# MPH: a Library for Coupling Climate Component Models on Distributed Memory Architectures

Chris H.Q. Ding and Yun He
CRD Division, Lawrence Berkeley National Laboratory
University of California, Berkeley, CA 94720, USA

**Abstract**

A growing trend in developing large and complex applications on today's Teraflops computers is to integrate stand-alone and/or semi-independent program components into a comprehensive simulation package. We study how such a multi-component multi-executable application can effectively run on distributed memory architectures. We identify four effective execution modes and develop the MPH library to support application developments for utilizing these modes. MPH performs component-name registration, resource allocation and initial component handshaking in a flexible way.

Keywords: multi-component, multi-executable, climate modeling, problem solving environment.

## 1 Introduction

With rapid increase of computing powers of the distributed-memory computers, clusters of Symmetric Multi-Processors (SMP), the application problems also grow rapidly both in scale and complexity. Effectively organizing large and complex simulation programs such that it is maintainable, re-useable, shareable and high-performance at same time, becomes an important task for high performance computing.

Multiple component approach as a way to organize software is a natural evolution for many large scale simulations, such as climate modeling, engine combustion simulations, etc. For example, in modeling long-term global climate, NCAR's community climate system model (CCSM)[4] consists of an atmosphere model, an ocean model, a sea-ice model and a land surface model. These model components interact with each other through a flux coupler component.

Very often, program components of the application are developed by different groups in different organizations. Thus effective management of large scale software systems typically follows the modular approach, i.e, each program component is a self-consistent, semi-independent system. Each component talks to other components through a well defined interface and data structures involved in the interface. This approach allows maximum flexibility and independence. The developers of a particular component can use whatever the algorithm and method they see fit, depending on suitability, time to completion, practicality etc. This trend is well reflected in the software industry. The prominent example is CORBA [2]. Another development along this line within the high performance computing community is the Common Component Architecture (CCA) project [5].

In this paper, we describe the MPH library for coupling stand-alone and/or semi-independent program components into a comprehensive simulation system. The development of MPH for climate/weather modeling community is driven by this component software trend; MPH, in turn, further promotes this trend in climate model software developments.

There are other software development trends that emphasize completeness of the software system. Here we mention two popular types. A framework paradigm defines most common data and software structures and provides full-feature functionality, which goes much beyond pure interface. Some examples are PETSc [1], POOMA [15], ESMF [11], to name a few. Another type is the Problem Solving Environment, which essentially defines all the structures and skeleton codes for solving many different problems within a clearly defined special domain, such as Purdue PSEs [10], ASCI PSE [13], or even more focused on special area such as NWChem [9]. However, our approach is on developing complex simulation packages that utilize stand-alone or semi-independent components which are not necessarily developed by same group or institutions. Building a comprehensive application system utilizing (and modifing) existing codes developed by different groups is one of the standard development approaches.

## 2 Multi-Component Multi-Executable Applications on Distributed Memory Architectures

Here we provide a systemtical study on how a multi-component multi-executable application codes can effectively run on distributed memory architectures. This study forms the basis for the development of MPH.

There are two interelated aspects on a multi-component application codes running on a distributed memory computer: (1) how different components are integrated into a single application software structure; (2) the execution modes of the application system on distributed memory computers. Several new concepts on distributed multi-component systems are formalized.

First, we preserve the stand-alone or semi-independent nature of each component model. That is, these stand-alone models are independently compiled to its own binary executable file. Depending upon some runtime parameter setting, each component either do a stand-alone computation, or interact with other independent executables. Thus executables are the base units of a multi-component simulation systems. Executables are not allowed to overlap on processors, i.e, each processor or MPI process is exclusively owned by an executable. This is dictated by the processor sharing policy on most current HPC platforms.

Second, an executable may contain several program components. Different components may share a global address space. All components are written as modules and are finally merged into one single source code. They are compiled into a single executable. For example, an atmosphere circulation model may contain air convection dynamics, vertical radiation and clouds physics, land-surface modules, modules for chemical tracers such as $CO_2$, etc. Most current HPC applications are of this type. In these executables, different components may run on different processor subsets; Some of components may also share same processor subset.

Therefore, on distributed memory computers, a multi-component user application system may consist of several executables, each of them could contain a number of program components. In MPH, we systematically study the following different possible combinations.

