# NAME

parallel_alternatives - Alternatives to GNU **parallel**

# DIFFERENCES BETWEEN GNU Parallel AND ALTERNATIVES

There are a lot programs that share functionality with GNU **parallel**. Some of these are specialized tools, and while GNU **parallel** can emulate many of them, a specialized tool can be better at a given task. GNU **parallel** strives to include the best of the general functionality without sacrificing ease of use.

**parallel** has existed since 2002-01-06 and as GNU **parallel** since 2010. A lot of the alternatives have not had the vitality to survive that long, but have come and gone during that time.

GNU **parallel** is actively maintained with a new release every month since 2010. Most other alternatives are fleeting interests of the developers with irregular releases and only maintained for a few years.

## SUMMARY LEGEND

The following features are in some of the comparable tools:

### Inputs

I1. Arguments can be read from stdin

I2. Arguments can be read from a file

I3. Arguments can be read from multiple files

I4. Arguments can be read from command line

I5. Arguments can be read from a table

I6. Arguments can be read from the same file using #! (shebang)

I7. Line oriented input as default (Quoting of special chars not needed)

### Manipulation of input

M1. Composed command

M2. Multiple arguments can fill up an execution line

M3. Arguments can be put anywhere in the execution line

M4. Multiple arguments can be put anywhere in the execution line

M5. Arguments can be replaced with context

M6. Input can be treated as the complete command line

### Outputs

O1. Grouping output so output from different jobs do not mix

O2. Send stderr (standard error) to stderr (standard error)

O3. Send stdout (standard output) to stdout (standard output)

O4. Order of output can be same as order of input

O5. Stdout only contains stdout (standard output) from the command

O6. Stderr only contains stderr (standard error) from the command

O7. Buffering on disk

O8. No temporary files left if killed

O9. Test if disk runs full during run

O10. Output of a line bigger than 4 GB

### Execution

E1. Run jobs in parallel

E2. List running jobs

E3. Finish running jobs, but do not start new jobs

E4. Number of running jobs can depend on number of cpus

E5. Finish running jobs, but do not start new jobs after first failure

E6. Number of running jobs can be adjusted while running

E7. Only spawn new jobs if load is less than a limit

E8. Full command has non-zero exit value if one job has non-zero exit value

E9. Jobs can be started without reading all input first

**Remote execution**

R1. Jobs can be run on remote computers

R2. Basefiles can be transferred

R3. Argument files can be transferred

R4. Result files can be transferred

R5. Cleanup of transferred files

R6. No config files needed

R7. Do not run more than SSHD's MaxStartups can handle

R8. Configurable SSH command

R9. Retry if connection breaks occasionally

**Semaphore**

S1. Possibility to work as a mutex

S2. Possibility to work as a counting semaphore

**Legend**

- = no

x = not applicable

ID = yes

Since each new version of the programs is not tested, the table may be outdated. Please file a bug report if you find errors (See REPORTING BUGS).

## GNU Parallel

Summary (see legend above):

I1 I2 I3 I4 I5 I6 I7

M1 M2 M3 M4 M5 M6

O1 O2 O3 O4 O5 O6 O7 O8 O9 O10

E1 E2 E3 E4 E5 E6 E7 E8 E9

R1 R2 R3 R4 R5 R6 R7 R8 R9

S1 S2

## DIFFERENCES BETWEEN xargs AND GNU Parallel

Summary (see legend above):

I1 I2 - - - - -

- M2 M3 - - -

- O2 O3 - O5 O6

E1 - - - - - - E8 E9

```
-----x---
--
```

**xargs** offers some of the same possibilities as GNU **parallel**.

**xargs** deals badly with special characters (such as space, \, ' and "). To see the problem try this:

```
touch important_file
touch 'not important_file'
ls not* | xargs rm
mkdir -p "My brother's 12\" records"
ls | xargs rmdir
touch 'c:\windows\system32\clfs.sys'
echo 'c:\windows\system32\clfs.sys' | xargs ls -l
```

You can specify **-0**, but many input generators are not optimized for using **NUL** as separator but are optimized for **newline** as separator. E.g. **awk**, **ls**, **echo**, **tar -v**, **head** (requires using **-z**), **tail** (requires using **-z**), **sed** (requires using **-z**), **perl** (**-0** and \0 instead of \n), **locate** (requires using **-0**), **find** (requires using **-print0**), **grep** (requires using **-z** or **-Z**), **sort** (requires using **-z**).

GNU **parallel**'s newline separation can be emulated with:

```
cat | xargs -d "\n" -n1 command
```

**xargs** can run a given number of jobs in parallel, but has no support for running number-of-cpu-cores jobs in parallel.

**xargs** has no support for grouping the output, therefore output may run together, e.g. the first half of a line is from one process and the last half of the line is from another process. The example **Parallel grep** cannot be done reliably with **xargs** because of this. To see this in action try:

```
parallel perl -e "'"'$a="1"."{}"x100000000;print $a,"\n"'"'" \
  '>' {} ::: a b c d e f g h
# Serial = no mixing = the wanted result
# 'tr -s a-z' squeezes repeating letters into a single letter
echo a b c d e f g h | xargs -P1 -n1 grep 1 | tr -s a-z
# Compare to 8 jobs in parallel
parallel -kP8 -n1 grep 1 ::: a b c d e f g h | tr -s a-z
echo a b c d e f g h | xargs -P8 -n1 grep 1 | tr -s a-z
echo a b c d e f g h | xargs -P8 -n1 grep --line-buffered 1 | \
  tr -s a-z
```

Or try this:

```
slow_seq() {
  echo Count to "$@"
  seq "$@" |
    perl -ne '$|=1; for(split//){ print; select($a,$a,$a,0.100);}'
}
export -f slow_seq
# Serial = no mixing = the wanted result
seq 8 | xargs -n1 -P1 -I {} bash -c 'slow_seq {}'
# Compare to 8 jobs in parallel
seq 8 | parallel -P8 slow_seq {}
seq 8 | xargs -n1 -P8 -I {} bash -c 'slow_seq {}'
```

**xargs** has no support for keeping the order of the output, therefore if running jobs in parallel using **xargs** the output of the second job cannot be postponed till the first job is done.

**xargs** has no support for running jobs on remote computers.

**xargs** has no support for context replace, so you will have to create the arguments.

If you use a replace string in **xargs** (**-I**) you can not force **xargs** to use more than one argument.

Quoting in **xargs** works like **-q** in GNU **parallel**. This means composed commands and redirection require using **bash -c**.

```
ls | parallel "wc {} >{}.wc"
ls | parallel "echo {}; ls {}|wc"
```

becomes (assuming you have 8 cores and that none of the filenames contain space, " or ').

```
ls | xargs -d "\n" -P8 -I {} bash -c "wc {} >{}.wc"
ls | xargs -d "\n" -P8 -I {} bash -c "echo {}; ls {}|wc"
```

A more extreme example can be found on: https://unix.stackexchange.com/q/405552/

https://www.gnu.org/software/findutils/

## DIFFERENCES BETWEEN find -exec AND GNU Parallel

Summary (see legend above):

- - - x - x -

- M2 M3 - - - -

- O2 O3 O4 O5 O6

- - - - - - -

- - - - - - - - -

x x

**find -exec** offers some of the same possibilities as GNU **parallel**.

**find -exec** only works on files. Processing other input (such as hosts or URLs) will require creating these inputs as files. **find -exec** has no support for running commands in parallel.

https://www.gnu.org/software/findutils/ (Last checked: 2019-01)

## DIFFERENCES BETWEEN make -j AND GNU Parallel

Summary (see legend above):

- - - - - - -

- - - - - -

O1 O2 O3 - x O6

E1 - - - E5 -

- - - - - - - - -

- -

**make -j** can run jobs in parallel, but requires a crafted Makefile to do this. That results in extra quoting to get filenames containing newlines to work correctly.

**make -j** computes a dependency graph before running jobs. Jobs run by GNU **parallel** does not depend on each other.

(Very early versions of GNU **parallel** were coincidentally implemented using **make -j**).

https://www.gnu.org/software/make/ (Last checked: 2019-01)

## DIFFERENCES BETWEEN ppss AND GNU Parallel

Summary (see legend above):

I1 I2 - - - - I7

M1 - M3 - - M6

O1 - - x - -

E1 E2 ?E3 E4 - - -

R1 R2 R3 R4 - - ?R7 ? ?

- -

**ppss** is a tool for running jobs in parallel.

The output of **ppss** is status information and thus not useful for using as input for another command. The output from the jobs are put into files.

The argument replace string ($ITEM) cannot be changed. Arguments must be quoted - thus arguments containing special characters (space '"&!*) may cause problems. More than one argument is not supported. Filenames containing newlines are not processed correctly. When reading input from a file null cannot be used as a terminator. **ppss** needs to read the whole input file before starting any jobs.

Output and status information is stored in ppss_dir and thus requires cleanup when completed. If the dir is not removed before running **ppss** again it may cause nothing to happen as **ppss** thinks the task is already done. GNU **parallel** will normally not need cleaning up if running locally and will only need cleaning up if stopped abnormally and running remote (**--cleanup** may not complete if stopped abnormally). The example **Parallel grep** would require extra postprocessing if written using **ppss**.

For remote systems PPSS requires 3 steps: config, deploy, and start. GNU **parallel** only requires one step.

### EXAMPLES FROM ppss MANUAL

Here are the examples from **ppss**'s manual page with the equivalent using GNU **parallel**:

```
1$ ./ppss.sh standalone -d /path/to/files -c 'gzip '

1$ find /path/to/files -type f | parallel gzip

2$ ./ppss.sh standalone -d /path/to/files \
     -c 'cp "$ITEM" /destination/dir '

2$ find /path/to/files -type f | parallel cp {} /destination/dir

3$ ./ppss.sh standalone -f list-of-urls.txt -c 'wget -q '

3$ parallel -a list-of-urls.txt wget -q

4$ ./ppss.sh standalone -f list-of-urls.txt -c 'wget -q "$ITEM"'

4$ parallel -a list-of-urls.txt wget -q {}

5$ ./ppss config -C config.cfg -c 'encode.sh ' -d /source/dir \
     -m 192.168.1.100 -u ppss -k ppss-key.key -S ./encode.sh \
     -n nodes.txt -o /some/output/dir --upload --download;
   ./ppss deploy -C config.cfg
   ./ppss start -C config
```

```
5$ # parallel does not use configs. If you want
   # a different username put it in nodes.txt: user@hostname
   find source/dir -type f |
     parallel --sshloginfile nodes.txt --trc {.}.mp3 \
       lame -a {} -o {.}.mp3 --preset standard --quiet

6$ ./ppss stop -C config.cfg

6$ killall -TERM parallel

7$ ./ppss pause -C config.cfg

7$ Press: CTRL-Z or killall -SIGTSTP parallel

8$ ./ppss continue -C config.cfg

8$ Enter: fg or killall -SIGCONT parallel

9$ ./ppss.sh status -C config.cfg

9$ killall -SIGUSR2 parallel
```

https://github.com/louwrentius/PPSS (Last checked: 2010-12)

## DIFFERENCES BETWEEN pexec AND GNU Parallel

Summary (see legend above):

I1 I2 - I4 I5 - -

M1 - M3 - - M6

O1 O2 O3 - O5 O6

E1 - - E4 - E6 -

R1 - - - - R6 - - -

S1 -

**pexec** is also a tool for running jobs in parallel.

### EXAMPLES FROM pexec MANUAL

Here are the examples from **pexec**'s info page with the equivalent using GNU **parallel**:

```
1$ pexec -o sqrt-%s.dat -p "$(seq 10)" -e NUM -n 4 -c -- \
     'echo "scale=10000;sqrt($NUM)" | bc'

1$ seq 10 | parallel -j4 'echo "scale=10000;sqrt({})" | \
     bc > sqrt-{}.dat'

2$ pexec -p "$(ls myfiles*.ext)" -i %s -o %s.sort -- sort

2$ ls myfiles*.ext | parallel sort {} ">{}.sort"

3$ pexec -f image.list -n auto -e B -u star.log -c -- \
     'fistar $B.fits -f 100 -F id,x,y,flux -o $B.star'
```

```
3$ parallel -a image.list \
     'fistar {}.fits -f 100 -F id,x,y,flux -o {}.star' 2>star.log

4$ pexec -r *.png -e IMG -c -o - -- \
     'convert $IMG ${IMG%.png}.jpeg ; "echo $IMG: done"'

4$ ls *.png | parallel 'convert {} {.}.jpeg; echo {}: done'

5$ pexec -r *.png -i %s -o %s.jpg -c 'pngtopnm | pnmtojpeg'

5$ ls *.png | parallel 'pngtopnm < {} | pnmtojpeg > {}.jpg'

6$ for p in *.png ; do echo ${p%.png} ; done | \
     pexec -f - -i %s.png -o %s.jpg -c 'pngtopnm | pnmtojpeg'

6$ ls *.png | parallel 'pngtopnm < {} | pnmtojpeg > {.}.jpg'

7$ LIST=$(for p in *.png ; do echo ${p%.png} ; done)
   pexec -r $LIST -i %s.png -o %s.jpg -c 'pngtopnm | pnmtojpeg'

7$ ls *.png | parallel 'pngtopnm < {} | pnmtojpeg > {.}.jpg'

8$ pexec -n 8 -r *.jpg -y unix -e IMG -c \
     'pexec -j -m blockread -d $IMG | \
      jpegtopnm | pnmscale 0.5 | pnmtojpeg | \
      pexec -j -m blockwrite -s th_$IMG'

8$ # Combining GNU B<parallel> and GNU B<sem>.
   ls *jpg | parallel -j8 'sem --id blockread cat {} | jpegtopnm |' \
     'pnmscale 0.5 | pnmtojpeg | sem --id blockwrite cat > th_{}'

   # If reading and writing is done to the same disk, this may be
   # faster as only one process will be either reading or writing:
   ls *jpg | parallel -j8 'sem --id diskio cat {} | jpegtopnm |' \
     'pnmscale 0.5 | pnmtojpeg | sem --id diskio cat > th_{}'
```

https://www.gnu.org/software/pexec/ (Last checked: 2024-06)

## DIFFERENCES BETWEEN xjobs AND GNU Parallel

**xjobs** is also a tool for running jobs in parallel. It only supports running jobs on your local computer.

**xjobs** deals badly with special characters just like **xargs**. See the section **DIFFERENCES BETWEEN xargs AND GNU Parallel**.

### EXAMPLES FROM xjobs MANUAL

Here are the examples from **xjobs**'s man page with the equivalent using GNU **parallel**:

```
1$ ls -1 *.zip | xjobs unzip

1$ ls *.zip | parallel unzip

2$ ls -1 *.zip | xjobs -n unzip
```

```
2$ ls *.zip | parallel unzip >/dev/null

3$ find . -name '*.bak' | xjobs gzip

3$ find . -name '*.bak' | parallel gzip

4$ ls -1 *.jar | sed 's/\(.*\)/\1 > \1.idx/' | xjobs jar tf

4$ ls *.jar | parallel jar tf {} '>' {}.idx

5$ xjobs -s script

5$ cat script | parallel

6$ mkfifo /var/run/my_named_pipe;
   xjobs -s /var/run/my_named_pipe &
   echo unzip 1.zip >> /var/run/my_named_pipe;
   echo tar cf /backup/myhome.tar /home/me >> /var/run/my_named_pipe

6$ mkfifo /var/run/my_named_pipe;
   cat /var/run/my_named_pipe | parallel &
   echo unzip 1.zip >> /var/run/my_named_pipe;
   echo tar cf /backup/myhome.tar /home/me >> /var/run/my_named_pipe
```

https://www.maier-komor.de/xjobs.html (Last checked: 2019-01)

## DIFFERENCES BETWEEN prll AND GNU Parallel

**prll** is also a tool for running jobs in parallel. It does not support running jobs on remote computers.

**prll** encourages using BASH aliases and BASH functions instead of scripts. GNU **parallel** supports scripts directly, functions if they are exported using **export -f**, and aliases if using **env_parallel**.

**prll** generates a lot of status information on stderr (standard error) which makes it harder to use the stderr (standard error) output of the job directly as input for another program.

### EXAMPLES FROM prll's MANUAL

Here is the example from **prll**'s man page with the equivalent using GNU **parallel**:

```
1$ prll -s 'mogrify -flip $1' *.jpg

1$ parallel mogrify -flip ::: *.jpg
```

https://github.com/exzombie/prll (Last checked: 2024-06)

## DIFFERENCES BETWEEN dxargs AND GNU Parallel

**dxargs** is also a tool for running jobs in parallel.

**dxargs** does not deal well with more simultaneous jobs than SSHD's MaxStartups. **dxargs** is only built for remote run jobs, but does not support transferring of files.

https://web.archive.org/web/20120518070250/http://www.semicomplete.com/blog/geekery/distributed-xargs.html (Last checked: 2019-01)

## DIFFERENCES BETWEEN mdm/middleman AND GNU Parallel

middleman(mdm) is also a tool for running jobs in parallel.

### EXAMPLES FROM middleman's WEBSITE

Here are the shellscripts of https://web.archive.org/web/20110728064735/http://mdm.berlios.de/usage.html ported to GNU **parallel**:

```
1$ seq 19 | parallel buffon -o - | sort -n > result
   cat files | parallel cmd
   find dir -execdir sem cmd {} \;
```

https://github.com/cklin/mdm (Last checked: 2014-06)

## DIFFERENCES BETWEEN xapply AND GNU Parallel

**xapply** can run jobs in parallel on the local computer.

### EXAMPLES FROM xapply's MANUAL

Here are the examples from **xapply**'s man page with the equivalent using GNU **parallel**:

```
1$ xapply '(cd %1 && make all)' */

1$ parallel 'cd {} && make all' ::: */

2$ xapply -f 'diff %1 ../version5/%1' manifest | more

2$ parallel diff {} ../version5/{} < manifest | more

3$ xapply -p/dev/null -f 'diff %1 %2' manifest1 checklist1

3$ parallel --link diff {1} {2} :::: manifest1 checklist1

4$ xapply 'indent' *.c

4$ parallel indent ::: *.c

5$ find ~ksb/bin -type f ! -perm -111 -print | \
     xapply -f -v 'chmod a+x' -

5$ find ~ksb/bin -type f ! -perm -111 -print | \
     parallel -v chmod a+x

6$ find */ -... | fmt 960 1024 | xapply -f -i /dev/tty 'vi' -

6$ sh <(find */ -... | parallel -s 1024 echo vi)

6$ find */ -... | parallel -s 1024 -Xuj1 vi

7$ find ... | xapply -f -5 -i /dev/tty 'vi' - - - - -

7$ sh <(find ... | parallel -n5 echo vi)

7$ find ... | parallel -n5 -uj1 vi
```

```
8$ xapply -fn "" /etc/passwd

8$ parallel -k echo < /etc/passwd

9$ tr ':' '\012' < /etc/passwd | \
     xapply -7 -nf 'chown %1 %6' - - - - - - -

9$ tr ':' '\012' < /etc/passwd | parallel -N7 chown {1} {6}

10$ xapply '[ -d %1/RCS ] || echo %1' */

10$ parallel '[ -d {}/RCS ] || echo {}' ::: */

11$ xapply -f '[ -f %1 ] && echo %1' List | ...

11$ parallel '[ -f {} ] && echo {}' < List | ...
```

https://www.databits.net/~ksb/msrc/local/bin/xapply/xapply.html (Last checked: 2010-12)

## DIFFERENCES BETWEEN AIX apply AND GNU Parallel

**apply** can build command lines based on a template and arguments - very much like GNU **parallel**. **apply** does not run jobs in parallel. **apply** does not use an argument separator (like **:::**); instead the template must be the first argument.

