Integrating Program Component Executables on Distributed Memory Architectures via MPH

Chris Ding and Yun He
Computational Research Division, Lawrence Berkeley National Laboratory
University of California, Berkeley, CA 94720, USA
chqding@lbl.gov, yhe@lbl.gov

Abstract

A growing trend in developing large and complex applications on today's Teraflop computers is to integrate stand-alone and/or semi-independent program components into a comprehensive simulation package. One example is the climate system model which consists of atmosphere, ocean, land-surface and sea-ice. Each component is semi-independent and has been developed at different institutions. We study how this multi-component multi-executable application can run effectively on distributed memory architectures. We identify five effective execution modes and develop the MPH library to support application developments utilizing these modes. MPH performs component-name registration, resource allocation and initial component handshaking in a flexible way.

Keywords: multi-component, multi-executable, component integration, climate modeling, distributed memory architecture.

1. Introduction

With the rapid increase in computing power of the distributed-memory computers, clusters of Symmetric Multi-Processors (SMP), the application problems also grow rapidly both in scale and complexity. Effectively organizing a large and complex simulation program such that it is maintainable, re-usable, sharable and efficient 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, and many others. For example, in modeling long-term global climate, the Community Climate System Model (CCSM)[5] consists of an atmosphere model, an ocean model, a sea-ice model and a land-surface model. These component models interact with each other through a flux coupler component.

Quite often, program components of a application are developed by different groups in different organizations. Effective management of large scale software systems typically follows the modular approach, in which each program component is a self-consistent, semi-independent system. Each component communicates with other components through a well defined interface. This approach allows maximum flexibility and independence. The developers of a particular component can use whichever algorithm and method they see fit, depending on suitability, running time, 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 [6].

On current distributed memory computers, it remains a cumbersome task for independent program components (or executables) to recognize each other. This is similar to finding and communicating to other processes under UNIX environment (imaging one submits several independent jobs under UNIX and the task is for each job to detect the existence of other jobs and communicate with them.) We refer to this task as handshaking. This crucial task enables stand-alone and/or semi-independent program components to function as a comprehensive application code/system.

In the following, we first examine several mechanisms that allow multiple components to be integrated into a single simulation system both in software integration and job execution. We provide several novel concepts in this direction (Section 2). In Section 3, we design the MPH library which facilitates the utilization of the above integration mechanism. Detailed functionality of MPH is explained in Section 4 and Section 5. Algorithms and implementations are discussed in Section 6, and applications in Section 7. Some related work is discussed in Section 8. Conclusion and discussions are given in Section 9.

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

In this paper, we define a program as an entire application with proper inputs and outputs. A program may consist of one or more executables (binary images); Each executable may consist of one or more components. Here a component is a semi-independent code segment with its own data and operations on them. Communication with different components is usually achieved through explicit data exchange, instead of accessing a common data area. Note that a component could be a rather large and complicated code. In the CCSM example, atmosphere, ocean, ice, and land are all components, each with its own data and associated functions performing relevant physical processes.

We provide a systematic study on how a multi-component multi-executable application code can effectively run on distributed memory architectures. This forms the basis for designing the software for the component handshaking process.

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) how the execution modes of the application system on distributed memory computers. Several new concepts on distributed multi-component systems are formalized here.

First, we preserve the stand-alone or semi-independent nature of each program component. That is, these stand-alone components 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 system. 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 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 which 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 mode 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 is compiled into a single executable. We mention it here 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 manner. The widely used Parallel Climate Model (PCM) [17] uses this mode.

All components are written as modules and are finally merged into one single source code. There are many programming issues associated with this tight software integration mechanism. Name conflicts have to be resolved. Static allocation will increase unnecessary memory usage. For example, 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 become more complicated. A large amount 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`) need to be inserted. The good feature of this approach is that the code is a single program, a practice that is familiar to most programmer including "beginners". The job launching process is also greatly simplified: it is merely launching an executable.

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

The entire application consists of several executables. Most, if not all, current HPC platforms adopt 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 reside on one 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 flexibility in software developments. Different components can use different programming languages, different internal structures and conventions, etc. Different components do not need to know the details of other components. They communicate with each other through a well defined common interface, which is the only constraint in development. CORBA takes 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 manageable, since there are only a few major HPC vendors.

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

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

The entire application consists of several executables, each of which contains several component programs. Different executables run on different set of processors. Within each executable, different components may or may not overlap on processors. The number of processors allocated to each executable is determined by the multi-executable job launching commands. However, within each executable, processor allocation per component is determined by the executable, not the job launching command. This is the most flexible and 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 treat the K runs as K independent jobs. The simulation results of the K runs are then averaged to get ensemble average. 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 simulation results on the current 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. The same executable is replicated multiple times (multiple instances) on different processor subsets. This enables running multiple ensembles simultaneously as a single job, and ensemble averaging being done on the fly. This eliminates large data output and storage for post-processing averaging, and enables nonlinear ensemble statistics which are otherwise impossible to compute at post-processing step.

