GNU Parallel Tutorial

This tutorial shows off much of GNU Parallel's functionality. The tutorial is meant to learn the options in GNU Parallel. The tutorial is not to show realistic examples from the real world.

Spend an hour walking through the tutorial. Your command line will love you for it.

Prerequisites

To run this tutorial you must have the following:

parallel >= version 20130814

Install the newest version with:

```
(wget -O - pi.dk/3 || curl pi.dk/3/) | bash
```

This will also install the newest version of the tutorial:

```
man parallel_tutorial
```

Most of the tutorial will work on older versions, too.

abc-file:

The file can be generated by:

```
parallel -k echo ::: A B C > abc-file
```

def-file:

The file can be generated by:

```
parallel -k echo ::: D E F > def-file
```

abc0-file:

The file can be generated by:

```
perl -e 'printf "A\0B\0C\0"' > abc0-file
```

abc_-file:

The file can be generated by:

```
perl -e 'printf "A_B_C_"' > abc_-file
```

tsv-file.tsv

The file can be generated by:

```
perl -e 'printf "f1\tf2\nA\tB\nC\tD\n"' > tsv-file.tsv
```

num30000

The file can be generated by:

```
perl -e 'for(1..30000){print "$_\n"}' > num30000
```

num1000000

The file can be generated by:

```
perl -e 'for(1..1000000){print "$_\n"}' > num1000000
```

num_%header

The file can be generated by:

```
(echo %head1; echo %head2; perl -e 'for(1..10){print
```

```
            "$_\n"}') > num_%header
```

For remote running: ssh login on 2 servers with no password in $SERVER1 and $SERVER2

```
        SERVER1=server.example.com
        SERVER2=server2.example.net
```

You must be able to:

```
    ssh $SERVER1 echo works
    ssh $SERVER2 echo works
```

It can be setup by running 'ssh-keygen -t dsa; ssh-copy-id $SERVER1' and using an empty pass phrase.

Input sources

GNU Parallel reads input from input sources. These can be files, the command line, and stdin (standard input or a pipe).

A single input source

Input can be read from the command line:

```
parallel echo ::: A B C
```

Output (the order may be different because the jobs are run in parallel):

```
A
B
C
```

The input source can be a file:

```
parallel -a abc-file echo
```

Output: Same as above.

STDIN (standard input) can be the input source:

```
cat abc-file | parallel echo
```

Output: Same as above.

Multiple input sources

GNU Parallel can take multiple input sources given on the command line. GNU Parallel then generates all combinations of the input sources:

```
parallel echo ::: A B C ::: D E F
```

Output (the order may be different):

```
A D
A E
A F
B D
B E
B F
C D
C E
C F
```

The input sources can be files:

```
parallel -a abc-file -a def-file echo
```

Output: Same as above.

STDIN (standard input) can be one of the input sources using '-':

```
cat abc-file | parallel -a - -a def-file echo
```

Output: Same as above.

Instead of -a files can be given after ':::::':

```
cat abc-file | parallel echo ::::: - def-file
```

Output: Same as above.

::: and :::: can be mixed:

```
parallel echo ::: A B C :::: def-file
```

Output: Same as above.

Matching arguments from all input sources

With --xapply you can get one argument from each input source:

```
parallel --xapply echo ::: A B C ::: D E F
```

Output (the order may be different):

```
A D
B E
C F
```

If one of the input sources is too short, its values will wrap:

```
parallel --xapply echo ::: A B C D E ::: F G
```

Output (the order may be different):

```
A F
B G
C F
D G
E F
```

Changing the argument separator.

GNU Parallel can use other separators than ::: or ::::. This is typically useful if ::: or :::: is used in the command to run:

```
parallel --arg-sep ,, echo ,, A B C ::::: def-file
```

Output (the order may be different):

```
A D
A E
A F
```

```
B D
B E
B F
C D
C E
C F
```

Changing the argument file separator:

```
parallel --arg-file-sep // echo ::: A B C // def-file
```

Output: Same as above.

Changing the argument delimiter

GNU Parallel will normally treat a full line as a single argument: It uses \n as argument delimiter. This can be changed with -d:

```
parallel -d _ echo :::: abc_-file
```

Output (the order may be different):

```
A
B
C
```

NULL can be given as \0:

```
parallel -d '\0' echo :::: abc0-file
```

Output: Same as above.

A shorthand for -d '\0' is -0 (this will often be used to read files from find ... -print0):

```
parallel -0 echo :::: abc0-file
```

Output: Same as above.

End-of-file value for input source

GNU Parallel can stop reading when it encounters a certain value:

```
parallel -E stop echo ::: A B stop C D
```

Output:

```
A
B
```

Skipping empty lines

Using --no-run-if-empty GNU Parallel will skip empty lines.

```
(echo 1; echo; echo 2) | parallel --no-run-if-empty echo
```

Output:

```
1
2
```

Building the command line

No command means arguments are commands

If no command is given after parallel the arguments themselves are treated as commands:

```
parallel ::: ls 'echo foo' pwd
```

Output (the order may be different):

```
[list of files in current dir]
foo
[/path/to/current/working/dir]
```

The command can be a script, a binary or a Bash function if the function is exported using 'export -f':

```
# Only works in Bash and only if $SHELL=.../bash
my_func() {
  echo in my_func $1
}
export -f my_func
parallel my_func ::: 1 2 3
```

Output (the order may be different):

```
in my_func 1
in my_func 2
in my_func 3
```

Replacement strings

The 5 replacement strings

GNU Parallel has several replacement strings. If no replacement strings are used the default is to append {}:

```
parallel echo ::: A/B.C
```

Output:

```
A/B.C
```

The default replacement string is {}:

```
parallel echo {} ::: A/B.C
```

Output:

```
A/B.C
```

The replacement string {.} removes the extension:

```
parallel echo {.} ::: A/B.C
```

Output:

```
A/B
```

The replacement string {/} removes the path:

```
parallel echo {/} ::: A/B.C
```

Output:

```
B.C
```

The replacement string {//} keeps only the path:

```
parallel echo {//} ::: A/B.C
```

Output:

```
A
```

The replacement string {/.} removes the path and the extension:

```
parallel echo {/.} ::: A/B.C
```

Output:

```
B
```

The replacement string {#} gives the job number:

```
parallel echo {#} ::: A B C
```

Output (the order may be different):

```
1
2
3
```

Changing the replacement strings

The replacement string {} can be changed with -I:

```
parallel -I ,, echo ,, ::: A/B.C
```

Output:

```
A/B.C
```

The replacement string {.} can be changed with --extensionreplace:

```
parallel --extensionreplace ,, echo ,, ::: A/B.C
```

Output:

```
A/B
```

The replacement string {/} can be replaced with --basenamereplace:

```
parallel --basenamereplace ,, echo ,, ::: A/B.C
```

Output:

```
B.C
```

The replacement string {//} can be changed with --dirnamereplace:

```
parallel --dirnamereplace ,, echo ,, ::: A/B.C
```

