Implementing Multidisciplinary and Multi-zonal Applications Using MPI

Samuel A. Fineberg[1]

Report NAS-95-003 January 1995

Computer Sciences Corporation
Numerical Aerodynamic Simulation
NASA Ames Research Center, M/S 258-6
Moffett Field, CA 94035-1000
(415)604-4319
e-mail: fineberg@nas.nasa.gov

Abstract

Multidisciplinary and *multi-zonal* applications are an important class of applications in the area of Computational Aerosciences. In these codes, two or more distinct parallel programs or copies of a single program are utilized to model a single problem. To support such applications, it is common to use a programming model where a program is divided into several single program multiple data stream (SPMD) applications, each of which solves the equations for a single physical discipline or grid zone. These SPMD applications are then bound together to form a single *multidisciplinary* or *multi-zonal* program in which the constituent parts communicate via point-to-point message passing routines. Unfortunately, simple message passing models, like Intel's NX library, only allow point-to-point and global communication within a single system-defined partition. This makes implementation of these applications quite difficult, if not impossible. In this report it is shown that the new Message Passing Interface (MPI) standard is a viable portable library for implementing the message passing portion of multidisciplinary applications. Further, with the extension of a portable loader, fully portable multidisciplinary application programs can be developed. Finally, the performance of MPI is compared to that of some native message passing libraries. This comparison shows that MPI can be implemented to deliver performance commensurate with native message passing libraries.

---

1. This work was supported through NASA contract NAS 2-12961.

1.0 Introduction and Background

Multidisciplinary and *multi-zonal* applications are an important class of programs in the area of Computational Aerosciences. In these codes, two or more distinct parallel applications or copies of a single application are utilized to model a single problem [BaW93]. To support such programs, it is common to use a programming model where an application is divided into several single program multiple data stream (SPMD) applications, each of which solves the equations for a single physical discipline or a particular portion of a data set (i.e., a grid zone). These SPMD applications are then bound together to form a single *multidisciplinary* or *multi--zonal* program in which the constituent parts communicate via point-to-point message passing routines. Unfortunately, simple message passing models, like Intel's message passing library (NX) or Thinking Machines' message passing library (CMMD), only allow point-to-point and global communication within a single system-defined partition. This makes implementation of multidisciplinary applications quite difficult, if not impossible.

Several non-portable libraries have been implemented to solve this problem. These include the intercube library for the iPSC/860 [Bar91] and the Map library for the Paragon [Fin93c]. Neither of these solutions allow a single source code to be used across multiple systems. To develop portable multidisciplinary programs, there are several requirements. First, one must have a portable message passing library that is capable of supporting multiple process[1] groups, collective communication within process groups, and inter-group communication. Second, one must have a portable loader that is capable of starting multiple, possibly different, programs as a single multidisciplinary application. Finally, this loader must have some way of telling the programs it has loaded about where the different applications reside. Otherwise it would be impossible to communicate between applications.

There are, of course, quite a few portable message passing libraries. Of these, several provide the support necessary for multidisciplinary process groups and collective communication. Two of these are PVM [GeS91, GeB93] and MPI [Mes94]. These libraries are available for most MPP systems as well as for networks of workstations. MPI was chosen as the preferable message passing library for several reasons. First, while MPI is still new, it is a standard. Therefore, it is not expected to undergo the constant changes that other libraries, most notably PVM, suffer from. In addition, from a performance perspective, MPI should perform better on MPP systems than PVM. This is primarily due MPI's statically defined group structures, and its ability to be implemented without buffering. While these factors should enable MPI to perform better than PVM, MPI will still be worse than native libraries until it is directly supported by vendors. To date, only IBM Research has provided a vendor optimized version of MPI. This version is still experimental, but it shows promise because its performance is as good or better than IBM's proprietary message passing library. Supported vendor implementa-

1. In this paper, the term "process" will be used instead of "processor" or "node." This refers to the fact that more than one MPI "process" may be present on a single processor of a parallel system.

tions of MPI should begin to appear in the coming year and hopefully will begin to replace vendor specific libraries. For more information on performance issues see Section 5.

Portable loaders, however, are far more difficult to find. PVM does provide program loading facilities, and does support multiple executables within a single job. However, it does not provide a portable means for determining where applications have been loaded. MPI does not provide any loading facilities, therefore, all loading must be done using means external to MPI. In this paper, a portable loader interface, MPIRUN, is described. MPIRUN may be implemented on virtually any MPP system or workstation network, and it is simpler than the loader provided by PVM. This simplicity makes it far easier to integrate MPIRUN with existing resource allocation and scheduling software. Finally, MPIRUN not only loads user programs, but also provides run-time loading information needed to initiate inter--application communication.

2.0 MPI Basics

MPI has several features that make it ideal for multidisciplinary program development. In this section some MPI basics will be presented, followed with the advanced features necessary for multidisciplinary and multi-zonal applications. This paper assumes that the reader has knowledge of some other message passing library, e.g., NX, CMMD, etc., and many of the details are left to the reader. For a complete specification of MPI see the standard [Mes94].

2.1 Basic send and receive operations

MPI provides a vast array of communication operations. Unfortunately, since the only guide to writing MPI programs to date is the standard [Mes94], one can easily become daunted by the amount of functionality provided by MPI. However, for most programs one can ignore most of these features. For simple point to point message passing most users can and should stick with the basic `MPI_Send` and `MPI_Recv` synchronous send/receive operations. These basic operations are analogous to the `csend` operation in NX or the `CMMD_send_block` in CMMD. MPI is also capable of performing asynchronous message passing, using the `MPI_Isend and MPI_Irecv` operations.[2] The `MPI_Send` operation is specified as follows:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

for C, or for FORTRAN:

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

2. MPI also provides several other “modes” for communication, i.e., synchronous, ready, buffered. In some cases these modes may provide easier conversion to MPI. However, the basic send and receive operations should provide the highest level of portability and performance.

```
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Several of these parameters should appear familiar to readers experienced with NX or CMMD. `buf` specifies what data to send. In MPI, buffers are "typed," i.e., all messages contain data of some specific type. The type could be `INTEGER`, `REAL`, `DOUBLE PRECISION`, etc. for FORTRAN; or `int`, `float`, `double`, etc. for C. In general, MPI supports any basic data type that the programming language (e.g., C or FORTRAN) supports. In addition, MPI supports "untyped" data by passing it as a series of bytes (using the `MPI_BYTE` data type).[3] When sending MPI messages, the `count` is the number of elements of data type "`datatype`" in `buf`. Therefore, if `buf` is an array of integers, `count` would be the number of integers in `buf` and `datatype` would be `MPI_INT` (for C) or `MPI_INTEGER` (for FORTRAN). If `buf` is a single double precision number, `count` would be 1 and `datatype` would be `MPI_DOUBLE` or `MPI_DOUBLE_PRECISION`. This differs from many other message passing systems because `count` is *not* the number of bytes in `buf`. This was implemented in order to ensure portability between systems that have different size data types. In addition, it enables MPI to be implemented for heterogenous environments, i.e., data can be converted between different formats.[4] `tag` is used as a selector between messages sent to the same process.[5] `dest` specifies the "rank" of the process to which the message is to be sent. A rank is roughly the same thing as a process or processor number in most systems. The difference is that all ranks are relative to some grouping of the system's processes specified by a "communicator" (`comm`). Normally, most programs can use the pre-defined communicator `MPI_COMM_WORLD`. This communicator includes all processes in a user's program, so a rank relative to it will be the same as a processor number on most systems. More information on communicators will be presented in Section 2.3. The final parameter of `MPI_Send`, `IERROR`, is used for returning an error value to FORTRAN programs (in C this value is returned directly by the function `MPI_Send`). This return value can be used to determine if the send was successful or not.

`MPI_Recv` is specified as follows:

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

3. Another advanced feature of MPI not discussed in this paper is derived datatypes. These allow buffers to contain elements that are different basic data types (like C structures). In addition, derived types can be used to specify strided vectors and other irregular data structures.

4. Support for heterogeneous environments (e.g., data conversion) is implementation dependent, not part of the standard. However, an MPI program written for a homogeneous environment would not have to be re-written to run with a heterogeneous MPI library.

5. Note that the MPI standard only guarantees that the tag field is 16 bits (0 to 65535). While most implementations support larger tags, it is advisable to keep tags within this limit.

Here, buf, count, and datatype specify the destination address, size, and the data type of the message being received. source is used to restrict the receive to messages sent by a process with a specific rank. Further, tag restricts the receive to messages sent with the same tag value. These values may be “wildcarded” by setting them to MPI_ANY_SOURCE or MPI_ANY_TAG if one wants to receive messages regardless of their source and/or tag. comm is the communicator mentioned before. The sending and receiving communicators must match, i.e., communicators may not be wildcarded like the source and tag fields. status is a variable in which message “status” informations is stored. This variable can then be used to determine information about the message received (i.e., size, sender, tag). Finally, MPI_Recv also returns an error value in the same way as described for MPI_Send.

Consider a simple example where the following program is run by two processes:

```
      program pingpong

      include 'mpif.h'
      integer ierr, rank, status(MPI_STATUS_SIZE)
      double precision buf(10)

c Initialize MPI Environment
      call MPI_Init(ierr)

c  Determine your rank in MPI_COMM_WORLD
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

      if (rank.eq.0) then
c Process 0 sends 10 double precision numbers to 1
        call MPI_Send(buf, 10, MPI_DOUBLE_PRECISION,
     $       1, 0, MPI_COMM_WORLD, ierr)
c Process 0 receives 10 double precision numbers from 1
        call MPI_Recv(buf, 10, MPI_DOUBLE_PRECISION,
     $       1, 1, MPI_COMM_WORLD, status, ierr)
      else
c Process 1 receives 10 double precision numbers from 0
        call MPI_Recv(buf, 10, MPI_DOUBLE_PRECISION,
     $       0, 0, MPI_COMM_WORLD, status, ierr)
c Process 1 sends 10 double precision numbers to 0
        call MPI_Send(buf, 10, MPI_DOUBLE_PRECISION,
     $       0, 1, MPI_COMM_WORLD, ierr)
      endif

