; and 4) merging of outputs from multiple models for input to another model.

MCT is invoked through the *use* statement of Fortran90/95. One *uses* MCT modules to gain access to MCT datatype definitions and library interfaces, one *declares* variables of MCT datatypes to express distributed data to be exchanged and transformed, and one *invokes* MCT library routines to perform parallel data transfer and transformation. This is analogous to importing a class, instantiating an object which is a member of that class, and invoking the class methods associated with the object. A simple example of how MCT is used to construct

[4] The choice of Fortran as MCT's implementation language was driven by Fortran's continuing dominance as the language of choice in scientific programming. The developers of MCT implemented OOP features manually in Fortran, and the use of the terms class and method follow Decyk et al. [4].

a GlobalSegMap domain decomposition descriptor is shown in the code fragment below. More detailed examples MCT usage can be found in [1,3] and in the example codes bundled in the MCT source distribution, which can be downloaded from the MCT Web site.

```
  use m_GlobalSegMap, only : GlobalSegMap, GlobalSegMap_Init => init
  implicit none
  type(GlobalSegMap) :: AtmGSMap
  integer, dimension(:) :: starts, lengths
  integer :: myRoot, myComm, myCompID ! MPI communicator, root process
  integer :: myCompID                 ! MCT component ID
! initialize segment start and length arrays starts(:) and lengths(:)...
:
! Create and initialize MCT GlobalSegMap
  call GlobalSegMap_init(AtmGSMap, starts, lengths, myRoot, myComm, &
                         myCompID)
```

3 Construction of the Multi-lingual Interfaces for MCT

The MCT API is expressed using Fortran derived types and pointers, complicating considerably the challenge of interfacing MCT to other programming languages. This is due to the lack of a specific standard for array descriptors in Fortran90/95[5]. Thus, interfacing available contemporary Fortran to other languages remains notoriously difficult. One solution is to hard-code wrappers, which can be cumbersome, time-consuming, hard to maintain, and error-prone. The only automatic solution known to the authors is a vendor-by-vendor implementation of array descriptors such as CHASM[5].

Our multilingual interfaces are defined using the Scientific Interface Definition Language (SIDL). These interfaces are processed by a language interoperability tool called Babel[6], which leverages the vendor-specific array descriptors provided by CHASM. Babel currently supports interoperability between C, f77, Fortran90/95, C++, Java, and Python. Babel is used to generate glue code from an iterface description, thus avoiding modification of the original source code. This has the important advantage of separation of concerns; that is, we view language interoperability as a distinct problem from the algorithmics of parallel coupling. Thus MCT's scientist-friendly Fortran-based programming model is untouched, while langauage interoperability is available to those who need it. Our use of Babel enables us to create multilingual MCT bindings and distribute them as a separate package that references MCT. This approach superior to ESMF, whose fundamental types (e.g., ESMF_Array), are implemented in C for interfacing ESMF with possible future C applications, while important functions (such as Regrid) are implemented in Fortran. Language interoperability is *internal* to the ESMF software, not necessarily a user feature.

[5] This has been rectified by the BINDC specification in Fortran2003, but this standard is only now beginning to be implemented by compiler vendors.

As mentioned earlier, the SIDL interfaces and classes for MCT are processed by Babel to generate interlanguage "glue" code to bridge the caller/callee language gap. This glue code comprises *skeletons* in the callee's programming language (Fortran in the case of MCT), the *internal object representation* (IOR), which is implemented in C, and *stubs* that are generated in the caller programming language (e.g., C++ and Python). We have inserted calls to the MCT library in the Babel-generated implementation files that initially contain the empty function definitions from the SIDL interfaces (we will refer to them as *.IMPL files*). The IOR and stubs that provide the inter-language glue are generated by Babel automatically and require no modifications by the user. Working MCT bindings and example codes for both C++ and Python can be downloaded from the MCT Web Site. Included with the bindings are the Babel base classes and other core pieces of glue code that must be compiled against a pre-installed MCT, eliminating the need to install Babel (which can be nontrivial).

An example SIDL code block for the MCT AttrVect class and excerpt of the associated .IMPL file for the skeleton code are shown in Figure 1. The directed dotted grey arrows on the figure show the calling path by which an application written in some non-Fortran language accesses MCT by first calling a stub in the application's implementation language, which in turn calls the C IOR, which then calls the Fortran skeleton, and via it MCT.

