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 | **::::** argfile(s)) ...

parallel --semaphore [options] *command*

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

DESCRIPTION

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

Start by watching the intro videos for a quick introduction:
http://www.youtube.com/playlist?list=PL284C9FF2488BC6D1

Then look at the **EXAMPLE**s after the list of **OPTIONS**. That will give you an idea of what GNU **parallel** is capable of.

Then spend an hour walking through the tutorial (**man parallel_tutorial**). Your command line will love you for it.

Finally you may want to look at the rest of this manual if you have special needs not already covered.

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, or a function. If it is a Bash function you need to **export -f** the function first. An alias will, however, not work (see why http://www.perlmonks.org/index.pl?node_id=484296).

{}

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 **-a**, **:::**, 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.

{.}

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 **.** till 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 **--er**.

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

{/}

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

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

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

{//}

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

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

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

{/.}

Basename of input line without extension. This replacement string will be replaced by the input with the directory and extension part removed. It is a combination of **{/}** and **{.}**.

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

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

{#}

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.

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

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

{*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 **-a** or **::::**) or with the *n*'th argument (when used with **-N**). If *n* is negative it refers to the *n*'th last argument.

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

{*n***.}**

Argument from input source *n* or the *n*'th argument without extension. It is a combination of **{***n***}** and **{.}**.

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

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

{*n***/}**

Basename of argument from input source *n* or the *n*'th argument. It is a combination of **{***n***}** and **{/}**.

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

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

{*n***//}**

Dirname of argument from input source *n* or the *n*'th argument. It 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 **-a** or **::::**) or with the *n*'th argument (when used with **-N**). See **dirname**(1).

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

{*n***/.}**

Basename of argument from input source *n* or the *n*'th argument without extension. It is a combination of **{***n***}**, **{/}**, and **{.}**.

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

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

::: *arguments*

Use arguments from the command line as input source instead of stdin (standard input). 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. See also **--arg-sep**.

stdin (standard input) will be passed to the first process run.

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 **::::** 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
```

:::: *argfiles*

Another way to write **-a** *argfile1* **-a** *argfile2* ...

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

See **-a**, **:::** and **--xapply**.

--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).

--arg-file *input-file*

-a *input-file*

Use *input-file* as input source. If you use this option, stdin (standard input) is given to the first process run. Otherwise, stdin (standard input) is redirected from /dev/null.

If multiple **-a** 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.

See also **--xapply** and **{*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 (beta testing)

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> >(zenity --progress --auto-kill) | wc

--basefile *file* (beta testing)

--bf *file* (beta testing)

file will be transferred to each sshlogin before a jobs 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 jobs. Multiple **--bf** can be specified to transfer more basefiles. The *file* will be transferred the same way as **--transfer**.

--basenamereplace *replace-str*

--bnr *replace-str*

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

--basenameextensionreplace *replace-str*

--bner *replace-str*

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

--bg

Run command in background thus GNU **parallel** will not wait for completion of the command before exiting. This is the default if **--semaphore** is set.

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

Implies **--semaphore**.

--bibtex (beta testing)

Print the BibTeX entry for GNU **parallel** and disable citation notice.

--block *size*

--block-size *size*

Size of block in bytes. The size can be postfixed with K, M, G, T, P, k, m, g, t, or p which would multiply the size with 1024, 1048576, 1073741824, 1099511627776, 1125899906842624, 1000, 1000000, 1000000000, 1000000000000, or 1000000000000000 respectively.

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 single record.

size defaults to 1M.

See **--pipe** for use of this.

--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 --transfer --return {.}.bz2
\
    --cleanup "zcat {} | bzip -9 >{.}.bz2"
```

With **--transfer** the file transferred to the remote computer will be removed on the remote computer. Directories created will not be removed - even if they are empty.

With **--return** the file transferred from the remote computer will be removed on the remote computer. Directories created will not be removed - even if they are empty.

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

--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 access using **{***n***}** or **{***n***.}**. E.g. **{3}** is the 3rd column.

--colsep implies **--trim rl**.

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

--compress (beta testing)

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 **lzop**, **pigz**, **gzip**, **pbzip2**, **plzip**, **bzip2**, **lzma**, **lzip**, **xz** in that

order, and use the first available.

--compress-program *prg* (beta testing)

Use *prg* for compressing temporary files. It is assumed that *prg -dc* will decompress stdin (standard input) to stdout (standard output).

--ctrlc

Sends SIGINT to tasks running on remote computers thus killing them.

--delimiter *delim*

-d *delim*

Input items are terminated by the specified character. Quotes and backslash are not special; every character in the input is taken literally. Disables the end-of-file string, which is treated like any other argument. This can be used when the input consists of simply newline-separated items, although it is almost always better to design your program to use --null where this is possible. The specified delimiter may be a single character, a C-style character escape such as \n, or an octal or hexadecimal escape code. Octal and hexadecimal escape codes are understood as for the printf command. Multibyte characters are not supported.

--dirnamereplace *replace-str*

--dnr *replace-str*

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

-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 ignored. If neither **-E** nor **-e** is used, no end of file string is used.

--delay *secs*

Delay starting next job *secs* seconds. GNU **parallel** will pause *secs* seconds after starting each job. *secs* can be less than 1 seconds.

--dry-run

Print the job to run on stdout (standard output), but do not run the job. Use **-v -v** to include the ssh/rsync wrapping if the job would be run on a remote computer. Do not count on this literaly, though, as the job may be scheduled on another computer or the local computer if : is in the list.

--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.

--env *var*

Copy 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 enviroment variables except for the ones mentioned in ~/.parallel/ignored_vars.

See also: **--record-env**.

--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. GNU **parallel** normally only

reads the next job to run. Implies **--progress**.

--fg

Run command in foreground thus GNU **parallel** will wait for completion of the command before exiting.

See also **--bg**, **man sem**.

Implies **--semaphore**.

--filter-hosts (alpha testing)

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

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.

--gnu

Behave like GNU **parallel**. If **--tollef** and **--gnu** are both set, **--gnu** takes precedence.

--group

Group output. Output from each jobs is grouped together and is only printed when the command is finished. stderr (standard error) first followed by stdout (standard output). This takes some CPU time. In rare situations GNU **parallel** takes up lots of CPU time and if it is acceptable that the outputs from different commands are mixed together, then disabling grouping with **-u** can speedup GNU **parallel** by a factor of 10.

--group is the default. Can be reversed with **-u**.

--help

-h

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

--halt-on-error <0|1|2>

--halt <0|1|2>

0 Do not halt if a job fails. Exit status will be the number of jobs failed. This is the default.

1 Do not start new jobs if a job fails, but complete the running jobs including cleanup. The exit status will be the exit status from the last failing job.

2 Kill off all jobs immediately and exit without cleanup. The exit status will be the exit status from the failing job.

--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}**.

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

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

If *regexp* is a number, it will match that many lines.

-I *replace-str*

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

--replace[=*replace-str*]

-i[*replace-str*]

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

--joblog *logfile* (beta testing)

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.

To convert the times into ISO-8601 strict do:

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

See also **--resume**.

--jobs *N*

-j *N*

--max-procs *N*

-P *N*

Number of jobslots. Run up to N jobs in parallel. 0 means as many as possible. Default is 100% which will run one job per CPU core.

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

--jobs *+N*

-j *+N*

--max-procs *+N*

-P *+N*

Add N to the number of CPU cores. Run this many jobs in parallel. See also **--use-cpus-instead-of-cores**.

