Dynamic Process Management in an MPI Setting

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

April 20, 1995

DRAFT

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. No changes to the existing MPI Standard are proposed, thus all present MPI programs will run unchanged.

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, 6]. 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 migrating from PVM to MPI are accustomed to using PVM's capabilities [4] for process management. (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

*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.

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 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.

We do not provide C or Fortran bindings in this document, instead using the same definition style used in the MPI standard itself.

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. MPI defines a standard interface for message-passing libraries.

Security An important function of the runtime environment 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 propose some minimal features designed to enhance security.

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, an IBM scheduler), 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.

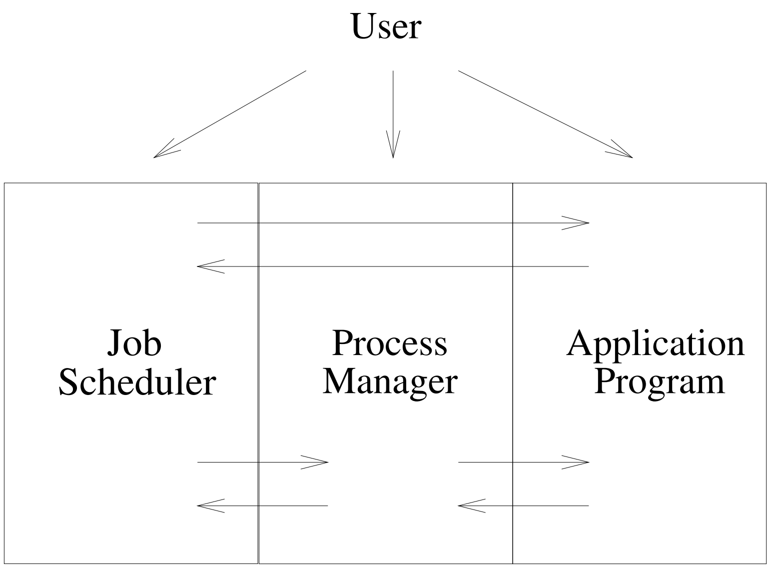

Figure 1: Structure of the Runtime Environment

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, including two based on MPI.

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 There are three schedulers for the Paragons operated by the CSCC at Caltech. The first two are for interactive use. Programs may be started by simply giving the number of nodes as an argument or by creating a named partition of a particular shape and then running within that partition. System calls to create partitions and run programs are provided. Partitions may be gang-scheduled.

The other is the NQS batch system, which is used during the production shift (evenings and weekends). Users submit jobs to a particular queue; NQS allocates the necessary resources and starts jobs. The jobs are usually shell scripts because they start in the user's home directory; a script is necessary to run a program in a different directory.

Workstation network managed by DQS DQS [5] 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 other parallel programs.

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. A user creates a "virtual machine" by starting "daemons" on an assortment of machines and then schedules jobs to run on it and manages his processes with the help of of the daemons. The virtual machine itself can be reconfigured from inside the user program. A difficulty with this approach is that the user is assumed to have the necessary permissions to execute such functions. This may be the case on a workstation network, but seldom on an MPP. 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.

2.3 Applications Requiring Direct Communication with the Runtime System

The existing MPI specification is adequate for most 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). In many cases this harness can be written in a simple scripting language like `csh` or `perl`, but some users prefer to use Fortran or C.

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 also 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 currently possible to write the parallel clients and servers in MPI, but because MPI does not provide the necessary interfaces between the application program and the job scheduler or process manager, other nonportable, machine specific libraries must be called in order for the client and server to communicate with one another. On the other hand, MPI does contain several features that make it relatively easy to add such interfaces, and we propose both a simple interface and a more complex but flexible one.

3 Extending MPI for Process Management

In this section we will first describe requirements for the interface which influence some of the decisions. Then we will (finally) propose a set of MPI extensions that will meet the requirements. Note that we think of ourselves as providing an interface to existing job scheduling and process management systems. If they do not exist, then we may want to be able to write them in MPI. Some proposals for spawning new processes in an "MPI way" have previously been made in [3], [8] and [6]. Our proposals here offer considerably more functionality and flexibility.

