PyCCSM: Prototyping a python-based community climate system model

Michael Tobis[1] Michael Steder[2] J. Walter Larson[3]
Raymond T. Pierrehumbert[4] Robert L. Jacob[5]
Everest T. Ong[6]

(Received 31 August 2006; revised 8 January 2010)

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 Python coupling and control infrastructure for version 3.0 of the Community Climate System Model (CCSM3). Our objective was to improve dramatically CCSM3's already flexible coupling facilities to enable research uses of the model not currently supported. 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 CCSM3, and a Python-based CCSM3 coupler (PyPCL) application. We report preliminary performance results for this new system, which we call PyCCSM. We find PyCCSM is significantly slower than its Fortran counterpart,

http://anziamj.austms.org.au/ojs/index.php/ANZIAMJ/article/view/121 gives this article, © Austral. Mathematical Soc. 2010. Published January 11, 2010. ISSN 1446-8735. (Print two pages per sheet of paper.)

and explain how PyPCL's performance may be improved to support production runs. 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.

Contents

1 Introduction

Coupled climate modelling is the simulation of the Earth's climate system using multiple interacting models of the atmosphere, oceans, biosphere, and cryosphere. The physical interactions (for example, energy and water flux exchanges) between ocean and atmosphere, and so forth—the *couplings*—are frequently implemented by using an additional software entity called a *flux coupler*, or simply a *coupler*.

Only recently has the coupled climate model become a practicable tool within the individual researcher's reach, enabled chiefly by *distributed memory parallelism* on commodity clusters using the Message Passing Interface (MPI[1]) library. Message passing introduces a new obstacle called the *parallel coupling problem* (PCP) [9]: Given N multiple, separately developed, message passing parallel models, combine them into an efficient parallel coupled model. The PCP is the central computational science challenge faced by developers of coupled climate, multiphysics, and multiscale models.

In our past work, we developed significant parallel coupling infrastructure, including the application independent coupling middelware package the Model Coupling Toolkit (MCT[2]) [8, 10] and the Community Climate System Model's[3] coupler CPL6 [2]. CPL6 is built on the CPL6 *toolkit* [2], which offers application specific classes and methods that ease significantly the construction of flux couplers. MCT, the CPL6 toolkit, and the CPL6 coupler are implemented in the Fortran programming language.[4]

CSMs are evolving into more complete *Earth system models*, including effects such as the carbon cycle, interactive vegetation, and continental scale ice sheets. This increase in system complexity has a concomitant increase in software complexity. This complexity must be tamed, and a high level scripting language such as Python is a natural choice.

In the present work, we describe our reimplementation of the CPL6 toolkit and the CPL6 coupler in the Python programming language. Our purpose in this effort is to

[1]`http://www.mcs.anl.gov/mpi/`

[2]`http://www.mcs.anl.gov/mct`

[3]We refer to version 3.0 of CCSM throughout this paper unless noted otherwise.

[4]Fortran90/95, in which MCT and CPL6 are written, does not offer explicit support for object oriented programming (OOP). We use the terms *class* and *method* following Decyk et al. [3]; a class is something implemented in a Fortran derived type, and a method is a subroutine or function that services a particular class. Fortran2003 offers the CLASS construct with OOP support, but this feature was not widely supported by compilers when this article was written.

1. explore the viability of run-time Python as a programmer productivity tool in high-performance computing;
2. extend CCSM's functionality by offering a Python based programming model for implementing different coupling strategies; and
3. enable higher levels of data object abstraction that will enable embedding of CCSM in more complex systems.

Examples of different coupling strategies include changing the number of components, changing the coupling mechanism from explicit to implicit coupling, and coupling CCSM to a mesoscale model to perform regional climate downscaling. Examples of embedded CCSM applications include ensemble run coordination, data assimilation, and objective parameter tuning.

