MPISH: A Parallel Shell for MPI Programs

Narayan Desai, Andrew Lusk, Rick Bradshaw, Ewing Lusk
Argonne National Laboratory, Argonne, Illinois 60439

Abstract

While previous work has shown MPI to provide capabilities for system software, actual adoption has not widely occurred. We discuss process management shortcomings in MPI implementations and their impact on MPI usability for system software and managment tasks. We introduce `MPISH`*, a parallel shell designed to address these issues. Keywords: MPI, System Software, System Management, Scalability*

1. Introduction

In this paper, we use "system software" or "system tools" to describe the software used to operate parallel machines, including resource managers, system management tools, and monitoring tools. This category also includes tools used for the diagnosis and repair of system problems. This paper does not discuss per-node software such as compilers or the node kernel. Rather, the tools described in this paper are used to manage and operate parallel machines. While much system software consists of long-running software components, a large set of ad hoc tools are also needed to operate, diagnose, and repair parallel systems. Improved scalability in system software is vital; system scalability is increasing far more quickly than human groups can effectively grow.

System software is currently the least scalable software commonly used on parallel systems. Toolsets consist mainly of serial tools; although some tools do implement a form of suboptimal ad hoc parallelism, parallelism usually is added by calling serial tools from inside parallelized loops. All common management services are fundamentally serial; scalability is provided through the addition of macroscopic parallelism, typically implemented by using backgrounded rsh loops or other looping constructs. This mixed mode of using serial programs in parallel is not well constructed; for example, no good mechanism exists for propagating errors among peer tool invocations in a single parallel task.

MPI is almost universally available on parallel machines, and work has already shown many benefits for MPI use among several classes of system software [6]. This work has, however, also revealed serious barriers to MPI adoption among system software developers and administrators.

To address these barriers, we have developed `MPISH`, a parallel shell capable of running MPI programs. To set the stage, in Section 2, we describe the UNIX interactive execution environment, highlighting features useful in system management tasks. In Section 3, we present shortcomings in current MPI process management systems that prevent MPI programs from being as usable to system administrators as their serial analogues. We then turn to `MPISH`: in Section 4 we describe its implementation and present examples of its use, and in Section 5 we discuss our experiences with `MPISH`, assess its success and limitations, and describe future directions for improvement.

2. Background

System administrators use machines differently from application users; users are expected to use machines only when they are properly functioning, while administrators are expected to fix machines when they break. This single distinction creates a large difference in selection criteria. Systems software and tools must be chosen for their ability to function properly in the face of system failures. We will describe the problems faced by system administrators and techniques used to address them. These techniques motivate features included in `MPISH`.

2.1. System Administration

System administration involves the maintenance of machines from initial construction through normal operations. Many acceptable solutions for initial system configuration exist [7, 5, 1] and hence are not the focus

of this paper. The majority of work done by system administrators, and similarly of system software, involves daily operations of machines. System software handles normal system tasks, such as job execution and system monitoring. System administrators get involved when things don't work properly.

Systems software must be able to aid in system diagnosis and repair in the face of a system fault. While failure is not a desirable outcome, it is sometimes unavoidable. Thus, reliable, quick, and predictable failure is a desirable feature. That is, if a program is to fail, it should do so quickly and reliably. This design goal is quite different from others in the parallel application community; hence, a different way of running parallel programs is needed.

System administrators tend to be tool users, not developers. Hence, the APIs used by system administrators are macroscopic, on the scale of whole programs, as opposed to the microscopic APIs used by applications programmers.

When system administrators do undertake software development, a problem often arises. While such administrators understand what is needed in good system software, they often are unfamiliar with sound programming techniques and lack knowledge about scalable algorithms – skills required to implement sound system software. (This topic is discussed extensively in [6].) Even when tools are well implemented, usability concerns may prevent tool adoption. The scalable UNIX tools[12], a set of MPI programs that implement parallel analogues to common UNIX tools, were soundly implemented and provided useful functions, but they were unwieldy in practice because of MPI execution issues.