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. In MPI, every process is a member of some communicator; when we allow MPI to create or destroy processes, all of the communicators that that process belongs to change. In order to keep collective operations on communicators meaningful (for example, what does a reduction mean when a process joins the reduction during the operation; for that matter, how is "during" defined), all changes to communicators are collective operations. In PVM terms, we will not allow a new process to join a group while a collective operation over that group is in progress. (Error handling is dealt with separately.)

Scalability and performance It must be possible to deal with large numbers of processes by exploiting potential scalability in the job scheduler or process manager. In addition, since each of the steps of allocating resources and starting processes can be very time consuming, we allow each of these steps to be non-blocking so that other work can take place during these steps.

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 and thus add a rich collection of non-blocking operations for process management.

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_Start, MPI_Test and MPI_Wait operations.

The MPI extensions we need will be dictated by the view we have taken of the runtime environment as consisting of a job scheduler, process manager, and communication library. To consider client-server applications we must also consider existing running parallel jobs as part of the environment as well.

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 MPI Inter-communicators

Since inter-communicators are sometimes thought of as one of the "exotic" features of MPI, in this section we briefly remind the reader of the main features of MPI's inter-communicators. For more on inter-communicators, see [7].

What we most typically refer to as a *communicator* in MPI is more precisely referred to as an *intra-communicator*. It encapsulates a context and a single group of processes, and all processes mentioned in communication operations that use this communicator are referred to by their rank in the group associated with the communicator.

An *inter-communicator* encapsulates a context and *two* groups of processes, referred to (from the point of view of a process belonging to the inter-communicator's local group) as the *local* group and the *remote* group. The local group is the "normal" group of the (inter-)communicator, whose size is returned by MPI_COMM_SIZE. The remote group is another group of processes, whose size is returned by MPI_COMM_REMOTE_SIZE. When a process is referred to by rank in an communication operation using the inter-communicator, that process is the one with that rank in the remote group. Thus intercommunicators are used precisely when we want to communicate between processes in different groups, using their ranks in their respective groups to identify them.

The two groups can easily be "joined" into an intra-communicator whose (only) group is the union of the two groups, by the collective operation MPI_INTERCOMM_MERGE.

We will use inter-communicators as a way to manage the distinction between two groups of processes when one group collectively creates the other group (spawning) or else wants to establish communication with an existing group (client-server). Details are given in the following sections.

3.2.2 Simple Interface to Process Manager

In this section we provide high-level functions both for process creation and for client-server requirements that are the easiest to use. They encapsulate interactions with all three components of the runtime environment at once, and are blocking calls. These provide a useful subset of operations; routines that provide more control and flexibility are described later.

The following is the simplest way to create new processes, and is used when one wants simply to expand the number of processes in a communicator. It makes several simplifying assumptions: that the existing processes are all executing the same executable on machines of the same architecture, and that the new processes do not need any new arguments. That is, this call is for expanding the number of processes in an SPMD computation. It assumes that the job scheduler can identify processors to run the new processes on without help from the user program.

MPI_COMM_MODIFY(oldcomm, count, newcomm)

IN	oldcomm	communicator of spawning group
IN	count	number of processes to change by (positive or negative)
OUT	newcomm	new intra-communicator with larger number of processes

This operation is collective over oldcomm. If count is positive, the spawned processes come into existence with an MPI_COMM_WORLD that includes both the processes in oldcomm and the count processes being spawned by this call. In newcomm the new processes are added with ranks higher than those of the original processes, or subtracted from the processes with the highest ranks. This call does not require any understanding of inter-communicators.

A process can determine if it was created by testing MPI_COMM_PARENT against MPI_COMM_NULL. The created processes begin execution at the beginning of their main program, just as if they had been started by the user.

Discussion: We could have two functions MPI_COMM_EXPAND and MPI_COMM_CONTRACT, but it seems simpler just to combine these.

Discussion: An alternative uses an INOUT communicator and a count that is significant only at the spawners.

The following function is the next-easiest way to create new processes, and is most like the PVM routine `pvm_spawn`. It starts 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 inter-communicator newcomm, which has the new processes as the remote group. If an expanded communicator is desired, an ordinary intra-communicator containing all processes can then be constructed using MPI_INTERCOMM_MERGE. Only one of the processes in oldcomm must supply the arguments that describe the new processes; that process is designated by root.

