MPISH2: Unix Integration for MPI Programs

Narayan Desai, Ewing Lusk, Rick Bradshaw

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

Abstract. While MPI is the most common mechanism for expressing parallelism, MPI programs remain poorly integrated in Unix environments. We introduce MPISH2, an MPI process manager analogous to serial Unix shells. It provides better integration capabilities for MPI programs by providing a uniform execution mechanism for parallel and serial programs, exposing return codes and standard I/O stream information.

1 Introduction

The shell is the most familiar interface to Unix systems. In general, it is the first contact that users have with Unix systems. Its ubiquity makes it the dominant mechanism through which command execution occurs.

Unix shells provide a rich environment for task automation, exposing command exit codes, providing control flow constructs, and organizing disparate programs into complex command pipelines. Users are familiar with the decomposition of complex tasks into the invocation of single-function utilities using these mechanisms.

While MPI is not as ubiquitous as Unix shells, it is the dominant mechanism used to express parallelism in scalable applications. Many high-performance implementations of MPI exist; indeed, MPI is so pervasive that a good MPI implementation is frequently cited as one of the requirements for new large-scale computational science machines.

Unfortunately, process management systems that can start MPI programs have not provided or exposed sufficient information for their composition with their serial analogues or even with each other. To address this issue, we have implemented `MPISH2`, a MPI process manager that provides a user interface and composition capabilities nearly identical to the Bourne shell.

2 Related Work

Unix shells have long been studied. Starting with the original shell included with early Unix systems [14], shells have been augmented into relatively full-featured programming languages, including data types [9]. Because of the familiarity of the shell interface to Unix users, many attempts have been made to present a shell-like interface for program execution on parallel systems.

- PDSH [12] is a program that uses `rsh` or `ssh` to execute tasks in parallel on many systems.
- The C3 tools [8] provide a similar execution mechanism that also runs tasks through `rsh`.
- Gridshell [15] provides a shell-like interface that enables access to Grid resources, including the queueing of jobs. It doesn't natively support the execution of parallel process; rather, it is subject to the limitations of the underlying resource management system used to implement this functionality.

These tools do an admirable job of starting processes scalably; however, they do not expose any of the Unix process information needed to embed parallel commands in more complex execution units. Discrete exit statuses are not returned for each process executed. Most important, none of these tools supports MPI process startup.

Historically, MPI startup mechanisms have scaled poorly and performed badly overall [5]. Two systems have addressed these issues over the past several years.

- MPD [4] uses a group of daemons, arranged in a ring topology, to scalably start MPICH2 processes.
- YOD [2] provides similar capabilities in the Cplant software stack.

Both of these process management systems provide highly scalable process startup services needed to start MPI processes, but neither system provides adequate information for use in shell-style programming. MPD provides access to all exit statuses and to standard I/O multiplexed into single streams. YOD provides similar access to standard I/O but fails to provide any access to return codes.

The work we present here has been motivated largely by the gains in system software scalability afforded by the use of MPI in system tools [6,7]. This approach also has proven quite positive in terms of overall performance gains. More surprising, tools implemented by using MPI-based scalable components have proved far easier to troubleshoot and debug than their ad hoc analogues. The need to execute large numbers of small scalable tools brings execution issues clearly into focus.

3 Design

When we were considering how to integrate MPI programs into a Unix environment, our highest priority was to retain standard Unix shell semantics. The overall goal was to support the execution of parallel programs using an interface indistinguishable from that used for serial programs. With such a uniform execution interface, parallel reimplementations of serial utilities could be automatically used by existing scripts.

The Bourne shell [1] was chosen as the language basis for our shell. Two major aspects of the Bourne shell are important: process semantics and control flow constructs. Unix process semantics provide access to information about child processes, including access to return codes, the ability to setup child process environment variables, and the ability to arrange commands into command pipelines that run concurrently. However, this data is available only for child processes that have been directly started. Without the existence of a Unix parent/child relationship, this information is not available and cannot be influenced in any way. Hence, the preservation of this relationship was an important design goal.

The control flow constructs available in the Bourne shell are fairly standard, including `while`, `if`, `for`, and `case`. Since these are the real workhorses of shell scripting, we attempted to keep their semantics as close as possible to the Bourne shell. However, minor enhancements were required in order to support startup of parallel processes.

3.1 MPISH2: A Parallel Shell

The difference between a normal shell and `MPISH2` is that `MPISH2` is a parallel program, consisting of multiple communicating Unix programs. A shell script, given to `MPISH2`, is executed by all of the `MPISH2` processes. The `MPISH2` processes communicate with each other (in a scalable fashion) using MPI. That is, `MPISH2` is itself an MPI program. Therefore, `MPISH2` must be started by the startup mechanism of the proper MPI implementation. We assume in this paper that `mpiexec` invokes this mechanism. Thus, a 100-process instance of `MPISH2` is started by a command line something like the following.

```
mpiexec -n 100 mpish2
```

In a cluster environment, the specification of which nodes `MPISH2` is run on depends on the particular MPI implementation being used. We have used MPICH2 [11], but `MPISH2`—being an MPI program—can be run by using any MPI implementation. Note, however, that because of the nonstandard nature of MPI startup, programs started by `MPISH2` must use `MPICH2`.

