Dynamic Process Management in an MPI Setting

William Gropp
Ewing Lusk *
Mathematics and Computer Science Division
Argonne National Laboratory
gropp@mcs.anl.gov
lusk@mcs.anl.gov

Abstract

We propose extensions to the Message-Passing Interface (MPI) Standard that provide for dynamic process management, including spawning of new processes by a running application. Such extensions are needed if more of the runtime environment for parallel programs is to be accessible to MPI programs or to be themselves written using MPI. The extensions proposed here are motivated by real applications and fit cleanly with existing concepts of MPI.

1 Introduction

During 1993 and 1994 a group composed of parallel computer vendors, library writers, and application scientists created a standard message passing library interface specification [1, 2, 4]. This group, which called itself the MPI Forum, chose to propose a standard only for the message-passing library, attempting to unify and subsume the plethora of existing libraries. They deliberately and explicitly did not propose a standard for how processes would be created in the first place, only for how they would communicate once they were created.

MPI users have asked that the Forum reconsider this issue for several reasons. The first is that workstation network users are accustomed to using PVM's capabilities [3] for process management and find no replacement for them in MPI, thus making PVM-to-MPI translations awkward. (On the other hand, dynamic process creation is often difficult or impossible on MPP's, limiting the portability of such PVM programs.) A second reason is that important classes of message-passing applications, such as client-server systems and task-farming jobs require dynamic process control. A third is that with such extensions it would be possible to write major parts of the parallel programming environment in MPI itself.

In this paper we describe an architecture of the system runtime environment of a parallel program that separates the functions of job scheduler, process manager, and message-passing system. We show how the existing MPI specification, which can serve handily as

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

a complete message-passing system, can be extended in a natural way to include an application interface to the system's job scheduler and process manager, or even to write those functions if they are not already provided. (A typical difference between an MPP and a workstation network is that the MPP comes with a built-in scheduler and process manager, whereas the workstation network does not. We will make this distinction clearer in Section 2.) The extensions are straightforward and fit will into the MPI framework, reusing several of its existing concepts and functions.

The paper is organized as follows. In Section 2 we describe in detail what we mean by each of the components of the parallel runtime environment—job scheduler, process manager, and message-passing system—and give several examples of complete systems with very different components. Section 3 contains the basic principles behind the design of the extensions and the definitions of the extensions themselves. It concludes with a comparison of the MPI approach defined here with the PVM approach to dynamic process management, which is quite different. Section 4 contains descriptions of several complete applications that make use of the extensions described here. In the conclusion we address issues of implementation status.

2 Runtime Environments of Parallel Programs

A parallel program does not execute in isolation; it must have computing and other resources allocated to it, its processes must be started and managed, and (presumably) its processes must communicate. MPI standardizes the communication aspect, but says nothing about the other aspects of the execution environment.

One reason that the MPI forum chose to (temporarily) ignore these aspects is that they vary so greatly in current parallel systems. In order to motivate the structure of the MPI extensions that we are going to propose in Section 3, we describe here the major components of a parallel runtime environment and give a number of examples of various instantiations of this structure.

2.1 Components

One way to decompose the complex runtime environment at a high level on today's parallel systems is to separate out the functions of *job scheduler*, *process manager*, *message-passing library*, and *security*.

Job Scheduler By the *job scheduler* we mean that part of the system that manages resources. It decides which processors will be allocated to the parallel job when it runs and when it will run. In some environments it is represented by a component of a sophisticated batch queueing system; in others it is represented by the user himself, who can start jobs whenever and wherever he likes on a network.

Process Manager Once processors have been allocated to a program, user processes must be started on those processors, and managed after startup. By "managed" we mean that signals must be deliverable, that `stdin`, `stdout`, and `stderr` must be handled in some reasonable way, and that orderly termination can be guaranteed. A minimal example is `rsh`, which starts processes and reroutes `stdin`, `stdout`, and `stderr` back to the originating process. A more complex example is given by `poe` on the IBM SP2 or `prun` on the Meiko CS-2, which start processes on processors given to them by the job scheduler and manage them until they are finished.