Output:

```
A
```

The replacement string {/.} can be changed with --basenameextensionreplace:

```
parallel --basenameextensionreplace ,, echo ,, ::: A/B.C
```

Output:

```
B
```

The replacement string {#} can be changed with --seqreplace:

```
parallel --seqreplace ,, echo ,, ::: A B C
```

Output (the order may be different):

```
1
2
3
```

Positional replacement strings

With multiple input sources the argument from the individual input sources can be access with {number}:

```
parallel echo {1} and {2} ::: A B ::: C D
```

Output (the order may be different):

```
A and C
A and D
B and C
B and D
```

The positional replacement strings can also be modified using / // /. and .:

```
parallel echo /={1/} //={1//} /.={1/.} .={1.} ::: A/B.C D/E.F
```

Output (the order may be different):

```
/=B.C //=A /.=B .=A/B
/=E.F //=D /.=E .=D/E
```

If a position is negative, it will refer to the input source counted from behind:

```
parallel echo 1={1} 2={2} 3={3} -1={-1} -2={-2} -3={-3} ::: A B ::: C D
::: E F
```

Output (the order may be different):

```
1=A 2=C 3=E -1=E -2=C -3=A
1=A 2=C 3=F -1=F -2=C -3=A
```

```
1=A 2=D 3=E -1=E -2=D -3=A
1=A 2=D 3=F -1=F -2=D -3=A
1=B 2=C 3=E -1=E -2=C -3=B
1=B 2=C 3=F -1=F -2=C -3=B
1=B 2=D 3=E -1=E -2=D -3=B
1=B 2=D 3=F -1=F -2=D -3=B
```

Input from columns

The columns in a file can be bound to positional replacement strings using --colsep. Here the columns are separated with TAB (\t):

```
parallel --colsep '\t' echo 1={1} 2={2} :::: tsv-file.tsv
```

Output (the order may be different):

```
1=f1 2=f2
1=A 2=B
1=C 2=D
```

Header defined replacement strings

With --header GNU Parallel will use the first value of the input source as the name of the replacement string. Only the non-modified version {} is supported:

```
parallel --header : echo f1={f1} f2={f2} ::: f1 A B ::: f2 C D
```

Output (the order may be different):

```
f1=A f2=C
f1=A f2=D
f1=B f2=C
f1=B f2=D
```

It is useful with --colsep for processing files with TAB separated values:

```
parallel --header : --colsep '\t' echo f1={f1} f2={f2} :::: tsv-file.tsv
```

Output (the order may be different):

```
f1=A f2=B
f1=C f2=D
```

More than one argument

With --xargs will GNU Parallel fit as many arguments as possible on a single line:

```
cat num30000 | parallel --xargs echo | wc -l
```

Output:

```
2
```

The 30000 arguments fitted on 2 lines.

The maximal length of a single line can be set with -s. With a maximal line length of 10000 chars 17 commands will be run:

```
cat num30000 | parallel --xargs -s 10000 echo | wc -l
```

Output:

```
17
```

For better parallelism GNU Parallel can distribute the arguments between all the parallel jobs when end of file is met.

Below GNU Parallel reads the last argument when generating the second job. When GNU Parallel reads the last argument, it spreads all the arguments for the second job over 4 jobs instead, as 4 parallel jobs are requested.

The first job will be the same as the --xargs example above, but the second job will be split into 4 evenly sized jobs, resulting in a total of 5 jobs:

```
cat num30000 | parallel --jobs 4 -m echo | wc -l
```

Output:

```
5
```

This is even more visible when running 4 jobs with 10 arguments. The 10 arguments are being spread over 4 jobs:

```
parallel --jobs 4 -m echo ::: {1..10}
```

Output:

```
1 2 3
4 5 6
7 8 9
10
```

A replacement string can be part of a word. -m will not repeat the context:

```
parallel --jobs 4 -m echo pre-{}-post ::: A B C D E F G
```

Output (the order may be different):

```
pre-A B-post
pre-C D-post
pre-E F-post
pre-G-post
```

To repeat the context use -X which otherwise works like -m:

```
parallel --jobs 4 -X echo pre-{}-post ::: A B C D E F G
```

Output (the order may be different):

```
pre-A-post pre-B-post
pre-C-post pre-D-post
pre-E-post pre-F-post
pre-G-post
```

To limit the number of arguments use -N:

```
parallel -N3 echo ::: A B C D E F G H
```

Output (the order may be different):

```
A B C
D E F
G H
```

-N also sets the positional replacement strings:

```
parallel -N3 echo 1={1} 2={2} 3={3} ::: A B C D E F G H
```

Output (the order may be different):

```
1=A 2=B 3=C
1=D 2=E 3=F
1=G 2=H 3=
```

-N0 reads 1 argument but inserts none:

```
parallel -N0 echo foo ::: 1 2 3
```

Output:

```
foo
foo
foo
```

Quoting

Command lines that contain special characters may need to be protected from the shell.

The perl program 'print "@ARGV\n"' basically works like echo.

```
perl -e 'print "@ARGV\n"' A
```

Output:

```
A
```

To run that in parallel the command needs to be quoted:

```
parallel perl -e 'print "@ARGV\n"' ::: This wont work
```

Output:

```
[Nothing]
```

To quote the command use -q:

```
parallel -q perl -e 'print "@ARGV\n"' ::: This works
```

Output (the order may be different):

```
This
works
```

Or you can quote the critical part using \':

```
parallel perl -e \''print "@ARGV\n"'\' ::: This works, too
```

Output (the order may be different):

```
This
works,
too
```

GNU Parallel can also \-quote full lines. Simply run:

```
parallel --shellquote
parallel: Warning: Input is read from the terminal. Only experts do this
on purpose. Press CTRL-D to exit.
perl -e 'print "@ARGV\n"'
[CTRL-D]
```

Output:

```
perl\ -e\ \'print\ \"@ARGV\\n\"\'
```

This can then be used as the command:

```
parallel perl\ -e\ \'print\ \"@ARGV\\n\"\' ::: This also works
```

Output (the order may be different):

```
This
also
works
```

Trimming space

Space can be trimmed on the arguments using --trim:

```
parallel --trim r echo pre-{}-post ::: ' A '
```

Output:

```
pre- A-post
```

To trim on the left side:

```
parallel --trim l echo pre-{}-post ::: ' A '
```

Output:

```
pre-A -post
```

To trim on the both sides:

```
parallel --trim lr echo pre-{}-post ::: ' A '
```

Output:

```
pre-A-post
```

Controling the output

The output can prefixed with the argument:

```
parallel --tag echo foo-{} ::: A B C
```

Output (the order may be different):

```
A       foo-A
B       foo-B
C       foo-C
```

To prefix it with another string use --tagstring:

```
parallel --tagstring {}-bar echo foo-{} ::: A B C
```

Output (the order may be different):

```
A-bar   foo-A
B-bar   foo-B
C-bar   foo-C
```