The key results reported here are the design strategy and significant body of Python code implemented to realise this vision, including Python bindings for MCT, a new set of Python MPI bindings capable of supporting multi-executable applications, a Python implementation of CCSM's CPL6 coupling library, and a Python implementation of the CCSM coupler. Taken together, these form PyCCSM. We report preliminary PyCCSM performance for a significant system integration test case. The expected benefit of this work is a considerably more flexible system that will enable climate researchers to undertake a wide variety of model studies that would require resources beyond their reach using the current version of CCSM.

2 The community climate system model

CCSM is an open source, coupled climate model used by an international community of hundreds of scientists. Users employ CCSM in numerical experiments to study climate variability, climate sensitivity, climate change, and paleoclimates. CCSM is one of the models used in studies summarised in the Intergovernmental Panel on Climate Change (IPCC[5]) Assessment Reports [11].

CCSM is a *multiple load-image parallel program.* CCSM has multiple load images because its components (atmosphere, ocean, sea-ice, land-surface, and coupler) are each implemented in a separate executable image. All the components execute concurrently in a *parallel composition* [6] and exchange data to implement coupling; a process composition strategy that has been used in all released versions of CCSM to date. In CCSM, all intercomponent data traffic is routed through the coupler. This *hub-and-spokes* architecture has been used in all versions of CCSM to date [2]; the coupler is the 'hub' and the client models—atmosphere, ocean, sea-ice, and land models—are the 'spokes.' CPL6 is the first version of the CCSM coupler to employ message passing parallelism. CPL6 scales well enough to avoid the potential bottleneck inherent in a hub-and-spokes design [2].

CCSM has a layered design (Figure 1). The lowest layer contains parallel computing middleware, specifically MPI. Immediately above this layer lie two pieces of coupling middleware, Multi-Process Handshaking (MPH) utilities and the Model Coupling Toolkit (MCT). MPH performs MPI communicator management and communicator splitting for MPI-based parallel coupled models. Instances of MPH across multiprocessor executables collaborate to build an agreed upon set of MPI communicators, one for each component model. MCT is parallel coupling middleware that eases the programming of parallel couplings in MPI-based applications. MCT addresses the parallel data processing part of the PCP [9], maximising developer flexibility in choices regarding parallel coupled model architecture. MCT provides classes and methods and a library of routines to effect distributed data description and parallel data transfer and transformation. The CPL6 toolkit provides classes that extend MCT to a higher level of data structure complexity. In the layer immediately above MCT and MPH lie the CPL6 internal classes and methods. The top layer in Figure 1 comprises the CPL6 model interface

[5]http://ipcc.ch

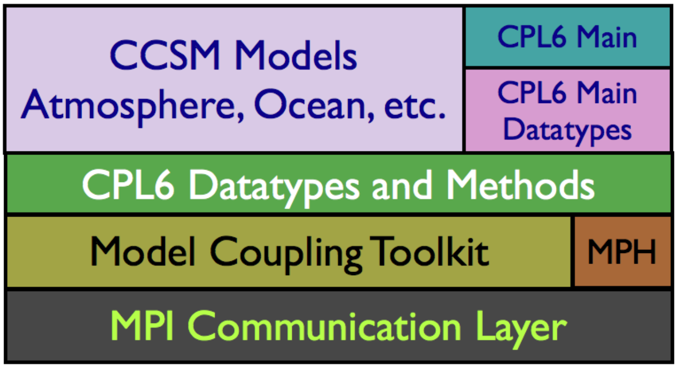

Figure 1: The CCSM Coupler CPL6 software stack.

datatypes that are used by the CCSM's client models and coupler. Virtually all of the code base described thus far is implemented in Fortran90, and much of it uses Fortran90 derived types and pointers. Greater detail on MPH, MCT, and CPL6 was provided by He and Ding [7], Larson and Jacob [10, 8], and Craig et al. [2], respectively.

