A Scalable Process-Management Environment for Parallel Programs*

Ralph Butler[1], William Gropp[2], and Ewing Lusk[2]

[1] University of North Florida
[2] Argonne National Laboratory

Abstract. We present a process management system for parallel programs such as those written using MPI. A primary goal of the system, which we call MPD (for multipurpose daemon), is to be scalable. By this we mean that startup of interactive parallel jobs comprising a thousand processes is quick, that signals can be quickly delivered to processes, and that `stdin`, `stdout`, and `stderr` are managed intuitively. Our primary target is parallel machines made up of clusters of SMPs, but the system is also useful in more tightly integrated environments. We describe how MPD enables much faster startup and better runtime management of MPICH jobs. We show how close control of `stdio` can support the easy implementation of a number of convenient system utilities, even a parallel debugger. MPD is implemented and freely distributed with MPICH.

1 Introduction

A parallel programming environment may be viewed as comprising three interacting components: a *job scheduler*, which decides what resources a parallel job consisting of multiple processes will run on; a *process manager*, which starts and terminates processes and provides them with a number of services; and a *parallel library* such as MPI, which a parallel application calls upon for communications. Since these components need to communicate with one another, they are often integrated into a single system. An important research question is to what extent they can be separated from one another with well-defined interfaces so that they can be independently developed. A further research question is whether the resulting system can be made scalable to jobs involving thousands of communicating processes. In this paper we focus on the process manager component. We describe a design and an implementation we call MPD (for multipurpose daemon) that provides both fast startup of parallel jobs and a flexible run-time environment that supports parallel libraries.

In Section 2 we summarize related work. In Section 3 we state our explicit design goals, how these goals lead to implementation decisions, and interesting features of the resulting system, including how it can be used to create a parallel debugger out of an existing single-process debugger. Section 4 summarizes

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

preliminary experiments that make us optimistic about the usefulness of MPD as a process manager for large-scale systems. We conclude with a summary of progress to date and a description of our future plans.

The MPD system is in use and is available as open source as part of the MPICH system, obtainable from http://www.mcs.anl.gov/mpi/mpich.

2 Related Work

All parallel computing environments that support execution of truly parallel programs (those in which any two processes can communicate with one another) have had to address at least some of the issues that we address with MPD. Parallel programming systems, such as PVM [10], P4 [7], and implementations of MPI such as MPICH [13] and LAM [6] all provide some mechanism for starting and running parallel programs, often with a specialized daemon process.

Many systems are intended to manage a collection of computing resources for both single-process and parallel jobs; see the survey by Baker, et. al. [3]. Typically, these use a daemon that manages individual processes, with emphasis on jobs involving only a single process. Widely used systems include PBS [17], LSF [18], DQS [8], and Loadleveler/POE [14]. The Condor system [15] is also widely used and supports parallel programs that use PVM [19]. Other, more specialized systems, such as MOSIX [4] and GLUnix [11], provide a form of single-system image support for clusters.

Harness [5, 16] shares with MPD the goal of supporting management of parallel jobs. Its primary research goal is to demonstrate the flexibility of the "plug-in" approach to application design, providing a wide range of services, whereas the MPD system focuses more specifically on the design and implementation of services required for process management of parallel jobs, including high-speed startup of large parallel jobs on clusters and scalable standard I/O management. The book [9] provides a good overview of metacomputing systems and issues.

3 Design of MPD

In this section we describe our goals in constructing MPD and outline the system's architecture.

3.1 Goals

Several explicit goals have governed the design of the MPD system.

Simplicity The persistent (across jobs) part of the system should be simple and robust. In the long run we expect this part to be runnable as root. If its behavior isn't completely transparent, we will never be able to convince system administrators to run the daemons as root.

Speed Startup of parallel jobs should be quick enough to provide an interactive "feel," so that large but short jobs make sense. Large (in number of processes) but short (in time) characterizes system utilities such as those described in [12]. Our immediate target is to start 1000 processes in a few seconds, while still providing a way for such processes to establish contact with one another. Our long-term goal is to support management of 10,000 processes.

Robustness The persistent part of the system should be at least moderately fault tolerant. An unexpected crash of one machine should not bring down the whole system. There should be no single "master" process.

Scalability The complexity or size of any component should not depend on the number of components.

Individual Process Environments It should be possible to start a parallel job in which the executable files, environment variables, and command-line arguments are different for each process. It should be possible to collect return codes individually from processes.