To see what commands will be run without running them:

```
parallel --dryrun echo {} ::: A B C
```

Output (the order may be different):

```
echo A
echo B
echo C
```

To print the command before running them use --verbose:

```
parallel --verbose echo {} ::: A B C
```

Output (the order may be different):

```
echo A
echo B
A
echo C
B
C
```

GNU Parallel will postpone the output until the command completes:

```
parallel -j2 'printf "%s-start\n%s" {} {};sleep {};printf "%s\n"
-middle;echo {}-end' ::: 4 2 1
```

Output:

```
2-start
2-middle
2-end
1-start
1-middle
1-end
4-start
4-middle
4-end
```

To get the output immediately use --ungroup:

```
parallel -j2 --ungroup 'printf "%s-start\n%s" {} {};sleep {};printf
"%s\n" -middle;echo {}-end' ::: 4 2 1
```

Output:

```
4-start
42-start
2-middle
2-end
1-start
1-middle
1-end
-middle
4-end
```

--ungroup is fast, but can cause half a line from one job to be mixed with half a line of another job. That has happend in the second line, where the line '4-middle' is mixed with '2-start'.

To avoid this use --linebuffer (which, however, is much slower):

```
parallel -j2 --linebuffer 'printf "%s-start\n%s" {} {};sleep {};printf
"%s\n" -middle;echo {}-end' ::: 4 2 1
```

Output:

```
4-start
2-start
2-middle
2-end
1-start
1-middle
1-end
4-middle
4-end
```

To force the output in the same order as the arguments use --keep-order/-k:

```
parallel -j2 -k 'printf "%s-start\n%s" {} {};sleep {};printf "%s\n"
-middle;echo {}-end' ::: 4 2 1
```

Output:

```
4-start
4-middle
4-end
2-start
2-middle
2-end
1-start
1-middle
1-end
```

Saving output into files

GNU Parallel can save the output of each job into files:

```
parallel --files ::: A B C
```

Output will be similar to:

```
/tmp/pAh6uWuQCg.par
/tmp/opjhZCzAX4.par
/tmp/W0AT_Rph2o.par
```

By default GNU Parallel will cache the output in files in /tmp. This can be changed by setting $TMPDIR or --tmpdir:

```
parallel --tmpdir /var/tmp --files ::: A B C
```

Output will be similar to:

```
/var/tmp/N_vk7phQRc.par
/var/tmp/7zA4Ccf3wZ.par
/var/tmp/LIuKgF_2LP.par
```

Or:

```
TMPDIR=/var/tmp parallel --files ::: A B C
```

Output: Same as above.

The output files can be saved in a structured way using --results:

```
parallel --results outdir echo ::: A B C
```

Output:

```
A
B
C
```

but also these files were generated containing the standard output (stdout) and standard error (stderr):

```
outdir/1/A/stderr
outdir/1/A/stdout
outdir/1/B/stderr
outdir/1/B/stdout
outdir/1/C/stderr
outdir/1/C/stdout
```

This is useful if you are running multiple variables:

```
parallel --header : --results outdir echo ::: f1 A B ::: f2 C D
```

Generated files:

```
outdir/f1/A/f2/C/stderr
outdir/f1/A/f2/C/stdout
outdir/f1/A/f2/D/stderr
outdir/f1/A/f2/D/stdout
outdir/f1/B/f2/C/stderr
outdir/f1/B/f2/C/stdout
outdir/f1/B/f2/D/stderr
outdir/f1/B/f2/D/stdout
```

The directories are named after the variables and their values.

Control the execution

Number of simultaneous jobs

The number of concurrent jobs is given with --jobs/-j:

```
/usr/bin/time parallel -N0 -j64 sleep 1 ::: {1..128}
```

With 64 jobs in parallel the 128 sleeps will take 2-8 seconds to run - depending on how fast your machine is.

By default --jobs is the same as the number of CPU cores. So this:

```
/usr/bin/time parallel -N0 sleep 1 ::: {1..128}
```

should take twice the time of running 2 jobs per CPU core:

```
/usr/bin/time parallel -N0 --jobs 200% sleep 1 ::: {1..128}
```

--jobs 0 will run as many jobs in parallel as possible:

```
/usr/bin/time parallel -N0 --jobs 0 sleep 1 ::: {1..128}
```

which should take 1-7 seconds depending on how fast your machine is.

--jobs can read from a file which is re-read when a job finishes:

```
echo 50% > my_jobs
/usr/bin/time parallel -N0 --jobs my_jobs sleep 1 ::: {1..128} &
sleep 1
echo 0 > my_jobs
wait
```

The first second only 50% of the CPU cores will run a job. The '0' is put into my_jobs and then the rest of the jobs will be started in parallel.

Instead of basing the percentage on the number of CPU cores GNU Parallel can base it on the number of CPUs:

```
parallel --use-cpus-instead-of-cores -N0 sleep 1 ::: {1..128}
```

Interactiveness

GNU Parallel can ask the user if a command should be run using --interactive:

```
parallel --interactive echo ::: 1 2 3
```

Output:

```
echo 1 ?...y
echo 2 ?...n
1
echo 3 ?...y
3
```

GNU Parallel can be used to put arguments on the command line for an interactive command such as emacs to edit one file at a time:

```
parallel --tty emacs ::: 1 2 3
```

Or give multiple argument in one go to open multiple files:

```
parallel -X --tty vi ::: 1 2 3
```

Timing

Some jobs do heavy I/O when they start. To avoid a thundering herd GNU Parallel can delay starting new jobs. --delay X will make sure there is at least X seconds between each start:

```
parallel --delay 2.5 echo Starting {}\;date ::: 1 2 3
```

Output:

```
Starting 1
Thu Aug 15 16:24:33 CEST 2013
Starting 2
Thu Aug 15 16:24:35 CEST 2013
Starting 3
Thu Aug 15 16:24:38 CEST 2013
```

If jobs taking more than a certain amount of time are known to fail, they can be stopped with --timeout:

```
parallel --timeout 2.1 sleep {}\; echo {} ::: 1 2 3 4
```

Output:

```
1
2
```

GNU Parallel can compute the median runtime for jobs and kill those that take more than 200% of the median runtime:

```
parallel --timeout 200% sleep {}\; echo {} ::: 2.1 2.2 3 7 2.3
```

Output:

```
2.1
2.2
3
2.3
```

Based on the runtime of completed jobs GNU Parallel can estimate the total runtime:

```
parallel --eta sleep ::: 1 3 2 2 1 3 3 2 1
```

Output:

```
Computers / CPU cores / Max jobs to run
1:local / 2 / 2

Computer:jobs running/jobs completed/%of started jobs/Average seconds to
complete
ETA: 2s 0left 1.11avg  local:0/9/100%/1.1s
```

Progress

GNU Parallel can give progress information with --progress:

```
parallel --progress sleep ::: 1 3 2 2 1 3 3 2 1
```

Output:

```
Computers / CPU cores / Max jobs to run
1:local / 2 / 2


Computer:jobs running/jobs completed/%of started jobs/Average seconds to
complete
local:0/9/100%/1.1s
```

A logfile of the jobs completed so far can be generated with --joblog:

```
parallel --joblog /tmp/log exit  ::: 1 2 3 0
cat /tmp/log
```

Output:

```
Seq      Host    Starttime        Runtime Send    Receive Exitval Signal
Command
 1       :       1376577364.974  0.008   0       0       1       0
exit 1
 2       :       1376577364.982  0.013   0       0       2       0
exit 2
 3       :       1376577364.990  0.013   0       0       3       0
exit 3
 4       :       1376577365.003  0.003   0       0       0       0
exit 0
```

The log contains the job sequence, which host the job was run on, the start time and run time, how much data was transferred if the job was run on a remote host, the exit value, the signal that killed the job, and finally the command being run.

