# Scalable Unix Commands for Parallel Processors: A High-Performance Implementation*

Emil Ong, Ewing Lusk, and William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439 USA

**Abstract.** We describe a family of MPI applications we call the Parallel Unix Commands. These commands are natural parallel versions of common Unix user commands such as `ls`, `ps`, and `find`, together with a few similar commands particular to the parallel environment. We describe the design and implementation of these programs and present some performance results on a 256-node Linux cluster. The Parallel Unix Commands are open source and freely available.

## 1 Introduction

The oldest Unix commands (`ls`, `ps`, `find`, `grep`, etc.) are built into the fingers of experienced Unix users. Their usefulness has endured in the age of the GUI not only because of their simple, straightforward design but also because of the way they work together. Nearly all of them do I/O through `stdin` and `stdout`, which can be redirected from/to files or through pipes to other commands. Input and output are lines of text, facilitating interaction among the commands in a way that would be impossible if these commands were GUI based.

In this paper we describe an extension of this set of tools into the parallel environment. Many parallel environments, such as Beowulf clusters and networks of workstations, consist of a collection of individual machines, with at least partially distinct file systems, on which these commands are supported. A user may, however, want to consider the collection of machines as a single parallel computer, and yet still use these commands. Unfortunately, many common tasks, such as listing files in a directory or processes running on each machine, can take unacceptably long times in the parallel environment if performed sequentially, and can produce an inconveniently large amount of output.

A preliminary version of the specification of our Parallel Unix Commands appeared in [4]. New in this paper are a refinement of the specification based on experience, a high-performance implementation based on MPI for improved scalability, and measurements of performance on a 256-node Unix cluster.

* 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.

The tools described here might be useful in the construction of a cluster-management system, but this collection of user commands does not itself purport to *be* a cluster-management system, which needs more specialized commands and a more extensive degree of fault tolerance. Although nothing prevents these commands from being run by `root` or being integrated into cluster-management scripts, their primary anticipated use is the same as that of the classic Unix commands: interactive use by ordinary users to carry out their ordinary tasks.

## 2 Design

In this section we describe the general principles behind this design and then the specification of the tools in detail.

### 2.1 Goals

The goals for this set of tools are threefold:

- They should be familiar to Unix users. They should have easy-to-remember names (we chose `pt<unix-command-name>`) and take the same arguments as their traditional counterparts to the extent consistent with the other goals.
- They should interact well with other Unix tools by producing output that can be piped to other commands for further processing, facilitating the construction of specialized commands on the command line in the classic Unix tradition.
- They should run at interactive speeds, as do traditional Unix commands. Parallel process managers now exist that can start MPI programs quickly, offering the same experience of immediate interaction with the parallel machine, while providing information from numerous individual machines.

### 2.2 Specifying Hosts

All the commands use the same approach to specifying the collection of hosts on which the given command is to run. A host list can be given either explicitly, as in the blank-separated list `'donner dasher blitzen'`, or implicitly in the form of a pattern like `ccn%d@1-32,42,65-96`, which represents the list `ccn1,...,ccn32,ccn42,ccn65,...,ccn96`.

All of the commands described below have a hosts argument as an (optional) first argument. If the environment variable `PT_MACHINE_FILE` is set, then the list of hosts is read from the file named by the value of that variable. Otherwise the first argument of a command is one of the following:

`-all` all of the hosts on which the user is allowed to run,
`-m` the following argument is the name of a file containing the host names,
`-M` the following argument is an explicit or pattern-based list of machines.

Thus

```
ptls -M "ccn%d-myr@129-256" -t /tmp/lusk
```

runs a parallel version of `ls -t` (see below) on the directory `/tmp/lusk` on nodes with names `ccn129-myr,...,ccn256-myr`.

### 2.3 The Commands

The Parallel Unix Commands are shown in Table 1. They are of three types: straightforward parallel versions of traditional commands with little or no output; parallel versions of traditional commands with specially formatted output; and new commands in the spirit of the traditional commands but particularly inspired by the parallel environment.

| Command | Description |
|---|---|
| `ptchgrp` | Parallel `chgrp` |
| `ptchmod` | Parallel `chmod` |
| `ptchown` | Parallel `chown` |
| `ptcp` | Parallel `cp` |
| `ptkillall` | Parallel `killall` (Linux semantics) |
| `ptln` | Parallel `ln` |
| `ptmv` | Parallel `mv` |
| `ptmkdir` | Parallel `mkdir` |
| `ptrm` | Parallel `rm` |
| `ptrmdir` | Parallel `rmdir` |
| `pttest[ao]` | Parallel `test` |

| Command | Description |
|---|---|
| `ptcat` | Parallel `cat` |
| `ptfind` | Parallel `find` |
| `ptls` | Parallel `ls` |
| `ptfps` | Parallel process space find |
| `ptdistrib` | Distribute files to parallel jobs |
| `ptexec` | Execute jobs in parallel |
| `ptpred` | Parallel predicate |

