A Multilingual Programming Model for Coupled Systems

E. T. Ong[*] J. Walter Larson[†‡§] B. Norris[†‡] M. Tobis[¶] M. Steder[¶]
R. L. Jacob[†‡]

December 3, 2007

Abstract

Multiphysics and multiscale simulation systems share a common software requirement—infrastructure to implement data exchanges between their constituent parts—often called the *coupling problem*. On distributed-memory parallel platforms, the coupling problem is complicated by the need to describe, transfer, and transform distributed data—known as the *parallel coupling problem*. Parallel coupling is emerging as a new grand challenge in computational science as scientists attempt to build multiscale and multiphysics systems on parallel platforms. An additional coupling problem in these systems is language interoperability between their constituent codes. We have created a multilingual parallel coupling programming model based on a successful open-source parallel coupling library, the Model Coupling Toolkit (MCT). This programming model's capabilities reach beyond MCT's native Fortran implementation to include bindings for the C++ and Python programming languages. We describe the method used to generate the interlanguage bindings. This approach enables an object-based programming model for implementing parallel couplings in non-Fortran coupled systems and in systems with language heterogeneity. We describe the C++ and Python versions of the MCT programming model and provide short examples. We report preliminary performance results for the MCT interpolation benchmark. We describe a major Python application that uses the MCT Python bindings, a Python implementation of the control and coupling infrastructure for the Community Climate System Model. We conclude with a discussion of the significance of this work to productivity computing in multidisciplinary computational science.

[*]Dept. of Atmospheric and Oceanic Sciences, University of Wisconsin, Madison, WI, USA
[†]Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL, USA. mailto:larson@mcs.anl.gov
[‡]Computation Institute, University of Chicago, Chicago, IL, USA
[§]ANU Supercomputer Facility, The Australian National University, Canberra, Australia
[¶]Dept. of Geophysical Sciences, University of Chicago, Chicago, IL, USA

1 Introduction

Computational scientists are positioned to embrace the complexity of the systems they model. 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 ([1] and references therein). Multiphysics models simulate complexity due to mutual interactions among a system's many constituent subsystems. Multiscale models depict complex phenomena arising from interactions between a system's multiple prevalent spatiotemporal scales. In both multiphysics and multiscale models, these interactions are *couplings*, and thus these models can be more generally called *coupled models*.

Coupled systems are often computationally intensive and as such are natural high-performance computing (HPC) applications. Most HPC platforms today are distributed-memory multiprocessor clusters programmed with a message-passing programming (MPP) model using the Message Passing Interface (MPI) standard. This approach engenders a resultant 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, by association, the PCP is an important emerging problem in computational science, as it affects the ability to construct complex multiphysics and multiscale models [2].

Typical coupled models have purpose-built *ad hoc* solutions to the PCP. For example, numerous multiphysics coupling packages exist in the climate area alone, including the Ocean-Atmosphere-Sea Ice-Surface coupler (OASIS; [3]), Projet d'Assimilation par Logiciel Multi-methodes (PALM;[4, 5]), the Flexible Modeling System (FMS; [6]), and the Earth System Modeling Framework (ESMF; [7, 8]). Multiscale coupling is a common problem in numerical weather prediction (NWP). Many forecast models refine their spatial grids using embedded *nests* [9, 10] to achieve higher resolution over areas of interest, and these nests may be in mutual interaction. NWP models incorporating nesting in a distributed-memory parallel environment include the Penn State/NCAR MM5 model [11, 12] and the Weather Research and Forecasting model (WRF; [13, 14]). Each of these climate and NWP 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. We define such tools broadly as *parallel coupling infrastructure* (PCI).

Generic infrastructure for partial solutions to the coupling and parallel coupling includes mesh management and intermesh interpolation packages [15, 16, 17] and packages for parallel data transfer (a.k.a. the

M×N problem; [18] and references therein). A number of commercial packages provide coupling infrastructure for multiphysics simulation in single-processor environments [19, 20]. Two packages exist that combine these capabilities to provide comprehensive toolsets for the PCP—the Mesh-based Parallel Coupling Code Interface (MPCCI; [21]) and the Model Coupling Toolkit (MCT; [1, 22, 23]).

