Coupling Multi-Component Models with MPH on Distributed Memory Computer Architectures

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

Abstract

A growing trend in developing large and complex applications on today's Teraflop scale computers is to integrate stand-alone and/or semi-independent program components into a comprehensive simulation package. One example is the Community Climate System Model which consists of atmosphere, ocean, land-surface and sea-ice components. Each component is semi-independent and has been developed at a different institution. We study how this multi-component, multi-executable application can run effectively on distributed memory architectures. For the first time, we clearly identify five effective execution modes and develop the MPH library to support application development 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, distributed memory architecture, climate modeling, CCSM, PCM.

1 Introduction

With rapid increase in computing power of distributed-memory computers, and clusters of Symmetric Multi-Processors (SMP) application problems grow rapidly both in scale and complexity. Effectively organizing a large and complex simulation program so that it is maintainable, re-usable, sharable and efficient becomes an important task for high performance computing. Building a comprehensive application system utilizing (and modifying) existing codes developed by different groups is a standard development approach.

Component-based software engineering (CBSE) is an emerging trend for software development in both research and applications. Effective management of large-scale software systems typically follows the modular approach, in which each program component is a self-consistent, semi-independent system. In this context, "component" is defined as a unit of software program that has its own functionality. Each component communicates with other components through well-defined interfaces. It provides services through export interfaces, and uses other components' services through import interfaces. 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, and practicality etc.

Quite often, program components of many large-scale applications, such as climate modeling, engine combustion simulations, chemistry applications, optimization problems, and many others, are developed by different groups in different organizations. Each component has either distinct physics or utilizes a distinct numerical algorithm. For example, in modeling long-term global climate, the Community Climate System Model (CCSM)[10] 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 (Figure 1). The atmosphere model itself includes a dynamics subcomponent and a physics subcomponent.

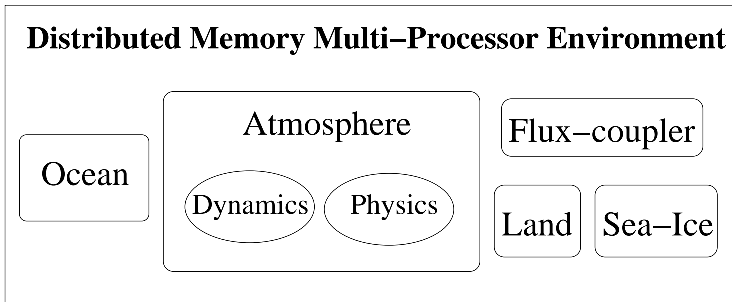

Figure 1: The schematic illustration of model components for a coupled climate model. Within MPH, any component can talk to any other component directly. In the Community Climate System Model (CCSM), all components communicate through a flux-coupler.

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 the UNIX environment (imagine 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.

Our work focus is on building a comprehensive computational code integrating several (semi-) independent domain-specific codes. The coupled climate system is the motivating example. The development of a MPH (Multi-Program Component Handshaking) library for the climate/weather modeling community is driven by the component software trend as represented by CORBA [8], and the Common Component Architecture (CCA) project [11, 2] (see Section 8 for details). MPH, in turn, further promotes this trend in climate model and other scientific software developments. MPH can be used as a part of these systems for multi-component/executable handshaking.

There are a large number of projects and software development to make parallel High Performance Computing (HPC) more suitable for component-based application code development. They are outlined in Section 8. Among these, NASA's Earth System Models Framework (ESMF) [22] is more closely related to the target application of MPH.

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 that facilitates the utilization of the above integration mechanism. Detailed functionalities of MPH are explained in Section 4 and Section 5. Algorithms and implementations are discussed

in Section 6, and applications in Section 7. Related work is presented in Section 8. Summary and discussion are given in Section 9. A preliminary result of this work was presented at the European Centre for Medium-Range Weather Forcasts (ECMWF) workshop on high performance computing[14].

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 functions upon 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 interrelated aspects of multi-component application codes running on a distributed memory computer: (1) How different components are integrated into a single application software structure; (2) What are 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 does a stand-alone computation, or interacts 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. 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 the 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 identify and 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 Aapplication (MIME)

In the following, we discuss each of these modes in some details. All of 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. For example, Parallel Climate Model (PCM) [31] 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 static 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.

Please see detailed examples of MCSE in Section 4.1.