### EXAMPLES FROM IBM's KNOWLEDGE CENTER

Here are the examples from IBM's Knowledge Center and the corresponding command using GNU **parallel**:

To obtain results similar to those of the ls command, enter:

```
1$ apply echo *
1$ parallel echo ::: *
```

To compare the file named a1 to the file named b1, and the file named a2 to the file named b2, enter:

```
2$ apply -2 cmp a1 b1 a2 b2
2$ parallel -N2 cmp ::: a1 b1 a2 b2
```

To run the who command five times, enter:

```
3$ apply -0 who 1 2 3 4 5
3$ parallel -N0 who ::: 1 2 3 4 5
```

To link all files in the current directory to the directory /usr/joe, enter:

```
4$ apply 'ln %1 /usr/joe' *
4$ parallel ln {} /usr/joe ::: *
```

https://www-01.ibm.com/support/knowledgecenter/ ssw_aix_71/com.ibm.aix.cmds1/apply.htm (Last checked: 2019-01)

## DIFFERENCES BETWEEN paexec AND GNU Parallel

Summary (see legend above):

I1 - - - - - I7

- - M3 - - -

(O1) O2 O3 (O4) (O5) O6 - O8 x -

E1 - - - (E5) - - -

R1 - - - x R6 - R8 R9

- -

**paexec** can run jobs in parallel on both the local and remote computers.

**paexec** has a job dependency facility so a job can depend on another job to be executed successfully. Sort of a poor-man's **make**. This can partly be emulated in GNU **parallel** with **tsort**.

**paexec** fails if output of a single line is > 2 GB. Output of a 2 GB line requires 6 GB RAM. Lines of standard output is interleaved (but there is no half line mixing), and output of standard error mixes. Combined with **paexec_reorder** output order can be the same as input order. In certain situations **paexec** will eat the last newline of standard output.

There seems to be no way to have the number og jobs depend on the number of CPU threads in a mixed server setup: E.g run 4 jobs on a remote server with 4 cores and 16 jobs on a remote server with 16 cores.

### EXAMPLES FROM man paexec

Here are the examples from **man paexec** with the equivalent using GNU **parallel**.

```
  1$ paexec -t '/usr/bin/ssh -x' -n 'host1 host2 host3' \
       -le -g -c calculate-me < tasks.txt |
     paexec_reorder -Mf -Sl

  # GNU Parallel cannot stop processing jobs that depend on another.
  # It can either try all:
  1$ tsort tasks.txt |
       parallel --ssh '/usr/bin/ssh -x' -S "host1,host2,host3" \
         --tagstring {#} --pipe -N1 --log my.log calculate-me

  # Or it can stop at the first failing:
  1$ tsort tasks.txt |
       parallel --ssh '/usr/bin/ssh -x' -S "host1,host2,host3" \
         --tagstring {#} --halt now,fail=1 --pipe -N1 --log my.log
calculate-me

  # To retry the the failed and missing tasks:
  1$ tsort tasks.txt |
       parallel --ssh '/usr/bin/ssh -x' -S "host1,host2,host3" \
         --tagstring {#} --halt now,fail=1 --pipe -N1 --joblog my.log \
         --resume-failed calculate-me

  2$ ls -1 *.wav | paexec -x -n +4 -c 'oggenc -Q'

  2$ ls -1 *.wav | parallel -j4 oggenc -Q

  3$ ls -1 *.wav | paexec -xCil -n+4 flac -f --silent

  3$ ls -1 *.wav | parallel --tagstring {#} -j4 'echo {}; flac -f --silent
{}'

  4$ { uname -s; uname -r; uname -m; } |
       paexec -x -lp -n+2 -c banner |
```

```
      paexec_reorder -l

  4$ { uname -s; uname -r; uname -m; } |
      parallel --tagstring '{#}' -k \
        'banner {} | perl -pe "s/^/getppid().\" \"/e"'

  5$ find . -name '*.dat' -print0 |
      paexec -0 -n+10 -C -J// scp // remoteserver:/remote/path

  5$ find . -name '*.dat' -print0 |
      parallel -0 -j10 -I// scp // remoteserver:/remote/path

  6$ ls -1 *.txt | paexec -n+10 -J%% -c 'awk "BEGIN {print
toupper(\"%%\")}"'

  6$ ls -1 *.txt | parallel -j10 -I%% 'awk "BEGIN {print toupper(\"%%\")}"'
```

**EXAMPLES FROM presentation/paexec.tex**

```
  7$ ls -1 *.wav | \
     paexec -x -c 'flac -s' -n +4 > /dev/null

  7$ ls -1 *.wav | \
     parallel -j4 flac -s > /dev/null

  8$ cat ~/bin/toupper
     #!/usr/bin/awk -f
     {
        print " ", toupper($0)
        print ""  # empty line -- end-of-task marker!
        fflush()  # We must flush stdout!
     }

     cat tasks
     apple
     bananas
     orange

     paexec -t ssh -c ~/bin/toupper -n 'server1 server2' < tasks

  8$ parallel --pipe -n1 -S server1,server2 ~/bin/toupper < tasks

  9$ paexec -lr -t ssh -c ~/bin/toupper -n 'server1 server2' < tasks

  9$ # GNU Parallel has no easy way to prepend the server
     parallel --tagstring {#} --pipe -n1 -S server1,server2 ~/bin/toupper <
 tasks
     cat tasks | parallel --tagstring {#} --pipe -n1 -S server1,server2
--plus \
      ~/bin/toupper '| perl -pe "s/^/{sshlogin}/"'  < tasks

  10$ paexec -n +4 -c ~/bin/toupper < tasks
```

```
10$ parallel -j4 --pipe -n1 ~/bin/toupper < tasks

11$ paexec -x -t ssh -n 'server1 server2' \
      -c "awk 'BEGIN {print toupper(ARGV[1])}' "  < tasks

11$ parallel  -S 'server1,server2' \
      "awk 'BEGIN {print toupper(ARGV[1])}'" < tasks

12$ paexec -x -C -t ssh -n 'server1 server2' \
      awk 'BEGIN {print toupper(ARGV[1])}' < tasks

12$ parallel -S 'server1,server2' -q \
      awk 'BEGIN {print toupper(ARGV[1])}' < tasks

13$ paexec -Z240 -x -t ssh -n 'server1 badhostname server2' \
      -c "awk 'BEGIN {print toupper(ARGV[1])}' " < tasks

13$ parallel --filter-hosts -S 'server1,badhostname,server2' \
      "awk 'BEGIN {print toupper(ARGV[1])}' " < tasks

14$ cat ~/bin/pbanner
    #!/usr/bin/env sh
    while read task; do
       banner -f M "$task" | pv -qL 300
       echo "$PAEXEC_EOT" # end-of-task marker
    done

    cat tasks
    pae
    xec

    paexec -l -mt='SE@X-L0S0!&' -c ~/bin/pbanner -n +2 < tasks |
      paexec_reorder -mt='SE@X-L0S0!&'

14$ paexec -y -lc ~/bin/pbanner -n+2 < tasks | paexec_reorder -y

14$ paexec -l -x -c banner -n+2 < tasks

14$ parallel --pipe -n1 -j2 ~/bin/pbanner < tasks

16$ cat ~/tmp/packages_to_build
    audio/cd-discid audio/abcde
    textproc/gsed audio/abcde
    audio/cdparanoia audio/abcde
    audio/id3v2 audio/abcde
    audio/id3 audio/abcde
    misc/mkcue audio/abcde
    shells/bash audio/abcde
    devel/libtool-base audio/cdparanoia
    devel/gmake audio/cdparanoia
    devel/libtool-base audio/id3lib
    devel/gmake audio/id3v2
    audio/id3lib audio/id3v2
```

```
    devel/m4 devel/bison
    lang/f2c devel/libtool-base
    devel/gmake misc/mkcue
    devel/bison shells/bash

    cat ~/bin/pkg_builder
    #!/usr/bin/awk -f

    {
       print "build " $0
       print "success" # build succeeded! (paexec -ms=)
       print ""        # end-of-task marker
       fflush()        # we must flush stdout
    }

    paexec -g -l -c ~/bin/pkg_builder -n 'server2 server1' \
      -t ssh < ~/tmp/packages_to_build | paexec_reorder

# GNU Parallel cannot postpone jobs that depend on another.
# In some cases this will work
16$ tsort ~/tmp/packages_to_build | parallel -S server2,server1 \
      --pipe -n1 ~/bin/pkg_builder

17$ cat ~/bin/pkg_builder
    #!/usr/bin/awk -f

    {
       print "build " $0
       if ($0 == "devel/gmake")
          print "failure" # Oh no...
          exit 255     # Exit value needed for GNU Parallel
       else
          print "success" # build succeeded!

       print ""        # end-of-task marker
       fflush()        # we must flush stdout
    }

    paexec -gl -c ~/bin/pkg_builder -n 'server2 server1' \
      -t ssh < ~/tmp/packages_to_build | paexec_reorder

# GNU Parallel cannot refrain from starting jobs, that depend on others
# In some cases this will work
17$ tsort ~/tmp/packages_to_build | parallel -S server2,server1 \
      --halt now,fail=1 --pipe -n1 ~/bin/pkg_builder

18$ cat ~/bin/pkg_builder
    #!/usr/bin/awk -f

    {
       "hostname -s" | getline hostname
       print "build " $0 " on " hostname
```

```
        if (hostname == "server1" && $0 == "textproc/gsed")
           exit 139
           # Damn it, I'm dying...
           # Exit value is needed by GNU Parallel
        else
           print "success" # Yes! :-)

        print ""        # end-of-task marker
        fflush()        # we must flush stdout
    }

    paexec -gl -Z300 -t ssh -c ~/bin/pkg_builder \
      -n 'server2 server1' < ~/tmp/packages_to_build |
      paexec_reorder > result

# GNU Parallel retries a job on another server, if --retries > 1
17$ tsort ~/tmp/packages_to_build | parallel -S server2,server1 \
      --halt now,fail=1 --retries 2 --pipe -n1 ~/bin/pkg_builder

18$ ls -1 *.wav | paexec -x -c 'flac -s' -n+3 >/dev/null

18$ ls -1 *.wav | parallel -j3 flac -s >/dev/null

19$ ls -1 *.wav | paexec -ixC -n+3 oggenc -Q | grep .

19$ ls -1 *.wav | parallel -j3 'echo {}; oggenc -Q {}' | grep .

20$ cat calc
    #!/bin/sh
    # $1 -- task given on input
    if test $1 = huge; then
        sleep 6
    else
        sleep 1
    fi
    echo "task $1 done"

    printf 'small1\nsmall2\nsmall3\nsmall4\nsmall5\nhuge\n' |
      time -p paexec -c ~/bin/calc -n +2 -xg | grep -v success

20$ printf 'small1\nsmall2\nsmall3\nsmall4\nsmall5\nhuge\n' |
      time -p parallel -j2 ~/bin/calc | grep -v success

21$ printf 'small1\nsmall2\nsmall3\nsmall4\nweight: huge 6\n' |
      time -p paexec -c ~/bin/calc -n +2 -x -W1 | grep -v success

21$ # GNU Parallel does not support weighted jobs.
    # It can be simulated by sorting:
    printf 'small1\nsmall2\nsmall3\nsmall4\nweight: huge 6\n' |
      perl -pe 's/^weight: (.*) (\d+)/$2 $1/ or s/^/1 /' |
      sort -nr | time parallel ~/bin/calc '{=s/^\d* //=}' |
      grep -v success
```

**EXAMPLES FROM paexec's example dir**

Here are the examples from **paexec**'s example dir with the equivalent using GNU **parallel**:

all_substr

```
$ paexec -lpe -c "`pwd`/cmd" -n +3 <<EOF

$ parallel -j3 --pipe -n1 --tagstring {#} \
    './cmd | perl -pe "s/^/getppid().\" \"/e"' <<EOF
```

cc_wrapper

```
$ paexec -c "env CC=gcc CFLAGS=-O2 `pwd`/cmd" \
          -n 'host1 host2' \
          -t '/usr/bin/ssh -x' <<EOF

$ parallel --pipe -n1 -S 'host1,host2' \
    "env CC=gcc CFLAGS=-O2 `pwd`/cmd" <<EOF

# This is not exactly the same, but avoids the wrapper
$ parallel -S host1,host2 gcc -O2 -c -o {.}.o {} <<EOF
```

cc_wrapper2

```
$ ls -1 $PWD/*.c | paexec -c "env $CC $CFLAGS -c " -n +4 -x

$ ls -1 $PWD/*.c | parallel -j4 "env $CC $CFLAGS -c"
```

dirtest

```
$ paexec -gx -l -c 'test -d' -md=';' -n +3 < tasks

# GNU Parallel cannot refrain from starting jobs, that depend on others
$ parallel -j3 --tag test -d '{= s/.*;// =}; echo $?'  < tasks
```

divide

```
$ paexec -s -l -c cmd_divide -n +3 <<EOF

# GNU Parallel cannot refrain from starting jobs, that depend on others
$ parallel -j3 --pipe -n1 cmd_divide <<EOF
```

make_package

```
1$ paexec -g -le -c "`pwd`/cmd" -n +3 < tasks | paexec_reorder -g -Ms

# GNU Parallel cannot refrain from starting jobs, that depend on others
1$ cat < tasks | parallel --pipe -n1 -j3 "`pwd`/cmd"

2$ paexec -g -le -c "`pwd`/cmd" -n +3 < tasks_cycle

2$ tsort < tasks_cycle | parallel --pipe -n1 -j3 "`pwd`/cmd"
```

toupper

```
$ input | paexec -c "`pwd`/cmd" -n +2 | cut -b 2-
```

```
$ input | parallel --pipe -n1 -j2 "`pwd`/cmd" | cut -b 2-

$ # Without the wrapper:
  input | parallel --pipe -n1 -j2 'awk {print\ toupper\(\$0\)}'
```

wav2flac

```
$ ls -1 "$dir"/*.wav | paexec -x -c 'flac --silent' -n +"$num"

$ ls -1 "$dir"/*.wav | parallel -j $num flac --silent
```

https://github.com/cheusov/paexec (Last checked: 2024-06)

## DIFFERENCES BETWEEN map(sitaramc) AND GNU Parallel

Summary (see legend above):

I1 - - I4 - - (I7)

M1 (M2) M3 (M4) M5 M6

- O2 O3 - O5 - - x x O10

E1 - - - - - -

- - - - - - - - -

- -

(I7): Only under special circumstances. See below.

(M2+M4): Only if there is a single replacement string.

**map** rejects input with special characters:

```
echo "The Cure" > My\ brother\'s\ 12\"\ records

ls | map 'echo %; wc %'
```

It works with GNU **parallel**:

```
ls | parallel 'echo {}; wc {}'
```

Under some circumstances it also works with **map**:

```
ls | map 'echo % works %'
```

But tiny changes make it reject the input with special characters:

```
ls | map 'echo % does not work "%"'
```

This means that many UTF-8 characters will be rejected. This is by design. From the web page: "As such, programs that *quietly handle them, with no warnings at all,* are doing their users a disservice."

**map** delays each job by 0.01 s. This can be emulated by using **parallel --delay 0.01**.

**map** prints '+' on stderr when a job starts, and '-' when a job finishes. This cannot be disabled. **parallel** has **--bar** if you need to see progress.

**map**'s replacement strings (% %D %B %E) can be simulated in GNU **parallel** by putting this in **~/.parallel/config**:

```
--rpl '%'
```

```
--rpl '%D $_=Q(::dirname($_));'
--rpl '%B s:.*/::;s:\.[^/.]+$::;'
--rpl '%E s:.*\.::'
```

**map** does not have an argument separator on the command line, but uses the first argument as command. This makes quoting harder which again may affect readability. Compare:

```
map -p 2 'perl -ne '"'"'/^\S+\s+\S+$/ and print $ARGV,"\n"'"'" *

parallel -q perl -ne '/^\S+\s+\S+$/ and print $ARGV,"\n"' ::: *
```

**map** can do multiple arguments with context replace, but not without context replace:

```
parallel --xargs echo 'BEGIN{'{}'}END' ::: 1 2 3

map "echo 'BEGIN{'%'}END'" 1 2 3
```

**map** has no support for grouping. So this gives the wrong results:

```
parallel perl -e '\$a=\"1{}\"x100000000\;print\ \$a,\"\\n\"' '>' {} \
  ::: a b c d e f
ls -l a b c d e f
parallel -kP4 -n1 grep 1 ::: a b c d e f > out.par
map -n1 -p 4 'grep 1' a b c d e f > out.map-unbuf
map -n1 -p 4 'grep --line-buffered 1' a b c d e f > out.map-linebuf
map -n1 -p 1 'grep --line-buffered 1' a b c d e f > out.map-serial
ls -l out*
md5sum out*
```

## EXAMPLES FROM map's WEBSITE

Here are the examples from **map**'s web page with the equivalent using GNU **parallel**:

```
1$ ls *.gif | map convert % %B.png         # default max-args: 1

1$ ls *.gif | parallel convert {} {.}.png

2$ map "mkdir %B; tar -C %B -xf %" *.tgz   # default max-args: 1

2$ parallel 'mkdir {.}; tar -C {.} -xf {}' :::  *.tgz

3$ ls *.gif | map cp % /tmp                # default max-args: 100

3$ ls *.gif | parallel -X cp {} /tmp

4$ ls *.tar | map -n 1 tar -xf %

4$ ls *.tar | parallel tar -xf

5$ map "cp % /tmp" *.tgz

5$ parallel cp {} /tmp ::: *.tgz

6$ map "du -sm /home/%/mail" alice bob carol
```

```
6$ parallel "du -sm /home/{}/mail" ::: alice bob carol
or if you prefer running a single job with multiple args:
6$ parallel -Xj1 "du -sm /home/{}/mail" ::: alice bob carol

7$ cat /etc/passwd | map -d: 'echo user %1 has shell %7'

7$ cat /etc/passwd | parallel --colsep : 'echo user {1} has shell {7}'

8$ export MAP_MAX_PROCS=$(( `nproc` / 2 ))

8$ export PARALLEL=-j50%
```

https://github.com/sitaramc/map (Last checked: 2020-05)

## DIFFERENCES BETWEEN ladon AND GNU Parallel

**ladon** can run multiple jobs on files in parallel.

**ladon** only works on files and the only way to specify files is using a quoted glob string (such as *.jpg). It is not possible to list the files manually.

As replacement strings it uses FULLPATH DIRNAME BASENAME EXT RELDIR RELPATH

These can be simulated using GNU **parallel** by putting this in **~/.parallel/config**:

```
--rpl 'FULLPATH $_=Q($_);chomp($_=qx{readlink -f $_});'
--rpl 'DIRNAME $_=Q(::dirname($_));chomp($_=qx{readlink -f $_});'
--rpl 'BASENAME s:.*/::;s:\.[^/.]+$::;'
--rpl 'EXT s:.*\.::'
--rpl 'RELDIR $_=Q($_);chomp(($_,$c)=qx{readlink -f $_;pwd});
       s:\Q$c/\E::;$_=::dirname($_);'
--rpl 'RELPATH $_=Q($_);chomp(($_,$c)=qx{readlink -f $_;pwd});
       s:\Q$c/\E::;'
```

**ladon** deals badly with filenames containing " and newline, and it fails for output larger than 200k:

```
ladon '*' -- seq 36000 | wc
```

### EXAMPLES FROM ladon MANUAL

It is assumed that the '--rpl's above are put in **~/.parallel/config** and that it is run under a shell that supports '**' globbing (such as **zsh**):

```
1$ ladon "**/*.txt" -- echo RELPATH

1$ parallel echo RELPATH ::: **/*.txt

2$ ladon "~/Documents/**/*.pdf" -- shasum FULLPATH >hashes.txt

2$ parallel shasum FULLPATH ::: ~/Documents/**/*.pdf >hashes.txt

3$ ladon -m thumbs/RELDIR "**/*.jpg" -- convert FULLPATH \
     -thumbnail 100x100^ -gravity center -extent 100x100 \
     thumbs/RELPATH

3$ parallel mkdir -p thumbs/RELDIR\; convert FULLPATH
     -thumbnail 100x100^ -gravity center -extent 100x100 \
```

```
     thumbs/RELPATH ::: **/*.jpg

4$ ladon "~/Music/*.wav" -- lame -V 2 FULLPATH DIRNAME/BASENAME.mp3

4$ parallel lame -V 2 FULLPATH DIRNAME/BASENAME.mp3 ::: ~/Music/*.wav
```

https://github.com/danielgtaylor/ladon (Last checked: 2024-06)

## DIFFERENCES BETWEEN jobflow AND GNU Parallel

Summary (see legend above):

I1 - - - - - I7

- - M3 - - (M6)

O1 O2 O3 - O5 O6 (O7) - - O10

E1 - - - - E6 -

- - - - - - - - -

- -

**jobflow** can run multiple jobs in parallel.

Just like **xargs** output from **jobflow** jobs running in parallel mix together by default. **jobflow** can buffer into files with **-buffered** (placed in /run/shm), but these are not cleaned up if **jobflow** dies unexpectedly (e.g. by Ctrl-C). If the total output is big (in the order of RAM+swap) it can cause the system to slow to a crawl and eventually run out of memory.