(1) Single-Component *executable*, Single-Executable *application* (SCSE)

(2) Multi-Component *executable*, Single-Executable *application* (MCSE)

(3) Single-Component *executable*, Multi-Executable *application* (SCME)

(4) Multi-Component *executable*, Multi-Executable *application* (MCME)

(5) Multi-Instance *executable*, Multi-Executable *application* (MIME)

In the following, we discuss each of these modes in some details. All these modes are supported by MPH in a unified interface. Interfaces for each modes are discussed in Section 3.

## 2.1 Single-Component *Executable*, Single-Executable *Application* (SCSE)

This is the conventional mode. The complete program is a single component, and it is compiled into a single executable. We mention it for completeness.

## 2.2 Multi-Component *Executable*, Single-Executable *Application* (MCSE)

The entire application is contained in a single executable. Components may run on different processor subsets. Two or more components may also run on a same processor subset; They will run one after another, in a sequential fashion. The widely used Parallel Climate Model (PCM) [16] uses this mode.

All components are written as modules and are finally merged into one single source code. In this tight software integration mechanism, there are many programming issues associated with this approach. Name conflicts have to be resolved. Static allocation will increase unnecessary memory usage: component A on processor group A will still allocate memory for statical allocations in module component B which actually sits in processor group B. Data inputs and outputs also become more complicated. A large number of coordination must be done to ensure consistency, user interface flexibility, etc. Furthermore, if one needs to create a stand-alone version of the component, sufficient modifications (such as preprocessor ifdef etc) need to be inserted. The good feature of this approach is that the code is a single program, something everyone (including those with least programming experience) understands. The job launching process is also simplified greatly: it is just like any other normal program.

## 2.3 Single-Component *Executable*, Multi-Executable *Application* (SCME)

The entire application consists of several executables. Most if not all current HPC platforms adopts a resource allocation policy that does not allow two executables overlap on the same subset of processors. (On clusters of SMP architectures, it is allowed that two executables resides on the SMP node, each occupying different set of processors.)

Each executable contains a single program component. Inside the executable, there are flags to detect if the executable is running in a stand-alone mode or in a joint multi-executable environment. This integration mechanism allows maximum flexiblity in software developments. Different components can use different programming languages, different internal structures and conventions, etc. Different components do not even know the source codes of other components. They communicate with each other through a well defined common interface, which is the only constraint in development. CORBA is taking this approach. The first version of Climate System Model also uses this approach. One issue with this approach is the job launching process. On different vendor systems, the launching mechanisms vary slightly. But this is manangeable, since major HPC vendors are rather limited.

It is possible that the non-overlapping resource allocation policy can be modified. In that case, however, the entire load balance in both data distribution and task distribution of a parallel application will become questionable, because suddenly a processor (or an SMP node) will have another user job that takes CPU cycles and memory away in an entirely unpredictable way.

## 2.4 Multi-Component *Executable*, Multi-Executable *Application* (MCME)

The entire application consists of several executables, each of them contains several component programs. Different executables runs on different set of processors. Within each executable, different components may or may not overlap on processors. The number of processors allocated for each executable are determined by the multi-executable job launching commands. However, within each executable, processor allocation for each component is determined by the executable, not the job launching command. This is the comprehensive mechanism.

The component software integration for each executable is the same as in MCSE (Section 2.2).

### 2.5 Multi-Instance *Executable* for Ensemble simulations

Ensemble simulation is a new emerging trend in climate modeling for assessing the uncertainties in climate predictions. In ensemble simulations, identical codes are run multiple times, each time with a different set of input parameters. Conventional approach is to do the $K$ runs as $K$ independent jobs. The simulation results of the $K$ runs are then averaged to get ensemble avarage. It is sometimes advantageous to do the $K$ runs simultaneously: (a) nonlinear order statistics can be computed by aggregating instantaneous fields from $K$ runs periodically (b) based on current simulation results on the $K$ runs, the future simulation direction can be dynamically adjusted at real time. Nonlinear statistics and dynamical control cannot be done if the $K$ runs are performed as independent runs. MPH provides a convenient framework to do the ensemble simulations.

One may use MCME for ensemble simulations by compiling $K$ different executables with names "ocean-1", "ocean-2", $\cdots$, "ocean-K", etc. These executables have identical source codes, except the component names and input/output file names are different. However, maintaining $K$ executables and keeping track of the component input/output names of each executable increase the complexity and thus chances of errors of a large and extensive simulation. It is desirable to maintain only one executable, but different input/output names can be passed on to different ensembles.