In some cases the situation is muddied by combining the functions of job scheduler and process manager in one piece of software. Examples of this approach are the batch queueing systems such as Condor, DQS, and LoadLeveler. Nonetheless, it will be convenient to consider them separately, since although they must communicate with one another they are separate functions that can be independently modified.

Message-Passing Library By the *message-passing library* we mean the library used by the application program for its interprocess communication. Programs containing only calls to a message-passing library can be extremely portable, since they fit cleanly into a variety of job scheduler - process manager environments.

Security An important function of the runtime environment that we have not discussed is *security*. The security system ensures that the job scheduler does not allocate resources to users or programs that should not have them, that the process manager does indeed control the processes that it starts, and that the message-passing library delivers messages only to their proper destinations. We will assume here that the security function is adequately integrated into the combination of the components of the runtime environment, and not discuss it further. It does not influence the *specification* of our proposed MPI extensions, although it is certainly an important part of their implementation.

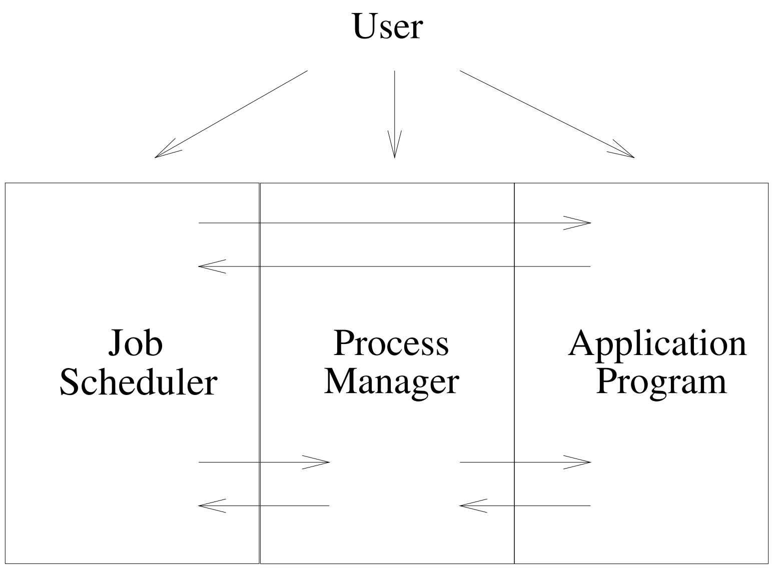

Figure 1: Structure of the Runtime Environment

These components need to communicate among themselves and with the user, but the timing and the paths of such communication vary from one environment to another. Some of the paths are illustrated in Figure 1.

For example, the job scheduler and the process manager must communicate so that the process manager can know where to start the user processes, The process manager and the message-passing library communicate in order for the message-passing library to know where the processes are and how to contact them. The user may interact only with the job scheduler (as in the case of LoadLeveler), directly with the process manager (`poe`, `prun`), or only with the application program `p4`. Finally, it may be useful for the application program to dynamically request more resources from the job scheduler.

2.2 Examples of Runtime Environments

To illustrate how the above framework allows us to describe a wide variety of actual systems, we give here some examples.

ANL's SP2 The SP2 at Argonne National Laboratory is scheduled by a locally written job scheduler quite different from the LoadLeveler product delivered with the SP2. It ensures that only one user has access to any SP node at a time and requires users to provide time limits for their jobs so that the machine can be tightly scheduled. Users submit scripts to the scheduler, which sets up calls to `poe`, IBM's process manager on the SP. The `poe` system interacts with a variety of message-passing libraries.

The Meiko CS-2 at LLNL Job scheduling is done by the user himself who inspects the state of the machine interactively and claims a partition with a fixed number of processors. He then invokes the process manager with the `prun` command, specifying exactly how many processes he wishes to execute in the given partition. `prun` starts processes that use Meiko's implementation of Intel's NX library, or MPI programs that run on top of this library.