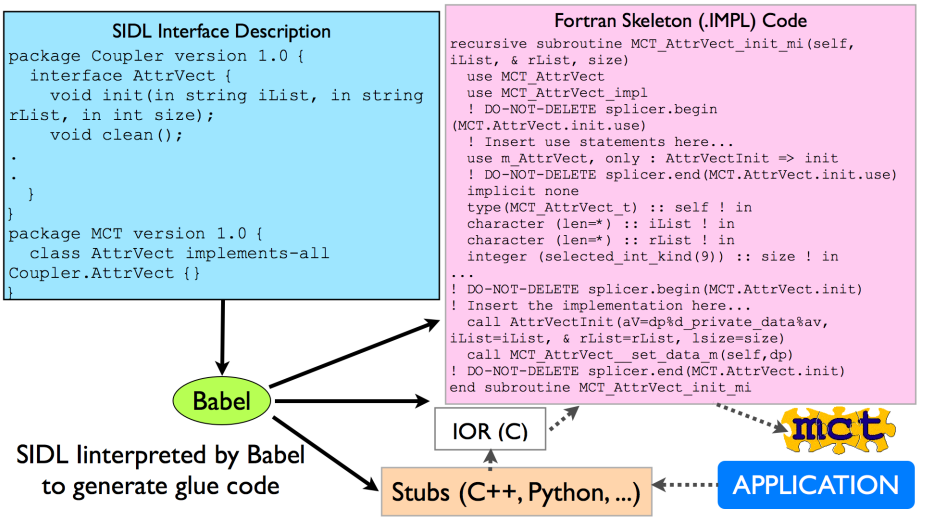

Fig. 1. Schematic for SIDL/Babel-based generation of MCT's multilingual interfaces and calling path from applications in other languages back to MCT

Our multilingual MCT bindings correspond to a *subset* of the MCT API because at this time Babel does not support the use of optional arguments, a Fortran feature that is widely used in MCT. For routines with only one or two

optional arguments, we have created static SIDL interfaces for each possible combination of optional arguments. For some of MCT's spatial integration and merging routines, which have in some cases four or more optional arguments, we decided to support only a subset of the possibilities, which will be expanded in the future as needed.

4 The MCT Multilingual Programming Model

MCT's native Fortran programming model described in section 2 consists of module usage, declaration of derived types, and invocation of library routines. The C++ and Python MCT programming models are analogous, but within each language's context. We will illustrate the differences in the respective programming models with a simple example taken from the example code in the MCT Fortran distribution and its C++ and Python counterparts. Below we show code excerpts from the MCT multilingual example applications and the original Fortran. The full example code demonstrates the kind of coupling found in climate models such as CCSM, with focus on atmosphere-ocean interactions via a third component called a coupler. The atmosphere, ocean, and coupler each execute on their own respective pool of MPI processes, making this example a parallel composition. The code fragments in this section are from the "coupler" parts of the respective example codes, and are for the initialization of an MCT GlobalSegMap domain decomposition descriptor.

4.1 C++

In C++, MCT exists as a *namespace* named MCT, and a collection of header files containing class declarations, in one-to-one correspondence with the set of Fortran modules in the original MCT source. The C++ MCT programming model differs from Fortran as follows: module use is replaced with inclusion of a Babel-generated header file; declaration is replaced with invocation of a no-argument constructor; and library routines are invoked through calls to Babel-generated C++ stubs that reference MCT functions. The code block below illustrates creation of an MCT GlobalSegMap in C++. The most subtle usage point is the use of SIDL arrays required by the MCT C++ interfaces.

```
#include "MCT_GlobalSegMap.hh"
...
  // Create SIDL arrays indexed (note starting index is 1 or
  // compatibility with Fortran
  int32_t dim1 = 1;
  int32_t lower1[1] = {1};
  sidl::array<int32_t> start =
      sidl::array<int32_t>::createRow(dim1, lower1, lower1);
  start.set(1, (myrank * localsize) + 1);
...
  // Create an MCT GlobalSegMap domain decomposition descriptor
  MCT::GlobalSegMap AtmGSMap = MCT::GlobalSegMap::_create();
  AtmGSMap.initd0(start, length, 0, comm, compid);
```

4.2 Python

In Python, MCT exists as a *package* named MCT. The Python MCT programming model differs from MCT's native Fortran as follows: module use is replaced with Python package import; declaration is replaced with invocation of a constructor; and library routines are invoked through calls to Babel-generated Python stubs. The code block below illustrates creation of an MCT GlobalSegMap in Python. Babel's support of SIDL arrays in Python is handled by the Numeric package, thus the creation of the arrays `start` and `length` as Numeric arrays.

```
import Numeric
from MCT import GlobalSegMap
...
   # Create start and length arrays--only the 'start' array shown here
   start = Numeric.zeros(2,Numeric.Int32)
   start[1] = (myrank*localsize)+1
...
   # Describe decomposition with MCT Global Seg Map
   AtmGSMap = GlobalSegMap.GlobalSegMap()
   AtmGSMap.initd0(start,length, 0,comm, compid)
```