      call MPI_Finalize(ierr)
      end
```

In this example, both processes determine what their “ranks” are relative to the communicator MPI_COMM_WORLD. Then, process 0 sends a message to process 1, and process 1 sends the data back to 0. Note that in a given communicator, all processes will be numbered from 0 to N-1 (where N is the total number of processes). The first message uses a tag value of 0, and the second one uses tag 1. Each message consists of 10 double precision floating point numbers.

2.2 Collective communication

MPI provides a wide range of collective communication operations including reductions, scans, broadcasts and barriers. MPI's collective operations are blocking, i.e., all processes must reach the collective operation before any may proceed past it. As an example, consider the global reduction operation:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
      MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM,
            IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Thus, an `MPI_Reduce` performs some type of reduction operation as specified by `op` (i.e., global sum, global max, global min, etc.) on the variable in `sendbuf` across all processes belonging to the provided communicator,[6] and the result is returned to the root process in `recvbuf`. The `MPI_Allreduce` command does the same thing, but the result is returned to the `recvbuf` variable in all processes that belong to the communicator. Therefore, an `MPI_Allreduce` can be thought of as an `MPI_Reduce` followed by a broadcast (`MPI_Bcast`).[7] This second option is similar to that provided by the "global" functions in Intel's NX library.

2.3 Groups, Contexts, and Communicators

One of the most powerful, and confusing, aspects of MPI is its use of groups, contexts, and communicators to provide a flexible and "safe" programming environment. All communication performed in MPI involves a *communicator*. In its simplest form, a communicator can be thought of as a *group*.[8] A group is simply a mapping from a rank (an integer between 0 and the number of processes - 1) to a physical process or processor. Therefore, for each communicator there will be a set of processes, and each process in this set will have a unique rank. Then messages

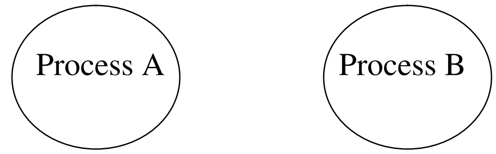
---

6. MPI collective communication operations involve only the processes defined by the communicator passed in to the collective communication routine. If this communicator is `MPI_COMM_WORLD`, the operation would involve all available processes. However, as will be detailed in the next section, communicators may only contain a subset of the available processes. This ability to perform collective communication on process subsets is very important for supporting multidisciplinary and multi-zonal applications.

7. A good implementation of `MPI_Allreduce` might use a different algorithm for the reduction and skip the broadcast.

8. Communicators actually consist of both a group and a "context." A context is a mechanism that can be used for protecting communication operations from interfering with each other. It is similar to a tag, but is not wildcardable. Therefore, a message sent using one context can not be received using a different context. This can be very useful for protecting library communication from other messages, but most users need not worry about this feature.

can be sent between processes using this rank. Therefore, as in Figure 1 the actual



`MPI_COMM_WORLD`	0	1
`MPI_COMM_SELF`	0	0
`comm_reverse`	1	0

Figure 1: Rank example.

processes (A and B) can be represented in different ways. For example, using `MPI_COMM_WORLD`, they have ranks 0 and 1 respectively. Further, MPI defines another communicator, `MPI_COMM_SELF`, that only contains the calling process. Therefore, every node will have rank 0 in `MPI_COMM_SELF`. We could also define another communicator, `comm_reverse`, that contains both process A and B from Figure 1, but reverses their ranks relative to `MPI_COMM_WORLD`. Thus, A has rank 0 in `MPI_COMM_WORLD`, and rank 1 in `comm_reverse`. B has rank 1 in `MPI_COMM_WORLD` and rank 0 in `comm_reverse`. Ranks are used as a shorthand method of specifying a process, and are only unique within a communicator. Thus, sending a message from process 0 to process 1 in `MPI_COMM_WORLD` is functionally equivalent to sending a message from process 1 to process 0 in `comm_reverse`.

Some simple operations on communicators are `MPI_Comm_size` and `MPI_Comm_rank`. `MPI_Comm_size` returns the number of processes defined by a communicator. `MPI_Comm_rank` specifies the rank of the calling process relative to the communicator. These commands are similar to `numnodes` and `mynode` in NX.

Communicators are not only used for renumbering nodes. They can also be used to break up the available processes in to disparate groups. Within each subgroup processes still have a rank (from zero to the subgroup size minus one). Note that messages sent using a particular communicator can not be received by any other communicator. Therefore, messages within a subgroup can not interfere with another subgroup. In addition, since collective communication operations occur within a communicator, separate collective operations may occur within each of these groups. Thus, it is possible to synchronize the processes in one communicator group without involving any processes outside of the communicator in that synchronization operation.

The easiest way to create subgroups is with the `MPI_Comm_split` command. This command is collective, so all processes belonging to the original communicator being spilt (`MPI_COMM_WORLD` in this example) must call it as follows, in C,

```
MPI_Comm_split(MPI_COMM_WORLD, color, 0, &newcomm);
```

or, in FORTRAN,

```
    call MPI_Comm_split(MPI_COMM_WORLD, color, 0, newcomm, ierr)
```

This routine would then create a new communicator (`newcomm`) in each process. The communicator generated in a particular process will include the group of processes for which the "color" (the value of the variable `color`) is the same. Therefore, there may be 1, 2, or more unique groups of processes created by an `MPI_Comm_split` command, as many as there are unique "colors." MPI also allows processes to specify what rank they wish to be in the new communicator by giving a "key" (equal to 0 in the example). Ties are broken by the rank in the source communicator (i.e., `MPI_COMM_WORLD` in the example), so if the key value is 0, the processes will be numbered from 0 to the appropriate group size with ranking in the same order as in the source communicator. For example, con-

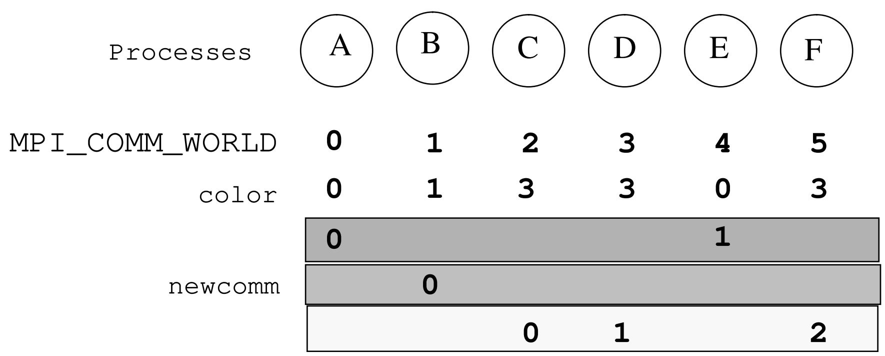


Figure 2: `MPI_Comm_split` example.

sider Figure 2. Here, each processor call `MPI_Comm_split` with the color value shown. This causes 3 different communicators to be created, one for each color. The communicator to which the calling process belongs is returned in `newcomm`. Within each group, processes are still numbered 0 to groupsize - 1. Note that there is no rank defined for process B in the communicator defined for processes A and E, thus, there is no way to send a message from process A to B using `newcomm`. However, it is still possible to send such a message using `MPI_COMM_WORLD`.

Now that it is possible to create separate communicators, it is still desirable to have a mechanism for communicating between these communicator groups. One method is simply to communicate through `MPI_COMM_WORLD`, which all processes will be members of. In addition, MPI provides a mechanism for sending messages between disparate communicators, i.e., *intercommunicators*. For example, consider Figure 3. We can send a message from process A to process B in two

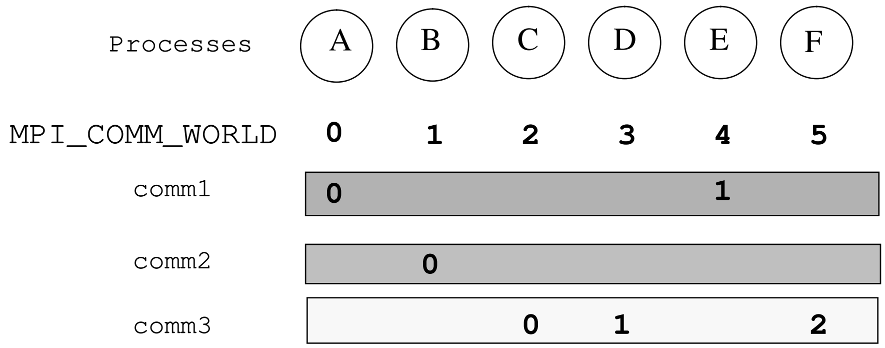


Figure 3: Intercommunicator example.

different ways. First, we could send a message from 0 to 1 in `MPI_COMM_WORLD`. Another option, however, would be to create an intercommunicator between `comm1` and `comm2`. Intercommunicators are used the same way as communicators in `MPI_Send` and `MPI_Recv`. However, the `dest` field in an `MPI_Send` will be relative to the remote group (if process A sends a message to a process with rank 0 using an intercommunicator created between `comm1` and `comm2`, the message would be sent to process B). The `source` field in an `MPI_Recv` using an intercommunicator also refers to the remote group (if process B receives a message from a process with rank 0 using an intercommunicator between `comm1` and `comm2`, the source would be process A). Therefore, to send a message from process A to process B using an intercommunicator between `comm1` and `comm2`, the message would be sent from process 0 to process 0.

To create an intercommunicator, however, some information must be known by both groups of processes. First, the local processes must know the rank of one process from the remote communicator. For this rank to make sense, it must be relative to some communicator that the local processes creating the intercommunicator also belong to. The processes, one in each of the communicator groups being joined, that have a rank known by everyone in both of the groups, are referred to as the local and remote *leaders*. It doesn't matter which processes are used as the group leaders, however, by convention, the group leaders in this paper will always be the processes with rank 0. Therefore, for Figure 3, we can use processes A, B, and C as the group leaders for `comm1`, `comm2`, and `comm3`. The call for creating an intercommunicator is as follows in C:

```
MPI_Intercomm_create(local_comm, local_leader, peer_comm,
                     remote_leader, tag, &newintercomm)
```

or, in FORTRAN

```
    call MPI_Intercomm_create(local_comm, local_leader,
   $     peer_comm, remote_leader, tag, newintercomm, ierr)
```

where `local_comm` is the local communicator, `local_leader` is the rank of the local leader within `local_comm`, `peer_comm` is the communicator to which both leaders belong (e.g., `MPI_COMM_WORLD`), and `remote_leader` is the rank of the remote leader within `remote_comm`. Therefore, for Figure 3, to join `comm2` and `comm3`, `local_leader` for the processes in `comm2` would be 0 with `local_comm` equal to `comm2` (this of course refers to process B). Remote leader for `comm2` would then be 2 and `peer_comm` would be `MPI_COMM_WORLD`. For the processes in `comm3`, `local_leader` would be 0 with `local_comm` set to `comm3` (this refers to process C), and remote leader would be 1 with `peer_comm` set to `MPI_COMM_WORLD` (referring to process B).The actual FORTRAN code for creating this intercommunicator between `comm2` and `comm3` of Figure 3 is as follows.

For processes in comm2:

```
    call MPI_Intercomm_create(comm2, 0, MPI_COMM_WORLD,
   $       2, tag, intercomm, ierr)
```

and for processes in comm3:

```
    call MPI_Intercomm_create(comm3, 0, MPI_COMM_WORLD,
   $       1, tag, intercomm, ierr)
```

The remaining parameter, `tag`, is an integer tag used to ensure that instances of `MPI_Intercomm_create` don't conflict, and `newintercomm` is the new intercommunicator to be created. `MPI_Intercomm_create` is a collective operation, so all processes in both of the communicator groups must call this operation at once.

3.0 Implementing Multidisciplinary Applications using MPI

Now, consider how one might map a multidisciplinary application on to MPI. As an example, consider the application described in Figure 4. Here the program con-

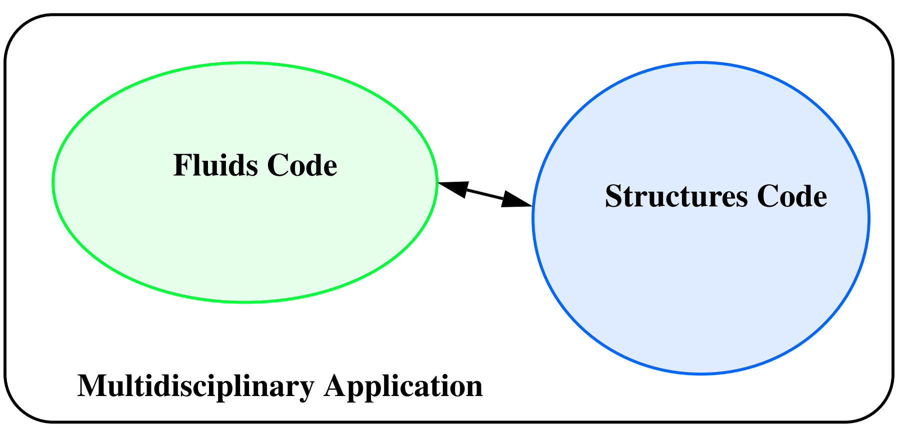


Figure 4: Block diagram of a simple multidisciplinary application.

sists of two distinct SPMD applications, one that simulates fluid dynamics (e.g., the airflow over an airplane wing), and another that simulates structures (e.g., flex in an airplane wing). In this example multidisciplinary application, both of these aspects are simulated to provide a more complete simulation. Ideally, one would want to simulate all aspects of an aircraft by integrating fluids, structures, thermal effects, etc. in to a complete multidisciplinary simulation.

3.1 Intra-discipline/zone Communication

As previously stated, one of the most common and successful methods for implementing multidisciplinary and multi-zonal applications is to take existing application codes based on a single discipline, and add communication of boundary information between these application codes to create a unified multidisciplinary application. Each of these single discipline codes is a SPMD message passing code with internal point to point and collective communication, I/O, and computation. The advantage of this technique is that one can use fully tested existing codes and therefore the development of the multidisciplinary application becomes primarily an integration problem. The problem, however, is that to use this technique, the underlying message passing system must allow applications to both operate as multiple independent SPMD programs, as well as allow communication between these independent tasks.

Consider how this fits in to the framework of MPI. As with any application, the

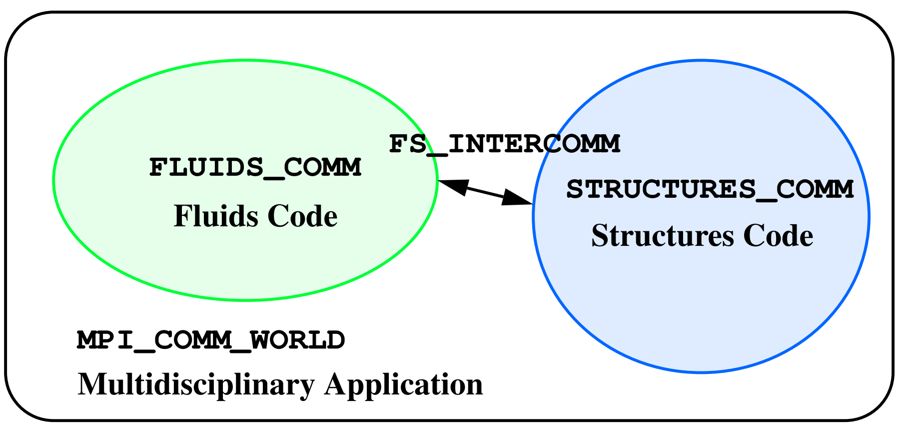


Figure 5: Block diagram of communicator assignments for a simple multidisciplinary application.

appropriate executable is loaded as the proper number of processes. Each process will have its own rank within the entire application, i.e., in `MPI_COMM_WORLD` (see Figure 5). Using this communicator it is possible for any processes to communicate. However, the problem with this view is that it makes the coding of the sub--programs much more difficult. For example, within `MPI_COMM_WORLD`, the fluids code processes may be numbered from 0 to 15, but this would mean that the first structures code process would be 16. This would require significant re-writing of the code since all process numbers for the structures code would be offset. In addition, it would not be possible to perform collective communication within either of the constituent codes without involving the other codes (since MPI collective communication involves all processes in the provided communicator). Therefore, one would generally want to split `MPI_COMM_WORLD` in to two pieces, one for each executable code. Again consider Figure 5, one can simply break `MPI_COMM_WORLD` in to two communicators, i.e., `FLUIDS_COMM` and `STRUCTURES_COMM`. Within each of these smaller communicators, all processes are numbered from 0 to N-1 (where N is the number of processes running the code), processes can now perform collective communication operations within their code group, and point to point operations within each communicator can not interfere with any processes outside of the scope of the communicator.

3.2 Inter-discipline/zone Communication

Now that communicators have been defined for communication within each of the codes, there still needs to be a mechanism for communicating between the codes. There are two ways that this can be done using MPI. One would be to use `MPI_COMM_WORLD`, and to refer to each process by its rank in this communicator for inter-group communication. The problem is that this reduces modularity and increases complexity since it becomes necessary to have every process keep track of how processes are allocated and the size of every group. For example, to determine the rank of the first structures code it would be necessary to know how many processes are allocated for fluids in `MPI_COMM_WORLD`, and it would also be necessary to know how processes are allocated (e.g., are they allocated in blocks, cyclically, subcubes, etc.). A better approach is to use *intercommunicators* as shown in Figure 5. Intercommunicators allow direct communication between processes belonging to disjoint groups and allow processes in one group to send messages to another using the ranks defined in the remote group. This means that only a single rank has to be used for each process, whether communication is within a communicator or to another communicator. Referring to Figure 5, it is possible to have fluids process 5 send a message to structures process 3 using `FS_INTERCOMM`. To create this communicator one can use `MPI_Intercomm_create` (see Section 2.3), however, this still requires the programmer to know some information about how processes have been allocated (i.e., who are the group leaders). Unfortunately, using MPI alone, this information is likely to be system dependent, and thus will not be portable,

4.0 MPIRUN

The problem that has not been addressed in the previous sections is how to establish the application structure as shown in Figure 5. In other words, an MPI program must be able to:

- load and run the correct executables,

- establish communicators for each executable (e.g., for the fluids and structures application codes), and
- create the fluid-structures intercommunicator.