MCT is an application-neutral, open-source package that supports rapid development of parallel coupling interfaces between MPI-based parallel codes. MCT is in active use as coupling middleware in numerous applications, including the Community Climate System Model version 3.0 (CCSM; [24]), the Regional Ocean Modeling System version 3.0 (ROMS; [25]), and the Weather Research and Forecasting Model [26]. MCT is distributed with the CCSM 3.0 and ROMS 3.0 source and exists as a stand-alone parallel I/O coupling package for WRF [27]. 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—due to its large legacy code base—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 to 1) 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) allow coupling of separately developed codes implemented in different languages, and 3) 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 compatible with the MCT philosophy of offering a minimally invasive programming model for coupling. We believe that the impact of the work reported here is allowing computational scientists to create coupled systems with greater language heterogeneity and increasing the pace of development for these systems by opening access to Python for the implementation of system integration codes such as couplers and drivers. We hope that our work will inspire teams of prospective coupled modelers to develop new multiscale and multiphysics simulation codes.

2 Coupled Systems

A complete discussion of the PCP is beyond the scope of this paper. Here we define the concepts that are relevant to the current discussion. Further details can be found in reference [28].

2.1 The Coupling Problem

Abstractly, one can define a single stand-alone model as a system having a set of *state variables*, and these state variables are the solution of a set of equations of evolution on a spatiotemporal domain. The equations of evolution are solved for the state by using the present state, and possibly a set of *input variables* that are at a minimum defined on the domain's boundary. For example, in an atmosphere model, the state might comprise the velocity components u, v, ω, temperature T, and specific humidity q, and the inputs might comprise surface boundary conditions and forcing data such as black-body surface temperature. Additionally, a system might coumpute *output variables* on its domain's boundary. For example, in an atmosphere model the output variables might include precipitation and wind stress exerted on the Earth's surface.

Numerical solution of a model's state is typically performed on a discretized version of the domain, which is often called the model's *mesh* or *grid*. The grid is a Cartesian product of a spatial mesh and a time discretization. Thus, solutions for the state, input, and output variables exist only at points on the mesh within the spatiotemporal domain.

A coupled system comprises two or more mutually interacting subsystem models called *constituents*. Each constituent solves its equations of evolution for its state, using a set of input variables, and producing a set of output variables; as before, the state is computed on the disretized spatiotemporal domain, and inputs and outputs exist on the domain boundary. The input and output variables for a constituent constitute the data that is exchanged with other constituents in coupling.

Two constituents are *coupled* if the following conditions hold: they coincide in time; their computational domains overlap; and outputs from one constituent serve as inputs to the other, or one constituent's inputs are computed the other constituent's outputs through some scientifically relevant variable transformation (e.g., computation of a radiative flux from a black-body temperature using the Stefan-Boltzmann law).

As a coupled system evolves in time, *coupling events* occur in which constituents exchange data necessary to the coupling process. These events either can occur predictably in time, following a *schedule*, or are not predictible but rather *threshhold-triggered*, based on some condition satisfied by the constituents' states. In some cases, the set of coupling events falls into a repeatable periodic schedule called a *coupling cycle*.

Coupling between two constituents may be classified as explicit or implicit. *Implicit coupling* between constituents occurs if their spatiotemporal domains overlap and they share common state variables, thus requiring a simultaneous, self-consistent solution to their equations of evolution. *Explicit coupling* between constituents occurs if there is no such direct connection between their respective states. Core-edge coupling in fusion simulation is an example of implicit coupling [29], because both the core and edge codes solve Maxwell's equations for electromagnetic fields. An example of explicit coupling is the set of interactions present in a coupled climate system model. Implict coupling's requirement for a simultaneous self-consistent solution begs the question of whether such systems should even be implemented by composing individual model codes as this strategy can deleteriously affect performance. Explicit coupling is a natural application for code composition and tools such as MCT, and for this reason we will focus on it exclusively for the remainder of this article.

