NAME

parallel_alternatives - Alternatives to GNU parallel

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.

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

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

SUMMARY 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 the complete command line

Outputs O1. Grouping output so output from different jobs do not mix O2. Send stderr (standard error) to stderr (standard error) O3. Send stdout (standard output) to stdout (standard output) O4. Order of output can be same as order of input O5. Stdout only contains stdout (standard output) from the command O6. Stderr only contains stderr (standard error) from the command O7. Buffering on disk O8. Cleanup of file if killed O9. Test if disk runs full during run O10. Output of a line bigger than 4 GB

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

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

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

Legend - = no x = not applicable ID = yes

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 O7 O8 O9 O10 E1 E2 E3 E4 E5 E6 E7 R1 R2 R3 R4 R5 R6 R7 R8 R9 S1 S2
find -exec: - - - x - x - M2 M3 - - - - O2 O3 O4 O5 O6 - - - - - - - - - - - - - - - - - x x
make -j: - - - - - - - O1 O2 O3 - x O6 E1 - - - E5 - - - - - - -

DIFFERENCES BETWEEN xargs AND GNU Parallel

Summary table (see legend above): I1 I2 - - - - - M2 M3 - - - O2 O3 - O5 O6 E1 - - - - - - - - - - - -
**xargs** offers some of the same possibilities as GNU **parallel**.

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

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

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

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

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

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

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

```
parallel perl -e '@a=10000001;print \n$a,\n
'  -n1 grep --line-buffered 1 | \
  tr tr a-z
```

Or try this:

```
slow_seq() { 
  echo Count to "$0"
  seq "$@" | 
    perl -ne 'S|a|=1; for(split//){ print; select($a,$a,$a,0.100);}'
}
export -f slow_seq
```

```
# Serial = no mixing = the wanted result
seq 8 | xargs -n1 -P1 -l {} bash -c 'slow_seq {}'
# Compare to 8 jobs in parallel
seq 8 | parallel -f 8 slow_seq
seq 8 | parallel -P8 slow_seq
```

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

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

**xargs** has no support for context replace, so you will have to create the arguments.
If you use a replace string in `xargs` (-I) you can not force `xargs` to use more than one argument.

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

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

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

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

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

**DIFFERENCES BETWEEN find -exec AND GNU Parallel**

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

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

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

**DIFFERENCES BETWEEN make -j AND GNU Parallel**

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

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

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

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

**DIFFERENCES BETWEEN ppss AND GNU Parallel**

Summary table (see legend above): I1 I2 - - - - I7 M1 - M3 - - M6 O1 - - x - - E1 E2 ?E3 E4 - - R1 R2 R3 R4 - - ?R7 ? ? - -

`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. Filenames containing newlines are not processed correctly. When reading input from a file null cannot be used as a terminator. `ppss` needs to read the whole input file before starting any jobs.

Output and status information is stored in `ppss_dir` and thus requires cleanup when completed. If the dir is not removed before running `ppss` again it may cause nothing to happen as `ppss` thinks the task is already done. GNU `parallel` will normally not need cleaning up if running locally and will only need cleaning up if stopped abnormally and running remote (no `--cleanup` may not complete if stopped abnormally). The example `Parallel grep` would require extra postprocessing if written using `ppss`.

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

**EXAMPLES FROM ppss MANUAL**

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

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

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

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

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

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

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

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

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

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

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

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

6$ ./ppss stop -C config.cfg

6$ killall -TERM parallel

7$ ./ppss pause -C config.cfg

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

8$ ./ppss continue -C config.cfg

8$ Enter: fg or killall -SIGCONT parallel

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

9$ killall -SIGUSR2 parallel

https://github.com/louwrentius/PPSS

DIFFERENCES BETWEEN pexec AND GNU Parallel

Summary table (see legend above): I1 I2 - I4 I5 - - M1 - M3 - - M6 O1 O2 O3 - O5 O6 E1 - - E4 - E6 - R1 - - - - R6 - - - S1 -
GNU Parallel alternatives