Loading is external to the scope of MPI, therefore, some machine specific mechanism must be used for loading the executables. Next, to establish the group communicators each process can simply feed an integer representing its executable group in to the "color" field of `MPI_Comm_split`. This can be provided in one of two ways, either the color can be hard-coded into each executable, or the color values can be distributed at run-time. The first approach is unacceptable because it requires re-compilation any time the number of zones or disciplines is changed, and it means that one must have separate executables for each zone in a multi--zonal application (i.e., where the same code is applied to different data sets each representing a zone). A better solution is to have this information distributed after the program is loaded. The final step is to set up intercommunicators. For this it is necessary to establish "well known" group leaders so that intercommunicators can be established with `MPI_Intercomm_create`. This can also be coded statically, but is also better done at run time to enhance flexibility and code re-use.

While it is possible to establish the correct environment on any machine, it is not possible to do so portably using MPI alone. Loading is completely non-portable. On a Paragon, loading is done with `nx_load`, on an iPSC/860 loading is done with `load`, on workstations loading is dependant on the underlying parallel environment being used (e.g., Argonne P4 [BuL92], PVM [GeB93], UNIX, IBM's POE [Ibm94]), etc. Further, since loading can be different, the means for distributing process allocation information at run time will also be non-portable. To simplify the process and to provide a portable means for specifying MPI applications, the MPIRUN loader was developed at NASA Ames. MPIRUN can be built on top of any implementation of MPI, and provides mechanisms for loading, establishes communicators for each executable, and distributes information about group leaders. Because MPIRUN contains all of the machine specific operations, programs using MPIRUN for loading and MPI for communication will be portable to any platform to which MPIRUN has been ported (currently MPIRUN runs on the Intel iPSC/860 and Paragon, workstations running MPI on top of P4, the IBM SP series, and the Thinking Machines CM-5).

MPIRUN can be used to start any MPI application, regardless of whether the program uses any of MPIRUN's special features. However, to use MPIRUN's ability to establish a multidisciplinary/multi-zonal application environment, MPIRUN application codes must link in the MPIRUN library as well as the native MPI library. They must also include the file "`mpirun.h`" (for C) or "`mpirunf.h`" (for FORTRAN). Finally, an MPIRUN application code must call the routine `MPIRUN_Init` immediately after calling `MPI_Init`, i.e, before any MPIRUN functions are called. Note that `MPIRUN_Init` uses MPI functions and therefore it will not work unless `MPI_Init` has been called first.

To run an MPIRUN program, the "mpirun" command is used. mpirun allows the user to specify what executables are to be loaded as well as how many processes should run each executable. Each of these sets of processes running a given executable is known as an *MPIRUN SPMD application*. mpirun also allows the user to pass arguments to the underlying system as well as to the user program. Processes are allocated by mpirun, assigned to MPI groups (encompassing each of the MPIRUN SPMD applications specified on the mpirun command line), and a communicator is formed for each group. In addition to starting processes and forming initial groups for each MPIRUN SPMD application, MPIRUN also creates several variables that enable a user to easily establish intercommunicators.

Again consider the fluid-structures example, here shown in Figure 6, the applica-

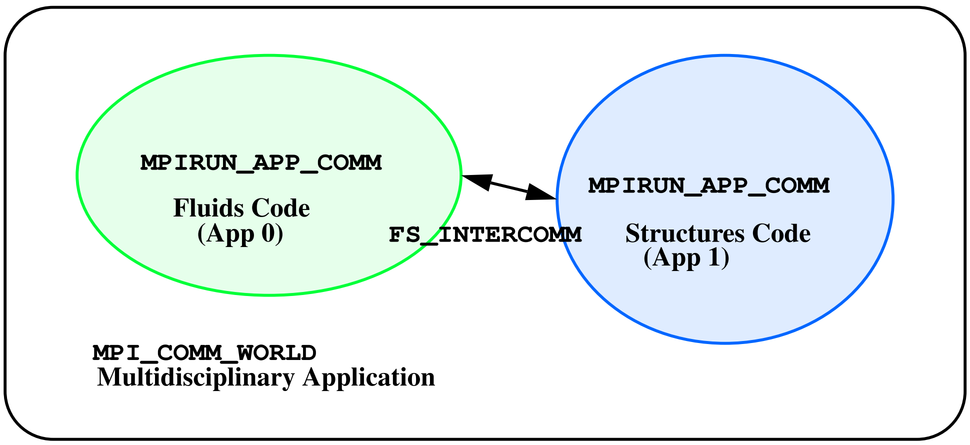


Figure 6: Block diagram of communicator assignments for a simple multidisciplinary application using MPIRUN.

tion consists of two MPI groups representing each MPIRUN SPMD application, each existing within MPI_COMM_WORLD. To aid in establishing this structure, the following pre-defined variables are provided by MPIRUN: MPIRUN_APP_-COMM, MPIRUN_NUM_APPS, MPIRUN_APP_ID, and MPIRUN_AP-P_LEADERS.

MPIRUN_APP_COMM

This is a communicator available to each process representing the set of processes to that belong to the same MPIRUN SPMD application as the calling process. This means that this communicator will be different for each MPIRUN SPMD application specified on the mpirun command line, however, the name is uniform throughout an MPIRUN program. This communicator should be used as the basis for communication inside of each SPMD application. For example, referring to Figure 6, a process running the fluids code can communicate to another fluids process using MPIRUN_APP_COMM. In addition, a structures process can communicate to a structures process using MPIRUN_APP_COMM. However, for a fluids process to communicate with a structures process an intercommunicator will have to be formed.

MPIRUN_NUM_APPS

This is simply the number of MPIRUN SPMD applications started by `mpirun`. Therefore for Figure 6, `MPIRUN_NUM_APPS` would be equal to 2.

MPIRUN_APP_ID

This is the "SPMD application ID" for the SPMD applications started by `mpirun`. Each MPIRUN SPMD application will have a unique ID ranging from 0 to `MPIRUN_NUM_APPS`-1. The variable `MPIRUN_APP_ID` is defined in every process as the application ID for the MPIRUN SPMD application to which the process belongs. Application IDs are allocated by MPIRUN following a deterministic allocation strategy. The `mpirun` command line is parsed from left to right, and as it is parsed, groups are allocated starting with application ID 0. Using this allocation strategy it should be possible to decide the application ID for a given MPIRUN group prior to run-time, so that programs can use application IDs to determine what applications belong to what groups. Thus, referring to Figure 6, the application may have been started with the following command:

```
mpirun -np 64 fluids_code : -np 32 structures_code
```

Thus, the fluids code would have `MPIRUN_APP_ID` equal to 0 and would be running on 64 processes, and the structures code would have `MPIRUN_APP_ID` equal to 1 and would run on 32 processes.

MPIRUN_APP_LEADERS

`MPIRUN_APP_LEADERS` is an array intended to facilitate the creation of intercommunicators. Recall that the `MPI_Intercomm_create` command requires the calling process to know the rank of at least one member of the remote group relative to some common communicator. Simply put, this rank is exactly what `MPIRUN_APP_LEADERS` provides. More specifically, `MPIRUN_APP_LEADERS[ID]` is defined as the rank relative to `MPI_COMM_WORLD` of the process within MPIRUN SPMD application number `ID` with rank 0. For the example in Figure 6 we could form the intercommunicator `FS_INTERCOMM` by having every user process execute the following sequence in C:

```
if (MPIRUN_APP_COMM == 0)
   ret=MPI_Intercomm_create(MPIRUN_APP_COMM, 0, MPI_COMM_WORLD,
         MPIRUN_APP_LEADERS[1], 0, &FS_INTERCOMM);
else
   ret=MPI_Intercomm_create(MPIRUN_APP_COMM, 0, MPI_COMM_WORLD,
         MPIRUN_APP_LEADERS[0], 0, &FS_INTERCOMM);
```

or, in FORTRAN:

```
    if (MPIRUN_APP_COMM .eq. 0)
      call MPI_Intercomm_create(MPIRUN_APP_COMM, 0,
   $     MPI_COMM_WORLD,MPIRUN_APP_LEADERS(1), 0,
   $     FS_INTERCOMM, ierr)
    else
      call MPI_Intercomm_create(MPIRUN_APP_COMM, 0,
   $     MPI_COMM_WORLD, MPIRUN_APP_LEADERS(0), 0,
   $     FS_INTERCOMM, ierr)
```