For explicit coupling between two constituents, the outputs from the source constituent may mapped to the inputs of the target constituent through a *coupling transformation*. The coupling transformation is a composition of a *mesh transformation* and a *field variable transformation*. Intergrid interpolation, often cast as a linear transformation, is a simple example of mesh transformation, but it can be in principle any mapping between two spatiotemporal discretizations and might be factored further into purely space and time components. Temporal mesh transformations in some cases may involve time averages of states and integrals of fluxes that are applied incrementally—through scaling by timestep size—in a target component. This strategy is employed by some climate models [30, 31]. The field variable transformation is application-specific, defined by the natural law relationships between a source constituent's outputs and a target constituent inputs. In general, the mesh and field variable transformations do not commute, creating source of coupling uncertainty.

In some cases, a constituent may receive the same field input from outputs of two other constituents, which will require *merging* this input data. Merging will have to be performed if three or more constituents coincide in time, their domains mutually intersect, and shared variables exist among the fields delivered from the source constituents to a target constituent, the simplest case being a *two-way merge* in which two constituents' outputs coincide to produce input for a third constituent. In a system with N constituents, up to an $N-1$-way merge is possible.

2.2 The Parallel Coupling Problem

The set of definitions and concepts presented in the previous section apply equally well to single processor and shared-memory parallel (SMP) or MPP multiprocessor systems. Distributed-memory parallelism creates additional challenges, and the definitions of Section 2.1 must be extended to accommodate the PCP that arises on MPP platforms. These additional challenges within the PCP can be summarized as *coupled model architecture* and *parallel data processing*. Coupled model architectural issues are primarily those of model mapping to processor sets (a.k.a. layout) and constituent execution scheduling. Parallel data processing comprises the set of operations necessary to accomplish interconstituent data interplay.

On distributed-memory platforms, the coupled-model developer faces a strategic decision regarding the mapping of the constituents to processors and the scheduling of their execution. Two main strategies exist—serial and parallel composition [32]. In a *serial composition*, all of the processors available are kept in a single pool, and the system's constituents each execute in turn using all of the processors available. In a *parallel composition* the set of available processors is divided into N disjoint groups called *cohorts*, and the constituents execute simultaneously, each on its own cohort. Serial composition has a simple conceptual design but can be a poor choice if the constituents do not have roughly the same parallel scalability; moreover, it restricts the model implementation to a single executable. Parallel composition offers the developer the option of sizing the cohorts based on their respective constituents' scalability; moreover, it enables the coupled model to be implemented as multiple executables. The chief disadvantage of parallel composition is that the concurrently executing constituents may be forced to wait for input data, causing cascading, hard-to-predict, and hard-to-control execution delays, which can complicate the coupled model's load balance. Such problems have been observed in the Community Climate System Model (CCSM) [33]. A third strategy, called *hybrid composition,* involves nesting one within the other to one or more levels (e.g., serial within parallel or vice versa). A fourth strategy, called *overlapping composition,* involves dividing the processor pool such that the constituents share some—but not all—of the processors in their respective cohorts; this approach may be useful in implementing implicit coupling, with the simultaneous self-consistent state solutions computed across the overlapping (shared) processors [29].

In a single global address space, description and transfer of coupling data are straightforward, and exchanges can be as simple as the use of global variables or the passing of arguments through function interfaces. In this situation, standards for describing field data and meshes for domain boundaries are sufficient. Distributed memory parallelism additionally requires *domain decomposition* descriptors for domain boundaries and their associated field data.

On MPP systems, the coupling transformation can have higher complexity than its SMP or single-processor counterpart. The mesh transformation can in principle require message-passing parallelism, and while this can in priniciple be true for the field variable transformation, it often is embarrassingly parallel. A further operation is often required—*data transfer*—either in the form of a parallel redistribution for a serial composition or as a parallel data transfer in the case of a parallel composition. The model developer decides where this additional factor in the composition of the coupling transformation is inserted in the coupling transformation, and again the order of operations will affect the result; it will have less of an impact on coupling uncertainties in the ordering of field variable and mesh transformations, its main effect appearing in roundoff-level differences caused by the reordering of arithmetic operations if computation is interleaved with the execution of the transfer. Additionally, the model developer has a choice in the *placement* of operations, that is, on which constituent's cohort the variable and mesh transformations should be performed—the source constituent, the destination constituent, on a subset of the union of their cohorts, or someplace else (i.e., delegated to another constituent—called a *coupler* [34]—with a separate set of processes).

