Jun 18th, 9:00 AM - 10:20 AM

MODPI: A parallel model data passing interface for integrating legacy environmental system models

Andre Q. Dozier
Colorado State University, andre.dozier@rams.colostate.edu

Olaf David
Colorado State University, odavid@colostate.edu

Yao Zhang
Colorado State University, rzzhangyao@gmail.com

Mazdak Arabi
Colorado State University, mazdak.arabi@colostate.edu

International Environmental Modelling and Software Society (iEMSs)
7th Int. Congress on Env. Modelling and Software, San Diego, CA, USA.
Daniel P. Ames, Nigel W. T. Quinn, and Andrea E. Rizzoli (Eds.)
http://www.iemss.org/society/index.php/iemss-2014-proceedings

MODPI: A parallel model data passing interface for integrating legacy environmental system models

André Q. Dozier [a], **Olaf David**[a], **Yao Zhang**[a] **and Mazdak Arabi**[a]

[a] *Colorado State University, Campus Delivery 1372, Fort Collins, CO, USA*
(andre.dozier@rams.colostate.edu, odavid@colostate.edu,rzzhangyao@gmail.com,
mazdak.arabi@Colostate.edu)

Abstract: Integration of environmental system models across multiple disparate disciplines has been an area of growing interest because such models typically only represent knowledge and advancements in one discipline or several, similar disciplines. Many challenges have arisen, though, in integrating such models due to varied programming expertise amongst modelers, dependence on specific frameworks or operating systems in addition to different programming languages and requirements for two-way feedbacks that are crucial for improving model representation of the physical reality. In response to these challenges, this paper presents a novel approach based on a publish-subscribe type system to both simplify and improve the model integration process. The approach allows access to legacy model variables at any point during simulation by separate processes on different machines in various programming languages, thus providing for two-way feedbacks between models. Additionally, a level of abstraction is given to the model integration process that allows researchers and other technical personnel to perform more detailed and interactive modeling, visualization, optimization, calibration, and uncertainty analysis without requiring deep understanding of interprocess communication. Utilizing this approach may significantly decrease developer time and potentially computation time, and enhance interdisciplinary research by providing detailed two-way feedback mechanisms between various simulation models with minimal changes to legacy code. A simple test case of a connection between a biogeochemical model and a finite element mass and heat transfer model demonstrates the utility and ease of the new model integration approach.

Keywords: Integrated Assessment and Modeling, Integrated Environmental Modeling, Interprocess Communication

1 INTRODUCTION

Academic disciplines in the geophysical sciences typically have boundaries or limitations within which students of a particular discipline seem to remain. However, interconnections between the various geophysical attributes of the real world do not seem to have such clearly defined boundaries. A more holistic modeling framework is required for understanding processes of and societal interactions with geophysical phenomena, and therefore integration of models across disciplines has become a focal point in assessing transdisciplinary problems for more than 30 years [Laniak et al., 2013]. However, existing generic solutions or platforms require geophysical researchers (i.e., not computer scientists) to have a significant depth and breadth of computer programming knowledge to overcome challenging systems modeling integration feats such as implementing complex, two-way feedbacks between models and establishing communication between models of different languages, frameworks, or architectural requirements, which is an important requirement for achieving interoperability between models and modeling frameworks [Laniak et al., 2013; Matott et al., 2009].

Many workflow systems or model integration platforms facilitate communication between models and subcomponents in a input-execute-output fashion, where output from one component (or model) is passed to the next component (or model), which can even be performed in an iterative fashion as is the case in OpenMI [Moore and Tindall, 2005]. In this sense, model integration platforms work much more like generic scientific workflow tools with spatial and temporal data transformations between nodes in the workflow graph [Curcin

and Ghanem, 2008; Deelman et al., 2009]. However, unidirectional interactions are insufficient for the types of feedbacks that are often required between geophysical, ecological, and socio-economic models within timeseries and iterative sub-loops as well as within optimization routines and uncertainty analyses. When the need for interactive feedbacks between models arises, the conceptual model integration can be oriented i) by running Model 1, then running Model 2, then Model 1 again, etc. until iterative convergence criteria are met, ii) by combining Models 1 and 2 into a new, super-model by compiling them together or by calling one inside the other, or iii) by pausing for interprocess communication between models or model components to send or receive updates to the model state. Each approach has its own barriers to implementation as well as advantages and disadvantages once implemented.