Just like **xargs** redirection and composed commands require wrapping with **bash -c**.

Input lines can at most be 4096 bytes.

**jobflow** is faster than GNU **parallel** but around 6 times slower than **parallel-bash**.

**jobflow** has no equivalent for **--pipe**, or **--sshlogin**.

**jobflow** makes it possible to set resource limits on the running jobs. This can be emulated by GNU **parallel** using **bash**'s **ulimit**:

```
jobflow -limits=mem=100M,cpu=3,fsize=20M,nofiles=300 myjob

parallel 'ulimit -v 102400 -t 3 -f 204800 -n 300 myjob'
```

### EXAMPLES FROM jobflow README

```
1$ cat things.list | jobflow -threads=8 -exec ./mytask {}

1$ cat things.list | parallel -j8 ./mytask {}

2$ seq 100 | jobflow -threads=100 -exec echo {}

2$ seq 100 | parallel -j100 echo {}

3$ cat urls.txt | jobflow -threads=32 -exec wget {}

3$ cat urls.txt | parallel -j32 wget {}

4$ find . -name '*.bmp' | \
```

```
     jobflow -threads=8 -exec bmp2jpeg {.}.bmp {.}.jpg

4$ find . -name '*.bmp' | \
     parallel -j8 bmp2jpeg {.}.bmp {.}.jpg

5$ seq 100 | jobflow -skip 10 -count 10

5$ seq 100 | parallel --filter '{1} > 10 and {1} <= 20' echo

5$ seq 100 | parallel echo '{= $_>10 and $_<=20 or skip() =}'
```

https://github.com/rofl0r/jobflow (Last checked: 2024-06)

## DIFFERENCES BETWEEN gargs AND GNU Parallel

**gargs** can run multiple jobs in parallel.

Older versions cache output in memory. This causes it to be extremely slow when the output is larger than the physical RAM, and can cause the system to run out of memory.

See more details on this in **man parallel_design**.

Newer versions cache output in files, but leave files in $TMPDIR if it is killed.

Output to stderr (standard error) is changed if the command fails.

### EXAMPLES FROM gargs WEBSITE

```
1$ seq 12 -1 1 | gargs -p 4 -n 3 "sleep {0}; echo {1} {2}"

1$ seq 12 -1 1 | parallel -P 4 -n 3 "sleep {1}; echo {2} {3}"

2$ cat t.txt | gargs --sep "\s+" \
     -p 2 "echo '{0}:{1}-{2}' full-line: \'{}\'"

2$ cat t.txt | parallel --colsep "\\s+" \
     -P 2 "echo '{1}:{2}-{3}' full-line: \'{}\'"
```

https://github.com/brentp/gargs (Last checked: 2016-08)

## DIFFERENCES BETWEEN orgalorg AND GNU Parallel

**orgalorg** can run the same job on multiple machines. This is related to **--onall** and **--nonall**.

**orgalorg** supports entering the SSH password - provided it is the same for all servers. GNU **parallel** advocates using **ssh-agent** instead, but it is possible to emulate **orgalorg**'s behavior by setting SSHPASS and by using **--ssh "sshpass ssh"**.

To make the emulation easier, make a simple alias:

```
alias par_emul="parallel -j0 --ssh 'sshpass ssh' --nonall --tag --lb"
```

If you want to supply a password run:

```
SSHPASS=`ssh-askpass`
```

or set the password directly:

```
SSHPASS=P4$$w0rd!
```

If the above is set up you can then do:

```
orgalorg -o frontend1 -o frontend2 -p -C uptime
par_emul -S frontend1 -S frontend2 uptime

orgalorg -o frontend1 -o frontend2 -p -C top -bid 1
par_emul -S frontend1 -S frontend2 top -bid 1

orgalorg -o frontend1 -o frontend2 -p -er /tmp -n \
  'md5sum /tmp/bigfile' -S bigfile
par_emul -S frontend1 -S frontend2 --basefile bigfile \
  --workdir /tmp md5sum /tmp/bigfile
```

**orgalorg** has a progress indicator for the transferring of a file. GNU **parallel** does not.

https://github.com/reconquest/orgalorg (Last checked: 2016-08)

## DIFFERENCES BETWEEN Rust parallel(mmstick) AND GNU Parallel

Rust parallel focuses on speed. It is almost as fast as **xargs**, but not as fast as **parallel-bash**. It implements a few features from GNU **parallel**, but lacks many functions. All these fail:

```
# Read arguments from file
parallel -a file echo
# Changing the delimiter
parallel -d _ echo ::: a_b_c_
```

These do something different from GNU **parallel**

```
# -q to protect quoted $ and space
parallel -q perl -e '$a=shift; print "$a"x100000000' ::: a b c
# Generation of combination of inputs
parallel echo {1} {2} ::: red green blue ::: S M L XL XXL
# {= perl expression =} replacement string
parallel echo '{= s/new/old/ =}' ::: my.new your.new
# --pipe
seq 100000 | parallel --pipe wc
# linked arguments
parallel echo ::: S M L :::+ sml med lrg ::: R G B :::+ red grn blu
# Run different shell dialects
zsh -c 'parallel echo \={} ::: zsh && true'
csh -c 'parallel echo \$\{\} ::: shell && true'
bash -c 'parallel echo \$\({}\) ::: pwd && true'
# Rust parallel does not start before the last argument is read
(seq 10; sleep 5; echo 2) | time parallel -j2 'sleep 2; echo'
tail -f /var/log/syslog | parallel echo
```

Most of the examples from the book GNU Parallel 2018 do not work, thus Rust parallel is not close to being a compatible replacement.

Rust parallel has no remote facilities.

It uses /tmp/parallel for tmp files and does not clean up if terminated abruptly. If another user on the system uses Rust parallel, then /tmp/parallel will have the wrong permissions and Rust parallel will fail. A malicious user can setup the right permissions and symlink the output file to one of the user's files and next time the user uses Rust parallel it will overwrite this file.

```
attacker$ mkdir /tmp/parallel
attacker$ chmod a+rwX /tmp/parallel
```

```
# Symlink to the file the attacker wants to zero out
attacker$ ln -s ~victim/.important-file /tmp/parallel/stderr_1
victim$ seq 1000 | parallel echo
# This file is now overwritten with stderr from 'echo'
victim$ cat ~victim/.important-file
```

If /tmp/parallel runs full during the run, Rust parallel does not report this, but finishes with success - thereby risking data loss.

https://github.com/mmstick/parallel (Last checked: 2016-08)

## DIFFERENCES BETWEEN parallelion AND GNU Parallel

Summary (see legend above):

- (I2) - I4 - - -

M1 - M3 - - M6

- O2 O3 - O5 (O6) - x x

E1 - - (E4) E5 - - E8 ?

- - - - - - - - -

- -

I2: I was unable to cannot get **parallelion** to read from a file.

O6: There is extra output if a job fails.

E4: The default number of parallel jobs is the number of cpu threads.

-- is needed to force args not be parsed as options:

```
parallelion 'echo {}' -- Runs without -v

parallelion 'echo {}' Runs with -v
```

The commands are run through **ion** shell.

Ctrl-C does not stop processing.

The **--log** is similar to syslog - not a table.

The progressbar is nice.

**parallelion** is fast: 0.1 ms/job. Similar to **parallel-bash**.

### EXAMPLES FROM parallelion

```
1$ parallelion --progress 'echo {}' {1..1000}

1$ parallel --bar echo {} ::: {1..1000}

2$ parallelion --progress 'echo {}' $(seq 1 999)

2$ seq 1 999 | parallel --bar echo
```

https://gitlab.redox-os.org/redox-os/parallel (Last checked: 2024-08)

## DIFFERENCES BETWEEN Rush AND GNU Parallel

**rush** (https://github.com/shenwei356/rush) is written in Go and based on **gargs**.

Just like GNU **parallel rush** buffers in temporary files. But opposite GNU **parallel rush** does not clean up, if the process dies abnormally.

**rush** has some string manipulations that can be emulated by putting this into ~/.parallel/config (/ is used instead of %, and % is used instead of ^ as that is closer to bash's ${var%postfix}):

```
--rpl '{:} s:(\.[^/]+)*$::'
--rpl '{:%([^}]+?)} s:$$1(\.[^/]+)*$::'
--rpl '{/:%([^}]*?)} s:.*/(.*)$$1(\.[^/]+)*$:$1:'
--rpl '{/:} s:(.*/)?([^/.]+)(\.[^/]+)*$:$2:'
--rpl '{@(.*?)} /$$1/ and $_=$1;'
```

## EXAMPLES FROM rush's WEBSITE

Here are the examples from **rush**'s website with the equivalent command in GNU **parallel**.

### 1. Simple run, quoting is not necessary

```
1$ seq 1 3 | rush echo {}

1$ seq 1 3 | parallel echo {}
```

### 2. Read data from file (`-i`)

```
2$ rush echo {} -i data1.txt -i data2.txt

2$ cat data1.txt data2.txt | parallel echo {}
```

### 3. Keep output order (`-k`)

```
3$ seq 1 3 | rush 'echo {}' -k

3$ seq 1 3 | parallel -k echo {}
```

### 4. Timeout (`-t`)

```
4$ time seq 1 | rush 'sleep 2; echo {}' -t 1

4$ time seq 1 | parallel --timeout 1 'sleep 2; echo {}'
```

### 5. Retry (`-r`)

```
5$ seq 1 | rush 'python unexisted_script.py' -r 1

5$ seq 1 | parallel --retries 2 'python unexisted_script.py'
```

Use **-u** to see it is really run twice:

```
5$ seq 1 | parallel -u --retries 2 'python unexisted_script.py'
```

### 6. Dirname (`{/}`) and basename (`{%}`) and remove custom suffix (`{^suffix}`)

```
6$ echo dir/file_1.txt.gz | rush 'echo {/} {%} {^_1.txt.gz}'

6$ echo dir/file_1.txt.gz |
     parallel --plus echo {//} {/} {%_1.txt.gz}
```

**7. Get basename, and remove last (`{.}`) or any (`{:}`) extension**

```
7$ echo dir.d/file.txt.gz | rush 'echo {.} {:} {%.} {%:}'

7$ echo dir.d/file.txt.gz | parallel 'echo {.} {:} {/.} {/:}'
```

**8. Job ID, combine fields index and other replacement strings**

```
8$ echo 12 file.txt dir/s_1.fq.gz |
     rush 'echo job {#}: {2} {2.} {3%:^_1}'

8$ echo 12 file.txt dir/s_1.fq.gz |
     parallel --colsep ' ' 'echo job {#}: {2} {2.} {3/:%_1}'
```

**9. Capture submatch using regular expression (`{@regexp}`)**

```
9$ echo read_1.fq.gz | rush 'echo {@(.+)_\d}'

9$ echo read_1.fq.gz | parallel 'echo {@(.+)_\d}'
```

**10. Custom field delimiter (`-d`)**

```
10$ echo a=b=c | rush 'echo {1} {2} {3}' -d =

10$ echo a=b=c | parallel -d = echo {1} {2} {3}
```

**11. Send multi-lines to every command (`-n`)**

```
11$ seq 5 | rush -n 2 -k 'echo "{}"; echo'

11$ seq 5 |
      parallel -n 2 -k \
        'echo {=-1 $_=join"\n",@arg[1..$#arg] =}; echo'

11$ seq 5 | rush -n 2 -k 'echo "{}"; echo' -J ' '

11$ seq 5 | parallel -n 2 -k 'echo {}; echo'
```

**12. Custom record delimiter (`-D`), note that empty records are not used.**

```
12$ echo a b c d | rush -D " " -k 'echo {}'

12$ echo a b c d | parallel -d " " -k 'echo {}'

12$ echo abcd | rush -D "" -k 'echo {}'

Cannot be done by GNU Parallel

12$ cat fasta.fa
>seq1
tag
>seq2
cat
gat
```

```
>seq3
attac
a
cat

12$ cat fasta.fa | rush -D ">" \
      'echo FASTA record {#}: name: {1} sequence: {2}' -k -d "\n"
    # rush fails to join the multiline sequences

12$ cat fasta.fa | (read -n1 ignore_first_char;
      parallel -d '>' --colsep '\n' echo FASTA record {#}: \
        name: {1} sequence: '{=2 $_=join"",@arg[2..$#arg]=}'
    )
```

**13. Assign value to variable, like `awk -v` (`-v`)**

```
13$ seq 1 |
      rush 'echo Hello, {fname} {lname}!' -v fname=Wei -v lname=Shen

13$ seq 1 |
      parallel -N0 \
        'fname=Wei; lname=Shen; echo Hello, ${fname} ${lname}!'

13$ for var in a b; do \
13$   seq 1 3 | rush -k -v var=$var 'echo var: {var}, data: {}'; \
13$ done
```

In GNU **parallel** you would typically do:

```
13$ seq 1 3 | parallel -k echo var: {1}, data: {2} ::: a b :::: -
```

If you *really* want the var:

```
13$ seq 1 3 |
      parallel -k var={1} ';echo var: $var, data: {}' ::: a b :::: -
```

If you *really* want the **for**-loop:

```
13$ for var in a b; do
      export var;
      seq 1 3 | parallel -k 'echo var: $var, data: {}';
    done
```

Contrary to **rush** this also works if the value is complex like:

```
My brother's 12" records
```

**14. Preset variable (`-v`), avoid repeatedly writing verbose replacement strings**

```
14$ # naive way
    echo read_1.fq.gz | rush 'echo {:^_1} {:^_1}_2.fq.gz'

14$ echo read_1.fq.gz | parallel 'echo {:%_1} {:%_1}_2.fq.gz'

14$ # macro + removing suffix
```

```
    echo read_1.fq.gz |
      rush -v p='{:^_1}' 'echo {p} {p}_2.fq.gz'

14$ echo read_1.fq.gz |
      parallel 'p={:%_1}; echo $p ${p}_2.fq.gz'

14$ # macro + regular expression
    echo read_1.fq.gz | rush -v p='{@(.+?)_\d}' 'echo {p} {p}_2.fq.gz'

14$ echo read_1.fq.gz | parallel 'p={@(.+?)_\d}; echo $p ${p}_2.fq.gz'
```

Contrary to **rush** GNU **parallel** works with complex values:

```
14$ echo "My brother's 12\"read_1.fq.gz" |
      parallel 'p={@(.+?)_\d}; echo $p ${p}_2.fq.gz'
```

**15. Interrupt jobs by `Ctrl-C`, rush will stop unfinished commands and exit.**

```
15$ seq 1 20 | rush 'sleep 1; echo {}'
    ^C

15$ seq 1 20 | parallel 'sleep 1; echo {}'
    ^C
```

**16. Continue/resume jobs (`-c`). When some jobs failed (by execution failure, timeout, or canceling by user with `Ctrl + C`), please switch flag `-c/--continue` on and run again, so that `rush` can save successful commands and ignore them in *NEXT* run.**

```
16$ seq 1 3 | rush 'sleep {}; echo {}' -t 3 -c
    cat successful_cmds.rush
    seq 1 3 | rush 'sleep {}; echo {}' -t 3 -c

16$ seq 1 3 | parallel --joblog mylog --timeout 2 \
      'sleep {}; echo {}'
    cat mylog
    seq 1 3 | parallel --joblog mylog --retry-failed \
      'sleep {}; echo {}'
```

Multi-line jobs:

```
16$ seq 1 3 | rush 'sleep {}; echo {}; \
      echo finish {}' -t 3 -c -C finished.rush
    cat finished.rush
    seq 1 3 | rush 'sleep {}; echo {}; \
      echo finish {}' -t 3 -c -C finished.rush

16$ seq 1 3 |
      parallel --joblog mylog --timeout 2 'sleep {}; echo {}; \
        echo finish {}'
    cat mylog
    seq 1 3 |
      parallel --joblog mylog --retry-failed 'sleep {}; echo {}; \
        echo finish {}'
```

**17. A comprehensive example: downloading 1K+ pages given by three URL list files using**

**`phantomjs save_page.js` (some page contents are dynamically generated by Javascript, so `wget` does not work). Here I set max jobs number (`-j`) as `20`, each job has a max running time (`-t`) of `60` seconds and `3` retry changes (`-r`). Continue flag `-c` is also switched on, so we can continue unfinished jobs. Luckily, it's accomplished in one run :)**

```
17$ for f in $(seq 2014 2016); do \
      /bin/rm -rf $f; mkdir -p $f; \
      cat $f.html.txt | rush -v d=$f -d = \
        'phantomjs save_page.js "{}" > {d}/{3}.html' \
        -j 20 -t 60 -r 3 -c; \
    done
```

GNU **parallel** can append to an existing joblog with '+':

```
17$ rm mylog
    for f in $(seq 2014 2016); do
      /bin/rm -rf $f; mkdir -p $f;
      cat $f.html.txt |
        parallel -j20 --timeout 60 --retries 4 --joblog +mylog \
          --colsep = \
          phantomjs save_page.js {1}={2}={3} '>' $f/{3}.html
    done
```

**18. A bioinformatics example: mapping with `bwa`, and processing result with `samtools`:**

```
18$ ref=ref/xxx.fa
    threads=25
    ls -d raw.cluster.clean.mapping/* \
      | rush -v ref=$ref -v j=$threads -v p='{}/{%}' \
      'bwa mem -t {j} -M -a {ref} {p}_1.fq.gz {p}_2.fq.gz >{p}.sam;\
      samtools view -bS {p}.sam > {p}.bam; \
      samtools sort -T {p}.tmp -@ {j} {p}.bam -o {p}.sorted.bam; \
      samtools index {p}.sorted.bam; \
      samtools flagstat {p}.sorted.bam > {p}.sorted.bam.flagstat; \
      /bin/rm {p}.bam {p}.sam;' \
      -j 2 --verbose -c -C mapping.rush
```

GNU **parallel** would use a function:

```
18$ ref=ref/xxx.fa
    export ref
    thr=25
    export thr
    bwa_sam() {
      p="$1"
      bam="$p".bam
      sam="$p".sam
      sortbam="$p".sorted.bam
      bwa mem -t $thr -M -a $ref ${p}_1.fq.gz ${p}_2.fq.gz > "$sam"
      samtools view -bS "$sam" > "$bam"
      samtools sort -T ${p}.tmp -@ $thr "$bam" -o "$sortbam"
      samtools index "$sortbam"
      samtools flagstat "$sortbam" > "$sortbam".flagstat
      /bin/rm "$bam" "$sam"
    }
    export -f bwa_sam
    ls -d raw.cluster.clean.mapping/* |
```

```
parallel -j 2 --verbose --joblog mylog bwa_sam
```

### Other rush features

**rush** has:

* **awk -v** like custom defined variables (**-v**)

  With GNU **parallel** you would simply set a shell variable:

  ```
  parallel 'v={}; echo "$v"' ::: foo
  echo foo | rush -v v={} 'echo {v}'
  ```

  Also **rush** does not like special chars. So these **do not work**:

  ```
  echo does not work | rush -v v=\" 'echo {v}'
  echo "My  brother's  12\"  records" | rush -v v={} 'echo {v}'
  ```

  Whereas the corresponding GNU **parallel** version works:

  ```
  parallel 'v=\"; echo "$v"' ::: works
  parallel 'v={}; echo "$v"' ::: "My  brother's  12\"  records"
  ```

* Exit on first error(s) (-e)

  This is called **--halt now,fail=1** (or shorter: **--halt 2**) when used with GNU **parallel**.

* Settable records sending to every command (**-n**, default 1)

  This is also called **-n** in GNU **parallel**.