With a joblog GNU Parallel can be stopped and later pickup where it left off. It it important that the input of the completed jobs is unchanged.

```
parallel --joblog /tmp/log exit  ::: 1 2 3 0
cat /tmp/log
parallel --resume --joblog /tmp/log exit  ::: 1 2 3 0 0 0
cat /tmp/log
```

Output:

```
Seq      Host    Starttime        Runtime Send    Receive Exitval Signal
Command
 1       :       1376580069.544  0.008   0       0       1       0
exit 1
 2       :       1376580069.552  0.009   0       0       2       0
exit 2
 3       :       1376580069.560  0.012   0       0       3       0
exit 3
 4       :       1376580069.571  0.005   0       0       0       0
exit 0


Seq      Host    Starttime        Runtime Send    Receive Exitval Signal
Command
 1       :       1376580069.544  0.008   0       0       1       0
```

```
exit 1
  2         :         1376580069.552   0.009   0      0      2      0
exit 2
  3         :         1376580069.560   0.012   0      0      3      0
exit 3
  4         :         1376580069.571   0.005   0      0      0      0
exit 0
  5         :         1376580070.028   0.009   0      0      0      0
exit 0
  6         :         1376580070.038   0.007   0      0      0      0
exit 0
```

Note how the start time of the last 2 jobs is clearly from the second run.

With --resume-failed GNU Parallel will re-run the jobs that failed:

```
parallel --resume-failed --joblog /tmp/log exit  ::: 1 2 3 0 0 0
cat /tmp/log
```

Output:

```
  Seq     Host     Starttime       Runtime Send   Receive Exitval Signal
Command
  1         :         1376580069.544   0.008   0      0      1      0
exit 1
  2         :         1376580069.552   0.009   0      0      2      0
exit 2
  3         :         1376580069.560   0.012   0      0      3      0
exit 3
  4         :         1376580069.571   0.005   0      0      0      0
exit 0
  5         :         1376580070.028   0.009   0      0      0      0
exit 0
  6         :         1376580070.038   0.007   0      0      0      0
exit 0
  1         :         1376580154.433   0.010   0      0      1      0
exit 1
  2         :         1376580154.444   0.022   0      0      2      0
exit 2
  3         :         1376580154.466   0.005   0      0      3      0
exit 3
```

Note how seq 1 2 3 have been repeated because they had exit value != 0.

Termination

For certain jobs there is no need to continue if one of the jobs fails and has an exit code != 0. GNU Parallel will stop spawning new jobs with --halt 1:

```
parallel -j2 --halt 1 echo {}\; exit {} ::: 0 0 1 2 3
```

Output:

```
  0
  0
  1
  parallel: Starting no more jobs. Waiting for 2 jobs to finish. This job
failed:
```

```
echo 1; exit 1
2
parallel: Starting no more jobs. Waiting for 1 jobs to finish. This job
failed:
echo 2; exit 2
```

With --halt 2 the running jobs will be killed immediately:

```
parallel -j2 --halt 2 echo {}\; exit {} ::: 0 0 1 2 3
```

Output:

```
0
0
1
parallel: This job failed:
echo 1; exit 1
```

GNU Parallel can retry the command with --retries. This is useful if a command fails for unkown reasons now and then.

```
parallel -k --retries 3 'echo tried {} >>/tmp/runs; echo completed {};
exit {}' ::: 1 2 0
cat /tmp/runs
```

Output:

```
completed 1
completed 2
completed 0

tried 1
tried 2
tried 1
tried 2
tried 1
tried 2
tried 0
```

Note how job 1 and 2 were tried 3 times, but 0 was not retried because it had exit code 0.

Limiting the ressources

To avoid overloading systems GNU Parallel can look at the system load before starting another job:

```
parallel --load 100% echo load is less than {} job per cpu ::: 1
```

Output:

```
[when then load is less than the number of cpu cores]
load is less than 1 job per cpu
```

GNU Parallel can also check if the system is swapping.

```
parallel --noswap echo the system is not swapping ::: now
```

Output:

```
[when then system is not swapping]
the system is not swapping now
```

GNU Parallel can run the jobs with a nice value. This will work both locally and remotely.

```
parallel --nice 17 echo this is being run with nice -n ::: 17
```

Output:

```
this is being run with nice -n 17
```

Remote execution

GNU Parallel can run jobs on remote servers. It uses ssh to communicate with the remote machines.

Sshlogin

The most basic sshlogin is -S host:

```
parallel -S $SERVER1 echo running on ::: $SERVER1
```

Output:

```
running on [$SERVER1]
```

To use a different username prepend the server with username@

```
parallel -S username@$SERVER1 echo running on ::: username@$SERVER1
```

Output:

```
running on [username@$SERVER1]
```

The special sshlogin ':' is the local machine:

```
parallel -S : echo running on ::: the_local_machine
```

Output:

```
running on the_local_machine
```

If ssh is not in $PATH it can be prepended to $SERVER1:

```
parallel -S '/usr/bin/ssh '$SERVER1 echo custom ::: ssh
```

Output:

```
custom ssh
```

Several servers can be given using multiple -S:

```
parallel -S $SERVER1 -S $SERVER2 echo ::: running on more hosts
```

Output (the order may be different):

```
running
on
more
hosts
```

Or they can be separated by ,:

```
parallel -S $SERVER1,$SERVER2 echo ::: running on more hosts
```

Output: Same as above.

The can also be read from a file (replace user@ with the user on $SERVER2):

```
echo $SERVER1 > nodefile
# Force 4 cores, special ssh-command, username
echo 4//usr/bin/ssh user@$SERVER2 >> nodefile
parallel --sshloginfile nodefile echo ::: running on more hosts
```

Output: Same as above.

The special --sshloginfile '..' reads from ~/.parallel/sshloginfile.

