NAME

parallel - build and execute shell command lines from standard input in parallel

SYNOPSIS

parallel [options] [*command* [arguments]] < list_of_arguments

parallel [options] [*command* [arguments]] (**:::** arguments | **:::+** arguments | **::::** argfile(s) | **::::+** argfile(s)) ...

parallel --semaphore [options] *command*

#!/usr/bin/parallel --shebang [options] [*command* [arguments]]

#!/usr/bin/parallel --shebang-wrap [options] [*command* [arguments]]

DESCRIPTION

STOP!

Read the **Reader's guide** below if you are new to GNU **parallel**.

GNU **parallel** is a shell tool for executing jobs in parallel using one or more computers. A job can be a single command or a small script that has to be run for each of the lines in the input. The typical input is a list of files, a list of hosts, a list of users, a list of URLs, or a list of tables. A job can also be a command that reads from a pipe. GNU **parallel** can then split the input into blocks and pipe a block into each command in parallel.

If you use xargs and tee today you will find GNU **parallel** very easy to use as GNU **parallel** is written to have the same options as xargs. If you write loops in shell, you will find GNU **parallel** may be able to replace most of the loops and make them run faster by running several jobs in parallel.

GNU **parallel** makes sure output from the commands is the same output as you would get had you run the commands sequentially. This makes it possible to use output from GNU **parallel** as input for other programs.

For each line of input GNU **parallel** will execute *command* with the line as arguments. If no *command* is given, the line of input is executed. Several lines will be run in parallel. GNU **parallel** can often be used as a substitute for **xargs** or **cat | bash**.

Reader's guide

GNU **parallel** includes the 4 types of documentation: Tutorial, how-to, reference and explanation/design.

Tutorial

If you prefer reading a book buy **GNU Parallel 2018** at https://www.lulu.com/shop/ole-tange/gnu-parallel-2018/paperback/product-23558902.html or download it at: https://doi.org/10.5281/zenodo.1146014 Read at least chapter 1+2. It should take you less than 20 minutes.

Otherwise start by watching the intro videos for a quick introduction: https://youtube.com/playlist?list=PL284C9FF2488BC6D1

If you want to dive deeper: spend a couple of hours walking through the tutorial (**man parallel_tutorial**). Your command line will love you for it.

How-to

You can find a lot of examples of use in **man parallel_examples**. They will give you an idea of what GNU **parallel** is capable of, and you may find a solution you can simply adapt to your situation.

If the example do not cover your exact needs, the options map (https://www.gnu.org/software/parallel/parallel_options_map.pdf) can help you identify options that are related, so you can look these up in the man page.

Reference

If you need a one page printable cheat sheet you can find it on:
https://www.gnu.org/software/parallel/parallel_cheat.pdf

The man page is the reference for all options, and reading the man page from cover to cover is probably not what you need.

Design discussion

If you want to know the design decisions behind GNU **parallel**, try: **man parallel_design**. This is also a good intro if you intend to change GNU **parallel**.

OPTIONS

command

Command to execute.

If *command* or the following arguments contain replacement strings (such as **{}**) every instance will be substituted with the input.

If *command* is given, GNU **parallel** solve the same tasks as **xargs**. If *command* is not given GNU **parallel** will behave similar to **cat | sh**.

The *command* must be an executable, a script, a composed command, an alias, or a function.

Bash functions: **export -f** the function first or use **env_parallel**.

Bash, Csh, or Tcsh aliases: Use **env_parallel**.

Zsh, Fish, Ksh, and Pdksh functions and aliases: Use **env_parallel**.

{}

Input line.

This replacement string will be replaced by a full line read from the input source. The input source is normally stdin (standard input), but can also be given with **--arg-file**, **:::**, or **::::**.

The replacement string **{}** can be changed with **-I**.

If the command line contains no replacement strings then **{}** will be appended to the command line.

Replacement strings are normally quoted, so special characters are not parsed by the shell. The exception is if the command starts with a replacement string; then the string is not quoted.

See also: **--plus {.} {/} {//} {/.} {#} {%} {***n***} {=***perl expression***=}**

{.}

Input line without extension.

This replacement string will be replaced by the input with the extension removed. If the input line contains **.** after the last **/**, the last **.** until the end of the string will be removed and **{.}** will be replaced with the remaining. E.g. *foo.jpg* becomes *foo*, *subdir/foo.jpg* becomes *subdir/foo*, *sub.dir/foo.jpg* becomes *sub.dir/foo*, *sub.dir/bar* remains *sub.dir/bar*. If the input line does not contain **.** it will remain unchanged.

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

See also: **{} --extensionreplace**

{/}

Basename of input line.

This replacement string will be replaced by the input with the directory part removed.

See also: **{} --basenamereplace**

{//}

Dirname of input line.

This replacement string will be replaced by the dir of the input line. See **dirname**(1).

See also: **{} --dirnamereplace**

{/.}

Basename of input line without extension.

This replacement string will be replaced by the input with the directory and extension part removed. **{/.}** is a combination of **{/}** and **{.}**.

See also: **{} --basenameextensionreplace**

{#}

Sequence number of the job to run.

This replacement string will be replaced by the sequence number of the job being run. It contains the same number as $PARALLEL_SEQ.

See also: **{} --seqreplace**

{%} (alpha testing)

Job slot number.

This replacement string will be replaced by the job's slot number between 1 and number of jobs to run in parallel. There will never be 2 jobs running at the same time with the same job slot number.

If the job needs to be retried (e.g using **--retries** or **--retry-failed**) the job slot is not automatically updated. You should then instead use **$PARALLEL_JOBSLOT**:

```
  $ do_test() {
      id="$3 {%}=$1 PARALLEL_JOBSLOT=$2"
      echo run "$id";
      sleep 1
      # fail if {%} is odd
      return `echo $1%2 | bc`
    }
  $ export -f do_test
  $ parallel -j3 --jl mylog do_test {%} \$PARALLEL_JOBSLOT {} ::: A B
 C D
  run A {%}=1 PARALLEL_JOBSLOT=1
  run B {%}=2 PARALLEL_JOBSLOT=2
  run C {%}=3 PARALLEL_JOBSLOT=3
  run D {%}=1 PARALLEL_JOBSLOT=1
  $ parallel --retry-failed -j3 --jl mylog do_test {%}
\$PARALLEL_JOBSLOT {} ::: A B C D
  run A {%}=1 PARALLEL_JOBSLOT=1
  run C {%}=3 PARALLEL_JOBSLOT=2
  run D {%}=1 PARALLEL_JOBSLOT=3
```

Notice how {%} and $PARALLEL_JOBSLOT differ in the retry run of C and D.

See also: **{} --jobs --slotreplace**

{*n*}

Argument from input source *n* or the *n*'th argument.

This positional replacement string will be replaced by the input from input source *n* (when used with **--arg-file** or **::::**) or with the *n*'th argument (when used with **-N** or **--colsep**).

If *n* is negative it refers to the *n*'th last argument.

See also: **{} {*n*.} {*n*/} {*n*//} {*n*/.} --colsep**

{*n*.}

Argument from input source *n* or the *n*'th argument without extension.

{*n***.}** is a combination of **{***n***}** and **{.}**.

This positional replacement string will be replaced by the input from input source *n* (when used with **--arg-file** or **::::**) or with the *n*'th argument (when used with **-N**). The input will have the extension removed.

See also: **{***n***} {.}**

{*n***/}**

Basename of argument from input source *n* or the *n*'th argument.

{*n***/}** is a combination of **{***n***}** and **{/}**.

This positional replacement string will be replaced by the input from input source *n* (when used with **--arg-file** or **::::**) or with the *n*'th argument (when used with **-N**). The input will have the directory (if any) removed.

See also: **{***n***} {/}**

{*n***//}**

Dirname of argument from input source *n* or the *n*'th argument.

{*n***//}** is a combination of **{***n***}** and **{//}**.

This positional replacement string will be replaced by the dir of the input from input source *n* (when used with **--arg-file** or **::::**) or with the *n*'th argument (when used with **-N**). See **dirname** (1).

See also: **{***n***} {//}**

{*n***/.}**

Basename of argument from input source *n* or the *n*'th argument without extension.

{*n***/.}** is a combination of **{***n***}**, **{/}**, and **{.}**.

This positional replacement string will be replaced by the input from input source *n* (when used with **--arg-file** or **::::**) or with the *n*'th argument (when used with **-N**). The input will have the directory (if any) and extension removed.

See also: **{***n***} {/.}**

{=*perl expression***=}**

Replace with calculated *perl expression*.

$_** will contain the same as **{}**. After evaluating *perl expression* **$_ will be used as the value. It is recommended to only change $_ but you have full access to all of GNU **parallel**'s internal functions and data structures.

The expression must give the same result if evaluated twice - otherwise the behaviour is undefined. E.g. in some versions of GNU **parallel** this will not work as expected:

```
parallel echo '{= $_= ++$wrong_counter =}' ::: a b c
```

A few convenience functions and data structures have been made:

Q(*string***)**

Shell quote a string. Example:

```
parallel echo {} is quoted as '{= $_=Q($_) =}' ::: \$PWD
```

pQ(*string***)**

Perl quote a string. Example:

```
parallel echo {} is quoted as '{= $_=pQ($_) =}' ::: \$PWD
```

uq() (or **uq**)

Do not quote current replacement string. Example:

```
parallel echo {} has the value '{= uq =}' ::: \$PWD
```

hash(val)

Compute B::hash(val). Example:

```
parallel echo Hash of {} is '{= $_=hash($_) =}' ::: a b c
```

total_jobs()

Number of jobs in total. Example:

```
parallel echo Number of jobs: '{= $_=total_jobs() =}' ::: a b c
```

slot() (alpha testing)

Slot number of job. Example:

```
parallel echo Job slot of {} is '{= $_=slot() =}' ::: a b c
```

seq()

Sequence number of job. Example:

```
parallel echo Seq number of {} is '{= $_=seq() =}' ::: a b c
```

@arg

The arguments counting from 1 ($arg[1] = {1} = first argument). Example:

```
parallel echo {1}+{2}='{=1 $_=$arg[1]+$arg[2] =}' \
  ::: 1 2 3 ::: 2 3 4
```

('{=1' forces this to be a positional replacement string, and therefore will not repeat the value for each arg.)

skip() (alpha testing)

Skip this job (see also **--filter**). Example:

```
parallel echo '{= $arg[1] >= $arg[2] and skip =}' \
  ::: 1 2 3 ::: 2 3 4
```

yyyy_mm_dd_hh_mm_ss(sec)

yyyy_mm_dd_hh_mm(sec)

yyyy_mm_dd(sec)

hh_mm_ss(sec)

hh_mm(sec)

yyyymmddhhmmss(sec)

yyyymmddhhmm(sec)

yyyymmdd(sec)

hhmmss(sec)

hhmm(sec)

Time functions. *sec* is number of seconds since epoch. If left out it will use current local time. Example:

```
parallel echo 'Now: {= $_=yyyy_mm_dd_hh_mm_ss() =}' ::: Dummy
parallel echo 'The end: {= $_=yyyy_mm_dd_hh_mm_ss($_) =}' \
  ::: 2147483648
```

Example:

```
seq 10 | parallel echo {} + 1 is {= '$_++' =}
parallel csh -c {= '$_="mkdir ".Q($_)' =} ::: '12" dir'
seq 50 | parallel echo job {#} of {= '$_=total_jobs()' =}
```

See also: **--rpl --parens {} {=***n perl expression***=} --filter**

{=*n perl expression***=}**

Positional equivalent to **{=***perl expression***=}**.

To understand positional replacement strings see **{***n***}**.

See also: **{=***perl expression***=} {***n***}**

{*rpl*:*format***}**

Format replacement string.

Use *format* to format *rpl*. *format* is a format string used in **printf**. *rpl* is a replacement string.

Examples:

```
{#:%04d} - Job number ({#}) with 4 digits prepended with 0
{%:%02d} - Job slot ({%}) with 2 digits prepended with 0
{:%6s} - Input line ({}) right aligned 6 chars wide
{/:%12s} - Basename ({/}) right aligned 12 chars wide
{2:%8.2f} - Second input source ({2}) 8 chars wide, 2 decimals
```

Format strings also works on replacement strings defined via **--rpl** that start with '{'.

See also: **{} {***n***} --rpl**

::: *arguments*

Use arguments on the command line as input source.

Unlike other options for GNU **parallel :::** is placed after the *command* and before the arguments.

The following are equivalent:

```
(echo file1; echo file2) | parallel gzip
parallel gzip ::: file1 file2
parallel gzip {} ::: file1 file2
parallel --arg-sep ,, gzip {} ,, file1 file2
parallel --arg-sep ,, gzip ,, file1 file2
parallel ::: "gzip file1" "gzip file2"
```

To avoid treating **:::** as special use **--arg-sep** to set the argument separator to something else.

If multiple **:::** are given, each group will be treated as an input source, and all combinations of input sources will be generated. E.g. ::: 1 2 ::: a b c will result in the combinations (1,a) (1,b) (1,c) (2,a) (2,b) (2,c). This is useful for replacing nested for-loops.

:::, **::::**, and **--arg-file** can be mixed. So these are equivalent:

```
parallel echo {1} {2} {3} ::: 6 7 ::: 4 5 ::: 1 2 3
parallel echo {1} {2} {3} :::: <(seq 6 7) <(seq 4 5) \
  :::: <(seq 1 3)
parallel -a <(seq 6 7) echo {1} {2} {3} :::: <(seq 4 5) \
  :::: <(seq 1 3)
parallel -a <(seq 6 7) -a <(seq 4 5) echo {1} {2} {3} \
  ::: 1 2 3
seq 6 7 | parallel -a - -a <(seq 4 5) echo {1} {2} {3} \
  ::: 1 2 3
seq 4 5 | parallel echo {1} {2} {3} :::: <(seq 6 7) - \
  ::: 1 2 3
```

See also: **--arg-sep --arg-file :::: :::+ ::::+ --link**

:::+ *arguments*

Like **:::** but linked like **--link** to the previous input source.

Contrary to **--link**, values do not wrap: The shortest input source determines the length.

Example:

```
parallel echo ::: a b c :::+ 1 2 3 ::: X Y :::+ 11 22
```

See also: **::::+ --link**

:::: *argfiles*

Another way to write **--arg-file** *argfile1* **--arg-file** *argfile2* ...

::: and **::::** can be mixed.

See also: **--arg-file ::: ::::+ --link**

::::+ *argfiles*

Like **::::** but linked like **--link** to the previous input source.

Contrary to **--link**, values do not wrap: The shortest input source determines the length.

See also: **--arg-file :::+ --link**

--null

-0

Use NUL as delimiter.

Normally input lines will end in \n (newline). If they end in \0 (NUL), then use this option. It is useful for processing arguments that may contain \n (newline).

Shorthand for **--delimiter '\0'**.

See also: **--delimiter**

--arg-file *input-file*

-a *input-file*

Use *input-file* as input source.

If multiple **--arg-file** are given, each *input-file* will be treated as an input source, and all combinations of input sources will be generated. E.g. The file **foo** contains **1 2**, the file **bar** contains **a b c**. **-a foo -a bar** will result in the combinations (1,a) (1,b) (1,c) (2,a) (2,b) (2,c). This is useful for replacing nested for-loops.

If *input-file* starts with **+** the file will be linked to the previous **--arg-file** E.g. The file **foo** contains **1 2**, the file **bar** contains **a b**. **-a foo -a +bar** will result in the combinations (1,a) (2,b) like **--link** instead of generating all combinations.

See also: **--link {***n***} :::: ::::+ :::**

--arg-file-sep *sep-str*

Use *sep-str* instead of **::::** as separator string between command and argument files.

Useful if **::::** is used for something else by the command.

See also: **::::**

--arg-sep *sep-str*

Use *sep-str* instead of **:::** as separator string.