MPI_SPAWN(oldcomm, root, arch_type, count, array_of_names, executable, argvector, flag, newcomm)

IN	oldcomm	communicator of spawning group
IN	root	rank of process supplying following arguments
IN	arch_type	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 (int)
OUT	newcomm	new inter-communicator including new processes as the remote group

The spawned processes are created with MPI_COMM_WORLD consisting of the processes spawned with this call, and in addition they have a predefined inter-communicator MPI_COMM_PARENT, in which the remote group consists of the spawning processes. The call blocks until all processes in oldcomm have called MPI_Spawn and all the spawned processes have called MPI_INIT, after which communication is possible. (We treat spawning of non-MPI processes below.) Note that this means that MPI_COMM_WORLD is different on the spawners and spawnees. Also note that by using MPI_COMM_SELF, a single process can create many others; this is useful both for master-slave programs and for dynamic sizing of parallel jobs from a single initial process. This function is similar to the one described in [8], but here the arguments are made explicit instead of being combined in a single string. In the case where job-scheduler-dependent information must be supplied, we do use a string, in the more flexible MPI_IALLOCATE described in Section 2.1. The array_of_names is an implementation-defined string that defines a particular processor. The value * specifies a processor of the same architecture type as given by the arch_type argument.

The arch_type is an implementation-defined string that defines a particular architecture. Two processors have the same architecture if the can run the same executable. The value * for architecture allows any architecture (this requires executables that can execute on any architecture, or executable names that can be parameterized by architecture). This string is significant only if array_of_names contains one or more entries with the value *.

> **Discussion:** The use of implementation-dependent strings may worry those interested in portability. We believe that, since these strings are used to interface to the outside world, they can not be completely defined. Filenames provide an example of a non-portable string; users are used to having to change filename when moving from system to system; these strings are really no different.

For client-server applications, we propose the following as the simplest, blocking calls.

MPI_CLIENT_CONNECT(mycomm, name, newcomm)

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

MPI_SERVER_CONNECT(mycomm, name, newcomm)

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

Disconnection occurs when processes call MPI_COMM_FREE on the inter-communicator.

Then any process in the client group can communicate with any process in the server group and vice versa, using the inter-communicator.

The form of the name argument has several possibilities. The most obvious is to use the `net-address:port-number` format that current systems will find most straightforward. However, in the long run, name servers of various kinds may require more flexibility.

3.2.3 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. A general mechanism for user definition of request types was described in [3], and our proposal here is quite similar although not identical. Because an implementation of extensions to MPI_Request provides a consistent way to implement the functions that we propose, it is useful to provide a standard extension to MPI for user-defined request types. However, since we do not propose here to use this general mechanism for creating and starting the requests used here, we relegate this proposal to the Appendix.

3.2.4 Interfacing to the Job Scheduler

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.

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 keeping with other MPI non-blocking operations, we call it IALLOCATE instead of IALLOCATE.

> **Discussion:** 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 cannot 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 that will create the application processes in response to one of the process-creation functions described here (like MPI_Spawn).

The following function is for resource allocation only; it does not start any processes.

MPI_IALLOCATE(num_requested, global_js_dep_string, array_of_local_js_dep_strings, arch_type, array_of_nodenames, hardness, array_of_requests)

IN	num_requested	number of hosts requested
IN	global_js_dep_string	special information parsed by job scheduler
IN	array_of_local_js_dep_strings	special information parsed by job scheduler, on a per-process basis
IN	arch_type	architecture type of machine to spawn on
IN	array_of_nodenames	array of hostnames (array of strings)
IN	hardness	whether the request is hard or soft (integer)
OUT	array_of_requests	set of requests

Authentication information, if required by the job scheduler, can be supplied in the job-scheduler-dependent string. The array of nodenames may contain wildcard indicators (MPI_ANYWHERE) 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 complete when it can allocate some processors even if it knows that it will not be able to satisfy the entire request. The executable files are not specified on this call, since this is just for resource allocation. We can attach them to the requests with the function described in the next section.

The strings containing job scheduler dependent strings are implementation defined (by the job scheduler, not MPI!). These may contain information to be applied to all processes (in global_js_dep_string) and information for a particular process (in array_of_local_js_dep_strings). An example of global_js_dep_string is `gang_schedule`; and example of array_of_local_js_dep_strings is

```
{ "large_memory",
  "local_hippi" }
```

The arguments arch_type and array_of_hostnames have the same meaning as for MPI_Spawn.

3.2.5 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 steps. 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 functions we present in this section, and then actual process startup can be accomplished with the existing MPI_START or MPI_STARTALL calls.