* Practical replacement strings

  {:} remove any extension

  With GNU **parallel** this can be emulated by:

  ```
  parallel --plus echo '{/\..*/}' ::: foo.ext.bar.gz
  ```

  {^suffix}, remove suffix

  With GNU **parallel** this can be emulated by:

  ```
  parallel --plus echo '{%.bar.gz}' ::: foo.ext.bar.gz
  ```

  {@regexp}, capture submatch using regular expression

  With GNU **parallel** this can be emulated by:

  ```
  parallel --rpl '{@(.*?)} /$$1/ and $_=$1;' \
    echo '{@\d_(.*).gz}' ::: 1_foo.gz
  ```

  {%.}, {%:}, basename without extension

  With GNU **parallel** this can be emulated by:

  ```
  parallel echo '{= s:.*/::;s/\..*// =}' ::: dir/foo.bar.gz
  ```

  And if you need it often, you define a **--rpl** in **$HOME/.parallel/config**:

  ```
  --rpl '{%.} s:.*/::;s/\..*//'
  --rpl '{%:} s:.*/::;s/\..*//'
  ```

  Then you can use them as:

  ```
  parallel echo {%.} {%:} ::: dir/foo.bar.gz
  ```

* Preset variable (macro)

E.g.

```
echo foosuffix | rush -v p={^suffix} 'echo {p}_new_suffix'
```

With GNU **parallel** this can be emulated by:

```
echo foosuffix |
  parallel --plus 'p={%suffix}; echo ${p}_new_suffix'
```

Opposite **rush** GNU **parallel** works fine if the input contains double space, ' and ":

```
echo "1'6\"  foosuffix" |
  parallel --plus 'p={%suffix}; echo "${p}"_new_suffix'
```

* Commands of multi-lines

While you *can* use multi-lined commands in GNU **parallel**, to improve readability GNU **parallel** discourages the use of multi-line commands. In most cases it can be written as a function:

```
seq 1 3 |
  parallel --timeout 2 --joblog my.log 'sleep {}; echo {}; \
    echo finish {}'
```

Could be written as:

```
doit() {
  sleep "$1"
  echo "$1"
  echo finish "$1"
}
export -f doit
seq 1 3 | parallel --timeout 2 --joblog my.log doit
```

The failed commands can be resumed with:

```
seq 1 3 |
  parallel --resume-failed --joblog my.log 'sleep {}; echo {};\
    echo finish {}'
```

https://github.com/shenwei356/rush (Last checked: 2017-05)

## DIFFERENCES BETWEEN ClusterSSH AND GNU Parallel

ClusterSSH solves a different problem than GNU **parallel**.

ClusterSSH opens a terminal window for each computer and using a master window you can run the same command on all the computers. This is typically used for administrating several computers that are almost identical.

GNU **parallel** runs the same (or different) commands with different arguments in parallel possibly using remote computers to help computing. If more than one computer is listed in **-S** GNU **parallel** may only use one of these (e.g. if there are 8 jobs to be run and one computer has 8 cores).

GNU **parallel** can be used as a poor-man's version of ClusterSSH:

```
parallel --nonall -S server-a,server-b do_stuff foo bar
```

https://github.com/duncs/clusterssh (Last checked: 2024-06)

## DIFFERENCES BETWEEN coshell AND GNU Parallel

**coshell** only accepts full commands on standard input. Any quoting needs to be done by the user.

Commands are run in **sh** so any **bash**/**tcsh**/**zsh** specific syntax will not work.

Output can be buffered by using **-d**. Output is buffered in memory, so big output can cause swapping and therefore be terrible slow or even cause out of memory.

https://github.com/gdm85/coshell (Last checked: 2019-01)

## DIFFERENCES BETWEEN spread AND GNU Parallel

Summary (see legend above):

- - - I4 - - I7

M1 - - - - -

O1 O2 O3 O4 O5 O6 - O8 - O10

- - - - - - -

- - - - - - - - -

- -

**spread** runs commands on all directories. It does not run jobs in parallel.

It can be emulated with GNU **parallel** using this Bash function:

```
spread() {
  _cmds() {
    perl -e '$"=" && ";print "@ARGV"' "cd {}" "$@"
  }
  parallel $(_cmds "$@")'|| echo exit status $?' ::: */
}
```

https://github.com/tfogo/spread (Last checked: 2024-06)

## DIFFERENCES BETWEEN pyargs AND GNU Parallel

**pyargs** deals badly with input containing spaces. It buffers stdout, but not stderr. It buffers in RAM. {} does not work as replacement string. It does not support running functions.

**pyargs** does not support composed commands if run with **--lines**, and fails on **pyargs traceroute gnu.org fsf.org**.

### Examples

```
seq 5 | pyargs -P50 -L seq
seq 5 | parallel -P50 --lb seq


seq 5 | pyargs -P50 --mark -L seq
seq 5 | parallel -P50 --lb \
  --tagstring OUTPUT'[{= $_=$job->replaced() =}]' seq
# Similar, but not precisely the same
seq 5 | parallel -P50 --lb --tag seq


seq 5 | pyargs -P50  --mark command
# Somewhat longer with GNU Parallel due to the special
#   --mark formatting
cmd="$(echo "command" | parallel --shellquote)"
wrap_cmd() {
   echo "MARK $cmd $@================================" >&3
   echo "OUTPUT START[$cmd $@]:"
   eval $cmd "$@"
   echo "OUTPUT END[$cmd $@]"
}
```

```
(seq 5 | env_parallel -P2 wrap_cmd) 3>&1
# Similar, but not exactly the same
seq 5 | parallel -t --tag command

(echo '1  2  3';echo 4 5 6) | pyargs  --stream seq
(echo '1  2  3';echo 4 5 6) | perl -pe 's/\n/ /' |
  parallel -r -d' ' seq
# Similar, but not exactly the same
parallel seq ::: 1 2 3 4 5 6
```

https://github.com/robertblackwell/pyargs (Last checked: 2024-01)

## DIFFERENCES BETWEEN concurrently AND GNU Parallel

**concurrently** runs jobs in parallel.

The output is prepended with the job number, and may be incomplete:

```
$ concurrently 'seq 100000' | (sleep 3;wc -l)
7165
```

When pretty printing it caches output in memory. Output mixes by using test MIX below whether or not output is cached.

There seems to be no way of making a template command and have **concurrently** fill that with different args. The full commands must be given on the command line.

There is also no way of controlling how many jobs should be run in parallel at a time - i.e. "number of jobslots". Instead all jobs are simply started in parallel.

https://github.com/kimmobrunfeldt/concurrently (Last checked: 2019-01)

## DIFFERENCES BETWEEN map(soveran) AND GNU Parallel

**map** does not run jobs in parallel by default. The README suggests using:

```
... | map t 'sleep $t && say done &'
```

But this fails if more jobs are run in parallel than the number of available processes. Since there is no support for parallelization in **map** itself, the output also mixes:

```
seq 10 | map i 'echo start-$i && sleep 0.$i && echo end-$i &'
```

The major difference is that GNU **parallel** is built for parallelization and **map** is not. So GNU **parallel** has lots of ways of dealing with the issues that parallelization raises:

- Keep the number of processes manageable
- Make sure output does not mix
- Make Ctrl-C kill all running processes

### EXAMPLES FROM maps WEBSITE

Here are the 5 examples converted to GNU Parallel:

```
1$ ls *.c | map f 'foo $f'
1$ ls *.c | parallel foo

2$ ls *.c | map f 'foo $f; bar $f'
2$ ls *.c | parallel 'foo {}; bar {}'
```

```
3$ cat urls | map u 'curl -O $u'
3$ cat urls | parallel curl -O

4$ printf "1\n1\n1\n" | map t 'sleep $t && say done'
4$ printf "1\n1\n1\n" | parallel 'sleep {} && say done'
4$ parallel 'sleep {} && say done' ::: 1 1 1

5$ printf "1\n1\n1\n" | map t 'sleep $t && say done &'
5$ printf "1\n1\n1\n" | parallel -j0 'sleep {} && say done'
5$ parallel -j0 'sleep {} && say done' ::: 1 1 1
```

https://github.com/soveran/map (Last checked: 2019-01)

## DIFFERENCES BETWEEN loop AND GNU Parallel

**loop** mixes stdout and stderr:

```
loop 'ls /no-such-file' >/dev/null
```

**loop**'s replacement string **$ITEM** does not quote strings:

```
echo 'two  spaces' | loop 'echo $ITEM'
```

**loop** cannot run functions:

```
myfunc() { echo joe; }
export -f myfunc
loop 'myfunc this fails'
```

### EXAMPLES FROM loop's WEBSITE

Some of the examples from https://github.com/Miserlou/Loop/ can be emulated with GNU **parallel**:

```
# A couple of functions will make the code easier to read
$ loopy() {
    yes | parallel -uN0 -j1 "$@"
  }
$ export -f loopy
$ time_out() {
    parallel -uN0 -q --timeout "$@" ::: 1
  }
$ match() {
    perl -0777 -ne 'grep /'"$1"'/,$_ and print or exit 1'
  }
$ export -f match

$ loop 'ls' --every 10s
$ loopy --delay 10s ls

$ loop 'touch $COUNT.txt' --count-by 5
$ loopy touch '{= $_=seq()*5 =}'.txt

$ loop --until-contains 200 -- \
    ./get_response_code.sh --site mysite.biz
$ loopy --halt now,success=1 \
    './get_response_code.sh --site mysite.biz | match 200'
```

```
$ loop './poke_server' --for-duration 8h
$ time_out 8h loopy ./poke_server

$ loop './poke_server' --until-success
$ loopy --halt now,success=1 ./poke_server

$ cat files_to_create.txt | loop 'touch $ITEM'
$ cat files_to_create.txt | parallel touch {}

$ loop 'ls' --for-duration 10min --summary
# --joblog is somewhat more verbose than --summary
$ time_out 10m loopy --joblog my.log ./poke_server; cat my.log

$ loop 'echo hello'
$ loopy echo hello

$ loop 'echo $COUNT'
# GNU Parallel counts from 1
$ loopy echo {#}
# Counting from 0 can be forced
$ loopy echo '{= $_=seq()-1 =}'

$ loop 'echo $COUNT' --count-by 2
$ loopy echo '{= $_=2*(seq()-1) =}'

$ loop 'echo $COUNT' --count-by 2 --offset 10
$ loopy echo '{= $_=10+2*(seq()-1) =}'

$ loop 'echo $COUNT' --count-by 1.1
# GNU Parallel rounds 3.3000000000000003 to 3.3
$ loopy echo '{= $_=1.1*(seq()-1) =}'

$ loop 'echo $COUNT $ACTUALCOUNT' --count-by 2
$ loopy echo '{= $_=2*(seq()-1) =} {#}'

$ loop 'echo $COUNT' --num 3 --summary
# --joblog is somewhat more verbose than --summary
$ seq 3 | parallel --joblog my.log echo; cat my.log

$ loop 'ls -foobarbatz' --num 3 --summary
# --joblog is somewhat more verbose than --summary
$ seq 3 | parallel --joblog my.log -N0 ls -foobarbatz; cat my.log

$ loop 'echo $COUNT' --count-by 2 --num 50 --only-last
# Can be emulated by running 2 jobs
$ seq 49 | parallel echo '{= $_=2*(seq()-1) =}' >/dev/null
$ echo 50| parallel echo '{= $_=2*(seq()-1) =}'

$ loop 'date' --every 5s
$ loopy --delay 5s date

$ loop 'date' --for-duration 8s --every 2s
```

```
$ time_out 8s loopy --delay 2s date

$ loop 'date -u' --until-time '2018-05-25 20:50:00' --every 5s
$ seconds=$((`date -d 2019-05-25T20:50:00 +%s` - `date  +%s`))s
$ time_out $seconds loopy --delay 5s date -u

$ loop 'echo $RANDOM' --until-contains "666"
$ loopy --halt now,success=1 'echo $RANDOM | match 666'

$ loop 'if (( RANDOM % 2 )); then
          (echo "TRUE"; true);
        else
          (echo "FALSE"; false);
        fi' --until-success
$ loopy --halt now,success=1 'if (( $RANDOM % 2 )); then
                                (echo "TRUE"; true);
                              else
                                (echo "FALSE"; false);
                              fi'

$ loop 'if (( RANDOM % 2 )); then
    (echo "TRUE"; true);
  else
    (echo "FALSE"; false);
  fi' --until-error
$ loopy --halt now,fail=1 'if (( $RANDOM % 2 )); then
                             (echo "TRUE"; true);
                           else
                             (echo "FALSE"; false);
                           fi'

$ loop 'date' --until-match "(\d{4})"
$ loopy --halt now,success=1 'date | match [0-9][0-9][0-9][0-9]'

$ loop 'echo $ITEM' --for red,green,blue
$ parallel echo ::: red green blue

$ cat /tmp/my-list-of-files-to-create.txt | loop 'touch $ITEM'
$ cat /tmp/my-list-of-files-to-create.txt | parallel touch

$ ls | loop 'cp $ITEM $ITEM.bak'; ls
$ ls | parallel cp {} {}.bak; ls

$ loop 'echo $ITEM | tr a-z A-Z' -i
$ parallel 'echo {} | tr a-z A-Z'
# Or more efficiently:
$ parallel --pipe tr a-z A-Z

$ loop 'echo $ITEM' --for "`ls`"
$ parallel echo {} ::: "`ls`"

$ ls | loop './my_program $ITEM' --until-success;
$ ls | parallel --halt now,success=1 ./my_program {}
```

```
$ ls | loop './my_program $ITEM' --until-fail;
$ ls | parallel --halt now,fail=1 ./my_program {}

$ ./deploy.sh;
  loop 'curl -sw "%{http_code}" http://coolwebsite.biz' \
    --every 5s --until-contains 200;
  ./announce_to_slack.sh
$ ./deploy.sh;
  loopy --delay 5s --halt now,success=1 \
  'curl -sw "%{http_code}" http://coolwebsite.biz | match 200';
  ./announce_to_slack.sh

$ loop "ping -c 1 mysite.com" --until-success; ./do_next_thing
$ loopy --halt now,success=1 ping -c 1 mysite.com; ./do_next_thing

$ ./create_big_file -o my_big_file.bin;
  loop 'ls' --until-contains 'my_big_file.bin';
  ./upload_big_file my_big_file.bin
# inotifywait is a better tool to detect file system changes.
# It can even make sure the file is complete
# so you are not uploading an incomplete file
$ inotifywait -qmre MOVED_TO -e CLOSE_WRITE --format %w%f . |
    grep my_big_file.bin

$ ls | loop 'cp $ITEM $ITEM.bak'
$ ls | parallel cp {} {}.bak

$ loop './do_thing.sh' --every 15s --until-success --num 5
$ parallel --retries 5 --delay 15s ::: ./do_thing.sh
```

https://github.com/Miserlou/Loop/ (Last checked: 2018-10)

## DIFFERENCES BETWEEN lorikeet AND GNU Parallel

**lorikeet** can run jobs in parallel. It does this based on a dependency graph described in a file, so this is similar to **make**.

https://github.com/cetra3/lorikeet (Last checked: 2018-10)

## DIFFERENCES BETWEEN spp AND GNU Parallel

**spp** can run jobs in parallel. **spp** does not use a command template to generate the jobs, but requires jobs to be in a file. Output from the jobs mix.

https://github.com/john01dav/spp (Last checked: 2024-06)

## DIFFERENCES BETWEEN paral AND GNU Parallel

**paral** prints a lot of status information and stores the output from the commands run into files. This means it cannot be used the middle of a pipe like this

```
paral "echo this" "echo does not" "echo work" | wc
```

Instead it puts the output into files named like **out_#_*command*.out.log**. To get a very similar behaviour with GNU **parallel** use **--results 'out_{#}_{=s/[^\sa-z_0-9]//g;s/\s+/_/g=}.log' --eta**

**paral** only takes arguments on the command line and each argument should be a full command. Thus it does not use command templates.

This limits how many jobs it can run in total, because they all need to fit on a single command line.

**paral** has no support for running jobs remotely.

### EXAMPLES FROM README.markdown

The examples from **README.markdown** and the corresponding command run with GNU **parallel** ( **--results 'out_{#}_{=s/[^\sa-z_0-9]//g;s/\s+/_/g=}.log' --eta** is omitted from the GNU **parallel** command):

```
1$ paral "command 1" "command 2 --flag" "command arg1 arg2"
1$ parallel ::: "command 1" "command 2 --flag" "command arg1 arg2"


2$ paral "sleep 1 && echo c1" "sleep 2 && echo c2" \
     "sleep 3 && echo c3" "sleep 4 && echo c4"  "sleep 5 && echo c5"
2$ parallel ::: "sleep 1 && echo c1" "sleep 2 && echo c2" \
     "sleep 3 && echo c3" "sleep 4 && echo c4"  "sleep 5 && echo c5"
   # Or shorter:
   parallel "sleep {} && echo c{}" ::: {1..5}


3$ paral -n=0 "sleep 5 && echo c5" "sleep 4 && echo c4" \
     "sleep 3 && echo c3" "sleep 2 && echo c2" "sleep 1 && echo c1"
3$ parallel ::: "sleep 5 && echo c5" "sleep 4 && echo c4" \
     "sleep 3 && echo c3" "sleep 2 && echo c2" "sleep 1 && echo c1"
   # Or shorter:
   parallel -j0 "sleep {} && echo c{}" ::: 5 4 3 2 1


4$ paral -n=1 "sleep 5 && echo c5" "sleep 4 && echo c4" \
     "sleep 3 && echo c3" "sleep 2 && echo c2" "sleep 1 && echo c1"
4$ parallel -j1 "sleep {} && echo c{}" ::: 5 4 3 2 1


5$ paral -n=2 "sleep 5 && echo c5" "sleep 4 && echo c4" \
     "sleep 3 && echo c3" "sleep 2 && echo c2" "sleep 1 && echo c1"
5$ parallel -j2 "sleep {} && echo c{}" ::: 5 4 3 2 1


6$ paral -n=5 "sleep 5 && echo c5" "sleep 4 && echo c4" \
     "sleep 3 && echo c3" "sleep 2 && echo c2" "sleep 1 && echo c1"
6$ parallel -j5 "sleep {} && echo c{}" ::: 5 4 3 2 1


7$ paral -n=1 "echo a && sleep 0.5 && echo b && sleep 0.5 && \
     echo c && sleep 0.5 && echo d && sleep 0.5 && \
     echo e && sleep 0.5 && echo f && sleep 0.5 && \
     echo g && sleep 0.5 && echo h"
7$ parallel ::: "echo a && sleep 0.5 && echo b && sleep 0.5 && \
     echo c && sleep 0.5 && echo d && sleep 0.5 && \
     echo e && sleep 0.5 && echo f && sleep 0.5 && \
     echo g && sleep 0.5 && echo h"
```

https://github.com/amattn/paral (Last checked: 2024-06)

## DIFFERENCES BETWEEN concurr AND GNU Parallel

**concurr** is built to run jobs in parallel using a client/server model.

### EXAMPLES FROM README.md

The examples from **README.md**:

```
1$ concurr 'echo job {#} on slot {%}: {}' : arg1 arg2 arg3 arg4
```

```
1$ parallel 'echo job {#} on slot {%}: {}' ::: arg1 arg2 arg3 arg4

2$ concurr 'echo job {#} on slot {%}: {}' :: file1 file2 file3
2$ parallel 'echo job {#} on slot {%}: {}' :::: file1 file2 file3

3$ concurr 'echo {}' < input_file
3$ parallel 'echo {}' < input_file

4$ cat file | concurr 'echo {}'
4$ cat file | parallel 'echo {}'
```

**concurr** deals badly empty input files and with output larger than 64 KB.

https://github.com/mmstick/concurr (Last checked: 2024-01)

## DIFFERENCES BETWEEN lesser-parallel AND GNU Parallel

**lesser-parallel** is the inspiration for **parallel --embed**. Both **lesser-parallel** and **parallel --embed** define bash functions that can be included as part of a bash script to run jobs in parallel.