--jobs *-N*

-j *-N*

--max-procs *-N*

-P *-N*

Subtract N from the number of CPU cores. Run this many jobs in parallel. If the evaluated number is less than 1 then 1 will be used. See also **--use-cpus-instead-of-cores**.

--jobs *N*%

-j *N*%

--max-procs *N*%

-P *N*%

Multiply N% with the number of CPU cores. Run this many jobs in parallel. If the evaluated number is less than 1 then 1 will be used. See also **--use-cpus-instead-of-cores**.

--jobs *procfile*

-j *procfile*

--max-procs *procfile*

-P *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.

--keep-order

-k

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** output will be sorted according to sshlogin.

-L *max-lines*

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

When used otherwise: Use at most *max-lines* 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.

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

--max-lines[=*max-lines*]

-l[*max-lines*]

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

When used otherwise: Synonym for the **-L** option. Unlike **-L**, the *max-lines* argument is optional. If *max-lines* 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.

--line-buffer

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 is slower than both **--group** and **--ungroup**.

--load *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.

--controlmaster (experimental)

-M (experimental)

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.

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

-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** 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.

See also **-X** for context replace. If in doubt use **-X** as that will most likely do what is needed.

--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**.

--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.

--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. **-j** adjusts how many computers to log into in parallel.

When using **--group** the output will be grouped by each server, so all the output from one server will be grouped together.

--output-as-files (beta testing)

--outputasfiles (beta testing)

--files (beta testing)

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.

--pipe

--spreadstdin

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**. The strings **--recstart** and **--recend** tell GNU **parallel** how a record starts and/or ends. The block read will have the final partial record removed before the block is passed on to the job. The partial record will be prepended to next block.

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

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

If both **--recstart** and **--recend** are given both will have to match to find a split position.

If neither **--recstart** nor **--recend** are given **--recend** defaults to '\n'. To have no record separator use **--recend ""**.

--files is often used with **--pipe**.

--plain

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

--progress

Show progress of computations. List the computers involved in the task with number of CPU cores 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**.

--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.

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

--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.

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

--max-line-length-allowed

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).

--number-of-cpus

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

--number-of-cores

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

--no-notice (beta testing)

Do not display citation notice. A citation notice is printed on stderr (standard error) only if stderr (standard error) is a terminal, the user has not specified **--no-notice**, and the user has not run **--bibtex** once.

--nice *niceness*

Run the command at this niceness. For simple commands you can just add **nice** in front of the command. But if the command consists of more sub commands (Like: ls|wc) then prepending **nice** will not always work. **--nice** will make sure all sub commands are niced.

--interactive

-p

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**.

--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. See the section PROFILE FILES for examples.

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: config

--quote

-q

Quote *command*. This will quote the command line so special characters are not interpreted by the shell. See the section QUOTING. Most people will never need this. Quoting is disabled by default.

--no-run-if-empty

-r

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

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

--record-env

Record current environment variables in ~/.parallel/ignored_vars. This is useful before using **--env** _.

See also **--env**.

--recstart *startstring*

--recend *endstring*

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 use **--recend ""**.

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

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

--regexp

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

--remove-rec-sep

--removerecsep

--rrs

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

Only used with **--pipe**.

--results *prefix*

--res *prefix*

Save the output into files. The files will be stored in a directory tree rooted at *prefix*. Within this directory tree, each command will result in two files: *prefix* /<ARGS>/stdout and *prefix*/<ARGS>/stderr, 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/I/b/III/stderr
  foo/a/I/b/IIII/stderr
  foo/a/II/b/III/stderr
  foo/a/II/b/IIII/stderr
  foo/a/I/b/III/stdout
  foo/a/I/b/IIII/stdout
  foo/a/II/b/III/stdout
  foo/a/II/b/IIII/stdout
```

and

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

will generate the files:

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

See also **--files**, **--header**, **--joblog**.

--resume (beta testing)

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**.

--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**.

--retries *n*

If a job fails, retry it on another computer. Do this *n* times. If there are fewer than *n* computers in **--sshlogin** GNU **parallel** will re-use the computers. This is useful if some jobs fail for no apparent reason (such as network failure).

--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 \
    --sshlogin server.example.com --return {.}.out 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*.

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

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

--return is often used with **--transfer** and **--cleanup**.

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

--round-robin

--round

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.

--keep-order will not work with **--round-robin** as it is impossible to track which input block corresponds to which output.

--max-chars=*max-chars*

-s *max-chars*

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.

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

--show-limits

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.

--semaphore

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

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

--semaphore implies **--semaphorename `tty`** unless **--semaphorename** is specified.

Used with **--fg**, **--wait**, and **--semaphorename**.

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

See also **man sem**.

--semaphorename *name*

--id *name*

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

--semaphoretimeout *secs* (not implemented)

If the semaphore is not released within secs seconds, take it anyway.

Implies **--semaphore**.

See also **man sem**.

--seqreplace *replace-str*

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

--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 traceroute

foss.org.my
```

```
debian.org
freenetproject.org
```

--shebang must be set as the first option.

--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 /the/original/parser
--with-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
/the/original/parser --with-options
```

E.g.

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

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

--shellquote

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

--skip-first-line

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

--sshdelay *secs*

Delay starting next ssh by *secs* seconds. GNU **parallel** will pause *secs* seconds after starting each ssh. *secs* can be less than 1 seconds.

-S *[ncpu/]sshlogin[,[ncpu/]sshlogin[,...]]*

--sshlogin *[ncpu/]sshlogin[,[ncpu/]sshlogin[,...]]*

Distribute jobs to remote computers. The jobs will be run on a list of remote computers. GNU **parallel** will determine the number of CPU cores on the remote computers and run the number of jobs as specified by **-j**. If the number *ncpu* is given GNU **parallel** will use this number for number of CPU cores on the host. Normally *ncpu* will not be needed.

An *sshlogin* is of the form:

```
[sshcommand [options]] [username@]hostname
```

The sshlogin must not require a password.

The sshlogin ':' is special, it means 'no ssh' and will therefore run on the local computer.

The sshlogin '..' is special, it read sshlogins from ~/.parallel/sshloginfile

The sshlogin '-' is special, too, it read sshlogins from stdin (standard input).

To specify more sshlogins separate the sshlogins by comma or repeat the options multiple times.

For examples: see **--sshloginfile**.

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

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

--sshlogin is often used with **--transfer**, **--return**, **--cleanup**, and **--trc**.

--sshloginfile *filename*

--slf *filename*

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-core-server.example.com
  2/my_other_username@my-dualcore.example.net
  # This server has SSH running on port 2222
  ssh -p 2222 server.example.net
  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 cores
  //usr/local/bin/myssh -p 2222 -l myusername
hexacpu.example.net
  # Use a different ssh program with 6 cores
  6//usr/local/bin/myssh -p 2222 -l myusername
hexacpu.example.net
  # Assume 16 cores on the local computer
  16/:
```

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.

The sshloginfile '..' is special, it read sshlogins from ~/.parallel/sshloginfile

The sshloginfile '.' is special, it read sshlogins from /etc/parallel/sshloginfile

The sshloginfile '-' is special, too, it read sshlogins from stdin (standard input).

--noswap

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.

--silent

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

--tty

Open terminal tty. If GNU **parallel** is used for starting an interactive program then this option may be needed. It will start 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. When the job is done, the next job will get the 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**.

--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**.

--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.

--timeout *val*

Time out for command. If the command runs for longer than *val* seconds it will get killed with SIGTERM, followed by SIGTERM 200 ms later, followed by SIGKILL 200 ms later.

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

--tollef (obsolete - will be retired 20140222)

Make GNU **parallel** behave more like Tollef's parallel command. It activates **-u**, **-q**, and **--arg-sep --**. It also causes **-l** to change meaning to **--load**.