Collective Identity of a Parallel Job It should be possible to treat a parallel job as a single entity that can be suspended, continued (signaled, in general), or killed collectively as if it were a single process. The system should manage `stdin`, `stdout`, and `stderr` in a useful and scalable way and allow them to be redirected as if the parallel job were a single process. An important component of a job's collective identity is its *termination*. All resources allocated for the job, such as files, System V IPC's, other processes, etc., must be reliably freed, even if the job terminates abnormally.

It is explicitly not a goal of the MPD system to provide scheduling services, which we believe to be a separate function from process management.

3.2 Deriving the Design from the Goals

The goals of simplicity and robustness lead us to adopt a multicomponent system. The *daemon* itself is persistent (may run for weeks or months at a time, starting many jobs), typically one instance per host in a TCP-connected network. *Manager* processes will be started by the daemons to control the application processes (*clients*) of a single parallel job and will provide most of the MPD features. The goal of speed requires that the daemons be in contact with one another prior to job startup, and the goals of scalability and "no master" suggest that the daemons be connected in a ring.[1] The services that the managers will provide (see Section 3.3) suggest that they be in contact as well, and the fastest way for them to form these connections is to inherit part of the ring connectivity of the daemons. Separate managers for each user process support the individual process environments. The goal of having a collective identity for a parallel job leads us to treat the `mpirun` or `mpiexec` process as such a representative, and use it to deliver signals and `stdin` to application processes and collect `stdout`

[1] While a ring is not ultimately scalable, it is more so than the typical star used in many process management systems, and our experiments have shown it feasible for the 1000-daemon domain.

and `stderr` output from them. This suggests that the `mpirun` process connect first to the daemon ring in order to start the job, and then switch the connection to the manager ring in order to control the job. The goal of speed suggests that these latter connections be restricted to a process running on the same host, either the daemon itself or a persistent gateway process if the daemon is run as root, so that authentication can be through the file system (a Unix rather than a network socket). We refer to all such processes as *console commands*. Finally, in order that this infrastructure be available to support MPI programs or other parallel tools, there needs to be *client library* that each application process may use to interact with its manager.

We do not specify how the daemons are started or connected, since the system provides a number of alternatives, and the process need not be particularly fast. A console command is started by the user, either interactively or under the control of a batch scheduler. The daemons fork and exec the managers, which use information given them by the daemons to connect themselves into a ring, then fork and exec the clients. The startup messages traverse the ring quickly, so most forking, execing, and connecting take place in parallel, leading to fast startup even for large jobs. The situation is then as shown in Figure 1, where the

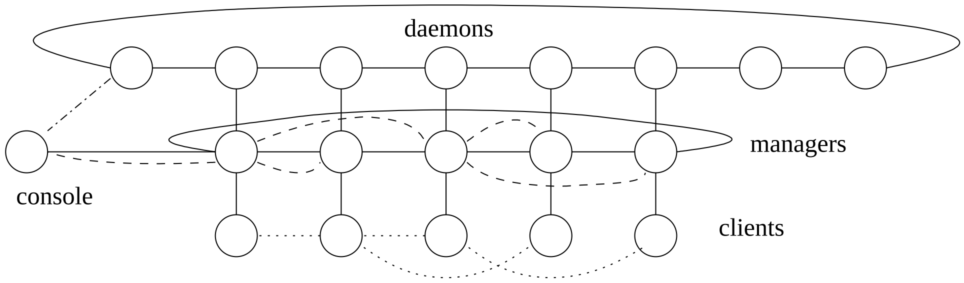

Fig. 1. Daemons with console process, managers, and clients

clients may be application MPI processes. Solid lines represent sockets, except for the vertical ones, which represent pipes. The dashed lines represent the trees of connections for forwarding `stdout` and `stderr`, and the dotted lines represent *potential* connections among the client processes. The dot-dashed line is the original connection from console to local daemon on a Unix socket, which is replaced during startup by the network connection to the first manager.

3.3 Interesting Features

Space restrictions prevent a complete description of all the features and capabilities of the MPD system, but in this section we mention a few highlights.

Security Whenever a process advertises a "listener" socket and accepts connections on it, the possibility exists that an unknown or even malicious process will connect. This is particularly dangerous if the process accepting the connection can start processes as the MPD daemon can. We currently use the "challenge-response" system described in [20]. In the long run, we expect to modify this component of the system to use more elaborate schemes and extend them to other connections such as client/gateway authentication. This will have little impact on the job startup speed since the daemon component startup is separate from job startup.