`MPISH2` scripts are Bourne-shell scripts (with some extensions described in Section 3.2) that are presented to the standard input of each `MPISH2` process. `MPISH2` must be parallel in order to properly provide all information about child processes. For example, using a traditional MPI process manager to run two parallel programs in a pipeline would look like the following.

```
mpiexec -np 10 prog1 | mpiexec -np 10 prog2
```

This command runs prog1 and sends the standard output of the first `mpiexec` to the second invocation of `mpiexec`. Handling of standard output is not specified by the MPI standard; however, many MPI process managers provide multiplexed standard output from all processes to the standard output of `mpiexec`. Likewise, `mpiexec` typically, though not universally, sends standard input of `mpiexec` to some number of the parallel process instances.

Under `MPISH2`, a similar command is used, together with a process management system for `MPISH2` startup.

```
mpiexec -np 10 mpish2
```

Once `MPISH2` is running, a command pipeline can be executed by using the following script.

```
prog1 | prog2
```

This script is run by every `MPISH2` instance, resulting in 10 instances of both `prog1` and `prog2`, connected rankwise into a pipeline. That is, standard output produced by the rank 0 instance of `prog1` is fed into the standard input of the rank 0 instance of `prog2`, and so forth. Additional utilities are provided, allowing interrank manipulation of I/O streams. These execution semantics provide more flexibility than those afforded by traditional process management systems.

3.2 Enabling Parallelism

Parallel process managers work in much the same way as serial process managers. They are responsible for post-fork/pre-exec process setup and the setup of standard I/O. The main difference between serial and parallel process managers is the need for parallel library bootstrapping. This bootstrapping consists of two main parts: the description of the parallel process topology and the communication setup.

Many mechanisms describe initial process topology at the time of parallel process startup. Typically, the topology specification consists of process count and some set of resources, usually a list of nodes on which the processes should be executed. This corresponds closely to the common arguments to `mpirun`. Alternatively, one can use `mpiexec`, specified by the MPI standard [10], for supplying the same data. Whatever the input format, this information is used for the same purpose: the description of MPI_COMM_WORLD for the new process. Each communicator has a specific size, and each component process has a specified rank in that communicator. This initial topology description is what differentiates one 32-node program from thirty-two 1-node programs.

`MPISH2` describes the initial communicator in terms of the parallel execution context of the client program. The notion of control flow groupings is maintained across the parallel shell. For example, if a parallel program is run on the first line of a script, it will be run on all processors, with an initial communicator identical to MPI_COMM_WORLD of the parent shell. Control flow constructs all affect this execution context for parallel programs. Their behavior can be most easily described in terms of `MPI_Comm_split`:

- *if* performs a two-way split, corresponding to the truth value of the predicate. Programs run in either branch will be grouped into parallel processes with

the other processes executed in the same branch. For example, when *if* is executed in an 8-process context, resulting in a 4-node true, 4-node false split, processes run in the true branch will be grouped into 4-node parallel processes with the other processes executed on the true branch. Similarly, the processes executed on the false branch will be grouped into a 4-process parallel process with the others started on the false branch.
- *case* performs an *N*-way split, operating similarly to *if.*
- *while* creates an execution context corresponding to all ranks for which the condition evaluates as true. All programs run in each iteration will be grouped according to this initial evaluation. The condition will be evaluated at the start of each iteration, continuing until all ranks evaluate false.
- *for* has no effect on parallel execution context because it is not conditional.

Note that each control flow statement now implicitly includes a synchronization barrier at its conclusion. This approach has the distinct advantage of retaining the character of serial Bourne shell control flow operations. In fact, for one-node executions of `MPISH2`, the behavior is precisely that of a serial Bourne shell.

The second important aspect of parallel process startup is communication bootstrapping. For disparate processes to begin acting as a single parallel entity, communication must be established. This is accomplished in different ways with different parallel libraries. MPICH2 uses an interface called PMI, or Process Manager Interface, to provide this information to client programs. PMI takes the form of a distributed database, providing standard *put, get* and *fence* operations. The client program is provided with connection information for its PMI instance and can use that data to connect to other processes.

4 Implementation

The implementation of `MPISH2` is based on a modified version of the `Minix` [13] shell, included with `Busybox` [3]. Three main modifications have been made.

