Dynamic Process Management in an MPI Setting

W. Gropp and E. Lusk*

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

We describe an architecture for the runtime environment for parallel applications as prelude to describing how parallel application might interface to their environment in a portable way. 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 and connection to existing processes to support client/server applications. 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, 5]. 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 [3] 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 third is that with such extensions it would be possible to write major parts of the parallel programming environment in MPI itself.

*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.

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 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 a summary of the types of functions now being considered by the MPI Forum. In the conclusion we summarize the current 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 [7], DQS [4], and LoadLeveler (IBM's scheduler for the SP-2). 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.

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.

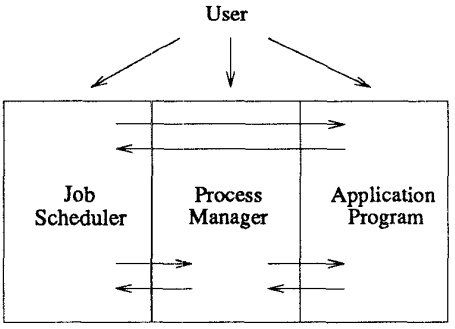

Figure 1: Structure of the Runtime Environment

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 [4] 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 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. Some process-management extensions to PVM are described in [6].

Workstation network with CARMI The Condor system at the University of Wisconsin has been an early progenitor of dynamic process-management systems. A recent, sophisticated, related system is CARMI, described in [8]. It currently supports PVM application programs.

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.

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 Dynamic Process Management

In this section we will first describe requirements for the interface which influence some of the decisions. Then we will describe very generally the families of new 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 [2], [9] and [5]. 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.

3.2 Overview of Mew MPI Functions

The original version of this paper contained a detailed proposal for new MPI functions to meet these requirements. Such details are omitted here both because of space limitations and because the MPI Forum has begun meeting since then and has evolved that original proposal into a new current draft. In particular, many important new contributions have been made by Bill Saphir of the NASA Ames Research Center, and considerable work has been done by Al Geist of Oak Ridge National Laboratory. The MPI Forum's discussion groups are open, and anyone desiring to keep abreast of the current discussion can do so by sending the message "`subscribe mpi-dynamic`" to `majordomo@mcs.anl.gov`. Here we give a brief overview of the evolving specification, as it stands in the summer of 1995.

Interactions with the job scheduler will occur through functions that allow the user program to discover pre-allocated resources and to request new ones. Interaction with the process manager occur through functions that create processes on allocated resources. It will possible to start both non-MPI process and MPI processes, thus allowing a distributed process manager itself to be written in MPI. It will also be possible to establish MPI communications with the new processes. We will use MPI `inter-communicators` as a way to manage the distinction between two groups of processes when one group collectively creates the other group (creating new processes) or else establishes communication with an existing group (client-server). It is likely that there will be a convenience function that combines resource allocation, process creation, and communication establishment through an MPI_SPAWN that is similar in functionality to `pvm_spawn`. Functions will also be provided to establish MPI communication among independently started processes, allowing parallel client-server applications.

4 Summary

We have outlined an approach to dynamic process management in MPI, focusing on the environments in which dynamic process management takes place and some of the types of applications that will use it. We concluded with a brief summary of the current scope of discussions now going on in the MPI Forum.

References

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

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

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

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

[6] R. Konuru, J. Casa, R. Prouty, S. Otto, and J. Walpole. A user-level process package for PVM. In *SHPCC94*, pages 48–55. IEEE Press, May 1994.

[7] M. Litzkow and M. Solomon. Supporting checkpointing and porcess migration outside the Unix kernel. *Usenix Winter Conference*, 1992.

[8] Jim Pruyne and Miron Livny. Parallel processing on dynamic resources with carmi. (submitted for publication).

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