3 The Model Coupling Toolkit

MCT [1, 22] is a software package that simplifies the programming of parallel coupling mechanisms in MPI-based applications. MCT provides a set of Fortran modules designed to emulate object-oriented classes and methods, and includes a library of routines that perform parallel data transfer and transformation. 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 in this section the use of the terms *class* and *method* follow Decyk et al. [35]. These classes and methods amount to programming shortcuts that are used *à la carte* to create custom parallel couplings. This approach allows significant architectural flexibility in coupled model design and implementation. For example, 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 (Table 1). Three datatypes constitute the MCT data model, encapsulating storage of multifield 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

Table 1: Functional Summary of MCT Classes

Functionality	MCT Class
Mesh Description	GeneralGrid
Field Data	AttrVect
Domain Decomposition	GlobalSegMap
Constituent PE Layouts	MCTWorld
One-Way Parallel Data Transfer Scheduling	Router
Two-Way Parallel Data Transpose Scheduling	Rearranger
Linear Transformation Parallel Linear Transformation	SparseMatrix SparseMatrixPlus
Time Integration Registers	Accumulator

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 nonblocking parallel data transfer and redistribution. MCT's transformation library routines support parallel linear transforms used for intergrid interpolation, time accumulation of flux and state data, computation of spatial integrals required for flux conservation diagnoses, and 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 that 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 a GlobalSegMap domain decomposition descriptor is shown in the code fragment below. More detailed examples MCT usage can be found in [1, 22] and in the example codes bundled in the MCT source distribution, which can be downloaded from the MCT Web site [23].

```
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)
```

4 Method

The MCT API is expressed by 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, which thwarts the use of packages used for f77/Python interoperability such as PyFort or f2py. The BINDC specification in Fortran2003 [36] is a possible solution to this problem, but this standard is only now beginning to be implemented by compiler vendors [37], and a further solution would be necessary to bridge between C and other programming languages. 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 available, widely portable, and automatic solution known to the authors is a vendor-by-vendor implementation of array descriptors such as CHASM [38].

Our multilingual interfaces are defined using the Scientific Interface Definition Language (SIDL). These interfaces are processed by a language interoperability tool called Babel[39], 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. We find this approach superior to and more capable than ESMF, whose fundamental types (e.g., ESMF_Array) are implemented in C for interfacing ESMF with possible future C applications, while important coupling functions (such as Regrid) are implemented in Fortran. Language interoperability is thus an *internal requirement* to the ESMF software,

and not necessarily a user feature.

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's 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 interlanguage 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.

Figure 1 shows an example SIDL code block for the MCT AttrVect class and excerpt of the associated .IMPL file for the skeleton code. 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.

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 as needed.

5 The MCT Multilingual Programming Model

MCT's native Fortran programming model described in Section 3 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. Below we show code excerpts from the MCT multilingual example applications corresponding to the original Fortran example in Section 3. 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.

5.1 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);
```

5.2 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

Table 2: 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

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)
```

6 Performance

Babel has been used successfully in various projects to generate interlanguage bindings with low performance overheads (e.g., see [40]). 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 2. The overhead decreases from 1.6% for small numbers of processors to less than 1% 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.

7 Case Study: PyCPL

The history of computing in the private sector shows a tendency to ever higher levels of abstraction. A significant recent advance has been the emergence of powerful, very high level languages, notably Python and Ruby. While emerging from the tradition of system administration scripting languages, these languages have advanced quite a distance from their roots. Their communities exhibit considerable sophistication and their libraries great breadth. Use of these languages has been shown to dramatically improve programmer productivity and code reuse [41].

Importing this strategy, and the so-called agile methodologies that have emerged along with it, into scientific programming promises the potential to offer large advantages and only modest disadvantages.

While the primary disadvantage of very high level languages as opposed to conventional compiled languages is performance, a large proportion of science codes as measured by line count is not in performance-critical code segments, being called once, or once per I/O operation, or once per time step. By leaving critical per spatial element computations in a compiled language, it is possible to replace much scientific code with a script without dramatically compromising performance [42].