**Table 1.** Parallel UNIX Commands

**Parallel Versions of Traditional Commands** The first part of Table 1 lists the commands that are simply common Unix commands that are to be run on each host. The semantics for many of these is very natural – the corresponding uniprocessor version of any command is run on every node specified. For example, the command

```
ptrm -M "node%d@1-5" -rf old_files/
```

is equivalent to running

```
rm -rf old_files/
```

on node1, node2, node3, node4, and node5. The command line arguments to most of the commands have the same meaning as their uniprocessor counterparts.

The exceptions `ptcp` and `ptmv` deserve special mention; the semantics of parallel copy and move are not necessarily obvious. The commands presented here perform one-to-many copies by using MPI and compression; `ptmv` deletes the local files that were copied if the copy was successful. The command line arguments for `ptcp` and `ptmv` are identical to their uniprocessor counterparts with the exception of an option flag, `-o`. This flag allows the user to specify whether compression is used in the transfer of data. In the future the flags may be expanded to allow for other customizations. Handling of directories as either source or destination is handled as in the normal version of `cp` or `mv`.

Parallel `test` also deserves explanation. There are two versions of parallel `test`; both run `test` on all specified nodes, but `pttesta` logically ANDs the results of the tests, while `pttesto` logically ORs the results of the tests. By default, `pttest` is an alias for `pttesto`. This link allows the natural semantics of `pttest` to detect failure on any node.

**Parallel Versions of Common UNIX Commands with Formatted Output** The second set of commands in Table 1 may produce a significant amount of output. In order to facilitate handling of this output, if the first argument to `ptfind`, `ptls`, or `ptcat` is `-h` (for "headers"), then the output from each host will be preceded by a line identifying the host. This is useful for piping into other commands such as `ptdisp` (see below). In the example

```
$ ptls -M "node%d@1-3" -h
[node1.domain.tld]
myfile1
[node2.domain.tld]
[node3.domain.tld]
myfile1
myfile2
```

the user has file `myfile1` on node1, no files in the current directory on node2, and the files `myfile1` and `myfile2` on node3. All other command line arguments to these commands have the same meaning as their uniprocessor counterparts.

To facilitate processing later in a pipeline by filters such as `grep`, we provide a filter that *spreads* the hostname across the lines of output, that is,

```
$ ptls -M "node%d@1-3" -h | ptspread
node1.domain.tld:  myfile1
node3.domain.tld:  myfile1
node3.domain.tld:  myfile2
```

**New parallel commands** The third part of Table 1 lists commands that are in the spirit of the other commands but have no non-parallel counterpart.

Many of the uses of `ps` are similar to the uses of `ls`, such as determining the age of a process (respectively, a file) or owner of a process (respectively, a file). Since a Unix file system typically contains a large number of files, the Unix command `find`, with its famously awkward syntax, provides a way to search the file system for files with certain combinations of properties. On a single system, there are typically not so many processes running that they cannot be perused with `ps` piped to `grep`, but on a parallel system with even a moderate number of hosts, a `ptps` could produce thousands of lines of output. Therefore, we have proposed and implemented a counterpart to `find`, called `ptfps`, that searches the process space instead of the file space. In the Unix tradition we retain the syntax of `find`. Thus

```
ptfps -all -user lusk
```

will list all the processes belonging to user `lusk` on all the machines in a format similar to the output of `ps`, and

```
ptfps -all -user gropp -time 3600 -cmd ^mpd
```

will list all processes owned by `gropp`, executing a command beginning with `mpd`, that have been running for more than an hour. Many more filtering specifications and output formats are available; see the (long) `man` page for `ptfps` for details.

The command `ptdistrib` is effectively a scheduler for running a command on a set of files over specified nodes. For example, to compile all of the C files in the current directory over all nodes currently available, then fetch back all the resulting files, the user might use the following command:

```
ptdistrib -all -f 'cc -c {}' *.c
```

Here, the {} is replaced by the names of the files given, one by one. See the `man` page for more information.

The command `ptexec` simply executes a command on all nodes. To determine, for example, which hosts were available for running jobs, the user might run the following command:

```
ptexec -all hostname
```

No special formatting of output or return code checking is done.