## 3 MPH: Multiple Program Components Handshaking

We have identified the typical modes for multi-components multi-executable applications in the above section. One common critical issue in these modes is that when different executable images are loaded onto different processor subsets, one executable does not know the existance of other executables. Each processor only knowns its processor ID within the entire processors allocated for this potentially multi-executable applications.

For different executables to recognize each other, the only way is to assign a unique name to each executable as the identifier for that executable. We then require a handshaking process to set up a registry of executable names and communication channels. On tightly coupled HPC platforms, we use MPI communicators for high performance and portability.

A multi-component executable may contain several components, and therefore each component requires a unique component name. With careful examination of the necessary steps involved, it turns out that the "executable name" is not necessary for multi-component executables. Complete specification of names for all components within the multi-component executable is sufficient. Of course, the component name on a single-component executable is sufficient for identifying both the component and the executable. For these reasons, we use component names throught this paper, the corresponding executables are always clear by the context. For the same reason, we call this process "components handshaking", instead of "executables handshaking".

The MPH library is developed to handle this critical initial components handshaking and registration process in a distributed environment. MPH supports component name registration, resource allocation for each component, different execution modes as discussed in Section 2, and standard-out redirection.

One design goal of MPH is the complete flexibility. The number of components and executables, names of each components, processor allocation are all determined by a components registration file read in when the multi-executable job starts on different subsets of processors. One can trivially insert more components or delete some components from the application system. We found that this is one important feature in climate model developments.

In next section we describe MPH main interface and additional MPH functionalities. In Section 5, we explain the algorithms and implmentation of MPH. In Section 6, we discuss the status of MPH and current applications that have adopted MPH.

## 4 MPH Main Interface and Functionality

A unified interface is provided for all different software integration modes. Due to the variety and level of complexity, we explain the interface in each integration modes separately. This will also serve as concrete introduction to these new concepts of component integration.

One point to bear in mind is that for a multi-component executable, usually a master program is present that prepares and initiates different components on different (or overlapping) subsets of processors. For a single-component executable, such a master program does not exist.

### 4.1 Single-Component Executable, Multi-Executable Application (SCME)

In this mode, each component has a main program and is a complete stand-alone executable. Each component calls the shared handshaking routine with an input nametag and an output which is a MPI communicator.

For example, using the climate modeling system as the example. On atmosphere component, in the main program, we call

```
atmosphere_World = MPH_components (name1="atmosphere")
```

On ocean component,in the main program, we call

```
ocean_World = MPH_components(name1="ocean")
```

Similarly, for "land", "ice", and "coupler" components. The names of the components are registered in "registration.in" file. The order of file names are irrelevant.

```
BEGIN
atmosphere
ocean
land
ice
coupler
END
```

An important feature of MPH is that the nametag is for identifying a given component; its exact name is entirely arbitrary. One may use "NCAR_atm", or "UCLA_atm", or any other names for atmosphere component. The only necessary constraint here is that the nametags called in atmosphere component must appear correctly in the registration file. In this way, nothing is hardwired into the implementation. Suppose later, one has a need to insert a graphics component to produce a movie about the simulation, one can simply add the nametag of the graphics into the registration file.

### 4.2 Multi-Component executable, Single-Executable application (MCSE)

In this mechanism, each component is a subroutine or a module, but all codes are compiled into a single executable. A master program will call the appropriate subroutine on the appropriate subset of processors. In the master program, the following call is made first:

```
exe_world = MPH_components (name1="atmosphere",
                            name2="ocean", name3="coupler")
```

This setup routine informs MPH that there will be 3 components, with nametags "atmosphere", "ocean" and "coupler". Here again, nametags are arbitrary, except they must match the "registration.in" file that determines which processors are associated with which component.

Afterwards in the master program, we call

```
if(PROC_in_component("ocean", comm))      call ocean_xyz(comm)
if(PROC_in_component("atmosphere",comm)) call atmosphere(comm)
if(PROC_in_component("coupler",comm)) call coupler_abc(comm)
```

Note that subroutine names do not have be to same as the corresponding nametags. We use "_xyz", "_abc" etc to emphasize this fact.

The resource allocation "registration.in" is a user-supplied file. It contains the list of component nametags and processor ranges. For example, one sample registration file is

```
BEGIN
Multi_Component_Begin
atmosphere  0   15
ocean       16  31
coupler     32  35
Multi_Component_End
END
```

for 3 components on 36 processors (or MPI processes). Here `Multi_Component_Begin` and `Multi_Component_End` specify the start and end of a multi-component executable. In this registration file, no component overlaps with another on the same processor.