This strategy applies directly to the sorts of model coupling supported by MCT. The most complex model based on MCT is CCSM [43]. We therefore put the Python MCT bindings to the test by rewriting the coupler layer for CCSM in Python. Here we provide a brief description of this work, but further details are available in reference [44].

CCSM is a coupled model that employs parallel computing. It is a multiple-load-image program because CCSM comprises five distinct components—atmosphere, ocean, sea-ice, land-surface, and coupler—each of which has a distinct executable image. Based on our definitions in Section 2 and CCSM's simultaneously executing constituents, the coupled model is a parallel composition.

In CCSM, all intercomponent data traffic is routed through the coupler. This hub-and-spokes architecture has been a part of CCSM since its initial version. The coupler in the current version of CCSM, called CPL6 [30], is the first version of the CCSM coupler to employ message-passing parallelism, allowing dedication of multiple processors to the coupling problem, in turn allowing CCSM to escape a coupling bottleneck as resources scale up. Such scalability was the foremost requirement for the design of CPL6.

The CPL6 hub application and high-level interfaces, together with CCSM's other components satisfy the definition of an *application framework*, according to Fayad et al. [45]. This fact has been leveraged in several interoperability experiments in which the IBIS [46] land-surface model has replaced CLM [47] and POP 2.0 with biogeochemistry has replaced CCSM's default ocean POP 1.4 [48], as have the HYCOM and MICOM

ocean models [49].

CPL6 has thus expanded substantially the scientific horizons of CCSM. Its existence has accelerated the inclusion of a biogeochemical cycle in CCSM. Adding biogeochemistry requires the ability to easily change the number of fields transferred between the models to allow for different groups of chemicals and to query the coupled system to determine which fields are active. This is made possible by the flexible data structures and methods provided by MCT and CPL6.

In spite of the flexibility of the current Fortran-based CPL6, at least four classes of investigation could benefit from an even more flexible coupling infrastructure within the CCSM. First, CCSM developers are prototyping a serial composition implementation of the system, a move motivated in part by a port of the model to the IBM BlueGene platform, but also to address observed delays caused by intercomponent data dependencies (see Section 2). Second, a range of coupling strategies is used to accelerate model convergence for studying very long paleoclimate phenomena, for instance, running for a century in a coupled atmosphere/ocean mode and then running a crude ocean for a millennium. Third, looser coupling may be enabled by a predictor-corrector strategy, as opposed to the current explicit/synchronous approach used in CCSM. Fourth, new physics—for example, a land ice sheet model—may need inclusion.

In each of these cases, the modifications in coupler logic are simple in principle but difficult and time consuming in practice. The bulk of the work done by the coupler logic, however, occurs at initialization time. Thus, only certain portions of the coupling code are performance-critical code when the simulation is ongoing, and these compute-intensive portions remain in Fortran but are accessed via Python.

One way to address such efforts is to view the CPL6 main program as a disposable entity, and its relatively small code base means one could in principle recode in Fortran a replacement to CPL6 that serves a specific scientific objective—an approach envisioned by MCT's creators. Alternatively, we can seek to generalize, orthogonalize, and encapsulate CPL6 methods in a more object-oriented strategy, facilitating their reuse in alternative modeling scenarios. This latter approach that drives our efforts toward higher-level coupler abstractions.

In exchange for this modest performance cost, scientists seeking to modify the code are presented with a far more readable and writable specification of the model's behavior. A significant fraction of advances in model-driven science emerges from modification of the models. By presenting a control layer capable of abstracting away the fussy details presented by compiled codes, we both demonstrate the possibility of a new approach to model construction and provide a platform for experimentation with alternative coupling schemes.

The version of the Python coupler we currently have implements some mathematics in a Python loop, which is causing some degradation in performance that may be optimized out in future. Still, we have a case directly comparable to a conventional CCSM run on 55 processors, where the performance is 65.4% of that of the pure Fortran CCSM. While we believe that performance can be improved substantially, this may be sufficient for many prototyping purposes. This performance impost is probably too high for production purposes, and optimization may be necessary by identifying and recoding compute-intensive portions of a prototype system in languages such as C or Fortran.