Useful if **:::** is used for something else by the command.

Also useful if you command uses **:::** but you still want to read arguments from stdin (standard input): Simply change **--arg-sep** to a string that is not in the command line.

See also: **:::**

--bar

Show progress as a progress bar.

In the bar is shown: % of jobs completed, estimated seconds left, and number of jobs started.

It is compatible with **zenity**:

```
seq 1000 | parallel -j30 --bar '(echo {};sleep 0.1)' \
  2> >(perl -pe 'BEGIN{$/="\r";$|=1};s/\r/\n/g' |
       zenity --progress --auto-kill) | wc
```

See also: **--eta --progress --total-jobs**

--basefile *file* (beta testing)

--bf *file* (beta testing)

file will be transferred to each sshlogin before first job is started.

It will be removed if **--cleanup** is active. The file may be a script to run or some common base data needed for the job. Multiple **--bf** can be specified to transfer more basefiles. The *file* will be transferred the same way as **--transferfile**.

See also: **--sshlogin --transfer --return --cleanup --workdir**

--basenamereplace *replace-str*

--bnr *replace-str*

Use the replacement string *replace-str* instead of **{/}** for basename of input line.

See also: **{/}**

--basenameextensionreplace *replace-str*

--bner *replace-str*

Use the replacement string *replace-str* instead of **{/.}** for basename of input line without extension.

See also: **{/.}**

--bin *binexpr*

Use *binexpr* as binning key and bin input to the jobs.

binexpr is [column number|column name] [perlexpression] e.g.:

```
3
Address
3 $_%=100
Address s/\D//g
```

Each input line is split using **--colsep**. The value of the column is put into $_, the perl expression is executed, the resulting value is is the job slot that will be given the line. If the value is bigger than the number of jobslots the value will be modulo number of jobslots.

This is similar to **--shard** but the hashing algorithm is a simple modulo, which makes it predictible which jobslot will receive which value.

The performance is in the order of 100K rows per second. Faster if the *bincol* is small (<10), slower if it is big (>100).

--bin requires **--pipe** and a fixed numeric value for **--jobs**.

See also: SPREADING BLOCKS OF DATA **--group-by --round-robin --shard**

--bg

Run command in background.

GNU **parallel** will normally wait for the completion of a job. With **--bg** GNU **parallel** will not wait for completion of the command before exiting.

This is the default if **--semaphore** is set.

Implies **--semaphore**.

See also: **--fg man sem**

--citation

Print the citation notice and BibTeX entry for GNU **parallel**, silence citation notice for all future runs, and exit. It will not run any commands.

If it is impossible for you to run **--citation** you can instead use **--will-cite**, which will run commands, but which will only silence the citation notice for this single run.

If you use **--will-cite** in scripts to be run by others you are making it harder for others to see the citation notice. The development of GNU **parallel** is indirectly financed through citations, so if your users do not know they should cite then you are making it harder to finance development. However, if you pay 10000 EUR, you have done your part to finance future development and should feel free to use **--will-cite** in scripts.

If you do not want to help financing future development by letting other users see the citation notice or by paying, then please consider using another tool instead of GNU **parallel**. You can find some of the alternatives in **man parallel_alternatives**.

--block *size* (beta testing)

--block-size *size* (beta testing)

Size of block in bytes to read at a time.

The *size* can be postfixed with K, M, G, T, P, k, m, g, t, or p.

GNU **parallel** tries to meet the block size but can be off by the length of one record. For performance reasons *size* should be bigger than a two records. GNU **parallel** will warn you and automatically increase the size if you choose a *size* that is too small.

size defaults to 1M.

A negative block size is not interpreted as a blocksize but as the number of blocks each jobslot should have. So **--block -3** will make 3 jobs for each jobslot. In other words: this will run 3*5 = 15 jobs in total:

```
parallel --pipe-part -a myfile --block -3 -j5 wc
cat myfile | parallel --pipe --block -3 -j5 wc
```

--pipe-part --block is an efficient alternative to **--round-robin** because data is never read by GNU **parallel**, but you can still have very few jobslots process huge amounts of data.

On the other hand, **--pipe --block** is quite *inefficient*: It reads the whole file into memory before splitting it. Thus input must be able to fit in memory.

If you use **--block** -*size*, input should be bigger than *size*+1 records.

See also: UNIT PREFIX **-N --pipe --pipe-part --round-robin --block-timeout**

--block-timeout *duration*

--bt *duration*

Timeout for reading block when using **--pipe**.

If it takes longer than *duration* to read a full block, use the partial block read so far.

duration is in seconds, but can be postfixed with s, m, h, or d.

See also: TIME POSTFIXES **--pipe --block**

--cat (beta testing)

Create a temporary file with content.

Normally **--pipe**/**--pipe-part** will give data to the program on stdin (standard input). With **--cat** GNU **parallel** will create a temporary file with the name in **{}**, so you can do: **parallel --pipe --cat wc {}**.

Implies **--pipe** unless **--pipe-part** is used.

See also: **--pipe --pipe-part --fifo**

--cleanup (beta testing)

Remove transferred files.

--cleanup will remove the transferred files on the remote computer after processing is done.

```
find log -name '*gz' | parallel \
  --sshlogin server.example.com --transferfile {} \
  --return {.}.bz2 --cleanup "zcat {} | bzip -9 >{.}.bz2"
```

With **--transferfile {}** the file transferred to the remote computer will be removed on the remote computer. Directories on the remote computer containing the file will be removed if they are empty.

With **--return** the file transferred from the remote computer will be removed on the remote computer. Directories on the remote computer containing the file will be removed if they are empty.

--cleanup is ignored when not used with **--basefile**, **--transfer**, **--transferfile** or **--return**.

See also: **--basefile --transfer --transferfile --sshlogin --return**

--color

Colour output.

Colour the output. Each job gets its own colour combination (background+foreground).

--color is ignored when using **-u**.

See also: **--color-failed**

--color-failed

--cf

Colour the output from failing jobs white on red.

Useful if you have a lot of jobs and want to focus on the failing jobs.

--color-failed is ignored when using **-u**, **--line-buffer** and unreliable when using **--latest-line**.

See also: **--color**

--colsep *regexp*

-C *regexp*

Column separator.

The input will be treated as a table with *regexp* separating the columns. The n'th column can be accessed using **{***n***}** or **{***n***.}**. E.g. **{3}** is the 3rd column.

If there are more input sources, each input source will be separated, but the columns from each input source will be linked. Here {4} refers to column 2 in input source 2:

```
parallel --colsep '-' echo {4} {3} {2} {1} \
  ::: A-B C-D ::: e-f g-h
```

--colsep implies **--trim rl**, which can be overridden with **--trim n**.

regexp is a Perl Regular Expression: https://perldoc.perl.org/perlre.html

See also: **--csv {***n***} --trim --link --match**

--combineexec *name*

--combine-executable *name*

Combine GNU **parallel** with another program into a single executable.

Let us say you have developed *myprg* which takes a single argument. You do not want to parallelize it yourself.

You could write a wrapper that uses GNU **parallel** called **myparprg**:

```
#!/bin/sh

parallel myprg ::: "$@"
```

But for others to use this, they need to install: GNU **parallel**, **myprg**, and **myparprg**.

It would be easier to install if all could be packed into a single executable.

If **myprg** is written in shell, you can use **--embed**.

If **myprg** is a binary you can use **--combineexec**.

Here we use **gzip** as example:

```
parallel --combineexec pargzip gzip -9 :::
```

You can now do:

```
./pargzip foo bar baz
```

If you want to pass options to **gzip** you can do:

```
parallel --combineexec pargzip gzip
```

Followed by:

```
./pargzip -1 ::: foo bar baz
```

See also: **--embed --shebang --shebang-wrap**

--compress

Compress temporary files.

If the output is big and very compressible this will take up less disk space in $TMPDIR and possibly be faster due to less disk I/O.

GNU **parallel** will try **pzstd**, **lbzip2**, **pbzip2**, **zstd**, **pigz**, **lz4**, **lzop**, **plzip**, **lzip**, **lrz**, **gzip**, **pxz**, **lzma**, **bzip2**, **xz**, **clzip**, in that order, and use the first available.

GNU **parallel** will use up to 8 processes per job waiting to be printed. See **man parallel_design** for details.

See also: **--compress-program**

--compress-program *prg*

--decompress-program *prg*

Use *prg* for (de)compressing temporary files.

It is assumed that *prg -dc* will decompress stdin (standard input) to stdout (standard output) unless **--decompress-program** is given.

See also: **--compress**

--csv

Treat input as CSV-format.

--colsep sets the field delimiter. **--csv** works very much like **--colsep** except it deals correctly with quoting. Compare:

```
echo '"1 big, 2 small","2""x4"" plank",12.34' |
  parallel --csv echo {1} of {2} at {3}
```

```
echo '"1 big, 2 small","2""x4"" plank",12.34' |
  parallel --colsep ',' echo {1} of {2} at {3}
```

Even quoted newlines are parsed correctly:

```
(echo '"Start of field 1 with newline'
 echo 'Line 2 in field 1";value 2') |
  parallel --csv --colsep ';' echo Field 1: {1} Field 2: {2}
```

When used with **--pipe** it will only pass full CSV-records.

See also: **--pipe --link {*n*} --colsep --header --match**

--ctag (obsolete: use **--color --tag**)

Color tag.

If the values look very similar looking at the output it can be hard to tell when a new value is used. **--ctag** gives each value a random color.

See also: **--color --tag**

--ctagstring *str* (obsolete: use **--color --tagstring**)

Color tagstring.

See also: **--color --ctag --tagstring**

--delay *duration*

Delay starting next job by *duration*.

GNU **parallel** will not start another job for the next *duration*.

duration is in seconds, but can be postfixed with s, m, h, or d.

If you append 'auto' to *duration* (e.g. 13m3sauto) GNU **parallel** will automatically try to find the optimal value: If a job fails, *duration* is increased by 30%. If a job succeeds, *duration* is decreased by 10%.

See also: TIME POSTFIXES **--retries --ssh-delay**

--delimiter *delim*

-d *delim*

Input records are terminated by *delim*.

The specified delimiter may be characters, C-style character escapes such as \n, or octal (\012) or hexadecimal (\x0A) escape codes. Octal and hexadecimal escape codes are understood as for the printf command.

See also: **--colsep**

--dirnamereplace *replace-str*

--dnr *replace-str*

Use the replacement string *replace-str* instead of **{*//*}** for dirname of input line.

See also: **{*//*}**

--dry-run (beta testing)

Print the job to run on stdout (standard output), but do not run the job.

Use **-v -v** to include the wrapping that GNU **parallel** generates (for remote jobs, **--tmux**, **--nice** , **--pipe**, **--pipe-part**, **--fifo** and **--cat**). Do not count on this literally, though, as the job may be scheduled on another computer or the local computer if : is in the list.

See also: **-v**

-E *eof-str*

Set the end of file string to *eof-str*.

If the end of file string occurs as a line of input, the rest of the input is not read. If neither **-E** nor **-e** is used, no end of file string is used.

--eof[=*eof-str*]

-e[*eof-str*]

This option is a synonym for the **-E** option.

Use **-E** instead, because it is POSIX compliant for **xargs** while this option is not. If *eof-str* is omitted, there is no end of file string. If neither **-E** nor **-e** is used, no end of file string is used.

--embed

Embed GNU **parallel** in a shell script.

If you need to distribute your script to someone who does not want to install GNU **parallel** you can embed GNU **parallel** in your own shell script:

```
parallel --embed > new_script
```

After which you add your code at the end of **new_script**. This is tested on **ash**, **bash**, **dash**, **ksh**, **sh**, and **zsh**.

--env *var* (alpha testing)

Copy exported environment variable *var*.

This will copy *var* to the environment that the command is run in. This is especially useful for remote execution.

In Bash *var* can also be a Bash function - just remember to **export -f** the function.

The variable '_' is special. It will copy all exported environment variables except for the ones mentioned in ~/.parallel/ignored_vars.

To copy the full environment (both exported and not exported variables, arrays, and functions) use **env_parallel**.

See also: **--record-env --session --sshlogin** *command* **env_parallel**

--eta

Show the estimated number of seconds before finishing.

This forces GNU **parallel** to read all jobs before starting to find the number of jobs (unless you use **--total-jobs**). GNU **parallel** normally only reads the next job to run.

The estimate is based on the runtime of finished jobs, so the first estimate will only be shown when the first job has finished.

Implies **--progress**.

See also: **--bar --progress --total-jobs**

--extensionreplace *replace-str*

--er *replace-str*

Use the replacement string *replace-str* instead of **{.}** for input line without extension.

See also: **{.}**

--fast (alpha testing)

Run jobs fast.

This disables a lot of functionality of GNU **parallel** to make jobs run as fast as possible: Think of it as the nitro racing car compared to the Volvo.

Useful for benchmarking and if you have 1000's of tiny jobs.

Compare:

```
# ~10 seconds/1000 jobs
seq 1000 | time parallel echo | wc -l
```

```
# ~1 seconds/1000 jobs
seq 1000 | time parallel --fast echo | wc -l
# ~10 seconds/1000000 jobs
seq 1000000 | time parallel --fast echo | wc -l
```

Fully supported:

{}, **--compress**/**--compress-program**, **--dry-run**, **--jobs**, **--keep-order**, **--nice**, **--tag**, **--tagstring** (fixed strings and **{}**), **--ungroup**.

Supported but slower (falls back to the job-queue path, ~0.5 ms/job):

All non-plain replacement strings: **{.} {/} {//} {/.} {#} {%} {***n***}**.

--tagstring with replacement strings other than **{}**.

Multiple input sources (multiple **:::**), **-a**/**--arg-file**, **::::**.

--null/**-0**, **-d**/**--delimiter**, **--colsep**/**-C**, **--csv**, **--link**/**:::+**/**::::+**, **-N** (for *N*>1), **-m**, **-X**, **--filter**, **--trim** (non-**n** values).

-E/**--eof**: end-of-file string not implemented.

Supported with slightly different semantics:

--group (default): output is grouped per chunk (several jobs). All stdout from jobs in a chunk is printed before all their stderr.

--halt/**--halt-on-error**: failure detection is per chunk, so it will not trigger until a full chunk of jobs has finished. The exit value is 0 unless **--halt** is set; use **--halt never** to propagate non-zero exit codes without stopping early.

{=*perl***=}**: Expressions that cannot be computed upfront (e.g. **slot()**) may give unexpected values.

--joblog/**--jl**: Command lines longer than 4 KB may be truncated.

--env: environment variables are inherited via normal **fork**; **parallel**'s explicit env-transfer mechanism is not invoked.

Broken but may be supported with slightly different semantics:

--timeout: applies to each chunk of jobs, not each individual job.

--bar/**--eta**/**--progress**: the inner **parallel** counts chunks, not individual jobs, so counts and time estimates are imprecise.

--delay: delay is applied after each chunk, not after each individual job.

--retries: These will retry the whole group of jobs.

--load/**--limit**/**--memfree**/**--noswap**: These will retry the whole group of jobs.

Not supported:

--results/**-o**: no per-job output file support.

--tty: jobs inherit **/bin/sh**'s stdin (a pipe), not a terminal.

-L/**--max-lines**/**-l**: multi-line records not supported.

--verbose/**-t**: echoing commands to stderr not implemented.

--total-jobs: total job count is not known ahead of time.

Impossible given current architecture:

--pipe/**--pipepart**/**--tee**/**--fifo**/**--cat**/**--header**/**--group-by**: separate execution modes incompatible with **--fast**.

SSH/remote options (**--sshlogin**, **--sshloginfile**/**--slf**, **--basefile**/**--bf**, **--transfer**, **--return**, **--cleanup**, **--trc**, **--filter-hosts**, **--workdir**, **--hostgroups**): **--fast** is local-only.

