MPH: a Library for Distributed Multi-Component Environment

Chris Ding and Yun He
NERSC Division, Lawrence Berkeley National Laboratory
University of California, Berkeley, CA 94720, USA

Abstract

Many current large and complex HPC applications are based on semi-independent program components developed by different groups or for different purposes. On distributed memory parallel supercomputers, how to perform component-name registration and initialize communications between independent components are among the first critical steps in establishing a distributed multi-component environment. Here we describe MPH, a multi-component handshaking library that resolves these tasks in a convenient and consistent way. MPH uses MPI for high performance and supports many PVM functionality. It supports two major parallel integration mechanism: multi-component multi-executable (MCME) and multi-component single-executable (MCME). It is a simple, easy-to-use module for developing practical codes, or as basis for larger software tools/frameworks.

Keywords: multi-component multi-executable, multi-component single-executable, distributed environment, software integration

1 Introduction

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

Multiple component approach as a way to organize software is a natural evolution for many large scale simulations, such as climate modeling, engine combustion simulations, etc. For example, in modeling long-term global climate, NCAR's community climate system model (CCSM)[3] consists of an atmosphere model, an ocean model, a sea-ice model and land surface model. These model components interact with each other through a flux coupler component.

Very often, program components of the simulation system are developed by different groups in different organizations. Thus effective management of large scale software systems typically follows the modular approach, i.e, each program component is a self-consistent, semi-independent system. Each component talks to other components through a well defined interface and data structures involved in the interface. This approach allows maximum flexibility and independence. The developers of a particular component can use whatever the algorithm and method they see fit, depending on suitability,

time to completion, practicality etc. This trend is well reflected in the software industry. The prominent example is CORBA [4]. Another development along this line within the high performance computing community is the Common Component Architecture (CCA) project [2].

We also note that there are other software development trends that emphasize completeness of the software system. Here we mention two popular types. A framework paradiagm defines most common data and software structures and provide full-feature functionality, which goes much beyond pure interface. Some examples are PETSc [12], POOMA [13], ESMF [5], 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 [14], ASCI PSE [1], or even more focused on special area such as NWChem [9]. However, our goal here is on developing complex simulation packages that utilizes semi-independent components which are developed by different groups.

Here we emphasize modularity in software developments, and therefore the (semi)independence of program components. Irrespect to components, object-oriented programming should be dopted every, especially within each program component (which could still be a large system such as an ocean model with up to 100,000 lines of Fortran codes).

When this multiple components approach is used in developing applications on distributed memory computers, different components often live on different subsets of processors. How to initially identify each component to all other components, i.e., a handshaking process is necessary to set a registry of components and communication channels (we deal with MPI communicators for high performance and portability).

Of course, it is possible to hardwire the codes so that each component knows who else would be present in the system at codes compilation time, as has been done in practice (CCSM, for example). However, this is highly restrictive. What if a new component is needed or an existing one is not needed? What if a component's name is changed? How many processors are allocated for each component? This hardwired approach is also non-standard — it is not easy to be adopted for other applications.

In this paper, we describe MPH, a library implemented as a Fortran 90 module, that handles this initial components handshaking and registration process in a distributed environment. It is flexible: the number of components, names of each components are all determined by a components registration file read in when the executables start on different subsets of processors. In some sense, MPH's main role is similar to the functionality provided by PVM [11], except its interface and usage is much simpler, the codes are much smaller (a few hundred lines, publically available online [7]). MPH also supports two software integration mechanism: multi-component multi-executable, and multi-component single-executable, with processor overlapping or non-overlapping. Furthermore MPH implementation scales up as log of number of processors, instead of linear in processors in PVM, particularly useful for large scale problems. MPH also promotes a certain number of programing styles that help making software more portable and re-useable.

2 A distributed multi-component environment

We focus on developing large scale simulation systems consists of (semi)independent components running on distributed memory HPC platforms. For this type of tightly integrated systems, we use MPI for

high performance and portability. MPH provides most necessary functionality and support for this multi-component environment.