The command `ptpred` runs a `test` on each specified node and outputs a 0 or 1 based on the result of the test. For example, to test for the existence of the file `myfile` on nodes node1, node2, and node3, the user might have the following session:

```
$ ptpred -M "node1 node2 node3" '-f myfile'
node1.domain.tld: 1
node2.domain.tld: 0
node3.domain.tld: 1
```

In this case, node1 and node3 have the file, but node2 does not. Note that `ptpred` prints the logical result of `test`, not the verbatim return value.

The output of `ptpred` can be customized:

```
$ ptpred -M "node1 node2 node3" '-f myfile' \
             'color black green' 'color black red'
node1.domain.tld: color black green
node2.domain.tld: color black red
node3.domain.tld: color black green
```

This particular customization is useful as input to `ptdisp`, which is a general display tool for displaying information about large groups of machines. As an example, Figure 1 shows some screenshots produced by `ptdisp`.

The command `ptdisp` accepts special input from standard input of the form

```
<hostname>: <command> [arguments]
```

where `command` is one of `color`, `percentage`, `text`, or a number. The output corresponding to each host is assigned to one member of an array of button boxes.

As an example, one might produce the screenshot on the left in Figure 1 with the following command:

```
ptpred -all '-f myfile' 'color black white' \
                         'color white black' \
| ptdisp -c -t "Where myfile exists"
```

to find on which nodes a particular file is present. The command `ptdisp` can

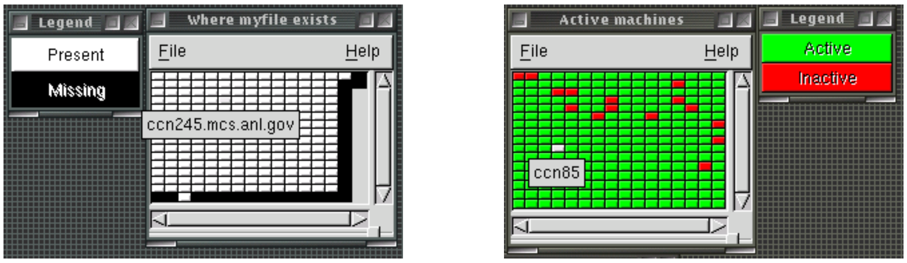


**Fig. 1.** Screenshots from `ptdisp`

confer scalability on the output of other commands not part of this tool set by serving as the last step in any pipeline that prepares lines of input in the form it accepts. Since it reads perpetually, it can even serve as a crude graphical system monitor, showing active machines, as on the right side of Figure 1. The command to produce this display is given in Section 3. The number of button boxes in the display adapts to the input. When the cursor is placed over a box, the node name automatically appears, and clicking on a button box automatically starts an `xterm` with an `ssh` to that host if possible, for remote examination.

## 3 Examples

Here we demonstrate the flexibility of the command set by presenting a few examples of their use.

– To look for nonstandard configuration files:

```
ptcp -all mpd.cfg /tmp/stdconfig; \
ptexec -all -h diff /etc/mpd.cfg /tmp/stdconfig \
| ptspread
```

This shows differences between a standard file and the version on each node.
– To look at the load average on the parallel machine:

```
ptexec -all 'echo -n `hostname` ; uptime' | awk '{ print $1 \
": percentage " $(NF-1)*25 }' | sed -e 's/,//g' | ptdisp
```

The `percentage` command to `ptdisp` shows color-coded load averages in a compact form.
– To continuously monitor the state of the machine (nodes up or down)

```
(echo "$LEGEND$: Active black green Inactive black red"; \
while true; do (enumnodes -M 'ccn%d@1-256' \
| awk '{print $1 ": 0"}') ; sh ptping.sh 'ccn%d@1-256'; \
sleep 5; done) | ptdisp -t "Active machines" -c
```

We assume here that `ptping` pings all the nodes. This is admittedly ugly, but it illustrates the power of the Unix command line and the interoperability of Unix commands. The output of this command is what appears on the right side of Figure 1.
– To kill a runaway job

```
ptfps -all -user ong -time 10000 -kill SIGTERM
```

## 4 Implementation

The availability of parallel process managers, such as MPD [2], that provide pre-emption of existing long-running jobs and fast startup of MPI jobs, has made it possible to write these commands as MPI application programs. Each command parses its hostlist arguments and then starts an MPI program (with `mpirun` or `mpiexec`) on the appropriate set of hosts. It is assumed that the process manager and MPI implementation can manage `stdout` from the individual processes in the same way that MPD does, by routing them to the `stdout` of the `mpirun` process. The graphical output of `ptdisp` is provided by GTK+ (See `http://www.gtk.org`).

Using MPI lets us take advantage of the MPI collective operations for scalability in delivering input arguments and/or data and collecting results. Some of the specific uses of MPI collective operations are as follows.