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

**EXAMPLES FROM pexec MANUAL**

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

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

2$ seq 10 | parallel -j4 'echo "scale=10000;sqrt({})" | \
   bc > sqrt-{}.dat`

2$ pexec -p "$(ls myfiles*.ext)" -i %s -o %s.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 \n   '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 | \n   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) \n   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 \n   'pexec -j -m blockread -d $IMG | \
   jpegtopnm | pnmscale 0.5 | pnmtojpeg | \
   pexec -j -m blockwrite -s th_$_IMG'

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

# If reading and writing is done to the same disk, this may be \n# faster as only one process will be either reading or writing: \nls *.jpg | parallel -j8 'sem --id diskio cat () | jpegtopnm |'
```
GNU Parallel alternatives

'pnmscale 0.5 | pnmtojpeg | sem --id diskio cat > th_{}'

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

DIFFERENCES BETWEEN xjobs AND GNU Parallel

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

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

EXAMPLES FROM xjobs MANUAL

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

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

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

DIFFERENCES BETWEEN prll AND GNU Parallel

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

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

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

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

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

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

https://github.com/exzombie/prll (Last checked: 2019-01)

DIFFERENCES BETWEEN dxargs AND GNU Parallel

dxargs is also a tool for running jobs in parallel.

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


DIFFERENCES BETWEEN mdm/middleman AND GNU Parallel

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

EXAMPLES FROM middleman's WEBSITE

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

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

https://github.com/cklin/mdm (Last checked: 2019-01)

DIFFERENCES BETWEEN xapply AND GNU Parallel

xapply can run jobs in parallel on the local computer.

EXAMPLES FROM xapply's MANUAL

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

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

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

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

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

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

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

4$ xapply 'indent' *.c

4$ parallel indent ::: *.c

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

DIFFERENCES BETWEEN AIX apply AND GNU Parallel

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

EXAMPLES FROM IBM's KNOWLEDGE CENTER

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

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

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

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

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

To run the `who` command five times, enter:

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

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

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


**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*.

**EXAMPLES FROM paexec’s EXAMPLE CATALOG**

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

`1_div_X_run`

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

`all_substr_run`

```
2$ ../../paexec -lp -c "`pwd`/all_substr_cmd" -n +3 <<EOF [...] 
2$ parallel echo () '|' `pwd`/all_substr_cmd <<EOF [...] 
```

`cc_wrapper_run`

```
3$ ../../paexec -c "env CC=gcc CFLAGS=-O2 `pwd`/cc_wrapper_cmd" \ 
   -n 'host1 host2' \ 
   -t '/usr/bin/ssh -x' <<EOF [...] 
3$ 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`

```
4$ ../../paexec -lp -c "`pwd`/toupper_cmd" -n +10 <<EOF [...] 
4$ parallel echo () '|'.toupper_cmd <<EOF [...] 
   # Without the wrapper: 
   parallel echo () '|' awk (print\ toupper\(\$0\))' <<EOF [...] 
```
DIFFERENCES BETWEEN map(sitaramc) AND GNU Parallel

Summary table (see legend above): I1 - - I4 - - (I7) M1 (M2) M3 (M4) M5 M6 - O2 O3 - O5 - - N/A N/A O10 E1 - - - - - - - - - - - - - - - - -

(I7): Only under special circumstances. See below.
(M2+M4): Only if there is a single replacement string.

map rejects input with special characters:

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

It works with GNU parallel:

```bash
ls | parallel 'echo %; wc %'
```

Under some circumstances it also works with map:

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

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

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

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

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

map prints '+' on stderr when a job starts, and '-' when a job finishes. This cannot be disabled.

parallel has --bar if you need to see progress.

