Scalable Unix Tools on Parallel Processors

William Gropp and Ewing Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, 60439

Abstract

The introduction of parallel processors that run a separate copy of Unix on each process has introduced new problems in managing the user's environment. This paper discusses some generalizations of common Unix commands for managing files (e.g., `ls`*) and processes (e.g.,* `ps`*) that are convenient and scalable. These basic tools, just like their Unix counterparts, are text based. We also discuss a way to use these with a graphical user interface (GUI). Some notes on the implementation are provided. Prototypes of these commands are publically available.*

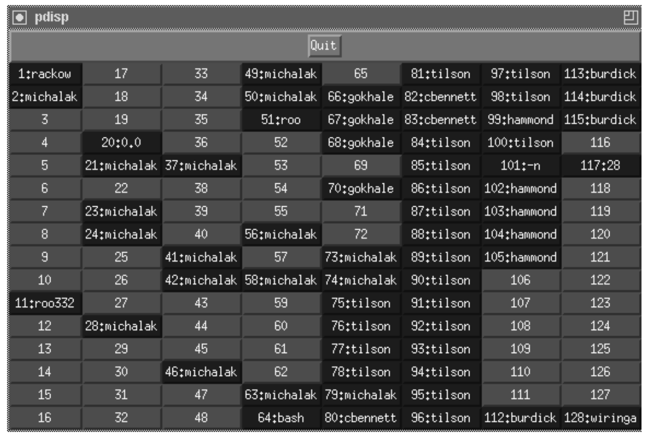

Figure 1: `pps -all aux | egrep -v \ "root|USER|$LOGNAME" | pdisp`.

1 Introduction

Many massively parallel processors (MPPs) are providing a full Unix environment on each processor. This has many advantages, including providing a standard environment that users are familiar with. The disadvantage is that many common tasks, such as listing processes and files on each processor, can now take a significant amount of time and generate too much output to be quickly grasped.

This paper discusses the design of versions of some commonly used Unix tools for this kind of parallel environment as well as some issues relevant to their implementation. Prototype versions of many of these programs have been written as shell scripts and are in use at the High Performance Computing Research Facility at the Argonne National Laboratory. These prototypes are available by anonymous ftp from `info.mcs.anl.gov` in file '`pub/ibm_sp1/ptools.tar.Z`'.

In designing these programs, we set several goals that we believe are crucial to their success. The tools should be:

- Familiar to Unix users. They should have easy-to-remember names (we chose to use `p<unix-command-name>`) and take the same arguments.
- Scalable. They should be fast enough to use with the same regularity that users use `ls` and `ps`, regardless of the number of processors.
- Not generate too much output. It should be possible to restrict the amount of output to a single screenful.

For example, to list the processes belonging to `joe`, we propose

```
pps -all aux | grep joe
```

which is almost identical to the uniprocessor version `ps aux | grep joe`.

The last requirement on the amount of output is difficult to make consistent with the first requirement. That is, if the natural extension of the Unix command to many processors would produce several hundred lines of output, we have no choice but to generate that data. However, we do provide two ways to help achieve this third goal. One way is to generate the output in a form that makes it easy for the user to provide his own filters. Another is to provide some additional programs that provide options that can help the user reduce the amount of output. An example of this is in looking for a file. On uniprocessor Unix systems, the command `ls` is often used to check if a file is present: the user types `ls filename` (or even just `ls`) and looks at the output to see if the file is indeed present. This is (usually) fine on a uniprocessor system, but on a parallel processor with individual file systems, this could generate hundreds of lines of output. Worse, if the file is present on most but not all processors, it is easy to miss that fact in the massive outpouring of data that executing `ls` on each processor could produce.

The solution to this problem lies in looking at other ways that Unix provides to answer the same question. For example, on a uniprocessor system the user could have executed

```
test -s filename
if ($status == 0) echo "no such file"
```

We provide a capability like this with `ppred`, where we have simplified the interface by combining the test with the action.

Managing processes has the same problem; executing `ps` on even a few processors can produce too much data to grasp easily. We introduce a command `pfps` that provides services similar to `find` applied to the space of processes instead of files.

An alternative way to manage large volumes of data is to use graphical rather than text-based display. We describe a program `pdisp` that can translate the output from our other tools into a graphical display.

In order to simplify the processing of any output from any of these tools by other Unix tools (including the graphical display tools we will discuss in Section 2.13), all output lines are prepended with the nodename of the processor.

It is particularly important that output be "line-atomic;" that is, output send to `stdout` from one processor should not appear in a line generated by another processor.

The general principle is important: disconnecting the functionality from the GUI. The use of ASCII text as an interface between tools is one of the most fundemental design features of Unix. It should not be abandoned because of the advent of GUI's; it remains relevant as the key to leveraging the power of software tools.