Paragon at Caltech

Workstation network managed by DQS DQS **??** is a batch scheduler for workstation networks developed at Florida State University. Users submit jobs to it and it allocates the necessary resources and starts jobs. It has an interface to `p4` that allows it to start parallel jobs written using `p4` but not (currently) any other library. Similarly, Condor, a batch scheduler, can start PVM jobs on the network it manages at the University of Wisconsin, but no others.

Basic workstation network with PVM One reason for PVM's popularity is that it can be viewed as a completely self-contained system that supplies its own process management and can be used to implement a job scheduler as well. On systems that have neither of these functions pre-installed, PVM can provide a complete solution. On the other hand, conflicts between existing process managers and PVM can inhibit the portability (to MPP's) of self-contained programs that assume all functionality will be provided by PVM.

(Need detailed description of how PVM handles all functions)

2.3 Applications Requiring Direct Communication with the Runtime System

The existing MPI specification is adequate for writing a very large number of parallel applications. In these applications, the job scheduler and process manager, whether simple or elaborate, allocate resources and manage user processes without interacting with the application program. In other applications, however, it is necessary that the *user level* of the application communicate with the job scheduler and process manager. Here we describe three broad classes of such applications. In Section 4 we will give concrete examples of each of these classes.

Task Farming By a "task farm" application we mean a program that manages the execution of a set of other, possibly sequential, programs. This situation often arises when one wants to run the same sequential program many times with varying input data. We call each invocation of the sequential program a *task*. It is often simplest to "parallelize" the existing sequential program by writing a parallel "harness" program that in turn devotes a separate, transient process to each task. When one task finishes, a new process is started to execute the next one. Even if the resources allocated to the job are fixed, the "harness" process must interact frequently with the process manager (even if this is just `rsh`, to start the new processes with the new input data).

Dynamic number of processes in parallel job The program wishes to decide *inside* the program to adjust the number of processes to fit the size of the problem. Furthermore, it may continue to add and subtract processes during the computation to fit separate phases of the computation, some of which may be more parallel than others. In order to do this, the application program will have to interact with the job scheduler (however it is implemented) to request and acquire or return computation resources. It will have to interact with the process manager to request that process be started, and in order to make the new processes known to the message-passing library so that the larger (or smaller) group of processes can communicate.

Client/Server This situation is the opposite of the situations above, where processes come and go upon request. In the client/server model, one set of processes is relatively permanent (the server, which we assume here is a parallel program). At unpredictable times, another (possibly parallel) program (the client) begins execution and must establish communication with the server. In this case the process manager must provide a way for the client to locate the server and communicate to the message-passing library that it must now support communications with a new collection of processes.

It is not currently possible to write any of these applications entirely in MPI, because MPI does not proved the necessary interfaces between the application program and the job scheduler or process manager. On the other hand, MPI does contain several features that make it relatively easy to add such interfaces.

3 Extending MPI for Process Management

In this section we will first describe requirements for the interface which influence some of the decision. then we will (finally) propose a set of MPI extensions that will meet the requirements. Not that we think of ourselves as providing an interface to existing job scheduling and process management systems. If they do not exist, then we want to be able to write them in MPI.

3.1 Requirements

Of course the most basic requirement is that we be able to write portable applications in the above classes, that can run in a variety of job scheduling – process management environments. In addition, we would like our interface to have a number of other properties.

Determinism The semantics of dynamic process creation must be carefully designed to avoid race conditions. For example, we will not allow a new process to join a group while a collective operation over that group is in progress.

Scalability It must be possible to deal with large numbers of processes by exploiting potential scalability in the job scheduler or process manager.

Economy We would like to take advantage of MPI's rich set of functions for dealing with asynchronicity, and avoid introducing new objects or functions if it can be avoided.

MPI's current design makes it far easier to meet these requirements than for other systems to do so. As will be seen in the next section, we will be able to eliminate race conditions by using MPI's communicators to encapsulate the collective act of changing the number of processes in a group. By adding new variants of the existing MPI_Request object, we will be able to take advantage of MPI's extensive set of functions for testing and waiting on numbers of requests.

3.2 MPI Extensions

We assume that reader is familiar with the MPI specification, particularly communicators (both intra and inter), persistent requests, and the family of MPI_Wait and MPI_Test operations.