--tmux/**--tmuxpane**/**--bg**/**--fg**/**--wait**/**--semaphore**: separate execution modes.

If you need an unsupported option: File a bug.

See **parallel_design** for the architecture details.

--fg

Run command in foreground.

With **--tmux** and **--tmuxpane** GNU **parallel** will start **tmux** in the foreground.

With **--semaphore** GNU **parallel** will run the command in the foreground (opposite **--bg**), and wait for completion of the command before exiting. Exit code will be that of the command.

See also: **--bg man sem**

--fifo (alpha testing)

Create a temporary fifo with content.

Normally **--pipe** and **--pipe-part** will give data to the program on stdin (standard input). With **--fifo** GNU **parallel** will create a temporary fifo with the name in **{}**, so you can do:

```
parallel --pipe --fifo wc {}
```

Beware: If the fifo is never opened for reading, the job will block forever:

```
seq 1000000 | parallel --fifo echo This will block forever
seq 1000000 | parallel --fifo 'echo This will not block < {}'
```

By using **--fifo** instead of **--cat** you may save I/O as **--cat** will write to a temporary file, whereas **--fifo** will not.

Implies **--pipe** unless **--pipe-part** is used.

See also: **--cat --pipe --pipe-part**

--filter *filter* (alpha testing)

Only run jobs where *filter* is true.

filter can contain replacement strings and Perl code. Example:

```
parallel --filter '{1}+{2}+{3} < 10' echo {1},{2},{3} \
  ::: {1..10} ::: {3..8} ::: {3..10}
```

Outputs: 1,3,3 1,3,4 1,3,5 1,4,3 1,4,4 1,5,3 2,3,3 2,3,4 2,4,3 3,3,3

```
parallel --filter '{1} < {2}*{2}' echo {1},{2} \
  ::: {1..10} ::: {1..3}
```

Outputs: 1,2 1,3 2,2 2,3 3,2 3,3 4,3 5,3 6,3 7,3 8,3

```
parallel --filter '{choose_k}' --plus echo {1},{2},{3} \
  ::: {1..5} ::: {1..5} ::: {1..5}
```

Outputs: 1,2,3 1,2,4 1,2,5 1,3,4 1,3,5 1,4,5 2,3,4 2,3,5 2,4,5 3,4,5

See also: **skip() --no-run-if-empty {choose_k}**

--filter-hosts (beta testing)

Remove down hosts.

For each remote host: check that login through ssh works. If not: do not use this host.

For performance reasons, this check is performed only at the start and every time **--sshloginfile** is changed. If an host goes down after the first check, it will go undetected until **--sshloginfile** is changed; **--retries** can be used to mitigate this.

Currently you can *not* put **--filter-hosts** in a profile, $PARALLEL, /etc/parallel/config or similar. This is because GNU **parallel** uses GNU **parallel** to compute this, so you will get an infinite loop. This will likely be fixed in a later release.

See also: **--sshloginfile --sshlogin --retries**

--gnu

Behave like GNU **parallel**.

This option historically took precedence over **--tollef**. The **--tollef** option is now retired, and therefore may not be used. **--gnu** is kept for compatibility, but does nothing.

--group

Group output.

Output from each job is grouped together and is only printed when the command is finished. Stdout (standard output) first followed by stderr (standard error).

This takes in the order of 0.5ms CPU time per job and depends on the speed of your disk for larger output.

--group is the default.

With **--fast**: output is grouped per chunk (a chunk contains multiple jobs). Within a chunk stdout and stderr of each job appear in natural order (jobs run sequentially), but the inner **parallel** buffers all of a chunk's stdout before all of its stderr. Use **--tag** for correct per-job labelling.

See also: **--line-buffer --ungroup --tag**

--group-by *val* (alpha testing)

Group input by value.

Combined with **--pipe**/**--pipe-part --group-by** groups lines with the same value into a record.

The value can be computed from the full line or from a single column.

val can be:

column number

Use the value in the column numbered.

column name

Treat the first line as a header and use the value in the column named.

(Not supported with **--pipe-part**).

perl expression

Run the perl expression and use $_ as the value.

column number perl expression

Put the value of the column put in $_, run the perl expression, and use $_ as the value.

column name perl expression

Put the value of the column put in $_, run the perl expression, and use $_ as the value.

(Not supported with **--pipe-part**).

Example:

```
UserID, Consumption
123,    1
123,    2
12-3,   1
221,    3
221,    1
2/21,   5
```

If you want to group 123, 12-3, 221, and 2/21 into 4 records and pass one record at a time to **wc**:

```
tail -n +2 table.csv | \
  parallel --pipe --colsep , --group-by 1 -kN1 wc
```

Make GNU **parallel** treat the first line as a header:

```
cat table.csv | \
  parallel --pipe --colsep , --header : --group-by 1 -kN1 wc
```

Address column by column name:

```
cat table.csv | \
  parallel --pipe --colsep , --header : --group-by UserID -kN1 wc
```

If 12-3 and 123 are really the same UserID, remove non-digits in UserID when grouping:

```
cat table.csv | parallel --pipe --colsep , --header : \
  --group-by 'UserID s/\D//g' -kN1 wc
```

See also: SPREADING BLOCKS OF DATA **--pipe --pipe-part --bin --shard --round-robin**

--help

-h

Print a summary of the options to GNU **parallel** and exit.

--halt-on-error *val*

--halt *val*

When should GNU **parallel** terminate?

In some situations it makes no sense to run all jobs. GNU **parallel** should simply stop as soon as a condition is met.

val defaults to **never**, which runs all jobs no matter what.

val can also take on the form of *when*,*why*.

when can be 'now' which means kill all running jobs and halt immediately, or it can be 'soon' which means wait for all running jobs to complete, but start no new jobs.

why can be 'fail=X', 'fail=Y%', 'success=X', 'success=Y%', 'done=X', or 'done=Y%' where X is the number of jobs that has to fail, succeed, or be done before halting, and Y is the percentage of jobs that has to fail, succeed, or be done before halting.

Example:

--halt now,fail=1

exit when a job has failed. Kill running jobs.

--halt soon,fail=3

exit when 3 jobs have failed, but wait for running jobs to complete.

--halt soon,fail=3%

exit when 3% of the jobs have failed, but wait for running jobs to complete.

--halt now,success=1

exit when a job has succeeded. Kill running jobs.

--halt soon,success=3

exit when 3 jobs have succeeded, but wait for

running jobs to complete.

--halt now,success=3%

exit when 3% of the jobs have succeeded. Kill running jobs.

--halt now,done=1

exit when a job has finished. Kill running jobs.

--halt soon,done=3

exit when 3 jobs have finished, but wait for running jobs to complete.

--halt now,done=3%

exit when 3% of the jobs have finished. Kill running jobs.

For backwards compatibility these also work:

0	never
1	soon,fail=1
2	now,fail=1
-1	soon,success=1
-2	now,success=1
1-99%	soon,fail=1-99%

--header *regexp*

Use regexp as header.

For normal usage the matched header (typically the first line: **--header '.*\n'**) will be split using **--colsep** (which will default to '\t') and column names can be used as replacement variables: **{column name}**, **{column name/}**, **{column name//}**, **{column name/.}**, **{column name.}**, **{=column name perl expression =}**, ..

For **--pipe** the matched header will be prepended to each output.

--header : is an alias for **--header '.*\n'**.

If *regexp* is a number, it is a fixed number of lines.

--header 0 is special: It will make replacement strings for files given with **--arg-file** or **::::**. It will make **{foo/bar}** for the file **foo/bar**.

See also: **--colsep --pipe --pipe-part --arg-file**

--hostgroups

--hgrp

Enable hostgroups on arguments.

If an argument contains '@' the string after '@' will be removed and treated as a list of hostgroups on which this job is allowed to run. If there is no **--sshlogin** with a corresponding group, the job will run on any hostgroup.

Example:

```
parallel --hostgroups \
```

```
--sshlogin @grp1/myserver1 -S @grp1+grp2/myserver2 \
--sshlogin @grp3/myserver3 \
echo ::: my_grp1_arg@grp1 arg_for_grp2@grp2 third@grp1+grp3
```

my_grp1_arg may be run on either **myserver1** or **myserver2**, **third** may be run on either **myserver1** or **myserver3**, but **arg_for_grp2** will only be run on **myserver2**.

See also: **--sshlogin $PARALLEL_HOSTGROUPS $PARALLEL_ARGHOSTGROUPS**

-I *replace-str*

Use the replacement string *replace-str* instead of **{}**.

See also: **{}**

--replace [*replace-str*]

-i [*replace-str*]

This option is deprecated; use **-I** instead.

This option is a synonym for **-I***replace-str* if *replace-str* is specified, and for **-I {}** otherwise.

See also: **{}**

--joblog *logfile*

--jl *logfile*

Logfile for executed jobs.

Save a list of the executed jobs to *logfile* in the following TAB separated format: sequence number, sshlogin, start time as seconds since epoch, run time in seconds, bytes in files transferred, bytes in files returned, exit status, signal, and command run.

For **--pipe** bytes transferred and bytes returned are number of input and output of bytes.

If **logfile** is prepended with '+' log lines will be appended to the logfile.

To convert the times into ISO-8601 strict do:

```
cat logfile | perl -a -F"\t" -ne \
  'chomp($F[2]=`date -d \@$F[2] +%FT%T`); print join("\t",@F)'
```

If the host is long, you can use **column -t** to pretty print it:

```
cat joblog | column -t
```

See also: **--resume --resume-failed --progress**

--jobs *num*

-j *num*

--max-procs *num*

-P *num*

Number of jobslots on each machine.

Run up to *num* jobs in parallel. Default is 100%.

num

Run up to *num* jobs in parallel.

0 Run as many as possible (this can take a while to determine).

Due to a bug **-j 0** will also evaluate replacement strings twice up to the number of joblots:

```
  # This will not count from 1 but from number-of-jobslots
  seq 10000 | parallel -j0   echo '{= $_ = $foo++; =}' |
head
  # This will count from 1
```

```
  seq 10000 | parallel -j100 echo '{= $_ = $foo++; =}' |
head
```

num%

Multiply the number of CPU threads by *num* percent. E.g. 100% means one job per CPU thread on each machine.

+num

Add *num* to the number of CPU threads.

-num

Subtract *num* from the number of CPU threads.

expr

Evaluate *expr*. E.g. '12/2' to get 6, '+25%' gives the same as '125%', or complex expressions like '+3*log(55)%' which means: multiply 3 by log(55), multiply that by the number of CPU threads and divide by 100, add this to the number of CPU threads.

An expression that evalutates to less that 1 is replaced with 1.

procfile

Read parameter from file.

Use the content of *procfile* as parameter for *-j*. E.g. *procfile* could contain the string 100% or +2 or 10.

If *procfile* is changed when a job completes, *procfile* is read again and the new number of jobs is computed. If the number is lower than before, running jobs will be allowed to finish but new jobs will not be started until the wanted number of jobs has been reached. This makes it possible to change the number of simultaneous running jobs while GNU **parallel** is running.

*num*auto

If *num* ends in auto, GNU **parallel** will use the value as a max value. If a job fails, GNU **parallel** will adjust the number of jobs down with a factor. If a job succeeds GNU **parallel** will adjust the number of jobs up with a small factor.

So if the number of jobs is a bit too high, GNU **parallel** will dynamically lower it.

Example:

```
  # This fails if there are more than n sleeps running
  n_sleeps() {
    nsleeps=$(ps aux | grep sleep | grep 12345 | wc -l)
    echo $nsleeps sleeps running now
    [ $nsleeps -lt $1 ]
  }
  export -f n_sleeps
  parallel --jobs 20auto 'sleep 2.${RANDOM}{}12345;
n_sleeps 5' ::: {1..100}
```

This will start 20 jobs in parallel. These will fail because more than 5 sleeps are running. GNU **parallel** will adjust the number of jobs until around 5 are running in parallel.

If the evaluated number is less than 1 then 1 will be used.

If **--semaphore** is set, the default is 1 thus making a mutex.

See also: **--use-cores-instead-of-threads --use-sockets-instead-of-threads**

--keep-order (alpha testing)

-k (alpha testing)

Keep sequence of output same as the order of input.

Normally the output of a job will be printed as soon as the job completes. Try this to see the difference:

```
parallel -j4 sleep {}\; echo {} ::: 2 1 4 3
parallel -j4 -k sleep {}\; echo {} ::: 2 1 4 3
```

If used with **--onall** or **--nonall** the output will grouped by sshlogin in sorted order.

--keep-order cannot keep the output order when used with **--pipe --round-robin**. Here it instead means, that the jobslots will get the same blocks as input in the same order in every run if the input is kept the same. Run each of these twice and compare:

```
seq 10000000 | parallel --pipe --round-robin 'sleep 0.$RANDOM; wc'
seq 10000000 | parallel --pipe -k --round-robin 'sleep 0.$RANDOM; wc'
```

-k only affects the order in which the output is printed - not the order in which jobs are run.

See also: **--group --line-buffer**

-L *recsize*

When used with **--pipe**: Read records of *recsize*.

When used otherwise: Use at most *recsize* nonblank input lines per command line. Trailing blanks cause an input line to be logically continued on the next input line.

-L 0 means read one line, but insert 0 arguments on the command line.

recsize can be postfixed with K, M, G, T, P, k, m, g, t, or p.

Implies **-X** unless **-m**, **--xargs**, or **--pipe** is set.

See also: UNIT PREFIX **-N --max-lines --block -X -m --xargs --pipe**

--max-lines [*recsize*]

-l[*recsize*]

When used with **--pipe**: Read records of *recsize* lines.

When used otherwise: Synonym for the **-L** option. Unlike **-L**, the *recsize* argument is optional. If *recsize* is not specified, it defaults to one. The **-l** option is deprecated since the POSIX standard specifies **-L** instead.

-l 0 is an alias for **-l 1**.

Implies **-X** unless **-m**, **--xargs**, or **--pipe** is set.

See also: UNIT PREFIX **-N --block -X -m --xargs --pipe**

--limit "*command args*"

Dynamic job limit.

Before starting a new job run *command* with *args*. The exit value of *command* determines what GNU **parallel** will do:

0 Below limit. Start another job.

1 Over limit. Start no jobs.

2 Way over limit. Kill the youngest job.

You can use any shell command. There are 3 predefined commands:

"io *n*"

Limit for I/O. The amount of disk I/O will be computed as a value 0-100,

where 0 is no I/O and 100 is at least one disk is 100% saturated.

"load *n*"

Similar to **--load**.

"mem *n*"

Similar to **--memfree**.

See also: **--memfree --load**

--latest-line

--ll

Print the latest line. Each job gets a single line on the screen that is updated with the last full line from currently running jobs.

The screen keeps the oldest running job in view, so younger jobs may no be visible. These will, however, be shown when the oldest job finishes.

Running this example makes it easer to understand. Note how the screen scrolls when the job at the top of the screen finishes:

```
  slow_seq() {
    # Run seq with 33 characters per second
    seq "$@" |
      perl -ne '$|=1; for(split//){ print; select($a,$a,$a,0.03);}'
  }
  export -f slow_seq
  parallel --shuf -j99 --ll --tag --bar --color slow_seq {} :::
{1..300}
```

See also: **--line-buffer**

--line-buffer

--lb

Buffer output on line basis.

--group will keep the output together for a whole job. **--ungroup** allows output to mixup with half a line coming from one job and half a line coming from another job. **--line-buffer** fits between these two: GNU **parallel** will print a full line, but will allow for mixing lines of different jobs.

--line-buffer takes more CPU power than both **--group** and **--ungroup**, but can be much faster than **--group** if the CPU is not the limiting factor.

Normally **--line-buffer** does not buffer on disk, and can thus process an infinite amount of data, but it will buffer on disk when combined with: **--keep-order**, **--results**, **--compress**, and **--files**. This will make it as slow as **--group** and will limit output to the available disk space.