To force GNU Parallel to treat a server having a given number of CPU cores prepend #/ to the sshlogin:

```
parallel -S 4/$SERVER1 echo force {} cpus on server ::: 4
```

Output:

```
force 4 cpus on server
```

Transferring files

GNU Parallel can transfer the files to be processed to the remote host. It does that using rsync.

```
echo This is input_file > input_file
parallel -S $SERVER1 --transfer cat ::: input_file
```

Output:

```
This is input_file
```

If the files is processed into another file, the resulting file can be transferred back:

```
echo This is input_file > input_file
parallel -S $SERVER1 --transfer --return {}.out cat {} ">"{}.out :::
input_file
cat input_file.out
```

Output: Same as above.

To remove the input and output file on the remote server use --cleanup:

```
echo This is input_file > input_file
parallel -S $SERVER1 --transfer --return {}.out --cleanup cat {}
">"{}.out ::: input_file
cat input_file.out
```

Output: Same as above.

There is a short hand for --transfer --return --cleanup called --trc:

```
echo This is input_file > input_file
parallel -S $SERVER1 --trc {}.out cat {} ">"{}.out ::: input_file
cat input_file.out
```

Output: Same as above.

Some jobs need a common database for all jobs. GNU Parallel can transfer that using --basefile which will transfer the file before the first job:

```
echo common data > common_file
parallel --basefile common_file -S $SERVER1 cat common_file\; echo {} :::
foo
```

Output:

```
common data
foo
```

To remove it from the remote host after the last job use --cleanup.

Working dir

The default working dir on the remote machines is the login dir. This can be changed with --workdir *mydir*.

Files transferred using --transfer and --return will be relative to *mydir* on remote computers, and the command will be executed in the dir *mydir*.

The special *mydir* value ... will create working dirs under ~/.parallel/tmp/ on the remote computers. If --cleanup is given these dirs will be removed.

The special *mydir* value . uses the current working dir. If the current working dir is beneath your home dir, the value . is treated as the relative path to your home dir. This means that if your home dir is different on remote computers (e.g. if your login is different) the relative path will still be relative to your home dir.

```
parallel -S $SERVER1 pwd ::: ""
parallel --workdir . -S $SERVER1 pwd ::: ""
parallel --workdir ... -S $SERVER1 pwd ::: ""
```

Output:

```
[the login dir on $SERVER1]
[current dir relative on $SERVER1]
[a dir in ~/.parallel/tmp/...]
```

Avoid overloading sshd

If many jobs are started on the same server, sshd can be overloaded. GNU Parallel can insert a delay between each job run on the same server:

```
parallel -S $SERVER1 --sshdelay 0.2 echo ::: 1 2 3
```

Output (the order may be different):

```
1
2
3
```

Sshd will be less overloaded if using --controlmaster, which will multiplex ssh connections:

```
parallel --controlmaster -S $SERVER1 echo ::: 1 2 3
```

Output: Same as above.

Ignore hosts that are down

In clusters with many hosts a few of the are often down. GNU Parallel can ignore those hosts. In this case the host 173.194.32.46 is down:

```
parallel --filter-hosts -S 173.194.32.46,$SERVER1 echo ::: bar
```

Output:

```
bar
```

Running the same commands on all hosts

GNU Parallel can run the same command on all the hosts:

```
parallel --onall -S $SERVER1,$SERVER2 echo ::: foo bar
```

Output (the order may be different):

```
foo
bar
foo
bar
```

Often you will just want to run a single command on all hosts with out arguments. --nonall is a no argument --onall:

```
parallel --nonall -S $SERVER1,$SERVER2 echo foo bar
```

Output:

```
foo bar
foo bar
```

When --tag is used with --nonall and --onall the --tagstring is the host:

```
parallel --nonall --tag -S $SERVER1,$SERVER2 echo foo bar
```

Output (the order may be different):

```
$SERVER1 foo bar
$SERVER2 foo bar
```

--jobs sets the number of servers to log in to in parallel.

Transfer environment variables and functions

Using --env GNU Parallel can transfer an environment variable to the remote system.

```
MYVAR='foo bar'
export MYVAR
parallel --env MYVAR -S $SERVER1 echo '$MYVAR' ::: baz
```

Output:

```
foo bar baz
```

This works for functions too if your shell is Bash:

```
# This only works in Bash
```

```
  my_func() {
    echo in my_func $1
  }
  export -f my_func
  parallel --env my_func -S $SERVER1 my_func ::: baz
```

Output:

```
  in my_func baz
```

GNU Parallel can copy all defined variables and functions to the remote system. It just needs to record which ones to ignore in ~/.parallel/ignored_vars. Do that by running this once:

```
  parallel --record-env
  cat ~/.parallel/ignored_vars
```

Output:

```
  [list of variables to ignore - including $PATH and $HOME]
```

Now all new variables and functions defined will be copied when using --env _:

```
  # The function is only copied if using Bash
  my_func2() {
    echo in my_func2 $VAR $1
  }
  export -f my_func2
  VAR=foo
  export VAR


  parallel --env _ -S $SERVER1 'echo $VAR; my_func2' ::: bar
```

Output:

```
  foo
  in my_func2 foo bar
```

Showing what is actually run

--verbose will show the command that would be run on the local machine. When a job is run on a remote machine this is wrapped with ssh and possibly transferring files and environment variables, setting the workdir, and setting --nice value. -vv shows all of this.

```
  parallel -vv -S $SERVER1 echo ::: bar
```

Output:

```
  ssh -tt -oLogLevel=quiet lo  'eval `echo $SHELL | grep "/t\{0,1\}csh" >
/dev/null  && echo setenv PARALLEL_SEQ '$PARALLEL_SEQ'\;  setenv
PARALLEL_PID '$PARALLEL_PID'  || echo PARALLEL_SEQ='$PARALLEL_SEQ'\;export
PARALLEL_SEQ\;  PARALLEL_PID='$PARALLEL_PID'\;export PARALLEL_PID` ;'  tty\
 \>/dev/null\ \&\&\ stty\ isig\ -onlcr\ -echo\;echo\ bar;
  bar
```

When the command gets more complex, the output is so hard to read, that it is only useful for debugging:

```
  my_func3() {
    echo in my_func $1 > $1.out
  }
  export -f my_func3
  parallel -vv --workdir ... --nice 17 --env _ --trc {}.out -S $SERVER1
my_func3 {} ::: abc-file
```