**lesser-parallel** implements a few of the replacement strings, but hardly any options, whereas **parallel --embed** gives you the full GNU **parallel** experience.

https://github.com/kou1okada/lesser-parallel (Last checked: 2024-06)

## DIFFERENCES BETWEEN npm-parallel AND GNU Parallel

**npm-parallel** can run npm tasks in parallel.

There are no examples and very little documentation, so it is hard to compare to GNU **parallel**.

https://github.com/spion/npm-parallel (Last checked: 2024-06)

## DIFFERENCES BETWEEN machma AND GNU Parallel

**machma** runs tasks in parallel. It gives time stamped output. It buffers in RAM.

### EXAMPLES FROM README.md

The examples from README.md:

```
1$ # Put shorthand for timestamp in config for the examples
   echo '--rpl '\
     \''{time} $_=::strftime("%Y-%m-%d %H:%M:%S",localtime())'\' \
     > ~/.parallel/machma
   echo '--line-buffer --tagstring "{#} {time} {}"' \
     >> ~/.parallel/machma

2$ find . -iname '*.jpg' |
     machma --  mogrify -resize 1200x1200 -filter Lanczos {}
   find . -iname '*.jpg' |
     parallel --bar -Jmachma mogrify -resize 1200x1200 \
       -filter Lanczos {}

3$ cat /tmp/ips | machma -p 2 -- ping -c 2 -q {}
3$ cat /tmp/ips | parallel -j2 -Jmachma ping -c 2 -q {}

4$ cat /tmp/ips |
     machma -- sh -c 'ping -c 2 -q $0 > /dev/null && echo alive' {}
4$ cat /tmp/ips |
     parallel -Jmachma 'ping -c 2 -q {} > /dev/null && echo alive'
```

```
5$ find . -iname '*.jpg' |
     machma --timeout 5s -- mogrify -resize 1200x1200 \
       -filter Lanczos {}
5$ find . -iname '*.jpg' |
     parallel --timeout 5s --bar mogrify -resize 1200x1200 \
       -filter Lanczos {}

6$ find . -iname '*.jpg' -print0 |
     machma --null --  mogrify -resize 1200x1200 -filter Lanczos {}
6$ find . -iname '*.jpg' -print0 |
     parallel --null --bar mogrify -resize 1200x1200 \
       -filter Lanczos {}
```

https://github.com/fd0/machma (Last checked: 2019-06)

## DIFFERENCES BETWEEN interlace AND GNU Parallel

Summary (see legend above):

- I2 I3 I4 - - -

M1 - M3 - - M6

- O2 O3 - - - - x x

E1 E2 - - - - -

- - - - - - - - -

- -

**interlace** is built for network analysis to run network tools in parallel.

**interface** does not buffer output, so output from different jobs mixes.

The overhead for each target is O(n*n), so with 1000 targets it becomes very slow with an overhead in the order of 500ms/target.

### EXAMPLES FROM interlace's WEBSITE

Using **prips** most of the examples from https://github.com/codingo/Interlace can be run with GNU **parallel**:

Blocker

```
commands.txt:
  mkdir -p _output_/_target_/scans/
  _blocker_
  nmap _target_ -oA _output_/_target_/scans/_target_-nmap
interlace -tL ./targets.txt -cL commands.txt -o $output

parallel -a targets.txt \
  mkdir -p $output/{}/scans/\; nmap {} -oA $output/{}/scans/{}-nmap
```

Blocks

```
commands.txt:
  _block:nmap_
  mkdir -p _target_/output/scans/
  nmap _target_ -oN _target_/output/scans/_target_-nmap
  _block:nmap_
  nikto --host _target_
interlace -tL ./targets.txt -cL commands.txt
```

```
_nmap() {
  mkdir -p $1/output/scans/
  nmap $1 -oN $1/output/scans/$1-nmap
}
export -f _nmap
parallel ::: _nmap "nikto --host" :::: targets.txt
```

Run Nikto Over Multiple Sites

```
interlace -tL ./targets.txt -threads 5 \
  -c "nikto --host _target_ > ./_target_-nikto.txt" -v

parallel -a targets.txt -P5 nikto --host {} \> ./{}_-nikto.txt
```

Run Nikto Over Multiple Sites and Ports

```
interlace -tL ./targets.txt -threads 5 -c \
  "nikto --host _target_:_port_ > ./_target_-_port_-nikto.txt" \
  -p 80,443 -v

parallel -P5 nikto --host {1}:{2} \> ./{1}-{2}-nikto.txt \
  :::: targets.txt ::: 80 443
```

Run a List of Commands against Target Hosts

```
commands.txt:
  nikto --host _target_:_port_ > _output_/_target_-nikto.txt
  sslscan _target_:_port_ >  _output_/_target_-sslscan.txt
  testssl.sh _target_:_port_ > _output_/_target_-testssl.txt
interlace -t example.com -o ~/Engagements/example/ \
  -cL ./commands.txt -p 80,443

parallel --results ~/Engagements/example/{2}:{3}{1} {1} {2}:{3} \
  ::: "nikto --host" sslscan testssl.sh ::: example.com ::: 80 443
```

CIDR notation with an application that doesn't support it

```
interlace -t 192.168.12.0/24 -c "vhostscan _target_ \
  -oN _output_/_target_-vhosts.txt" -o ~/scans/ -threads 50

prips 192.168.12.0/24 |
  parallel -P50 vhostscan {} -oN ~/scans/{}-vhosts.txt
```

Glob notation with an application that doesn't support it

```
interlace -t 192.168.12.* -c "vhostscan _target_ \
  -oN _output_/_target_-vhosts.txt" -o ~/scans/ -threads 50

# Glob is not supported in prips
prips 192.168.12.0/24 |
  parallel -P50 vhostscan {} -oN ~/scans/{}-vhosts.txt
```

Dash (-) notation with an application that doesn't support it

```
interlace -t 192.168.12.1-15 -c \
```

```
    "vhostscan _target_ -oN _output_/_target_-vhosts.txt" \
    -o ~/scans/ -threads 50


  # Dash notation is not supported in prips
  prips 192.168.12.1 192.168.12.15 |
    parallel -P50 vhostscan {} -oN ~/scans/{}-vhosts.txt
```

Threading Support for an application that doesn't support it

```
  interlace -tL ./target-list.txt -c \
    "vhostscan -t _target_ -oN _output_/_target_-vhosts.txt" \
    -o ~/scans/ -threads 50


  cat ./target-list.txt |
    parallel -P50 vhostscan -t {} -oN ~/scans/{}-vhosts.txt
```

alternatively

```
  ./vhosts-commands.txt:
    vhostscan -t $target -oN _output_/_target_-vhosts.txt
  interlace -cL ./vhosts-commands.txt -tL ./target-list.txt \
    -threads 50 -o ~/scans


  ./vhosts-commands.txt:
    vhostscan -t "$1" -oN "$2"
  parallel -P50 ./vhosts-commands.txt {} ~/scans/{}-vhosts.txt \
    :::: ./target-list.txt
```

Exclusions

```
  interlace -t 192.168.12.0/24 -e 192.168.12.0/26 -c \
    "vhostscan _target_ -oN _output_/_target_-vhosts.txt" \
    -o ~/scans/ -threads 50


  prips 192.168.12.0/24 | grep -xv -Ff <(prips 192.168.12.0/26) |
    parallel -P50 vhostscan {} -oN ~/scans/{}-vhosts.txt
```

Run Nikto Using Multiple Proxies

```
   interlace -tL ./targets.txt -pL ./proxies.txt -threads 5 -c \
     "nikto --host _target_:_port_ -useproxy _proxy_ > \
      ./_target_-_port_-nikto.txt" -p 80,443 -v


   parallel -j5 \
     "nikto --host {1}:{2} -useproxy {3} > ./{1}-{2}-nikto.txt" \
     :::: ./targets.txt ::: 80 443 :::: ./proxies.txt
```

https://github.com/codingo/Interlace (Last checked: 2019-09)

## DIFFERENCES BETWEEN otonvm Parallel AND GNU Parallel

I have been unable to get the code to run at all. It seems unfinished.

https://github.com/otonvm/Parallel (Last checked: 2024-06)

## DIFFERENCES BETWEEN k-bx par AND GNU Parallel

**par** requires Haskell to work. This limits the number of platforms this can work on.

**par** does line buffering in memory. The memory usage is 3x the longest line (compared to 1x for **parallel --lb**). Commands must be given as arguments. There is no template.

These are the examples from https://github.com/k-bx/par with the corresponding GNU **parallel** command.

```
par "echo foo; sleep 1; echo foo; sleep 1; echo foo" \
    "echo bar; sleep 1; echo bar; sleep 1; echo bar" && echo "success"
parallel --lb ::: "echo foo; sleep 1; echo foo; sleep 1; echo foo" \
    "echo bar; sleep 1; echo bar; sleep 1; echo bar" && echo "success"


par "echo foo; sleep 1; foofoo" \
    "echo bar; sleep 1; echo bar; sleep 1; echo bar" && echo "success"
parallel --lb --halt 1 ::: "echo foo; sleep 1; foofoo" \
    "echo bar; sleep 1; echo bar; sleep 1; echo bar" && echo "success"


par "PARPREFIX=[fooechoer] echo foo" "PARPREFIX=[bar] echo bar"
parallel --lb --colsep , --tagstring {1} {2} \
  ::: "[fooechoer],echo foo" "[bar],echo bar"


par --succeed "foo" "bar" && echo 'wow'
parallel "foo" "bar"; true && echo 'wow'
```

https://github.com/k-bx/par (Last checked: 2019-02)

## DIFFERENCES BETWEEN parallelshell AND GNU Parallel

**parallelshell** does not allow for composed commands:

```
# This does not work
parallelshell 'echo foo;echo bar' 'echo baz;echo quuz'
```

Instead you have to wrap that in a shell:

```
parallelshell 'sh -c "echo foo;echo bar"' 'sh -c "echo baz;echo quuz"'
```

It buffers output in RAM. All commands must be given on the command line and all commands are started in parallel at the same time. This will cause the system to freeze if there are so many jobs that there is not enough memory to run them all at the same time.

https://github.com/keithamus/parallelshell (Last checked: 2019-02)

https://github.com/darkguy2008/parallelshell (Last checked: 2019-03)

## DIFFERENCES BETWEEN shell-executor AND GNU Parallel

**shell-executor** does not allow for composed commands:

```
# This does not work
sx 'echo foo;echo bar' 'echo baz;echo quuz'
```

Instead you have to wrap that in a shell:

```
sx 'sh -c "echo foo;echo bar"' 'sh -c "echo baz;echo quuz"'
```

It buffers output in RAM. All commands must be given on the command line and all commands are

started in parallel at the same time. This will cause the system to freeze if there are so many jobs that there is not enough memory to run them all at the same time.

https://github.com/royriojas/shell-executor (Last checked: 2024-06)

## DIFFERENCES BETWEEN non-GNU par AND GNU Parallel

**par** buffers in memory to avoid mixing of jobs. It takes 1s per 1 million output lines.

**par** needs to have all commands before starting the first job. The jobs are read from stdin (standard input) so any quoting will have to be done by the user.

Stdout (standard output) is prepended with o:. Stderr (standard error) is sendt to stdout (standard output) and prepended with e:.

For short jobs with little output **par** is 20% faster than GNU **parallel** and 60% slower than **xargs**.

https://github.com/UnixJunkie/PAR

https://savannah.nongnu.org/projects/par (Last checked: 2019-02)

## DIFFERENCES BETWEEN fd AND GNU Parallel

**fd** does not support composed commands, so commands must be wrapped in **sh -c**.

It buffers output in RAM.

It only takes file names from the filesystem as input (similar to **find**).

https://github.com/sharkdp/fd (Last checked: 2019-02)

## DIFFERENCES BETWEEN lateral AND GNU Parallel

**lateral** is very similar to **sem**: It takes a single command and runs it in the background. The design means that output from parallel running jobs may mix. If it dies unexpectly it leaves a socket in ~/.lateral/socket.PID.

**lateral** deals badly with too long command lines. This makes the **lateral** server crash:

```
lateral run echo `seq 100000| head -c 1000k`
```

Any options will be read by **lateral** so this does not work (**lateral** interprets the **-l**):

```
lateral run ls -l
```

Composed commands do not work:

```
lateral run pwd ';' ls
```

Functions do not work:

```
myfunc() { echo a; }
export -f myfunc
lateral run myfunc
```

Running **emacs** in the terminal causes the parent shell to die:

```
echo '#!/bin/bash' > mycmd
echo emacs -nw >> mycmd
chmod +x mycmd
lateral start
lateral run ./mycmd
```

Here are the examples from https://github.com/akramer/lateral with the corresponding GNU **sem** and GNU **parallel** commands:

```
1$ lateral start
   for i in $(cat /tmp/names); do
     lateral run -- some_command $i
   done
   lateral wait

1$ for i in $(cat /tmp/names); do
     sem some_command $i
   done
   sem --wait

1$ parallel some_command :::: /tmp/names

2$ lateral start
   for i in $(seq 1 100); do
     lateral run -- my_slow_command < workfile$i > /tmp/logfile$i
   done
   lateral wait

2$ for i in $(seq 1 100); do
     sem my_slow_command < workfile$i > /tmp/logfile$i
   done
   sem --wait

2$ parallel 'my_slow_command < workfile{} > /tmp/logfile{}' \
     ::: {1..100}

3$ lateral start -p 0 # yup, it will just queue tasks
   for i in $(seq 1 100); do
     lateral run -- command_still_outputs_but_wont_spam inputfile$i
   done
   # command output spam can commence
   lateral config -p 10; lateral wait

3$ for i in $(seq 1 100); do
     echo "command inputfile$i" >> joblist
   done
   parallel -j 10 :::: joblist

3$ echo 1 > /tmp/njobs
   parallel -j /tmp/njobs command inputfile{} \
     ::: {1..100} &
   echo 10 >/tmp/njobs
   wait
```

https://github.com/akramer/lateral (Last checked: 2024-06)

## DIFFERENCES BETWEEN with-this AND GNU Parallel

The examples from https://github.com/amritb/with-this.git and the corresponding GNU **parallel** command:

```
with -v "$(cat myurls.txt)" "curl -L this"
```

```
parallel curl -L ::: myurls.txt

with -v "$(cat myregions.txt)" \
  "aws --region=this ec2 describe-instance-status"
parallel aws --region={} ec2 describe-instance-status \
  :::: myregions.txt

with -v "$(ls)" "kubectl --kubeconfig=this get pods"
ls | parallel kubectl --kubeconfig={} get pods

with -v "$(ls | grep config)" "kubectl --kubeconfig=this get pods"
ls | grep config | parallel kubectl --kubeconfig={} get pods

with -v "$(echo {1..10})" "echo 123"
parallel -N0 echo 123 ::: {1..10}
```

Stderr is merged with stdout. **with-this** buffers in RAM. It uses 3x the output size, so you cannot have output larger than 1/3rd the amount of RAM. The input values cannot contain spaces. Composed commands do not work.

**with-this** gives some additional information, so the output has to be cleaned before piping it to the next command.

https://github.com/amritb/with-this.git (Last checked: 2024-06)

## DIFFERENCES BETWEEN Tollef's parallel (moreutils) AND GNU Parallel

Summary (see legend above):

- - - I4 - - I7

- - M3 - - M6

- O2 O3 - O5 O6 - x x

E1 - - - - - E7

- x x x x x x x

- -

### EXAMPLES FROM Tollef's parallel MANUAL

**Tollef** parallel sh -c "echo hi; sleep 2; echo bye" -- 1 2 3

**GNU** parallel "echo hi; sleep 2; echo bye" ::: 1 2 3

**Tollef** parallel -j 3 ufraw -o processed -- *.NEF

**GNU** parallel -j 3 ufraw -o processed ::: *.NEF

**Tollef** parallel -j 3 -- ls df "echo hi"

**GNU** parallel -j 3 ::: ls df "echo hi"

(Last checked: 2019-08)

## DIFFERENCES BETWEEN rargs AND GNU Parallel

Summary (see legend above):

I1 - - - - - I7

- - M3 M4 - -

- O2 O3 - O5 O6 - O8 -

```
E1 - - E4 - - -
- - - - - - - - -
- -
```

**rargs** has elegant ways of doing named regexp capture and field ranges.

With GNU **parallel** you can use **--rpl** to get a similar functionality as regexp capture gives, and use **join** and **@arg** to get the field ranges. But the syntax is longer. This:

```
--rpl '{r(\d+)\.\.(\d+)} $_=join"$opt::colsep",@arg[$$1..$$2]'
```

would make it possible to use:

```
{1r3..6}
```

for field 3..6.

For full support of {n..m:s} including negative numbers use a dynamic replacement string like this:

```
PARALLEL=--rpl\ \''{r((-?\d+)?)\.\.((-?\d+)?)((:([^}]*))?)}
        $a = defined $$2 ? $$2 < 0 ? 1+$#arg+$$2 : $$2 : 1;
        $b = defined $$4 ? $$4 < 0 ? 1+$#arg+$$4 : $$4 : $#arg+1;
        $s = defined $$6 ? $$7 : " ";
        $_ = join $s,@arg[$a..$b]'\'
export PARALLEL
```

You can then do:

```
head /etc/passwd | parallel --colsep : echo ..={1r..} ..3={1r..3} \
  4..={1r4..} 2..4={1r2..4} 3..3={1r3..3} ..3:-={1r..3:-} \
  ..3:/={1r..3:/} -1={-1} -5={-5} -6={-6} -3..={1r-3..}
```

**EXAMPLES FROM rargs MANUAL**

```
1$ ls *.bak | rargs -p '(.*)\.bak' mv {0} {1}

1$ ls *.bak | parallel mv {} {.}

2$ cat download-list.csv |
     rargs -p '(?P<url>.*),(?P<filename>.*)' wget {url} -O {filename}

2$ cat download-list.csv |
     parallel --csv wget {1} -O {2}
# or use regexps:
2$ cat download-list.csv |
     parallel --rpl '{url} s/,.*//' --rpl '{filename} s/.*?,//' \
       wget {url} -O {filename}

3$ cat /etc/passwd |
     rargs -d: echo -e 'id: "{1}"\t name: "{5}"\t rest: "{6..::}"'

3$ cat /etc/passwd |
     parallel -q --colsep : \
       echo -e 'id: "{1}"\t name: "{5}"\t rest: "{=6
$_=join":",@arg[6..$#arg]=}"'
```

https://github.com/lotabout/rargs (Last checked: 2020-01)

## DIFFERENCES BETWEEN threader AND GNU Parallel

Summary (see legend above):

I1 - - - - - -

M1 - M3 - - M6

O1 - O3 - O5 - - x x

E1 - - E4 - - -

- - - - - - - - -

- -

Newline separates arguments, but newline at the end of file is treated as an empty argument. So this runs 2 jobs:

```
echo two_jobs | threader -run 'echo "$THREADID"'
```

**threader** ignores stderr, so any output to stderr is lost. **threader** buffers in RAM, so output bigger than the machine's virtual memory will cause the machine to crash.

https://github.com/voodooEntity/threader (Last checked: 2024-06)

## DIFFERENCES BETWEEN runp AND GNU Parallel

Summary (see legend above):

I1 I2 - - - - -

M1 - (M3) - - M6

O1 O2 O3 - O5 O6 - x x -

E1 - - - - - -

- - - - - - - - -

- -

(M3): You can add a prefix and a postfix to the input, so it means you can only insert the argument on the command line once.