Fault Tolerance If a daemon dies, this fact is detected and the ring is reknit. This provides a minimal sort of fault tolerance, since the ring remains intact. A new MPD daemon can be inserted in the ring where the old one was, but this process is not (yet) automatic.

Signals Signals can be delivered to client processes by their managers. We currently use this capability in two specific ways. First, signals delivered to a console process are propagated to the clients, so that a parallel application as a whole can be suspended with `cntl-Z`, continued, and killed with `cntl-C`, just as if it were a single process. Second, in the `ch_p4mpd` device in the MPICH implementation of MPI, client processes can interrupt one another with requests to dynamically establish client-to-client connections. Such requests go up into the manager ring from the originating client, around the ring to the manager of the target process, which signals its client.

Support for MPI Implementations Currently MPD provides direct support for the MPICH implementation of MPI. The `ch_p4mpd` device distributed with Version 1.2 of MPICH makes direct calls to the client library component of the MPD system to find out a process's rank, where other processes are and how to contact them, and so forth. In our next major release of MPICH, the support will be indirect, through a general parallel-library-to-process-manager interface we will describe elsewhere.

On clusters of SMPs, it is easy to specify that multiple processes are to be started on the same machine and share memory. Specifically, `mpirun -np 180 -g 2 cpi` starts processes in groups of two and places in their environment a key that can be used to acquire group-attached shared memory and other information needed to set up multimethod communication for an MPI implementation. Other communication mechanisms (such as VIA) will be supported over time.

Handling Standard I/O Mangers capture the `stdout` and `stderr` of their clients, and forward them up a pair of binary trees of socket connections, each manager merging `stdout` and `stderr` from its client with that from each of its two children. A command line option tells the managers to provide a rank label on each line of output from their clients.

Standard input (to `mpirun`, for example) by default is delivered to the client managed by manager 0. This seems to be what most MPI users expect, and what most MPI implementations do. (The MPI standard does not specify.) However, control messages can be used to change this behavior to direct `stdin` to any specific client or broadcast it to all clients.

Client Wrapping The semantics of the Unix `fork` and `exec` system calls provide useful benefits. When a manager forks a client process, for example, it first sets up the manager-client pipes for control messages and standard I/O. The "lower" ends of these pipes are inherited by any process that the client forks. Thus, even though the client is not using any of the client library, managers can manage clients that themselves run the "real" application process. We call this scheme *client wrapping*. Thus `mpirun -np 16 nice -5 myprog` lowers the priority of a parallel job to be run on one's colleagues' workstations, and `mpirun -np 16 pty myprog` can be used when `myprog` needs to be attached to a terminal (otherwise our capture of `stdin` and `stdout` modifies their buffering behavior). (The program `pty` is distributed with the MPD system.)

Putting It All Together The combination of I/O management, especially redirection of `stdin`, line labels on `stdout`, and client wrapping can be surprisingly powerful. We have used these features of the MPD system to add an option to `mpirun` that invokes `gdb` as a client wrapper and dynamically redirects `stdin`. While `mpirun -np 3 cpi` runs `cpi` directly as an MPI job, `mpirun -np 3 -d cpi` runs each `cpi` process under the control of (wrapped by) the `gdb` debugger. (Other sequential debuggers could be used, but are not yet supported.) Thus multiple instances of `gdb` are being run. Output of the `gdb`'s is labeled by process rank. The "`(gdb)`" prompts are intercepted by the `mpirun` process and counted, so that it can issue an "`(mpigdb)`" prompt when one has been received from each process. In addition, `mpirun -d` uses the "`z`" command (one of the few single letters not already claimed by `gdb`) to redirect `stdin` to a specific `gdb` instance or to all processes. Thus processes can be stepped and breakpoints can be set either collectively or individually, and collectively printing a variable will provide all values with rank labels. An example terminal session showing how this works can be seen at http://www.mcs.anl.gov/mpi/mpich/mpd/mpigdb.script.