With **--keep-order --line-buffer** will output lines from the first job continuously while it is running, then lines from the second job while that is running. It will buffer full lines, but jobs will not mix. Compare:

```
  parallel -j0 'echo [{};sleep {};echo {}]' ::: 1 3 2 4
  parallel -j0 --lb 'echo [{};sleep {};echo {}]' ::: 1 3 2 4
  parallel -j0 -k --lb 'echo [{};sleep {};echo {}]' ::: 1 3 2 4
```

See also: **--group --ungroup --keep-order --tag**

--link

--xapply

Link input sources.

Read multiple input sources like the command **xapply**. If multiple input sources are given, one argument will be read from each of the input sources. The arguments can be accessed in the command as **{1}** .. **{*n*}**, so **{1}** will be a line from the first input source, and **{6}** will refer to the line with the same line number from the 6th input source.

Compare these two:

```
parallel echo {1} {2} ::: 1 2 3 ::: a b c
parallel --link echo {1} {2} ::: 1 2 3 ::: a b c
```

Arguments will be recycled if one input source has more arguments than the others:

```
parallel --link echo {1} {2} {3} \
  ::: 1 2 ::: I II III ::: a b c d e f g
```

See also: **--header :::+ ::::+**

--load *max-load*

Only start jobs if load is less than max-load.

Do not start new jobs on a given computer unless the number of running processes on the computer is less than *max-load*. *max-load* uses the same syntax as **--jobs**, so *100%* for one per CPU is a valid setting. Only difference is 0 which is interpreted as 0.01.

See also: **--limit --jobs**

--controlmaster

-M

Use ssh's ControlMaster to make ssh connections faster.

Useful if jobs run remote and are very fast to run. This is disabled for sshlogins that specify their own ssh command.

See also: **--ssh --sshlogin**

-m

Multiple arguments.

Insert as many arguments as the command line length permits. If multiple jobs are being run in parallel: distribute the arguments evenly among the jobs. Use **-j1** or **--xargs** to avoid this.

If **{}** is not used the arguments will be appended to the line. If **{}** is used multiple times each **{}** will be replaced with all the arguments.

Support for **-m** with **--sshlogin** is limited and may fail.

If in doubt use **-X** as that will most likely do what is needed.

See also: **-X --xargs**

--match *regexp*

Match input source with regexp to set replacement fields.

With **--match** you can often avoid pre-processing your input with **awk** or similar to extract the relevant fields.

You can access each capture group i.e. the parenthesis in regexp '(...)' with a replacement field. The replacement fields are named **{m.n}** where m is the input source and n is the capture group number:

```
parallel --match '(.*),(.*)_(.*)' echo {1.2} {1.3} {1.1} \
  ::: Gold,Heart_of
parallel --match '(.*),' --match '([a-z]+)' echo {1.1}{2.1} \
  ::: Milli,bar ::: 10ways
```

To reuse a **--match** simply use **+n** where n is the input source. E.g. **+2** for the second:

```
  parallel --match +2 --match '([A-Za-z]+)' echo {1.1} {2.1} \
    ::: /Improbability/ ::: 10drive
```

To only set **--match** for input source 2, make a dummy **--match** for input source 1:

```
  parallel --match '' --match '([a-z]+)' echo {1} {2.1} \
    ::: Telephone ::: 10sanitizer
```

See also: **{*n*} --colsep**

--memfree *size*

Minimum memory free when starting another job.

The *size* can be postfixed with K, M, G, T, P, k, m, g, t, or p.

If the jobs take up very different amount of RAM, GNU **parallel** will only start as many as there is memory for. If less than *size* bytes are free, no more jobs will be started. If less than 50% *size* bytes are free, the youngest job will be killed (as per **--term-seq**), and put back on the queue to be run later.

See also: UNIT PREFIX **--term-seq --memsuspend**

--memsuspend *size*

Suspend jobs when there is less memory available.

If the available memory falls below 2 * *size*, GNU **parallel** will suspend some of the running jobs. If the available memory falls below *size*, only one job will be running.

If a single job fits in the given size, all jobs will complete without running out of memory. If you have swap available, you can usually lower *size* to around half the size of a single job - with the slight risk of swapping a little.

Jobs will be resumed when more RAM is available - typically when the oldest job completes.

--memsuspend only works on local jobs because there is no obvious way to suspend remote jobs.

size can be postfixed with K, M, G, T, P, k, m, g, t, or p.

See also: UNIT PREFIX **--memfree**

--milestone *str*

Set milestones where all previous jobs must have finished.

Example:

```
  doit() {
      start=$(date +%H:%M:%S)
      # Sleep up to 5 sec
      sleep $((RANDOM % 5000))e-3
      end=$(date +%H:%M:%S)
      echo Start: $start End: $end
  }
  export -f doit
  parallel --tag --milestone // doit ::: a b c // 1 2 // A B // I II
II
```

This will run (a, b, c), (1, 2), (A, B), (I, II, III). The jobs in a batch will be run in parallel, but the batches will be run in serial.

Note how the start time for each batch is later than the end time of the previous batch.

Cartesian product example:

```
  parallel --tag --milestone // doit ::: Frist_run // Second_run :::
file{1..10}
```

This will complete First_run with all files before starting Second_run on all files.

Advanced cartesian example (buggy):

```
  parallel --tag --milestone // doit ::: a b // 1 2 3 ::: A B C // I
II
```

This runs: [a] x [A B C], [a] x [I II], [b] x [A B C], [b] x [I II], [1] x [A B C], [1] x [I II], [2] x [A B C], [2] x [I II], [3] x [A B C], [3] x [I II] = (a A, a B, a C), (a I, a II), (b A, b B, b C), (b I, b II), (1 A, 1 B, 1 C), (1 I, 1 II), (2 A, 2 B, 2 C), (2 I, 2 II), (3 A, 3 B, 3 C), (3 I, 3 II).

The jobs in a batch will be run in parallel, but the batches will be run in serial.

When the bug is fixed, it will run: [a b] x [A B C], [a b] x [I II], [1 2 3] x [A B C], [1 2 3] x [I II] = (a A, a B, a C, b A, b B, b C), (a I, a II, b I, b II), (1 A, 1 B, 1 C, 2 A, 2 B, 2 C, 3 A, 3 B, 3 C), (1 I, 1 II, 2 I, 2 II, 3 I, 3 II).

--minversion *version*

Print the version GNU **parallel** and exit.

If the current version of GNU **parallel** is less than *version* the exit code is 255. Otherwise it is 0.

This is useful for scripts that depend on features only available from a certain version of GNU **parallel**:

```
   parallel --minversion 20170422 &&
     echo halt done=50% supported from version 20170422 &&
     parallel --halt now,done=50% echo ::: {1..100}
```

See also: **--version**

--max-args *max-args*

-n *max-args*

Use at most *max-args* arguments per command line.

Fewer than *max-args* arguments will be used if the size (see the **-s** option) is exceeded, unless the **-x** option is given, in which case GNU **parallel** will exit.

-n 0 means read one argument, but insert 0 arguments on the command line.

max-args can be postfixed with K, M, G, T, P, k, m, g, t, or p (see UNIT PREFIX).

Implies **-X** unless **-m** is set.

See also: **-X -m --xargs --max-replace-args**

--max-replace-args *max-args*

-N *max-args*

Use at most *max-args* arguments per command line.

Like **-n** but also makes replacement strings **{1}** .. **{***max-args***}** that represents argument 1 .. *max-args*. If too few args the **{***n***}** will be empty.

-N 0 means read one argument, but insert 0 arguments on the command line.

This will set the owner of the homedir to the user:

```
  tr ':' '\n' < /etc/passwd | parallel -N7 chown {1} {6}
```

Implies **-X** unless **-m** or **--pipe** is set.

max-args can be postfixed with K, M, G, T, P, k, m, g, t, or p.

When used with **--pipe -N** is the number of records to read. This is somewhat slower than **--block**.

See also: UNIT PREFIX **--pipe --block -m -X --max-args**

--nonall (alpha testing)

> **--onall** with no arguments.
>
> Run the command on all computers given with **--sshlogin** but take no arguments. GNU **parallel** will log into **--jobs** number of computers in parallel and run the job on the computer. **-j** adjusts how many computers to log into in parallel.
>
> This is useful for running the same command (e.g. uptime) on a list of servers.
>
> See also: **--onall --sshlogin**

--onall (alpha testing)

> Run all the jobs on all computers given with **--sshlogin**.
>
> GNU **parallel** will log into **--jobs** number of computers in parallel and run one job at a time on the computer. The order of the jobs will not be changed, but some computers may finish before others.
>
> When using **--group** the output will be grouped by each server, so all the output from one server will be grouped together.
>
> **--joblog** will contain an entry for each job on each server, so there will be several job sequence 1.
>
> See also: **--nonall --sshlogin**

--open-tty

-o

> Open terminal tty.
>
> Similar to **--tty** but does not set **--jobs** or **--ungroup**.
>
> See also: **--tty**

--output-as-files

--outputasfiles

--files

--files0

> Save output to files.
>
> Instead of printing the output to stdout (standard output) the output of each job is saved in a file and the filename is then printed.
>
> **--files0** uses NUL (\0) instead of newline (\n) as separator.
>
> See also: **--results**

--pipe (beta testing)

--spreadstdin (beta testing)

> Spread input to jobs on stdin (standard input).
>
> Read a block of data from stdin (standard input) and give one block of data as input to one job.
>
> The block size is determined by **--block** (default: 1M).
>
> Except for the first and last record GNU **parallel** only passes full records to the job. The strings **--recstart** and **--recend** determine where a record starts and ends: The border between two records is defined as **--recend** immediately followed by **--recstart**. GNU **parallel** splits exactly after **--recend** and before **--recstart**. The block will have the last partial record removed before the block is passed on to the job. The partial record will be prepended to next block.
>
> You can limit the number of records to be passed with **-N**, and set the record size with **-L**.
>
> **--pipe** maxes out at around 1 GB/s input, and 100 MB/s output. If performance is important

use **--pipe-part**.

--fifo and **--cat** will give stdin (standard input) on a fifo or a temporary file.

If data is arriving slowly, you can use **--block-timeout** to finish reading a block early.

The data can be spread between the jobs in specific ways using **--round-robin**, **--bin**, **--shard** , **--group-by**. See the section: SPREADING BLOCKS OF DATA

See also: **--block --block-timeout --recstart --recend --fifo --cat --pipe-part -N -L --round-robin**

--pipe-part (alpha testing)

Pipe parts of a physical file.

--pipe-part works similar to **--pipe**, but is much faster. 5 GB/s can easily be delivered.

--pipe-part has a few limitations:

- The file must be a normal file or a block device (technically it must be seekable) and must be given using **--arg-file** or **::::**. The file cannot be a pipe, a fifo, or a stream as they are not seekable.

  If using a block device with lot of NUL bytes, remember to set **--recend ''**.

- Record counting (**-N**) and line counting (**-L**/**-l**) do not work. Instead use **--recstart** and **--recend** to determine where records end.

See also: **--pipe --recstart --recend --arg-file ::::**

--plain

Ignore **--profile**, $PARALLEL, and ~/.parallel/config.

Ignore any **--profile**, $PARALLEL, and ~/.parallel/config to get full control on the command line (used by GNU **parallel** internally when called with **--sshlogin**).

Security note: The $PARALLEL environment variable is parsed as additional command-line options, including options whose values are evaluated as expressions (e.g. **--blocksize**, **--timeout**, **--delay**). In privileged contexts â€” setuid wrappers, sudo scripts, or any situation where $PARALLEL may be set by a less-trusted party â€” pass **--plain** to prevent $PARALLEL (and user profiles) from influencing the invocation.

Example wrapper script running as root:

```
#!/bin/sh
exec parallel --plain "$@"
```

See also: **--profile**

--plus

Add more replacement strings.

Activate additional replacement strings: {+/} {+.} {+..} {+...} {..} {...} {/..} {/...} {##}. The idea being that '{+foo}' matches the opposite of '{foo}' so that:

{} = {+/}/{/} = {.}.{+.} = {+/}/{/.}.{+.} = {..}.{+..} = {+/}/{/..}.{+..} = {...}.{+...} = {+/}/{/...}.{+...}

{##} is the total number of jobs to be run. It is incompatible with **-X**/**-m**/**--xargs**.

{0%} zero-padded jobslot.

{0#} zero-padded sequence number.

{slot-1} jobslot - 1 (i.e. counting from 0).

{seq-1} sequence number - 1 (i.e. counting from 0).

{choose_k} is inspired by n choose k: Given a list of n elements, choose k. k is the number of input sources and n is the number of arguments in an input source. The content of the input sources must be the same and the arguments must be unique.

{uniq} skips jobs where values from two input sources are the same.

Shorthands for variables:

```
{slot}          $PARALLEL_JOBSLOT (see {%})
{sshlogin}      $PARALLEL_SSHLOGIN
{host}          $PARALLEL_SSHHOST
{agrp}          $PARALLEL_ARGHOSTGROUPS
{hgrp}          $PARALLEL_HOSTGROUPS
```

The following dynamic replacement strings are also activated. They are inspired by bash's parameter expansion:

```
{:-str}         str if the value is empty
{:num}          remove the first num characters
{:pos:len}      substring from position pos length len
{#regexp}       remove prefix regexp (non-greedy)
{##regexp}      remove prefix regexp (greedy)
{%regexp}       remove postfix regexp (non-greedy)
{%%regexp}      remove postfix regexp (greedy)
{/regexp/str}   replace one regexp with str
{//regexp/str} replace every regexp with str
{^str}          uppercase str if found at the start
{^^str}         uppercase str
{,str}          lowercase str if found at the start
{,,str}         lowercase str
```

See also: **--rpl {}**

--process-slot-var *varname*

Set the environment variable *varname* to the jobslot number-1.

```
seq 10 | parallel --process-slot-var=name echo '$name' {}
```

--progress

Show progress of computations.

List the computers involved in the task with number of CPUs detected and the max number of jobs to run. After that show progress for each computer: number of running jobs, number of completed jobs, and percentage of all jobs done by this computer. The percentage will only be available after all jobs have been scheduled as GNU **parallel** only read the next job when ready to schedule it - this is to avoid wasting time and memory by reading everything at startup.

By sending GNU **parallel** SIGUSR2 you can toggle turning on/off **--progress** on a running GNU **parallel** process.

See also: **--eta --bar --joblog**

--max-line-length-allowed

Print maximal command line length.

Print the maximal number of characters allowed on the command line and exit (used by GNU **parallel** itself to determine the line length on remote computers).

See also: **--show-limits**

--number-of-cpus (obsolete)

Print the number of physical CPU cores and exit.

--number-of-cores

Print the number of physical CPU cores and exit (used by GNU **parallel** itself to determine the number of physical CPU cores on remote computers).

See also: **--number-of-sockets --number-of-threads --use-cores-instead-of-threads --jobs**

--number-of-sockets

Print the number of filled CPU sockets and exit (used by GNU **parallel** itself to determine the number of filled CPU sockets on remote computers).

See also: **--number-of-cores --number-of-threads --use-sockets-instead-of-threads --jobs**

--number-of-threads

Print the number of hyperthreaded CPU cores and exit (used by GNU **parallel** itself to determine the number of hyperthreaded CPU cores on remote computers).

See also: **--number-of-cores --number-of-sockets --jobs**

--no-keep-order

Overrides an earlier **--keep-order** (e.g. if set in **~/.parallel/config**).

--nice *niceness*

Run the command at this niceness.

By default GNU **parallel** will run jobs at the same nice level as GNU **parallel** is started - both on the local machine and remote servers, so you are unlikely to ever use this option.