8 Conclusions

We have created a set of multilingual bindings for the Model Coupling Toolkit. These bindings were created by 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 toward 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 an object-oriented Python reimplementation of the CCSM coupler (pyCPL), resulting in a Pythonic CCSM that supports Fortran models interacting via a Python coupler. This large proof-of-concept application leads us to believe that low-level, elemental coupling tools deployable from a high-productivity programming language are a compelling alternative to other productivity computing approaches such as calling frameworks employing components (e.g., ESMF).

Acknowledgements:

This work was supported by the US Department of Energy's Scientific Discovery through Advanced Computing program under Contract DE-AC02-06CH11357 and by the National Science Foundation under award ATM-0121028. The ANU Supercomputer Facility is supported in part by the Australian Department of Education, Science, and Training through its National Collaborative Research Infrastructure Strategy.

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] S. L. Graham and M. Snir and C. A. Patterson (Eds).: Getting Up to Speed: The Future of Supercomputing. The National Academies Press, Washington, D. C. (2004)

[3] 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)

[4] Buis, S., Piacentini, A., Declat, D., Group, T.P.: PALM: A computational framework for assembling high-performance computing applications. Concurrency and Computation: Practice and Experience **18**(2) (2006) 231–245

[5] Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique: Projet d'Assimilation par Logiciel Multi-methodes (PALM) Web site. `http://cerfacs.fr/~palm` (2007)

[6] Geophysical Fluid Dynamics Laboratory: Flexible Modeling System Web site. `http://gfdl.noaa.gov/fms/` (2007)

[7] Hill, C., DeLuca, C., Balaji, V., Suarez, M., da Silva, A., The ESMF Joint Specification Team: The architecture of the earth system modeling framework. Computing in Science and Engineering **6** (2004) 18–28

[8] The ESMF Development Team: Earth System Modeling Framework (ESMF) Web site. `http://esmf.ucar.edu/` (2007)

[9] Smolarkiewicz, P.K., Grell, G.A.: A class of monotone interpolation schemes. Journal of Computational Physics **101**(3) 431–440

[10] Grell, G., Dudhia, J., Stauffer, D.: A description of the fifth-generation Penn State/NCAR mesoscale model MM5. NCAR Tech. Note 398, NCAR, Boulder, CO (1995)

[11] National Center for Atmospheric Resarch: MM5 Model Web site. `http://www.mmm.ucar.edu/mm5/` (2007)

[12] Michalakes, J.G.: The same-source parallel MM5. Journal of Scientific Computing **8**(1) (2000) 5–12

[13] Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., Wang, W.: The Weather Research and Forecast model: Software architecture and performance. In Zwieflhofer, W., Mozdzynski, G., eds.: Proceedings of the Eleventh ECMWF Workshop on the Use of High Performance Computing in Meteorology, Singapore, World Scientific (2005) 156–168

[14] WRF Development Team: Weather Resarch and Forecasting Model web site. `http://www.wrf-model.org/` (2007)

[15] Chesshire, G., Henshaw, W.D.: A scheme for conservative interpolation on overlapping grids. SIAM Journal of Scientific Computing **15**(4) (1994) 819–845

[16] Jones, P.W.: First and second-order conservative remapping schemes for grids in spherical coordinates. Monthly Weather Review **127** (1999) 2204–2210

[17] Brown, D.L., Henshaw, W.D., Quinlan, D.J.: Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In: Proc. SIAM Conference on Object-Oriented Methods for Scientific Computing. (1999)

[18] Bertrand, F., Bramley, R., Bernholdt, D.E., Kohl, J.A., Sussman, A., Larson, J.W., Damevski, K.B.: Data redistribution and remote method invocation for coupled components. Journal of Parallel and Distributed Computing **66**(7) (2006) 931–946

[19] Lethbridge, P.: Multiphysics analysis. The Industrial Physicist **10**(6) (2004) 26–29

[20] COMSOL, Incorporated: COMSOL Web site. `http://www.comsol.com` (2007)