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

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

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

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

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

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

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

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

map has no support for grouping. So this gives the wrong results:
parallel perl -e '
$a="1{a}"x10000000;print $a,"\n"'

parallel -l a b c d e f

ls -l a b c d e f

parallel -kP4 -n1 grep 1 ::: a b c d e f > out.par

map -n1 -p 4 'grep --line-buffered 1' a b c d e f > out.map-linebuf

map -n1 -p 1 'grep --line-buffered 1' a b c d e f > out.map-serial

ls -l out*

md5sum out*

EXAMPLES FROM map's WEBSITE

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

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

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

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

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

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

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

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

4$ ls *.tar | parallel tar -xf

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

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

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

6$ parallel "du -sm /home/{}/mail" ::: alice bob carol

or if you prefer running a single job with multiple args:

6$ parallel -Xj1 "du -sm /home/{}/mail" ::: alice bob carol

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

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

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

8$ export PARALLEL=-j50%

https://github.com/sitaramc/map (Last checked: 2020-05)
DIFFERENCES BETWEEN ladon AND GNU Parallel

ladon can run multiple jobs on files in parallel.

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

As replacement strings it uses FULLPATH DIRNAME BASENAME EXT RELDIR RELPATH

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

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

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

```bash
ladon "*" -- seq 36000 | wc
```

EXAMPLES FROM ladon MANUAL

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

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

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

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

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

https://github.com/danielgtaylor/ladon (Last checked: 2019-01)

DIFFERENCES BETWEEN jobflow AND GNU Parallel

jobflow can run multiple jobs in parallel.

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

**jobflow** gives no error if the command is unknown, and like **xargs** redirection and composed commands require wrapping with **bash** `-c`.

Input lines can at most be 4096 bytes. You can at most have 16 `{}`'s in the command template. More than that either crashes the program or simple does not execute the command.

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

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

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

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

**EXAMPLES FROM jobflow README**

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

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

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

```
2$ seq 100 | parallel -j100 echo {}
```

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

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

```
4$ find . -name '*.bmp' | \
   jobflow -threads=8 -exec bmp2jpeg {}.bmp {}.jpg
```

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

https://github.com/rofl0r/jobflow

**DIFFERENCES BETWEEN gargs AND GNU Parallel**

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

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

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

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

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

**EXAMPLES FROM gargs WEBSITE**

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

```
1$ seq 12 -1 1 | parallel -P 4 -n 3 "sleep (1); echo (2) (3)"
```

```
2$ cat t.txt | gargs --sep "\s+" \
```
GNU Parallel alternatives

- `p 2 "echo \'(0):(1)-(2)' full-line: \\
'(\{})\''`

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

https://github.com/brentp/gargs

DIFFERENCES BETWEEN orgalorg AND GNU Parallel

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

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

To make the emulation easier, make a simple alias:

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

If you want to supply a password run:

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

or set the password directly:

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

If the above is set up you can then do:

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

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

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

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

https://github.com/reconquest/orgalorg

DIFFERENCES BETWEEN Rust parallel AND GNU Parallel

Rust parallel focuses on speed. It is almost as fast as xargs. It implements a few features from GNU parallel, but lacks many functions. All these fail:

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

These do something different from GNU parallel

```
# -q to protect quoted $ and space
parallel -q perl -e '$a=shift; print "$a"x10000000' ::: a b c
# Generation of combination of inputs
```
GNU Parallel alternatives

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

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

Rust parallel has no remote facilities.

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

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

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

https://github.com/mmstick/parallel

DIFFERENCES BETWEEN Rush AND GNU Parallel

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

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

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

  --rpl '(?:s:(\^[\^]+)+$::'
  --rpl '(?:%([^]+)+)) s:$1\^[\^]+)+$::'
  --rpl '\(?:s:\([^]+\)) s:.*\$/\([^\^[^/]+\)+\$::'
  --rpl '\(?:s:\([^]+\))/\([^\[\^/]+\)+\$::'
  --rpl '\(?:s:\([^]+\))/\([^\[\^/]+\)+\$::'

EXAMPLES FROM rush's WEBSITE

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

1. Simple run, quoting is not necessary

  $ seq 1 3 | rush echo ()
$ seq 1 3 | parallel echo ()

2. Read data from file (`-i`)
$ rush echo {} -i data1.txt -i data2.txt
$ cat data1.txt data2.txt | parallel echo {}

3. Keep output order (`-k`)
$ seq 1 3 | rush 'echo {}' -k
$ seq 1 3 | parallel -k echo {}

4. Timeout (`-t`)
$ time seq 1 | rush 'sleep 2; echo {}' -t 1
$ time seq 1 | parallel --timeout 1 'sleep 2; echo ()'

5. Retry (`-r`)
$ seq 1 | rush 'python unexisted_script.py' -r 1
$ seq 1 | parallel --retries 2 'python unexisted_script.py'