Setting **--nice** will override this nice level. If the nice level is smaller than the current nice level, it will only affect remote jobs (e.g. if current level is 10 then **--nice 5** will cause local jobs to be run at level 10, but remote jobs run at nice level 5).

--interactive

-p

Ask user before running a job.

Prompt the user about whether to run each command line and read a line from the terminal. Only run the command line if the response starts with 'y' or 'Y'. Implies **-t**.

--_parset *type*,*varname*

Used internally by **parset**.

Generate shell code to be eval'ed which will set the variable(s) *varname*. *type* can be 'assoc' for associative array or 'var' for normal variables.

The only supported use is as part of **parset**.

--parens *parensstring*

Use *parensstring* instead of **{==}**.

Define start and end parenthesis for **{=perl expression=}**. The left and the right parenthesis can be multiple characters and are assumed to be the same length. The default is **{==}** giving **{=** as the start parenthesis and **=}** as the end parenthesis.

Another useful setting is **,,,,** which would make both parenthesis **,,**:

```
parallel --parens ,,,, echo foo is ,,s/I/O/g,, ::: FII
```

See also: **--rpl {=***perl expression***=}**

--profile *profilename*

-J *profilename*

Use profile *profilename* for options.

This is useful if you want to have multiple profiles. You could have one profile for running jobs in parallel on the local computer and a different profile for running jobs on remote computers.

profilename corresponds to the file ~/.parallel/*profilename*.

You can give multiple profiles by repeating **--profile**. If parts of the profiles conflict, the later

ones will be used.

Default: ~/.parallel/config

See also: PROFILE FILES

--quote

-q

Quote *command*.

If your command contains special characters that should not be interpreted by the shell (e.g. ; \ | *), use **--quote** to escape these. The command must be a simple command (see **man bash**) without redirections and without variable assignments.

Most people will not need this. Quoting is disabled by default.

See also: QUOTING *command* **--shell-quote uq() Q()**

--no-run-if-empty

-r

Do not run empty input.

If the stdin (standard input) only contains whitespace, do not run the command.

If used with **--pipe** this is slow.

See also: *command* **--pipe --interactive**

--noswap

Do not start job is computer is swapping.

Do not start new jobs on a given computer if there is both swap-in and swap-out activity.

The swap activity is only sampled every 10 seconds as the sampling takes 1 second to do.

Swap activity is computed as (swap-in)*(swap-out) which in practice is a good value: swapping out is not a problem, swapping in is not a problem, but both swapping in and out usually indicates a problem.

--memfree and **--memsuspend** may give better results, so try using those first.

See also: **--memfree --memsuspend**

--record-env

Record exported environment.

Record current exported environment variables in **~/.parallel/ignored_vars**. This will ignore variables currently set when using **--env** _. So you should set the variables/fuctions, you want to use *after* running **--record-env**.

See also: **--env --session env_parallel**

--recstart *startstring*

--recend *endstring*

Split record between *endstring* and *startstring*.

If **--recstart** is given *startstring* will be used to split at record start.

If **--recend** is given *endstring* will be used to split at record end.

If both **--recstart** and **--recend** are given the combined string *endstringstartstring* will have to match to find a split position. This is useful if either *startstring* or *endstring* match in the middle of a record.

If neither **--recstart** nor **--recend** are given, then **--recend** defaults to '\n'. To have no record separator (e.g. for binary files) use **--recend ""**.

--recstart and **--recend** are used with **--pipe**.

Use **--regexp** to interpret **--recstart** and **--recend** as regular expressions. This is slow,

however.Use **--remove-rec-sep** to remove **--recstart** and **--recend** before passing the block to the job.

See also: **--pipe --regexp --remove-rec-sep**

--regexp

Use **--regexp** to interpret **--recstart** and **--recend** as regular expressions. This is slow, however.

See also: **--pipe --regexp --remove-rec-sep --recstart --recend**

--remove-rec-sep

--removerecsep

--rrs

Remove record separator.

Remove the text matched by **--recstart** and **--recend** before piping it to the command.

Only used with **--pipe/--pipe-part**.

See also: **--pipe --regexp --pipe-part --recstart --recend**

--results *name*

--res *name*

Save the output into files.

Simple string output dir

If *name* does not contain replacement strings and does not end in **.csv/.tsv**, the output will be stored in a directory tree rooted at *name*. Within this directory tree, each command will result in three files: *name*/<ARGS>/stdout and *name*/<ARGS>/stderr, *name*/<ARGS>/seq, where <ARGS> is a sequence of directories representing the header of the input source (if using **--header :**) or the number of the input source and corresponding values.

E.g:

```
parallel --header : --results foo echo {a} {b} \
  ::: a I II ::: b III IIII
```

will generate the files:

```
foo/a/II/b/III/seq
foo/a/II/b/III/stderr
foo/a/II/b/III/stdout
foo/a/II/b/IIII/seq
foo/a/II/b/IIII/stderr
foo/a/II/b/IIII/stdout
foo/a/I/b/III/seq
foo/a/I/b/III/stderr
foo/a/I/b/III/stdout
foo/a/I/b/IIII/seq
foo/a/I/b/IIII/stderr
foo/a/I/b/IIII/stdout
```

and

```
parallel --results foo echo {1} {2} ::: I II ::: III IIII
```

will generate the files:

```
foo/1/II/2/III/seq
foo/1/II/2/III/stderr
foo/1/II/2/III/stdout
foo/1/II/2/IIII/seq
```

```
foo/1/II/2/IIII/stderr
foo/1/II/2/IIII/stdout
foo/1/I/2/III/seq
foo/1/I/2/III/stderr
foo/1/I/2/III/stdout
foo/1/I/2/IIII/seq
foo/1/I/2/IIII/stderr
foo/1/I/2/IIII/stdout
```

CSV file output

If *name* ends in **.csv**/**.tsv** the output will be a CSV-file named *name*.

.csv gives a comma separated value file. **.tsv** gives a TAB separated value file.

-.csv/**-.tsv** are special: It will give the file on stdout (standard output).

JSON file output

If *name* ends in **.json** the output will be a JSON-file named *name*.

-.json is special: It will give the file on stdout (standard output).

Replacement string output file

If *name* contains a replacement string and the replaced result does not end in /, then the standard output will be stored in a file named by this result. Standard error will be stored in the same file name with '.err' added, and the sequence number will be stored in the same file name with '.seq' added.

E.g.

```
parallel --results my_{} echo ::: foo bar baz
```

will generate the files:

```
my_bar
my_bar.err
my_bar.seq
my_baz
my_baz.err
my_baz.seq
my_foo
my_foo.err
my_foo.seq
```

Replacement string output dir

If *name* contains a replacement string and the replaced result ends in /, then output files will be stored in the resulting dir.

E.g.

```
parallel --results my_{}/ echo ::: foo bar baz
```

will generate the files:

```
my_bar/seq
my_bar/stderr
my_bar/stdout
my_baz/seq
my_baz/stderr
my_baz/stdout
my_foo/seq
my_foo/stderr
my_foo/stdout
```

See also: **--output-as-files --tag --header --joblog**

--resume

Resumes from the last unfinished job.

By reading **--joblog** or the **--results** dir GNU **parallel** will figure out the last unfinished job and continue from there. As GNU **parallel** only looks at the sequence numbers in **--joblog** then the input, the command, and **--joblog** all have to remain unchanged; otherwise GNU **parallel** may run wrong commands.

See also: **--joblog --results --resume-failed --retries**

--resume-failed

Retry all failed and resume from the last unfinished job.

By reading **--joblog** GNU **parallel** will figure out the failed jobs and run those again. After that it will resume last unfinished job and continue from there. As GNU **parallel** only looks at the sequence numbers in **--joblog** then the input, the command, and **--joblog** all have to remain unchanged; otherwise GNU **parallel** may run wrong commands.

See also: **--joblog --resume --retry-failed --retries**

--retry-failed

Retry all failed jobs in joblog.

By reading **--joblog** GNU **parallel** will figure out the failed jobs and run those again.

--retry-failed ignores the command and arguments on the command line: It only looks at the joblog.

Differences between --resume, --resume-failed, --retry-failed

In this example **exit {= $_%=2 =}** will cause every other job to fail.

```
timeout -k 1 4 parallel --joblog log -j10 \
  'sleep {}; exit {= $_%=2 =}' ::: {10..1}
```

4 jobs completed. 2 failed:

```
Seq [...] Exitval Signal Command
10 [...] 1 0 sleep 1; exit 1
9 [...] 0 0 sleep 2; exit 0
8 [...] 1 0 sleep 3; exit 1
7 [...] 0 0 sleep 4; exit 0
```

--resume does not care about the Exitval, but only looks at Seq. If the Seq is run, it will not be run again. So if needed, you can change the command for the seqs not run yet:

```
parallel --resume --joblog log -j10 \
  'sleep .{}; exit {= $_%=2 =}' ::: {10..1}

Seq [...] Exitval Signal Command
[... as above ...]
1 [...] 0 0 sleep .10; exit 0
6 [...] 1 0 sleep .5; exit 1
5 [...] 0 0 sleep .6; exit 0
4 [...] 1 0 sleep .7; exit 1
3 [...] 0 0 sleep .8; exit 0
2 [...] 1 0 sleep .9; exit 1
```

--resume-failed cares about the Exitval, but also only looks at Seq to figure out which commands to run. Again this means you can change the command, but not the arguments. It will run the failed seqs and the seqs not yet run:

```
parallel --resume-failed --joblog log -j10 \
```

```
  'echo {};sleep .{}; exit {= $_%=3 =}' ::: {10..1}

Seq [...] Exitval Signal Command
[... as above ...]
10 [...] 1 0 echo 1;sleep .1; exit 1
8 [...] 0 0 echo 3;sleep .3; exit 0
6 [...] 2 0 echo 5;sleep .5; exit 2
4 [...] 1 0 echo 7;sleep .7; exit 1
2 [...] 0 0 echo 9;sleep .9; exit 0
```

--retry-failed cares about the Exitval, but takes the command from the joblog. It ignores any arguments or commands given on the command line:

```
parallel --retry-failed --joblog log -j10 this part is ignored

Seq [...] Exitval Signal Command
[... as above ...]
10 [...] 1 0 echo 1;sleep .1; exit 1
6 [...] 2 0 echo 5;sleep .5; exit 2
4 [...] 1 0 echo 7;sleep .7; exit 1
```

See also: **--joblog --resume --resume-failed --retries**

--retries *n* (beta testing)

Try failing jobs *n* times.

If a job fails, retry it on another computer on which it has not failed. Do this *n* times. If there are fewer than *n* computers in **--sshlogin** GNU **parallel** will re-use all the computers. This is useful if some jobs fail for no apparent reason (such as network failure).

n=0 means infinite.

See also: **--term-seq --sshlogin**

--return *filename* (beta testing)

Transfer files from remote computers.

--return is used with **--sshlogin** when the arguments are files on the remote computers. When processing is done the file *filename* will be transferred from the remote computer using **rsync** and will be put relative to the default login dir. E.g.

```
echo foo/bar.txt | parallel --return {.}.out \
  --sshlogin server.example.com touch {.}.out
```

This will transfer the file *$HOME/foo/bar.out* from the computer *server.example.com* to the file *foo/bar.out* after running **touch foo/bar.out** on *server.example.com*.

```
parallel -S server --trc out/./{}.out touch {}.out ::: in/file
```

This will transfer the file *in/file.out* from the computer *server.example.com* to the files *out/in/file.out* after running **touch in/file.out** on *server*.

```
echo /tmp/foo/bar.txt | parallel --return {.}.out \
  --sshlogin server.example.com touch {.}.out
```

This will transfer the file */tmp/foo/bar.out* from the computer *server.example.com* to the file */tmp/foo/bar.out* after running **touch /tmp/foo/bar.out** on *server.example.com*.

Multiple files can be transferred by repeating the option multiple times:

```
echo /tmp/foo/bar.txt | parallel \
  --sshlogin server.example.com \
  --return {.}.out --return {.}.out2 touch {.}.out {.}.out2
```

--return is ignored when used with **--sshlogin :** or when not used with **--sshlogin**.

For details on transferring see **--transferfile**.

See also: **--transfer --transferfile --sshlogin --cleanup --workdir**

--round-robin (alpha testing)

--round (alpha testing)

Distribute chunks of standard input in a round robin fashion.

Normally **--pipe** will give a single block to each instance of the command. With **--round-robin** all blocks will at random be written to commands already running. This is useful if the command takes a long time to initialize.

With **--keep-order** and **--round-robin** the jobslots will get the same blocks as input in the same order in every run if the input is kept the same. See details under **--keep-order**.

--round-robin implies **--pipe**, except if **--pipe-part** is given.

See the section: SPREADING BLOCKS OF DATA.

See also: **--bin --group-by --shard**

--rpl '*tag perl expression*'

Define replacement string.

Use *tag* as a replacement string for *perl expression*. This makes it possible to define your own replacement strings. GNU **parallel**'s 7 replacement strings are implemented as:

```
--rpl '{} '
--rpl '{#} 1 $_=$job->seq()'
--rpl '{%} 1 $_=$job->slot()'
--rpl '{/} s:.*/::'
--rpl '{//} $Global::use{"File::Basename"} ||=
           eval "use File::Basename; 1;"; $_ = dirname($_);'
--rpl '{/.} s:.*/::; s:\.[^/.]+$::;'
--rpl '{.} s:\.[^/.]+$::'
```

The **--plus** replacement strings are implemented as:

```
--rpl '{+/} s:/[^/]*$:: || s:.*$::'
--rpl '{+.} s:.*\.:: || s:.*$::'
--rpl '{+..} s:.*\.([^/.]+\.[^/.]+)$:$1: || s:.*$::'
--rpl '{+...} s:.*\.([^/.]+\.[^/.]+\.[^/.]+)$:$1: || s:.*$::'
--rpl '{..} s:\.[^/.]+\.[^/.]+$::'
--rpl '{...} s:\.[^/.]+\.[^/.]+\.[^/.]+$::'
--rpl '{/..} s:.*/::; s:\.[^/.]+\.[^/.]+$::'
--rpl '{/...} s:.*/::; s:\.[^/.]+\.[^/.]+\.[^/.]+$::'
--rpl '{choose_k}
       for $t (2..$#arg){ if($arg[$t-1] ge $arg[$t]) { skip() } }'
--rpl '{##} 1 $_=total_jobs()'
--rpl '{0%} 1 $f=1+int((log($Global::max_jobs_running||1)/
                        log(10))); $_=sprintf("%0${f}d",slot())'
--rpl '{0#} 1 $f=1+int((log(total_jobs())/log(10)));
           $_=sprintf("%0${f}d",seq())'
--rpl '{seq(.*?)} $_=eval q{$job->seq()}.qq{$$1}'
--rpl '{slot(.*?)} $_=eval q{$job->slot()}.qq{$$1}'

--rpl '{:-([^}]+?)} $_ ||= $$1'
--rpl '{:(\d+?)} substr($_,0,$$1) = ""'
--rpl '{:(\d+?):(\d+?)} $_ = substr($_,$$1,$$2);'
--rpl '{#([^#}][^}]*?)} $nongreedy=::make_regexp_ungreedy($$1);
                        s/^$nongreedy(.*)/$1/;'
```

```
--rpl '{##([^#}][^}]*?)} s/^$$1//;'
--rpl '{%([^}]+?)} $nongreedy=::make_regexp_ungreedy($$1);
                   s/(.*)$nongreedy$/$1/;'
--rpl '{%%([^}]+?)} s/$$1$//;'
--rpl '{/([^}]+?)/([^}]*?)} s/$$1/$$2/;'
--rpl '{^([^}]+?)} s/^($$1)/uc($1)/e;'
--rpl '{^^([^}]+?)} s/($$1)/uc($1)/eg;'
--rpl '{,([^}]+?)} s/^($$1)/lc($1)/e;'
--rpl '{,,([^}]+?)} s/($$1)/lc($1)/eg;'

--rpl '{slot} 1 $_="\${PARALLEL_JOBSLOT}";uq()'
--rpl '{host} 1 $_="\${PARALLEL_SSHHOST}";uq()'
--rpl '{sshlogin} 1 $_="\${PARALLEL_SSHLOGIN}";uq()'
--rpl '{hgrp} 1 $_="\${PARALLEL_HOSTGROUPS}";uq()'
--rpl '{agrp} 1 $_="\${PARALLEL_ARGHOSTGROUPS}";uq()'
```

If the user defined replacement string starts with '{' it can also be used as a positional replacement string (like **{2.}**).