We will first present a very simple interface that combines access to the job scheduler and process manager, yet provides for at least one kind of dynamic process control and for some client server applications. These will be straightforward, blocking, communicator-based operations. Then we will show how increased flexibility and efficiency can be obtained by breaking these into component operations, which are non-blocking and operate directly on dynamic processes represented by a new variety of MPI_Request object.

3.2.1 Simple Interface to Process Manager

The following function is the easiest way to create new processes. It startss the same executable with the same argument list on a set of processors. It is a collective operation over the processes in the group associated with oldcomm, and returns an intercommunicator newcomm, which has the new processes as the remote group. The spawned processes will have the usual MPI_COMM_WORLD, consisting of those processes spawned with a single call, and will also have a new predefined communicator, MPI_COMM_PARENT, which has as its remote group the spawning processes. An ordinary intracommunicator containing all processes can then be constructed using MPI_INTERCOMM_MERGE.

MPI_SPAWN(oldcomm, archtype, count, array_of_names, executable, argvector, flag, newcomm)

IN	oldcomm	communicator of spawning group
IN	archtype	architecture type of machine to spawn on
IN	count	number of processes to spawn (int)
IN	array_of_names	array of hostnames (array of strings)
IN	executable	executable file for new processes to execute (string)
IN	argvector	arguments to be passed to new processes (array of strings)
IN	flag	options (integer)
OUT	newcomm	new intercommunicator including new processes as the remote group

> Some people are uncomfortable with the complexity of intercommunicators. A different version of MPI_SPAWN could just return an expanded intracommunicator instead of a new intercommunicator.

There are at least two reasons why the high-level, simple interface is not enough. In the first place, these operations are liable to be expensive, and so it would be nice not to have to block waiting for their completion while other useful work could be done. Secondly, they encapsulate too much, combining unnecessarily interactions with both the scheduler and the process manager. In the next few sections we break these higher-level functions into their component parts for greater control, and make the components non-blocking. Non-blocking operations in general return MPI_Request objects, so we begin by discussing requests in general.

3.2.2 New Types of `MPI_Requests`

In the following sections, we will introduce several non-blocking operations. Rather than introducing new versions of MPI_WAIT, MPI_TEST, etc., we propose to use the existing, rich set of MPI functions. In order to allow this, we must allow an MPI_Request to represent some new kinds of requests. Since we will then need to determine the type of a request, we need something like:

MPI_REQUEST_TYPE(request, type)

IN	request	request being queried (handle)
IN	type	one of a predefined set of request types (int)

Since it seems likely that we will continue to need to deal with new kinds of requests (for example, for I/O), we might also allow users to define requests, with

MPI_NEW_REQUEST_TYPE(type, test_function, wait_function, free_function)

IN	type	new type being defined (int)
IN	wait_function	function to be called when this request is waited on
IN	test_function	function to be called when this request is tested
IN	free_function	function to be called when this request is freed

The requests created by these calls will be *persistent*, that is, they will need to be freed explicitly. To provide scalability, in addition to the usual MPI_Request_free, function, we provide

MPI_REQUEST_FREE_ALL(num_requests, array_of_requests)

IN	num_requests	number of requests in the array
IN	array_of_requests	set of requests to be freed

3.2.3 Interfacing to the Job Scheduler

Here we give a set of functions for interfacing directly to the job scheduler. The first one can be used to obtain the hostnames used in a call to MPI_SPAWN. Since interaction with the job scheduler can be time-consuming, we make this a non-blocking operation, which returns an array of requests to be waited on later.

> In general, this function does not start any new processes; rather, it obtains resources from the job scheduler for use by other functions. However, we need to take into account those job schedulers that provide this function without starting processes, such as LoadLeveler or DQS. In those cases, the executables may not be the application executables, but rather interface processes will create the application processes in response to one of the process-creation functions described here (like MPI_Spawn).