Figure 2 summarises the class hierarchy and functionalities of the MCT and CPL6 classes used in CCSM. CPL6's internal classes (middle layer in Figure 2) are those visible inside the CPL6 application but not seen by the physical component models. In addition to datatypes, both MCT and the CPL6 toolkit offer library routines to support parallel coupling operations. These include parallel transfers Send() and Recv(), intermesh interpolation, data merging, flux conservation enforcement, and computation of diagnostic and flux quantities. The CPL6 Main Datatypes are those visible to CCSM's client models, and are the high level interaction mechanism between them and the coupler. Parallel data transfer is encapsulated by the Contract class, which holds all of the data necessary to describe a single model-coupler interaction. The physical fields under exchange is set through the Fields module, which contains the master list of all coupling fields. The interface module contains all the routines a client model uses to interact with the coupler.

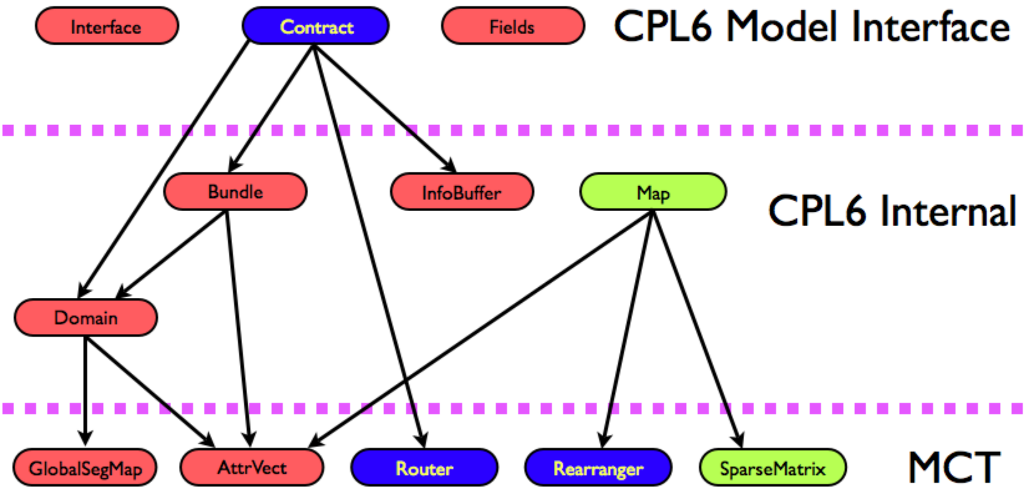

Figure 2: The class hierarchy diagram for CPL6 and its relationship to MCT classes. Directed arrows denote dependencies; that is, an arrow points from a class to one needed to construct it. Colour key by class functionality: pink, data description; blue, data transfer; and green, data transformation.

CPL6 represented a breakthrough in coupler technology as it allows one to make two key scientific modifications to CCSM with relative ease—*model substitution* and *addition of coupling fields*. Model substitution is the replacement of a client model with another implementation (for example, changing the atmosphere model); this has enabled numerous CCSM experiments using different ocean and land models. The ability to add additional coupling fields such as trace chemical concentrations exchanged between atmosphere, ocean, and biosphere accelerated dramatically the addition of a biogeochemical cycle to CCSM. The CPL6 hub application and high-level interfaces, together with CCSM's other components satisfy the definition of an *application framework* stated by Fayad et al. [5]. CPL6 has thus expanded substantially the scientific horizons of CCSM.

