PyMCT and PyCPL: Refactoring the Community Climate System Model

M. Tobis[*] M. Steder[*] R. L. Jacob[†‡]

R. T. Pierrehumbert[*] J. Walter Larson[§¶‖‡] E. T. Ong[‖]

February 13, 2007

Abstract

Coupled climate models are multiphysics models comprising multiple separately developed codes that are combined into a single physical system. This composition of codes is amenable to a scripting solution, and Python is a language that offers many desirable properties for this task. We have prototyped a version of the Community Climate System Model (CCSM) with coupling infrastructure written in Python. Our objective was to improve dramatically CCSM's already flexible coupling infrastructure to enable research uses of the model not currently supported. Here we report the progress in the first steps in this effort: the construction of Python bindings for the Model Coupling Toolkit, a key piece of third-party coupling middleware used in CCSM, and a Python-based CCSM coupler application. We find that the choice of Python over the original Fortran implementation in the coupler imposes minimal visible performance impact to the overall coupled system. We believe our results augur well for the use of Python in the top-level coupling and organisation of large parallel multiphysics and multiscale applications.

[*]Dept. of Geophysical Sciences, University of Chicago, Chicago, IL, USA. mailto: tobis@gmail.com

[†]Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, Illinois, USA

[‡]Computation Institute, University of Chicago, Chicago, IL, USA

[§]ANU Supercomputer Facility, The Australian National University, Canberra, Australia

[¶]Australian Partnership for Advanced Computing (APAC)

[‖]Dept. of Atmospheric and Oceanic Sciences, University of Wisconsin, Madison, WI, USA

Contents

1 Introduction **2**

2 The Community Climate System Model Software Stack **3**

3 Python Coupling Infrastructure for CCSM **5**
3.1 Performance Issues 5
3.2 Design 6
3.3 Proof-of-Concept 7

4 Future Work **8**

1 Introduction

Coupled climate models are now the state-of-the-art simulation tool for studies of climate variability, climate sensitivity, climate change, and past climates. These models typically comprise a number of computationally intensive components, including an atmospheric general circulation model (GCM), an ocean GCM, a dynamic-thermodynamic sea ice model, and a land-surface model. Some coupled models contain an additional software entity that mediates exchanges between the subsystems. This component is called a *flux coupler*, or simply a *coupler*. Many of these models contain other refinements such as a river transport model that routes freshwater runoff from the land surface for input into the ocean. Future coupled climate models will evolve into more complete earth system models, adding in effects such as the carbon cycle, interactive vegetation, and continental-scale ice sheets.

Though the first attempt at coupled climate modeling took place nearly forty years ago[9], only recently has the coupled climate model become a practicable tool within the individual researcher's reach. The key enabling technology has been parallel computing, and in particular *distributed-memory parallelism* (a.k.a. message-passing) on commodity clusters.

Message-passing introduces a new obstacle to be surmounted–the *parallel coupling problem* (PCP)[8]. That is, given N multiple, separately-developed message-passing parallel models, combine them into an efficient parallel coupled model. One strategy for addressing the PCP is to use a library called the Model Coupling Toolkit (MCT)[8, 7].

We report preliminary results in our effort to build Python interfaces for MCT (pyMCT), and then exploit these interfaces to recreate CCSM's coupling infrastructure in Python. Our purpose in pursuing this strategy is

to explore the viability of run-time Python as a programmer productivity tool in high performance computing.

The application of CCSM that draws the most attention is as a climate prognostication tool. Used this way, no major changes are required to the code base; the principal challenge in such usage is simply the marshalling and management of very large computational resources, sufficient for running an ensemble of simulations of several centuries of the evolution of the climate system.

However, such efforts depend in turn on more speculative work with model codes, whether in theoretical studies with speculative scenarios, or in replicating geological evidence of paleoclimates. Such usages of models frequently require substantial rewrites by individual researchers, and it is here that the opacity and brittleness of the released models comes back to restrict further progress. Time spent by researchers in coding and debugging is time not spent thinking about the science. Even worse, time *not* spent by reserachers testing and verifying their modifications can lead to incorrect conclusions.

The expected benefit of this work—and our primary motivation—is a considerably more flexible CCSM that will facilitate a wide variety of model studies that would require considerably more programmer time and effort using the current version of CCSM.

2 The Community Climate System Model Software Stack

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. In terms of coupled model architecture such a design, with multiple parallel entities scheduled concurrently is called a *parallel composition*[5].