Use -u to see it is really run twice:
$ seq 1 | parallel -u --retries 2 'python unexisted_script.py'

6. Dirname (`{/}`) and basename (`{%}`) and remove custom suffix (`{^suffix}`)
$ echo dir/file_1.txt.gz | rush 'echo /{}/{} (^_1.txt.gz)'
$ echo dir/file_1.txt.gz | parallel --plus echo /{}/{} (^_1.txt.gz)

7. Get basename, and remove last (`{.}`) or any (`{::}`) extension
$ echo dir.d/file.txt.gz | rush 'echo {}.{} {}:: {}::'
$ echo dir.d/file.txt.gz | parallel 'echo {}.{} {}.{}' /

8. Job ID, combine fields index and other replacement strings
$ echo 12 file.txt dir/s_1.fq.gz | rush 'echo job # : {2} {2.} {3%:^_1}'
$ echo 12 file.txt dir/s_1.fq.gz | parallel --colsep ' ' 'echo job # : {2} {2.} {3/:^_1}'

9. Capture submatch using regular expression (`{@regexp}`)
$ echo read_1.fq.gz | rush 'echo @{(.+)_d}'
10. Custom field delimiter (`-d`)  
$ echo a=b=c | rush "echo (1) (2) (3)" -d =  
$ echo a=b=c | parallel -d = echo (1) {2} {3}

11. Send multi-lines to every command (`-n`)  
$ seq 5 | rush -n 2 -k 'echo "()"'; echo'  
$ seq 5 | parallel -n 2 -k \  
  'echo {-1 $=_join"
",@arg[1..$#arg] =}; echo'  
$ seq 5 | rush -n 2 -k 'echo "()"'; echo' -J '  
$ seq 5 | parallel -n 2 -k 'echo {}; echo'

12. Custom record delimiter (`-D`), note that empty records are not used.  
$ echo a b c d | rush -D " " -k 'echo {}'  
$ echo a b c d | parallel -d " " -k 'echo {}'  
$ echo abcd | rush -D "" -k 'echo {}'

Cannot be done by GNU Parallel

$ cat fasta.fa  
>seq1  
tag  
>seq2  
cat  
gat  
>seq3  
attac  
a  
cat  

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

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

13. Assign value to variable, like `awk -v` (`-v`)  
$ seq 1 |  
  rush 'echo Hello, {fname} {lname}!' -v fname=Wei -v lname=Shen
$ seq 1 |
    parallel -N0 \
    'fname=Wei; lname=Shen; echo Hello, ${fname} ${lname}!'

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

In GNU parallel you would typically do:

$ seq 1 3 | parallel -k echo var: (1), data: (2) ::: a b :::: -

If you really want the var:

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

If you really want the for-loop:

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

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

My brother's 12" records

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

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

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

# macro + removing suffix
$ echo read_1.fq.gz |
    rush -v p='{:^_1}' 'echo {p} {p}_2.fq.gz'

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

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

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

Contrary to rush GNU parallel works with complex values:

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

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

$ seq 1 20 | rush 'sleep 1; echo {}'
16. Continue/resume jobs (`-c`). When some jobs failed (by execution failure, timeout, or canceling by user with `Ctrl + C`), please switch flag `-c/--continue` on and run again, so that `rush` can save successful commands and ignore them in NEXT run.

```
$ seq 1 3 | rush 'sleep {}; echo {}' -t 3 -c
$ cat successful_cmds.rush
$ seq 1 3 | parallel --joblog mylog --timeout 2 'sleep {}; echo {}'
$ cat mylog
$ seq 1 3 | parallel --joblog mylog --retry-failed 'sleep {}; echo {}'
```