However, the current CCSM framework offers no easy solutions to two broad classes of scientific questions:

1. How does one modify CCSM's coupling strategy?
2. How does one embed CCSM in larger systems?

Coupling strategy changes require reprogramming parts of the CPL6 application. In principle, one can view the CPL6 main program as a disposable entity, recoding a Fortran replacement that serves a specific scientific objective—an approach compatible with the design philosophies of MCT and CPL6. A Python system, however, would be more amenable to rapid prototyping than its Fortran counterpart. Embedding CCSM will likely require abstraction of CPL6's classes to a higher level, and Fortran's support of OOP is problematic. Specifically, in CCSM Fortran class dependency relationships are hand coded through *delegation*.[6] As a result, MCT classes derived from lower level MCT classes have inheritance manually implemented. This is why MCT's class hierarchy is fairly shallow—MCT's developers considered constructs such as CPL6's Domain and Bundle unsustainable. CPL6 was programmed with considerably less OOP discipline; CPL6 classes such as the Domain and Bundle

[6] Decyk [4] et al. explain how delegation is used in Fortran90 to support method inheritance.

have only those methods implemented that were required for their use in the CPL6 application.

We believe a more flexible Python-based coupling infrastructure solves the aforementioned problems, and supports rapid prototyping of new coupled climate and Earth system model applications.

3 Python coupling infrastructure for CCSM

Our immediate goal is a Python reimplementation of CCSM's coupling infrastructure. This required Python interfaces for MCT, MPH, the CPL6 toolkit and the CPL6 coupler. This software construction project faced two key technical challenges: language interoperability, specifically, exporting Fortran classes and methods to Python; and supporting multilanguage, multiple executable codes communicating via MPI. Below we describe how we surmounted these obstacles.

3.1 Python MPI interfaces

Numerous Python interfaces to MPI exist, including PyMPI,[7] PyPAR,[8] Scientific Python,[9] and MyMPI.[10] For CCSM, we must support a multiexecutable application that bridges both Fortran and Python. At the time we started our work, PyPAR was rudimentary, and we were unaware of the MPI implementation within Scientific Python. We considered PyMPI and rejected it as a set of Python MPI bindings because it is a Python interpreter as well as a set of MPI bindings and is thus the main parallel application that controls the initialisation of MPI. This approach was not compatible with our requirement

[7]http://pympi.sourceforge.net/
[8]http://sourceforge.net/projects/pypar/
[9]http://numpy.scipy.org/
[10]http://sourceforge.net/projects/pydusa/

of allowing CCSM's components to call MPI_Init independently. PyMPI would assume that all the processors in MPI_COMM_WORLD were Python processors executing the same startup code and would hang when the other executables didn't execute the same code.

MyMPI enabled us to get multiexecutable Fortran/Python support working. However, as development progressed, more features and MPI functions were needed. MyMPI was at the time of this writing still a low-level library. To meet our requirements, we developed our own set of Python MPI bindings, based on MyMPI, called MaroonMPI (MMPI). More information about MMPI and its source code are available from the MMPI Google Code site.[11]

3.2 PyMPH

Sharing a set of MPI communicators between the Python and Fortran languages in a multiexecutable application proved difficult. One problem was the construction of communicators. In CCSM this is delegated to MPH. MPH is, in principle, a powerful tool for managing interprocessor communications in a multiple executable context. It is also a pure Fortran90 code and uses derived types in its implementation, confronting us with a Fortran language interoperability problem (see Section 3.3 for more details). We faced a choice of coding Python wrappers for MPH or reimplementation.

After analysing how MPH is used in CCSM, we estimated it could be quickly replaced with about 200 lines of Python code. It was necessary to design a subsystem that could allocate communicators to processor groups that could successfully interoperate between Fortran90 and Python. While the design of this tool is relatively simple, its implementation was not straightforward as cross-language multiprocessor multiexecutable debugging tools are of necessity implemented on an as-needed basis. To support this new initialisation, we also needed to add a new function to the Fortran version of CPL6 library

[11] http://code.google.com/p/maroonmpi/

that supported this simple CCSM-only initialization. The only Fortran code modified is within the CPL6 library.