Conceptually, the first approach may be the easiest to implement, but it is likely the slowest computationally. The second approach usually involves more programming, which typically entails breaking one of the models into smaller components (typically three components initialization, run, and finalization), ensuring that values are passed correctly into sub-model components, or writing and reading input and output files for another model during execution. Disadvantages of this second approach include that models must be inherently interoperable (e.g., both models are written in native code such as Fortran and C) if they are to be compiled together, which makes further development of the super-model dependent on both models. Upkeep is difficult, and models no longer retain individuality. If instead one model is run within the other, there are overheads associated with writing files, running a sub-process, and reading output files, which may be computationally costly.

The third approach introduces a much more extensive and sophisticated feedback mechanism to the field of integrated modeling than the first two approaches. In addition, it allows for interoperability between models with varying architectural, platform, or license dependencies, and yet maintains model individuality. However, this approach requires much more programming expertise to achieve. For example, a detailed understanding of client-server (i.e., socket) programming is required for a user to retrieve or set values from within the execution of a model as was accomplished by Becker and Schuttrumpf [2011]. In addition to the workflow controller needing to know when to transfer values from one model to another, both models need to know when to stop and wait for values, or stop and send values. Thus, the extent to which a geophysical scientist would be capable of performing this type of interactive model integration is extremely limited, and consequently negatively impacting its acceptance in the broader modeling community.

Advances have been made in supporting interprocess communication within frameworks like the Common Component Architecture (CCA) for models dependent on high-performance computing [Larson et al., 2004], and the Earth System Modeling Framework (ESMF) for models of a gridded nature common to those in atmospheric models [Hill et al., 2004]. However, to attempt transforming some legacy environmental models into grid-based computations may be infeasible in many cases due to fundamental changes in the modeling approach, and to link such models with other disciplinary models is still a task that remains to be done. We are not aware of any previously developed standard and implementation thereof that generically decouples computations and disciplinary models using interprocess communication without requiring a particular format for numerical computations (e.g., gridded calculations). Such a standard may help to meet cross-platform interoperability goals as stated by Laniak et al. [2013] due to its ability to perform fine-grained manipulation of a model state, and address development barriers when attempting integration across different programming languages, compilers, and platforms [Matott et al., 2009].

Therefore, the goal of this paper is to facilitate and abstract the legacy model integration process to include complex, multi-directional interactions between models and components of different architectures and maintain model individuality without requiring onerous programming knowledge, an important criterion [Laniak et al., 2013]. Objectives of this study are i) to design and develop an abstracted interface, referred to herein as MOdel Data Passing Interface (MODPI), for model integration that simplifies complex interactions between legacy models and modeling platforms of disparate disciplines, and ii) to demonstrate the abstracted model integration system with a useful test case. To accomplish this, a publish-subscribe concept is combined with a parallel programming interface known as the Message Passing Interface (MPI) to provide users with read and write access to any state variable within a legacy model during its execution. MODPI is highly interoperable between languages and has been preliminarily tested with the following programming (and scripting) languages: Fortran, C, Java, and MATLAB script, and could be theoretically applied to many others such as C++, languages in the .NET framework, Python, R, and other popular languages. However, the full generic model integration interface and library MODPI has only been built and implemented extensively

for Fortran and C for the purposes of this paper.

2 METHODOLOGY

During development of the MOdel Data Passing Interface (MODPI), we found that abstracting model integration improved the ease of its us by improving its intuitive usage in addition to decreasing invasiveness of the sample implementation of MODPI. The framework MODPI offers, therefore, could potentially overcome many challenges to model integration, such as 1) retain model individuality and minimize code changes within legacy model code base, 2) provide two-way feedback mechanisms between models or other components during execution by establishing access to state variables by name, 3) and establish independence from a single machine, operating system, model architecture, and programming language. The method of abstraction described herein has theoretically overcome all three challenges, but has only been tested and proven in this study to overcome the first two.