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[1], 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 CCSM software stack is shown in Figure 1. At the lowest level lies parallel computing middleware in the form of MPI and OpenMP. Immedi-

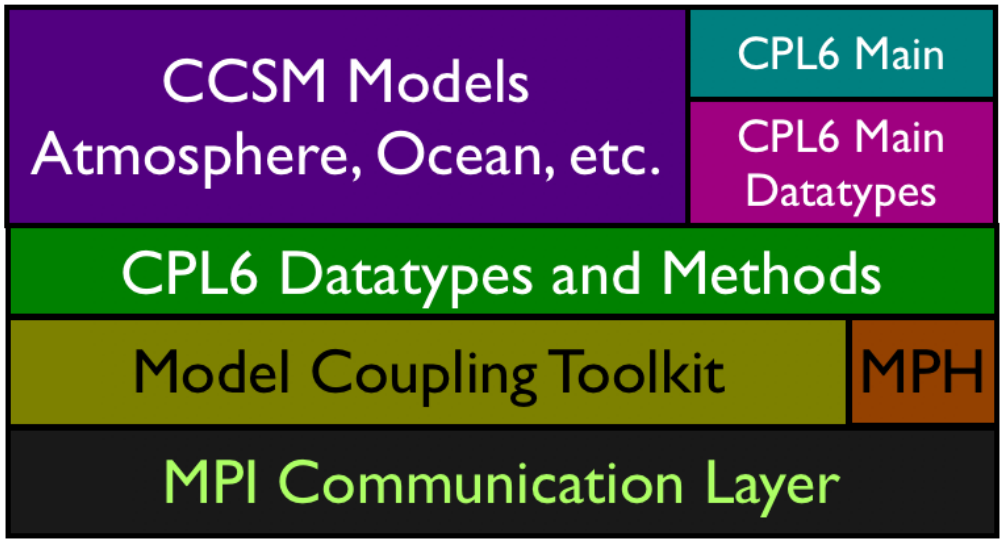

Figure 1: The CCSM Coupler CPL6 software stack.

ately above this layer lie two pieces of coupling middleware, Multi-Process Handshaking utilities (MPH) and the Model Coupling Toolkit (MCT).

MPH is a set of Fortran utilities that perform the type of communicator management and communicator splitting required by MPI-based parallel coupled models[6]. In the current CCSM implementation, its role is limited to model instantiation, where various instances of MPH across multi-processor executables collaborate to build an agreed-upon set of MPI communicators.

MCT is a Fortran toolkit and library that provides a powerful set of programming short-cuts that support the implementation of parallel coupling mechanisms[8, 7].

In the layer immediately above MCT and MPH lies the lowest level of CPL6, software, namely its internal datatypes and methods. The top layer in Figure 1 comprises the CPL6 model interface datatypes that are used by the components of CCSM and the main coupler application.

The CPL6 hub application and high-level interfaces, together with CCSM's other components satisfy satisfy the definition of an *application framework* according to Fayad *et al.* [3]. This fact has been leveraged in several interoperability experiments in which the IBIS[4] land-surface model has replaced CLM (Art Mirin, private communication), and POP 2.0 with biogeochemistry has replaced CCSM's default ocean POP 1.4 (Mat Multrud, private communication) as have the HYCOM and MICOM ocean models (Nancy Norton, private communication).

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.

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 re-code in Fortran a replacement to CPL6 that serves a specific scientific objective. Alternatively, we can seek to generalize, orthogonalize and encapsulate CPL6 methods in a more object-oriented strategy, facilitating their reuse in alternative modeling scenarios. It is this latter approach that drives our efforts.

3 Python Coupling Infrastructure for CCSM

3.1 Performance Issues

As presently constructed, computationally intensive codes in very high level languages such as Python are typically one to two orders of magnitude slower in performance than comparable codes in conventional compiled languages. It is commonly concluded that, whatever its productivity advantages, Python is not an appropriate run-time component for high performance computing applications. There are three aspects to our strategy that overcome this objection.

The first, well-known to the Python community, is language interoperability. When Python is used in performance-critical applications, the critical code blocks are commonly written in a high performance compiled language such as C. In the present case, much of the coupler's work is in fact delegated to existing software libraries. A larger fraction can be so delegated should it prove necessary.

A more unusual aspect of the strategy depends specifically on the design of MCT. MCT is intended as a totally and almost linearly scalable coupling layer. Thus, to the extent that the coupler actually impacts performance, more processors can be delegated to the coupler process. In practical implementations of CCSM, the coupler takes only a very smalll fraction of the total computational resources, so that there is substantial room increasing this fraction without dramatically impacting the cost of a run.