[21] Joppich, W., Kurschner, M., the MpCCI Team: MpCCI - a tool for the simulation of coupled applications. Concurrency and Computation: Practice and Experience **18**(2) (2006) 183–192

[22] 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

[23] The MCT Development Team: Model Coupling Toolkit (MCT) Web site. `http://www.mcs.anl.gov/mct/` (2007)

[24] National Center for Atmospheric Resarch: Community climate system model web site. `http://www.ccsm.ucar.edu/` (2007)

[25] Ocean Modeling Group, Institute for Marine and Coastal Sciences, Rutgers University: Regoinal Ocean Modeling System (ROMS) Web site. `http://www.myroms.org/` (2007)

[26] Dan Schaffer: Coupling implementation of the WRF I/O API. `http://www-ad.fsl.noaa.gov/ac/schaffer/mct_wrf_io_api.html` (2004)

[27] Christopher Moore: Coupling atmosphere and ocean models for the study of hurricane-like vortex generation. `http://nctr-people.pmel.noaa.gov/cmoore/wrf-roms/index.html` (2007)

[28] Larson, J.W.: Some organising principles for coupling in multiphysics and multiscale models. Preprint ANL/MCS-P1414-0207, Mathematics and Computer Science Division, Argonne National Laboratory (2007)

[29] Cary, J.R., Candy, J., Cohen, R.H., Krasheninnikov, S., McCune, D.C., Estep, D.J., Larson, J., Malony, A.D., Worley, P.H., Carlsson, J.A., Hakim, A.H., Hamill, P., Kruger, S., Muzsala, S., Pletzer, A., Shasharina, S., Wade-Stein, D., Wang, N., McInnes, L., Wildey, T., Casper, T., Diachin, L., Epperly, T., Rognlien, T.D., Fahey, M.R., Kuehn, J.A., Morris, A., Shende, S., Feibush, E., Hammett, G.W., Indireshkumar, K., Ludescher, C., Randerson, L., Stotler, D., Pigarov, A.Y., Bonoli, P., Chang, C.S., D'Ippolito, D.A., Colella, P., Keyes, D.E., Bramley, R., Myra, J.R.: Introducing FACETS, the framework application for core-edge transport simulations. Journal of Physics Conference Series **78** (2007) 0120086–0120089

[30] Craig, A.P., Kaufmann, B., Jacob, R., Bettge, T., Larson, J., Ong, E., Ding, C., He, H.: CPL6: The new extensible high-performance parallel coupler for the Community Climate System Model. Int. J. High Perf. Comp. App. **19**(3) (2005) 309–327

[31] Bettge, T., Craig, A., James, R., Wayland, V., Strand, G.: The DOE Parallel Climate Model (PCM): The Computational Highway and Backroads. In Alexandrov, V.N., Dongarra, J.J., Tan, C.J.K., eds.: Proc. International Conference on Computational Science (ICCS) 2001. Volume 2073 of Lecture Notes in Computer Science., Berlin, Springer-Verlag (2001) 148–156

[32] Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison Wesley, Reading, Massachusetts (1995)

[33] Carr, G. Private communication (2005)

[34] Bryan, F.O., Kauffman, B.G., Large, W.G., Gent, P.R.: The NCAR CSM flux coupler. NCAR Tech. Note 424, NCAR, Boulder, CO (1996)

[35] Decyk, V.K., Norton, C.D., Syzmanski, B.K.: Expressing object-oriented concepts in Fortran90. ACM Fortran Forum **16**(1) (1997) 13–18

[36] ISO/IEC Joint Technical Committee 1, Subcommittee 22, Working Group 5: Information Technology–Programming Languages–Fortran–Part 1: Base Language. Standard Definition ISO/IEC 1539-1:2004, International Standardization Organization, Geneva, Switzerland (2004)

[37] Chivers, I.D., Sleightholme, J.: Compiler support for the Fortran 2003 standard. ACM Fortran Forum **26**(2) (2007) 25–27

[38] 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

[39] Dahlgren, T., Epperly, T., Kumfert, G.: Babel User's Guide. CASC, Lawrence Livermore National Laboratory. version 0.9.0 edn. (January 2004)