As we developed MODPI, we aimed at retaining model individuality and minimizing code changes within legacy environmental system models while still allowing for state variable access during simulation, and therefore implemented a coding mechanism known as “events” in many higher level languages such as Java or .NET, or “callbacks” in lower-level languages. MODPI uses the term “events” because it utilizes new object-oriented features of Fortran. “Events” refer to locations in the code of a model or component of a software package where something of interest happens or is about to happen. “Subscribers” are programs that wrap the model with intentions of getting or setting model data at an event of interest. When subscribers “subscribe” to an event, a procedure or function pointer is added to a list that the event then loops through and executes when “fired” or “published”. When an event is fired, the “run” procedure is executed and event data is passed to the subroutine in addition to the subscriber object. When no subscribers exist, the firing method of the event simply exits without knowing or caring whether any subscribers exist. Two interfaces exist to allow extra flexibility, one for subroutines with event data passed as an argument, and one for objects that extend the subscriber base class. Event data is passed to all subscribers, which allows read and write access to local data during simulation. In cases where global data exists, data can be exchanged by directly accessing global data without assigning data to the event. Events add value to MODPI by decoupling the model from the integration platform while still allowing access to model state, whether local or global, during its simulation. More advantages are discussed in the following section.

A particular implementation of the Message Passing Interface (MPI) known as Open MPI [Gabriel et al., 2004] was utilized for this demonstration of the MODPI interface, although other implementations could potentially be utilized. Open MPI is a well-developed MPI implementation that provides the necessary interprocess communication between models in a fashion that ensures message deliverance, detects deadlocks and killed processes automatically. For purposes of environmental modeling frameworks, these properties are desirable in order to ensure that data is passed between models correctly.

An implementation of the MODPI interface has been implemented for the purposes of this paper, and has been applied to the linkage of DayCent [Parton et al., 1994] and HYDRUS-1D [Šimnek et al., 2008]. The example model shown in Fig. 1 is DayCent, which provides daily simulation of biogeochemical fluxes, and may benefit from improved soil water dynamics at a subdaily scale [Yuan et al., 2011], and has therefore been linked with HYDRUS-1D using MODPI, which acts to exchange soil water content data every day during simulation between HYDRUS and DayCent. In this way, soil water state within each layer of the HYDRUS model does not have to be reset and updated by DayCent every time the finite element model for HYDRUS is to be run, but HYDRUS maintains state and DayCent updates only the needed values. This linkage is not the topic of this paper, but does demonstrate the utility of MODPI to enhance scientific model integration development. Two events within HYDRUS are defined: one event fires prior to solving for soil water content, where it receives soil infiltration and evapotranspiration estimates from DayCent, and the other event fires prior to writing output files, where HYDRUS sends DayCent results of its soil water content calculations.

The MODPI implementation built for the purposes of this paper simply reads a file supplied by the user that defines 1) the process (i.e., the model) with which data will be exchanged, 2) MPI options (namely, tags), 3) the events to subscribe to, 4) whether to receive data or send data at that event, and 5) the variable within the code base of the model containing the desired data that is to be sent or received. MODPI depends only

```
use modevent

! Defines the interface for sending / receiving
interface
    subroutine ExchangeValue(variable,vartype,
        otherprocid,tag,eventdata)
        import :: ieventdata
        character(len=*) :: variable
        integer :: vartype,otherprocid,tag
        class (ieventdata), pointer :: eventdata
    end subroutine
end interface

! Defines the interface for getting events
interface
    subroutine FindEvent(eventname, e)
        import :: event
        character(len=*) :: eventname
        type(event), pointer :: e
    end subroutine
end interface

! Initialize MODPI, MPI, subscribe to model events
subroutine modpi_init(receiver, sender, getevent)
    procedure(ExchangeValue) :: receiver
    procedure(ExchangeValue) :: sender
    procedure(FindEvent) :: getevent
    ...
end subroutine

! Finalize MODPI
subroutine modpi_finalize()
    ...
end subroutine
```

```
program daycentmodpi_main
  ! use statements
  use modpi
  use daycentmodpi
  use codeevents_ddc
  implicit none

  ! declarations
  integer :: eventi

  ! other things to change within daycent
  eventi = OnDayCentInit%subscribe(initialize_connection)

  ! initialize modpi
  call modpi_init(recv, send, getevent)

  ! start daycent
  call ddcent_main

  ! finalize modpi
  call modpi_finalize()

end program daycentmodpi_main
```