Finally, such computations that the coupler performs in CCSM are not

on the critical performance path. The CCSM coupler is idle much of the time, awaiting completion of the more computationally demanding physical calculations.

All of these factors work in favor of our design, so that the appealing but intuitively implausible idea of Python at runtime in a high performance model can be achieved.

3.2 Design

Our first objective was to demonstrate that a scripting language could be a part of the runtime of a high performance model. To achieve this goal, we sought to achieve the minimum disruption of the CCSM.

To understand our strategy, it is sufficient to refer to Figure 1. The software stack llustrated there equally described the physics component and the coupler component of the official CCSM release. A Python coupler must simply provide the same MPI interface to the physical components as does the CCSM coupler. Because we had already completed the goal of providing Babel couplings to MCT in the context of another project, and because Babel provides interlanguage interoperability among a set of languages which includes Python, this was the natural place to put the interlanguage boundary.

This presented us with the requirement to reimplement CPL6 in Python, rather than providing Python bindings for CPL6. Because our conclusion is that CPL6 is not a sufficiently general design, we hope eventually to draw upon the incremental *agile* development strategies which Python is known, in order to provide a more flexible, more robust, and more elegant implementation of the standard coupler. Nevertheless, the smallest hurdle, to ensure that the Python coupler is feasible, is to slavishly replicate the Fortran source in an essentially line-for-line translation.

After looking at what MPH is used for in CCSM we estimated it could be quickly replaced with an equivalent piece of Python in about 200 lines of code. It was necessary to design a subsystem that could allocate communicators to processor groups that could successfully interoperate between Fortran 90 and Python. To support this new initialization we also needed to add a new function to the Fortran version of CPL6 library that supported this simple CCSM-only initialization. The only Fortran code modified is within the CPL6 library and the component source code remains untouched.

As development progressed, features and MPI functions were needed which were not supported by existing Python MPI strategies. To meet the changing demands of the project we developed our own set of Python MPI bindings, based on the MyMPI[13] package. Tentatively called MMPI[10],

this work will be reported in detail elsewhere.

In addition to MMPI, the task of communicator splitting served by MPH was also replicated in Python. It was necessary to replace MPH in the component models to enforce compatibility with our communicator tools, which requires re-linking of the component models but imposes no changes to their source code whatsoever. While even this requirement disappoints us somewhat, we encourage comparison with alternative framework strategies which require substantial rewrites of all physics components.

MCT offers a user-friendly API for programming parallel couplings. This API, however is expressed using Fortran derived types and Fortran 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 or derived types in the Fortran90 standard, and the nonexistence of even a standard API for querying them.

Interfacing Fortran90 to other languages is consequently notoriously difficult, and the only solution is a vendor-by-vendor implementation such as CHASM[11]. Our Python interfaces were crafted by leveraging CHASM via software called Babel[2], which currently supports the following languages: C, Fortran (through 95), C++, Java, and Python. Usage of Babel requires the developer wishing to export library code written in a particular language for use by other languages to describe the codes interfaces using the *Scientific Interface Definition Language* (SIDL). This interface description is porcessed by Babel to provide run-time bindings in C, along with callable interfaces in the target languages. Babel-generated MCT interlanguage bindings can be downloaded from the MCT Web Site[12].

Armed with a working version of pyMCT, re-programming of the Python version of CPL6 classes was straightforward. Just as CPL6 wraps MCT objects and methods in its own derived types, PyCPL wraps PyMCT types with Python classes that provide the necessary coupler logic and behavior. In most cases the Python classes are just translations of their Fortran counterparts. PyCPL as it stands is merely a close translation of CPL6. We plan to provide extensions and improvements in the future to increase the functionality of the coupling layer.

We thus succeeded in our goal of providing a completely transparent replacement coupler design. We emphasize that no changes whatsoever are required to the component model source.

3.3 Proof-of-Concept

PyCCSM remains an incomplete implementation of CCSM, but key elements of it are complete and undergoing evaluation. For example the parallel data

transfer system is exchanging the right number of messages of the right length with their data arranged in the right order. For a multiple executable multiple processor per executable architecture that requires exchanges of communication contracts at instantiation, this is no small feat. There is a significant amount of run-time computation in the coupler, which was manually translated. This last, anticipated hurdle remains. Nevertheless we are now in a position to claim that the conceptual basis for the design is sound.

We have tested our Python-based CCSM coupling code stack on a benchmark application: coupling of CCSM *data* models. Data models exsist for each of the atmosphere, ocean, land-surface and sea-ice components of CCSM, each serving as a proxy for a real or *live* model. Data models provide to the coupler all of the output fields expected from the component it portrays, and accept from the coupler a default set of input fields. This allows the coupler to exercise all of its functionality. Taken as such, the data models form an integration test for the coupler.