2.2. Debugging Methodologies

Most system administration activities center on finding and resolving system faults. Administrators use interactive shells and commands to locate the sources of system faults. These tools are usually serial and help to locate faults in a single node. Tools that *collectively* find and correct problems are far less common. Use of MPI in this area provides the most promise for scalable system debuggers.

Current techniques use interactive logins or, in some cases, tools such as `pdsh`[14] that encapsulate the use of serial shells on multiple nodes. System administrators frequently compose multiple simple tools to provide complicated fuctionality. A common goal of this process is to filter data based on node conditions. The use of serial programs to collect this data has one severe limitation: collective interpretation of the data is left to the user. The introduction of MPI in these tools would allow for parallel debugging tools to automatically locate collective problems.

2.3. UNIX Process Semantics

Shells are the most ubiquitous tool in UNIX environments. Users are familiar with such shells and use them even for task automation. Serially, users are able to start a login session, run programs interactively, and compose these tools to automatically postprocess tool output or initiate other operations.

Shells are able to fulfill this important role through use of the UNIX process management model. Parent processes, or the initiator of a new process, occupy a special role in the execution of the child. Parents are able to set initial conditions for execution, including environment variable and open file handles for `stdin`, `stdout`, and `stderr`. The ability to set up file handles can be used to construct command pipelines. Parents are able to easily and reliably signal their children. Parents are also signaled upon child termination. The parent process is the only process from which exit status can be determined. These features are all needed to build the interactive UNIX environments with which users are familiar. Because of the particular tasks administrators perform, these capabilities are extremely useful on a daily basis.

3. Process Management and MPI

A number of process managers (e.g. YOD[2], PBS[13], LSF[15], POE[16]) are in use for starting parallel programs. `MPICH` is not such a process manager, and in fact needs a process manager to run it. Rather, it provides a rich environment for running a parallel program, incorporating g those features that enhance the implementation of system management programs in MPI. While these process management systems are able to start up MPI programs, they aren't able to provide the same capabilities for composition and automation as their serial counterparts.

When considering the usage of MPI in systems software and tools, issues of process management present themselves as the most serious obstacle to widespread adoption. We will describe the function of MPI process managers, and the features lacked for system administration tasks.

3.1. MPI Process Management

All MPI implementations include different systems for process management. The purpose of these systems is to start processes that can be integrated into a single parallel process. Unfortunately, the MPI specification does not speak in detail about startup mechanisms; `mpiexec` is discussed, but uniform command-line semantics do not lead to uniformity of internal functionality. This bootstrapping function includes remote execution functionality and some manner of coordination mechanism so that instances of the parallel process can locate one another and synchronize properly for initialization.

MPI implementation bootstrapping is a well-understood issue; nearly every implementation uses a different mechanism. PMI (Process Management Interface) has been developed by the MPICH2 [11] team to provide a uniform startup mechanism for all MPICH2-based jobs. The LAM[3] MPI implementation includes an analogous interface that starts LAM-based MPI processes. We focus in this paper on PMI.

3.2. PMI

PMI is an API defined to separate process manager implementation from MPI library implementation while providing certain services that may be needed by an MPI implementation. An example of such a service is the publication of "contact information," so that one MPI process can dynamically connect to another when it first sends a message to the other process. Such information is known by the process manager, since it was the entity that started the other process. Therefore, some interaction with the process manager may be necessary. The point is that any MPI implementation (or other parallel program, for that matter) using PMI can be run under any process manager that implements it. In PMI, contact information is exchanged by a general put/get/fence mechanism, which permits a variety of scalable implementations.

PMI also includes other functions needed by parallel programs and is not peculiar to MPI. One example is the dynamic, remote creation of new processes, as needed by `MPI_Spawn`. The PMI interface and one implementation by the MPD process manager is described in [4].