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

MPI_SET_EXEC(request, executable)

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

sets the name of the file to be executed, and

MPI_SET_ARGS(request, args)

IN	request	request (handle)
IN	args	array of strings

sets the command-line arguments that process will receive.

Discussion: These same operations perhaps should also be defined for arrays of requests. On the other hand, they are not expensive operations.

Discussion: Following the reasoning for MPI_PUT_ATTR, the request is considered an IN argument instead of an INOUT argument.

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 or MPI_STARTALL routines,

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 the second stage 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 determine which state has been completed. Note that this level of process management allows us to manage non-MPI processes, since communicators are not involved. In order to get the status, we need a routine to extract the value from the status argument to the MPI_WAIT call.

MPI_GET_RETURN_CODE(request, code)

IN	request	request (handle)
OUT	code	exit status of process

to retrieve the exit status (from a `return n` or `exit(n)` in C or `STOP n` in Fortran).

If the started processes are MPI processes (that is, if they call MPI_INIT), then the communicator that includes them can be constructed with the MPI_COMM_PARENT_CREATE function described below. In either case, at this stage we have allocated resources and the processes have been started. The third component of the runtime environment is the message-passing library. In order to communicate with these processes, we must interface with the message-passing library. In MPI terms, this means setting up communicators.

Once the processes have been started, we may or may not wish to establish communication with them. Note that the MPI_IALLOCATE/MPI_START/MPI_WAIT mechanism can be used by a process manager written in MPI to manage non-MPI processes as well as MPI processes. If the application program is creating new processes, however, it is likely that it will want to communicate with them, via an inter-communicator. The complication is that creation of this inter-communicator is a collective operation over all processes involved, yet we don't want the spawning process(es) to block if the spawned processes are not even going to call MPI_INIT. Our solution is to create a "stub" inter-communicator which will become valid if and when the spawned processes call MPI_INIT.

MPI_COMM_PARENT_CREATE(localcomm, num_requests, array_of_requests, intercomm)

IN	localcomm	communicator of the spawning processes
IN	num_requests	number of requests in array
IN	array_of_requests	requests representing processes to be created
OUT	intercomm	new inter-communicator, which may become MPI_COMM_PARENT in the created processes.

After MPI_WAITALL indicates that all of these processes are running, the inter-communicator is valid on the spawning side and the spawned processes have the inter-communicator MPI_COMM_PARENT defined, along with the usual MPI_COMM_WORLD.

It is erroneous to use MPI_COMM_PARENT_CREATE with processes that are not MPI jobs.

> **Discussion:** An alternative would be to have the inter-communicator returned as a field in the status object when the requests have reached the state where the processes have been started. This eliminates the need for MPI_COMM_PARENT_CREATE at the cost of complexifying the status object, which is a user-visible structure. It also does not solve the problem of specifying the intent of the spawned processes — are they MPI processes that will be communicated with using MPI calls or are they stand-alone programs (either MPI programs or non-MPI programs).

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

It is the responsibility of an implementation to translate between signals; in other words, a `SIGINT` that has value `3` on system A must be delivered as a `SIGINT` on system b, even if `SIGINT` on system b uses the value `5` for `SIGINT`. If the signal can not be delivered because there is no corresponding signal, the error code is MPI_ERR_INVALID_SIGNAL.

> **Discussion:** Should there be a single-request version of this, and the above renamed MPI_SIGNAL_ALL? Is the notion of signals too Unix-centric?

> **Discussion:** Should there be a way to deliver a signal to a processes that has been spawned, so that there is no request object? It would be a collective operation on an inter-communicator.

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

```
MPI_SPAWN(comm,arch\_type,num,hostnames,executable,argvecs,flag,newcomm)
```

can be written as (without error handling)

```
MPI_IALLOCATE(num,advice,(char **)0,arch_type,hostnames,MPI_HARD,requests)
for (i=0;i<num;i++) {
    MPI_SET_EXEC( requests[i], executable );
    MPI_SET_ARGS( requests[i], argvecs );
    }
MPI_WAITALL(num,requests,statuses)    /* to get processors allocated */
MPI_COMM_PARENT_CREATE(comm,num,requests,newcomm) /* get communicator */
MPI_STARTALL(num,requests)            /* to start processes */
MPI_WAITALL(num,requests,statuses)    /* to wait for processes to start */
```

3.2.6 Clients and Servers

In a previous section we described simplified, blocking calls for a client and server. In general, just as MPI needs non-blocking message-passing operations, we need non-blocking operations.