Figure 1. The API for MODPI (left) and an example program that implements MODPI for DayCent (right). The API requires building three subroutines where the receiver and sender subroutines implement ExchangeValue and the subroutine that retrieves model events implements FindEvent, and executing modpi_init(receiver, sender, getevent) and modpi_finalize() subroutines before starting the model and after the model executes, respectively. In the example program on the right, the three subroutines and events are defined in separate modules named daycentmodpi and codeevents_ddc, respectively.

on MPI and the Events class, and does not depend on any particular environmental model, nor do models that implement MODPI actually require the MODPI library to build the core model component. Instead, a model developer that wishes to implement MODPI for his or her own model must first add events to their model in locations of interest, which may be before a particular parameter is used for calculations or after a model output has already been calculated. Secondly, the model developer must build a wrapper for the model that initializes MODPI with three subroutines that in turn automatically subscribe to events in the model that either receive or send data based on the MODPI input file provided to the wrapper as arguments at the command line.

The MODPI interface is defined on the left and exemplified on the right in Fig. 1. As seen on the right in the figure, the wrapper itself must "use" or import a module that contains the three subroutines to pass to MODPI, the MODPI module itself, and the modules or libraries of the original model containing any data that is manipulated outside of the MODPI interface, which can be seen in the line where a subroutine named "initialize_connection" is subscribed to the event in DayCent named "OnDayCentInit". Then, the system initializes DayCent its connection by calling "modpi_init(·)", followed by running the model via its main subroutine, and finalized by calling "modpi_finalize()". The three subroutines supplied by the wrapper that are passed into modpi_init(·) define how to 1) receive data, 2) send data, and 3) get event objects from the specific model of interest. In this case, these subroutines are found in the daycentmodpi Fortran module. The "recv" and "send" subroutines implement the same interface as "ExchangeValue" found on the left in Fig. 1. The "getevent" subroutine implements the same interface as "FindEvent", also found on the left in Fig. 1. Further description for these subroutines is found below. A failure is detected during integrated modeling when modpi_finalize() is not called by all models. Thus, although not shown here, the HYDRUS model wrapper also implements a similar procedure to that shown on the right in Fig. 1.