MPH allows components to overlap on their processor allocations. For example, in

```
BEGIN
Multi_Component_Begin
atmosphere   0    15
ocean       16    31
coupler      0    15  ! overlap with atmosphere
Multi_Component_End
END
```

The "atmosphere" component overlaps with "coupler" component. This feature allows more flexibility in code structure. It is users' responsibility to know who is overlapping with who else, and invoke components appropriately. One can use the logical function `PROC_in_component("ocean", ocean_comm)` to check if "ocean" covers this processor, and obtain the correct "ocean" communicator "ocean_comm". When sending to components on the overlapped processors, we recommend to use message tags to distinguish different components.

## 4.3 Multi-Component Multi-Executable Application (MCME)

This is the most flexible mode. Suppose we have following example contains 3 executables: 1st executable has 3 components: atmosphere, land, chemistry; 2nd executable has 2 components: ocean, ice; 3rd executable has a single component: coupler. Each component could contain up to 10 components.

On the atm-land-chem executable, we invoke MPH by

```
mpi_exec_world = MPH_components(name1="atmosphere",
                               name2="land", name3="chemistry")
```

On the ocean_ice executable, we invoke as

```
mpi_exec_world = MPH_components(name1="ocean",name2="ice"),
```

In the coupler.F file, the coupler component is invoked as

```
mpi_exec_world = MPH_components(name1="coupler")
```

The following registration file is used for this 3-executable problem:

```
BEGIN
Multi_Component_Begin
atmosphere 0  3
land       0  3   ! overlap with atmosphere
chemistry  4  7
Multi_Component_End
Multi_Component_Begin
ocean      0  15
ice        16 31
Multi_Component_End
coupler
END
```

The single-component executable with component `coupler` is listed directly. Within the first multi-component executable, atmosphere and land components overlap completely on processors allocations.

## 4.4 Multi-Instance Executable for Ensemble Simulations

Multi-instance executable is a special type of executables. It differs from regular single-component and multi-component executables in that this particular executable is replicated multiple times (multiple instances) on different processor subsets. There is no limit of the number of instances in such special executables.

A multi-instance executable is setup by envoking

```
Ocean_world = MPH_multi_instance("Ocean")
```

Note that the component name prefix "Ocean" determines that all instances of this executable must have component names using this prefix.

The number of instances and specific component names for these instances are specified in the runtime resource allocation/registration file. An example of 3 instances could look like this:

```
BEGIN
Multi_Instance_Begin
Ocean1     0  15  infile_1  outfile_1  logfile_1  alpha=3  debug=off
Ocean2    16  31  infile_2  outfile_2  beta=4.5   debug=on
Ocean3    32  47  infile_3  dynamics=finite_volume
Multi_Instance_End
statistics
END
```

Here `Multi_Instance_Begin` and `Multi_Instance_End` specify the start and end of a multi-instance executable.

Upon invocation of multi-instance executable, MPH replicate 3 instances of "Ocean" as 3 components, on the specified MPI processes. Each component will have the expanded component names (Ocean1, Ocean2, and Ocean3) as specified in the registration file.

In this registration file, a single-component executable with name "statistics" is also present. This executable is invoked as before (cf. Section 4.1); it collects instentaneous fields, compute statistics and controls evolution of each "Ocean" instances. Any other mix of single-component and/or multi-component executables may coexist with multi-instance executables.

Up to 5 character strings can be appended in the same line of the instance_name in the registration file. This is for passing input/output file names and parameters to the specific Ocean instance. MPH_Ensemble also provides several functions to get values for specific parameters. Examples are

```
alpha    = MPH_get_argument(int="alpha")
flag     = MPH_get_argument(char="debug")
filename = MPH_get_argument(field=1)
```

Thus `alpha` will get integer 3 if a string "alpha=3" is present, `flag` will get string "off" if a string "debug=off" is present, and `filename` will get string "infile_3" if a string "infile_3" is in the first field. This command line argument passage uses the function overloading feature of Fortran 90.