3.3. Issues

While PMI is suitable for correctly bootstrapping MPI processes at high speed, MPD—the PMI implementation included with MPICH2—does not integrate well into interactive, UNIX-style environments. Many of the familiar functions provided by UNIX shells for serial programs are missing from parallel execution. Also, complexity is an issue. Users need to specify parallel processes in a different way from serial processes. The execution path taken by this process is more complicated than the serial analogue.

Serial UNIX programs have a rich set of functionality to allow the composition of programs and noninteractive automation of complicated tasks. The `mpiexec` execution model isolates the user from the processes started, thus removing the capabilities users depend on during interactive sessions. Incorporating MPI programs into command pipelines and deriving exit status for individual ranks are not possible in a standard-compliant way. Hence, parallel programs are less useful as components of a toolset. And since this is the main mode in which programs are used by system administrators, MPI tools are essentially unusable.

4. MPISH

System administrators live in an imperative, "command line" world. They prefer simple tools[9] that can be composed to perform complicated tasks. Monolithic applications are routinely ignored,: they may perform particular tasks well, but their applicability to a large range of problems is limited. This usage model leads to the need to implement whole tools using MPI, as opposed to publishing libraries. The goal must be to make running parallel programs as simple as running serial programs.

`MPISH` has been designed as a parallel analogue to a serial shell, such as the Bourne shell. In the interactive serial world, shells provide a user with a foothold on the system for running programs. This model has proved quite useful; shells are the workhorse of any serious UNIX user. Shells provide several important functions: an interactive mechanism to run programs, pipes, signaling, and conditional execution. With these functions, users can easily run programs, compose tools to achieve complex goals, and write programs that noninteractively use these tools in complex ways.

In the following subsections, we describe the implementation of `MPISH` and then give concrete examples of scaling management tasks by using `MPISH`.

4.1. Implementation

`MPISH` is implemented as a `C` program written using the `MPICH2` MPI implementation. It implements the PMI interface used by `MPICH2` for application bootstrapping. This enables `MPISH` to start MPI applica-

tions without depending on an external process manager once it is initialized. The result is faster client program execution and more resiliency in the face of transient problems. Also, because the UNIX parent/child relationship is maintained, `MPISH` can detect when client programs exit and can uniformly signal children. Once `MPISH` has been started, it processes commands either interactively or from a script file. Each line is interpreted in two parts: a location specification and a command, for example:

```
global:echo ‘‘Running on  ‘‘ ‘hostname‘
0,2,4:mpisync /source /target
```

The location specification can be either "global" or a set of ranks where the command should run. The command is interpreted as a Bourne shell command. The environment for this execution is similar to that provided by serial shells, `pdsh` [14], or the `C3` [8] tools, with the addition of a PMI instance. This addition allows parallel commands to bootstrap and execute properly. Hence, commands embedded in `MPISH` scripts can be either parallel or serial commands and can be used interchangeably.

Since the PMI implementation included with `MPISH` is written using MPI, it is portable to any MPI implementation. Unfortunately, the MPI specification[10] does not specify job startup mechanisms, so this approach is useful only for starting MPI processes that use PMI as a bootstrapping mechanism. Basically, PMI is a distributed database with synchronization operations, so its implementation in MPI is straightforward. This approach yields a shell that can be linked with any MPI implementation that can natively execute any PMI-based MPI program.

Control structure has been implemented by passing this task to the underlying shell. Commands can be complex shell expressions including conditionals and loops. This approach is the source of one shortcoming in the current implementation. Shell logic is confined to a single command invocation; that is, each command is implemented as a single shell invocation. The approach is good enough for many applications, such as noninteractive job scripts; however, it leaves something to be desired in more complex cases. We avoided including a domain-specific command/scripting language during `MPISH`'s initial implementation, outside of the location directive. We took this approach because of a lack of familiarity with the problem space.

Interactive use is implemented by means of a console, which runs on the user's `tty`. The console connects to the rank 0 of `MPISH`, which distributes stdin to all ranks and collects stdout from all ranks. Thus, it bypasses the nonspecified stdin/out handling in the MPI implemetation.