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 to 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 a 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 development. Different components can use different programming languages, different internal structures, 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 development constraint. CORBA takes this approach. The first version of the Community 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.

Please see detailed examples of SCME in Section 4.2.

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. Both MCSE and SCME could be viewed as special cases of MCME.

The component software integration for each executable is the same as in MCSE (Section 2.2). Please see detailed examples of MCME in Section 4.3.

2.5 Multi-Instance Executable for Ensemble simulations (MIME)

Ensemble simulation is used frequently 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. The conventional approach is to treat K runs as K independent jobs. The simulation results of the K runs are then averaged to get an 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 in 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 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 as a 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.

Please see detailed examples of MIME in Section 4.4.

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, each executable is not aware of the existence of the other executables. Each processor only knows its own processor ID within the world of processors allocated for this potentially multi-executable application.

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 component 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 complete flexibility. The number of components and executables, names of each component, and 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 an important feature in climate model development.

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. In addition, a few further utilities are provided as well.

4 MPH Main Interface and Functionality

A unified interface is provided for all the 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 a concrete introduction to these new concepts of compo-

nent 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 need to exist for a single-component executable.

4.1 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. Using a climate system model as an example, 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 their processor allocations. This feature allows more flexibility in code structure. It is the 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 using message tags to distinguish different components.

4.2 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 returns the MPI communicator for the component.

For example, in the main program of an 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 is irrelevant.

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

An important feature of MPH is the name-tag 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 used in the atmosphere component must appear correctly in the registration file. In this way, nothing is hard-coded 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 component into the registration file.

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

This is the most flexible of the modes described in Section 2. 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.

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

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

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

```
      ocean_ice_world = MPH_components_setup(name1="ocean",name2="ice"),
```

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

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

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

```
BEGIN
Multi_Component_Begin  ! first multi-component executable
atmosphere 0  15
land       0  15       ! overlap with atmosphere
chemistry  16 19
Multi_Component_End
Multi_Component_Begin  ! second multi-component executable
ocean      0  15
ice        16 31
Multi_Component_End
coupler                ! a single-component executable
END
```

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

4.4 Multi-Instance Executable for Ensemble Simulations (MIME)

A 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 to the number of instances in this type of executable.

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 executable
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             ! a single-component executable
END
```

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

Upon invocation of the 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

```
statistics_world = MPH_components_setup(name1="statistics")
```

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 (separated by white spaces) can be appended at the end of each line of the instances in the registration file. This is for passing input/output file names and parameters to the specific instances. MPH 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=filename)
```

Thus `alpha2` will get the value integer 3 if a string "alpha=3" is present, `beta` will get the value real 4.5 if a string "beta=4.5" is present, and `filename` will get string "infile_3" 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 noting that this parameter passing feature also works for the components of multi-component executables.

We suggest two examples where multi-instance-components could be used. In a typical ensemble simulation example, 4 ocean ensembles are running concurrently using a 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

```
call MPH_comm_join("atmosphere", "ocean", comm_joined)
```

The output, `comm_joined` communicator, will contain all processors in both components, with processors in the "atmosphere" component ranked first (rank 0 - 15) and processors in the "ocean" component ranked second (rank 16 - 23) assuming the atmosphere has 16 processors and the ocean has 8 processors. If one reverses "atmosphere" with "ocean" in the call, then the ocean processors will rank 0 - 7 and the atmosphere processors will rank 8-23 in the `comm_joined` communicator. 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 data to ocean processor 3, 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 is the same as `MPI_Comm_World` for a simple multi-component application. The reason we did not use an 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 is more widely used.

5.3 Inquiry about 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():      find local processor id in a component.
MPH_global_proc_id():     find global processor id.
MPH_comp_name(cid):       find component name given component id.
MPH_total_components():   find number of total components.
MPH_exe_up_proc_limit():  find lower processor limit of a component
                          in the executable world given component id.
```

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 of these messages 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 systems provide a "log" mode, i.e., writes from different processors will be buffered and appended in some (random) order, such as Parallel File System (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 other file systems, such as IBM SP's General Parallel File System (GPFS), there is no such "log" mode. Although MPI-IO [17] 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 the 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.)

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

```
call 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 a command line or in a 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 presently a minority of applications. But the number of multi-executable applications are increasing as the size and complexity of the "Grand Challenge" problems 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 (Multi Program Multa Data) mode, "-pgmmodel mpmd" to launch such a job. Different executables are specified in a command file using "-cmdfile". Similar commands exist for HP AlphaServer SC and SGI Origin (detailed launching commands for each platform are described in details in test examples available online [13]).