MPI_IALLOCATE(num_requested, js_dep_string, arch_type, array_of_nodenames, array_of_executables, hardness, array_of_requests)

IN	num_requested	number of hosts requested
IN	js_dep_string	special information parsed by job scheduler
IN	arch_type	architecture type of machine to spawn on
IN	array_of_nodenames	array of hostnames (array of strings)
IN	array_of_executables	array of executable files
IN	hardness	whether the request is hard or soft (integer)
OUT	array_of_requests	set of requests

The array of nodenames may contain wildcard indicators, to allow the job scheduler to pick the processors to be used. A *hard* allocation request is required to eventually return the entire number of processors requested, whereas a *soft* allocation request may return allocate some processors even if it knows that it will not be able to satisfy the entire request.

3.2.4 Interfacing to the Process Manager

While MPI_SPAWN conveniently captures many aspects of resource allocation and process startup in one call, we need more detailed control of these step. Our model will be that one calls MPI_IALLOCATE to reserve processors, getting back a set of requests. These requests can be further modified with funtions we present in this section, and then actual process startup can be accomplished with the existing MPI_START call. (There will also be a scalable version, for starting multiple requests at once.)

First, there are routines for setting attributes of requests that may not have been set with MPI_IALLOCATE.

MPI_SET_EXEC(request, executable)

INOUT	request	request (handle)
IN	executable	name of executable file

sets the name of the file to be executed, and

MPI_SET_ARGS(request, args)

INOUT	request	request (handle)
IN	args	array of strings

sets the command-line arguments that process will receive.

These same operations perhaps should also be defined for arrays of requests.

We may also need functions to extract attributes of requests, like

MPI_GET_NODENAME(request, hostname)

IN	request	request (handle)
OUT	hostname	the name of the host associated with the request

to retrieve the hostname that was filled in by the job scheduler for a particular request, and

MPI_GET_JSINFO(request, js_dep_info)

IN	request	request (handle)
OUT	js_dep_info	job-scheduler-dependent information

to retrieve special information dependent on the job scheduler being used.

Once the requests have been set up, they can be initiated with the existing MPI_Start routine, or a set of them can be started with:

MPI_START_ALL(num_requests, array_of_requests)

IN	num_request	number of requests in array (int)
INOUT	array_of_requests	requests (array of handles)

Process requests have *two* stages, and we wait on both. The first stage is completed when the process has been started. The second stage is completed when the processes exits. (We can think of it as the MPI interface to the Process Manager's handling of the signal `SIGCHILD`. The status returned by the usual wait and test routines can be used to determing which state has been completed. Note that this level of process management allows us to manage non-MPI processes, since communicators are not involved. If the started processes are MPI processes (that is, if they call MPI_INIT), then the communicator that includes them can be constructed with one of the following two routines.

MPI_ATTACH(comm, num_requests, array_of_requests)

INOUT	comm	communicator which is expanded to include new processes
IN	num_requests	number of requests in array (int)
IN	array_of_requests	requests representing the processes to be attached to

This operation is collective over all the processes involved in the sense that the attaching group must all call MPI_ATTACH, while the processes that are attached must call MPI_INTERCOMM_REPLACE with MPI_COMM_WORLD and MPI_COMM_PARENT.

MPI_REMOTE_ATTACH(oldcomm, num_requests, array_of_requests, newcomm)

IN	oldcomm	communicator
IN	num_requests	number of requests in array (int)
IN	array_of_requests	the requests representing the processes to be attached to
IN	newcomm	new intercommunicator for the new processes

This operation is collective over oldcomm, and returns an intercommunicator, One can think of this poeration as very much like MPI_INTERCOMM_MERGE, where the remote processes are represented by and array of requests rather than by the remote group in an intercommunicator.

We also need

MPI_DETACH(comm, flag)

INOUT	comm	communicator
IN	flag	indicates whether the calling process is detaching (int)

This operation is collective over comm. What happens to comm?

We don't need MPI_REMOTE_DETACH(intercomm)

IN	intercomm	communicator

because its function can be accomplished with MPI_COMM_FREE.

We will also need to ask the process manager to deliver signals to processes, which are represented by requests:

MPI_SIGNAL(signal, num_requests, array_of_requests)

IN	signal	signal type (int)
IN	num_requests	number of requests in array
IN	array_of_requests	requests representing processes to be signalled

Should there be a single-request version of this, and the above renamed MPI_SIGNAL_ALL?