Not giving '--' is unsupported.

Do not use --tollef unless you know what you are doing.

To override use **--gnu**.

--verbose

-t

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

See also **-v**, **-p**.

--transfer (beta testing)

Transfer files to remote computers. **--transfer** is used with **--sshlogin** when the arguments are files and should be transferred to the remote computers. The files will be transferred using **rsync** and will be put relative to the default work dir. If the path contains /./ the remaining path will be relative to the work dir. E.g.

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

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 /tmp/foo/bar.txt | parallel \
  --sshlogin server.example.com --transfer wc
```

This will transfer the file *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*.

--transfer is often used with **--return** and **--cleanup**.

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

--trc *filename*

Transfer, Return, Cleanup. Short hand for:

--transfer --return *filename* **--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.

--ungroup

-u

Ungroup output. Output is printed as soon as possible and by passes 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:

parallel -j0 'sleep {};echo -n start{};sleep {};echo {}end' ::: 1 2 3 4

parallel -u -j0 'sleep {};echo -n start{};sleep {};echo {}end' ::: 1 2 3 4

It also disables **--tag**. GNU **parallel** runs faster with **-u**. Can be reversed with **--group**.

--extensionreplace *replace-str*

--er *replace-str*

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

--use-cpus-instead-of-cores

Count the number of physical CPUs instead of CPU cores. When computing how many jobs to run simultaneously relative to the number of CPU cores you can ask GNU **parallel** to instead look at the number of physical CPUs. This will make sense for computers that have hyperthreading as two jobs running on one CPU with hyperthreading will run slower than two jobs running on two physical CPUs. Some multi-core CPUs can run faster if only one thread is running per physical CPU. Most users will not need this option.

-v

Verbose. Print the job to be run on stdout (standard output). Can be reversed with **--silent**. See also **-t**.

Use **-v -v** to print the wrapping ssh command when running remotely.

--version

-V

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

--workdir *mydir* (alpha testing)

--wd *mydir* (alpha testing)

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

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

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

To see the difference try:

parallel -S server pwd ::: ""

parallel --wd . -S server pwd ::: ""

parallel --wd ... -S server pwd ::: ""

--wait

Wait for all commands to complete.

Implies **--semaphore**.

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.

--xapply

Read multiple input sources like **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 --xapply echo {1} {2} ::: 1 2 3 ::: a b c
```

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

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

See also **--header**.

EXAMPLE: Working as xargs -n1. Argument appending

GNU **parallel** can work similar to **xargs -n1**.

To compress all html files using **gzip** run:

find . -name '*.html' | parallel gzip

If the file names may contain a newline use **-0**. Substitute FOO BAR with FUBAR in all files in this dir and subdirs:

find . -type f -print0 | parallel -q0 perl -i -pe 's/FOO BAR/FUBAR/g'

Note **-q** is needed because of the space in 'FOO BAR'.

EXAMPLE: Reading arguments from command line

GNU **parallel** can take the arguments from command line instead of stdin (standard input). To compress all html files in the current dir using **gzip** run:

parallel gzip ::: *.html

To convert *.wav to *.mp3 using LAME running one process per CPU core run:

parallel lame {} -o {.}.mp3 ::: *.wav

EXAMPLE: Inserting multiple arguments

When moving a lot of files like this: **mv *.log destdir** you will sometimes get the error:

bash: /bin/mv: Argument list too long

because there are too many files. You can instead do:

ls | grep -E '\.log$' | parallel mv {} destdir

This will run **mv** for each file. It can be done faster if **mv** gets as many arguments that will fit on the line:

ls | grep -E '\.log$' | parallel -m mv {} destdir

EXAMPLE: Context replace

To remove the files *pict0000.jpg* .. *pict9999.jpg* you could do:

seq -w 0 9999 | parallel rm pict{}.jpg

You could also do:

seq -w 0 9999 | perl -pe 's/(.*)/pict$1.jpg/' | parallel -m rm

The first will run **rm** 10000 times, while the last will only run **rm** as many times needed to keep the command line length short enough to avoid **Argument list too long** (it typically runs 1-2 times).

You could also run:

seq -w 0 9999 | parallel -X rm pict{}.jpg

This will also only run **rm** as many times needed to keep the command line length short enough.

EXAMPLE: Compute intensive jobs and substitution

If ImageMagick is installed this will generate a thumbnail of a jpg file:

convert -geometry 120 foo.jpg thumb_foo.jpg

This will run with number-of-cpu-cores jobs in parallel for all jpg files in a directory:

ls *.jpg | parallel convert -geometry 120 {} thumb_{}

To do it recursively use **find**:

find . -name '*.jpg' | parallel convert -geometry 120 {} {}_thumb.jpg

Notice how the argument has to start with **{}** as **{}** will include path (e.g. running **convert -geometry 120 ./foo/bar.jpg thumb_./foo/bar.jpg** would clearly be wrong). The command will generate files like ./foo/bar.jpg_thumb.jpg.

Use **{.}** to avoid the extra .jpg in the file name. This command will make files like ./foo/bar_thumb.jpg:

find . -name '*.jpg' | parallel convert -geometry 120 {} {.}_thumb.jpg

EXAMPLE: Substitution and redirection

This will generate an uncompressed version of .gz-files next to the .gz-file:

parallel zcat {} ">"{.} ::: *.gz

Quoting of > is necessary to postpone the redirection. Another solution is to quote the whole command:

parallel "zcat {} >{.}" ::: *.gz

Other special shell characters (such as * ; $ > < | >> <<) also need to be put in quotes, as they may otherwise be interpreted by the shell and not given to GNU **parallel**.

EXAMPLE: Composed commands

A job can consist of several commands. This will print the number of files in each directory:

ls | parallel 'echo -n {}" "; ls {}|wc -l'

To put the output in a file called <name>.dir:

ls | parallel '(echo -n {}" "; ls {}|wc -l) > **{}.dir'**

Even small shell scripts can be run by GNU **parallel**:

find . | parallel 'a={}; name=${a##*/}; upper=$(echo "$name" | tr "[:lower:]" "[:upper:]"); echo "$name - $upper"'

ls | parallel 'mv {} "$(echo {} | tr "[:upper:]" "[:lower:]")"'

Given a list of URLs, list all URLs that fail to download. Print the line number and the URL.

cat urlfile | parallel "wget {} 2>/dev/null || grep -n {} urlfile"

Create a mirror directory with the same filenames except all files and symlinks are empty files.

cp -rs /the/source/dir mirror_dir; find mirror_dir -type l | parallel -m rm {} '&&' touch {}

Find the files in a list that do not exist

cat file_list | parallel 'if [! -e {}] ; then echo {}; fi'

EXAMPLE: Calling Bash functions

If the composed command is longer than a line, it becomes hard to read. In Bash you can use functions. Just remember to **export -f** the function.

```
doit() {
  echo Doing it for $1
```

```
  sleep 2
  echo Done with $1
}
export -f doit
parallel doit ::: 1 2 3

doubleit() {
  echo Doing it for $1 $2
  sleep 2
  echo Done with $1 $2
}
export -f doubleit
parallel doubleit ::: 1 2 3 ::: a b
```

To do this on remote servers you need to transfer the function using **--env**:

```
parallel --env doit -S server doit ::: 1 2 3
parallel --env doubleit -S server doubleit ::: 1 2 3 ::: a b
```

EXAMPLE: Removing file extension when processing files

When processing files removing the file extension using **{.}** is often useful.