- `MPI_Bcast` uses `ptcp` to move data to the target nodes.
- `MPI_Reduce`, with `MPI_MIN` as the reduction operation, is used in many commands for error checking.
- `MPI_Reduce`, with `MPI_LOR` or `MPI_LAND` as the reduction operation, is used in `pttest`.
- `MPI_Gather` is used in `ptdistrib` to collect data enabling dynamic reconfiguration of the list of nodes work is distributed to.
- Dynamically-created MPI communicators other than `MPI_COMM_WORLD` are used when the task is different on different nodes. An example of this situation occurs when the target specified in the `ptcp` command turns out to be a file on some nodes and a directory on others.

The implementation of ptcp is roughly that described in [5]. Parallelism is achieved at three levels: writing the file to the local file systems on each host is done in parallel; a scalable implementation of MPI_Bcast provides parallelism in the sending of data; and the files are sent in blocks, providing pipeline parallelism. We also use compression to reduce the amount of data that must be transferred over the network. Directory hierarchies are tarred as they are being sent.

A user may have different user ids on different machines. Whether these scalable Unix commands allow for this situation depends on the MPI implementation with which they are linked. In the case of MPICH [3], for example, it is possible for a user to run a single MPI job on a set of machines where the user has different user ids.

## 5 Performance

To justify the claims of scalability, we have carried out a small set of experiments on Argonne's 256-node Chiba City cluster [1]. Execution times for simple commands are dominated by parallel process startup time. Commands that require substantial data movement are dominated by the bandwidth of the communication links among the hosts and the algorithms used to move data. Timings for a trivial parallel task and one involving data movement are shown in Table 2. Our copy test copies a 10MB file that is randomly generated and does not compress well. With text data the effective bandwidth would be even higher. In Figure 2

| **Number of Machines** | 1 | 11 | 50 | 100 | 150 | 241 |
|---|---|---|---|---|---|---|
| Time in seconds of a parallel copy of 10MB over Fast Ethernet | 5.6 | 8.1 | 10.5 | 12.2 | 13.8 | 14.3 |
| Time in seconds of a parallel execution of hostname | 0.8 | 0.9 | 1.2 | 1.5 | 1.8 | 1.9 |

**Table 2.** Performance of some commands

we compare ptpc with two other mechanisms for copying a file to the local file systems on other nodes. The simplest way to do this is to call rcp or scp in a loop. Figure 2 shows how quickly this method becomes inferior to more scalable approaches. The "chi_file" curve is for a sophisticated system specifically developed for the Chiba City cluster [1]. This system, written in Perl, takes advantage of the specific topology of the Chiba City network and the way certain file systems are cross-mounted. The general, portable, MPI-based approach used by ptcp performs better.

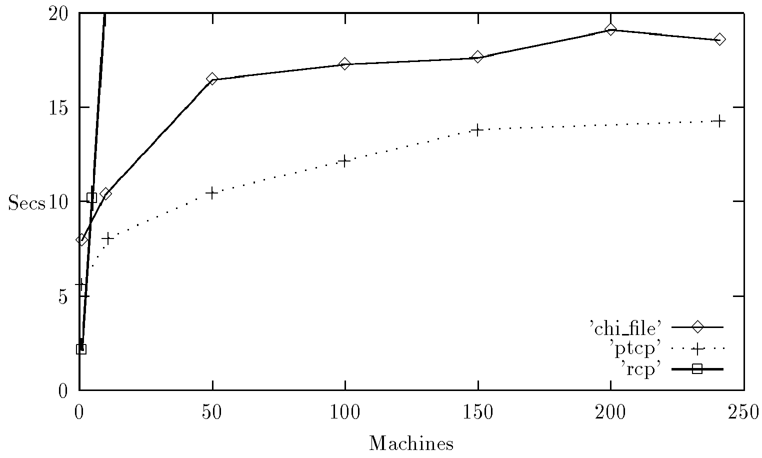


**Fig. 2.** Comparative Performance of ptcp

## 6 Conclusion

We have presented a design for an extension of the classical Unix tools to the parallel domain, together with a scalable implementation using MPI. The tools are available at `http://www.mcs.anl.gov/mpi`. The distribution contains all the necessary programs, complete source code, and `man` pages for all commands with much more detail than has been possible to present here. An MPI implementation is required; while any implementation should suffice, these commands have been most extensively tested with MPICH [3] and the MPD process manager [2]. The tools are portable and can be installed on parallel machines running Linux, FreeBSD, Solaris, IRIX, or AIX.

## References

1. Chiba City home page. http://www.mcs.anl.gov/chiba.
2. 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.
3. William Gropp and Ewing Lusk. MPICH. World Wide Web. `ftp://info.mcs.anl.gov/pub/mpi`.
4. 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.
5. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.