One may use MCME for ensemble simulations by compiling K different executables with names such as "ocean-1", "ocean-2", ..., "ocean-K". These executables have identical source codes, except that 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 ensemble simulation. It is desirable to maintain only one executable, while different input/output names can be passed on to different runs in an ensemble.

3 MPH: Multiple Program-Component 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 is not aware of the existence of other executables. Each processor only knowns its own 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. We then require a process of handshaking 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, 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 throughout this paper. The corresponding executable is clear by the context. For the same reason, we call this process "component handshaking", instead of "executable 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 component registration file that is read in when the multi-executable job is launched on different subsets of processors. One can easily insert or delete components from the application system. We found that this is one important feature in climate model developments.

MPH has the following characteristics: (a) It allows flexible component names. As application code is developed, component models and their names evolve, e.g., the atmosphere model in CCSM changed from CCM to CAM. Component names cannot be hardwired into the coupler. (b) It includes several component integration mechanisms. In the previous coupler model, each component model is a single executable. In the related PCM (parallel climate model), each component model is a subroutine, and all program components are compiled into a single executable. As CCSM evolves, a component model could have several sub-components. Finally, ensemble simulations require yet another multi-instance mechanism. (c) It has flexible resource allocation. Processor allocation must be flexible and only need to be specified at runtime through a simple control mechanism. All these requirements are met by MPH. In addition, a number of further utilities are provided as well.

4 MPH Main Interface and Functionality

A unified interface is provided for all different software integration mechanisms (modes). Due to their varieties and different levels of complexity, we explain the interface in each integration mode separately. This will also serve as concrete introduction to these new concepts of component integration.

Since this is an application oriented software design, we outline some of the concrete main coding examples, both to help understand these new component integration modes and to demonstrate the ease of use of these interfaces. One point to bear in mind is that for a multi-component executable, a master program is usually needed to prepare and initiate different components on different (or overlapping) subsets of processors. Such a master program does not exist for a single-component executable.

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

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

Using the climate modeling system as an example, in the main program of atmosphere component, we call

```
atmosphere_World =                          &
   MPH_components_setup (name1="atmosphere")
```

It is similar for "ocean", "land", "ice", and "coupler" components. The names of the components are registered in "processors_map.in" file. The order of file names are irrelevant.

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

An important feature of MPH is that the name-tag is for identifying a given component; its actual 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 name-tags called in atmosphere component must appear correctly in the registration file. In this way, nothing is hardcoded into the implementation. Imaging that later on, one needs to insert a visualization component to produce a movie about the simulation, one can simply add the name-tag 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_setup (            &
            name1="atmosphere", name2="ocean", &
            name3="coupler")
```

This setup routine informs MPH that there will be 3 components, with name-tags ”atmosphere”, ”ocean” and ”coupler”. Here again, name-tags are arbitrary, except they must match the ”processors_map.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 the same as the corresponding name-tags. We use ”_xyz”, ”_abc” etc to emphasize this fact.

The resource allocation “processors_map.in” is a user-supplied file. It contains the list of component name-tags 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. This feature allows more flexibility in code structure. It is users’ responsibility to know what is overlapping with what 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 data 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 the following example that contains 3 executables: The 1st executable has 3 components: atmosphere, land, chemistry; the 2nd executable has 2 components: ocean, ice; the 3rd executable has a single component: coupler. Each executable could contain up to 10 components.

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

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

On the ocean-ice executable, the component is invoked as

```
mpi_exec_world = MPH_components_setup (  &
    name1="ocean",name2="ice")
```

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

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

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

```
BEGIN
Multi_Component_Begin  ! 1st multi-comp exec
atmosphere  0  15
land        0  15      ! overlap with atm
chemistry  16  19
Multi_Component_End
Multi_Component_Begin  ! 2nd multi-comp exec
ocean      0  15
ice       16 31
Multi_Component_End
coupler                ! a single-comp exec
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 executable. 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 this type of executables.