The first was driven by the fact that `MPISH2` is a parallel, not serial, process. A parallel execution context—that is, a grouping of discrete processes in order to form a parallel process—must be tracked on each instance of `MPISH2`. Initially, it corresponds to MPI_COMM_WORLD; however, as the script executes, the execution context is modified by control flow constructs, as described in Section 3.2. Changes in the parallel execution context are tracked by using an MPI communicator. This communicator is passed to any new PMI instances created, thereby maintaining cohesion between parallel processes executed in the same context.

The second modification was the creation of a PMI implementation to service requests from client processes. In order to support parallel library bootstrapping, a discrete PMI implementation is provided for each program started by the shell. Setup of this instance consists of initial data structure creation and socket setup. During client execution, the client program will connect and submit commands.

Many of the commands, like *put*, which stores a value in a distributed database, will be serviced locally; however, some, like *get*, or *fence* may require communication with other parts of the same PMI instance. All communication operations are implemented by using MPI collective and asynchronous operations. *Fence* is implemented by using `MPI_Barrier`. The implementation of *get* is more complicated. When a PMI instance receives a *get* request, it checks whether the value is already stored locally. If it is, the request is immediately serviced. If not, a message is sent to the PMI instance with the next higher rank. Each process also receives queries for unknown values asynchronously. If the local process has the value, it responds to the querier; otherwise, it forwards the request to the next rank in the PMI instance. Disparate PMI instances in the same `MPISH2` instance are differentiated based on a private communicator. This communicator is `MPI_Comm_dup`'ed at PMI instance initiation time, so each PMI instance has a unique one.

The third, and perhaps most complex, modification was to the control flow construct to provide topology descriptions for client processes. In a typical serial shell, control flow constructs use only return codes and have no side effects. In `MPISH2`, however, control flow constructs also affect the parallel execution context by calling `MPI_Comm_Split` after predicate execution. For example, in serial shells, the shell executes the *if* predicate and either the true or false branch depending on a zero or nonzero return code, respectively. `MPISH2` executes the same operations, but with the addition of a call to `MPI_Comm_Split` using zero/nonzero exit status. Other control flow constructs were similarly modified.

None of these modifications proved complicated, and the overall semantics of the `MPISH2` remains very close to the semantics of the Bourne shell. At the same time, these modifications provide a wealth of new capabilities to Unix users.

4.1 Utilties

A parallel execution environment isn't really complete without a set of parallel programs useful for writing basic programs. These programs are analogous to `test` or `wc` for serial shells. We have implemented a variety of small utilities, suffixed with the `.mpi` extension, to address this issue. The following is a list of basic parallel commands, with a short description of each.

- `rank.mpi` displays the process's rank in the current execution context.
- `size.mpi` displays the size of the current execution context.
- `once.mpi` exits with a return code of 0 once per physical node present.
- `zoom.mpi` provides access to scalable numeric reductions for the provided argument.
- `pflatten.mpi` sends all stdout streams to process 0.
- `ptee.mpi` forwards stdin from process 0 to all processes. It functions like a parallel version of tee.
- `pcoalesce.mpi` coalesces stdout from all nodes, producing rank delimited lines on processor 0.

– `bcast.mpi` broadcasts the data from one process, specified as an argument, to all other processes. This data is produced on stdout.
– `stagein.mpi` downloads a file from a http server and broadcasts to all nodes, eventually writing it to disk on each.
– `stageout.mpi` uploads files, tagged with rank, to the fileserver from all clients.
– `rsync.mpi` synchronizes files from process 0 to all other processes. This program can handle all regular and special files.
– `time.mpi` times the execution of a parallel program, producing a single wall time result.

Each of these programs is a simple MPI program. Nothing special is required to write a utility, as MPISH2 can run arbitrary MPI programs.

5 Use

`MPISH2` is useful across the same broad range of problems as are standard shell scripts, with the added ability to run concurrent, parallel programs. It can easily be used for tasks ranging from the most trivial to those that can strongly benefit from access to parallelism and scalable tools.

5.1 Examples

The following examples illustrate various language features.

Basic Parallel File List The first example copies the file from one source to all machines. The `find` command produces a list of files on standard out and `ptee.mpi` replicates this I/O stream to all other ranks. Each rank writes runs `ls` on each of these files.

```
find /path -type f | ptee.mpi | xargs ls -l
```

5.2 Basic Collective Diagnostics

The second example runs the command `fix_network` on the node with the highest network interface error count.

```
max.mpi get_net_err_cnt
if [ ’’${?}’’ -eq 0 ] ; then
   fix_network
fi
```