3.3 PyMCT

Fortran is notoriously difficult to interface with other programming languages because of the lack of a specific standard for array descriptors (also known as 'dope vectors') in the Fortran90/95 standard. Furthermore, this standard fails to offer even a standard API for querying the dope vectors to ascertain array dimensionality, extent, stride, starting address, and base type (for example, REAL and INTEGER). Fortran77 was easy to interface to other languages because it enforced 'call by reference'; that is, one supplied an array's starting address, and further information about the array was not part of the datatype. The Fortran2003 standard offers the BINDC attribute to datatype declaration; at the time of this writing BINDC was not widely available in compilers. Thus, the only solution was a vendor-by-vendor implementation such as CHASM [13].[12]

The MCT API is expressed by using Fortran derived types and Fortran pointers, confronting us again with the Fortran interlanguage interoperability problem. MCT constitutes approximately $50,000$ lines of code; too large to reimplement within the scope of our project. Instead, we chose to wrap MCT using the Babel language interoperability tool, which enables us to generate Python bindings for MCT automatically.

Babel relies on interface description using the *Scientific Interface Definition Language* (SIDL). We created interface blocks with the MCT's subroutine and function interface signatures and stored these in a 'SIDL file.' We next ran Babel using the SIDL file as input to generate an *internal object representation*

[12]Between the time of submission and time of publication of this article, Fortran compiler developers have made significant progress in implementing BINDC; Chivers and Sleightholme [1] provide a list of compilers that now support BINDC.

(IOR) of the interfaces in the C programming language. We then invoked Babel using the IOR to create a 'stub' API callable from Python.

The resulting Python API for MCT, PyMCT, is a *restricted* API. This is due to the use of optional arguments in the native Fortran API for MCT. SIDL does not support optional arguments; one must instead program a SIDL representation for each possible calling signature, and for MCT some routines have a large number of possibilities. That said, PyMCT is sufficiently complete to support a wide variety of multiphysics and multiscale parallel coupling applications, including CPL6.

Ong et al. [12] discuss in detail the construction of PyMCT using Babel, along with simple Python programming examples employing PyMCT. One can download PyMCT bundled with a C++ API for MCT from the MCT Web site.

3.4 PyCPL

Armed with a working version of PyMCT, reimplementation of the Python version of CPL6 classes was straightforward. Just as CPL6 derives its classes and methods from MCT classes and methods in its own derived types (Bundle, Domain, Contract, Map), PyCPL derives its classes from PyMCT classes and methods to provide the necessary coupler logic and behavior. In most cases the Python classes are just translations of their Fortran counterparts. Because Python is an OOP-friendly language we took the opportunity to engineer some things differently in PyCPL. For example, the CPL6 Bundle type is not used. Instead of recreating the Bundle we opted to make PyCPL rely more on MCT's built-in types that already have the requisite functionality.

The PyCPL application is largely a translation from Fortran of CPL6 into Python. 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. We made no changes whatsoever to the client models’ source code.

The source code of PyCPL and PyCCSM, along with installation instructions and a wiki recording project development, are available from the PyCCSM Google Code site.[13]

4 Testbed case: CCSM with ‘live models’

CCSM supports three types of each client model: *dead models*, which provide no time-evolving state; *data models*, which provide the coupler a time-evolving output state read from data files; and *live models*, which compute a time-evolving state in response to coupler input and supply the coupler with time-evolving input. Live models are used for production runs. We chose to test PyCPL with live models, which amounts to an integration test for PyCCSM. In the development of CPL6 this was the final test case before the code underwent scientific tests involving long model integrations to validate CCSM’s climate.