A multi-instance executable is setup by invoking

```
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   ! a multi-instance exec
Ocean1  0 15 inf1 outf1 logf alpha=3 debug=on
Ocean2 16 31 inf2 outf2 beta=4.5 debug=off
Ocean3 32 47 inf3 dynamics=finite_volume
Multi_Instance_End
statistics             ! a single-component exec
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 replicates 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 instantaneous fields, computes statistics and controls evolution of each "Ocean" instance. 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 to each line of the instance_name in the registration file. This is for passing input/output file names and parameters to the specific instances. MPH_Ensemble also provides a function interface to get values for specific parameters. Examples are

```
call MPH_get_argument("alpha",alpha2)
call MPH_get_argument("beta",beta)
call MPH_get_argument(field_num=1,field_val=fname)
```

Thus `alpha2` will get integer 3 if a string "alpha=3" is present, `beta` will get real 4.5 if a string "beta=4.5" is present, and `fname` will get string "infile3" if such a string is in the first field. This command line argument passage uses the function overloading feature of Fortran 90. It is worth to note that this parameter passing feature also works for the components of multi-component executbales.

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 emission 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 providing the basic handshaking, 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 one reverses "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 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. For example, if 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 tightly 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 decipher.

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 buffered 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 [9] 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 IBM SP does support buffered I/O, similar to "log" mode; but it supports only one such I/O stream, not multiple `stdout` streams for multi-executable jobs; 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 can be 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.

6 Algorithms and Implementation for MPH

Most applications currently running on HPC platforms are still single executable codes, so multi-executable jobs as discussed in this paper are at present a minority of applications. But the number of multi-executable jobs are increasing as the size and complexity of the "grand challenge" problems being solved on current large scale computers grow. 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 platform are described in details in test examples available online [7]).

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 architectures). 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, when a job is launched, besides the single global MPI communicator for `MPI_Comm_World`, no other MPI communicators are formed (for individual executables); each processor does not know which executables are loaded onto other processors. MPH establishes the multi-component 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 its components could overlap on processor subsets. We first describe MPH implementation for single-component executables. Later we describe it 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 broadcast to all processors. Based on the 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. Furthermore, 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 partially or completely overlap on processors. (This allows a single unified user interface for all five 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 to create a C++ version later.

7 Applications

The development of the MPH library is primarily motivated by the Community Climate System Model (CCSM) development, as mentioned earlier. The large number of different components in CCSM, including atmosphere, ocean, land, ice, flux coupler and many other potential components such as biochemistry, 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 for the multi-component single-executable mode (see Sections 2.2 and 3.2) for the PCM model. 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) is implemented in MPH version 4 to support climate ensemble simulations, a new emerging trend to ascertain the uncertainty in climate predictions.

Currently, all MPH functionalities are 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 [7].

MPH has been adopted in CCSM development[5]. CCSM is the U.S. flag-ship coupled climate model system most widely used in long-term climate system modeling in the U.S. MPH is adopted in NCAR's Weather Research and Forecast (WRF) model [18], the new generation of the mesoscale model (MM5) [15]. Many countries use MM5 for their regional mid-range weather/climate forecast. MPH is also adopted in Colorado State University's geodesic grid coupled model[3]. A Model Coupling Toolkit [13] for communication between different component models also uses MPH.

8 Generally Related Software systems

The development of MPH for climate/weather modeling community is driven by the component software trend as represented by CORBA and CCA. MPH, in turn, further promotes this trend in climate model and other scientific 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 [16], ESMF [12], and CACTUS [4] 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 [11], ASCI PSE [14], or even more focused on special area such as NWChem [10]. 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 institution. Building a comprehensive application system utilizing (and modifying) existing codes developed by different groups is one of the standard development approaches. MPH can be used as a part of these systems for multi-component/executable handshaking.

9 Summary and Discussions

We describe the rational, functionality and implementation of MPH for integrating stand-alone and/or semi-independent program components into a comprehensive simulation system. On today's Teraflop computers, as the problems being attacked become ever larger and complex, this 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 five 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 integrating independent program components together. With convenient MPH testing codes, compile/run scripts on all major platforms, this work also promotes the use of the multi-component multi-executable approach in the climate modeling software developments.

MPH handles the critical task of helping each stand-alone component-model executable to recognize the existence of other components within CCSM and getting neces-

sary processor information. MPH provides several possible independent component integration mechanisms. It allows component model processor geometries to be specified in a small input file. It also provides facilities for standard out redirection and joining of MPI communicators.

Some further work of component integration mechanisms of MPH are: (a) flexible way to handle SMP nodes, i.e., recognizing a 16-cpu SMP node could be carved into different number of MPI tasks; (b) dynamic component model processor allocation or migration; (c) an extension of MPH to do model integration over the grid; and (d) a C/C++ version of MPH.

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

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] The CACTUS Code Server. http://www.cactuscode.org/

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

[6] 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/

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

[8] 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 Computation Series. The MIT Press, 279pp. See more info at http://www.epm.ornl.gov/pvm/

[9] 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/

[10] 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/

[11] 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/

[12] 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/

[13] 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.

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

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

[16] 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/

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

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