Create a directory for each zip-file and unzip it in that dir:

parallel 'mkdir {.}; cd {.}; unzip ../{}' ::: *.zip

Recompress all .gz files in current directory using **bzip2** running 1 job per CPU core in parallel:

parallel "zcat {} | bzip2 >{.}.bz2 && rm {}" ::: *.gz

Convert all WAV files to MP3 using LAME:

find sounddir -type f -name '*.wav' | parallel lame {} -o {.}.mp3

Put all converted in the same directory:

find sounddir -type f -name '*.wav' | parallel lame {} -o mydir/{/.}.mp3

EXAMPLE: Removing two file extensions when processing files and calling GNU Parallel from itself

If you have directory with tar.gz files and want these extracted in the corresponding dir (e.g foo.tar.gz will be extracted in the dir foo) you can do:

ls *.tar.gz| parallel --er {tar} 'echo {tar}|parallel "mkdir -p {.} ; tar -C {.} -xf {.}.tar.gz"'

EXAMPLE: Download 10 images for each of the past 30 days

Let us assume a website stores images like:

```
http://www.example.com/path/to/YYYYMMDD_##.jpg
```

where YYYYMMDD is the date and ## is the number 01-10. This will download images for the past 30 days:

parallel wget http://www.example.com/path/to/'$(date -d "today -{1} days" +%Y%m%d)_{2}.jpg' ::: $(seq 30) ::: $(seq -w 10)

$(date -d "today -{1} days" +%Y%m%d) will give the dates in YYYYMMDD with {1} days subtracted.

EXAMPLE: Breadth first parallel web crawler/mirrorer

This script below will crawl and mirror a URL in parallel. It downloads first pages that are 1 click down, then 2 clicks down, then 3; instead of the normal depth first, where the first link link on each page is fetched first.

Run like this:

PARALLEL=-j100 ./parallel-crawl http://gatt.org.yeslab.org/

Remove the **wget** part if you only want a web crawler.

It works by fetching a page from a list of URLs and looking for links in that page that are within the same starting URL and that have not already been seen. These links are added to a new queue. When all the pages from the list is done, the new queue is moved to the list of URLs and the process is started over until no unseen links are found.

```
  #!/bin/bash

  # E.g. http://gatt.org.yeslab.org/
  URL=$1
  # Stay inside the start dir
  BASEURL=$(echo $URL | perl -pe 's:#.*::; s:(//.*/)[^/]*:$1:')
  URLLIST=$(mktemp urllist.XXXX)
  URLLIST2=$(mktemp urllist.XXXX)
  SEEN=$(mktemp seen.XXXX)

  # Spider to get the URLs
  echo $URL >$URLLIST
  cp $URLLIST $SEEN

  while [ -s $URLLIST ] ; do
    cat $URLLIST |
      parallel lynx -listonly -image_links -dump {} \; wget -qm -l1 -Q1 {}
\; echo Spidered: {} \>\&2 |
      perl -ne 's/#.*//; s/\s+\d+.\s(\S+)$/$1/ and do { $seen{$1}++ or
print }' |
      grep -F $BASEURL |
      grep -v -x -F -f $SEEN | tee -a $SEEN > $URLLIST2
    mv $URLLIST2 $URLLIST
  done

  rm -f $URLLIST $URLLIST2 $SEEN
```

EXAMPLE: Process files from a tar file while unpacking

If the files to be processed are in a tar file then unpacking one file and processing it immediately may be faster than first unpacking all files.

tar xvf foo.tgz | perl -ne 'print $l;$l=$_;END{print $l}' | parallel echo

The Perl one-liner is needed to avoid race condition.

EXAMPLE: Rewriting a for-loop and a while-read-loop

for-loops like this:

```
  (for x in `cat list` ; do
    do_something $x
  done) | process_output
```

and while-read-loops like this:

```
cat list | (while read x ; do
  do_something $x
done) | process_output
```

can be written like this:

cat list | parallel do_something | process_output

For example: Find which host name in a list has IP address 1.2.3 4:

cat hosts.txt | parallel -P 100 host | grep 1.2.3.4

If the processing requires more steps the for-loop like this:

```
(for x in `cat list` ; do
  no_extension=${x%.*};
  do_something $x scale $no_extension.jpg
  do_step2 <$x $no_extension
done) | process_output
```

and while-loops like this:

```
cat list | (while read x ; do
  no_extension=${x%.*};
  do_something $x scale $no_extension.jpg
  do_step2 <$x $no_extension
done) | process_output
```

can be written like this:

cat list | parallel "do_something {} scale {.}.jpg ; do_step2 <{} {.}" | process_output

EXAMPLE: Rewriting nested for-loops

Nested for-loops like this:

```
(for x in `cat xlist` ; do
  for y in `cat ylist` ; do
    do_something $x $y
  done
done) | process_output
```

can be written like this:

parallel do_something {1} {2} :::: xlist ylist | process_output

Nested for-loops like this:

```
(for gender in M F ; do
  for size in S M L XL XXL ; do
    echo $gender $size
  done
done) | sort
```

can be written like this:

parallel echo {1} {2} ::: M F ::: S M L XL XXL | sort

EXAMPLE: Finding the lowest difference between files

diff is good for finding differences in text files. **diff | wc -l** gives an indication of the size of the difference. To find the differences between all files in the current dir do:

parallel --tag 'diff {1} {2} | wc -l' ::: * ::: * | sort -nk3

This way it is possible to see if some files are closer to other files.

EXAMPLE: for-loops with column names

When doing multiple nested for-loops it can be easier to keep track of the loop variable if is is named instead of just having a number. Use **--header :** to let the first argument be an named alias for the positional replacement string:

```
  parallel --header : echo {gender} {size} ::: gender M F ::: size S M L XL
 XXL
```

This also works if the input file is a file with columns:

```
  cat addressbook.tsv | parallel --colsep '\t' --header : echo {Name}
{E-mail address}
```

EXAMPLE: Count the differences between all files in a dir

Using **--results** the results are saved in /tmp/diffcount*.

```
  parallel --results /tmp/diffcount "diff -U 0 {1} {2} |tail -n +3 |grep -v
 '^@'|wc -l" ::: * ::: *
```

To see the difference between file A and file B look at the file '/tmp/diffcount 1 A 2 B' where spaces are TABs (\t).

EXAMPLE: Speeding up fast jobs

Starting a job on the local machine takes around 3 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. Compare the speed of these:

```
    seq -w 0 9999 | parallel touch pict{}.jpg

    seq -w 0 9999 | parallel -X touch pict{}.jpg
```

If your program cannot take multiple arguments, then you can use GNU **parallel** to spawn multiple GNU **parallel**s:

```
    seq -w 0 999999 | parallel -j10 --pipe parallel -j0 touch pict{}.jpg
```

If **-j0** normally spawns 506 jobs, then the above will try to spawn 5060 jobs. It is likely that you this way will hit the limit of number of processes and/or filehandles. Look at 'ulimit -n' and 'ulimit -u' to raise these limits.

EXAMPLE: Using shell variables

When using shell variables you need to quote them correctly as they may otherwise be split on spaces.

Notice the difference between:

```
V=("My brother's 12\" records are worth <\$\$\$>"'!' Foo Bar)
parallel echo ::: ${V[@]} # This is probably not what you want
```

and:

```
V=("My brother's 12\" records are worth <\$\$\$>"'!' Foo Bar)
parallel echo ::: "${V[@]}"
```

When using variables in the actual command that contains special characters (e.g. space) you can quote them using **'"$VAR"'** or using "'s and **-q**:

```
V="Here  are  two "
parallel echo "'$V'" ::: spaces
parallel -q echo "$V" ::: spaces
```

EXAMPLE: Group output lines

When running jobs that output data, you often do not want the output of multiple jobs to run together. GNU **parallel** defaults to grouping the output of each job, so the output is printed when the job finishes. If you want the output to be printed while the job is running you can use **-u**.