Behind the seemingly different job launching commands on different platforms, the internal system environments are identical. When a job with K executables is launched on the specified SMP node and processor domains, all executables share the same `MPI_Comm_World`, but with different logical processor IDs (MPI process IDs on a 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 the same processors, the processor ID assignments are unique.

However, when a job is launched, only the single global MPI communicator for `MPI_Comm_World` is created, 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 compo-

nent. This task depends crucially on the fact that each component has a unique component-name provided by the run-time registration file, as explained in Section 4.

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 the 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 broadcast to all processors. Now every processor knows how many number of executables (number of components) in the entire application. Different executables differ because they have unique component "names". Each name is then mapped into a sequential number, a unique component_id. Now processors allocated to the same executable will obtain the same component_id. Thus, processors allocated to different executables will have different component_ids. 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 this case, we create component communicators by repeatedly invoking `MPI_Comm_Split`, creating one component communicator at a time. This order-C (C is the total number of components) strategy is partially dictated by the restriction for invoking `MPI_Comm_Split` that all processors must participate.

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. 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 was first developed for the

single-component multi-executable mode (see Sections 2.3 and 4.2) for the CCSM model. MPH version 2 was developed for the multi-component single-executable mode (see Sections 2.2 and 4.1) for the PCM model. MPH version 3 was developed for the multi-component multi-executable mode (see Sections 2.4 and 4.3) to provide a unified user interface for MPH1 and MPH2. The multi-instance-executable and the command line argument passing (discussed in Section 4.4) are implemented in MPH version 4 to support climate ensemble simulations to ascertain the uncertainty in climate predictions.

Currently, all MPH functionalities work on the IBM SP, SGI Origin, HP AlphaServer SC, 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 [13].

MPH has been adopted in CCSM development[10]. CCSM is the U.S. flagship coupled climate model system most widely used in long-term climate system modeling in the U.S. MPH has been adopted in NCAR's Weather Research and Forecast (WRF) model [32]. MPH is also used in the Colorado State University's geodesic grid coupled model[9]. A Model Coupling Toolkit [23] for communication between different component models uses MPH. Edinburgh Parallel Computing Centre (EPCC) at the University of Edinburgh uses MPH for ensemble simulations.

8 Related Work

Component-based software engineering (CBSE) trend is well reflected in the software industry. The prominent examples are Visual Basic [30], CORBA, COM [29], and Enterprise JavaBeans [15]. In the scientific high performance computing area, CCA provides a specification of a component environment. In its core implementation CCAFFEINE[1], CCA provides a light-weight programming model that supports the distributed memory single program multiple data (SPMD) mode. Identical frameworks containing the same component objects are instantiated on all processors. Different components on the same processor communicate with each other through interfaces/ports, and MPI communications are used for parallelism between the same component on different processors. Some other implementations are: Unitah [12], GrACE [27], CCAT [4], and XCAT [33].

Besides CCA, there are many domain-specific frameworks and problem solving environments (PSE) that emphasize completeness of the software system. Frameworks often have more assumptions about a particular structure or workflow for the specific application domain. For example, the ESMF project is funded to facilitate coupling earth system model components and to promote organization interoperability for the weather and climate community. Recently, there is increased collaboration between the ESMF and CCA communities [24, 34].

A framework paradigm defines most common data and software structures, a large set of commonly used numerical algorithms, and provides a full-featured functionality, which goes beyond pure interface. Some other examples are PETSc [3], POOMA [28], Overture [5], Hypre [7], and CACTUS [6], to name a few. Problem solving environments essentially define all the structures and skeleton codes for solving many different problems within a clearly defined special domain, such as Purdue PSEs [19], ASCI PSE [26], Jaco3 [20], and JULIUS [21] or even more narrowly focused on a special area such as NWChem [18]. Developing codes using tools from frameworks or PSEs is fast and straightforward, but the codes are no longer independent [25]. It becomes heavily and tightly coupled to the framework or PSEs.

In comparison with all these development, MPH aims at quick adaptation of existing MPMD capabilities of current major HPC platforms for component-based application development. MPH fills this critical gap before the more comprehensive software systems become widely accepted and adopted.