MPH supports component name registration, resource allocation for each component, different software integration mechanism, standard-out redirection, etc., with complete flexibility.

Using multi-component approach in codes development for distributed memory computers will lead to two different components integration mechanism which leads to different job execution modes:

(1) Multi-Component Multi-Executable (MCME)

(2) Multi-Component Single-Executable (MCSE)

We discuss them in some details.

2.1 MCME mechanism

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

Note that in MCME, no component overlap on subset of processors. With multiple executable images, components overlap is not feasible in most parallel computer systems todays, because in this integration mechanism, each processor group would have two separate user executables running at same time. On most distributed systems, each node is dedicated to the user who first occupies it, no other users or second job of the same user is allowed. It is possible that this resource allocation policy can be modified. In that case, however, the entire load balance in both data distribution and task distribution of a parallel application will become questionable, because suddenly a processor (or a SMP node) will have another user job that takes CPU cycles and memory away in an entirely unpredictable way.

2.2 MCSE mechanism

All components are written as modules and are finally merged into one single source codes. In this tight software integration mechanism, there are many programing issues associated with this approach. Name conflicts has to be resolved. Static allocation will increase unnecessary memory usage: component A on processor group A will still allocate memory for statical allocations in module component B which actually sits in processor group B. Data inputs and outputs also becomes more complicated. A large number of coordination must be done to ensure consistency, user interface flexibility, etc. Furthermore, if one needs to create a standard-alone version of the component, sufficient modifications (such as preprocessor ifdef etc) needs to be inserted. The good feature of this approach is that the codes is a

single program, something everyone (including those with least programming experience) understand. The job launching process is also simplified greatly: it is just like any other normal program.

Note that in MCSE, different components may overlap, i.e., two different components could run on the same set of processors. they will run one after another, in a sequential fashion. This integration mechanism allow more flexibility. For example, the parallel climate model [10] uses this integration mechanism.

3 Interface and Functionality

3.1 MPH setup

Because different execution integration mechanism run differently, different MPH invocation procedures are necessary. For example, in Single Excutation integration mechanism, there is a master program that prepares and initiate different components on different (or overlapping) subsets of processors, whereas no master program exists for Multiple Executable integration mechanism. Other differences also exist.

Despite these differences, the calling procedures can be designed to be very similar. We describe them below.

(1) Multi-Component Multi-Executable (MCME)

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

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

```
call MPH_setup ("atmosphere", atmosphere_World)
```

On ocean component,in the main program, we call

```
call MPH_setup ("ocean", ocean_World)
```

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

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

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

(2) Multi-Component Single-Executable (MCSE)

In this integration mechanism, each component is a subroutine. but all subroutines 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 MPH_set is called:

```
call MPH_setup_SE(
        "atmosphere",     ! "atmosphere" is present
        "ocean",          ! "ocean" is present
        "coupler"         ! "coupler"  is present
   )        ! You can add more components here.
```

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

Afterwards in the master program, we call

```
if(PE_in_component("ocean", comm))     call ocean_v1(comm)
if(PE_in_component("atmosphere",comm)) call atmosphere_v2(comm)
if(PE_in_component("coupler",comm)) call coupler_v3(comm)
```

Note that subroutine names do not have be to same as the corresponding nametags. We use "_v1", "_v2" etc to emphasize this fact.

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

```
PROCESSOR_MAP
BEGIN
atmosphere  0  15
ocean       16 30
coupler     31 32
land        33 35
ice         36 39
END
```

for 5 components on 40 processors. In this "processor.map" file, no component overlap with another on the same processor.