Compare the output of:

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

to the output of:

parallel -u traceroute ::: foss.org.my debian.org freenetproject.org

EXAMPLE: Tag output lines

GNU **parallel** groups the output lines, but it can be hard to see where the different jobs begin. **--tag** prepends the argument to make that more visible:

parallel --tag traceroute ::: foss.org.my debian.org freenetproject.org

Check the uptime of the servers in *~/.parallel/sshloginfile*:

parallel --tag -S .. --nonall uptime

EXAMPLE: Keep order of output same as order of input

Normally the output of a job will be printed as soon as it completes. Sometimes you want the order of the output to remain the same as the order of the input. This is often important, if the output is used as input for another system. **-k** will make sure the order of output will be in the same order as input even if later jobs end before earlier jobs.

Append a string to every line in a text file:

cat textfile | parallel -k echo {} append_string

If you remove **-k** some of the lines may come out in the wrong order.

Another example is **traceroute**:

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

will give traceroute of foss.org.my, debian.org and freenetproject.org, but it will be sorted according to which job completed first.

To keep the order the same as input run:

parallel -k traceroute ::: foss.org.my debian.org freenetproject.org

This will make sure the traceroute to foss.org.my will be printed first.

A bit more complex example is downloading a huge file in chunks in parallel: Some internet connections will deliver more data if you download files in parallel. For downloading files in parallel

see: "EXAMPLE: Download 10 images for each of the past 30 days". But if you are downloading a big file you can download the file in chunks in parallel.

To download byte 10000000-19999999 you can use **curl**:

curl -r 10000000-19999999 http://example.com/the/big/file > **file.part**

To download a 1 GB file we need 100 10MB chunks downloaded and combined in the correct order.

seq 0 99 | parallel -k curl -r \ {}0000000-{}9999999 http://example.com/the/big/file > **file**

EXAMPLE: Parallel grep

grep -r greps recursively through directories. On multicore CPUs GNU **parallel** can often speed this up.

find . -type f | parallel -k -j150% -n 1000 -m grep -H -n STRING {}

This will run 1.5 job per core, and give 1000 arguments to **grep**.

EXAMPLE: Using remote computers

To run commands on a remote computer SSH needs to be set up and you must be able to login without entering a password (The commands **ssh-copy-id** and **ssh-agent** may help you do that).

If you need to login to a whole cluster, you typically do not want to accept the host key for every host. You want to accept them the first time and be warned if they are ever changed. To do that:

```
# Add the servers to the sshloginfile
(echo servera; echo serverb) > .parallel/my_cluster
# Make sure .ssh/config exist
touch .ssh/config
cp .ssh/config .ssh/config.backup
# Disable StrictHostKeyChecking temporarily
(echo 'Host *'; echo StrictHostKeyChecking no) >> .ssh/config
parallel --slf my_cluster --nonall true
# Remove the disabling of StrictHostKeyChecking
mv .ssh/config.backup .ssh/config
```

The servers in **.parallel/my_cluster** are now added in **.ssh/known_hosts**.

To run **echo** on **server.example.com**:

```
seq 10 | parallel --sshlogin server.example.com echo
```

To run commands on more than one remote computer run:

```
seq 10 | parallel --sshlogin server.example.com,server2.example.net echo
```

Or:

```
seq 10 | parallel --sshlogin server.example.com \
  --sshlogin server2.example.net echo
```

If the login username is *foo* on *server2.example.net* use:

```
seq 10 | parallel --sshlogin server.example.com \
  --sshlogin foo@server2.example.net echo
```

To distribute the commands to a list of computers, make a file *mycomputers* with all the computers:

```
server.example.com
```

```
foo@server2.example.com
server3.example.com
```

Then run:

```
seq 10 | parallel --sshloginfile mycomputers echo
```

To include the local computer add the special sshlogin ':' to the list:

```
server.example.com
foo@server2.example.com
server3.example.com
:
```

GNU **parallel** will try to determine the number of CPU cores on each of the remote computers, and run one job per CPU core - even if the remote computers do not have the same number of CPU cores.

If the number of CPU cores on the remote computers is not identified correctly the number of CPU cores can be added in front. Here the computer has 8 CPU cores.

```
seq 10 | parallel --sshlogin 8/server.example.com echo
```

EXAMPLE: Transferring of files

To recompress gzipped files with **bzip2** using a remote computer run:

```
find logs/ -name '*.gz' | \
  parallel --sshlogin server.example.com \
  --transfer "zcat {} | bzip2 -9 >{.}.bz2"
```

This will list the .gz-files in the *logs* directory and all directories below. Then it will transfer the files to *server.example.com* to the corresponding directory in *$HOME/logs*. On *server.example.com* the file will be recompressed using **zcat** and **bzip2** resulting in the corresponding file with *.gz* replaced with *.bz2*.

If you want the resulting bz2-file to be transferred back to the local computer add *--return {.}.bz2*:

```
find logs/ -name '*.gz' | \
  parallel --sshlogin server.example.com \
  --transfer --return {.}.bz2 "zcat {} | bzip2 -9 >{.}.bz2"
```

After the recompressing is done the *.bz2*-file is transferred back to the local computer and put next to the original *.gz*-file.

If you want to delete the transferred files on the remote computer add *--cleanup*. This will remove both the file transferred to the remote computer and the files transferred from the remote computer:

```
find logs/ -name '*.gz' | \
  parallel --sshlogin server.example.com \
  --transfer --return {.}.bz2 --cleanup "zcat {} | bzip2 -9 >{.}.bz2"
```

If you want run on several computers add the computers to *--sshlogin* either using ',' or multiple *--sshlogin*:

```
find logs/ -name '*.gz' | \
  parallel --sshlogin server.example.com,server2.example.com \
  --sshlogin server3.example.com \
  --transfer --return {.}.bz2 --cleanup "zcat {} | bzip2 -9 >{.}.bz2"
```

You can add the local computer using *--sshlogin :*. This will disable the removing and transferring for the local computer only:

```
find logs/ -name '*.gz' | \
  parallel --sshlogin server.example.com,server2.example.com \
  --sshlogin server3.example.com \
  --sshlogin : \
  --transfer --return {.}.bz2 --cleanup "zcat {} | bzip2 -9 >{.}.bz2"
```

Often *--transfer*, *--return* and *--cleanup* are used together. They can be shortened to *--trc*:

```
find logs/ -name '*.gz' | \
  parallel --sshlogin server.example.com,server2.example.com \
  --sshlogin server3.example.com \
  --sshlogin : \
  --trc {.}.bz2 "zcat {} | bzip2 -9 >{.}.bz2"
```

With the file *mycomputers* containing the list of computers it becomes:

```
find logs/ -name '*.gz' | parallel --sshloginfile mycomputers \
  --trc {.}.bz2 "zcat {} | bzip2 -9 >{.}.bz2"
```

If the file *~/.parallel/sshloginfile* contains the list of computers the special short hand *-S ..* can be used:

```
find logs/ -name '*.gz' | parallel -S .. \
  --trc {.}.bz2 "zcat {} | bzip2 -9 >{.}.bz2"
```

EXAMPLE: Distributing work to local and remote computers

Convert *.mp3 to *.ogg running one process per CPU core on local computer and server2:

```
parallel --trc {.}.ogg -S server2,: \
'mpg321 -w - {} | oggenc -q0 - -o {.}.ogg' ::: *.mp3
```

EXAMPLE: Running the same command on remote computers

To run the command **uptime** on remote computers you can do:

parallel --tag --nonall -S server1,server2 uptime

--nonall reads no arguments. If you have a list of jobs you want run on each computer you can do:

parallel --tag --onall -S server1,server2 echo ::: 1 2 3

Remove **--tag** if you do not want the sshlogin added before the output.