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 requests that it can test and wait on to tell whether a client wishes to connect. We assume here that both client and server may be parallel programs, whose processes are already in internal communication via a communicator. Both of the following operations are collective over the communicator comm.

MPI_IACCEPT(comm, name, num_requests, array_of_requests)

IN	comm	server's communicator
IN	name	well-known name by which the server can be contacted (string)
IN	num_requests	number of requests in array (int)
OUT	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(comm, name, request)

IN	comm	client's communicator
IN	name	well-known name by which the server can be contacted (string)
OUT	request	request that is satisfied when a server accepts the connection

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

Discussion: We propose building in a certain level of security into the client-server connection process in the following way. An extra argument to MPI_IACCEPT would consist of the name of a function to be called to provide validation for the client that is attempting to connect. The client, in an extra argument on the call to MPI_ICONTACT, would provide a key that was validated by the routine passed to MPI_IACCEPT. If the key was not validated, the request would have no effect on the requests supplied to MPI_IACCEPT.

It is useful if it is not required that the name argument to the above calls be known until execution time. The following routine provides a way for the application to request a name, but be given a different one if the system prefers it that way.

MPI_GET_SERVER_NAME(requested_name, given_name)

IN	requested_name	name that the server would like to be known by
OUT	given_name	name supplied by the system

In order for communication to take place, an inter-communicator must be created connecting the client with the server. The following routine serves this purpose.

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 inter-communicator for the new processes

This operation is collective over oldcomm, and returns an inter-communicator One can think of this operation as very much like MPI_INTERCOMM_CREATE, where the remote processes are represented by an array of requests rather than by another group.

Discussion: One option would be to combine this call with the previous ones, but that would require that the inter-communicator being constructed be in an undefined state between the request and the time that the requests were satisfied. Still another option would be to have the inter-communicator returned as part of the status object on the MPI_WAIT. We have chosen to make separate the process of constructing the inter-communicator after the MPI_WAITs have been completed.

We don't need MPI_REMOTE_DETACH(intercomm)

IN	intercomm	communicator

because its function can be accomplished with MPI_COMM_FREE.

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,      MPI_Remote_attach(comm,1,request[],
                  newcomm)                               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)
```

3.3 Comparisons with PVM routines

(This section is still incomplete.)

PVM contains a simple set of process-management functions. In this section we compare the functionality of the proposed MPI routines, operating in conjunction with a job scheduler and process manager, with the functionality provided by PVM. (The job scheduler and process manager may or may not be supplied as part of an MPI implementation.) The PVM version here is Version 3.3 as described in the recent book [4].

The philosophical difference is that PVM essentially forces the user program to assume the functions of the job scheduler and process manager. This is acceptable when these functions are not provided in the operating environment, which is often the case on workstation networks. However, it does not work well on MPP's. In particular, the notion of a PVM *daemon*, a system process set running in advance on every machine that the user's program will use, is very much a part of the system interface. For example, there is is a system call to obtain the PVM process identifier of the daemon associated with any process. In MPI, such a daemon process, if it exists, is considered to be part of the implementation, not part of the standard interface specification.

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.

We briefily list here the PVM routines related to job scheduling and process management. In most cases they communicate explicitly with the PVM daemon, which in turn provides job scheduling and process management functions. In order to provide alternatives to the fixed way in which the daemon performs these tasks, the daemon can be directed to call the `pvm_reg_` routines to carry them out, but the daemon itself cannot be replaced.

`pvm_mytid`	`enrolls process in VM`
`pvm_exit`	`disenrolls process in VM`
`pvm_spawn`	`creates new processes`
`pvm_catchout`	`manages stdout from remote processes`
`pvm_parent`	`find spawner`
`pvm_tid_to_host`	`enquire where process is running`
`pvm_config`	`obtain scheduling and process manager info`
`pvm_reg_rm`	`interpose own resource manager`
`pvm_reg_hoster`	`interpose own host selector`
`pvm_reg_tasker`	`interpose own process creator`