Output will be similar to:

```
  ssh server mkdir -p .parallel/tmp/hk-31483-1; rsync -rlDzR -essh
./abc-file server:.parallel/tmp/hk-31483-1;ssh -tt -oLogLevel=quiet server
  'eval `echo $SHELL | grep "/t\{0,1\}csh" > /dev/null  && echo setenv
PARALLEL_SEQ '$PARALLEL_SEQ'\;  setenv PARALLEL_PID '$PARALLEL_PID'  ||
echo PARALLEL_SEQ='$PARALLEL_SEQ'\;export PARALLEL_SEQ\;
PARALLEL_PID='$PARALLEL_PID'\;export PARALLEL_PID` ;'  tty\ \>/dev/null\
\&\&\ stty\ isig\ -onlcr\ -echo\;mkdir\ -p\ .parallel/tmp/hk-31483-1\;\ cd\
 .parallel/tmp/hk-31483-1\ \&\&\ echo\ \$SHELL\ \|\ grep\
\"/t\\\{0,1\\\}csh\"\ \>\ /dev/null\ \&\&\ setenv\ my_func3\ \\\(\\\)\\\
\\\{\\\ \\\ echo\\\ in\\\ my_func\\\ \\\$1\\\ \\\>\\\ \\\$1.out\"'
  '\"\\\}\ \&\&\ setenv\ VAR\ foo\ \&\&\ setenv\ my_func2\ \\\(\\\)\\\
\\\{\\\ \\\ echo\\\ in\\\ my_func2\\\ \\\$VAR\\\ \\\$1\"'
  '\"\\\}\ \|\|\ export\ my_func3=\\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\
my_func\\\ \\\$1\\\ \\\>\\\ \\\$1.out\"'
  '\"\\\}\ \&\&\ export\ VAR=foo\ \&\&\ export\ my_func2=\\\(\\\)\\\
\\\{\\\ \\\ echo\\\ in\\\ my_func2\\\ \\\$VAR\\\ \\\$1\"'
  '\"\\\}\ \&\&\ eval\ my_func3\"\$my_func3\"\ \&\&\ eval\
my_func2\"\$my_func2\"\;\\nice\ -n17\ /bin/bash\ -c\ my_func3\\\
abc-file;_EXIT_status=$?; mkdir -p .; rsync --rsync-path=cd\
.parallel/tmp/hk-31483-1/.\;\ rsync -rlDzR -essh server:abc-file.out .;ssh
server rm\ -f\ .parallel/tmp/hk-31483-1/abc-file\;rm\ -f\
.parallel/tmp/hk-31483-1/abc-file.out\;rm -rf .parallel/tmp/hk-31483-1\;;
exit $_EXIT_status;
```

--pipe

The --pipe functionality puts GNU Parallel in a different mode: Instead of treating the data on stdin (standard input) as arguments for a command to run, the data will be sent to stdin (standard input) of the command.

The typical situation is:

```
  command_A | command_B | command_C
```

where command_B is slow, and you want to speed up command_B.

Chunk size

By default GNU Parallel will start an instance of command_B, read a chunk of 1 MB, and pass that to the instance. Then start another instance, read another chunk, and pass that to the second instance.

```
  cat num1000000 | parallel --pipe wc
```

Output (the order may be different):

```
  165668   165668 1048571
  149797   149797 1048579
  149796   149796 1048572
  149797   149797 1048579
  149797   149797 1048579
```

```
149796   149796  1048572
 85349    85349   597444
```

The size of the chunk is not exactly 1 MB because GNU Parallel only passes full lines - never half a line, thus the blocksize is only average 1 MB. You can change the block size to 2 MB with --block:

```
cat num1000000 | parallel --pipe --block 2M wc
```

Output (the order may be different):

```
315465   315465 2097150
299593   299593 2097151
299593   299593 2097151
 85349    85349  597444
```

GNU Parallel treats each line as a record. If the order of record is unimportant (e.g. you need all lines processed, but you do not care which is processed first), then you can use --round-robin. Without --round-robin GNU Parallel will start a command per block; with --round-robin only the requested number of jobs will be started (--jobs). The records will then be distributed between the running jobs:

```
cat num1000000 | parallel --pipe -j4 --round-robin wc
```

Output will be similar to:

```
149797   149797 1048579
299593   299593 2097151
315465   315465 2097150
235145   235145 1646016
```

One of the 4 instances got a single record, 2 instances got 2 full records each, and one instance got 1 full and 1 partial record.

Records

GNU Parallel sees the input as records. The default record is a single line.

Using -N140000 GNU Parallel will read 140000 records at a time:

```
cat num1000000 | parallel --pipe -N140000 wc
```

Output (the order may be different):

```
140000   140000   868895
140000   140000   980000
140000   140000   980000
140000   140000   980000
140000   140000   980000
140000   140000   980000
140000   140000   980000
 20000    20000   140001
```

Notice that the last job could not get the full 140000 lines, but only 20000 lines.

If a record is 75 lines -L can be used:

```
cat num1000000 | parallel --pipe -L75 wc
```

Output (the order may be different):

```
165600   165600 1048095
149850   149850 1048950
149775   149775 1048425
149775   149775 1048425
149850   149850 1048950
149775   149775 1048425
 85350    85350  597450
    25       25      176
```

Notice GNU Parallel still reads a block of around 1 MB; but instead of passing full lines to 'wc' it passes full 75 lines at a time. This of course does not hold for the last job (which in this case got 25 lines).

Record separators

GNU Parallel uses separators to determine where two record split.

--recstart gives the string that starts a record; --recend gives the string that ends a record. The default is --recend '\n' (newline).

If both --recend and --recstart are given, then the record will only split if the recend string is immediately followed by the recstart string.

Here the --recend is set to ', ':

```
  echo /foo, bar/, /baz, qux/, | parallel -kN1 --recend ', ' --pipe echo
JOB{#}\;cat\;echo END
```

Output:

```
  JOB1
  /foo, END
  JOB2
  bar/, END
  JOB3
  /baz, END
  JOB4
  qux/,
  END
```

Here the --recstart is set to '/':

```
  echo /foo, bar/, /baz, qux/, | parallel -kN1 --recstart '/' --pipe echo
JOB{#}\;cat\;echo END
```

Output:

```
  JOB1
  /foo, barEND
  JOB2
  /, END
  JOB3
  /baz, quxEND
  JOB4
  /,
  END
```

Here both --recend and --recstart are set:

```
echo /foo, bar/, /baz, qux/, | parallel -kN1 --recend ', ' --recstart '/'
--pipe echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
/foo, bar/, END
JOB2
/baz, qux/,
END
```

Note the difference between setting one string and setting both strings.

With --regexp the --recend and --recstart will be treated as a regular expression:

```
echo foo,bar,_baz,__qux, | parallel -kN1 --regexp --recend ',_+' --pipe
echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
foo,bar,_END
JOB2
baz,__END
JOB3
qux,
END
```

GNU Parallel can remove the record separators with --remove-rec-sep/--rrs:

```
echo foo,bar,_baz,__qux, | parallel -kN1 --rrs --regexp --recend ',_+'
--pipe echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
foo,barEND
JOB2
bazEND
JOB3
qux,
END
```

Header

If the input data has a header, the header can be repeated for each job by matching the header with --header. If headers start with %:

```
cat num_%header | parallel --header '(%.*\n)*' --pipe -N3 echo
JOB{#}\;cat
```

Output (the order may be different):

```
JOB1
%head1
%head2
1
2
```

```
3
JOB2
%head1
%head2
4
5
6
JOB3
%head1
%head2
7
8
9
JOB4
%head1
%head2
10
```

If the header is 2 lines, --header 2 will work:

```
cat num_%header | parallel --header 2 --pipe -N3 echo JOB{#}\;cat
```

Output: Same as above.