[40] 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)

[41] Ousterhout, J.: Scripting: Higher-level programming for the 21st century. IEEE Computer **31**(3) 31–41

[42] Langtangen, H.P.: Python Scripting for Computational Science. Springer, Berlin (2005)

[43] Collins, W.D., Bitz, C.M., Blackmon, M.L., Bonan, G.B.., Bretherton, C.S., Carton, J.A., Chang, P., Doney, S.C., Hack, J.J., Henderson, T.B., Kiehl, J.T., Large, W.G., McKenna, D.S., Santer, B.D., , Smith, R.D.: The community climate system model: CCSM3. Journal of Climate **19**(11) (2006) 2122–2143

[44] Tobis, M., Steder, M., Jacob, R.L., Pierrehumbert, R.T., Larson, J.W., Ong, E.T.: PyMCT and PyCPL: Refactoring the Community Climate System Model Using Python. Preprint ANL/MCS-P1415-0207 (2007)

[45] Fayad, M.E., Johnson, R.E., Schmidt, D.C.: Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley and Sons, New York (1999)

[46] Foley, J.A.: An integrated biosphere model of land surface processes, terrestial carbon balance, and vegetation dynamics. Global Biogeochemical Cycles **10**(4) (1996) 603–628

[47] Mirin, A. Private communication (2006)

[48] Maltrud, M. Private communication (2006)

[49] Norton, N. Private communication (2006)

Figures

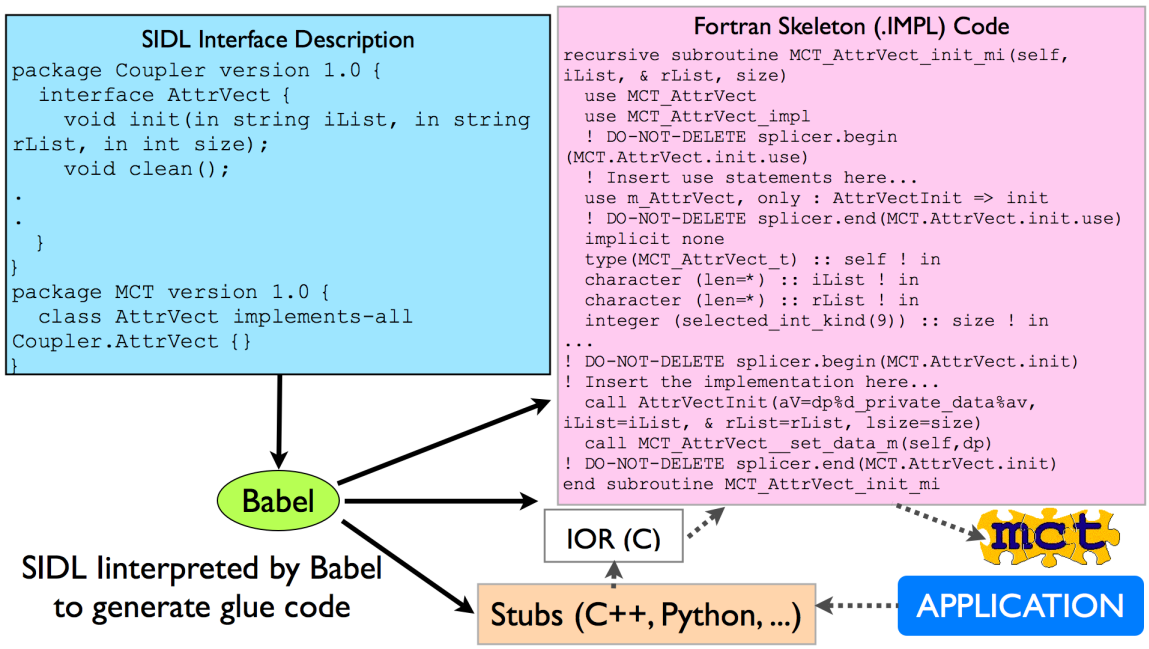

Figure 1: Schematic for SIDL/Babel-based generation of MCT's multilingual interfaces and calling path from applications in other languages back to MCT.