Preliminary results have only been obtained on small clusters but are thus far encouraging. The slowdown of the coupler code is on the order of a factor of 2.5, but this is in a system where the coupler code is on the critical computational path. In practice, most or all of this slowdown will be masked by the time that the physics components spend in computations. We anticipate a very small performance penalty in the completed system.

4 Future Work

Our immediate development path is to complete pyCCSM, a Python-based version of CCSM with live models. We are confident this goal will be achieved in the near future and look forward to reporting how pyCPL performs in this context. If successful, it will mark an important step forward in applying productivity computing to a grand challenge computational science application.

Our ultimate objective remains scientific productivity rather than programming productivity. As programmers, we are in a service relationship toward physical scientists. It is their needs, rather than our interests as software professionals, that should drive our work. Toward that end, our objective is to embed PyCCSM in various run control environments, combining the model control with the ensemble control in a single scripting environment. The goal of a clear-cut control layer in the model is ultimately to provide ever higher levels of abstraction, allowing for statistical analysis, objective tuning, and automated selection not only of parameters but of model instantiations.

Given one definition of a computational framework[3], the work presented here marks the prototyping of a major scientific software application framework in Python. Given Python's general programmer-friendliness, it is clear that in this context one will not only be able to leverage the composition character of software frameworks, but also enjoy a free hand to modify both the components and the framework itself. We believe this demonstrates that Python can provide formidable competitors to conventional software frameworks in the scientific productivity computing arena.

Acknowledgements: This work is primarily a product of the University of Chicago's Climate Systems Center, funded by the United States National Science Foundation (NSF) Information Technology Research (ITR) program under award ATM-0121028. The work on the SIDL description for MCT was supported by the United States Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) [14] initiative, through the Center for Component Technology for Terascale Simulation Software (CCTTSS). Argonne National Laboratory is operated by UChicago Argonne LLC for the United States Department of Energy under contract DE-AC02-06CH11357.

The authors thank Dr. Margaret Kahn of the ANU Supercomputer Facility for organising the Python workshop that served as a catalyst for writing this article. Many thanks are also due for the kind assistance of Tom Epperly, Susan Coughlin, Boyana Norris, Matt Knepley and Craig Rasmussen.

References

[1] Anthony P. Craig, Brian Kaufmann, Robert Jacob, Tom Bettge, Jay Larson, Everest Ong, Chris Ding, and Helen He. cpl6: The new extensible high-performance parallel coupler for the community climate system model. *Int. J. High Perf. Comp. App.*, 19(3):309–327, 2005.

[2] Tamara Dahlgren, Thomas Epperly, and Gary Kumfert. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.9.0 edition, January 2004.

[3] Mohamed E. Fayad, Ralph E. Johnson, and Douglas C. Schmidt. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley and Sons, New York, 1999.

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

[5] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley, Reading, Massachusetts, 1995.

[6] Helen He and Chris Ding. Coupling multi-component models by mph on distributed memory computer architectures. *Int. J. High Perf. Comp. App.*, 19(3):329–240, 2005.

[7] Robert Jacob, Jay Larson, and Everest Ong. M×n communication and parallel interpolation in ccsm3 using the model coupling tookit. *Int. J. High Perf. Comp. App.*, 19(3):293–308, 2005.

[8] Jay Larson, Robert Jacob, and Everest Ong. The model coupling toolkit: A new fortran90 toolkit for building multi-physics parallel coupled models. *Int. J. High Perf. Comp. App.*, 19(3):277–292, 2005.

[9] Syukuro Manabe and Kirk Bryan. Climate calculations with a combined ocean-atmosphere model. *Journal of the Atmospheric Sciences*, 26(4):786–789, 1969.

[10] Michael Steder. Mmpi home page. `http://www.penzilla.net/mmpi/`, 2006.

[11] Craig E. Rasmussen, Matt J. Sottile, Sameer S. Shende, and Allen D. Malony. Bridging the language gap in scientific computing: The CHASM approach. *Concurrency and Computation: Practice and Experience*, 18(2):151–162, 2006.

[12] The MCT Development Team. Model coupling toolkit (mct) web site. `http://www.mcs.anl.gov/mct/`, 2006.

[13] Timothy H. Kaiser. MYMPI web site. `http://peloton.sdsc.edu/~tkaiser/mympi/`, 2006.

[14] U. S. Dept. of Energy. SciDAC Initiative homepage. `http://www.osti.gov/scidac/`, 2003.