Shebang

Input data and parallel command in the same file

GNU Parallel is often called as:

```
cat input_file | parallel command
```

With --shebang the input_file and parallel can be combined into the same script.

UNIX-scripts start with a shebang line like:

```
#!/bin/bash
```

GNU Parallel can do that, too. With --shebang the arguments can be listed in the file. The parallel command is the first line of the script:

```
#!/usr/bin/parallel --shebang -r echo

foo
bar
baz
```

Output (the order may be different):

```
foo
bar
baz
```

Parallelizing existing scripts

GNU Parallel is often called as:

```
cat input_file | parallel command
parallel command ::: foo bar
```

If command is a script parallel can be combined into a single file so:

```
cat input_file | command
command foo bar
```

will run the script in parallel.

This perl script perl_echo works like echo:

```
#!/usr/bin/perl


print "@ARGV\n"
```

It can be called as:

```
parallel perl_echo ::: foo bar
```

By changing the #!-line it can be run in parallel

```
#!/usr/bin/parallel --shebang-wrap /usr/bin/perl


print "@ARGV\n"
```

Thus this will work:

```
perl_echo foo bar
```

Output (the order may be different):

```
foo
bar
```

This technique can be used for:

Perl:

        #!/usr/bin/parallel --shebang-wrap /usr/bin/perl

Python:

        #!/usr/bin/parallel --shebang-wrap /usr/bin/python

Bash:

        #!/usr/bin/parallel --shebang-wrap /bin/bash

R:

        #!/usr/bin/parallel --shebang-wrap /usr/bin/Rscript --vanilla --slave

GNUplot:

        #!/usr/bin/parallel --shebang-wrap ARG={} /usr/bin/gnuplot

Ruby:

        #!/usr/bin/parallel --shebang-wrap /usr/bin/ruby

Semaphore

GNU Parallel can work as a counting semaphore. This is slower and less efficient than its normal mode.

An alias for 'parallel --semaphore' is 'sem'. The default is to allow only one program to run at a time

(technically called a mutex). The program is started in the background. Use --wait for all 'sem's to finish:

```
sem 'sleep 1; echo The first finished' &&
  echo The first is now running in the background &&
  sem 'sleep 1; echo The second finished' &&
  echo The second is now running in the background
sem --wait
```

Output:

```
The first is now running in the background
The first finished
The second is now running in the background
The second finished
```

The command can be run in the foreground with --fg:

```
sem --fg 'sleep 1; echo The first finished' &&
  echo The first finished running in the foreground &&
  sem --fg 'sleep 1; echo The second finished' &&
  echo The second finished running in the foreground
sem --wait
```

The difference between this and just running the command, is that a mutex is set, so if other sems were running in the background only one would run at the same time.

To tell the difference between which semaphore is used, use --semaphorename/--id. Run this in one terminal:

```
sem --id my_id -u 'echo First started; sleep 10; echo The first finished'
```

and simultaneously this in another terminal:

```
sem --id my_id -u 'echo Second started; sleep 10; echo The second finished'
```

Note how the second will only be started when the first has finished.

Counting semaphore

A mutex is like having a single toilet: When it is in use everyone else will have to wait. A counting semaphore is like having multiple toilets: Several people can use the toilets, but when they all are in use, everyone else will have to wait.

sem can emulate a counting semaphore. Use --jobs to set the number of toilets:

```
sem --jobs 3 --id my_id -u 'echo First started; sleep 5; echo The first finished' &&
sem --jobs 3 --id my_id -u 'echo Second started; sleep 6; echo The second finished' &&
sem --jobs 3 --id my_id -u 'echo Third started; sleep 7; echo The third finished' &&
sem --jobs 3 --id my_id -u 'echo Fourth started; sleep 8; echo The fourth finished' &&
sem --wait --id my_id
```

Output:

```
First started
Second started
Third started
The first finished
Fourth started
The second finished
The third finished
The fourth finished
```

Informational

GNU Parallel has some options to give short information about the configuration.

--help will print a summary of the most important options:

```
parallel --help
```

Output:

```
Usage:
parallel [options] [command [arguments]] < list_of_arguments
parallel [options] [command [arguments]] (::: arguments|::::
argfile(s))...
cat ... | parallel --pipe [options] [command [arguments]]


-j n            Run n jobs in parallel
-k              Keep same order
-X              Multiple arguments with context replace
--colsep regexp     Split input on regexp for positional replacements
{} {.} {/} {/.} {#}  Replacement strings
{3} {3.} {3/} {3/.}  Positional replacement strings


-S sshlogin     Example: foo@server.example.com
--slf ..        Use ~/.parallel/sshloginfile as the list of sshlogins
--trc {}.bar    Shorthand for --transfer --return {}.bar --cleanup
--onall         Run the given command with argument on all sshlogins
--nonall        Run the given command with no arguments on all sshlogins


--pipe          Split stdin (standard input) to multiple jobs.
--recend str    Record end separator for --pipe.
--recstart str  Record start separator for --pipe.


See 'man parallel' for details


When using GNU Parallel for a publication please cite:


O. Tange (2011): GNU Parallel - The Command-Line Power Tool,
;login: The USENIX Magazine, February 2011:42-47.
```

When asking for help, always report the full output of:

```
parallel --version
```

Output:

```
GNU parallel 20130822
```

```
  Copyright (C) 2007,2008,2009,2010,2011,2012,2013 Ole Tange and Free
Software Foundation, Inc.
  License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
  This is free software: you are free to change and redistribute it.
  GNU parallel comes with no warranty.


  Web site: http://www.gnu.org/software/parallel


  When using GNU Parallel for a publication please cite:


  O. Tange (2011): GNU Parallel - The Command-Line Power Tool,
  ;login: The USENIX Magazine, February 2011:42-47.
```

In scripts --minversion can be used to ensure the user has at least this version:

```
  parallel --minversion 20130722 && echo Your version is at least 20130722.
```

Output:

```
  20130722
  Your version is at least 20130722.
```

If using GNU Parallel for research the BibTeX citation can be generated using --bibtex.

```
  parallel --bibtex
```

Output:

```
  @article{Tange2011a,
   title = {GNU Parallel - The Command-Line Power Tool},
   author = {O. Tange},
   address = {Frederiksberg, Denmark},
   journal = {;login: The USENIX Magazine},
   month = {Feb},
   number = {1},
   volume = {36},
   url = {http://www.gnu.org/s/parallel},
   year = {2011},
   pages = {42-47}
  }
```

With --max-line-length-allowed GNU Parallel will report the maximal size of the command line:

```
  parallel --max-line-length-allowed
```

Output (may vary on different systems):

```
  131071
```

--number-of-cpus and --number-of-cores run system specific code to determine the number of CPUs and CPU cores on the system. On unsupported platforms they will return 1:

```
  parallel --number-of-cpus
  parallel --number-of-cores
```

Output (may vary on different systems):

```
4
64
```

Profiles

The defaults for GNU Parallel can be changed systemwise by putting the command line options in /etc/parallel/config. They can be changed for a user by putting them in ~/.parallel/config.