It is recommended to only change $_ but you have full access to all of GNU **parallel**'s internal functions and data structures.

Here are a few examples:

```
Is the job sequence even or odd?
--rpl '{odd} $_ = seq() % 2 ? "odd" : "even"'
Pad job sequence with leading zeros to get equal width
--rpl '{0#} $f=1+int("".(log(total_jobs())/log(10)));
  $_=sprintf("%0${f}d",seq())'
Job sequence counting from 0
--rpl '{#0} $_ = seq() - 1'
Job slot counting from 2
--rpl '{%1} $_ = slot() + 1'
Remove all extensions
--rpl '{:} s:(\.[^/]+)*$::'
```

You can have dynamic replacement strings by including parenthesis in the replacement string and adding a regular expression between the parenthesis. The matching string will be inserted as $$1:

```
parallel --rpl '{%(.*?)} s/$$1//' echo {%.tar.gz} ::: my.tar.gz
parallel --rpl '{:+(.+?)} s:$$1(\.[^/]+)*$::' \
  echo {:+_file} ::: my_file.tar.gz
parallel -n3 --rpl '{/:+(.*?)} s:.*/(.*)$$1(\.[^/]+)*$:$1:' \
  echo job {#}: {2} {2.} {3/:+_1} ::: a/b.c c/d.e f/g_1.h.i
```

You can even use multiple matches:

```
parallel --rpl '{/(.+?)/(.*?)} s/$$1/$$2/;'
  echo {/replacethis/withthis} {/b/C} ::: a_replacethis_b

parallel --rpl '{(.*?)/(.*?)} $_="$$2$_$$1"' \
  echo {swap/these} ::: -middle-
```

See also: **{=***perl expression***=} --parens**

--rsync-opts *options*

Options to pass on to **rsync**.

Setting **--rsync-opts** takes precedence over setting the environment variable $PARALLEL_RSYNC_OPTS.

--max-chars *max-chars*

-s *max-chars*

Limit length of command.

Use at most *max-chars* characters per command line, including the command and initial-arguments and the terminating nulls at the ends of the argument strings. The largest allowed value is system-dependent, and is calculated as the argument length limit for exec, less the size of your environment. The default value is the maximum.

max-chars can be postfixed with K, M, G, T, P, k, m, g, t, or p (see UNIT PREFIX).

Implies **-X** unless **-m** or **--xargs** is set.

See also: **-X -m --xargs --max-line-length-allowed --show-limits**

--show-limits

Display limits given by the operating system.

Display the limits on the command-line length which are imposed by the operating system and the **-s** option. Pipe the input from /dev/null (and perhaps specify --no-run-if-empty) if you don't want GNU **parallel** to do anything.

See also: **--max-chars --max-line-length-allowed --version**

--semaphore

Work as a counting semaphore.

--semaphore will cause GNU **parallel** to start *command* in the background. When the number of jobs given by **--jobs** is reached, GNU **parallel** will wait for one of these to complete before starting another command.

--semaphore implies **--bg** unless **--fg** is specified.

The command **sem** is an alias for **parallel --semaphore**.

See also: **man sem --bg --fg --semaphore-name --semaphore-timeout --wait**

--semaphore-name *name*

--id *name*

Use **name** as the name of the semaphore.

The default is the name of the controlling tty (output from **tty**).

The default normally works as expected when used interactively, but when used in a script *name* should be set. *$$* or *my_task_name* are often a good value.

The semaphore is stored in ~/.parallel/semaphores/

Implies **--semaphore**.

See also: **man sem --semaphore**

--semaphore-timeout *secs*

--st *secs*

If *secs* > 0: If the semaphore is not released within *secs* seconds, take it anyway.

If *secs* < 0: If the semaphore is not released within *secs* seconds, exit.

secs is in seconds, but can be postfixed with s, m, h, or d (see the section TIME POSTFIXES).

Implies **--semaphore**.

See also: **man sem**

--seqreplace *replace-str*

Use the replacement string *replace-str* instead of **{#}** for job sequence number.

See also: **{#}**

--session

Record names in current environment in **$PARALLEL_IGNORED_NAMES** and exit.

Only used with **env_parallel**. Aliases, functions, and variables with names in **$PARALLEL_IGNORED_NAMES** will not be copied. So you should set variables/function you want copied *after* running **--session**.

It is similar to **--record-env**, but only for this session.

Only supported in **Ash, Bash, Dash, Ksh, Sh, and Zsh**.

See also: **--env --record-env env_parallel**

--shard *shardexpr*

Use *shardexpr* as shard key and shard input to the jobs.

shardexpr is [column number|column name] [perlexpression] e.g.:

```
3
Address
3 $_%=100
Address s/\d//g
```

Each input line is split using **--colsep**. The string of the column is put into $_, the perl expression is executed, the resulting string is hashed so that all lines of a given value is given to the same job slot.

This is similar to sharding in databases.

The performance is in the order of 100K rows per second. Faster if the *shardcol* is small (<10), slower if it is big (>100).

--shard requires **--pipe** and a fixed numeric value for **--jobs**.

See the section: SPREADING BLOCKS OF DATA.

See also: **--bin --group-by --round-robin**

--shebang

--hashbang

GNU **parallel** can be called as a shebang (#!) command as the first line of a script. The content of the file will be treated as inputsource.

Like this:

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

https://ftpmirror.gnu.org/parallel/parallel-20120822.tar.bz2
https://ftpmirror.gnu.org/parallel/parallel-20130822.tar.bz2
https://ftpmirror.gnu.org/parallel/parallel-20140822.tar.bz2
```

--shebang must be set as the first option.

On FreeBSD **env** is needed:

```
#!/usr/bin/env -S parallel --shebang -r wget

https://ftpmirror.gnu.org/parallel/parallel-20120822.tar.bz2
https://ftpmirror.gnu.org/parallel/parallel-20130822.tar.bz2
https://ftpmirror.gnu.org/parallel/parallel-20140822.tar.bz2
```

There are many limitations of shebang (#!) depending on your operating system. See details on https://www.in-ulm.de/~mascheck/various/shebang/

See also: **--shebang-wrap**

--shebang-wrap

GNU **parallel** can parallelize scripts by wrapping the shebang line. If the program can be run like this:

```
cat arguments | parallel the_program
```

then the script can be changed to:

```
#!/usr/bin/parallel --shebang-wrap /original/parser --options
```

E.g.

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

If the program can be run like this:

```
cat data | parallel --pipe the_program
```

then the script can be changed to:

```
#!/usr/bin/parallel --shebang-wrap --pipe /orig/parser --opts
```

E.g.

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

--shebang-wrap must be set as the first option.

See also: **--shebang**

--shell-completion *shell*

Generate shell completion code for interactive shells.

Supported shells: bash zsh.

Use *auto* as *shell* to automatically detect running shell.

Activate the completion code with:

```
zsh% eval "$(parallel --shell-completion auto)"
bash$ eval "$(parallel --shell-completion auto)"
```

Or put this `/usr/share/zsh/site-functions/_parallel`, then `compinit` to generate `~/.zcompdump`:

```
#compdef parallel

(( $+functions[_comp_parallel] )) ||
  eval "$(parallel --shell-completion auto)" &&
  _comp_parallel
```

--shell-quote

Does not run the command but quotes it. Useful for making quoted composed commands for GNU **parallel**.

Multiple **--shell-quote** with quote the string multiple times, so **parallel --shell-quote | parallel --shell-quote** can be written as **parallel --shell-quote --shell-quote**.

See also: **--quote**

--shuf

Shuffle jobs.

When having multiple input sources it is hard to randomize jobs. **--shuf** will generate all jobs, and shuffle them before running them. This is useful to get a quick preview of the results before running the full batch.

Combined with **--halt soon,done=1%** you can run a random 1% sample of all jobs:

```
parallel --shuf --halt soon,done=1% echo ::: {1..100} ::: {1..100}
```

See also: **--halt**

--skip-first-line

Do not use the first line of input (used by GNU **parallel** itself when called with **--shebang**).

--sql *DBURL* (obsolete)

Use **--sql-master** instead.

--sql-master *DBURL*

Submit jobs via SQL server. *DBURL* must point to a table, which will contain the same information as **--joblog**, the values from the input sources (stored in columns V1 .. Vn), and the output (stored in columns Stdout and Stderr).

If *DBURL* is prepended with '+' GNU **parallel** assumes the table is already made with the correct columns and appends the jobs to it.

If *DBURL* is not prepended with '+' the table will be dropped and created with the correct amount of V-columns unless

--sqlmaster does not run any jobs, but it creates the values for the jobs to be run. One or more **--sqlworker** must be run to actually execute the jobs.

If **--wait** is set, GNU **parallel** will wait for the jobs to complete.

The format of a DBURL is:

```
[sql:]vendor://[[user][:pwd]@][host][:port]/[db]/[table]
```

E.g.

```
sql:mysql://hr:hr@localhost:3306/hrdb/jobs
mysql://scott:tiger@my.example.com/pardb/paralleljobs
sql:oracle://scott:tiger@ora.example.com/xe/parjob
postgresql://scott:tiger@pg.example.com/pgdb/parjob
pg:///parjob
sqlite3:///%2Ftmp%2Fpardb.sqlite/parjob
sqlite:///file_in_current_dir.sqlite/my_table
csv:///%2Ftmp%2Fpardb/parjob
csv:///./file_in_current_dir
pg:////
```

Notice how / in the path of sqlite and CSV must be encoded as %2F. Except the last / in CSV which must be a /.

db and *table* defaults to $USER: pg://// = pg:///$USER/$USER

It can also be an alias from ~/.sql/aliases:

```
:myalias mysql:///mydb/paralleljobs
```

See also: **--sql-and-worker --sql-worker --joblog**

--sql-and-worker *DBURL*

Shorthand for: **--sql-master** *DBURL* **--sql-worker** *DBURL*.

See also: **--sql-master --sql-worker**

--sql-worker *DBURL*

Execute jobs via SQL server. Read the input sources variables from the table pointed to by *DBURL*. The *command* on the command line should be the same as given by **--sqlmaster**.

If you have more than one **--sqlworker** jobs may be run more than once.

If **--sqlworker** runs on the local machine, the hostname in the SQL table will not be ':' but

instead the hostname of the machine.

See also: **--sql-master --sql-and-worker**

--ssh *sshcommand* (alpha testing)

GNU **parallel** defaults to using **ssh** for remote access. This can be overridden with **--ssh**. It can also be set on a per server basis with **--sshlogin**.

See also: **--sshlogin**

--ssh-delay *duration*

Delay starting next ssh by *duration*.

GNU **parallel** will not start another ssh for the next *duration*.

duration is in seconds, but can be postfixed with s, m, h, or d.

See also: TIME POSTFIXES **--sshlogin --delay**

--sshlogin *[@hostgroups/][ncpus/][[user][:[password]]@]host[:port][,...]* (alpha testing)

--sshlogin *@hostgroup/*

-S *[@hostgroups/][ncpus/][ssh command][[user][:[password]]@]host[:port][,...]* (alpha testing)

-S *@hostgroup/*

Distribute jobs to remote computers.

The jobs will be run on a list of remote computers.

@hostgroups/

One or more groups this sshlogin belongs to. Multiple groups are separated by '+'. The *sshlogin* will always be added to a hostgroup named the same as *sshlogin*.

If only the *@hostgroup* is given, only the sshlogins in that hostgroup will be used. Multiple *@hostgroup* can be given.

See **--hostgroup**.

Examples: **@prod/**, **@dev+remote/**

ncpus/

Force number of CPU threads.

GNU **parallel** will determine the number of CPUs on the remote computers and run the number of jobs as specified by **-j**. If the number *ncpus* is given GNU **parallel** will use this number for number of CPU threads on the host. Normally *ncpus* will not be needed.

Examples: **4/**, **12/**

ssh command

The *ssh command* to use. The *ssh command* must be followed by a space.

Example: **/usr/bin/lsh -z** , **autossh -C**

user

User name to log in as. Defaults to the current user name.

Examples: **alice**, **bob**

:password

Use *password* for authentication (using **sshpass**). **password** cannot contain space. If *password* is omitted, GNU **parallel** will use **$SSHPASS**. If **:** is omitted use **ssh**'s default authentication. In

this case login must not require a password (**ssh-agent** and **ssh-copy-id** may help with that).

Examples: **:mypassword**, **:**

host

Hostname or IP address of server. (This is what you will use the most).

Examples: **server01**, **10.1.2.3**, **[2001:470:142:4::a]**, **2001:470:142:5::116**.

Ranges of hostnames can be given in [] like this: **server[1,3,8-10]** (for **server1, server3, server8, server9, server10**) or **server[001,003,008-010]** (for **server001, server003, server008, server009, server010**). With Bash's brace expansion you can do: **-S{dev,prod}[001-100]** to get **-Sdev[001-100] -Sprod[001-100]** More []'s are allowed: **server[01-10].cluster[1-5].example.net**

:port

Port number to connect to.

Examples: **:22**, **:2222**.

For IPv6 you can use **p** or **#** instead of **:**.

Examples: **[2001:470:142:4::a]:2222**, **2001:470:142:5::116p2222**, **2001:470:142:5::116#22222**

There are 3 names with special meaning:

:

Means 'no ssh' and will therefore run on the local computer.

..

Read sshlogins from **~/.parallel/sshloginfile** or **$XDG_CONFIG_HOME/parallel/sshloginfile**

-

Read sshlogins from stdin (standard input).

To specify more sshlogins separate the sshlogins by comma, newline (in the same string), or repeat the options multiple times.

GNU **parallel** splits on , (comma) so if your sshlogin contains , (comma) you need to replace it with \, or ,,

See **--sshloginfile** for complete examples.

The remote host must have GNU **parallel** installed.

--sshlogin is known to cause problems with **-m** and **-X**.

See also: **--basefile --transferfile --return --cleanup --trc --sshloginfile --workdir --filter-hosts --ssh**

--sshloginfile *filename* (beta testing)

--slf *filename* (beta testing)

File with sshlogins. The file consists of sshlogins on separate lines. Empty lines and lines starting with '#' are ignored. Example:

```
server.example.com
username@server2.example.com
8/my-8-cpu-server.example.com
2/my_other_username@my-dualcore.example.net
# These servers have SSH running on port 2222
```

```
ssh -p 2222 server.example.net
server01.example.net:2222
4/ssh -p 2222 quadserver.example.net
# Use a different ssh program
myssh -p 2222 -l myusername hexacpu.example.net
# Use a different ssh program with default number of CPUs
//usr/local/bin/myssh -p 2222 -l myusername hexacpu
# Use a different ssh program with 6 CPUs
6//usr/local/bin/myssh -p 2222 -l myusername hexacpu
# Assume 16 CPUs on the local computer
16/:
# Use password for authentication
user:password@host
# Use $SSHPASS for authentication
user:@host
# Use $SSHPASS for authentication and current username
:@host
# Use password for authentication and current username
:password@host
# Login in as bob with :p@ss:w0rd@ as password
bob::p@ss:w0rd@@host
# Put server1 in hostgroup1
@hostgroup1/server1
# Put myusername@server2 in hostgroup1+hostgroup2
@hostgroup1+hostgroup2/myusername@server2
# Force 4 CPUs and put 'ssh -p 2222 server3' in hostgroup1
@hostgroup1/4/ssh -p 2222 server3
# TODO example with ,,
```

When using a different ssh program the last argument must be the hostname.

Multiple **--sshloginfile** are allowed.

GNU **parallel** will first look for the file in current dir; if that fails it look for the file in ~/.parallel.

There are 3 names with special meaning:

..

Read sshlogins from **~/.parallel/sshloginfile**

.

Read sshlogins from **/etc/parallel/sshloginfile**

-

Read sshlogins from stdin (standard input).

If the sshloginfile is changed it will be re-read when a job finishes though at most once per second. This makes it possible to add and remove hosts while running.

This can be used to have a daemon that updates the sshloginfile to only contain servers that are up:

```
cp original.slf tmp2.slf
while [ 1 ] ; do
  nice parallel --nonall -j0 -k --slf original.slf \
    --tag echo | perl 's/\t$//' > tmp.slf
  if diff tmp.slf tmp2.slf; then
    mv tmp.slf tmp2.slf
  fi
  sleep 10
done &
```

```
parallel --slf tmp2.slf ...
```

See also: **--filter-hosts**

--slotreplace *replace-str*

Use the replacement string *replace-str* instead of **{%}** for job slot number.

See also: **{%}**

--silent

Silent.

The job to be run will not be printed. This is the default. Can be reversed with **-v**.

See also: **-v**

--template *file=repl*

--tmpl *file=repl*

Replace replacement strings in *file* and save it in *repl*.

All replacement strings in the contents of *file* will be replaced. All replacement strings in the name *repl* will be replaced.

With **--cleanup** the new file will be removed when the job is done.

If *my.tmpl* contains this:

```
Xval: {x}
Yval: {y}
FixedValue: 9
# x with 2 decimals
DecimalX: {=x $_=sprintf("%.2f",$_) =}
TenX: {=x $_=$_*10 =}
RandomVal: {=1 $_=rand() =}
```

it can be used like this:

```
myprog() { echo Using "$@"; cat "$@"; }
export -f myprog
parallel --cleanup --header : --tmpl my.tmpl={#}.t myprog {#}.t \
  ::: x 1.234 2.345 3.45678 ::: y 1 2 3
```

See also: **{} --cleanup**

--tty

Open terminal tty.

If GNU **parallel** is used for starting a program that accesses the tty (such as an interactive program) then this option may be needed. It will default to starting only one job at a time (i.e. **-j1**), not buffer the output (i.e. **-u**), and it will open a tty for the job.

You can of course override **-j1** and **-u**.

Using **--tty** unfortunately means that GNU **parallel** cannot kill the jobs (with **--timeout**, **--memfree**, or **--halt**). This is due to GNU **parallel** giving each child its own process group, which is then killed. Process groups are dependant on the tty.

See also: **--ungroup --open-tty**

--tag

Tag lines with arguments.

Each output line will be prepended with the arguments and TAB (\t). When combined with **--onall** or **--nonall** the lines will be prepended with the sshlogin instead.

--tag is ignored when using **-u**.

See also: **--tagstring --ctag**

--tagstring *str*

Tag lines with a string.

Each output line will be prepended with *str* and TAB (\t). *str* can contain replacement strings such as **{}**.

--tagstring is ignored when using **-u**, **--onall**, and **--nonall**.

See also: **--tag --ctagstring**

--tee (alpha testing)

Pipe all data to all jobs.

Used with **--pipe**/**--pipe-part** and **:::**.

```
seq 1000 | parallel --pipe --tee -v wc {} ::: -w -l -c
```

How many numbers in 1..1000 contain 0..9, and how many bytes do they fill:

```
seq 1000 | parallel --pipe --tee --tag \
  'grep {1} | wc {2}' ::: {0..9} ::: -l -c
```

How many words contain a..z and how many bytes do they fill?

```
parallel -a /usr/share/dict/words --pipe-part --tee --tag \
  'grep {1} | wc {2}' ::: {a..z} ::: -l -c
```

See also: **::: --pipe --pipe-part**

--term-seq *sequence*

Termination sequence.

When a job is killed due to **--timeout**, **--memfree**, **--halt**, or abnormal termination of GNU **parallel**, *sequence* determines how the job is killed. The default is:

```
TERM,200,TERM,100,TERM,50,KILL,25
```

which sends a TERM signal, waits 200 ms, sends another TERM signal, waits 100 ms, sends another TERM signal, waits 50 ms, sends a KILL signal, waits 25 ms, and exits. GNU **parallel** detects if a process dies before the waiting time is up.

See also: **--halt --timeout --memfree**

--total-jobs *jobs*

--total *jobs*

Provide the total number of jobs for computing ETA which is also used for **--bar**.

Without **--total-jobs** GNU Parallel will read all jobs before starting a job. **--total-jobs** is useful if the input is generated slowly.

See also: **--bar --eta**

--tmpdir *dirname*

Directory for temporary files.

GNU **parallel** normally buffers output into temporary files in /tmp. By setting **--tmpdir** you can use a different dir for the files. Setting **--tmpdir** is equivalent to setting $TMPDIR.

See also: **--compress $TMPDIR $PARALLEL_REMOTE_TMPDIR**

--tmux (alpha testing)

Use **tmux** for output. Start a **tmux** session and run each job in a window in that session. No other output will be produced.

See also: **--tmuxpane**

--tmuxpane (alpha testing)

Use **tmux** for output but put output into panes in the first window. Useful if you want to monitor the progress of less than 100 concurrent jobs.

See also: **--tmux**

--timeout *duration*

Time out for command. If the command runs for longer than *duration* seconds it will get killed as per **--term-seq**.

If *duration* is followed by a % then the timeout will dynamically be computed as a percentage of the median average runtime of successful jobs. Only values > 100% will make sense.

duration is in seconds, but can be postfixed with s, m, h, or d.

See also: TIME POSTFIXES **--term-seq --retries**

--verbose

-t

Print the job to be run on stderr (standard error).

See also: **-v --interactive**

--transfer (beta testing)

Transfer files to remote computers.

Shorthand for: **--transferfile {}**.

See also: **--transferfile**.

--transferfile *filename* (beta testing)

--tf *filename*

Transfer *filename* to remote computers.

--transferfile is used with **--sshlogin** to transfer files to the remote computers. The files will be transferred using **rsync** and will be put relative to the work dir.

The *filename* will normally contain a replacement string.

If the path contains /./ the remaining path will be relative to the work dir (for details: see **rsync**). If the work dir is **/home/user**, the transferring will be as follows:

```
/tmp/foo/bar   => /tmp/foo/bar
tmp/foo/bar    => /home/user/tmp/foo/bar
/tmp/./foo/bar => /home/user/foo/bar
tmp/./foo/bar  => /home/user/foo/bar
```

Examples

This will transfer the file *foo/bar.txt* to the computer *server.example.com* to the file *$HOME/foo/bar.txt* before running **wc foo/bar.txt** on *server.example.com*:

```
echo foo/bar.txt | parallel --transferfile {} \
  --sshlogin server.example.com wc
```

This will transfer the file */tmp/foo/bar.txt* to the computer *server.example.com* to the file */tmp/foo/bar.txt* before running **wc /tmp/foo/bar.txt** on *server.example.com*:

```
echo /tmp/foo/bar.txt | parallel --transferfile {} \
  --sshlogin server.example.com wc
```

This will transfer the file */tmp/foo/bar.txt* to the computer *server.example.com* to the file *foo/bar.txt* before running **wc ./foo/bar.txt** on *server.example.com*:

```
echo /tmp/./foo/bar.txt | parallel --transferfile {} \
  --sshlogin server.example.com wc {= s:.*/\./:./: =}
```

--transferfile is often used with **--return** and **--cleanup**. A shorthand for **--transferfile {}** is **--transfer**.

--transferfile is ignored when used with **--sshlogin :** or when not used with **--sshlogin**.

See also: **--workdir --sshlogin --basefile --return --cleanup**

--trc *filename*

Transfer, Return, Cleanup. Shorthand for: **--transfer --return** *filename* **--cleanup**

See also: **--transfer --return --cleanup**

--trim <n|l|r|lr|rl>

Trim white space in input.

n

No trim. Input is not modified. This is the default.

l

Left trim. Remove white space from start of input. E.g. " a bc " -> "a bc ".

r

Right trim. Remove white space from end of input. E.g. " a bc " -> " a bc".

lr

rl

Both trim. Remove white space from both start and end of input. E.g. " a bc " -> "a bc". This is the default if **--colsep** is used.

See also: **--no-run-if-empty {} --colsep**

--ungroup

-u

Ungroup output.

Output is printed as soon as possible and bypasses GNU **parallel** internal processing. This may cause output from different commands to be mixed thus should only be used if you do not care about the output. Compare these:

```
seq 4 | parallel -j0 \
  'sleep {};echo -n start{};sleep {};echo {}end'
seq 4 | parallel -u -j0 \
  'sleep {};echo -n start{};sleep {};echo {}end'
```

It also disables **--tag**. GNU **parallel** outputs faster with **-u**. Compare the speeds of these:

```
parallel seq ::: 300000000 >/dev/null
parallel -u seq ::: 300000000 >/dev/null
parallel --line-buffer seq ::: 300000000 >/dev/null
```

Can be reversed with **--group**.

See also: **--line-buffer --group**

--unsafe

GNU **parallel** tries to be conservative to avoid surprising results. **--unsafe** will allow GNU **parallel** to run in environments and on input that are untested and thus may cause surprising results and even security issues, where an evil attacker can influence the results. Think attacks similar to Shellshock: https://en.wikipedia.org/wiki/Shellshock_(software_bug)

If you are forced to use **--unsafe** all the time for something that is safe, it is time to file a bug report and have a discussion how to make your situation well tested.

--use-sockets-instead-of-threads

See also: **--use-cores-instead-of-threads**

--use-cores-instead-of-threads

--use-cpus-instead-of-cores (obsolete)

Determine how GNU **parallel** counts the number of CPUs.

GNU **parallel** uses this number when the number of jobslots (**--jobs**) is computed relative to the number of CPUs (e.g. 100% or +1).

CPUs can be counted in three different ways:

sockets

The number of filled CPU sockets (i.e. the number of physical chips).

cores

The number of physical cores (i.e. the number of physical compute cores).

threads

The number of hyperthreaded cores (i.e. the number of virtual cores - with some of them possibly being hyperthreaded)

Normally the number of CPUs is computed as the number of CPU threads. With **--use-sockets-instead-of-threads** or **--use-cores-instead-of-threads** you can force it to be computed as the number of filled sockets or number of cores instead.

Most users will not need these options.

--use-cpus-instead-of-cores is a (misleading) alias for **--use-sockets-instead-of-threads** and is kept for backwards compatibility.

See also: **--number-of-threads --number-of-cores --number-of-sockets**

-v

Verbose.

Print the job to be run on stdout (standard output). Can be reversed with **--silent**.

Use **-v -v** to print the wrapping ssh command when running remotely.

See also: **-t**

--version

-V

Print the version GNU **parallel** and exit.

--workdir *mydir*

--wd *mydir*

Jobs will be run in the dir *mydir*. The default is the current dir for the local machine, and the login dir for remote computers.

Files transferred using **--transferfile** and **--return** will be relative to *mydir* on remote computers.

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

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

To see the difference try:

```
parallel -S server pwd ::: ""
```

```
parallel --wd . -S server pwd ::: ""
parallel --wd ... -S server pwd ::: ""
```

mydir can contain GNU **parallel**'s replacement strings.

--wait

Wait for all commands to complete.

Used with **--semaphore** or **--sqlmaster**.

See also: **man sem**

-X

Multiple arguments with context replace. Insert as many arguments as the command line length permits. If multiple jobs are being run in parallel: distribute the arguments evenly among the jobs. Use **-j1** to avoid this.

If **{}** is not used the arguments will be appended to the line. If **{}** is used as part of a word (like *pic{}.jpg*) then the whole word will be repeated. If **{}** is used multiple times each **{}** will be replaced with the arguments.

Normally **-X** will do the right thing, whereas **-m** can give unexpected results if **{}** is used as part of a word.

Support for **-X** with **--sshlogin** is limited and may fail.

See also: **-m**

--exit

-x

Exit if the size (see the **-s** option) is exceeded.

--xargs

Multiple arguments. Insert as many arguments as the command line length permits.

If **{}** is not used the arguments will be appended to the line. If **{}** is used multiple times each **{}** will be replaced with all the arguments.

Support for **--xargs** with **--sshlogin** is limited and may fail.

See also: **-X**

EXAMPLES

See: **man parallel_examples**

SPREADING BLOCKS OF DATA

--round-robin, **--pipe-part**, **--shard**, **--bin** and **--group-by** are all specialized versions of **--pipe**.

In the following *n* is the number of jobslots given by **--jobs**. A record starts with **--recstart** and ends with **--recend**. It is typically a full line. A chunk is a number of full records that is approximately the size of a block. A block can contain half records, a chunk cannot.

--pipe starts one job per chunk. It reads blocks from stdin (standard input). It finds a record end near a block border and passes a chunk to the program.

--pipe-part starts one job per chunk - just like normal **--pipe**. It first finds record endings near all block borders in the file and then starts the jobs. By using **--block -1** it will set the block size to size-of-file/*n*. Used this way it will start *n* jobs in total.

--round-robin starts *n* jobs in total. It reads a block and passes a chunk to whichever job is ready to read. It does not parse the content except for identifying where a record ends to make sure it only passes full records.

--shard starts *n* jobs in total. It parses each line to read the string in the given column. Based on this

string the line is passed to one of the *n* jobs. All lines having this string will be given to the same jobslot.

--bin works like **--shard** but the value of the column must be numeric and is the jobslot number it will be passed to. If the value is bigger than *n*, then *n* will be subtracted from the value until the value is smaller than or equal to *n*.

--group-by starts one job per chunk. Record borders are not given by **--recend**/**--recstart**. Instead a record is defined by a group of lines having the same string in a given column. So the string of a given column changes at a chunk border. With **--pipe** every line is parsed, with **--pipe-part** only a few lines are parsed to find the chunk border.

--group-by can be combined with **--round-robin** or **--pipe-part**.

TIME POSTFIXES

Arguments that give a duration are given in seconds, but can be expressed as floats postfixed with **s**, **m**, **h**, or **d** which would multiply the float by 1, 60, 60*60, or 60*60*24. Thus these are equivalent: 100000 and 1d3.5h16.6m4s.

UNIT PREFIX

Many numerical arguments in GNU **parallel** can be postfixed with K, M, G, T, P, k, m, g, t, or p which would multiply the number with 1024, 1048576, 1073741824, 1099511627776, 1125899906842624, 1000, 1000000, 1000000000, 1000000000000, or 1000000000000000, respectively.

You can even give it as a math expression. E.g. 1000000 can be written as 1M-12*2.024*2k.

QUOTING

GNU **parallel** is very liberal in quoting. You only need to quote characters that have special meaning in shell:

```
( ) $ ` ' " < > ; | \
```

and depending on context these needs to be quoted, too:

```
~ & ! ? space * { #
```

Therefore most people will never need more quoting than putting '\' in front of the special characters.

Often you can simply put \' around every ':

```
perl -ne '/^\S+\s+\S+$/ and print $ARGV,"\n"' file
```

can be quoted:

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

However, when you want to use a shell variable you need to quote the $-sign. Here is an example using $PARALLEL_SEQ. This variable is set by GNU **parallel** itself, so the evaluation of the $ must be done by the sub shell started by GNU **parallel**:

```
seq 10 | parallel -N2 echo seq:\$PARALLEL_SEQ arg1:{1} arg2:{2}
```

If the variable is set before GNU **parallel** starts you can do this:

```
VAR=this_is_set_before_starting
echo test | parallel echo {} $VAR
```

Prints: **test this_is_set_before_starting**

It is a little more tricky if the variable contains more than one space in a row:

```
VAR="two  spaces  between  each  word"
echo test | parallel echo {} \'"$VAR"\'
```

Prints: **test two spaces between each word**

If the variable should not be evaluated by the shell starting GNU **parallel** but be evaluated by the sub shell started by GNU **parallel**, then you need to quote it:

```
echo test | parallel VAR=this_is_set_after_starting \; echo {} \$VAR
```

Prints: **test this_is_set_after_starting**

It is a little more tricky if the variable contains space:

```
echo test |\
  parallel VAR='"two  spaces  between  each  word"' echo {} \'"$VAR"\'
```

Prints: **test two spaces between each word**

$$ is the shell variable containing the process id of the shell. This will print the process id of the shell running GNU **parallel**:

```
seq 10 | parallel echo $$
```

And this will print the process ids of the sub shells started by GNU **parallel**.