2 The tools

The tools that we have implemented fall into three broad classes: programs for manipulating the file system, programs for manipulating the process space, and programs for running arbitrary commands on all processors.

Command	Action
`pcp`	Parallel copy (for systems with local disks on each node).
`pcat`	Parallel concatenation of files
`pls`	Parallel directory list (`ls`).
`prm`	Parallel remove
`pmv`	Parallel move
`pfind`	Parallel `find`
`pps`	Parallel `ps`
`pfps`	Parallel process find
`pkill`	Parallel process kill
`pexec`	Run a command on all selected processors
`ptest`	Run `test` on all selected processors, anding the results and returning a single status value.
`ppred`	Run a command when a condition is satisfied
`pdistrib`	Run a command on a collection of files
`pdisp`	Display the output of a command graphically

The programs that generate output (such as `pls`) use the same format as their Unix counterparts except that the name of the processor that generated the output is prepended to each output line. This allows the output to be sorted by processor. An alternate approach is to separate the output by processor name; we did not do this because it makes it harder to use the output as input to other programs.

All of the commands take as their first argument a specification of the processors to run on.

2.1 Specifying Nodes

Nodes may be specified in several ways. The simplist specification is a list of node names:

```
node3,workstation2,big-server
```

This method is adequate for small numbers of nodes. For larger numbers of nodes, we introduce the *domain*, which is a collection of nodes. For example, a domain may include every node in an MPP. A domain may either be specified by a name defined when the tools are installed on a system (such as "paragon" or "sp1") or a name of a file (preceeded by @) that contains a list of nodes. For example, if the file 'mynodes' contains

```
node3
workstation2
big-server
```

then the specification @mynodes specifies the same nodes as the first example above.

Within a domain, it is often desirable to select a subset of nodes. This is done by numbering the nodes consequetively within a domain, starting from one. The numbers are specified as any combination of individual node numbers and ranges of consequtive nodes, separated by commas. For example, 1,4-8,17 specifies nodes 1, 4, 5, 6, 7, 8, and 17 in the given domain. A set of nodes within a domain is specified by giving the domain, followed by a colon, followed by the list of nodes. For examples, using the domain defined above, the nodes node3 and big-server could be specified with

```
@mynodes:1,3
```

There should always be a default domain. For example, the domain that the processor from which a command was run belongs to is often a good choice of default domain. For MPPs with front-end processors, the default domain on the front-end processors should be the MPP.

Multiple domains may be specified; for example,

```
1,3-5,17,@mynodes:2-3
```

specifies nodes 1, 3–5, and 17 in the default domain and nodes 2–3 in the domain mynodes. We chose numbers for input because they are concise. For output, the node name is probably better, though for some uses, just the number in the domain would be easier (for example, in placing output in a GUI display).

We provide the routine phostname as a parallel version of hostname; this provides a simple way to convert a nodelist to the names of the nodes. Note that phostname can be implemented by using pexec with program hostname

To summarize, the nodelist may be specified by the following YACC-like grammar, where [a] denotes an optional a and single quotes surround terminal symbols, and <...> surround descriptions of simple tokens like integers and filenames.

```
nodelist -> '-all'
nodelist -> domain ':' nodelist
nodelist -> range [ , nodelist ]
nodelist -> nodenum [ , nodelist ]
nodelist -> nodename [ , nodelist ]
domain   -> <predefinedname>
domain   -> '@' <valid filename>
range    -> nodenum '-' nodenum
nodenum  -> <integer>
nodenum  -> 'last'
nodename -> <any valid nodename>
```

Nodes are numbered from one. The special value -all denotes all nodes. The special nodenum last denotes the number of nodes in the current domain. These numbers may refer to nodes in an MPP or to members of a workstation network.

2.2 Parallel ps

The parallel ps has the same format as ps with the exeception of the specification of the processors to run on. The output is similar except that each output has the processor number prepended. The output is not sorted by processor number.

For example, to find all "defunct" processes on a 64 node system, use

```
pps 1-64 aux | grep '<defunct>' | sort
```

Note that this does not run grep and sort in parallel. An alternative approach is described below that uses pfps.

2.3 Parallel ls

The command pls runs ls on the specified systems.

Exceptions: The option -t to ls sorts the files by time; the output from pls will preserve this only on a processor-by-processor basis. The behavior is the same for all other options to ls that sort the output.

2.4 Parallel cat, cp, mv, and rm

The command pcp copies a file from a single location to the local disks on a specified list of processors. For example, to copy 'mycode' to '/tmp/myname/mycode' on processors 1 and 32 through 63, use

```
pcp 1,32-63 mycode /tmp/myname/mycode
```

We considered using the name pdist rather than the name pcp because pcp does a one-to-many copy. We decided that pcp was a better choice because a common use is the parallel version of

```
cp mycode /tmp/myname/mycode
```

that is, the distribution of an executable or data file to the local disks.