Profiles work the same way, but have to be referred to with --profile:

```
echo '-S :,'$SERVER1 > ~/.parallel/cluster
echo '--nice 17' >> ~/.parallel/cluster
echo '--filter-hosts' >> ~/.parallel/cluster
echo '--timeout 300%' >> ~/.parallel/cluster
echo '--env _' >> ~/.parallel/cluster
parallel --profile cluster echo ::: A B C
```

Output:

```
A
B
C
```

Profiles can be combined:

```
echo '-vv --dry-run' > ~/.parallel/dryverbose
parallel --profile dryverbose --profile cluster echo ::: A B C
```

Output:

```
  ssh -tt -oLogLevel=quiet lo  'eval `echo $SHELL | grep "/t\{0,1\}csh" >
/dev/null  && echo setenv PARALLEL_SEQ '$PARALLEL_SEQ'\;  setenv
PARALLEL_PID '$PARALLEL_PID'  || echo PARALLEL_SEQ='$PARALLEL_SEQ'\;export
PARALLEL_SEQ\;  PARALLEL_PID='$PARALLEL_PID'\;export PARALLEL_PID` ;'  tty\
 \>/dev/null\ \&\&\ stty\ isig\ -onlcr\ -echo\;echo\ \$SHELL\ \|\ grep\
\"/t\\\{0,1\\\}csh\"\ \>\ /dev/null\ \&\&\ setenv\ SERVER1\ lo\ \&\&\
setenv\ MYVAR\ foo\\\ bar\ \&\&\ setenv\ VAR\ foo\ \&\&\ setenv\ my_func\
\\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\ my_func\\\ \\\$1\"'
  '\"\\\}\ \&\&\ setenv\ my_func2\ \\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\
my_func2\\\ \\\$VAR\\\ \\\$1\"'
  '\"\\\}\ \|\|\ export\ SERVER1=lo\ \&\&\ export\ MYVAR=foo\\\ bar\ \&\&\
export\ VAR=foo\ \&\&\ export\ my_func=\\\(\\\)\\\ \\\{\\\ \\\ echo\\\
in\\\ my_func\\\ \\\$1\"'
  '\"\\\}\ \&\&\ export\ my_func2=\\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\
my_func2\\\ \\\$VAR\\\ \\\$1\"'
  '\"\\\}\ \&\&\ eval\ my_func\"\$my_func\"\ \&\&\ eval\
my_func2\"\$my_func2\"\;\\nice\ -n17\ /bin/bash\ -c\ echo\\\ A;
  ssh -tt -oLogLevel=quiet lo  'eval `echo $SHELL | grep "/t\{0,1\}csh" >
/dev/null  && echo setenv PARALLEL_SEQ '$PARALLEL_SEQ'\;  setenv
PARALLEL_PID '$PARALLEL_PID'  || echo PARALLEL_SEQ='$PARALLEL_SEQ'\;export
PARALLEL_SEQ\;  PARALLEL_PID='$PARALLEL_PID'\;export PARALLEL_PID` ;'  tty\
 \>/dev/null\ \&\&\ stty\ isig\ -onlcr\ -echo\;echo\ \$SHELL\ \|\ grep\
\"/t\\\{0,1\\\}csh\"\ \>\ /dev/null\ \&\&\ setenv\ SERVER1\ lo\ \&\&\
setenv\ MYVAR\ foo\\\ bar\ \&\&\ setenv\ VAR\ foo\ \&\&\ setenv\ my_func\
\\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\ my_func\\\ \\\$1\"'
  '\"\\\}\ \&\&\ setenv\ my_func2\ \\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\
```

```
my_func2\\\ \\\$VAR\\\ \\\$1\"'
   '\"\\\}\ \|\|\ export\ SERVER1=lo\ \&\&\ export\ MYVAR=foo\\\ bar\ \&\&\
export\ VAR=foo\ \&\&\ export\ my_func=\\\(\\\)\\\ \\\{\\\ \\\ echo\\\
in\\\ my_func\\\ \\\$1\"'
   '\"\\\}\ \&\&\ export\ my_func2=\\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\
my_func2\\\ \\\$VAR\\\ \\\$1\"'
   '\"\\\}\ \&\&\ eval\ my_func\"\$my_func\"\ \&\&\ eval\
my_func2\"\$my_func2\"\;\\nice\ -n17\ /bin/bash\ -c\ echo\\\ B;
   ssh -tt -oLogLevel=quiet lo  'eval `echo $SHELL | grep "/t\{0,1\}csh" >
/dev/null  && echo setenv PARALLEL_SEQ '$PARALLEL_SEQ'\;  setenv
PARALLEL_PID '$PARALLEL_PID'  || echo PARALLEL_SEQ='$PARALLEL_SEQ'\;export
PARALLEL_SEQ\;  PARALLEL_PID='$PARALLEL_PID'\;export PARALLEL_PID` ;'  tty\
 \>/dev/null\ \&\&\ stty\ isig\ -onlcr\ -echo\;echo\ \$SHELL\ \|\ grep\
\"/t\\\{0,1\\\}csh\"\ \>\ /dev/null\ \&\&\ setenv\ SERVER1\ lo\ \&\&\
setenv\ MYVAR\ foo\\\ bar\ \&\&\ setenv\ VAR\ foo\ \&\&\ setenv\ my_func\
\\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\ my_func\\\ \\\$1\"'
   '\"\\\}\ \&\&\ setenv\ my_func2\ \\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\
my_func2\\\ \\\$VAR\\\ \\\$1\"'
   '\"\\\}\ \|\|\ export\ SERVER1=lo\ \&\&\ export\ MYVAR=foo\\\ bar\ \&\&\
export\ VAR=foo\ \&\&\ export\ my_func=\\\(\\\)\\\ \\\{\\\ \\\ echo\\\
in\\\ my_func\\\ \\\$1\"'
   '\"\\\}\ \&\&\ export\ my_func2=\\\(\\\)\\\ \\\{\\\ \\\ echo\\\ in\\\
my_func2\\\ \\\$VAR\\\ \\\$1\"'
   '\"\\\}\ \&\&\ eval\ my_func\"\$my_func\"\ \&\&\ eval\
my_func2\"\$my_func2\"\;\\nice\ -n17\ /bin/bash\ -c\ echo\\\ C;
```

Spread the word

I hope you have learned something from this tutorial.

If you like GNU Parallel:

● Give a demo at your local user group/team/colleagues

● Post the intro videos and the tutorial on Reddit, Diaspora*, forums, blogs, Identi.ca, Google+, Twitter, Facebook, Linkedin, mailing lists

● Request or write a review for your favourite blog or magazine

● Invite me for your next conference

If you use GNU Parallel for research:

● Please cite GNU Parallel in you publications (use --bibtex)

If GNU Parallel saves you money:

● (Have your company) donate to FSF https://my.fsf.org/donate/