Both CCSM and PyCCSM were benchmarked for a five day simulation using the same configuration. The atmosphere model used a T42 grid with 64 latitudes and 128 longitudes; the land model uses the same grid over the Earth’s land masses. The ocean grid was the gx1v3 grid with 384 latitudes and 320 longitudes; the sea ice model uses this grid in regions where sea ice is present. We timed these runs on the Jazz cluster at the Laboratory Computing Resource Center at Argonne National Laboratory. Jazz has 350 nodes of 2.4 GHz Pentium Xeon processors and has a gigabit ethernet interconnect. The compilers used in this study were Absoft 9.0 Fortran and gcc 3.2.3. Version 2.4 of the Python interpreter was used. The test case configuration used a total of 55 processors with 24 ocean processors, 16 land processors, eight ice processors, six land processors, and one processor running the coupler. At present, we are able to get the Python coupler to run reliably only on one

[13] http://code.google.com/p/pyccsm/

node. CCSM takes 361 seconds to run the five day benchmark, and PyCCSM accomplishes this task in 741 seconds. The Python system is slower by a factor of 2.05.

The performance results are disappointing. However, computationally intensive codes in high level languages like Python are typically one to two orders of magnitude slower in performance than comparable codes in conventional compiled languages. One likely cause is the large amount of flux calculations that are performed in Python; these sections of code were manually translated from their Fortran90 counterparts. These code blocks contain large nested loops with numerous arithmetic operations, for which Python is known to function much more slowly than compiled languages like C or Fortran. Loop and array based math processing in Python is best handled by the NumPy package,[14] which offers a Python front end to f77-compatible arrays and computation routines implemented in C. Reformulation of the PyCPL flux calculations in terms of NumPy arrays, with the flux computations dispatched to the appropriate NumPy methods will likely speed up substantially this part of PyCPL. We believe that once PyCPL is running reliably on multiple processors, we can exploit its scalability to reduce the current Python overhead to an acceptable level. We base this prediction on MCT's previously demonstrated scalability [8, 10] and on the low measured overheads due to Babel on MCT's multilingual interfaces [12]. This, combined with the fact the CPL6 coupler is idle much of the time, awaiting data from the CCSM client models, can be used to achieve acceptable performance. Specifically, for a given problem, we predict we will be able to run PyCPL on significantly more processors than CPL6 does in CCSM, and get sufficient PyCPL performance to attain production level PyCCSM throughput.

[14] http://numpy.scipy.org

5 Conclusions and future work

We have built a Python implementation of the coupling infrastructure for a major climate modelling system, PyCCSM. PyCCSM is the first parallel coupled climate model built using Python. This also marks the prototyping in Python of a grand challenge scientific software application framework. We have benchmarked PyCCSM for a significant CCSM test case.

At present, PyCCSM runs reliably with only a single coupler processor, but with multiple processors for atmosphere, ocean, ice, and land models. In this configuration, PyCCSM is roughly twice as slow as the native Fortran CCSM. Further study of PyCPL's behaviour on multiple processors is required to enable run-time reliability. Once this is accomplished, we will focus on improving PyCCSM's performance through scalability and delegating where appropriate calculations currently performed in Python to NumPy.

We are at a crossroads. We could work to complete PyCCSM based on CCSM 3.0, or wait until the release of CCSM 4.0 in April 2010. CCSM4 has dispensed with many of the CPL6 classes and is using MCT datatypes more explicitly. The coupler is now the driver for the entire model and can execute the model as a single or as multiple executables, and supports both parallel and serial [6] process compositions. CCSM4's coupler, CPL7, will be much easier to reimplement in Python compared to CPL6. Given the wider implementation by compilers of the Fortran2003 BINDC attribute [1], we will also investigate the viability of using this feature in place of Babel to implement Fortran/Python interoperability. We are strongly considering applying the lessons learnt thus far to CCSM4 once it is released.

Our ultimate objective is to embed PyCCSM in run control environments, for example combining the model control with the ensemble control in a single scripting environment, which would simplify considerably the process by which ensemble climate model scenarios are generated. This toolset for reprogramming the model control layer will enable many previously unattempted use cases for coupled climate models, including statistical analysis of model output, objective tuning of model parameters, data assimilation, and automated run-time selection model components. Given Python's programmer friendliness, in this context one not only will be able to leverage the composition character of software frameworks but will also enjoy a free hand to modify both the components and the framework itself. We believe that Python thus can provide formidable competition to conventional software frameworks in the scientific productivity computing arena.