```
seq 10 | parallel echo \$\$
```

If the special characters should not be evaluated by the sub shell then you need to protect it against evaluation from both the shell starting GNU **parallel** and the sub shell:

```
echo test | parallel echo {} \\\$VAR
```

Prints: **test $VAR**

GNU **parallel** can protect against evaluation by the sub shell by using -q:

```
echo test | parallel -q echo {} \$VAR
```

Prints: **test $VAR**

This is particularly useful if you have lots of quoting. If you want to run a perl script like this:

```
perl -ne '/^\S+\s+\S+$/ and print $ARGV,"\n"' file
```

It needs to be quoted like one of these:

```
ls | parallel perl -ne '/^\\S+\\s+\\S+\$/\ and\ print\ \$ARGV,\"\\n\"'
ls | parallel perl -ne \''/^\S+\s+\S+$/ and print $ARGV,"\n"'\'
```

Notice how spaces, \'s, "'s, and $'s need to be quoted. GNU **parallel** can do the quoting by using option -q:

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

However, this means you cannot make the sub shell interpret special characters. For example because of **-q** this WILL NOT WORK:

```
ls *.gz | parallel -q "zcat {} >{.}"
ls *.gz | parallel -q "zcat {} | bzip2 >{.}.bz2"
```

because > and | need to be interpreted by the sub shell.

If you get errors like:

```
sh: -c: line 0: syntax error near unexpected token
sh: Syntax error: Unterminated quoted string
sh: -c: line 0: unexpected EOF while looking for matching `''
sh: -c: line 1: syntax error: unexpected end of file
zsh:1: no matches found:
```

then you might try using **-q**.

If you are using **bash** process substitution like **<(cat foo)** then you may try **-q** and prepending *command* with **bash -c**:

```
ls | parallel -q bash -c 'wc -c <(echo {})'
```

Or for substituting output:

```
ls | parallel -q bash -c \
  'tar c {} | tee >(gzip >{}.tar.gz) | bzip2 >{}.tar.bz2'
```

Conclusion: If this is confusing consider avoiding having to deal with quoting by writing a small script or a function (remember to **export -f** the function) and have GNU **parallel** call that.

LIST RUNNING JOBS

If you want a list of the jobs currently running you can run:

```
killall -USR1 parallel
```

GNU **parallel** will then print the currently running jobs on stderr (standard error).

COMPLETE RUNNING JOBS BUT DO NOT START NEW JOBS

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

```
killall -HUP parallel
```

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

ENVIRONMENT VARIABLES

$PARALLEL_HOME

> Dir where GNU **parallel** stores config files, semaphores, and caches information between invocations. If set to a non-existent dir, the dir will be created.
>
> Default: $HOME/.parallel.

$PARALLEL_ARGHOSTGROUPS

> When using **--hostgroups** GNU **parallel** sets this to the hostgroups of the job.
>
> Remember to quote the $, so it gets evaluated by the correct shell. Or use **--plus** and {agrp}.

$PARALLEL_HOSTGROUPS

When using **--hostgroups** GNU **parallel** sets this to the hostgroups of the sshlogin that the job is run on.

Remember to quote the $, so it gets evaluated by the correct shell. Or use **--plus** and {hgrp}.

$PARALLEL_JOBSLOT

Set by GNU **parallel** and can be used in jobs run by GNU **parallel**. Remember to quote the $, so it gets evaluated by the correct shell. Or use **--plus** and {slot}.

$PARALLEL_JOBSLOT is the jobslot of the job. It is equal to {%} unless the job is being retried. See {%} for details.

$PARALLEL_PID

Set by GNU **parallel** and can be used in jobs run by GNU **parallel**. Remember to quote the $, so it gets evaluated by the correct shell.

This makes it possible for the jobs to communicate directly to GNU **parallel**.

Example: If each of the jobs tests a solution and one of jobs finds the solution the job can tell GNU **parallel** not to start more jobs by: **kill -HUP $PARALLEL_PID**. This only works on the local computer.

$PARALLEL_RSYNC_OPTS

Options to pass on to **rsync**. Defaults to: -rlDzR.

$PARALLEL_SHELL

Use this shell for the commands run by GNU **parallel**:

- $PARALLEL_SHELL. If undefined use:
- The shell that started GNU **parallel**. If that cannot be determined:
- $SHELL. If undefined use:
- /bin/sh

$PARALLEL_SSH

GNU **parallel** defaults to using the **ssh** command for remote access. This can be overridden with $PARALLEL_SSH, which again can be overridden with **--ssh**. It can also be set on a per server basis (see **--sshlogin**).

$PARALLEL_SSHHOST

Set by GNU **parallel** and can be used in jobs run by GNU **parallel**. Remember to quote the $, so it gets evaluated by the correct shell. Or use **--plus** and {host}.

$PARALLEL_SSHHOST is the host part of an sshlogin line. E.g.

```
4//usr/bin/specialssh user@host
```

becomes:

```
host
```

$PARALLEL_SSHLOGIN

Set by GNU **parallel** and can be used in jobs run by GNU **parallel**. Remember to quote the $, so it gets evaluated by the correct shell. Or use **--plus** and {sshlogin}.

The value is the sshlogin line with number of threads removed. E.g.

```
4//usr/bin/specialssh user@host
```

becomes:

```
/usr/bin/specialssh user@host
```

$PARALLEL_SEQ

Set by GNU **parallel** and can be used in jobs run by GNU **parallel**. Remember to quote the $, so it gets evaluated by the correct shell.

$PARALLEL_SEQ is the sequence number of the job running.

Example:

```
seq 10 | parallel -N2 \
  echo seq:'$'PARALLEL_SEQ arg1:{1} arg2:{2}
```

{#} is a shorthand for $PARALLEL_SEQ.

$PARALLEL_TMUX

Path to **tmux**. If unset the **tmux** in $PATH is used.

$TMPDIR

Directory for temporary files.

See also: **--tmpdir**

$PARALLEL_REMOTE_TMPDIR

Directory for temporary files on remote servers.

See also: **--tmpdir**

$PARALLEL

The environment variable $PARALLEL will be used as default options for GNU **parallel**. If the variable contains special shell characters (e.g. $, *, or space) then these need to be to be escaped with \.

Example:

```
cat list | parallel -j1 -k -v ls
cat list | parallel -j1 -k -v -S"myssh user@server" ls
```

can be written as:

```
cat list | PARALLEL="-kvj1" parallel ls
cat list | PARALLEL='-kvj1 -S myssh\ user@server' \
  parallel echo
```

Notice the \ after 'myssh' is needed because 'myssh' and 'user@server' must be one argument.

See also: **--profile**

DEFAULT PROFILE (CONFIG FILE)

The global configuration file /etc/parallel/config, followed by user configuration file ~/.parallel/config (formerly known as .parallelrc) will be read in turn if they exist. Lines starting with '#' will be ignored. The format can follow that of the environment variable $PARALLEL, but it is often easier to simply put each option on its own line.

Options on the command line take precedence, followed by the environment variable $PARALLEL, user configuration file ~/.parallel/config, and finally the global configuration file /etc/parallel/config.

Note that no file that is read for options, nor the environment variable $PARALLEL, may contain retired options such as **--tollef**.

PROFILE FILES

If **--profile** set, GNU **parallel** will read the profile from that file rather than the global or user configuration files. You can have multiple **--profiles**.

Profiles are searched for in **~/.parallel**. If the name starts with **/** it is seen as an absolute path. If the name starts with **./** it is seen as a relative path from current dir.

Example: Profile for running a command on every sshlogin in ~/.ssh/sshlogins and prepend the output with the sshlogin:

```
echo --tag -S .. --nonall > ~/.parallel/nonall_profile
parallel -J nonall_profile uptime
```

Example: Profile for running every command with **-j-1** and **nice**

```
echo -j-1 nice > ~/.parallel/nice_profile
parallel -J nice_profile bzip2 -9 ::: *
```

Example: Profile for running a perl script before every command:

```
echo "perl -e '\$a=\$\$; print \$a,\" \",'\$PARALLEL_SEQ',\" \";';" \
  > ~/.parallel/pre_perl
parallel -J pre_perl echo ::: *
```

Note how the $ and " need to be quoted using \.

Example: Profile for running distributed jobs with **nice** on the remote computers:

```
echo -S .. nice > ~/.parallel/dist
parallel -J dist --trc {.}.bz2 bzip2 -9 ::: *
```

EXIT STATUS

Exit status depends on **--halt-on-error** if one of these is used: success=X, success=Y%, fail=Y%.

0 All jobs ran without error. If success=X is used: X jobs ran without error. If success=Y% is used: Y% of the jobs ran without error.

1-100

Some of the jobs failed. The exit status gives the number of failed jobs. If Y% is used the exit status is the percentage of jobs that failed.

101 More than 100 jobs failed.

255 Other error.

-1 (In joblog and SQL table)

Killed by Ctrl-C, timeout, not enough memory or similar.

-2 (In joblog and SQL table)

skip() was called in **{= =}**.

-1000 (In SQL table)

Job is ready to run (set by --sqlmaster).

-1220 (In SQL table)

Job is taken by worker (set by --sqlworker).

If fail=1 is used, the exit status will be the exit status of the failing job.

DIFFERENCES BETWEEN GNU Parallel AND ALTERNATIVES

See: **man parallel_alternatives**

BUGS

Quoting of newline

Because of the way newline is quoted this will not work:

```
echo 1,2,3 | parallel -vkd, "echo 'a{}b'"
```

However, these will all work:

```
echo 1,2,3 | parallel -vkd, echo a{}b
echo 1,2,3 | parallel -vkd, "echo 'a'{}'b'"
echo 1,2,3 | parallel -vkd, "echo 'a'"{}"'b'"
```

Speed

Startup

GNU **parallel** is slow at starting up - around 250 ms the first time and 150 ms after that.

Job startup

Starting a job on the local machine takes around 3-10 ms. This can be a big overhead if the job takes very few ms to run. Often you can group small jobs together using **-X** which will make the overhead less significant. Or you can run multiple GNU **parallel**s as described in **EXAMPLE: Speeding up fast jobs**.

SSH

When using multiple computers GNU **parallel** opens **ssh** connections to them to figure out how many connections can be used reliably simultaneously (Namely SSHD's MaxStartups). This test is done for each host in serial, so if your **--sshloginfile** contains many hosts it may be slow.

If your jobs are short you may see that there are fewer jobs running on the remote systems than expected. This is due to time spent logging in and out. **-M** may help here.

Disk access

A single disk can normally read data faster if it reads one file at a time instead of reading a lot of files in parallel, as this will avoid disk seeks. However, newer disk systems with multiple drives can read faster if reading from multiple files in parallel.

If the jobs are of the form read-all-compute-all-write-all, so everything is read before anything is written, it may be faster to force only one disk access at the time:

```
sem --id diskio cat file | compute | sem --id diskio cat > file
```

If the jobs are of the form read-compute-write, so writing starts before all reading is done, it may be faster to force only one reader and writer at the time:

```
sem --id read cat file | compute | sem --id write cat > file
```

If the jobs are of the form read-compute-read-compute, it may be faster to run more jobs in parallel than the system has CPUs, as some of the jobs will be stuck waiting for disk access.

--nice limits command length

The current implementation of **--nice** is too pessimistic in the max allowed command length. It only uses a little more than half of what it could. This affects **-X** and **-m**. If this becomes a real problem for you, file a bug-report.

Aliases and functions do not work

If you get:

```
Can't exec "command": No such file or directory
```

or:

```
open3: exec of by command failed
```

or:

```
/bin/bash: command: command not found
```

it may be because *command* is not known, but it could also be because *command* is an alias or a function. If it is a function you need to **export -f** the function first or use **env_parallel**. An alias will only work if you use **env_parallel**.

Database with MySQL fails randomly

The **--sql*** options may fail randomly with MySQL. This problem does not exist with PostgreSQL.

REPORTING BUGS

Report bugs to <parallel@gnu.org> or https://savannah.gnu.org/bugs/?func=additem&group=parallel

When you write your report, please keep in mind, that you must give the reader enough information to be able to run exactly what you run. So you need to include all data and programs that you use to show the problem.

See a perfect bug report on https://lists.gnu.org/archive/html/bug-parallel/2015-01/msg00000.html

Your bug report should always include:

- The error message you get (if any). If the error message is not from GNU **parallel** you need to show why you think GNU **parallel** caused this.
- The complete output of **parallel --version**. If you are not running the latest released version (see https://ftp.gnu.org/gnu/parallel/) you should specify why you believe the problem is not fixed in that version.
- A minimal, complete, and verifiable example (See description on https://stackoverflow.com/help/mcve).

  It should be a complete example that others can run which shows the problem including all files needed to run the example. This should preferably be small and simple, so try to remove as many options as possible.

  A combination of **yes**, **seq**, **cat**, **echo**, **wc**, and **sleep** can reproduce most errors.

  If your example requires large files, see if you can make them with something like **seq 100000000 > bigfile** or **yes | head -n 1000000000 > file**. If you need multiple columns: **paste <(seq 1000) <(seq 1000 1999)**

  If your example requires remote execution, see if you can use **localhost** - maybe using another login.

  If you have access to a different system (maybe a VirtualBox on your own machine), test if your MCVE shows the problem on that system. If it does not, read below.
- The output of your example and what you expected instead. If your problem is not easily reproduced by others, the output might help them figure out the problem.
- Whether you have watched the intro videos (https://www.youtube.com/playlist?list=PL284C9FF2488BC6D1), walked through the tutorial (man parallel_tutorial), and read the examples (man parallel_examples).

Bug dependent on environment

If you suspect the error is dependent on your environment or distribution, please see if you can reproduce the error on one of these VirtualBox images: https://sourceforge.net/projects/virtualboximage/files/ https://www.osboxes.org/virtualbox-images/

Specifying the name of your distribution is not enough as you may have installed software that is not in the VirtualBox images.

If you cannot reproduce the error on any of the VirtualBox images above, see if you can build a VirtualBox image on which you can reproduce the error. If not you should assume the debugging will be done through you. That will put a lot more burden on you and it is extra important you give any information that help. In general the problem will be fixed faster and with much less work for you if you can reproduce the error on a VirtualBox - even if you have to build a VirtualBox image.

In summary

Your report must include:

- **parallel --version**
- output + error message
- full example including all files
- VirtualBox image, if you cannot reproduce it on other systems

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 **--csv** it uses the Perl module Text::CSV.

For remote usage it uses **rsync** with **ssh**.

SEE ALSO

parallel_tutorial(1), **env_parallel**(1), **parset**(1), **parsort**(1), **parallel_alternatives**(1), **parallel_design**(7), **niceload**(1), **sql**(1), **ssh**(1), **ssh-agent**(1), **sshpass**(1), **ssh-copy-id**(1), **rsync**(1)