If you have a lot of hosts use '-j0' to access more hosts in parallel.

EXAMPLE: Parallelizing rsync

rsync is a great tool, but sometimes it will not fill up the available bandwidth. This is often a problem when copying several big files over high speed connections.

The following will start one **rsync** per big file in *src-dir* to *dest-dir* on the server *fooserver*:

cd src-dir; find . -type f -size +100000 | parallel -v ssh fooserver mkdir -p /dest-dir/{//}\;rsync -Havessh {} fooserver:/dest-dir/{}

The dirs created may end up with wrong permissions and smaller files are not being transferred. To fix those run **rsync** a final time:

rsync -Havessh src-dir/ fooserver:/dest-dir/

If you are unable to push data, but need to pull them and the files are called digits.png (e.g. 000000.png) you might be able to do:

seq -w 0 99 | parallel rsync -Havessh fooserver:src-path/*{}.png destdir/

EXAMPLE: Use multiple inputs in one command

Copy files like foo.es.ext to foo.ext:

ls *.es.* | perl -pe 'print; s/\.es//' | parallel -N2 cp {1} {2}

The perl command spits out 2 lines for each input. GNU **parallel** takes 2 inputs (using **-N2**) and replaces {1} and {2} with the inputs.

Count in binary:

parallel -k echo ::: 0 1 ::: 0 1 ::: 0 1 ::: 0 1 ::: 0 1 ::: 0 1

Print the number on the opposing sides of a six sided die:

parallel --xapply -a <(seq 6) -a <(seq 6 -1 1) echo

parallel --xapply echo :::: <(seq 6) <(seq 6 -1 1)

Convert files from all subdirs to PNG-files with consecutive numbers (useful for making input PNG's for **ffmpeg**):

parallel --xapply -a <(find . -type f | sort) -a <(seq $(find . -type f|wc -l)) convert {1} {2}.png

Alternative version:

find . -type f | sort | parallel convert {} {#}.png

EXAMPLE: Use a table as input

Content of table_file.tsv:

```
foo<TAB>bar
baz <TAB> quux
```

To run:

```
cmd -o bar -i foo
cmd -o quux -i baz
```

you can run:

parallel -a table_file.tsv --colsep '\t' cmd -o {2} -i {1}

Note: The default for GNU **parallel** is to remove the spaces around the columns. To keep the spaces:

parallel -a table_file.tsv --trim n --colsep '\t' cmd -o {2} -i {1}

EXAMPLE: Run the same command 10 times

If you want to run the same command with the same arguments 10 times in parallel you can do:

seq 10 | parallel -n0 my_command my_args

EXAMPLE: Working as cat | sh. Resource inexpensive jobs and evaluation

GNU **parallel** can work similar to **cat | sh**.

A resource inexpensive job is a job that takes very little CPU, disk I/O and network I/O. Ping is an example of a resource inexpensive job. wget is too - if the webpages are small.

The content of the file jobs_to_run:

```
ping -c 1 10.0.0.1
wget http://example.com/status.cgi?ip=10.0.0.1
ping -c 1 10.0.0.2
wget http://example.com/status.cgi?ip=10.0.0.2
...
ping -c 1 10.0.0.255
wget http://example.com/status.cgi?ip=10.0.0.255
```

To run 100 processes simultaneously do:

parallel -j 100 < jobs_to_run

As there is not a *command* the jobs will be evaluated by the shell.

EXAMPLE: Processing a big file using more cores

To process a big file or some output you can use **--pipe** to split up the data into blocks and pipe the blocks into the processing program.

If the program is **gzip -9** you can do:

cat bigfile | parallel --pipe --recend '' -k gzip -9 >**bigfile.gz**

This will split **bigfile** into blocks of 1 MB and pass that to **gzip -9** in parallel. One **gzip** will be run per CPU core. The output of **gzip -9** will be kept in order and saved to **bigfile.gz**

gzip works fine if the output is appended, but some processing does not work like that - for example sorting. For this GNU **parallel** can put the output of each command into a file. This will sort a big file in parallel:

cat bigfile | parallel --pipe --files sort | parallel -Xj1 sort -m {} ';' rm {} >**bigfile.sort**

Here **bigfile** is split into blocks of around 1MB, each block ending in '\n' (which is the default for **--recend**). Each block is passed to **sort** and the output from **sort** is saved into files. These files are passed to the second **parallel** that runs **sort -m** on the files before it removes the files. The output is saved to **bigfile.sort**.

EXAMPLE: Running more than 500 jobs workaround

If you need to run a massive amount of jobs in parallel, then you will likely hit the filehandle limit which is often around 500 jobs. If you are super user you can raise the limit in /etc/security/limits.conf but you can also use this workaround. The filehandle limit is per process. That means that if you just spawn more GNU **parallel**s then each of them can run 500 jobs. This will spawn up to 2500 jobs:

cat myinput | parallel --pipe -N 50 --round-robin -j50 parallel -j50 your_prg

This will spawn up to 250000 jobs (use with caution - you need 250 GB RAM to do this):

cat myinput | parallel --pipe -N 500 --round-robin -j500 parallel -j500 your_prg

EXAMPLE: Working as mutex and counting semaphore

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

A counting semaphore will allow a given number of jobs to be started in the background. When the number of jobs are running in the background, GNU **sem** will wait for one of these to complete before starting another command. **sem --wait** will wait for all jobs to complete.

Run 10 jobs concurrently in the background:

```
for i in *.log ; do
  echo $i
  sem -j10 gzip $i ";" echo done
done
```

```
sem --wait
```

A mutex is a counting semaphore allowing only one job to run. This will edit the file *myfile* and prepends the file with lines with the numbers 1 to 3.

```
seq 3 | parallel sem sed -i -e 'i{}' myfile
```

As *myfile* can be very big it is important only one process edits the file at the same time.

Name the semaphore to have multiple different semaphores active at the same time:

```
seq 3 | parallel sem --id mymutex sed -i -e 'i{}' myfile
```

EXAMPLE: Start editor with filenames from stdin (standard input)

You can use GNU **parallel** to start interactive programs like emacs or vi:

cat filelist | parallel --tty -X emacs

cat filelist | parallel --tty -X vi

If there are more files than will fit on a single command line, the editor will be started again with the remaining files.

EXAMPLE: Running sudo

sudo requires a password to run a command as root. It caches the access, so you only need to enter the password again if you have not used **sudo** for a while.

The command:

```
parallel sudo echo ::: This is a bad idea
```

is no good, as you would be prompted for the sudo password for each of the jobs. You can either do:

```
sudo echo This
parallel sudo echo ::: is a good idea
```

or:

```
sudo parallel echo ::: This is a good idea
```

This way you only have to enter the sudo password once.

EXAMPLE: GNU Parallel as queue system/batch manager

GNU **parallel** can work as a simple job queue system or batch manager. The idea is to put the jobs into a file and have GNU **parallel** read from that continuously. As GNU **parallel** will stop at end of file we use **tail** to continue reading:

true >**jobqueue**; **tail -f jobqueue | parallel**

To submit your jobs to the queue:

echo my_command my_arg >> **jobqueue**

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

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

There is a a small issue when using GNU **parallel** as queue system/batch manager: You have to submit JobSlot number of jobs before they will start, and after that you can submit one at a time, and job will start immediately if free slots are available. Output from the running or completed jobs are held

back and will only be printed when JobSlots more jobs has been started (unless you use --ungroup or -u, in which case the output from the jobs are printed immediately). E.g. if you have 10 jobslots then the output from the first completed job will only be printed when job 11 has started, and the output of second completed job will only be printed when job 12 has started.