Multi-line jobs:

```
$ seq 1 3 |
  parallel --joblog mylog --timeout 2 'sleep {}; echo {}';
  echo finish {}' -t 3 -c -C finished.rush
$ cat finished.rush
$ seq 1 3 |
  parallel --joblog mylog --timeout 2 'sleep {}; echo {}';
  echo finish {}' -t 3 -c -C finished.rush

$ seq 1 3 |
  parallel --joblog mylog --timeout 2 'sleep {}; echo {}';
  echo finish {}' -t 3 -c -C finished.rush
$ cat mylog
$ seq 1 3 |
  parallel --joblog mylog --retry-failed 'sleep {}; echo {}';
  echo finish {}'
```

17. A comprehensive example: downloading 1K+ pages given by three URL list files using `phantomjs save_page.js` (some page contents are dynamically generated by Javascript, so `wget` does not work). Here I set max jobs number (`-j`) as `20`, each job has a max running time (`-t`) of `60` seconds and `3` retry changes (`-r`). Continue flag `-c` is also switched on, so we can continue unfinished jobs. Luckily, it's accomplished in one run :)

```
$ for f in $(seq 2014 2016); do
  /bin/rm -rf $f;
  mkdir -p $f;
  cat $f.html.txt | rush -v d=$f d=
    'phantomjs save_page.js "{}" > {d}/(3).html' \
    --retries 4 --joblog +mylog \ 
    --colsep = \
  done
```

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

```
$ rm mylog
$ for f in $(seq 2014 2016); do
  /bin/rm -rf $f; mkdir -p $f;
  cat $f.html.txt |
  parallel -j20 --timeout 60 --retries 4 --joblog +mylog \
  --colsep = \
```
GNU Parallel alternatives

phantomjs save_page.js (1)=(2)=(3) '=>' $f/(3).html

done

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

```bash
$ ref=ref/xxx.fa
$ threads=25
$ ls -d raw.cluster.clean.mapping/* \ 
  | rush -v ref=$ref -v j=$threads -v p='{}/{}' \ 
  'bwa mem -t {j} -M -a {ref} {p}_1fq.gz {p}_2fq.gz >{p}.sam; \ 
  samtools view -bS {p}.sam > {p}.bam; \ 
  samtools sort -T {p}.tmp -@ {j} {p}.bam -o {p}.sorted.bam; \ 
  samtools index {p}.sorted.bam; \ 
  samtools flagstat {p}.sorted.bam > {p}.sorted.bam.flagstat; \ 
  /bin/rm {p}.bam {p}.sam;' \ 
  -j 2 --verbose -c -C mapping.rush
```

GNU parallel would use a function:

```bash
$ ref=ref/xxx.fa
$ export ref
$ thr=25
$ export thr
$ bwa_sam() { 
  p="$1"
  bam="$p".bam
  sam="$p".sam
  sortbam="$p".sorted.bam
  bwa mem -t $thr -M -a $ref ${p}_1fq.gz ${p}_2fq.gz >"$sam"
  samtools view -bS "$sam" > "$bam"
  samtools sort -T ${p}.tmp -@ $thr "$bam" -o "$sortbam"
  samtools index "$sortbam"
  samtools flagstat "$sortbam" > "$sortbam".flagstat 
  /bin/rm "$sortbam" "$sam"
}
$ export -f bwa_sam
$ ls -d raw.cluster.clean.mapping/* | parallel -j 2 --verbose --joblog mylog bwa_sam
```

Other rush features

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

  With GNU parallel you would simply set a shell variable:

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

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

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

  Whereas the corresponding GNU parallel version works:

  ```bash
  parallel 'v=""; echo "$v" :::: works 
  parallel 'v={}; echo "$v" :::: "My brother's 12" records
  ```
* Exit on first error(s) (-e)
  This is called --halt now, fail=1 (or shorter: --halt 2) when used with GNU parallel.

* Settable records sending to every command (-n, default 1)
  This is also called -n in GNU parallel.