File staging support is included in `MPISH` to meet specific requirements on a local system without a global filesystem. This support consists of an implicit call to the MPI program `stagein`, which passes through a command line argument. This feature was implemented so that interactive invocations would always have the correct filesystem data for execution. Use of this feature is optional, so `MPISH` can be used on systems with global filesystems.

4.2. Examples of Use

`MPISH` is useful whenever a set of parallel tasks is executed. `MPISH` is started on a set of nodes established as a parallel execution context. All command locations are interpreted in that context, thereby allowing the user to focus on the task at hand and ignore the mechanics of specifying all details about a parallel process. We describe here three common instances of `MPISH` usage: in batch scheduled jobs, in a parallel build script, and in interactive debugging of system problems. (Other examples have been published in [6].)

Each of the examples demonstrates the benefits of making parallel applications easily accessible to system administrators. This approach enables new levels of scalability and conciseness in systems software. Moreover, because `MPISH` is similar to traditional shells, the process of composing a series of MPI and serial applications into a single, interactive result becomes intuitive. This approach allows users to continue using a well-established and familiar paradigm.

4.2.1. Batch Job Scripts The most common use of `MPISH` is during parallel jobs run though a batch scheduler. The user specifies a job script and a set of input and output directories. Once the job is scheduled, `MPISH` is executed on the nodes, and filesystem data is staged in. The user script is then executed on nodes, and once all commands have completed, output data is staged out.

This is certainly a simple example, but it provides several benefits over the serial job script traditionally used. Users can run serial and parallel tasks interchangeably. Administrators can easily integrate parallel tasks into the job prologue and epilogue. Also, from the perspective of a system software implementor, a user job of this form is easier to manage. Serial user jobs consist of a serial script executed on a single node, which initiates the execution of the serial and parallel mechanisms. After startup, the connection between the user job script and the remote user processes is fairly tenuous. The startup mechanism pro-

vided by MPISH preserves the traditional UNIX process group mode. Hence, running processes can be reliably signaled and killed from a single point of control, thereby allowing user jobs to be treated as coherent entities.

Another benefit is the use of the "global" location specification for parallel execution. This allows users to write scripts that run without modification regardless of execution scale. When serial scripts are used, the size of the parallel program is embedded, so scripts include size details about the running job. Users typically keep multiple copies of the same job script for different job sizes; with MPISH this practice is no longer required.

4.2.2. Parallel Build Environment Another good example of MPISH use is a system build script. An initial node software installation can be performed from a canonical installation. This system building process, known as imaging, typically is implemented serially; however, substantial performance improvements can be gained through the use of mpisync[6], a parallel version of rsync. Throughout the imaging process, this step is the only one that benefits from using parallel tools, so the rest of the process still occurs serially; however, the serial tasks are executed simultaneously. This operation can be run on any number of nodes and even performs well on single nodes, since the performance of mpisync scales with the performance of MPI_Bcast. As this operation is well optimized in all MPI implementations, an administrator can be confident in the function and performance of system imaging, regardless of the quantity of nodes requiring imaging.

4.2.3. Interactive System Debugging MPISH can be used for interactive system debugging. Upon initialization, one MPISH rank is started on each node requiring debugging. A local console is connected, allowing the user to dynamically specify parallel diagnosis commands and iteratively focus on subgroups as the problem is traced. This approach allows the introduction of parallel diagnosis techniques. Most system diagnosis tools are serial and are usually cobbled together through ad hoc mechanisms to analyze multinode problems. The use of MPI to automate analysis of multinode problems is an obvious improvement. Several of these parallel debugging programs have been implemented and have proved quite useful. The most advanced of these is a program that monitors the health of a network from the perspective of a set of nodes. That is, the program checks the status of a subset of the network for a given set of nodes and the routes between them.