The command `pcat` concatenates files from the specified nodes onto standard output (`stdout`). We note there are aspects of this command that are inherently non-scalable; however, it is so useful that it needs to be provided. The command

```
pcat 1-10 /tmp/testfile > myfile
```

concatenates the file '`/tmp/testfile`' on nodes one through ten to the file '`myfile`'. The results are concatenated in the listed node-number order.

The command `prm` executes `rm` on the specified nodes.

The command `pmv` executes `mv` on the specified nodes. Files may only be moved within a single processor. That is, a file may be moved from one place to another on the local disk of a processor, for each processor selected.

Note that in all of these cases, the interactive option (`-i`) is not supported.

2.5 Parallel find

The command `pfind` executes the Unix command `find` on the specified list of processors. For example, to find all of the files on the local disks that are older than two days, use

```
pfind 1-128 /tmp -atime ... -print
```

2.6 Parallel process find

Many of the uses of `ps` are similar to the uses of `ls`, such as determining the age of a process (resp. file) or owner of a process (resp. file). Because a file system often contains large numbers of individual files, the Unix command `find` provides a way to find files that satisfy some common properties. Because the number of processes is relatively small, there has been no counterpart to `find` for processes. However, with 30 to 60 processes on each processor, a `ps` of even a small parallel system can generate hundreds to thousands of lines of output. In this section, we propose a process find (and its parallel version) that provides the same style of functionality that `find` provides for the file system.

Just as with `find`, multiple matching criteria are and'ed together. For example, to find out which processes named `bigjob` have been running for at least one day, use

```
pfps -all -tn bigjob -stime 1:0:0 -print
```

The options for `pfps` are given in Table 1.

2.7 Parallel predicate

This command uses a user-specified predicate to select which nodes to execute a user-specified command on.

```
ppred nodespec predicate action
```

For example, to find out on what processors in a 128 node system the file `/tmp/prog` is *not* present, you can use (assuming `csh` is the shell)

```
ppred 1-128 '\!-s /tmp/prog' 'echo $hostname'
```

(note the escape on the c-shell 'not' symbol `!` and the use of `'...'` to prevent premature evaluation of the predicate and action.

2.8 Parallel test

This command forms the logical 'and' of the results of running `test` on each selected node.

```
ptest nodespec testcondition
```

For example, to check if all processors have the file `/tmp/myprog`, you can use

```
ptest 1-128 '-s /tmp/myprog'
```

2.9 Parallel kill

The command `pkill` kills a named process on the selected nodes. It is basically a simplification of `pfps`; the command

```
pkill 1,10-24 SIGQUIT -tn myprogram
```

is equivalent to

```
pfps 1,10-24 -tn myprogram -kill SIGQUIT
```

2.10 Parallel execution

The command `pexec` provides a way to execute an arbitray command or (uniprocessor) Unix program on a list of processors. The format of this command is `pexec nodelist ...command...`. For example, to run `ps` on each node *and* grep for `<defunct>` in parallel, use

```
pexec 1-64 "ps aux | grep '<defunct>'" | sort
```

Option	Description
`-n` name	Match with the name of the process. The name may contain wildcards.
`-tn`	Match the tail name of the executable
`-o` owner	Match with the owner (by name) of the process. By default, only the user name of the caller is matched. Use `-o '*'` to match any user name.
`-pty` name	Match with the controlling terminal of the process
`-rtime` hh:mm	Match with jobs that have run hh:mm time or longer.
`-stime` dd:hh:mm	Match with jobs that started at least dd days, hh hours, and mm minutes ago.
`-r` state	Match with jobs in the specified run state
`-or`	Combine matching criteria by or'ing them.
`-print`	Causes matching jobs to be printed in the selected ps format.
`-id`	Causes matching jobs to be printed as `nodename:pid`.
`-sort`	Causes the output to be sorted by nodename
`-exec` pgm args	Executes pgm for each matching process. Similar to `find`, the string \{\} stands for the pid of the matched process, and \; indicates the end of the list of arguments to give to the program.
`-kill` signal	Causes all matched processes to be killed with the specified signal. The signal value may be either the number or the name (for example, `-kill 9` and `-kill SIGQUIT` are the same).
`-nice` n	Sets the nice value of matched jobs.

Table 1: Options for the `pfps` command

The prototype implementation uses `pexec` to implement many of the functions described in this paper. Any output generated from the commands is prepended by the name of the processor that generated it.

2.11 Parallel execute script

The command `pexscr` takes input from standard input and executes each line on the indicated processor. The format of the input is

```
processor_name arbitrary_command
```

This format matches the output format from the other parallel commands, allowing `awk` or `perl` to construct command scripts to execute from the output of the parallel commands.