```
pvm_tasks             enquire about other processes
pvm_addhosts          start more daemons on other machines
pvm_delhosts          remove daemons on other machines
pvm_sendsig           send a signal to a process
pvm_setopt            set options for process manager
pvm_getopt            enquire about options for process manager
```

It is worth mentioning here also the PVM dynamic group management routines, which are not collective:

```
pvm_joingroup
pvm_lvgroup
```

The semantics of these routines require the existence of a "group server" process that keeps track of which processes are members of which groups. Collective operations such as `pvm_broadcast` and `pvm_reduce` take place over these dynamic groups, but the responsibility for ensuring reasonable semantics for these operations belongs to the user, since processes can appear and disappear at any time, and can join or leave groups without notifying other processes. MPI avoids these problems by requiring that group management routines, and the above dynamic process management routines, be collective operations over the appropriate communicators.

4 Complete, Portable Applications

This section will eventually include code fragments from running applications, once we have implemented (as MPE routines) the functions described here.

Dynamic sizing example. Here we give a very simple example of changing the size of MPI_COMM_WORLD during the run.

```
#include "mpi.h"
main(int argc, char **argv)
{
int n;
MPI_Init( &argc, &argv );
if (MPI_COMM_PARENT == MPI_COMM_NULL) {
    puts( "Number of processors?" );
    scanf( "%d", &n );
    }
MPI_Comm_modify( &MPI_COMM_WORLD, n );
/*
   Parallel code, using MPI_COMM_WORLD
 */
MPI_Finalize();
}
```

Task farm example. This is a fairly sophisticated example. It always keeps a request for ten nodes outstanding, but starts jobs as soon as possible. To avoid spin-waits on the allocation and running of jobs, it uses MPI_WAITSOME on an array of requests that includes both allocation requests and started jobs. The index `alloc_top` gives the number of allocation requests currently active; `r_top` gives the total number of active requests (both allocations and started processes).

The programs in this example are *not* MPI jobs; MPI is simply being used to start and manage the programs. For simplicity, we have not included any code to decide when the program is done or to describe the program to be run and its arguments. Note that MPI_CANCEL can be used to cancel any unneeded allocation requests.

```
#include "mpi.h"
main( int argc, char **argv )
{
MPI_Request r[20];
MPI_Status  s[20];
int         idx[20], nout;
int         alloc_top, r_top;
int         rc;


MPI_Iallocate( 10, (char *)0, (char **)0, "*", (char **)0, MPI_HARD, r );
alloc_top = 10;
r_top     = 10;
while (!done) {
    MPI_Waitsome( r_top, r, &nout, idx, s );
    for (i=0; i<nout; i++) {
        if (idx[i] < alloc_top) {
            /* Processor is ready.  Start program */
            j = idx[i];
            MPI_Set_exec( r[j], program_name );
            MPI_Set_args( r[j], program_args );
            MPI_Start( r[j] );
            r[r_top] = r[j];
            r[j] = r[alloc_top];
            r[alloc_top] = MPI_REQUEST_NULL;
            alloc_top--;
            }
        else {
            /* Program has finished */
            j = idx[i];
            MPI_Get_return_code( &s[i], &rc );
            /* Make use of return code ... */
            /* Note that r[j] is MPI_REQUEST_NULL already
               (the wait does it) */
            }
        }
```

```
    /* Repack request array and issue additional allocations */
    j = alloc_top;
    for (i=alloc_top; i<r_top; i++) {
        if (r[i] != MPI_REQUEST_NULL)
            r[j++] = r[i];
        }
    r_top = j;
    MPI_Iallocate( 20 - r_top, (char *)0, (char **)0, "*", (char **)0,
                   MPI_HARD, r + r_top);
    }
MPI_Finalize();
return 0;
}
```

Client-server example. Connecting the SP2 to the SGI Onyx for display of application results in the CAVE virtual reality environment.

This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected.

Here is the server. It accepts a single connection and then processes data until it receives a message with tag `1`. A message with tag `0` tells the server to exit.

```
#include "mpi.h"
main( int argc, char **argv )
{
MPI_Comm client;
MPI_Status status;
double buf[MAX_DATA];
int    again;

MPI_Init( &argc, &argv );
while (1) {
    MPI_Server_connect( "cave:1234", MPI_COMM_WORLD, &client );
    again = 1;
    while (again) {
        MPI_Recv( buf, MAX_DATA, MPI_DOUBLE, 0, MPI_ANY_TAG,
                  client, &status );
        switch (status.tag) {
            case 0: MPI_Comm_free( &client );
                    MPI_Finalize();
                    return 0;
            case 1: MPI_Comm_free( &client );
                    again = 0;
                    break;
            case 2: /* do something */
            ...
            default:
```

```
                    MPI_Abort( MPI_COMM_WORLD, "Unexpected message type" );
                }
            }
        }
}
```

Here is the client.