We suggest two examples where multi-instance-components are used. In a typical ensemble simulation example, 4 ocean ensembles are running concurrently using multi-instance executable, while a single-component executable is running simultaneously collecting statistics and controlling the evolution of different ensembles. In a global warming scenario simulation, 3 instances of an atmospheric model are running concurrently, each testing a different warming scenario with different $CO_2$ emmission rates, but all couple to the same ocean circulation model which feels the "average" effects of the atmosphere. The ocean model uses a multi-component executable.

## 5 Other MPH Functionalities

### 5.1 Joining Two Components

Besides solving the basic handshaking problem, MPH also provides a number of other functionalities for the ease of communication between components.

A joint communicator between any two components could be created by a call to

```
comm_new = MPH_comm_join ("atmosphere", "ocean")
```

The output `comm_new` communicator will contain all processors in both components, with processors in "atmosphere" component ranked first (rank 0 - 15) and processors in "ocean" component ranked second (rank 16 - 23) assuming atmosphere has 16 processors and ocean has 8 processors. If you reverse atmosphere with ocean in the call, then ocean processors will rank 0 - 7 and atmosphere processors will rank 8-23. With this joint communicator, collective operations such as a data redistribution could easily be performed.

### 5.2 Inter-Component Communications

MPI communication between local processors and remote processors (processors on other components) are invoked through component names and the local id. E.g., a processor on atmosphere wants to send Process 3 on ocean, it invokes

```
MPI_send(..., MPH_global_id("ocean", 3),MPH_Global_World,.....)
```

MPH_Global_World is the global communicator within this part of the application. It will be MPI_Comm_World for a simple multi-component application. The reason we did not use inter-communicator is because the entire application is assumed to run on a tighly coupled HPC computer with a single MPI_Comm_World. An inter-communicator would be more appropriate for a heterogeneous client-server environment, where CORBA or DCE are more widely used.

## 5.3 Inquiry on multi-component environment

MPH also provides a set of inquiry functions to get information about the multi-component environment. At run time, a component simply calls these subroutines to find out the processor configuration, component-name, etc. Some examples are:

```
MPH_local_proc_id()
MPH_global_proc_id()
MPH_comp_name()
MPH_total_components()
MPH_exe_up_proc_limit()
MPH_exe_low_proc_limit().
```

## 5.4 Multi-Channel Output

Suppose we have an application with five components running. Each component normally prints out messages by `print *, write(*)` for monitoring, control, diagnostics, and other purposes. If nothing special is done, all these messages sent to `stdout` will go to the session launching terminal. The mixed output would be extremely difficult to dicipher.

The ideal solution to this problem is for each component to write to its own output (log) file. In practice, however, there are a number of difficulties. First, file systems on different platforms are typically very different. Some of the parallel file system on the platform provides a "log" mode, i.e., writes from different processors will be bufferred and appended in some (random) order, such as PFS on Intel Paragon (without this "log" mode, in the usual "unformatted" mode, different writes could over-write each other and cause error conditions). In these cases, we need to modify these `print *, write(*)` statements and file `open` statements to achieve the desired effects. However, many existing components contain very large number of these statements which will be very time-consuming to modify. We need to find a way to do this automatically.

On many file systems, such as IBM SP's GPFS, there is no such "log" mode. Although MPI-IO [8] does support the "log" mode, the write statement syntax in MPIO is sufficiently different from `print *, write(*)` that makes a simple script-based automatic preprocessing difficult. (We emphasize here that the `stdout` on SP does support buffered I/O, similar to "log" mode; but it supports only one such I/O stream, not multiple `stdout` streams; that is the difficulty).

MPH resolves this difficulty by redirecting the `stdout`. Typically, local processor 0 of each component is responsible for print out messages. The `stdout`for this processor is redirected by

```
MPH_redirect_output(component_name)
```

and the output messages from each component will go to `component_name.log` file. All other occasional writes from all other processors are stored in one combined standard output file. The log file names of those components are defined by run time environment variables either in command line or in batch run script. This method is originally implemented in NCAR's CCSM code.

## 6 Algorithms and Implementation for MPH

Although most applications running on HPC platforms are single exectable codes, multi-executable jobs as discussed in this paper are still minority of applications. But multi-executable jobs are growing as the size and complexity of the "grand challenge" problems being solved on current large scale computers. It is important to understand this multi-executable job environment for the implementation of MPH.

Currently, all major HPC platforms support multi-executable jobs using an MPP-run like command. For example, on IBM SP, we use the MPMD mode, "-pgmmodel mpmd" to launch such a job. Different executables are specified in a command file using "-cmdfile". Similar commands exist for Compaq Alpha clusters, and SGI Origin (detailed launching commands for each plotform are described in details in test examples available online [6]).