**runp** runs 10 jobs in parallel by default. **runp** blocks if output of a command is > 64 Kbytes. Quoting of input is needed. It adds output to stderr (this can be prevented with -q)

### Examples as GNU Parallel

```
base='https://images-api.nasa.gov/search'
query='jupiter'
desc='planet'
type='image'
url="$base?q=$query&description=$desc&media_type=$type"

# Download the images in parallel using runp
curl -s $url | jq -r .collection.items[].href | \
  runp -p 'curl -s' | jq -r .[] | grep large | \
  runp -p 'curl -s -L -O'

time curl -s $url | jq -r .collection.items[].href | \
  runp -g 1 -q -p 'curl -s' | jq -r .[] | grep large | \
  runp -g 1 -q -p 'curl -s -L -O'

# Download the images in parallel
```

```
curl -s $url | jq -r .collection.items[].href | \
  parallel curl -s | jq -r .[] | grep large | \
  parallel curl -s -L -O

time curl -s $url | jq -r .collection.items[].href | \
  parallel -j 1 curl -s | jq -r .[] | grep large | \
  parallel -j 1 curl -s -L -O
```

Run some test commands (read from file)

```
# Create a file containing commands to run in parallel.
cat << EOF > /tmp/test-commands.txt
sleep 5
sleep 3
blah     # this will fail
ls $PWD  # PWD shell variable is used here
EOF

# Run commands from the file.
runp /tmp/test-commands.txt > /dev/null

parallel -a /tmp/test-commands.txt > /dev/null
```

Ping several hosts and see packet loss (read from stdin)

```
# First copy this line and press Enter
runp -p 'ping -c 5 -W 2' -s '| grep loss'
localhost
1.1.1.1
8.8.8.8
# Press Enter and Ctrl-D when done entering the hosts

# First copy this line and press Enter
parallel ping -c 5 -W 2 {} '| grep loss'
localhost
1.1.1.1
8.8.8.8
# Press Enter and Ctrl-D when done entering the hosts
```

Get directories' sizes (read from stdin)

```
echo -e "$HOME\n/etc\n/tmp" | runp -q -p 'sudo du -sh'

echo -e "$HOME\n/etc\n/tmp" | parallel sudo du -sh
# or:
parallel sudo du -sh ::: "$HOME" /etc /tmp
```

Compress files

```
find . -iname '*.txt' | runp -p 'gzip --best'

find . -iname '*.txt' | parallel gzip --best
```

Measure HTTP request + response time

```
export CURL="curl -w 'time_total:  %{time_total}\n'"
CURL="$CURL -o /dev/null -s https://golang.org/"
```

```
perl -wE 'for (1..10) { say $ENV{CURL} }' |
   runp -q  # Make 10 requests

perl -wE 'for (1..10) { say $ENV{CURL} }' | parallel
# or:
parallel -N0 "$CURL" ::: {1..10}
```

Find open TCP ports

```
cat << EOF > /tmp/host-port.txt
localhost 22
localhost 80
localhost 81
127.0.0.1 443
127.0.0.1 444
scanme.nmap.org 22
scanme.nmap.org 23
scanme.nmap.org 443
EOF

1$ cat /tmp/host-port.txt |
     runp -q -p 'netcat -v -w2 -z' 2>&1 | egrep '(succeeded!|open)$'

# --colsep is needed to split the line
1$ cat /tmp/host-port.txt |
     parallel --colsep ' ' netcat -v -w2 -z 2>&1 |
     egrep '(succeeded!|open)$'
# or use uq for unquoted:
1$ cat /tmp/host-port.txt |
     parallel netcat -v -w2 -z {=uq=} 2>&1 |
     egrep '(succeeded!|open)$'
```

https://github.com/jreisinger/runp (Last checked: 2020-04)

## DIFFERENCES BETWEEN papply AND GNU Parallel

Summary (see legend above):

- - - I4 - - -

M1 - M3 - - M6

- - O3 - O5 - - x x O10

E1 - - E4 - - -

- - - - - - - - -

- -

**papply** does not print the output if the command fails:

```
$ papply 'echo %F; false' foo
"echo foo; false" did not succeed
```

**papply**'s replacement strings (%F %d %f %n %e %z) can be simulated in GNU **parallel** by putting this in **~/.parallel/config**:

```
--rpl '%F'
--rpl '%d $_=Q(::dirname($_));'
--rpl '%f s:.*/::;'
```

```
--rpl '%n s:.*/::;s:\.[^/.]+$::;'
--rpl '%e s:.*\.::'
--rpl '%z $_=""'
```

**papply** buffers in RAM, and uses twice the amount of output. So output of 5 GB takes 10 GB RAM.

The buffering is very CPU intensive: Buffering a line of 5 GB takes 40 seconds (compared to 10 seconds with GNU **parallel**).

**Examples as GNU Parallel**

```
1$ papply gzip *.txt

1$ parallel gzip ::: *.txt

2$ papply "convert %F %n.jpg" *.png

2$ parallel convert {} {.}.jpg ::: *.png
```

https://pypi.org/project/papply/ (Last checked: 2020-04)

## DIFFERENCES BETWEEN async AND GNU Parallel

Summary (see legend above):

- - - I4 - - I7

- - - - - M6

- O2 O3 - O5 O6 - x x O10

E1 - - E4 - E6 -

- - - - - - - - -

S1 S2

**async** is very similary to GNU **parallel**'s **--semaphore** mode (aka **sem**). **async** requires the user to start a server process.

The input is quoted like **-q** so you need **bash -c "...;..."** to run composed commands.

**Examples as GNU Parallel**

```
1$ S="/tmp/example_socket"

1$ ID=myid

2$ async -s="$S" server --start

2$ # GNU Parallel does not need a server to run

3$ for i in {1..20}; do
       # prints command output to stdout
       async -s="$S" cmd -- bash -c "sleep 1 && echo test $i"
   done

3$ for i in {1..20}; do
       # prints command output to stdout
       sem --id "$ID" -j100% "sleep 1 && echo test $i"
       # GNU Parallel will only print job when it is done
```

```
        # If you need output from different jobs to mix
        # use -u or --line-buffer
        sem --id "$ID" -j100% --line-buffer "sleep 1 && echo test $i"
    done

4$ # wait until all commands are finished
    async -s="$S" wait

4$ sem --id "$ID" --wait

5$ # configure the server to run four commands in parallel
    async -s="$S" server -j4

5$ export PARALLEL=-j4

6$ mkdir "/tmp/ex_dir"
    for i in {21..40}; do
      # redirects command output to /tmp/ex_dir/file*
      async -s="$S" cmd -o "/tmp/ex_dir/file$i" -- \
        bash -c "sleep 1 && echo test $i"
    done

6$ mkdir "/tmp/ex_dir"
    for i in {21..40}; do
      # redirects command output to /tmp/ex_dir/file*
      sem --id "$ID" --result '/tmp/my-ex/file-{=$_=""=}'"$i" \
        "sleep 1 && echo test $i"
    done

7$ sem --id "$ID" --wait

7$ async -s="$S" wait

8$ # stops server
    async -s="$S" server --stop

8$ # GNU Parallel does not need to stop a server
```

https://github.com/ctbur/async/ (Last checked: 2024-06)

## DIFFERENCES BETWEEN pardi AND GNU Parallel

Summary (see legend above):

I1 I2 - - - - I7

M1 - - - - M6

O1 O2 O3 O4 O5 - O7 - - O10

E1 - - E4 - - -

- - - - - - - - -

- -

**pardi** is very similar to **parallel --pipe --cat**: It reads blocks of data and not arguments. So it cannot insert an argument in the command line. It puts the block into a temporary file, and this file name

(%IN) can be put in the command line. You can only use %IN once.

It can also run full command lines in parallel (like: **cat file | parallel**).

**EXAMPLES FROM pardi test.sh**

```
1$ time pardi -v -c 100 -i data/decoys.smi -ie .smi -oe .smi \
     -o data/decoys_std_pardi.smi \
        -w '(standardiser -i %IN -o %OUT 2>&1) > /dev/null'

1$ cat data/decoys.smi |
     time parallel -N 100 --pipe --cat \
       '(standardiser -i {} -o {#} 2>&1) > /dev/null; cat {#}; rm {#}' \
       > data/decoys_std_pardi.smi

2$ pardi -n 1 -i data/test_in.types -o data/test_out.types \
          -d 'r:^#atoms:' -w 'cat %IN > %OUT'

2$ cat data/test_in.types |
     parallel -n 1 -k --pipe --cat --regexp --recstart '^#atoms' \
       'cat {}' > data/test_out.types

3$ pardi -c 6 -i data/test_in.types -o data/test_out.types \
          -d 'r:^#atoms:' -w 'cat %IN > %OUT'

3$ cat data/test_in.types |
     parallel -n 6 -k --pipe --cat --regexp --recstart '^#atoms' \
       'cat {}' > data/test_out.types

4$ pardi -i data/decoys.mol2 -o data/still_decoys.mol2 \
          -d 's:@<TRIPOS>MOLECULE' -w 'cp %IN %OUT'

4$ cat data/decoys.mol2 |
     parallel -n 1 --pipe --cat --recstart '@<TRIPOS>MOLECULE' \
       'cp {} {#}; cat {#}; rm {#}' > data/still_decoys.mol2

5$ pardi -i data/decoys.mol2 -o data/decoys2.mol2 \
          -d b:10000 -w 'cp %IN %OUT' --preserve

5$ cat data/decoys.mol2 |
     parallel -k --pipe --block 10k --recend '' --cat \
       'cat {} > {#}; cat {#}; rm {#}' > data/decoys2.mol2
```

https://github.com/UnixJunkie/pardi (Last checked: 2021-01)

**DIFFERENCES BETWEEN bthread AND GNU Parallel**

Summary (see legend above):

- - - I4 - - -

- - - - - M6

O1 - O3 - - - O7 O8 - -

E1 - - - - - -

- - - - - - - - -

- -

**bthread** takes around 1 sec per MB of output. The maximal output line length is 1073741759.

You cannot quote space in the command, so you cannot run composed commands like **sh -c "echo a; echo b"**.

https://gitlab.com/netikras/bthread (Last checked: 2024-06)

## DIFFERENCES BETWEEN simple_gpu_scheduler AND GNU Parallel

Summary (see legend above):

I1 - - - - - I7

M1 - - - - M6

- O2 O3 - - O6 - x x O10

E1 - - - - - -

- - - - - - - - -

- -

### EXAMPLES FROM simple_gpu_scheduler MANUAL

```
1$ simple_gpu_scheduler --gpus 0 1 2 < gpu_commands.txt

1$ parallel -j3 --shuf \
   CUDA_VISIBLE_DEVICES='{=1 $_=slot()-1 =} {=uq;=}' \
     < gpu_commands.txt

2$ simple_hypersearch \
     "python3 train_dnn.py --lr {lr} --batch_size {bs}" \
     -p lr 0.001 0.0005 0.0001 -p bs 32 64 128 |
     simple_gpu_scheduler --gpus 0,1,2

2$ parallel --header : --shuf -j3 -v \
     CUDA_VISIBLE_DEVICES='{=1 $_=slot()-1 =}' \
     python3 train_dnn.py --lr {lr} --batch_size {bs} \
     ::: lr 0.001 0.0005 0.0001 ::: bs 32 64 128

3$ simple_hypersearch \
     "python3 train_dnn.py --lr {lr} --batch_size {bs}" \
     --n-samples 5 -p lr 0.001 0.0005 0.0001 -p bs 32 64 128 |
     simple_gpu_scheduler --gpus 0,1,2

3$ parallel --header : --shuf \
     CUDA_VISIBLE_DEVICES='{=1 $_=slot()-1; seq()>5 and skip() =}' \
     python3 train_dnn.py --lr {lr} --batch_size {bs} \
     ::: lr 0.001 0.0005 0.0001 ::: bs 32 64 128

4$ touch gpu.queue
   tail -f -n 0 gpu.queue | simple_gpu_scheduler --gpus 0,1,2 &
   echo "my_command_with | and stuff > logfile" >> gpu.queue

4$ touch gpu.queue
   tail -f -n 0 gpu.queue |
     parallel -j3 CUDA_VISIBLE_DEVICES='{=1 $_=slot()-1 =} {=uq;=}' &
   # Needed to fill job slots once
   seq 3 | parallel echo true >> gpu.queue
   # Add jobs
```

```
echo "my_command_with | and stuff > logfile" >> gpu.queue
# Needed to flush output from completed jobs
seq 3 | parallel echo true >> gpu.queue
```

https://github.com/ExpectationMax/simple_gpu_scheduler (Last checked: 2024-06)

## DIFFERENCES BETWEEN parasweep AND GNU Parallel

**parasweep** is a Python module for facilitating parallel parameter sweeps.

A **parasweep** job will normally take a text file as input. The text file contains arguments for the job. Some of these arguments will be fixed and some of them will be changed by **parasweep**.

It does this by having a template file such as template.txt:

```
Xval: {x}
Yval: {y}
FixedValue: 9
# x with 2 decimals
DecimalX: {x:.2f}
TenX: ${x*10}
RandomVal: {r}
```

and from this template it generates the file to be used by the job by replacing the replacement strings.

Being a Python module **parasweep** integrates tighter with Python than GNU **parallel**. You get the parameters directly in a Python data structure. With GNU **parallel** you can use the JSON or CSV output format to get something similar, but you would have to read the output.

**parasweep** has a filtering method to ignore parameter combinations you do not need.

Instead of calling the jobs directly, **parasweep** can use Python's Distributed Resource Management Application API to make jobs run with different cluster software.

GNU **parallel --tmpl** supports templates with replacement strings. Such as:

```
Xval: {x}
Yval: {y}
FixedValue: 9
# x with 2 decimals
DecimalX: {=x $_=sprintf("%.2f",$_) =}
TenX: {=x $_=$_*10 =}
RandomVal: {=1 $_=rand() =}
```

that can be used like:

```
parallel --header : --tmpl my.tmpl={#}.t myprog {#}.t \
  ::: x 1 2 3 ::: y 1 2 3
```

Filtering is supported as:

```
parallel --filter '{1} > {2}' echo ::: 1 2 3 ::: 1 2 3
```

https://github.com/eviatarbach/parasweep (Last checked: 2021-01)

## DIFFERENCES BETWEEN parallel-bash(2021) AND GNU Parallel

Summary (see legend above):

I1 I2 - - - - -

- - M3 - - M6

- O2 O3 - O5 O6 - O8 x O10

E1 - - - - - -

- - - - - - - - -

- -

**parallel-bash** is written in pure bash. It is really fast (overhead of ~0.05 ms/job compared to GNU **parallel**'s 3-10 ms/job). So if your jobs are extremely short lived, and you can live with the quite limited command, this may be useful.

It works by making a queue for each process. Then the jobs are distributed to the queues in a round robin fashion. Finally the queues are started in parallel. This works fine, if you are lucky, but if not, all the long jobs may end up in the same queue, so you may see:

```
$ printf "%b\n" 1 1 1 4 1 1 1 4 1 1 1 4 |
    time parallel -P4 sleep {}
(7 seconds)
$ printf "%b\n" 1 1 1 4 1 1 1 4 1 1 1 4 |
    time parallel-bash -p 4 -c sleep {}
(12 seconds)
```

Because it uses bash lists, the total number of jobs is limited to 167000..265000 depending on your environment. You get a segmentation fault, when you reach the limit.

Ctrl-C does not stop spawning new jobs. Ctrl-Z does not suspend running jobs.

### EXAMPLES FROM parallel-bash

```
1$ some_input | parallel-bash -p 5 -c echo

1$ some_input | parallel -j 5 echo

2$ parallel-bash -p 5 -c echo < some_file

2$ parallel -j 5 echo < some_file

3$ parallel-bash -p 5 -c echo <<< 'some string'

3$ parallel -j 5 -c echo <<< 'some string'

4$ something | parallel-bash -p 5 -c echo {} {}

4$ something | parallel -j 5 echo {} {}
```

https://github.com/Akianonymus/parallel-bash/ (Last checked: 2024-06)

### DIFFERENCES BETWEEN parallel-bash(2024) AND GNU Parallel

Summary (see legend above):

I1 I2 - - - - -

- - M3 - - M6

- O2 O3 - O5 O6 - O8 x O10

E1 - - - - - -

- - - - - - - - -

- -

**parallel-bash** is written in pure bash. It is really fast (overhead of ~0.05 ms/job compared to GNU **parallel**'s 3-10 ms/job). So if your jobs are extremely short lived, and you can live with the quite limited command, this may be useful.

It seems the number of jobs must be divisible by **-p**, so it sometimes does not run the jobs:

```
# Does nothing
$ seq 3 | parallel-bash -p 4 bash -c 'touch myfile-{}'
```

This should create myfile-1..3, but creates nothing.

It splits the input into queues. Each queue is of length **-p**. So this will make 250 queues and run all 250 processes in parallel:

```
$ seq 1000 | parallel-bash -p 4 bash -c 'sleep {}'
```

This is quite different from **parallel-bash**(2021) where **-p** is the number of workers - similar to **--jobs** in GNU **parallel**.

In other words: **parallel-bash** does *not* quarantee that only 4 jobs will be run in parallel. This can overload your machine:

```
# Warning: This will start 25000 processes - not just 4
$ seq 100000 | parallel-bash -p 4 sleep {}
```

If you are unlucky all long jobs may end up in the same queue:

```
$ printf "%b\n" 1 1 1 1 5 5 5 5 1 1 1 1 |
    time parallel -P4 sleep {}
(7 seconds)
$ printf "%b\n" 1 1 1 1 5 5 5 5 1 1 1 1 |
    time ./parallel-bash.bash -p 4 -c sleep {}
(20 seconds)
```

Ctrl-C kills the jobs (as expected). Ctrl-Z does not suspend running jobs.

**EXAMPLES FROM parallel-bash**

```
1$ main() { echo "${1}" ;}
   export -f main

1$ printf "%b\n" {1..1000} | ./parallel-bash -p 10 main {}

1$ printf "%b\n" {1..1000} | parallel -j 100 main {}

2$ # Number of inputs must be divisible by 5
   some_input | parallel-bash -p 5 echo

2$ some_input | parallel -j 5 echo

3$ # Number of inputs must be divisible by 5
   parallel-bash -p 5 echo < some_file

3$ parallel -j 5 echo < some_file

4$ # Number of lines in 'some string' must be divisible by 5
   parallel-bash -p 5 echo <<< 'some string'
```

```
4$ parallel -j 5 -c echo <<< 'some string'

5$ something | parallel-bash -p 5 echo {}

5$ something | parallel -j 5 echo {}
```

https://github.com/Akianonymus/parallel-bash/ (Last checked: 2024-06)

## DIFFERENCES BETWEEN bash-concurrent AND GNU Parallel

**bash-concurrent** is more an alternative to **make** than to GNU **parallel**. Its input is very similar to a Makefile, where jobs depend on other jobs.

It has a nice progress indicator where you can see which jobs completed successfully, which jobs are currently running, which jobs failed, and which jobs were skipped due to a depending job failed. The indicator does not deal well with resizing the window.

Output is cached in tempfiles on disk, but is only shown if there is an error, so it is not meant to be part of a UNIX pipeline. If **bash-concurrent** crashes these tempfiles are not removed.

It uses an O(n*n) algorithm, so if you have 1000 independent jobs it takes 22 seconds to start it.

https://github.com/themattrix/bash-concurrent (Last checked: 2024-06)

## DIFFERENCES BETWEEN spawntool AND GNU Parallel

Summary (see legend above):

I1 - - - - - -

M1 - - - - M6

- O2 O3 - O5 O6 - x x O10

E1 - - - - - -

- - - - - - - - -

- -

**spawn** reads a full command line from stdin which it executes in parallel.

http://code.google.com/p/spawntool/ (Last checked: 2021-07)

## DIFFERENCES BETWEEN go-pssh AND GNU Parallel

Summary (see legend above):

- - - - - - -

M1 - - - - -

O1 - - - - - - x x O10

E1 - - - - - -

R1 R2 - - - R6 - - -

- -

**go-pssh** does **ssh** in parallel to multiple machines. It runs the same command on multiple machines similar to **--nonall**.

The hostnames must be given as IP-addresses (not as hostnames).

Output is sent to stdout (standard output) if command is successful, and to stderr (standard error) if the command fails.