Note that MPI_SPAWN can be written in terms of the lower-level routines. For example,

```
MPI_SPAWN(comm,archtype,num,hostnames,executables,argvecs,flag,newcomm)
```

can be written as

```
MPI_IALLOCATE(num,advice,archtype,hostnames,executables,MPI_HARD,requests)
MPI_WAITALL(num,requests,statuses)     /* to get processors allocated */
MPI_STARTALL(num,requests)             /* to start processes */
MPI_WAITALL(num,requests,statuses)     /* to wait for processes to start */
MPI_REMOTE_ATTACH(comm,num,requests,newcomm) /* to create communicator */
```

Do we need two spawns, one to create new intercommunicator and another just to expand existing, or just rely on merging in the second case?

Is there a way to deliver a signal to a process created with MPI_SPAWN, where there is no request object?

3.2.5 Clients and Servers

The new functions needed for client-server applications begin with routines needed so that the clients can find the server. The first one is called by the server in order to announce that it is ready to accept connections. It provides an array of request that it can test and wait on to tell whether a client wishes to connect.

MPI_IACCEPT(name, num_requests, array_of_requests)

IN	name	well-known name by which the server can be contacted (string)
IN	num_requests	number of requests in array (int)
INOUT	array_of_requests	requests representing "ports" on which clients can connect

The next routine is called by the client in order to make contact with the server.

MPI_ICONTACT(name, request)

IN	name	well-known name by which the server can be contacted (string)
INOUT	requests	request that is satisfied when a server accepts the connection

The above are both non-blocking calls, to allow each process to overlap computation with the possibly time-consuming task of establishing the connection.

The sequence of events for a sequential client contacting a parallel server might look like this:

```
            Client                                     Server
            ------                                     ------

MPI_Icontact(server_name,request)         MPI_Iaccept(myname,num,requests)
MPI_Wait(request,status)                  MPI_Waitsome(num,requests,numready,
                                                        which,statuses)
MPI_Remote_attach(comm,1,request,newcomm) MPI_Remote_attach(comm,1,request[],newcomm)
( MPI communication in newcomm )          ( MPI communication in newcomm )
MPI_Comm_free(newcomm)                    MPI_Comm_free(newcomm)
MPI_Finalize()                            (process other requests, loop back
                                             to accept again)
(exit)
```

The following are the higher-level, blocking calls. (These should be moved to near MPI_SPAWN).

MPI_CLIENT_CONNECT(name, mycomm, newcomm)

IN	name	well-known name by which the server can be contacted (string)
IN	mycomm	communicator of the client, over which this call is collective
OUT	newcomm	new communicator, which includes the server

MPI_SERVER_CONNECT(name, mycomm, newcomm)

IN	name	well-known name by which the server can be contacted (string)
IN	mycomm	communicator of the server, over which this call is collective
OUT	newcomm	new communicator, which includes the client

The connections are taken down with MPI_DETACH.

3.3 Comparisons with PVM routines

PVM has popularized a set of process-management functions. In this section we compare the functionality of the proposed MPI routines with the functionality provided by PVM.

The philosophical difference is that PVM essentially allows the user program to assume the functions of the job scheduler and process manager. This is a good thing when these functions are not provided in the operating environment, which is often the case on workstation networks. However, we would like MPI programs to mesh smoothly with existing schedulers and process managers where they exist (as on most MPP's), as well as provide mechanisms for them to be written in MPI itself when that is required.

(Discussion of PVM's process management. Problems with race conditions due to spawn not being collective. Handling of dynamic groups. Usefulness of MPI's communicators. What corresponds to the "Virtual Machine"?)

4 Complete, Portable Applications

- Overbeek's molecular biology task farm
- Virtual Reality CAVE client/server
- Some dynamic sizing: maybe a climate model setup-compute-teardown

5 Summary

This section will describe initial reactions to this proposal, and discuss the possibility of trial implementations (as MPE routines) in MPICH.

References

[1] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[2] The MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Los Alamitos, California, November 1993. IEEE computer Society Press.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.