EXAMPLE: GNU Parallel as dir processor

If you have a dir in which users drop files that needs to be processed you can do this on GNU/Linux (If you know what **inotifywait** is called on other platforms file a bug report):

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

This will run the command **echo** on each file put into **my_dir** or subdirs of **my_dir**.

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

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

If the files to be processed are in a tar file then unpacking one file and processing it immediately may be faster than first unpacking all files. Set up the dir processor as above and unpack into the dir.

Using GNU Parallel as dir processor has the same limitations as using GNU Parallel as queue system/batch manager.

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 this:

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
```

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: To avoid dealing with the quoting problems it may be easier just to write 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 **SIGTERM** to GNU **parallel**:

killall -TERM 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_PID

The environment variable $PARALLEL_PID is set by GNU **parallel** and is visible to the jobs started from GNU **parallel**. This makes it possible for the jobs to communicate directly to GNU **parallel**. Remember to quote the $, so it gets evaluated by the correct shell.

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 -TERM $PARALLEL_PID**. This only works on the local computer.

$PARALLEL_SEQ

$PARALLEL_SEQ will be set to the sequence number of the job running. Remember to quote the $, so it gets evaluated by the correct shell.

Example:

seq 10 | parallel -N2 echo seq:'$'PARALLEL_SEQ arg1:{1} arg2:{2}

$TMPDIR

Directory for temporary files. See: **--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

can be written as:

cat list | PARALLEL="-kvj1" parallel ls

cat list | parallel -j1 -k -v -S"myssh user@server" ls

can be written as:

cat list | PARALLEL='-kvj1 -S myssh\ user@server' parallel echo

Notice the \ in the middle is needed because 'myssh' and 'user@server' must be one argument.

DEFAULT PROFILE (CONFIG FILE)

The file ~/.parallel/config (formerly known as .parallelrc) will be read if it exists. Lines starting with '#' will be ignored. It can be formatted like the environment variable $PARALLEL, but it is often easier to simply put each option on its own line.

Options on the command line takes precedence over the environment variable $PARALLEL which takes precedence over the file ~/.parallel/config.

PROFILE FILES

If **--profile** set, GNU **parallel** will read the profile from that file instead of ~/.parallel/config. You can have multiple **--profiles**.

Example: Profile for running a command on every sshlogin in ~/.ssh/sshlogins and prepend the output with the sshlogin:

```
echo --tag -S .. --nonall > ~/.parallel/n
parallel -Jn 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

If **--halt-on-error** 0 or not specified:

0 All jobs ran without error.

1-253

Some of the jobs failed. The exit status gives the number of failed jobs

254 More than 253 jobs failed.

255 Other error.

If **--halt-on-error** 1 or 2: Exit status of the failing job.

DIFFERENCES BETWEEN GNU Parallel AND ALTERNATIVES

There are a lot programs with some of the functionality of GNU **parallel**. GNU **parallel** strives to include the best of the functionality without sacrificing ease of use.

SUMMARY TABLE

The following features are in some of the comparable tools:

Inputs I1. Arguments can be read from stdin I2. Arguments can be read from a file I3. Arguments can be read from multiple files I4. Arguments can be read from command line I5. Arguments can be read from a table I6. Arguments can be read from the same file using #! (shebang) I7. Line oriented input as default (Quoting of special chars not needed)

Manipulation of input M1. Composed command M2. Multiple arguments can fill up an execution line M3. Arguments can be put anywhere in the execution line M4. Multiple arguments can be put anywhere in the execution line M5. Arguments can be replaced with context M6. Input can be treated as complete execution line

Outputs O1. Grouping output so output from different jobs do not mix O2. Send stderr (standard error) to stderr (standard error) O3. Send stdout (standard output) to stdout (standard output) O4. Order of output can be same as order of input O5. Stdout only contains stdout (standard output) from the command O6. Stderr only contains stderr (standard error) from the command

Execution E1. Running jobs in parallel E2. List running jobs E3. Finish running jobs, but do not start new jobs E4. Number of running jobs can depend on number of cpus E5. Finish running jobs, but do not start new jobs after first failure E6. Number of running jobs can be adjusted while running

Remote execution R1. Jobs can be run on remote computers R2. Basefiles can be transferred R3. Argument files can be transferred R4. Result files can be transferred R5. Cleanup of transferred files R6. No config files needed R7. Do not run more than SSHD's MaxStartup can handle R8. Configurable SSH command R9. Retry if connection breaks occasionally

Semaphore S1. Possibility to work as a mutex S2. Possibility to work as a counting semaphore

Legend - = no x = not applicable ID = yes

As every new version of the programs are not tested the table may be outdated. Please file a bug-report if you find errors (See REPORTING BUGS).

parallel: I1 I2 I3 I4 I5 I6 I7 M1 M2 M3 M4 M5 M6 O1 O2 O3 O4 O5 O6 E1 E2 E3 E4 E5 E6 R1 R2 R3 R4 R5 R6 R7 R8 R9 S1 S2

xargs: I1 I2 - - - - - - M2 M3 - - - - O2 O3 - O5 O6 E1 - - - - - - - - - - x - - - - -

find -exec: - - - x - x - - M2 M3 - - - - - O2 O3 O4 O5 O6 - - - - - - - - - - - - - - - - x x

make -j: - - - - - - - - - - - - - O1 O2 O3 - x O6 E1 - - - E5 - - - - - - - - - - - -

ppss: I1 I2 - - - - I7 M1 - M3 - - M6 O1 - - x - - E1 E2 ?E3 E4 - - R1 R2 R3 R4 - - ?R7 ? ? - -

pexec: I1 I2 - I4 I5 - - M1 - M3 - - M6 O1 O2 O3 - O5 O6 E1 - - E4 - E6 R1 - - - - R6 - - - S1 -

xjobs: TODO - Please file a bug-report if you know what features xjobs supports (See REPORTING BUGS).

prll: TODO - Please file a bug-report if you know what features prll supports (See REPORTING BUGS).

dxargs: TODO - Please file a bug-report if you know what features dxargs supports (See REPORTING BUGS).

mdm/middelman: TODO - Please file a bug-report if you know what features mdm/middelman supports (See REPORTING BUGS).

xapply: TODO - Please file a bug-report if you know what features xapply supports (See REPORTING BUGS).

paexec: TODO - Please file a bug-report if you know what features paexec supports (See REPORTING BUGS).

ClusterSSH: TODO - Please file a bug-report if you know what features ClusterSSH supports (See REPORTING BUGS).

DIFFERENCES BETWEEN xargs AND GNU Parallel

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

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

```
touch important_file
touch 'not important_file'
ls not* | xargs rm
mkdir -p "My brother's 12\" records"
ls | xargs rmdir
```

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

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

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

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

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

```
  parallel perl -e '\$a=\"1{}\"x10000000\;print\ \$a,\"\\n\"' '>' {} ::: a
b c d e f
  ls -l a b c d e f
  parallel -kP4 -n1 grep 1 > out.par ::: a b c d e f
  echo a b c d e f | xargs -P4 -n1 grep 1 > out.xargs-unbuf
  echo a b c d e f | xargs -P4 -n1 grep --line-buffered 1 >
out.xargs-linebuf
  echo a b c d e f | xargs -n1 grep --line-buffered 1 > out.xargs-serial
  ls -l out*
  md5sum out*
```

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

xargs has no support for running jobs on remote computers.