9 Summary and Discussion

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

We have systematically studied the practical modes of a multi-executable application code that can be effectively executed on current major HPC platforms. The resulting five modes of execution are discussed in detail in Sections 2 and 3. MPH is developed to support all the modes 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 climate modeling software development.

MPH handles the critical task of helping each stand-alone component-model executable to recognize the existence of other components within CCSM and getting necessary 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. MPH is only needed at the initial setup, and its overhead is trivial.

Some further work for component integration mechanisms of MPH are: (a) 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) a 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 the large and comprehensive applications described here will help motivate HPC vendors to develop/implement more useful user interfaces for this type of application.

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 SciDAC Climate Project funded 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. Most of the code development work are performed on the NERSC (National Energy Research Scientific Computing Center) IBM SP machine at the Lawrence Berleley National Laboratory.

References

[1] B.A. Allan, R.C. Armstrong, A.P. Wolfe, J.Ray, D.E. Bernholdt, and J.A Kohl. The CCA Core Specification in a Distributed Memory SPMD Framework, Journal of Concurrency Computatation: Practice and Experience, 2002(14),1-23.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker and B. Smolenski. Towards a Common Component Architecture for High-Performance Scientific Computing. Eight IEEE International Symposium on High Perfomance Distributed Computing, August 1999.

[3] 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/

[4] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko and M. Yechuri. A Component Based Services Architecture for Building Distributed Applications. Proceedings High Performance Distributed Computing Conference, 2000.

[5] Brown, D., Chesshire, G., Henshaw, W., and Quinlan, D., OVERTURE: An Object-Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments. Proceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997.

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

[7] E. Chow, A. Cleary and R. Falgout. Design of the HYPRE Preconditioner Library. In Object Oriented Methods for Interoperable Scientific and Engineering Computing, M. E. Henderson, C. R. Anderson and S. L. Lyons, SIAM, Philadelphia, PA, 1999.

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

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

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

[11] Common Component Architecture Forum. http://www.cca-forum.org/

[12] J. Davison de St. Germain, J. McCorquodale, S.G. Parker, C.R. Johnson. Unitah: A Massively Parallel Problem Solving Environment. HPDC'00 : Ninth IEEE International Symposium on High Performance and Distributed Computing, August 2000.

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

[14] C. Ding and Y. He. Climate MOdeling: Coupling Component Models by MPH for Distributed Multi-Component Environment. *Realizing Teracomputing, Proceedings of the Tenth Workshop on the Use of High Performance Computing in Meteorology*, European Centre for Medium-Range Weather Forecasts, Reading, England, November 2002.

[15] R. Englander and M. Loukides. *Developing Java Beans (Java Series)*. O'Riley Associates, 1997. http://www.java.sun.com/products/javabeans.

[16] 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/

[17] 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/

[18] 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/

[19] 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/

[20] Jaco3: Industrial Design PSE http://www.arttic.com/projects/jaco3

[21] PSE for Engineering Simulations. http://www.6s.org

[22] 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/

[23] 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.

[24] J.W. Larson, B. Norris, E.T. Ong, D.E. Bernholdt, J.B. Drake, W.R. El Wasif, M.W. Ham, C.E. Rasmussen, G. Kumfert, D.S. Katz, S. Zhou, C. Deluca, and N. S. Collins, Components, the Common Component Architecture, and the climate/ocean/weather community. 20th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology, January 2004, seattle, WA.

[25] S. Lefantzi, J. Ray, and H. N. Najim. Using the Common Componentn Architexture to Design High Performance Scientific Simulation Codes. Proceedings of International Parallel and Distributed Processing Symposium, 2003.

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

[27] M. Parashar, J.C. Browne et al. A Common Data Management Infrastructure for Parallel Adaptive Algorithms for PDE Solutions. Proceedings of Supercomputing'97, November 1997.

[28] 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/

[29] R. Sessions. *COM and DCOM: Microsoft's vision for Distributed Objects*. John Wiley & Sons, 1997.

[30] Visual Basic webpage. http://msdn.microsoft.com/vbasic.

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

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

[33] Xcat homepage. http://www.extreme.indiana.edu/ccat/. Also http://www.extreme.indiana.edu/xcat/

[34] S. Zhou. Using Earth System Modeling Framework and Common Component Architecture to Couple Models in Weather and Climate. 20th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology, January 2004, seattle, WA.