2.12 Parallel distribute execution

The command `pdistrib` takes a list of files and a command to apply to the files, and distributes the processing of the files across the specified processors. For example

```
pdistrib -all "cc -c" *.c
```

causes the compilation of all of the C files in the current directory to be distributed across all available processors.

2.13 Parallel display

The command `pdisp` takes input from standard input and displays it.

The options for `pdisp` are

Option	Action
`-yes colorname`	Color of nodes appearing in input
`-no colorname`	Color of nodes not appearing in input
`-down colorname`	Color of down nodes
`-text string`	Text for nodes appearing in input. This string may contain formatting information such as `$3` for the third token in the line.
`-small`	Do not display text unless button pressed (produces small display)
`-store`	Save text with node; pushing the left mouse button will display the text.
`-layout RxC`	Layout of `R` rows and `C` columns
`-domain name`	Name of the machine's domain
`-pserver name`	Use a pre-existing display
`-pstart name`	Make this a `pdisp` display server

For example, to graphically display the nodes on which the program `bigjob` is running, use

```
pfps -all -tn bigjob | pdisp
```

The options `-pstart` and `-pserver` allow the display window to be reused by several commands. For example, to create a display, then display the nodes without the file '`/tmp/mydata`', and then display the nodes with defunct jobs, do

```
pdisp -pstart mydisp
ppred -all \!-s /tmp/mydata | \
                pdisp -pserver mydisp
pps -all aux | grep '<defunct>' | \
               pdisp -pserver mydisp
```

A sample display is shown in Figure 1.

Each node on the display in Figure 1 is actually a button. For example, using the middle button of a three-button mouse pops up an xterm on the indicated node. Pushing the left button pops up all of the output associated with that node.

2.14 Parallel partition info

Many MPP's provide a mechanism for reserving parts or all of the MPP for use by a single program. These are often called partitions. The command `pinfo` displays the partitions in use and the user that owns that partition. It takes many of the same arguments as `pdisp`. A sample display is shown in Figure 2 (node 32 is down).

Figure 2: Display of partition availability and users of partitions.

3 Implementation

It is important that these commands themselves execute in parallel. In interactive use, it is common to expect a command to complete in a second or less. The parallel version of the same command should not take much longer. This requires that the commands be executed in parallel.

A simple way to arrange for parallel execution is to use *recursive subdivision*. Each node is given some number of processes to run a command on. It divides that list in half, and sends the upper half to the first processor in that half. This process continues until only one process is left. This takes $\log p$ steps for p processes. A simple form of this is shown in Figure 3 for `pls`. This sample code has no error checking and assumes a single range of processors from `start` to `end`. The names of the nodes are `spnode`i, for $i = 1, \ldots$

Various optimizations of this process are possible. For example, for small numbers, the recursive subdivision may be replaced with a simple loop. Other optimizations can take advantage of the particular structure of a parallel machine, adapting the subdivision strategy to the available communication network and services.

In order to provide maximum parallelism, each subdivision must execute the subdivided processes in the background. It is important to ensure that a command does not return until all of its children have completed.

Many of these commands can be implemented in

```
#! /bin/csh
set start = $1
set end   = $2
shift
shift
set nodename = `hostname`
ls $* |& sed "s/^/$nodename/g"
while (1)
  if ($start >= $end) break
  # Compute the separator for the tree
  @ sep = ($start + $end) / 2
  if ($sep == $start) then
    @ sep = $sep + 1
    if ($sep > $end) break
  endif
  rsh spnode$sep -n /tmp/pls $sep $end $* &
  @ end = $sep - 1
end
wait
```

Figure 3: Simplified code for parallel `ls`

terms of `pexec` or `ppred`, perhaps combined with some relatively simple `awk` or `perl` scripts. We have chosen to provide a larger set of commands because they represent common cases for which we believe shortcuts should be provided.

On many systems, the time to load a program from a central filesystem can be significant. On these systems, programs (including these tools) should be loaded from local disks (assumed to be '`/tmp`'). Our prototype implementation includes a program `ptinit` that copies the codes to `/tmp`. The programs that are executed by these tools (e.g., `ps`) should also reside on local disks where possible.

To handle the case where a node in the list is not responding or unavailable, the programs should issue a warning message and skip to the next process in the upper half to insure that processors are not missed because their 'parent' in the subdivision tree was not available. Because it is time-comsuming to detect down nodes, a replicated database of down nodes should be used.

The implementation of parallel exec should sort the input script and use the recursive subdivision (or at least collect all commands for the same processor together and send them in a lump).

Also note that all of these commands can execute faster if a server process is always running on each of the parallel processors. Such a server is not required however; the prototype implementation is written entirely in terms of shell scripts.