Acknowledgments This work was produced by the University of Chicago Climate Systems Center, funded by the United States National Science Foundation Information Technology Research program under award ATM-0121028. PyMCT development was supported by the United States Department of Energy's Scientific Discovery through Advanced Computing initiative through the Common Component Architecture project. 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 Australian National University for organising the Python workshop where this work was presented. We also thank Tom Epperly, Susan Coughlin, Boyana Norris, Matt Knepley, and Craig Rasmussen for their advice and assistance.

References

[1] Ian D. Chivers and Jane Sleightholme. Compiler support for the fortran 2003 standard. *SIGPLAN Fortran Forum*, 28(1):26–28, 2009. doi:10.1145/1520752.1520755 C1122, C1126

[2] 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. doi:10.1177/1094342005056117 C1114, C1116, C1117

[3] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Syzmanski. Expressing object-oriented concepts in Fortran90. *SIGPLAN Fortran Forum*, 16(1):13–18, 1997. doi:10.1145/263877.263880 C1114

[4] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Syzmanski. How to support inheritance and run-time polymorphism in fortran 90. *Computer Physics Communications*, 115(1):9–17, 1998. doi:10.1016/S0010-4655(98)00101-5 C1119

[5] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley and Sons, New York, 1999. C1119

[6] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley, Reading, Massachusetts, 1995. C1116, C1126

[7] 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. doi:10.1177/1094342005056118 C1117

[8] Robert Jacob, Jay Larson, and Everest Ong. $\mathsf{M} \times \mathsf{N}$ communication and parallel interpolation in CCSM3 using the Model Coupling Tookit. *Int. J. High Perf. Comp. App.*, 19(3):293–308, 2005. doi:10.1177/1094342005056116 C1114, C1117, C1125

[9] J. Walter Larson. Ten organising principles for coupling in multiphysics and multiscale models. In Wayne Read, Jay W. Larson, and A. J. Roberts, editors, *Proceedings of the 13th Biennial Computational Techniques and Applications Conference, CTAC-2006*, volume 48 of *ANZIAM J.*, pages C1090–C1111, February 2009.

http://anziamj.austms.org.au/ojs/index.php/ANZIAMJ/article/view/138[February 20, 2009]. C1114, C1116

[10] 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. doi:10.1177/1094342005056115 C1114, C1117, C1125

[11] Intergovernmental Panel on Climate Change. *Climate Change 2007—The Physical Science Basis: Working Group I Contribution to the Fourth Assessment Report of the IPCC*. Cambridge University Press, Cambridge, 2007. C1116

[12] Everest T. Ong, J. Walter Larson, Boyana Norris, Robert L. Jacob, Michael Tobis, and Michael Steder. A multilingual programming model for coupled systems. *International Journal for Multiscale Computational Engineering*, 6(1):39–51, 2008. doi:10.1615/IntJMultCompEng.v6.i1.40 C1123, C1125

[13] 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. doi:10.1002/cpe.909 C1122

Author addresses

1. **Michael Tobis**, Institute for Geophysics, University of Texas, Austin, TX, USA.
   mailto:tobis@gmail.com
2. **Michael Steder**, Department of Geophysical Sciences, University of Chicago, Chicago, IL, USA.
3. **J. Walter Larson**, Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL, USA; Computation

Institute, University of Chicago, Chicago, IL, USA; School of Computer Science, The Australian National University, Canberra, AUSTRALIA.

4. **Raymond T. Pierrehumbert**, Department of Geophysical Sciences, University of Chicago, Chicago, IL, USA.
5. **Robert L. Jacob**, Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL, USA; Computation Institute, University of Chicago, Chicago, IL, USA.
6. **Everest T. Ong**, Department of Atmospheric and Oceanic Sciences, University of Wisconsin, Madison, WI, USA.