4 Experiments

Most development of MPD has been on workstation networks where startup of 32-process jobs on five workstations is virtually instantaneous, compared with the approximately 1.5 seconds per process required by the `ch_p4` version of MPICH. An early test of the feasibility of using the ring topology showed that a message could make 1024 hops around the ring in less than .4 seconds, which gave us confidence that the ring would not impose scalability limits, at least in the near term. Recently we began experiments on Chiba City, a testbed for parallel computer science research [1]. We performed one set of tests on 211 nodes connected by Fast Ethernet. We were interested only in process startup time, and so tested execution of trivial parallel jobs. Typical experiments included

```
time mpirun -np 211 hostname
time mpirun -np 422 -g 2 hostname
```

We found that starting 211 processes (one on each node) and collecting the `stdout` output of `hostname` took about 2 seconds to execute. Starting twice as many processes (one for each CPU) took about 3.5 seconds, including setting up the relatively complex `stdout` tree and collecting the output. Sending a message around the ring of 211 MPD daemons took only .13 seconds. More experiments are ongoing, and we will soon be able to report on MPI jobs on Chiba City.

5 Future Development

The existing MPD system, consisting of daemons, managers, console commands, and client library, meets our goals of simplicity, robustness, and scalability. It is used for fast startup of MPI jobs and others on systems with hundreds of machines. The flexibility of its `stdio` control mechanism has provided unexpected benefits, such a "poor man's" parallel debugger. It meets our goals for the collective identity of a parallel job. It does not yet meet all of our goals with respect to individual process environments, although that is coming very soon.

In the near term, we expect to use the system to implement the dynamic process creation part of MPI-2 in MPICH. The design presented here, with a simple daemon and a separate manager process providing most of the features needed by user jobs, allows the daemons to be run as root while the managers are run as user processes. We expect to begin running the daemon as root on some large-scale multi-user systems, in order to provide a persistent job management system. This will require increased attention to security issues as well as a precise definition of how MPD will interoperate with a full-featured scheduling system such as the Maui scheduler [2]. We believe that the MPD daemons can also begin to provide more services, such as run-time performance monitoring.

In the long run, as machines grow from hundreds to thousands of nodes, our rings of daemons and managers may have to grow into a more sophisticated structure, such as rings of rings, in order to continue to provide fast startup. We anticipate that this can be done without substantially changing the MPD design presented here. We will also need a more sophisticated output merger in order to provide scalable `stdout`, for example for large-scale parallel debugging.

In summary, we are finding the MPD system already a useful contribution to one's parallel programming environment and expect its applicability to expand in the near future. We also view its design as a valuable starting point for future research into large-scale parallel job execution environments.

References

1. Chiba City home page. http://www.mcs.anl.gov/chiba.
2. The Maui scheduler home page. http://maui-scheduler.mhpcc.edu/new_doc, http://www.mhpcc.edu/maui.
3. M. A. Baker, G. C. Fox, and H. W. Yau. Review of cluster management software. *NHSE Review*, 1(1), May 1996.
4. Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX distributed operating system: Load balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1993.

5. Micah Beck, Jack J. Dongarra, Graham E. Fagg, G. Al Geist, Paul Gray, James Kohl, Mauro Migliardi, Keith Moore, Terry Moore, Philip Papadopoulous, Stephen L. Scott, and Vaidy Sunderam. HARNESS: A next generation distributed virtual machine. *International Journal on Future Generation Computer Systems*, 15(5/6), 1999.
6. Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.
7. Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.
8. DQS home page. http://www.scri.fsu.edu/~pasko/dqs.html.
9. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
10. 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, Cambridge, Mass., 1994.
11. Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: A Global Layer Unix for a network of workstations. *Software—Practice and Experience*, 28(9):929–961, July 1998.
12. William Gropp and Ewing Lusk. Scalable Unix tools on parallel processors. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 56–62. IEEE Computer Society Press, 1994.
13. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
14. IBM. *Loadleveler: Using and Administering*, version 2 release 1 edition, November 1998. SA22-7311-00.
15. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A hunter of idle workstations. In *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, San Jose, Calif., June 1988.
16. M. Migliardi and V. Sunderam. PVM emulation in the Harness metacomputing system: A plug-in based approach. In J. J. Dongarra, E. Luque, and Tomas Margalef, editors, *Recent advances in parallel virtual machine and message passing interface: 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29, 1999: Proceedings*, volume 1697 of *Lecture Notes in Computer Science*, pages 117–124, Berlin, 1999. Springer-Verlag.
17. PBS home page. http://pbs.mrj.com/.
18. Load Sharing Facility (LSF). http://www.platform.com.
19. J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Future Generation Computer Systems*, 12(1):67–85, May 1996.
20. Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.