Job Script The third example is a job script for a queueing system. This script runs the prologue, epilogue and file staging commands once per physical node (hostname). Of these commands, the prologue and epilogue are serial, while the file staging commands are parallel. Once setup has completed, the user job is run (under the user's UID), and cleanup is performed. Not only can serial and parallel programs be interchanged, but standard shell scripting mechanisms (like the use of `su`) can also be used with parallel programs.

```
#!/usr/bin/env mpish2
user="${1}"
userscript="${2}"
indir="${3}"
outdir="${4}"

once.mpi
once=''${?}''
if [  ``${once}'' −eq 0 ] ; then
    # run the prologue once per node
    /usr/sbin/prologue
    if [ ! −z "${indir}"] ; then
        su "${user}" stagein.mpi "${indir}"
    fi
fi

su "${user}" mpish2 "${userscript}"


if [ ``${once}'' −eq 0 ] ; then
    if [ ! −z "${outdir}" ] ; then
        su "${user}" stageout.mpi "${outdir}"
    fi
    /usr/sbin/epilogue
fi
```

Several active execution contexts are used in this program. Two instances of a context containing each physical node are created by the script. The first is used for job setup (e.g., prologue and file staging), and the second is used for job cleanup. The user's job script is executed in the global execution context.

Benchmarking Scripts The fourth example provides a basic illustration of concurrency. Benchmarking scripts are often implemented as a *for* loop that sequentially executes program runs with different sizes, for example, a script such as the following.

```
#!/bin/sh
for i in 2 4 8 16 32; do
  time mpirun −np $i program
```

```
done
```

Such a script does a reasonable job of running benchmarks; however, numerous processor resources are wasted in the first few iterations of the loop if the full number of nodes is reserved for the full duration of the execution.

This process can be run far more efficiently if test cases are executed concurrently. First, the application is run on all nodes. Second, the nodes are grouped into partitions, each with a different power of two size, up to half the total number of nodes. Each of these partitions runs a different size test case concurrently. The following example is a concurrent benchmarking script. It is assumed that the script is run on largest size being benchmarked, in this case 32 nodes.

```
#!/usr/bin/env mpish2
rank=`rank.mpi`

slot="0"
basenum="2"
count="1"

time.mpi -t "size=32" progname

while [ "$slot" -eq "0" ] ; do
    remainder=`expr "$rank" - "$basenum"`
    if [ "$remainder" -lt "$basenum" ] ; then
        slot="$count"
    else
        basenum=`expr "$basenum" "*" "2"`
        count=`expr $count + 1`
    fi
done

case $slot
    1)
     time.mpi -t "size=2" progname
     ;;
    2)
     time.mpi -t "size=4" progname
     ;;
    3)
     time.mpi -t "size=8" progname
     ;;
    4)
     time.mpi -t "size=16" progname
     ;;
esac
```

6 Conclusions and Further Work

We have presented MPISH2, a parallel process manager for MPI programs that provides an interface almost indistinguishable from the standard Unix Bourne shell. It enables the use of MPI in Unix environments in a seamless manner not previously possible. The addition of scalable utilities and simple, Bourne shell style control to Unix environments enables a variety of system and user tasks to be implemented in a scalable and elegant fashion.

The current design and implementation have two main limitations. The first is that all control flow constructs now impact parallel execution context, so serial conditional execution must be separated from parallel conditional execution. The second limitation is that control flow constructs have implicit barriers around them. This can reduce the amount of concurrency available to users. Both of these issues bear further examination.

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. S. R. Bourne. An introduction to the unix shell. *Bell System Technical Journal*, 57(2):2797–2822, Jul-Aug 1978.
2. Ron Brightwell and Lee Ann Fisk. Scalable parallel application launch on cplant. In *Proceedings of SC 2001*, 2001.
3. Busybox home page. `http://www.busybox.net`.
4. R. Butler, N. Desai, A. Lusk, and E. Lusk. The process management component of a scalable system software environment. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER03)*, pages 190–198. IEEE Computer Society, 2003.
5. 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.
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. Narayan Desai, Andrew Lusk, Rick Bradshaw, and Ewing Lusk. MPISH: A parallel shell for MPI programs. In *Proceedings of the 1st Workshop on System Management Tools for Large-Scale Parallel Systems (IPDPS '05)*, Denver, Colorado, USA, april 2005.

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. David G. Korn, Charles J. Northrup, and Jeffery Korn. The new Korn shell. *The Linux Journal*, 27, July 1996.
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. Pdsh:parallel distributed shell. `http://www.llnl.gov/linux/pdsh/pdsh.html`.
13. Andrew Tannenbaum. *Operating Systems, Design and Implementation*. Prentice Hall, 1987.
14. K. Thompson. The unix command language. *Structured Programming*, pages 375–384, 1975.
15. E. Walker, T. Minyard, and J. Boisseau. Gridshell: A login shell for orchestrating and coordinating applications in a grid enabled environment. In *Proceedings of the International Conference on Computing, Communications and Control Technologies*, pages 182–187, 2004.