```
endif
```

For more details on both the `mpirun` command and on the special features provided to MPIRUN applications, see the `mpirun` man page. To find out how to obtain the MPIRUN package send e-mail to `fineberg@nas.nasa.gov`.

5.0 Performance

In this section MPI will be briefly compared with the native message passing layer on three systems, the IBM SP-1, the IBM SP-2, and the Intel Paragon. The best version of MPI currently available for the Paragon is the Argonne/MS State implementation[9] (MPICH), however, for the SP-1 and SP-2 there is an IBM developed version of MPI available (MPI-F) [Fra94] as well as MPICH. Referring to Table 1,

TABLE 1. Message Passing Library Performance

Machine	*Message Layer*	*Latency (μsec)*	*Bandwidth (MB/sec)*
Paragon	NX	149	43.2
Paragon	MPICH	201	28.0
SP-1	MPL/p	36	8.8
SP-1	MPI-F	37	8.8
SP-2	MPL	43	35.6
SP-2	MPICH	52	35.5
SP-2	MPI-F	41	35.6

the latency and bandwidth of the MPI libraries were compared to the fastest proprietary vendor libraries.[10] These libraries are MPL/p for the SP-1, which is a fast version of MPL developed by IBM research specifically for the SP-1 (also known as EUIH). For the SP-2, MPL is the normal SP-2 message passing library, there is no IBM research version of MPL for the SP-2. For the Paragon, the only vendor supplied library is NX.

As can be see from Table 1, there is a significant, but not prohibitive, penalty for using a non-vendor supplied MPI implementation (i.e., MPICH) for point-to-point communication. This effect appears as higher latency and in some cases lower bandwidth. For most systems, only the latency is effected, however, in the case of the Paragon, memory copying is quite slow and this limits the performance of MPICH since it adds an extra buffering step.[11] On the SP-2, memory copies are fast, so MPICH has higher latency but virtually identical bandwidth (see Figures 7 and 8). For the vendor supplied version of MPI (MPI-F), the penalty is essentially non-existent on the SP-1, and the MPI-F performance is measurably better than MPL on the SP-2. While this advantage of MPI-F over MPL is probably due to

9. This MPI implementation is available via anonymous FTP at `info.mcs.anl.gov`.

10. These experiments were run in August and September 1994. The SP-2 used has 64 "wide" processing nodes with 128 MB of RAM per node. The Paragon had 208 compute nodes, 32MB per node, and the communication coprocessors were not enabled. The SP-1 had 128 nodes, 64MB/node.

11. This effect would probably have been less had the communication coprocessor been enabled, however, this was not available on the NAS Paragon at the time the experiments were performed.

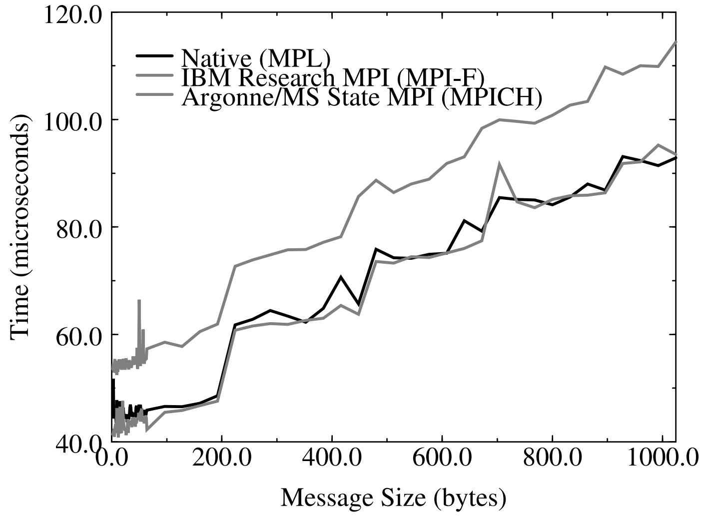


Figure 7: Comparison of SP-2 Message Latency

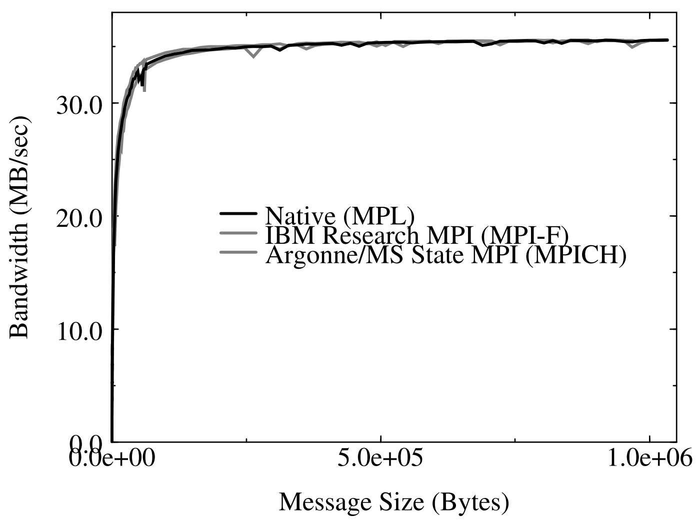


Figure 8: Comparison of SP-2 Message Bandwidth

optimizations used by IBM research that were not possible in the "production" message passing library, it provides evidence that there are no "flaws" in MPI preventing it from performing well.

For collective communication, even with the added complexity of communicators, MPI-F also performs better than MPL on the SP-2. In Figures 9 and 10, two representative forms of collective communication are shown for the SP-2, barrier synchronization and broadcast of a 1K message. As can be seen, MPI-F gets consistently better collective communication performance over MPL and MPICH. This is due to two factors, the lower latency of MPI-F, and the use of better algorithms than MPL. The algorithmic improvement is most evident for Barrier Synchronization. MPICH is significantly worse than MPL, however the performance loss is less than a factor of two. This is relatively good given the approximately 20% higher latency provided by MPICH.

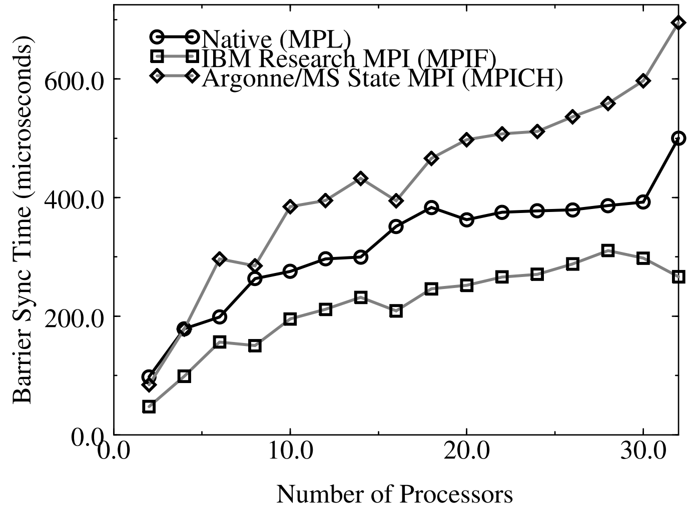


Figure 9: Comparison of SP-2 Synchronization Performance

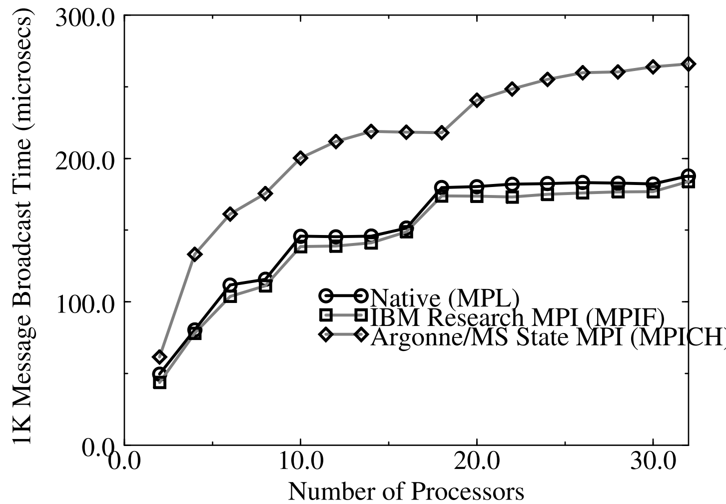


Figure 10: Comparison of SP-2 Broadcast Performance

6.0 Summary

In this paper it has been shown that it is possible to create multidisciplinary applications using MPI for communication and that MPI is capable of providing performance commensurate with proprietary message passing libraries. Further, a portable loader interface has been created to simplify program initiation enable these MPI codes to be portable. Thus, using MPI and MPIRUN it is now possible to create portable multidisciplinary and multi-zonal applications. Further, the MPIRUN interface was designed to be generic enough so that it can be implemented on any MIMD architecture, as evidenced by the variety of machines it currently runs on.

7.0 References[12]

[BaW93] E. Barszcz, S. Weeratunga, and E. Pramono, *A Model for Executing Multidisciplinary and Multizonal Programs*, Report Number RNR-93-009, NASA Ames Research Center, 1993.

[Bar91] E. Barszcz, *Intercube Communication for the iPSC/860*, Report Number RNR-91-030, NASA Ames Research Center, 1991.

[BuL92] R. Butler and E. Lusk, *User's Guide to the P4 Programming System*, Tech. Report RM-ANL-92/17, Argonne National Laboratory, 1992.

[Fin93c] S. Fineberg, *Implementing the NHT-1 Application I/O Benchmark*, Report RND-93-007, NASA Ames Research Center, 1993.

[Fra94] H. Franke, *MPI-F: An MPI Implementation for IBM SP-1/SP-2, Version 1.30*, Technical Report, IBM T. J. Watson Research Center, 1994.

[GeB93] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sundaram, *PVM 3 User's Guide and Reference Manual*, Report ORNL/TM-12187, Engineering, Physics, and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, 1993.

[GeS91] G. Geist and V. Sunderam, *Network Based Concurrent Computing on the PVM System*, Tech. Report TM-11760, Oak Ridge National Laboratory, 1991.

[Ibm94] IBM, *IBM AIX Parallel Environment Operation and Use Release 2.0*, International Business Machines Corp., 1994.

[Mes94] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Computer Science Dept. Technical Report CS-94-230, University of Tennessee, 1994.

12. NAS technical reports are available by sending e-mail to `doc-center@nas.nasa.gov` or via WWW at URL: "`http://www.na.nasa.gov`".

<table>
<tr><td></td><td>NAS TECHNICAL REPORT</td></tr>
<tr><td></td><td>Title:
Implementing Multidisciplinary and
Multi-zonal Applications Using MPI</td></tr>
<tr><td></td><td>Author(s):
Samuel A. Fineberg</td></tr>
<tr><td>Two reviewers must sign.</td><td>Reviewers:
“I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document.”
Signed: ______________________________
Name: ______________________________
Signed: ______________________________
Name: ______________________________</td></tr>
<tr><td>After approval, assign NAS Report number.</td><td>Branch Chief:
Approved: ______________________________</td></tr>
<tr><td>Date:</td><td>NAS ReportNumber:</td></tr>
</table>