### EXAMPLES FROM go-pssh

```
  1$ go-pssh -l <ip>,<ip> -u <user> -p <port> -P <passwd> -c "<command>"

  1$ parallel -S 'sshpass -p <passwd> ssh -p <port> <user>@<ip>' \
       --nonall "<command>"

  2$ go-pssh scp -f host.txt -u <user> -p <port> -P <password> \
       -s /local/file_or_directory -d /remote/directory

  2$ parallel --nonall --slf host.txt \
       --basefile /local/file_or_directory/./ --wd /remote/directory
       --ssh 'sshpass -p <password> ssh -p <port> -l <user>' true

  3$ go-pssh scp -l <ip>,<ip> -u <user> -p <port> -P <password> \
       -s /local/file_or_directory -d /remote/directory

  3$ parallel --nonall -S <ip>,<ip> \
       --basefile /local/file_or_directory/./ --wd /remote/directory
       --ssh 'sshpass -p <password> ssh -p <port> -l <user>' true
```

https://github.com/xuchenCN/go-pssh (Last checked: 2021-07)

### DIFFERENCES BETWEEN go-parallel AND GNU Parallel

Summary (see legend above):

I1 I2 - - - - (I7)

- - M3 - - M6

- O2 O3 - O5 - - x x - O10

E1 - - E4 - - -

- - - - - - - - -

- -

**go-parallel** uses Go templates for replacement strings. Quite similar to the *{= perl expr =}* replacement string.

The basic replacement strings can be emulated by putting this into **~/.parallel/config**:

```
  --rpl '{{.Input}} '
  --rpl '{{.Time}} use DateTime; $_= DateTime->from_epoch(time);'
  --rpl '{{.Start}} use DateTime; $_= DateTime->from_epoch($^T);'
```

Then you can do:

```
  seq 10 | parallel sleep {{.Input}}';' echo {{.Start}} {{.Time}}
  seq 10 | go-parallel -t 'bash -c "sleep {{.Input}}; echo \"{{.Start}}\"
\"{{.Time}}\""'
```

If the input is too long (64K), you get no error:

```
   perl -e 'print "works."."x"x100' | parallel.go -t 'echo {{noExt .Input}}
 '
   perl -e 'print "fails."."x"x100_000_000' | parallel.go -t 'echo {{noExt
.Input}} '
```

Special chars are quoted:

```
echo '"&+<>' | go-parallel echo
"&+<>
```

but not shell quoted when using replacement strings:

```
echo '"&+<>' | go-parallel -t 'echo {{.Input}}'
&#34;&amp;&#43;&lt;&gt;
```

### EXAMPLES FROM go-parallel

```
  1$ go-parallel -a ./files.txt -t 'cp {{.Input}} {{.Input | dirname |
dirname}}'

  1$ parallel -a ./files.txt cp {} '{= $_=::dirname(::dirname($_)) =}'

  2$ go-parallel -a ./files.txt -t 'mkdir -p {{.Input}} {{noExt .Input}}'

  2$ parallel -a ./files.txt echo mkdir -p {} {.}

  3$ go-parallel -a ./files.txt -t 'mkdir -p {{.Input}} {{.Input | basename
 | noExt}}'

  3$ parallel -a ./files.txt echo mkdir -p {} {/.}

  4$ time find ~/src/go -type f | go-parallel md5sum > /dev/null

  4$ time find ~/src/go -type f | parallel md5sum > /dev/null
     # Though you would probably do this instead:
     time find ~/src/go -type f | parallel -X md5sum > /dev/null
```

https://github.com/mylanconnolly/parallel (Last checked: 2024-06)

## DIFFERENCES BETWEEN p AND GNU Parallel

Summary (see legend above):

- - - I4 - - x

- - - - - M6

- O2 O3 - O5 O6 - x x - O10

E1 - - - - - -

- - - - - - - - -

- -

**p** is a tiny shell script. It can color output with some predefined colors, but is otherwise quite limited.

It maxes out at around 116000 jobs (probably due to limitations in Bash).

### EXAMPLES FROM p

Some of the examples from **p** cannot be implemented 100% by GNU **parallel**: The coloring is a bit different, and GNU **parallel** cannot have **--tag** for some inputs and not for others.

The coloring done by GNU **parallel** is not exactly the same as **p**.

```
  1$ p -bc blue "ping 127.0.0.1" -uc red "ping 192.168.0.1" \
```

```
   -rc yellow "ping 192.168.1.1" -t example "ping example.com"

1$ parallel --lb -j0 --color --tag ping \
   ::: 127.0.0.1 192.168.0.1 192.168.1.1 example.com

2$ p "tail -f /var/log/httpd/access_log" \
   -bc red "tail -f /var/log/httpd/error_log"

2$ cd /var/log/httpd;
   parallel --lb --color --tag tail -f ::: access_log error_log

3$ p tail -f "some file" \& p tail -f "other file with space.txt"

3$ parallel --lb tail -f ::: 'some file' "other file with space.txt"

4$ p -t project1 "hg pull project1" -t project2 \
   "hg pull project2" -t project3 "hg pull project3"

4$ parallel --lb hg pull ::: project{1..3}
```

https://github.com/rudymatela/evenmoreutils/blob/master/man/p.1.adoc (Last checked: 2022-04)

## DIFFERENCES BETWEEN senechal AND GNU Parallel

Summary (see legend above):

I1 - - - - - -

M1 - M3 - - M6

O1 - O3 O4 - - - x x -

E1 - - - - - -

- - - - - - - - -

- -

**seneschal** only starts the first job after reading the last job, and output from the first job is only printed after the last job finishes.

1 byte of output requites 3.5 bytes of RAM.

This makes it impossible to have a total output bigger than the virtual memory.

Even though output is kept in RAM outputing is quite slow: 30 MB/s.

Output larger than 4 GB causes random problems - it looks like a race condition.

This:

```
echo 1 | seneschal  --prefix='yes `seq 1000`|head -c 1G' >/dev/null
```

takes 4100(!) CPU seconds to run on a 64C64T server, but only 140 CPU seconds on a 4C8T laptop. So it looks like **seneschal** wastes a lot of CPU time coordinating the CPUs.

Compare this to:

```
echo 1 | time -v parallel -N0 'yes `seq 1000`|head -c 1G' >/dev/null
```

which takes 3-8 CPU seconds.

### EXAMPLES FROM seneschal README.md

```
1$ echo $REPOS | seneschal --prefix="cd {} && git pull"

# If $REPOS is newline separated
1$ echo "$REPOS" | parallel -k "cd {} && git pull"
# If $REPOS is space separated
1$ echo -n "$REPOS" | parallel -d' ' -k "cd {} && git pull"

COMMANDS="pwd
sleep 5 && echo boom
echo Howdy
whoami"

2$ echo "$COMMANDS" | seneschal --debug

2$ echo "$COMMANDS" | parallel -k -v

3$ ls -1 | seneschal --prefix="pushd {}; git pull; popd;"

3$ ls -1 | parallel -k "pushd {}; git pull; popd;"
# Or if current dir also contains files:
3$ parallel -k "pushd {}; git pull; popd;" ::: */
```

https://github.com/TheWizardTower/seneschal (Last checked: 2022-06)

## DIFFERENCES BETWEEN async AND GNU Parallel

Summary (see legend above):

x x x x x x x

- x x x x x

x O2 O3 O4 O5 O6 - x x O10

E1 - - E4 - - -

- - - - - - - - -

S1 S2

**async** works like **sem**.

### EXAMPLES FROM async

```
1$ S="/tmp/example_socket"

   async -s="$S" server --start

   for i in {1..20}; do
       # prints command output to stdout
       async -s="$S" cmd -- bash -c "sleep 1 && echo test $i"
   done

   # wait until all commands are finished
   async -s="$S" wait

1$ S="example_id"
```

```
    # server not needed

    for i in {1..20}; do
        # prints command output to stdout
        sem --bg --id "$S" -j100% "sleep 1 && echo test $i"
    done

    # wait until all commands are finished
    sem --fg --id "$S" --wait

2$ # configure the server to run four commands in parallel
    async -s="$S" server -j4

    mkdir "/tmp/ex_dir"
    for i in {21..40}; do
        # redirects command output to /tmp/ex_dir/file*
        async -s="$S" cmd -o "/tmp/ex_dir/file$i" -- \
          bash -c "sleep 1 && echo test $i"
    done

    async -s="$S" wait

    # stops server
    async -s="$S" server --stop

2$ # starting server not needed

    mkdir "/tmp/ex_dir"
    for i in {21..40}; do
        # redirects command output to /tmp/ex_dir/file*
        sem --bg --id "$S" --results "/tmp/ex_dir/file$i{}" \
          "sleep 1 && echo test $i"
    done

    sem --fg --id "$S" --wait

    # there is no server to stop
```

https://github.com/ctbur/async (Last checked: 2023-01)

## DIFFERENCES BETWEEN tandem AND GNU Parallel

Summary (see legend above):

- - - I4 - - x

M1 - - - - M6

- - O3 - - - - x - -

E1 - E3 - E5 - -

- - - - - - - - -

- -

**tandem** runs full commands in parallel. It is made for starting a "server", running a job against the server, and when the job is done, the server is killed.

More generally: it kills all jobs when the first job completes - similar to '--halt now,done=1'.

**tandem** silently discards some output. It is unclear exactly when this happens. It looks like a race condition, because it varies for each run.

```
$ tandem "seq 10000" | wc -l
6731 <- This should always be 10002
```

### EXAMPLES FROM Demo

```
tandem \
  'php -S localhost:8000' \
  'esbuild src/*.ts --bundle --outdir=dist --watch' \
  'tailwind -i src/index.css -o dist/index.css --watch'

# Emulate tandem's behaviour
PARALLEL='--color --lb  --halt now,done=1 --tagstring '
PARALLEL="$PARALLEL'"'{=s/ .*//; $_.=".".$app{$_}++;=}'"'"
export PARALLEL

parallel ::: \
  'php -S localhost:8000' \
  'esbuild src/*.ts --bundle --outdir=dist --watch' \
  'tailwind -i src/index.css -o dist/index.css --watch'
```

### EXAMPLES FROM tandem -h

```
# Emulate tandem's behaviour
PARALLEL='--color --lb  --halt now,done=1 --tagstring '
PARALLEL="$PARALLEL'"'{=s/ .*//; $_.=".".$app{$_}++;=}'"'"
export PARALLEL

1$ tandem 'sleep 5 && echo "hello"' 'sleep 2 && echo "world"'

1$ parallel ::: 'sleep 5 && echo "hello"' 'sleep 2 && echo "world"'

# '-t 0' fails. But '--timeout 0 works'
2$ tandem --timeout 0 'sleep 5 && echo "hello"' \
     'sleep 2 && echo "world"'

2$ parallel --timeout 0 ::: 'sleep 5 && echo "hello"' \
     'sleep 2 && echo "world"'
```

### EXAMPLES FROM tandem's readme.md

```
# Emulate tandem's behaviour
PARALLEL='--color --lb  --halt now,done=1 --tagstring '
PARALLEL="$PARALLEL'"'{=s/ .*//; $_.=".".$app{$_}++;=}'"'"
export PARALLEL

1$ tandem 'next dev' 'nodemon --quiet ./server.js'

1$ parallel ::: 'next dev' 'nodemon --quiet ./server.js'

2$ cat package.json
   {
```

```
    "scripts": {
      "dev:php": "...",
      "dev:js": "...",
      "dev:css": "..."
    }
  }

  tandem 'npm:dev:php' 'npm:dev:js' 'npm:dev:css'

# GNU Parallel uses bash functions instead
2$ cat package.sh
  dev:php() { ... ; }
  dev:js() { ... ; }
  dev:css() { ... ; }
  export -f dev:php dev:js dev:css

  . package.sh
  parallel ::: dev:php dev:js dev:css

3$ tandem 'npm:dev:*'

3$ compgen -A function | grep ^dev: | parallel
```

For usage in Makefiles, include a copy of GNU Parallel with your source using `parallel --embed`. This has the added benefit of also working if access to the internet is down or restricted.

https://github.com/rosszurowski/tandem (Last checked: 2023-01)

## DIFFERENCES BETWEEN rust-parallel(aaronriekenberg) AND GNU Parallel

Summary (see legend above):

I1 I2 I3 - - - -

- - - - - M6

O1 O2 O3 - O5 O6 - x - O10

E1 - - E4 - - -

- - - - - - - - -

- -

**rust-parallel** has a goal of only using Rust. It seems it is impossible to call bash functions from the command line. You would need to put these in a script.

Calling a script that misses the shebang line (#! as first line) fails.

### EXAMPLES FROM rust-parallel's README.md

```
$ cat >./test <<EOL
echo hi
echo there
echo how
echo are
echo you
EOL

1$ cat test | rust-parallel -j5
```

```
1$ cat test | parallel -j5

2$ cat test | rust-parallel -j1

2$ cat test | parallel -j1

3$ head -100 /usr/share/dict/words | rust-parallel md5 -s

3$ head -100 /usr/share/dict/words | parallel md5 -s

4$ find . -type f -print0 | rust-parallel -0 gzip -f -k

4$ find . -type f -print0 | parallel -0 gzip -f -k

5$ head -100 /usr/share/dict/words |
     awk '{printf "md5 -s %s\n", $1}' | rust-parallel

5$ head -100 /usr/share/dict/words |
     awk '{printf "md5 -s %s\n", $1}' | parallel

6$ head -100 /usr/share/dict/words | rust-parallel md5 -s |
     grep -i abba

6$ head -100 /usr/share/dict/words | parallel md5 -s |
     grep -i abba
```

https://github.com/aaronriekenberg/rust-parallel (Last checked: 2023-01)

## DIFFERENCES BETWEEN parallelium AND GNU Parallel

Summary (see legend above):

- I2 - - - - -

M1 - - - - M6

O1 - O3 - - - - x - -

E1 - - E4 - - -

- - - - - - - - -

- -

**parallelium** merges standard output (stdout) and standard error (stderr). The maximal output of a command is 8192 bytes. Bigger output makes **parallelium** go into an infinite loop.

In the input file for **parallelium** you can define a tag, so that you can select to run only these commands. A bit like a target in a Makefile.

Progress is printed on standard output (stdout) prepended with '#' with similar information as GNU **parallel**'s **--bar**.

### EXAMPLES

```
$ cat testjobs.txt
#tag common sleeps classA
(sleep 4.495;echo "job 000")
:
(sleep 2.587;echo "job 016")
```

```
#tag common sleeps classB
(sleep 0.218;echo "job 017")
:
(sleep 2.269;echo "job 040")

#tag common sleeps classC
(sleep 2.586;echo "job 041")
:
(sleep 1.626;echo "job 099")

#tag lasthalf, sleeps, classB
(sleep 1.540;echo "job 100")
:
(sleep 2.001;echo "job 199")

1$ parallelium -f testjobs.txt -l logdir -t classB,classC

1$ cat testjobs.txt |
     parallel --plus --results logdir/testjobs.txt_{0#}.output \
       '{= if(/^#tag /) { @tag = split/,|\s+/ }
           (grep /^(classB|classC)$/, @tag) or skip =}'
```

https://github.com/beomagi/parallelium (Last checked: 2023-01)

## DIFFERENCES BETWEEN forkrun AND GNU Parallel

Summary (see legend above):

I1 - - - - - I7

- - - - - -

- O2 O3 - O5 - - - - O10

E1 - - E4 - - -

- - - - - - - - -

- -

**forkrun** blocks if it receives fewer jobs than slots:

```
echo | forkrun -p 2 echo
```

or when it gets some specific commands e.g.:

```
f() { seq "$@" | pv -qL 3; }
seq 10 | forkrun f
```

It is not clear why.

It is faster than GNU **parallel** (overhead: 1.2 ms/job vs 3 ms/job), but way slower than **parallel-bash** (0.059 ms/job).

Running jobs cannot be stopped by pressing CTRL-C.

**-k** is supposed to keep the order but fails on the MIX testing example below. If used with **-k** it caches output in RAM.

If **forkrun** is killed, it leaves temporary files in **/tmp/.forkrun.*** that has to be cleaned up manually.

**EXAMPLES**

```
1$ time find ./ -type f |
     forkrun -l512 -- sha256sum 2>/dev/null | wc -l
1$ time find ./ -type f |
     parallel -j28 -m -- sha256sum 2>/dev/null | wc -l

2$ time find ./ -type f |
     forkrun -l512 -k -- sha256sum 2>/dev/null | wc -l
2$ time find ./ -type f |
     parallel -j28 -k -m -- sha256sum 2>/dev/null | wc -l
```

https://github.com/jkool702/forkrun (Last checked: 2023-02)

## DIFFERENCES BETWEEN parallel-sh AND GNU Parallel

Summary (see legend above):

I1 I2 - I4 - - -

M1 - - - - M6

O1 O2 O3 - O5 O6 - - - O10

E1 - - E4 - - -

- - - - - - - - -

- -

**parallel-sh** buffers in RAM. The buffering data takes O(n^1.5) time:

2MB=0.107s 4MB=0.175s 8MB=0.342s 16MB=0.766s 32MB=2.2s 64MB=6.7s 128MB=20s 256MB=64s 512MB=248s 1024MB=998s 2048MB=3756s

It limits the practical usability to jobs outputting < 256 MB. GNU **parallel** buffers on disk, yet is faster for jobs with outputs > 16 MB and is only limited by the free space in $TMPDIR.

**parallel-sh** can kill running jobs if a job fails (Similar to **--halt now,fail=1**).

**EXAMPLES**

```
1$ parallel-sh "sleep 2 && echo first" "sleep 1 && echo second"

1$ parallel ::: "sleep 2 && echo first" "sleep 1 && echo second"

2$ cat /tmp/commands
   sleep 2 && echo first
   sleep 1 && echo second

2$ parallel-sh -f /tmp/commands

2$ parallel -a /tmp/commands

3$ echo -e 'sleep 2 && echo first\nsleep 1 && echo second' |
     parallel-sh

3$ echo -e 'sleep 2 && echo first\nsleep 1 && echo second' |
     parallel
```

https://github.com/thyrc/parallel-sh (Last checked: 2023-04)

## DIFFERENCES BETWEEN bash-parallel AND GNU Parallel

Summary (see legend above):

- I2 - - - - I7

M1 - M3 - M5 M6

- O2 O3 - - O6 - O8 - O10

E1 - - - - - -

- - - - - - - - -

- -

**bash-parallel** is not as much a command as it is a shell script that you have to alter. It requires you to change the shell function process_job that runs the job, and set $MAX_POOL_SIZE to the number of jobs to run in parallel.

It is half as fast as GNU **parallel** for short jobs.

https://github.com/thilinaba/bash-parallel (Last checked: 2024-06)

## DIFFERENCES BETWEEN PaSH AND GNU Parallel

Summary (see legend above): N/A

**pash** is quite different from GNU **parallel**. It is not a general parallelizer. It takes a shell script and analyses it and parallelizes parts of it by replacing the parts with commands that will give the same result.

This will replace **sort** with a command that does pretty much the same as **parsort --parallel=8** (except somewhat slower):

```
pa.sh --width 8 -c 'cat bigfile | sort'
```

However, even a simple change will confuse **pash** and you will get no parallelization:

```
pa.sh --width 8 -c 'mysort() { sort; }; cat bigfile | mysort'
pa.sh --width 8 -c 'cat bigfile | sort | md5sum'
```

From the source it seems **pash** only looks at: awk cat col comm cut diff grep head mkfifo mv rm sed seq sort tail tee tr uniq wc xargs

For pipelines where these commands are bottlenecks, it might be worth testing if **pash** is faster than GNU **parallel**.

**pash** does not respect $TMPDIR but always uses /tmp. If **pash** dies unexpectantly it does not clean up.

https://github.com/binpash/pash (Last checked: 2023-05)

## DIFFERENCES BETWEEN korovkin-parallel AND GNU Parallel

Summary (see legend above):

I1 - - - - - -

M1 - - - - M6

- - O3 - - - - x x -

E1 - - - - - -

R1 - - - - R6 x x -

- -

**korovkin-parallel** prepends all lines with some info.

The output is colored with 6 color combinations, so job 1 and 7 will get the same color.

You can get similar output with:

```
  (echo ...) |
    parallel --color -j 10 --lb --tagstring \
      '[l:{#}:{=$_=sprintf("%7.03f",::now()-$^T)=} {=$_=hh_mm_ss($^T)=}
{%}]'
```

Lines longer than 8192 chars are broken into lines shorter than 8192. **korovkin-parallel** loses the last char for lines exactly 8193 chars long.

Short lines from different jobs do not mix, but long lines do:

```
  fun() {
    perl -e '$a="'$1'"x1000000; for(1..'$2') { print $a };';
    echo;
  }
  export -f fun
  (echo fun a 100;echo fun b 100) | korovkin-parallel | tr -s abcdef
  # Compare to:
  (echo fun a 100;echo fun b 100) | parallel | tr -s abcdef
```

There should be only one line of a's and one line of b's.

