GNU Parallel: The Command-Line Power Tool

OLE TANGE

Ole Tange works in bioinformatics in Copenhagen. He is active in the free software community and is best known for his "patented Web shop" that shows the dangers of software patents (http://ole.tange.dk/swpat). He will be happy to go to your conference to give a talk about GNU Parallel.
ole@tange.dk

For daily sysadmin work you often need to do the same specific task to a lot of files/users/hosts/tables. Very often it doesn't matter which one is done first and it would be fine to do them in parallel—but you do not want to run them *all* in parallel, as that will slow your computer to a crawl. This is what GNU Parallel can help you do.

In this article you will see examples of uses of GNU Parallel. The examples are kept simple to make it easy to understand the idea, but there's nothing that prevents you from using GNU Parallel for more complex tasks. More complex examples can be found at http://www.gnu.org/software/parallel/man.html.

History of GNU Parallel

GNU Parallel started out as two separate programs: xxargs and parallel. The purpose of xxargs was to work like xargs but deal nicely with space and quotes. The purpose of parallel was simply to run jobs in parallel. Both programs originated in 2001. In 2005 the two programs were merged, since xxargs was very often used with parallel, and thus the name xxargs was abandoned.

GNU Parallel therefore has two main objectives: replace xargs and run commands in parallel.

In 2010 Parallel was adopted as an official GNU tool and the name was changed to GNU Parallel. For a short video showing simple usage of the tool go to http://nd.gd/0s.

Your First Parallel Job

GNU Parallel is available as a package for most UNIX distributions. See http://www.gnu.org/s/parallel if it is not obvious how to install it on your system. After installation find a bunch of files on your computer and gzip them in parallel:

```
parallel gzip ::: *
```

Here your shell expands * to the files, ::: tells GNU Parallel to read arguments from the command line, and gzip is the command to run. The jobs are then run in parallel. After you have gziped the files, you can recompress them with bzip2:

```
parallel "zcat {} | bzip2 >{.}.bz2" ::: *
```

Here {} is being replaced with the file name. The output from zcat is piped to bzip2, which then compresses the output. The {.} is the file name with its extension stripped (e.g., foo.gz becomes foo), so the output from file.gz is stored in file.bz2.

GNU Parallel tries to be very liberal in quoting, so the above could also be written:

```
parallel zcat {} "|" bzip2 ">"{.}.bz2 ::: *
```

Only the chars that have special meaning in shell need to be quoted.

Reading Input

As we have seen, input can be given on the command line. Input can also be piped into GNU Parallel:

```
find . -type f | parallel gzip
```

GNU Parallel uses newline as a record separator and deals correctly with file names containing a word space or a dot. If you have normal users on your system, you will have experienced file names like these. If your users are really mean and write file names containing newlines, you can use NULL as a record separator:

```
find . -type f -print0 | parallel -0 gzip
```

You can read from a file using -a:

```
parallel -a filelist gzip
```

If you use more than one -a, a line from each input file will be available as {#}:

```
parallel -a sourcelist -a destlist gzip {1} ">"{2}
```

The same goes if you read a specific number of arguments at a time using -N:

```
cat filelist | parallel -N 3 diff {1} {2} ">" {3}
```

If your input is in columns you can split the columns using --colsep:

```
cat filelist.tsv | parallel --colsep '\t' diff {1} {2} ">" {3}
```

--colsep is a regexp, so you can match more advanced column separators.

Building the Command to Run

Just like xargs, GNU Parallel can take multiple input lines and put those on the same line. Compare these:

```
ls *.gz | parallel mv {} archive
ls *.gz | parallel -X mv {} archive
```

The first will run mv for every .gz file, whereas the second will fit as many files into {} as possible before running.