5 Performance

Babel has been used successfully in various projects to generate interlanguage bindings with low performance overheads (e.g., see [7]). We evaluated the performance of the MCT C++ API for MCT's atmosphere-ocean parallel interpolation benchmark. The atmosphere-to-ocean grid operations are: 720 interpolation calls to regrid a bundle of two fields, and two sets of 720 interpolastion calls to regrid six fields. The atmospheric grid is the CCSM 3.0 T340 grid (512 latitudes by 1024 longitudes), and the ocean grid is the POP 0.1° grid (3600×2400 grid points). We ran the experiments on the Jazz cluster in the Laboratory Computing Resource Center at Argonne National Laboratory. Jazz is a 350-node cluster of 2.4 GHz Pentium Xeon processors connected by a Myrinet 2000 switch. The compilers used in this study were Absoft 9.0 Fortan and gcc 3.2.3. Timings (in seconds) for this benchamark for both MCT's native Fortran and for the Babel-generated C++ API are summarized in Table 1. The overhead decreases from 1.6% for small numbers of processors to less than one percent for larger numbers of processors, where the amount of work performed by MCT is enough to amortize the cost of executing the interlanguage glue code for each call to MCT.

Table 1. Timings (in seconds) of the MCT A2O Benchmark

Number of Processors	8	16	32
Native Fortran	1985.6	1084.6	556.9
C++ via Babel	2016.5	1085.1	559.6

6 Conclusions

We have created a set of multilingual bindings for the Model Coupling Toolkit. These bindings were created using the Babel language interoperability tool from a SIDL description of the MCT API. The resultant glue code is sufficiently robust to support proof-of-concept example applications, and impose relatively little performance overhead. The Babel-generated glue code and C++ and Python coupling example codes are now publicly available for download at the MCT Web site. The MCT programming model has been expanded beyond its native Fortran, making this robust and well-tested parallel coupling package available for use in coupling MPI-based parallel applications implemented in other languages. This is a first step towards our long-term goal of enabling fast prototyping of large, multilingual parallel coupled systems. The multilingual MCT bindings are also capable of supporting coupling of parallel applications implemeted in multiple programming language. We have employed them in a daring object-oriented Python re-implementation of the CCSM coupler (pyCPL), resulting in a Pythonic CCSM that supports Fortran models interacting via a Python coupler. Promising preliminary peformance results are a matter for further study.

Acknowledgements. This work was supported by the US Department of Energy's Scientific Discovery through Advanced Computing under Contract DE-AC02-06CH11357, by the National Science Foundation under award ATM-0121028, and by the Australian Partnership for Advanced Computing.

References

1. Larson, J., Jacob, R., Ong, E.: The Model Coupling Toolkit: A new Fortran90 toolkit for building multi-physics parallel coupled models. Int. J. High Perf. Comp. App. **19**(3) (2005) 277–292
2. Valcke, S., Caubel, A., Vogelsang, R., Declat, D.: Oasis3 ocean atmosphere sea ice soil user's guide. Technical Report TR/CMGC/04/68, CERFACS, Toulouse, France (2004)
3. Jacob, R., Larson, J., Ong, E.: M×N communication and parallel interpolation in CCSM3 using the Model Coupling Tookit. Int. J. High Perf. Comp. App. **19**(3) (2005) 293–308
4. Decyk, V.K., Norton, C.D., Syzmanski, B.K.: Expressing object-oriented concepts in Fortran90. ACM Fortran Forum **16**(1) (1997) 13–18
5. Rasmussen, C.E., Sottile, M.J., Shende, S.S., Malony, A.D.: Bridging the language gap in scientific computing: The CHASM approach. Concurrency and Computation: Practice and Experience **18**(2) (2006) 151–162
6. Dahlgren, T., Epperly, T., Kumfert, G.: Babel User's Guide. CASC, Lawrence Livermore National Laboratory. version 0.9.0 edn. (January 2004)
7. Kohn, S., Kumfert, G., Painter, J., Ribbens, C.: Divorcing language dependencies from a scientific software library. In: Proc. Tenth SIAM Conference on Parallel Processing in Scientific Computing, Portsmouth, VA (August 2001)