Behind the seemingly different job launching commands on different platforms, the internal system environments are identical. When a job with $K$ executables starts on the specified processor domains, all executables share the same `MPI_Comm_World`, but with different logical processor IDs (MPI process IDs on cluster of SMP architetcures). How the processor IDs are assigned to each executable depends on the job launching commands. Since no executable can overlap on same processors, the processor ID assignments are unique.

However, at job start, no MPI communicator is formed for each executable, since each processor does not know what executables are loaded onto other processors. MPH establishes the multi-component environment multi-executable environment by first creating local communicators for each component. This task depends crucially on the fact that each component has a unique component-name provided by the run-time registration file.

It is important to distinguish executables from components. A single-component executable has one component, thus its communicator is unique. A multi-component executable has several components, and components could overlap on processor subsets. We first describe MPH implementation for single-component executables. Later we describe MPH implementation for multi-component executables.

### (1) Single-Component Executable Handshaking

Upon startup, the information in the registration file is read by the root processor (global Processor ID = 0) and broadcasts to all processors. Based on number of executables (number of components), each executable obtains a unique component_id. Using this component_id as "color", MPH calls `MPI_Comm_Split()` to split `MPI_Comm_World` into non-overlapping local communicators, each covering exactly the appropriate processor-subset for the component.

Once component communicators are established, information exchange between different components can be conveniently handled by the rank-0 processors in each component. Also, two components can be joined by merging their communicators.

### (2) Multi-Component Executable Handshaking

If the components within each executable are non-overlapping (on processors), all components can be established using a single invocation of `MPI_Comm_Split()` to split the current communicator for the executable into communicators one for each component.

MPH allows different components within an executable to be partially or completely overlapping on processors. (This allows a single unified user interface for all four software integration modes). In these cases, we create component communicators by repeatedly invoking `MPI_Comm_Split`, creating one component communicator at a time.

The codes are written in Fortran 90 for supporting CCSM development at present. We plan create a C++ version later.

## 7 Applications

The development of the MPH library is primarily motivated for NCAR Community Climate System Model (CCSM) development, as mentioned earlier. The large number of different components in CCSM, atmosphere, ocean, land, ice, flux coupler and many other potential components such as biochemistry, graphics for visulization, etc., require a general purpose handshaking library to setup the distributed multi-component environment.

MPH is an application driven software development. MPH version 1 is first developed for the single-component multi-executable mode (see Sections 2.3 and 3.3) for the CCSM model. MPH version 2 is then developed for the multi-component single-executable mode (see Sections 2.2 and 3.2) for the PCM model[16]. MPH version 3 is developed for the multi-component multi-executable mode (see Sections 2.4 and 3.4) to provide a unified user interface for MPH1 and MPH2. The multi-instance-component and the command line argument passing (discussed in Section 3.4) are currently being implemented to support climate ensemble simulations, a new emerging trend to ascertain the uncertainty in climate predictions.

All MPH funtionalities (except multi-instance-component in Section 3.4) are currently working on IBM SP, SGI Origin, Compaq AlphaSC, and Linux clusters. Source codes and instructions on how to compile and run on all these platforms are publicly available on our MPH web site [6].

MPH is currently been adopted in CCSM development[4]. CCSM is the U.S. flag-ship coupled climate model system most widely in long-term climate system modeling in the U.S. and in the world. MPH is adopted in NCAR's Weather Research and Forcast (WRF) model [17], which is the new generation of the mesoscale model (MM5) [14]. Several dozen countries use MM5 for their routine regional mid-range weather/climate forecasts. MPH is also adopted in Colorado State University's icosahedra grid coupled model[3]. Some others that show interest are: SGI for coupled model; a group in Germany, for coupled climate model; and a group in UK, for ensemble simulations. A Model Coupling Toolkit [12] for communication between different model components also uses MPH.

## 8 Summary and Discussions

We describe the rational, functionality and implementation of MPH for coupling stand-alone and/or semi-independent program components into a comprehensive simulation system. On today's Teraflop computers, as the problems been attacked become ever larger and complex, this application software development approach becomes necessary. The development of MPH for climate/weather modeling community is driven by this trend which in turn further promotes this trend.