The {} can be put anywhere in the command, but if it is part of a word, that word will be repeated when using -X:

```
(echo 1; echo 2) | parallel -X echo foo bar{}baz quux
```

will repeat bar-baz and print:

```
foo bar1baz bar2baz quux
```

If you do not give a command to run, GNU Parallel will assume the input lines are command lines and run those in parallel:

```
(echo ls; echo grep root /etc/passwd) | parallel
```

Controlling the Output

One of the problems with running jobs in parallel is making sure the output of the running commands do not get mixed up. traceroute is a good example of this as it prints out slowly and parallel traceroutes will get mixed up. Try:

```
traceroute foss.org.my & traceroute debian.org & traceroute freenetproject.
org & wait
```

and compare the output to:

```
parallel traceroute ::: foss.org.my debian.org freenetproject.org
```

As you can see, GNU Parallel only prints out when a job is done—thereby making sure the output is never mixed with other jobs. If you insist, GNU Parallel can give you the output immediately with -u, but output from different jobs may mix.

For some input, you want the output to come in the same order as the input. -k does that for you:

```
parallel -k echo {}';' sleep {} ::: 3 2 1 4
```

This will run the four commands in parallel, but postpone the output of the two middle commands until the first is finished.

Execution of the Jobs

GNU Parallel defaults to run one job per CPU core in parallel. You can change this with -j. You can put an integer as the number of jobs (e.g., -j 4 for four jobs in parallel) or you can put a percentage of the number of CPU cores (e.g., -j 200% to run two jobs per CPU core):

```
parallel -j200% gzip ::: *
```

If you pass -j a file name, the parameter will be read from that file:

```
parallel -j /tmp/number_of_jobs_to_run gzip ::: *
```

The file will be read again after each job finishes. This way you can change the number of jobs running during a parallel run. This is particularly useful if it is a very long run and you need to prioritize other tasks on the computer.

To list the currently running jobs you need to send GNU Parallel SIGUSR1:

```
killall -USR1 parallel
```

GNU Parallel will then print the currently running jobs on STDERR.

If you regret starting a lot of jobs, you can simply break GNU Parallel, but if you want to make sure you do not have half-completed jobs, you should send the signal SIGTERM to GNU Parallel:

```
killall -TERM parallel
```

This will tell GNU Parallel not to start any new jobs but to wait until the currently running jobs are finished before exiting.

When monitoring the progress on screen it is often nice to have the output prepended with the command that was run. -v will do that for you:

```
parallel -v gzip ::: *
```

If you want to monitor the progress as it goes you can also use --progress and --eta:

```
parallel --progress gzip ::: *
parallel --eta gzip ::: *
```

This is especially useful for debugging when jobs run on remote computers.

Remote Computers

GNU Parallel can use the CPU power of remote computers to help do computations. As an example, we will recompress .gz files into .bz2 files, but you can just as easily do other compute-intensive jobs such as video encoding or image transformation.

You will need to be able to log in to the remote host using ssh without entering a password (ssh-agent may be handy for that). To transfer files, rsync needs to be installed, and to help GNU Parallel figure out how many CPU cores each computer has, GNU Parallel should also be installed on the remote computers.

Try this simple example to see that your setup is working:

```
parallel --sshlogin yourserver.example.com hostname';' echo {} ::: 1 2 3
```

This should print out the hostname of your server three times, each followed by the numbers 1 2 3. --sshlogin can be shortened to -S. To use more than one server, do:

```
parallel -S yourserver.example.com,server2.example.net hostname';' echo {} :::
1 2 3
```

If you have a different login name, just prepend login@ to the server name—just as you would with ssh. You can also give more than one -S instead of using a comma:

```
parallel -S yourserver.example.com -S mylogin@server2.example.net hostname';'
echo {} ::: 1 2 3
```

The special sshlogin ':' is your local computer:

```
parallel -S yourserver.example.com -S mylogin@server2.example.net -S :
hostname';' echo {} ::: 1 2 3
```

In this case you may see that GNU Parallel runs all three jobs on your local computer ,because the jobs are so fast to run.