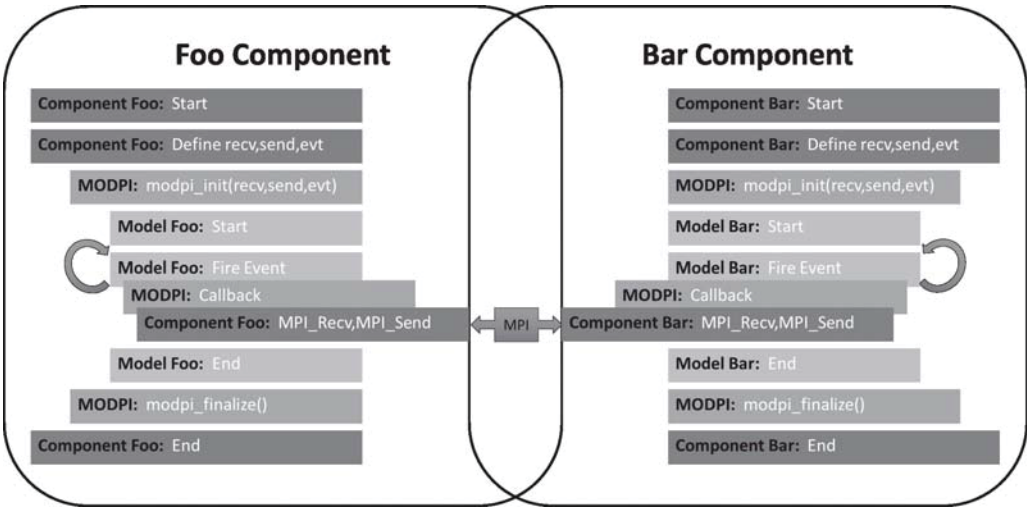

Figure 2. A simplified flow chart of how MODPI could work with one event inside two arbitrary models "Foo" and "Bar". Components represent the wrapper implementing MODPI for each model.

To run the system, the "mpirun" program within Open MPI distribution packages is called with an application (*.app) that defines working directories, paths to wrapper programs for each model within the MODPI implementation (in this case "daycentmodpi" and "hydrusmodpi" were the wrapper programs implementing MODPI for DayCent and HYDRUS, respectively), the required command line input arguments for each wrapper program, and the path of the MODPI input file. Thus, the MPI implementation starts DayCent and Hydrus models as two separate processes and runs them in parallel, sending data over the network (or over shared memory in advanced MPI implementations when the processes are running on the same physical machine) between models at events specified by the modeler in the MODPI input files. It is important to note that there needs to be a MODPI input file for each model wrapper, where the wrapper for DayCent had one MODPI input file associated with it, as did the the wrapper for HYDRUS.

Modifications to code for the DayCent-HYDRUS connection were very minimal. DayCent has 214 source files with about 27,400 actual lines of code (about 19,500 (123 files) are Fortran, 5,400 (80 files) are C, and the rest are a combination of other header or make files). Only 3 files with a total of 175 (0.7%) lines of code were added to the project to make it compatible with HYDRUS, 109 of which convert data from its representation in DayCent to its representation in HYDRUS. Data conversions like these may be incorporated into future versions of MODPI for even more generic model integration. For HYDRUS-1D, there are about 9060 lines (10 files) of pure Fortran code, and only 38 (0.4%) new lines of code within 1 extra file were added. In both DayCent and HYDRUS there were a few lines in the original source code that were changed, and main program routine was moved its own separate file in order to build each model as a library to be used by the MODPI wrapper component. Flags are used primarily to switch off certain unnecessary calculations only when the models are connected. No major changes were made in the code so that the models still operate and run completely on their own as if no changes were made (i.e., they do not depend on each other), but if the MODPI wrapper has subscribed to the events, HYDRUS and DayCent are then connected.

The general flow of coupled models within MODPI as shown in Fig. 2 is important to understand prior to implementing MODPI. Using the arbitrary names found in the figure, the executable provided by Open MPI starts both components "Foo" and "Bar". Each component calls modpi_init(·) to subscribe to model events, starts model execution, and on completion calls modpi_finalize(). MODPI automatically subscribes to events and sets up receivers and senders to send and receive variables at the events of interest. During simulation of the model, when the events of interest fire, they execute the subscribed senders and receivers built by the wrapping component program. Senders and receivers are subroutines that obtain a variable by name and then send or receive the variable using MPI. Because the MODPI implementation in this paper is built in Fortran, variables were obtained by name in a "select case" statement. As long as all symbols are exported

into the model library, model data can be accessed by name using the symbol table within the library, but this can be difficult when dealing with allocatable and high dimensional arrays in Fortran. Generic reflection could also potentially be utilized in higher level languages, and initial implementations have been started in both MATLAB and Java.

In order to implement MODPI for other models, a fairly straightforward set of steps should be followed. First, insert events into the model at locations of interest, where data of interest has just been set (when sending data) or is about to be set (when receiving data). If data is global, an event with no event data is sufficient. However, if data is local, build a subclass of "ieventdata" that holds the event's data, or place new variables in a globally accessible module and exchange data through those new variables. Ensure that the model is compiled to allow access to these events and associated data from the compiled library. Second, build the sender and receiver subroutines that attain access to model variables by name, whether by "select case" statements or through reflection. Within these subroutines, call MPI_Send(·) or MPI_Recv(·), respectively, in order to exchange data for each variable of interest. Third, build a subroutine that retrieves model events by name. Lastly, build a program that initializes MODPI by calling MODPI_Init(receiver, sender, getevent), runs the model, calls MODPI_Finalize(), and then finishes. For most models, running the model from a library requires that the main entry point ("program" in Fortran) be changed to a regular subroutine that a new "main" simply calls. Additionally, remove non-standard program exits because they throw an MPI "not finalized" error (e.g., many developers use "stop" in Fortran even to end normal model execution; these should be removed).