Just like GNU **parallel korovkin-parallel** offers a master/slave model, so workers on other servers can do some of the tasks. But contrary to GNU **parallel** you must manually start workers on these servers. The communication is neither authenticated nor encrypted.

It caches output in RAM: a 1GB line uses ~2.5GB RAM

https://github.com/korovkin/parallel (Last checked: 2023-07)

## DIFFERENCES BETWEEN xe AND GNU Parallel

Summary (see legend above):

I1 I2 - I4 - - I7

M1 - M3 M4 - M6

- O2 O3 - O5 O6 - O8 - O10

E1 - - E4 - - -

---

--

**xe** has a peculiar limitation:

```
  echo /bin/echo | xe {} OK
  echo echo | xe /bin/{} fails
```

### EXAMPLES

Compress all .c files in the current directory, using all CPU cores:

```
  1$ xe -a -j0 gzip -- *.c

  1$ parallel gzip ::: *.c
```

Remove all empty files, using lr(1):

```
2$ lr -U -t 'size == 0' | xe -N0 rm

2$ lr -U -t 'size == 0' | parallel -X rm
```

Convert .mp3 to .ogg, using all CPU cores:

```
3$ xe -a -j0 -s 'ffmpeg -i "${1}" "${1%.mp3}.ogg"' -- *.mp3

3$ parallel ffmpeg -i {} {.}.ogg ::: *.mp3
```

Same, using percent rules:

```
4$ xe -a -j0 -p %.mp3 ffmpeg -i %.mp3 %.ogg -- *.mp3

4$ parallel --rpl '% s/\.mp3// or skip' ffmpeg -i %.mp3 %.ogg ::: *.mp3
```

Similar, but hiding output of ffmpeg, instead showing spawned jobs:

```
5$ xe -ap -j0 -vvq '%.{m4a,ogg,opus}' ffmpeg -y -i {} out/%.mp3 -- *

5$ parallel -v --rpl '% s/\.(m4a|ogg|opus)// or skip' \
     ffmpeg -y -i {} out/%.mp3 '2>/dev/null' ::: *

5$ parallel -v ffmpeg -y -i {} out/{.}.mp3 '2>/dev/null' ::: *
```

https://github.com/leahneukirchen/xe (Last checked: 2023-08)

## DIFFERENCES BETWEEN sp AND GNU Parallel

Summary (see legend above):

- - - I4 - - -
M1 - M3 - - M6
- O2 O3 - O5 (O6) - x x O10
E1 - - - - - -
- - - - - - - - -
- -

**sp** has very few options.

It can either be used like:

```
sp command {} option :: arg1 arg2 arg3
```

which is similar to:

```
parallel command {} option ::: arg1 arg2 arg3
```

Or:

```
sp command1 :: "command2 -option" :: "command3 foo bar"
```

which is similar to:

```
parallel ::: command1 "command2 -option" "command3 foo bar"
```

**sp** deals badly with too many commands: This causes **sp** to run out of file handles and gives data loss.

For each command that fails, **sp** will print an error message on stderr (standard error).

You cannot used exported shell functions as commands.

**EXAMPLES**

```
1$ sp echo {} :: 1 2 3

1$ parallel echo {} ::: 1 2 3

2$ sp echo {} {} :: 1 2 3

2$ parallel echo {} {} :: 1 2 3

3$ sp echo 1 :: echo 2 :: echo 3

3$ parallel ::: 'echo 1' 'echo 2' 'echo 3'

4$ sp a foo bar :: "b 'baz  bar'" :: c

4$ parallel ::: 'a foo bar' "b 'baz  bar'" :: c
```

https://github.com/SergioBenitez/sp (Last checked: 2023-10)

## DIFFERENCES BETWEEN repeater AND GNU Parallel

Summary (see legend above):

- - - - - - -

- - - - - -

- O2 O3 x - O6 - x x ?O10

E1 - - - E5 - -

- - - - - - - - -

- -

**repeater** runs the same job repeatedly. In other words: It does not read arguments, thus is it an alternative for GNU **parallel** for only quite limited applications.

**repeater** has an overhead of around 0.23 ms/job. Compared to GNU **parallel**'s 2-3 ms this is fast. Compared to **bash-parallel**'s 0.05 ms/job it is slow.

### Memory use and run time for large output

Output takes O(n^2) time for output of size n. 10 MB takes ~1 second, 30 MB takes ~7 seconds, 100 MB takes ~60 seconds, 300 MB takes ~480 seconds, 1000 MB takes ~10000 seconds.

100 MB of output takes around 1 GB of RAM.

```
# Run time = 15 sec
# Memory use = 20 MB
# Output = 1 GB per job
\time -v parallel -j1 seq ::: 120000000 120000000 >/dev/null

# Run time = 4.7 sec
# Memory use = 95 MB
```

```
    # Output = 8 MB per job
    \time -v repeater -w 1 -n 2 -reportFile ./run_output seq 1200000
>/dev/null

    # Run time = 42 sec
    # Memory use = 277 MB
    # Output = 27 MB per job
    \time -v repeater -w 1 -n 2 -reportFile ./run_output seq 3600000
>/dev/null

    # Run time = 530 sec
    # Memory use = 1000 MB
    # Output = 97 MB per job
    \time -v repeater -w 1 -n 2 -reportFile ./run_output seq 12000000
>/dev/null

    # Run time = 2h41m
    # Memory use = 8.6 GB
    # Output = 1 GB per job
    \time -v repeater -w 1 -n 2 -reportFile ./run_output seq 120000000
>/dev/null
```

For even just moderate sized outputs GNU **parallel** will be faster and use less memory.

**EXAMPLES**

```
  1$ repeater -n 100 -w 10 -reportFile ./run_output
       -output REPORT_FILE -progress BOTH curl example.com

  1$ seq 100 | parallel --joblog run.log --eta curl example.com > output

  2$ repeater -n 100 -increment -progress HIDDEN -reportFile foo
       echo "this is increment: " INC
  2$ seq 100 | parallel echo {}
  2$ seq 100 | parallel echo '{= $_ = ++$myvar =}'
```

https://github.com/baalimago/repeater (Last checked: 2023-12)

## DIFFERENCES BETWEEN parallelize AND GNU Parallel

Summary (see legend above):

I1 - - - - - I7

- - - - - M6

O1 - O3 O4 O5 - O7 - - -

E1 - - E4 - - -

- - - - - - - - -

- -

**parallelize** runs the full line as a command. If the command is not found, there is no warning.

The output at most ~1000000 lines/s. If the lines are short this is quite slow. The lines can at most be 2047999 bytes long. Longer lines cause segfault.

**EXAMPLES**

```
simple.dat:

  sleep 5
  foo
  cat alire.toml
  loc src/parallelize.adb
  sh loc src/*.ad?

1$ bin/parallelize -v <simple.dat

1$ parallel <simple.dat
```

https://github.com/simonjwright/parallelize (Last checked: 2024-04)

## DIFFERENCES BETWEEN pararun/paramap AND GNU Parallel

Summary (see legend above):

I1 - - - - - I7

M1 - M3 M4 - M6

- O2 O3 (O4) O5 O6 - - x -

E1 - (E3) E4 (E5) - - E8 E9

- - - - - - - - -

- -

**pararun** gets tricky when you need to quote more than 2 levels:

```
$ echo "$bmps" | parallel -q perl -e 'print @ARGV,"\n"' {}

$ echo "$bmps" | pararun 'perl -e '"'"'print @ARGV,"\n"'"'"' "$1"'
```

**paramap** tries to keep order of output, and works for short lines on stdout:

```
$ printf '%s\n' 2 1 4 3 | paramap -j4 'sleep $1; echo -n $1; sleep $1;
echo $1'
22
11
44
33
```

But adds newlines if there is no newline:

```
$ printf '%s\n' 2 1 4 3 | paramap -j4 'echo -n $1'
2
1
4
3

$ printf '%s\n' 2 1 4 3 | parallel -j4 'echo -n {}'
2143
```

**paramap** does not deal with stderr:

```
$ printf '%s\n' 2 1 4 3 |
```

```
    paramap -j4 'sleep $1; echo -n $1 >&2; sleep $1; echo $1 >&2'
121
342
3
4
```

**paramap** only prints, when all jobs are done. GNU **parallel** prints as soon as possible.

Fails **paramap** to run (due to final ;):

```
$ echo a | paramap 'echo $1;'
```

Long lines mix:

```
printf '%s\n' 2 1 4 3 |
  paramap -j4 'perl -e "print \"$1\"x3000_000,\" \"" ' |
  tr -s 1-4
```

And really long lines fail:

```
$ printf '%s\n' 1 2 3 4 |
    paramap -j2 'perl -e "map { print \"$1\"x5000_000 } 1..1000,\" \"" ' |
wc
    4       8 2840130828

$ printf '%s\n' 1 2 3 4 |
    parallel -j2 'perl -e "map { print \"{}\"x5000_000 } 1..1000,\" \"" '
| wc
    0       1 20020000000
```

## EXAMPLES

```
1$ bmps=$(find dir/ -type f -name '*.bmp')
1$ echo "$bmps" | pararun 'magick "$1" "${1%.*}".png'
```

If you like bash's parameter expansion, use --plus to get familar {1%.*}:

```
1$ echo "$bmps" | parallel --plus magick {1} {1%.*}.png
```

https://git.sr.ht/~q3cpma/scripts/ (Last checked: 2025-01)

## DIFFERENCES BETWEEN concur AND GNU Parallel

Summary (see legend above):

I1 - - I4 - - (I7)

- - M3 - - -

O1 - O3 - - - - x x O10

E1 - (E3) E4 - - - - -

- - - - - - - - -

- -

**concur** outputs in json. Output is cached in RAM and takes up a factor of 6: 1 GB output uses 6 GB RAM.

Standard input is split on space - not newline.

It can stop when the first job stops.

It gives the wrong output on jobs like:

```
concur 'perl -e "print(shift)"' "foo"
concur 'bash -c "sleep {{1}}; touch a{{1}}"' 3 2 1
```

## EXAMPLES

```
1$ concur "dig @{{1}} ocw.mit.edu" 1.1.1.1 9.9.9.9 8.8.8.8 94.140.14.14 \
   208.67.222.222

1$ parallel --results -.json  "dig @{1} ocw.mit.edu" 1 ::: 1.1.1.1
9.9.9.9\
   8.8.8.8 94.140.14.14 208.67.222.222 | jq .

2$ echo "foo bar baz" | concur "touch /tmp/{{1}}.foo"

2$ printf "%s\n" foo bar baz | parallel "touch /tmp/{1}.foo"

3$ concur "ping -c 1 {{1}}" www.mit.edu www.ucla.edu www.slashdot.org

3$ parallel --results -.json "ping -c 1 {1}" ::: www.mit.edu \
   www.ucla.edu www.slashdot.org | jq .

4$ concur "ping -c 1 {{1}}" www.mit.edu www.ucla.edu www.slashdot.org |
   jq '.command[0]'

4$ parallel --results -.json "ping -c 1 {1}" ::: www.mit.edu www.ucla.edu
 \
   www.slashdot.org | jq -s '.[1]'
```

https://github.com/ewosborne/concur (Last checked: 2025-05)

## Todo

https://github.com/rustunit/parallelrun?tab=readme-ov-file

https://github.com/justanhduc/task-spooler

https://manpages.ubuntu.com/manpages/xenial/man1/tsp.1.html

https://www.npmjs.com/package/concurrently

http://code.google.com/p/push/ (cannot compile)

https://github.com/krashanoff/parallel

https://github.com/Nukesor/pueue

https://arxiv.org/pdf/2012.15443.pdf KumQuat

https://github.com/JeiKeiLim/simple_distribute_job

https://github.com/reggi/pkgrun - not obvious how to use

https://github.com/benoror/better-npm-run - not obvious how to use

https://github.com/bahmutov/with-package

https://github.com/flesler/parallel

https://github.com/Julian/Verge

https://vicerveza.homeunix.net/~viric/soft/ts/

https://github.com/chapmanjacobd/que

## TESTING OTHER TOOLS

There are certain issues that are very common on parallelizing tools. Here are a few stress tests. Be warned: If the tool is badly coded it may overload your machine.

### MIX: Output mixes

Output from 2 jobs should not mix. If the output is not used, this does not matter; but if the output *is* used then it is important that you do not get half a line from one job followed by half a line from another job.

If the tool does not buffer, output will most likely mix now and then.

This test stresses whether output mixes.

```
  #!/bin/bash

  paralleltool="parallel -j 30"

  cat <<-'EOF' > mycommand
  #!/bin/bash

  # If a, b, c, d, e, and f mix: Very bad
  perl -e 'print STDOUT "a"x3000_000," "'
  perl -e 'print STDERR "b"x3000_000," "'
  perl -e 'print STDOUT "c"x3000_000," "'
  perl -e 'print STDERR "d"x3000_000," "'
  perl -e 'print STDOUT "e"x3000_000," "'
  perl -e 'print STDERR "f"x3000_000," "'
  echo "stdout line 1 of id $@"
  echo "stderr line 1 of id $@" >&2
  perl -e 'print STDOUT "A"x3000_000," "'
  perl -e 'print STDERR "B"x3000_000," "'
  perl -e 'print STDOUT "C"x3000_000," "'
  perl -e 'print STDERR "D"x3000_000," "'
  perl -e 'print STDOUT "E"x3000_000," "'
  perl -e 'print STDERR "F"x3000_000," "'
  echo "stdout line 2 of id $@"
  echo "stderr line 2 of id $@" >&2
  EOF
  chmod +x mycommand

  # Run 30 jobs in parallel
  seq 30 |
    $paralleltool -j 30  ./mycommand > >(tr -s a-zA-Z) 2> >(tr -s a-zA-Z
>&2)

  # 'a c e' and 'b d f' should always stay together
  # For each job there be 2 lines of standard output and standard error
  # They should not be interleaved with other id's
```

### STDERRMERGE: Stderr is merged with stdout

Output from stdout and stderr should not be merged, but kept separated.

This test shows whether stdout is mixed with stderr.

```
#!/bin/bash

paralleltool="parallel -j0"

cat <<-EOF > mycommand
#!/bin/bash

echo stdout
echo stderr >&2
echo stdout
echo stderr >&2
EOF
chmod +x mycommand

# Run one job
echo |
  $paralleltool ./mycommand > stdout 2> stderr
cat stdout
cat stderr
```

### RAM: Output limited by RAM

Some tools cache output in RAM. This makes them extremely slow if the output is bigger than physical memory and crash if the output is bigger than the virtual memory.

```
#!/bin/bash

paralleltool="parallel -j0"

cat <<'EOF' > mycommand
#!/bin/bash

# Generate 1 GB output
yes "`perl -e 'print \"c\"x30_000'`" | head -c 1G
EOF
chmod +x mycommand

# Run 20 jobs in parallel
# Adjust 20 to be > physical RAM and < free space on /tmp
seq 20 | time $paralleltool ./mycommand | wc -c
```

### DISKFULL: Incomplete data if /tmp runs full

If caching is done on disk, the disk can run full during the run. Not all programs discover this. GNU Parallel discovers it, if it stays full for at least 2 seconds.

```
#!/bin/bash

paralleltool="parallel -j0"
```

```
# This should be a dir with less than 100 GB free space
smalldisk=/tmp/shm/parallel

TMPDIR="$smalldisk"
export TMPDIR

max_output() {
    # Force worst case scenario:
    # Make GNU Parallel only check once per second
    sleep 10
    # Generate 100 GB to fill $TMPDIR
    # Adjust if /tmp is bigger than 100 GB
    yes | head -c 100G >$TMPDIR/$$
    # Generate 10 MB output that will not be buffered
    # due to full disk
    perl -e 'print "X"x10_000_000' | head -c 10M
    echo This part is missing from incomplete output
    sleep 2
    rm $TMPDIR/$$
    echo Final output
}

export -f max_output
seq 10 | $paralleltool max_output | tr -s X
```

### CLEANUP: Leaving tmp files at unexpected death

Some tools do not clean up tmp files if they are killed. If the tool buffers on disk, they may not clean up, if they are killed.

```
#!/bin/bash

paralleltool=parallel

ls /tmp >/tmp/before
seq 10 | $paralleltool sleep &
pid=$!
# Give the tool time to start up
sleep 1
# Kill it without giving it a chance to cleanup
kill -9 $!
# Should be empty: No files should be left behind
diff <(ls /tmp) /tmp/before
```

### SPCCHAR: Dealing badly with special file names.

It is not uncommon for users to create files like:

```
My brother's 12" *** record  (costs $$$).jpg
```

Some tools break on this.

```
#!/bin/bash

paralleltool=parallel
```

```
touch "My brother's 12\" *** record  (costs \$\$\$).jpg"
ls My*jpg | $paralleltool ls -l
```

### COMPOSED: Composed commands do not work

Some tools require you to wrap composed commands into **bash -c**.

```
echo bar | $paralleltool echo foo';' echo {}
```

### ONEREP: Only one replacement string allowed

Some tools can only insert the argument once.

```
echo bar | $paralleltool echo {} foo {}
```

### INPUTSIZE: Length of input should not be limited

Some tools limit the length of the input lines artificially with no good reason. GNU **parallel** does not:

```
perl -e 'print "foo."."x"x100_000_000' | parallel echo {.}
```

GNU **parallel** limits the command to run to 128 KB due to execve(1):

```
perl -e 'print "x"x131_000' | parallel echo {} | wc
```

### NUMWORDS: Speed depends on number of words

Some tools become very slow if output lines have many words.

```
#!/bin/bash

paralleltool=parallel

cat <<-EOF > mycommand
#!/bin/bash

# 10 MB of lines with 1000 words
yes "`seq 1000`" | head -c 10M
EOF
chmod +x mycommand

# Run 30 jobs in parallel
seq 30 | time $paralleltool -j0 ./mycommand > /dev/null
```

### 4GB: Output with a line > 4GB should be OK

```
#!/bin/bash

paralleltool="parallel -j0"

wcc() {
    parallel --recend '' --block 100M --pipe 'LC_ALL=C wc' |
      datamash -W sum 1 sum 2 sum 3
}

cat <<-EOF > mycommand
#!/bin/bash
```

```
perl -e '\$a="a"x1000_000; for(1..5000) { print \$a }'
EOF
chmod +x mycommand

# Run 1 job
seq 1 | $paralleltool ./mycommand | wcc
```

## AUTHOR

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.

This helps funding further development; and it won't cost you a cent. If you pay 10000 EUR you should feel free to use GNU Parallel without citing.

Copyright (C) 2007-10-18 Ole Tange, http://ole.tange.dk

Copyright (C) 2008-2010 Ole Tange, http://ole.tange.dk

Copyright (C) 2010-2026 Ole Tange, http://ole.tange.dk and Free Software Foundation, Inc.

Parts of the manual concerning **xargs** compatibility is inspired by the manual of **xargs** from GNU findutils 4.4.2.

## LICENSE

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or at your option any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

### Documentation license I

Permission is granted to copy, distribute and/or modify this documentation under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the file LICENSES/GFDL-1.3-or-later.txt.

### Documentation license II

You are free:

**to Share**

to copy, distribute and transmit the work

**to Remix**

to adapt the work

Under the following conditions:

**Attribution**

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

**Share Alike**

If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

With the understanding that:

**Waiver**

Any of the above conditions can be waived if you get permission from the copyright holder.

**Public Domain**

Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

**Other Rights**

In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

**Notice**

For any reuse or distribution, you must make clear to others the license terms of this work.

A copy of the full license is included in the file as LICENCES/CC-BY-SA-4.0.txt

## DEPENDENCIES

GNU **parallel** uses Perl, and the Perl modules Getopt::Long, IPC::Open3, Symbol, IO::File, POSIX, and File::Temp. For remote usage it also uses rsync with ssh.

## SEE ALSO

**find**(1), **xargs**(1), **make**(1), **pexec**(1), **ppss**(1), **xjobs**(1), **prll**(1), **dxargs**(1), **mdm**(1)