We have systematically studied practical modes that a multi-executable application code can be effectively executed on current major HPC platforms. The resulting four modes are discussed in details in Sections 2 and 3. These form the basis that MPH is developed to support them by providing a simple, flexible and unified interface for coupling independent components together. With convenient MPH testing codes, compile/run scripts on all major plotforms, this work also promotes the use of the multi-component multi-executable approach in the climate modeling software developments.

We hope that the utilization of multi-component multi-executable approach for large and comprehensive appliocations described here will help HPC vendors to develop/implement more useful user interface.

**Acknowledgement.** MPH is developed in collaboration with Tony Craig, Brian Kauffman, Vince Wayland and Tom Bettge of National Center of Atmospheric Research, and Rob Jacobs and Jay Larson of Argonne National Laboratory. This work is supported by the U.S. Department of Energy, Office of Biological and Environmental Research, Climate Change Prediction Program, and Office of Computational and

Technology Research, Division of Mathematical, Information, and Computational Sciences, under contract number DE-AC03-76SF00098.

## References

[1] S. Balay, K. Buschelman, W.D.Gropp, D. Kaushik, L.C. McInnes and B.F. Smith. 2001. PETSc: Portable, Extensible Toolkit for Scientific Computation Homepage: http://www-fp.mcs.anl.gov/petsc/

[2] CORBA: Common Object Request Broker Architecture. http://www.corba.org/

[3] Colorado State University General Circulation Model. http://kiwi.atmos.colostate.edu/BUGS/

[4] Community Climate System Model. http://www.ccsm.ucar.edu/

[5] L. Curfman, D. Gannon, S. Kohn, C. Rasmussen, D. Bernholdt, J, Kohl, J. Nieplocha, R. Armstrong, S. Parker, etc. Common Component Architecture Forum. http://www.acl.lanl.gov/cca-forum/

[6] C. Ding and Yun He. MPH: a Library for Distributed Multi-Component Environment http://www.nersc.gov/research/SCG/acpi/MPH/

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, 1994. PVM: Parallel Virtual Machine, a User's Guide and Tutorial for Networked Parallel Computing. Scientific and Engineering Computaiton Series. The MIT Press, 279pp. See more info at http://www.epm.ornl.gov/pvm/

[8] W. Gropp, E. Lusk, and R. Thakur, 1999. Using MPI-2, published by MIT Press, 382pp. See more info at http://www.mpi-forum.org/

[9] High Performance Computational Chemistry Group, W.R. Wiley Environmental Molecular Sciences Laboratory, Pacific Northwest National Laboratory. NWChem computational chemistry package. http://www.emsl.pnl.gov:2080/docs/nwchem/

[10] E.N. Houstis, J.R. Rice, N. Ramakrishnan, T. Drashansky, S. Weerawarana, A. Joshi, and C.E. Houstis, 1998. Multidisciplinary Problem Solving Environments for Computational Science. Advances in Computers, Vol. 46 (M. Zelkowitz, ed.), Academic Press, 401–438. See more info about Purdue Problem Solving Environments at http://www.cs.purdue.edu/research/cse/pses/

[11] T. Killeen, J. Marshall, A. Silva, C. Hill, V. Balaji, and C. DeLuca, etc. Earth System Modeling Framework. http://www.esmf.ucar.edu/ http://sdcd.gsfc.nasa.gov/ESS/esmf_tasc/

[12] J.W. Larson, R.L. Jacob, I.T. Foster, and J. Guo, Model Coupling Toolkit, Argonne National Laboratory, Tech report. http://www-unix.mcs.anl.gov/ larson/mct.

[13] S. Louis, and J. May, etc. ASCI Problem Solving Environment. http://www.llnl.gov/asci/pse/

[14] PSU/NCAR Mesoscale Model. http://www.mmm.ucar.edu/mm5/

[15] J.V.W. Reynders, P.J. Hinker, J.C. Cummings, S.R. Atlas, S. Banerjee, W.F. Humphrey, S.R.Karmesin, K. Keahey, M. Srikant, and M. Tholburn, 1995. POOMA: A Framework for Scientific Simulation on Parallel Architectures, Supercomputing 95. See more info about POOMA: Parallel Object-Oriented Methods and Applications at http://www.acl.lanl.gov/pooma/

[16] W. Washington, J. Arblaster, T. Bettge, J. Meehl, G. Strand, and V. Wayland. Parallel Climate Model. http://www.cgd.ucar.edu/pcm/

[17] Weather Research and Forcasting (WRF) model. http://www.wrf-model.org/