```
#include "mpi.h"
main( int argc, char **argv )
{
MPI_Comm server;
MPI_Status status;
double buf[MAX_DATA];
int    again;

MPI_Init( &argc, &argv );
MPI_Client_connect( "cave:1234", MPI_COMM_WORLD, &server );
while (!done) {
    tag = 2; /* Action to perform */
    MPI_Send( buf, n, MPI_DOUBLE, 0, tag, client );
    /* etc */
    }
MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, client );
MPI_Comm_free( &client );
MPI_Finalize();
}
```

If the server needs to manage multiple connections at once, it can use MPI_IACCEPT instead. The client need not be changed.

5 Summary

We list here the new functions proposed here for MPI, in several categories:

Simple Version

```
MPI_COMM_MODIFY
MPI_SPAWN
MPI_CLIENT_CONNECT
MPI_SERVER_CONNECT
```

Complete Version

```
MPI_IALLOCATE
MPI_SET_EXEC
MPI_SET_ARGS
MPI_GET_NODENAME
MPI_GET_JSINFO
MPI_START_ALL
MPI_COMM_PARENT_CREATE
MPI_REMOTE_ATTACH
MPI_SIGNAL
MPI_IACCEPT
MPI_ICONTACT
```

We are actively seeking comments on this proposal with the intention of making it clean, complete, and scalable. We are currently building the necessary support for these routines into the substructure of the MPICH implementation of MPI. This will allow us and others to experiment with versions of the routines described here. In order to ensure that no one confuses them with the official MPI Standard, they will appear with the MPE_ (instead of MPI_) prefix in MPICH. If and when an expanded MPI specification for dynamic process management is adopted, then MPICH will, of course, follow the Standard.

References

[1] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[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] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, Jean-Pierre Prost, and Marc Snir. MPI on IBM SP1/SP2: Current status and future directions. unpublished.

[4] 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.

[5] Tom Green and Jeff Snyder. DQS, a distributed queuing system. Technical Report FSU-SCRI-92-115, Florida State University, August 1992.

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

[7] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical report, Mississippi State University — Dept. of Computer Science, April 1994. Draft version.

[8] Anthony Skjellum, Nathan E. Doss, Kishore Viswanathan, Aswini Chowdappa, and Purushotham V. Bangalore. Extending the message passing interface (MPI). Mississippi State University, 1994.

Appendix A

Here we propose a set of operations for managing user-defined requests. It may be possible to layer the proposals described here on top of such requests. This approach strikes us as possible, but cumbersome for users. It does, however, suggest an implementation strategy.

Since we will need to determine the type of a request, we need something like:

MPI_REQUEST_TYPE(request, type)

IN	request	request being queried (handle)
OUT	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, create_function, start_function, test_function, wait_function, free_function)

OUT	type	new type being defined (int)
IN	create_function	function to be called to create this request
IN	start_function	function to be called to start the operation represented by this request
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 above call defines new request types. We also need to create them, with something like:

MPI_REQUEST_CREATE(type, private_data, request)

IN	type	type if request being defined (int)
IN	private_data	user data to be attached to this request
OUT	request	the new request

private_data is a data structure that contains whatever data is necessary to maintain the function of this request. We can get it back from the request with:

MPI_REQUEST_GETDATA(request, private_ data)

IN	request	an existing request
OUT	private_data	user data attached to this request

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

A similar, but not identical, mechanism has been proposed in [3].

Appendix B

There are two approaches to providing encryption of messages to enhance security of MPI applications running on a network. In many cases, the underlying transport mechanism (at least for some of the communications) might have a hardware encryption mechanism. We propose a built in communicator attribute MPI_ENCRYPT that requests that messages sent in that communicator be encrypted if possible. A more flexible approach is provided by supplying an arbitrary user function to be called to filter messages being sent and received. This filter is also attached to a communicator as an attribute MPI_FILTER. No new MPI functions are required for this functionality; only reserved attribute key values.