But MPH allows components to overlap on their processor allocations. The following is a legitimate processor.map file for 64 processors:

```
atmosphere  0    23
land        0    23   ! overlap with atmoshpere
coupler     24   29
biosphere   30   31
ocean       32   63
ice         32   63    ! overlap with ocean
```

It is users' responsibility to know who is overlapping with who else, and invoke components appropriately. One can always use the logical function `PE_in_component("ocean", ocean_comm)` to check if "ocean" covers this processor, and obtain the correct "ocean" communicator "ocean_comm".

A simpler version of `MPH_setup` for MCSE is also provided for the case: (1) the master program simply launches separate components on separate subset of nodes. (2) Each component remains on the same subset of nodes during entire computation. This situation is very similar to the MCME case, the only exception is that each component is now a module (or a subroutine), instead of a separate executable image. This case allows a much simpler interface, although it can also be accomplished using `MPH_setup_SE()` in above. The master program calls the following:

```
      call MPH_setup_MCSE(
     &      atmosphere = atm_subroutine,    ! on atmosphere processors
     &      ocean      = ocean_subroutine,  ! on ocean processors
     &      coupler    = coupler_subroutine,! on coupler processors
            communicater = MyWorld) ! get proper communicator on this proc
```

Here "atmosphere", "ocean", "coupler", "communicater" are KEYWORDS. Additional keywords "sea_ice", "land", "biosphere", "io" are supported. Order of names are irrelevant. Components are all optional: You may invoke "ocean=ocean_v1, ice=ice_v3" only. Inside `MPH_setup_MCSE`, different subroutines, `atm_subroutine`, `ocean_subroutine`, `ice_subroutine` are called on different processors specified in "processor.map" file.

3.2 Joining two components

Besides solving the basic handshaking problem, MPH also provide 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

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

The output `Comm-new` communicator will contain all processors in both components, with processors in "atmosphere" component ranked first (rank 0 - 15) and processors in "ocean" component ranked second (rank 16 - 23) assuming atmosphere has 16 processors and ocean has 8 processors. If you reverse

atmosphere with ocean in the call, then ocean processors will rank 0 - 7 and atmosphere processors will rank 8-23. With this joint communicator, collective operations such as a data redistribution could easily be performed.

3.3 Inter-component communications

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

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

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

3.4 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_component_name()
MPH_total_components()
MPH_up_proc_limit()
MPH_low_proc_limit().
```

3.5 Standard Output

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

The ideal solution to this problem is for each component to write to its own output (log) file. In practice, however, there are a number of difficulties. First, file systems on different platforms are typically very different. Some of the parallel file system on the platform provides a "log" mode, i.e., writes from different processors will be bufferred and appended in some (random) order, such as PFS on Intel Paragon (without this "log" mode, in the usual "unformatted" mode, different writes could

over-write each other and cause error conditions). In these cases, we need to modify the 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 find a way to do this automatically.

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

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