5. Summary and Plans

This paper introduces MPISH, a parallel shell capable of natively starting MPI processes. Over the past 18 months of regular use, MPISH has proven its utility in numerous situations, some of which have been discussed above. MPISH provides an environment similar to the standard UNIX interactive environment, while allowing the seamless integration of scalable system tools implemented with MPI. This approach enables the use of MPI in even the most low-level cases, where manual techniques traditionally have been used.

While MPISH has enabled a variety of interesting ways to expand the use of MPI in system software, there remain a number of areas where MPISH could be improved. The most severe limitation is the underlying parsing of commands by the Bourne shell. Since each of these commands is run in a separate subshell, persistent changes to the user environment cannot be made. We plan to reimplement the command parser to tokenize the entire command, as opposed to just stripping out the location specification. This approach will also allow the addition of built-in commands, as normal shells have. An example that would be quite useful is a timing routine, which would aid in simplifying benchmarking scripts.

Location specifications could use augmentation as well. The "global" specification is useful in that it provides the same result regardless of the execution scale. This capability proves exceptionally useful when automating tasks. We feel that the abstract aspect of this directive is the beneficial part. To this end, more abstract location specifications could easily be added. Adding fractional specifications (half of nodes) and series of sizes (powers of two) would make certain tasks much easier.

Our current PMI implementation supports only one MPI command per command pipeline. The reason is that MPISH does not parse command structures. Once MPISH implements shell constructs, looping, and conditional execution, it will be able to provide a PMI instance per forked command.

Moreover, unlike traditional UNIX shells, MPISH has no notion of job control. Since job control is a frequently used feature, such a mechanism would be useful. Supporting this will be easy once MPISH handles all command parsing internally.

In spite of these limitations, MPISH provides a unique capability: the ability to run parallel and serial programs through a uniform interface. This interface makes MPI much more accessible to system administrators for task automation and system debugging.

Acknowledgments

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

References

[1] Paul Anderson and Alastair Scobie. Large scale Linux configuration with LCFG. In USENIX [17], pages 363–372.

[2] Ron Brightwell and Lee Ann Fisk. Scalable parallel application launch on cplant. In *Proceedings of SC 2001*, 2001.

[3] 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.

[4] R. Butler, W. Gropp, and E. Lusk. A scalable process-management environment for parallel programs. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 168–175, September 2000.

[5] N. Desai, R. Bradshaw, R. Evard, and A. Lusk. Bcfg : a configuration management tool for heterogeneous environments. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER03)*, pages 500–503. IEEE Computer Society, 2003.

[6] Narayan Desai, Rick Bradshaw, Andrew Lusk, and Ewing Lusk. MPI cluster system software. In Dieter Kranzlmuller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 3241 in Springer Lecture Notes in Computer Science, pages 277–286. Springer, 2004. 11th European PVM/MPI Users' Group Meeting.

[7] Brian Elliot Finley. VA SystemImager. In USENIX [17], pages 181–186.

[8] R. Flannery, A. Geist, B. Luethke, and S. L. Scott. Cluster command & control (c3) tools suite. In *Proceedings of the Third Distributed and Parallel Systems Conference*. Kluwer Academic Publishers, 2000.

[9] Mike Gancarz. *The UNIX Philosophy*. Digital Press, 1994.

[10] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[11] MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.

[12] Emil Ong, Ewing Lusk, and William Gropp. Scalable Unix commands for parallel processors: A high-performance implementation. In Y. Cotronis and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 410–418. Springer-Verlag, September 2001. 8th European PVM/MPI Users' Group Meeting.

[13] PBS home page. http://pbs.mrj.com/.

[14] Pdsh:parallel distributed shell. `http://www.llnl.gov/linux/pdsh/pdsh.html`.

[15] Load Sharing Facility (LSF). http://www.platform.com.

[16] POE home page. `http://publib.boulder.ibm.com/clresctr/windows/public/pebooks.html`.

[17] USENIX, editor. *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, Berkeley, CA, USA, 2000. USENIX.