3 Discussion

The MOdel Data Passing Interface (MODPI) is presented as a potential solution to some of the contemporary model integration challenges including two-way feedbacks between large legacy models of different architectures or programming languages. MODPI provides access to variables within the simulation of a legacy model to other processes (models or components) running in parallel. MODPI abstracts the model integration process by providing users the ability to select a point in the model to exchange data at an "event", send or receive data by name, and select which other process to send to without requiring compiling models together. Utilization of events to abstract within-simulation data access allows 1) minimal code changes within legacy models but still provide exchange of all data during simulation, 2) the capability to decouple models seamlessly from the model integration framework, 3) legacy models to be compiled without any additional dependencies such as MPI or other coupled models with the exception of the event library, 4) further individual model development and versioning while maintaining interoperability within the model framework, and 5) insert an intuitive level of abstraction into the model integration process. Utilizing the parallel programming interface MPI granted MODPI the capability of maintaining system state within each model and changing only required data at the moment of interest, without requiring extensive modifications to code.

A few practical considerations to consider before implementing MODPI for a particular model are to ensure 1) there is need for within-simulation data access and exchange, 2) simulation times are long enough between each MPI call to justify having network communication which can slow execution time (otherwise, it may be faster to compile them together, or utilize events to decouple the models without interprocess communication), and 3) the framework or architecture supports MPI. If one or more of the three considerations do not justify using MODPI, the various model integration patterns or standards that were discussed in the introduction may easily take preference. However, if all three of the considerations above are true, then the abstraction of the model integration process that MODPI offers may serve to facilitate and speed model integration significantly. Additionally, if sufficient computation can be performed in each model in parallel, MODPI may actually speed model execution as well as allow for model comparison and averaging during simulation.

The simple test case used in this paper demonstrates how MODPI maintains model individuality while still allowing for complex, two-way feedbacks between models. Code changes were demonstrated to be minimal, being about 0.4% and 0.7% of HYDRUS and DayCent, respectively. Less than 10 lines of code were added to the original code base of each model, and no subroutines or methods were shifted around. The abstraction of the model integration process that MODPI offers sped the process of integration significantly by providing a framework for understanding model integration between DayCent and HYDRUS. Due to the scale

of the example models, the fact that computations are minimal and extremely fast between timesteps, and that the two models are not well suited for parallel computations, network communication at each timestep was excessive and slowed model execution down by an order of magnitude. Although this may have been largely due to running the model connection setup on a virtual machine, network communication overhead is an important consideration when connecting models that do not require much computation time between network communication instances.

4 Future Work

Future work and developments on MODPI would be to make it 1) more interoperable, 2) more generic, and 3) to improve its speed and analyze causes of slow execution. In order to improve its interoperability, a full MODPI implementation in pure C/C++ should be built, followed by Java, MATLAB, Python, R, and .NET, because these are common and popular languages for environmental systems models. Tests of MODPI should also be be run across operating systems, and even attempting to have one model running on a Windows operating system while another is on a Linux or Solaris distribution in order to circumvent challenges due to platform-specific modeling tools.

Another future avenue of work is to make MODPI more generic in terms of accessing variables within an environmental systems model, so that the variable does not have to be explicitly accounted for in the model integration API, but can be obtained through reflection or searching the library symbol table. When performing generic model data access, though, there also needs to be a generic form of indexing multi-dimensional array variables, and a good manner of performing this generic within-array access has not yet been explored. As model data is passed from one model to the next, MODPI does not perform any generic data transformations (whether spatial, temporal, or other), which would lend to its strength and applicability as a model integration framework.