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

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

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

ls | parallel "wc {} > {}.wc"

becomes (assuming you have 8 cores)

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

and

ls | parallel "echo {}; ls {}|wc"

becomes (assuming you have 8 cores)

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

DIFFERENCES BETWEEN find -exec AND GNU Parallel

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

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

DIFFERENCES BETWEEN make -j AND GNU Parallel

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

make -j has no support for grouping the output, therefore output may run together, e.g. the first half of a line is from one process and the last half of the line is from another process. The example **Parallel grep** cannot be done reliably with **make -j** because of this.

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

DIFFERENCES BETWEEN ppss AND GNU Parallel

ppss is also a tool for running jobs in parallel.

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

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

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

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

EXAMPLES FROM ppss MANUAL

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

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

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

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

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

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

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

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

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

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

5 # parallel does not use configs. If you want a different username put it in nodes.txt: user@hostname

5 find source/dir -type f | parallel --sshloginfile nodes.txt --trc {.}.mp3 lame -a {} -o {.}.mp3 --preset standard --quiet

6 ./ppss stop -C config.cfg

6 killall -TERM parallel

7 ./ppss pause -C config.cfg

7 Press: CTRL-Z or killall -SIGTSTP parallel

8 ./ppss continue -C config.cfg

8 Enter: fg or killall -SIGCONT parallel

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

9 killall -SIGUSR2 parallel

DIFFERENCES BETWEEN pexec AND GNU Parallel

pexec is also a tool for running jobs in parallel.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8 Combining GNU **parallel** and GNU **sem**.

8 ls *jpg | parallel -j8 'sem --id blockread cat {} | jpegtopnm |' \ 'pnmscale 0.5 | pnmtojpeg | sem --id blockwrite cat > th_{}'

8 If reading and writing is done to the same disk, this may be faster as only one process will be either reading or writing:

8 ls *jpg | parallel -j8 'sem --id diskio cat {} | jpegtopnm |' \ 'pnmscale 0.5 | pnmtojpeg | sem --id diskio cat > th_{}'

DIFFERENCES BETWEEN xjobs AND GNU Parallel

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

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

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

1 ls -1 *.zip | xjobs unzip

1 ls *.zip | parallel unzip

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

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

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

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

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

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

5 xjobs -s script

5 cat script | parallel

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

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

DIFFERENCES BETWEEN prll AND GNU Parallel

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

prll encourages using BASH aliases and BASH functions instead of scripts. GNU **parallel** will never support running aliases (see why http://www.perlmonks.org/index.pl?node_id=484296). However, scripts, composed commands, or functions exported with **export -f** work just fine.

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

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

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

parallel mogrify -flip ::: *.jpg

DIFFERENCES BETWEEN dxargs AND GNU Parallel

dxargs is also a tool for running jobs in parallel.

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

DIFFERENCES BETWEEN mdm/middleman AND GNU Parallel

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

Here are the shellscripts of http://mdm.berlios.de/usage.html ported to GNU **parallel**:

seq 19 | parallel buffon -o - | sort -n > **result**

cat files | parallel cmd

find dir -execdir sem cmd {} \;

DIFFERENCES BETWEEN xapply AND GNU Parallel

xapply can run jobs in parallel on the local computer.

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

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

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

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

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

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

3 parallel --xapply diff {1} {2} :::: manifest1 checklist1

4 xapply 'indent' *.c

4 parallel indent ::: *.c

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

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

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

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

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

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

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

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

8 xapply -fn "" /etc/passwd

8 parallel -k echo < /etc/passwd

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

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

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

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

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

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

DIFFERENCES BETWEEN paexec AND GNU Parallel

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

paexec requires commands to print a blank line as the last output. This means you will have to write a wrapper for most programs.

paexec has a job dependency facility so a job can depend on another job to be executed successfully. Sort of a poor-man's **make**.

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

1_div_X_run:

```
../../paexec -s -l -c "`pwd`/1_div_X_cmd" -n +1 <<EOF [...]
parallel echo {} '|' `pwd`/1_div_X_cmd <<EOF [...]
```

all_substr_run:

```
../../paexec -lp -c "`pwd`/all_substr_cmd" -n +3 <<EOF [...]
parallel echo {} '|' `pwd`/all_substr_cmd <<EOF [...]
```

cc_wrapper_run:

```
../../paexec -c "env CC=gcc CFLAGS=-O2 `pwd`/cc_wrapper_cmd" \
           -n 'host1 host2' \
           -t '/usr/bin/ssh -x' <<EOF [...]
parallel echo {} '|' "env CC=gcc CFLAGS=-O2 `pwd`/cc_wrapper_cmd" \
           -S host1,host2 <<EOF [...]
# This is not exactly the same, but avoids the wrapper
parallel gcc -O2 -c -o {.}.o {} \
           -S host1,host2 <<EOF [...]
```

toupper_run:

```
../../paexec -lp -c "`pwd`/toupper_cmd" -n +10 <<EOF [...]
parallel echo {} '|' ./toupper_cmd <<EOF [...]
# Without the wrapper:
parallel echo {} '| awk {print\ toupper\(\$0\)}' <<EOF [...]
```

DIFFERENCES BETWEEN ClusterSSH AND GNU Parallel

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

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

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

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

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

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. Half of the startup time is spent finding the maximal length of a command line. Setting **-s** will remove this part of the startup time.

Job startup

Starting a job on the local machine takes around 3 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.

Using **--ungroup** the 3 ms can be lowered to around 2 ms.

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 MaxStartup). 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 remove 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

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. An alias will, however, not work (see

why http://www.perlmonks.org/index.pl?node_id=484296), so change your alias to a script.

REPORTING BUGS

Report bugs to <bug-parallel@gnu.org> or
https://savannah.gnu.org/bugs/?func=additem&group=parallel

Your bug report should always include:

- The error message you get (if any).
- The complete output of **parallel --version**. If you are not running the latest released version you should specify why you believe the problem is not fixed in that version.
- A complete example that others can run that shows the problem. This should preferably be small and simple. A combination of **yes**, **seq**, **cat**, **echo**, and **sleep** can reproduce most errors. If your example requires large files, see if you can make them by something like **seq 1000000** > **file** or **yes | head -n 10000000** > **file**. If your example requires remote execution, see if you can use **localhost** - maybe using another login.
- The output of your example. 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 (http://www.youtube.com/playlist?list=PL284C9FF2488BC6D1), walked through the tutorial (man parallel_tutorial), and read the EXAMPLE section in the man page (man parallel - search for EXAMPLE:).

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:
http://sourceforge.net/projects/virtualboximage/files/

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, you should assume the debugging will be done through you. That will put 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 less work for you if you can reproduce the error on a VirtualBox.

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.

Parts of the manual concerning **xargs** compatibility is inspired by the manual of **xargs** from GNU findutils 4.4.2.

LICENSE

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 <http://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 fdl.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 cc-by-sa.txt.

DEPENDENCIES

GNU **parallel** uses Perl, and the Perl modules Getopt::Long, IPC::Open3, Symbol, IO::File, POSIX, and File::Temp. For remote usage it also uses rsync with ssh.

SEE ALSO

ssh(1), **rsync**(1), **find**(1), **xargs**(1), **dirname**(1), **make**(1), **pexec**(1), **ppss**(1), **xjobs**(1), **prll**(1), **dxargs**(1), **mdm**(1)