If you have a file containing a list of the sshlogins to use, you can tell GNU Parallel to use that file:

```
parallel --sshloginfile mylistofsshlogins hostname';' echo {} ::: 1 2 3
```

The special sshlogin .. will read the sshloginfile ~/.parallel/sshloginfile:

```
parallel -S .. hostname';' echo {} ::: 1 2 3
```

Transferring Files

If your servers are not sharing storage (using NFS or something similar), you often need to transfer the files to be processed to the remote computers and the results back to the local computer.

To transfer a file to a remote computer, you will use --transfer:

```
parallel -S .. --transfer gzip '< {} | wc -c' ::: *.txt
```

Here we transfer each of the .txt files to the remote servers, compress them, and count how many bytes they now take up.

After a transfer you often will want to remove the transferred file from the remote computers. --cleanup does that for you:

```
parallel -S .. --transfer --cleanup gzip '< {} | wc -c' ::: *.txt
```

When processing files the result is often a file that you want copied back, after which the transferred and the result file should be removed from the remote computers:

```
parallel -S .. --transfer --return {.}.bz2 --cleanup zcat '< {} | bzip2 >{.}.bz2'
::: *.gz
```

Here the .gz files will be transferred and then recompressed using zcat and bzip2. The resulting .bz2 file is transferred back, and the .gz and the .bz2 files are removed from the remote computers. The combination --transfer --cleanup --return foo is used so often that it has its own abbreviation: --trc foo.

You can specify multiple --trc if your command generates multiple result files.

GNU Parallel will try to detect the number of cores on remote computers and run one job per CPU core even if the computers have different number of CPU cores:

```
parallel -S .. --trc {.}.bz2 zcat '< {} | bzip2 >{.}.bz2' ::: *.gz
```

GNU Parallel as Part of a Script

The more you practice using GNU Parallel, the more places you will see it can be useful. Every time you write a for loop or a while-read loop, consider if this could be done in parallel. Often the for loop can completely be replaced with a single line using GNU Parallel; if the jobs you want to run in parallel are very complex, you may want to make a script and have GNU Parallel call that script. Occasionally your for loop is so complex that neither of these is an option.

This is where parallel --semaphore can help you out. sem is the short alias for parallel --semaphore.

```
for i in `ls *.log` ; do
        [... a lot of complex lines here ...]
        sem -j4 --id my_id do_work $i
done
sem --wait --id my_id
```

This will run do_work in the background until four jobs are running. Then sem will wait until one of the four jobs has finished before starting another job. The last line (sem --wait) pauses the script until all the jobs started by sem have finished. my_id is a unique string used by sem to identify this script, since you may have other sems running at the same time. If you only run sem from one script at a time, --id my_id can be left out.

A special case is sem -j1, which works like a mutex and is useful if you only want one program running at a time.

GNU Parallel as a Job Queue Manager

With a few lines of code, GNU Parallel can work as a job queue manager:

```
echo >jobqueue; tail -f jobqueue | parallel
```

To submit your jobs to the queue, do:

```
echo my_command my_arg >> jobqueue
```

You can, of course, use -S to distribute the jobs to remote computers:

```
echo >jobqueue; tail -f jobqueue | parallel -S ..
```

If you have a dir into which users drop files that need to be processed, you can do this on GNU/Linux:

```
inotifywait -q -m -r -e CLOSE_WRITE --format %w%f my_dir | parallel -u echo
```

This will run the command echo on each file put into my_dir or subdirs of my_dir. Here again you can use -S to distribute the jobs to remote computers:

```
inotifywait -q -m -r -e CLOSE_WRITE --format %w%f my_dir | parallel -S ..  -u echo
```

See You on the Mailing List

I hope you have thought of situations where GNU Parallel can be of benefit to you. If you like GNU Parallel please let others know about it through email lists, forums, blogs, and social networks. If GNU Parallel saves you money, please donate to the FSF https://my.fsf.org/donate. If you have questions about GNU Parallel, join the mailing list at http://lists.gnu.org/mailman/listinfo/parallel.