MODPI should automatically handle MPI calls without requiring a user to explicitly perform MPI calls. There may be cases where a model requires connection with multiple instances of another, as may the case be when connecting a 1-D model with a 2-D, spatially-distributed model. Thus, MODPI should add the capability to broadcast variables to multiple other processes. MODPI can and perhaps be implemented within other model integration standards essentially acting as a new component that gives more control and access to the more generic and well-developed model integration packages.

Speed of the interprocess communications between models within the MODPI framework in relation to process computation should be investigated in order to determine any bottlenecks and potential avenues to address network communication overhead. This could be done by testing speed differences between virtual machine execution and physical machine execution. Also, speeds should be compared between coupled models that were compiled together, that were decoupled using the publish-subscribe concept described above, and that were connected using MODPI. Foreseeable speed improvements could incorporate aggregation of all contiguous sends or receives into one larger network message. However, the novelty behind MODPI is not its speedup capability, which is possible where parallel computations are possible, but its abstraction of the model integration process for read and write data access during environmental or physical systems model simulation.

References

Becker, B. P. and Schuttrumpf, H. (2011). An OpenMI module for the groundwater flow simulation programme Feflow. *J. Hydroinformatics*, 13(1):1–12.

Curcin, V. and Ghanem, M. (2008). Scientific workflow systems-can one size fit all? In *Cairo Int. Biomed. Eng. Conf.*, Cairo, Egypt. IEEE.

Deelman, E., Gannon, D., Shields, M., and Taylor, I. (2009). Workflows and e-Science: An overview of workflow system features and capabilities. *Futur. Gener. Comput. Syst.*, 25(5):528–540.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open

MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc. 11th Eur. PVM/MPI Users' Gr. Meet.*, Budapest, Hungary.

Hill, C., DeLuca, C., Balaji, Suarez, M., and Da Silva, A. (2004). The architecture of the earth system modeling framework. *Comput. Sci. Eng.*, 6(1):18–28.

Laniak, G. F., Olchin, G., Goodall, J., Voinov, A., Hill, M., Glynn, P., Whelan, G., Geller, G., Quinn, N., Blind, M., Peckham, S., Reaney, S., Gaber, N., Kennedy, R., and Hughes, A. (2013). Integrated environmental modeling: A vision and roadmap for the future. *Environ. Model. Softw.*, 39:3–23.

Larson, J. W., Norris, B., Ong, E. T., Bernholdt, D. E., Drake, J. B., Elwasif, W. R., Ham, M. W., Rasmussen, C. E., Kumfert, G., Katz, D. S., Zhou, S., DeLuca, C., and Collins, N. S. (2004). Components, the Common Component Architecture, and the Climate/Weather/Ocean Community. In *84th Am. Meteorol. Soc. Annu. Meet.*

Matott, L. S., Babendreier, J. E., and Purucker, S. T. (2009). Evaluating uncertainty in integrated environmental models: A review of concepts and tools. *Water Resour. Res.*, 45(6):n/a–n/a.

Moore, R. V. and Tindall, C. I. (2005). An overview of the open modelling interface and environment (the OpenMI). *Environ. Sci. Policy*, 8(3):279–286.

Parton, W. J., Ojima, D. S., Cole, C. V., and Schimel, D. S. (1994). A general model for soil organic matter dynamics: sensitivity to litter chemistry, texture and management. In Bryant, R. B. and Arnoldm, R. W., editors, *Quant. Model. Soil Form. Process.*, pages 147–167. Soil Science Society of America, Madison, WI.

Šimnek, J., van Genuchten, M. T., and Šejna, M. (2008). Development and Applications of the HYDRUS and STANMOD Software Packages and Related Codes. *Vadose Zo. J.*, 7(2):587.

Yuan, F., Meixner, T., Fenn, M. E., and Šimnek, J. (2011). Impact of transient soil water simulation to estimated nitrogen leaching and emission at high- and low-deposition forest sites in Southern California. *J. Geophys. Res.*, 116(G3):G03040.