```
MPH_redirect_output(component_name)
```

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

4 Implementation

A design goal for MPH is to scale to large number of nodes and large number of components. For this purpose, we implement MPH using a two-stage fan-in-fan-out scheme, with a timing complexity proportional to log(P) (P is the total number of nodes).

It is important to note that on most current HPC platforms, with K components in K executables, when components start, they all share the same `MPI_Comm_World`, but with different logical processor ids. No local MPI communicator exists for each component. MPH establish the multi-component environment by first creating local communicators for each component. We use `MPI_Comm_Split` to split `MPI_Comm_world` into non-overlap local communicators, making use of the fact each component has a unique component-name provided by the run-time registration file.

To facilitate information exchange, we create another communicator `COMM_master`, which consists of K processor, one from each component. Every processors in `COMM_master` are ranked 0 in their own component. With these two layered communicators, all relevant information can be gathered and exchanged using fan-in-fan-out method. For example, the global processor ids of a component are first fan-in to master processor (rank 0) on each component using the local communicator; then fan-in again to the global master processor using `COMM_master`. All these information are then broadcasted (fan-out) first within masters (using `COMM_master`), and then within each component. Note that processor ids of each component are not required to be consecutive. This adds some flexibility in processor allocations.

As mentioned earlier, in MCSE mode, different components are allowed to overlap on processors. In these cases, the above implementation fails since it requires each processor belonging to one and one component only. In this integration mechanism, we create the local communicators in a different

algorithm. In this algorithm we loop over all components and repeatedly create the appropriate local communicator one at a time, instead of creating them simultaneously using a simple `MPI_Comm_Split`. Most of other codes remain same.

The codes are written in Fortran 90 (for suppporting CCSM development at present. We will create a C++ version later). A separate module for each integration/execution mode. All three integration mechanism have slightly different invocation mechanism (or interfaces), but share all lower-level building blocks, and same utilities. For convenience, we also provide a combined implmentation of three integration/execution modes with three separate communication modules included in one file "mph.F', one for each mode.

MPH is currently working on IBM SP, SGI Origin, Compaq AlphaSC and Cray T3E. Source codes and instructions on how to compile and run on all these platforms are publicly available on our MPH web site [7].

5 Applications

The development of the MPH library is primarily motivated for NCAR Community Climate System Model (CCSM) development, as mentioned earlier. The large number of different components in CCSM, atmosphere, ocean, land, ice, flux coupler and many other potential components such as biochemistry, graphics for visulization, etc., requries a general purpose handshaking libarary to setup the distributed multi-component environment. MPH is currently used in CCSM development. A larger scale utility Model Coupling Toolkit [6] uses MPH for this purpose.

An important decision in using the multi-component paradiagm in an application codes is to choose between the MCSE or MCME integration/execution mechanisms. This is largely determined by how the component programs are developed, and if a tight MCSE integration is necessary. Fortunately, with MPH, the coding efforts required to switch between different integration mechanism are much reduced.

Another design goal of MPH is that codes for different integration mechanism could co-exist in a single program, and easily switch between different integration mechanism using `#ifdef` CPP preprocessor options.

In our online distribution, we have a complete coding example of 3-component application. The three component programs "atm.F", "ocean.F" and "coupler.F" contains the source codes which can be easily converted to/from stand-alone, MCME or MCSE with 3-4 lines changes. Note that a master program (the main program) is needed for MCSE.

A by-product of MPH is that it strongly promote a good programming style which emphasizes modularity and codes re-useability. For exmaple, in any component, a user-defined communicator `MPI_my_world` is mandatory, no `MPI_Comm_World` is allowed. This way, the component can use any communicator from a MPH-type multi-component environment without a single line change of codes. Another example is data redistribution, say between two components. A single redistribution tool can be written that assumes all processors inside a single communicator with processors ranked from 0 to P-1. Using `MPH_comm_join` to join the two relevant components, the redistribution tool can be directly invoked on the joint communicator. to perform the redistribution task without any code change.

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 Laborotory.

References

[1] ASCI Problem Solving Environment. http://www.llnl.gov/asci/pse/

[2] Common Component Architecture Forum. http://www.acl.lanl.gov/cca-forum/

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

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

[5] Earth System Modeling Framework. http://sdcd.gsfc.nasa.gov/ESS/esmf_tasc/

[6] 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.

[7] Multi Program-Components Handshaking Library. http://www.nersc.gov/research/SCG/acpi/MPH/

[8] Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/

[9] NWChem computational chemistry package. http://www.emsl.pnl.gov:2080/docs/nwchem/nwchem.html

[10] Parallel Climate Model. http://www.cgd.ucar.edu/pcm/

[11] Parallel Virtual Machine. http://www.epm.ornl.gov/pvm/pvm_home.html.

[12] PETSc: Portable, Extensible Toolkit for Scientific Computation. http://www-fp.mcs.anl.gov/petsc/

[13] POOMA: Parallel Object-Oriented Methods and Applications. http://www.acl.lanl.gov/pooma/

[14] Problem Solving Environments. http://www-cgi.cs.purdue.edu/cgi-bin/acc/pses.cgi