* Practical replacement strings
  
  {} remove any extension
  
  With GNU parallel this can be emulated by:
  
  ```
  parallel --plus echo '{/\..*/}' :::: foo.ext.bar.gz
  ```

  {{suffix}}, remove suffix
  
  With GNU parallel this can be emulated by:
  
  ```
  parallel --plus echo '{%bar.gz}' :::: foo.ext.bar.gz
  ```

  @regexp, capture submatch using regular expression
  
  With GNU parallel this can be emulated by:
  
  ```
  parallel --rpl '{@\(.*\)}/$1/ and $_=1;' \
    echo '{@\d.(.*)\(\)gz}' :::: l_foo.gz
  ```

  {%}, {%:}, basename without extension
  
  With GNU parallel this can be emulated by:
  
  ```
  parallel echo '{= s:.*/::;s/\..*// =}' :::: dir/foo.bar.gz
  ```

  And if you need it often, you define a --rpl in $HOME/.parallel/config:
  
  ```
  --rpl '{\} s:.*/::;s/\..*//'
  --rpl '{\} s:.*/::;s/\..*//'
  ```

  Then you can use them as:
  
  ```
  parallel echo {%} {%} :::: dir/foo.bar.gz
  ```

* Preset variable (macro)

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

  With GNU parallel this can be emulated by:
  
  ```
  echo foosuffix |
  parallel --plus 'p=%suffix); echo ${p}_new_suffix'
  ```

  Opposite rush GNU parallel works fine if the input contains double space, ' and ":
  
  ```
  echo "1'6"" foosuffix" |
  parallel --plus 'p=${suffix); echo "${p}"_new_suffix'
  ```

* Commands of multi-lines

  While you can use multi-lined commands in GNU parallel, to improve readability GNU parallel discourages the use of multi-line commands. In most cases it can be written as a function:
  
  ```
  seq 1 3 |
  parallel --timeout 2 --joblog my.log 'sleep {}; echo {}; \\n  echo finish {}'
  ```
Could be written as:

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

The failed commands can be resumed with:

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

https://github.com/shenwei356/rush

DIFFERENCES BETWEEN ClusterSSH AND GNU Parallel

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

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

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

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

https://github.com/duncs/clusterssh

DIFFERENCES BETWEEN coshell AND GNU Parallel

coshell only accepts full commands on standard input. Any quoting needs to be done by the user. Commands are run in sh so any bash/tcsh/zsh specific syntax will not work.

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

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

DIFFERENCES BETWEEN spread AND GNU Parallel

spread runs commands on all directories.

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

spread() {
    _cmds() {
        perl -e "$"=" && "print "ARGV"; "cd {}" "$0"
    } parallel $(/_cmds "$0")' | echo exit status $? :: :: */
}

This works except for the --exclude option.

(Last checked: 2017-11)
DIFFERENCES BETWEEN pyargs AND GNU Parallel

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

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

Examples

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

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

seq 5 | pyargs -P50 --mark command
    # Somewhat longer with GNU Parallel due to the special
    # --mark formatting
    cmd="$(echo "command" | parallel --shellquote)"
    wrap_cmd() {
        echo "MARK $cmd $@=""""""""""""""""">&3
        echo "OUTPUT START[$cmd $@]:"
        eval $cmd "$@
        echo "OUTPUT END[$cmd $@]"
    }
    (seq 5 | env_parallel -P2 wrap_cmd) 3>&1
    # Similar, but not exactly the same
seq 5 | parallel -t --tag command

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

https://github.com/robertblackwell/pyargs (Last checked: 2019-01)

DIFFERENCES BETWEEN concurrently AND GNU Parallel

**concurrently** runs jobs in parallel.

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

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

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

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

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

https://github.com/kimmobrunfeldt/concurrently (Last checked: 2019-01)
DIFFERENCES BETWEEN map(soeveran) AND GNU Parallel

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

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

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

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

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

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

EXAMPLES FROM maps WEBSITE

Here are the 5 examples converted to GNU Parallel:

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

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

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

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

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

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

DIFFERENCES BETWEEN loop AND GNU Parallel

**loop** mixes stdout and stderr:

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

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

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

**loop** cannot run functions:

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

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

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

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

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

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

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

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

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

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

$ loop 'echo hello'
$ loopy echo hello

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

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

$ loop 'echo $COUNT' --count-by 2 --offset 10
```
GNU Parallel alternatives

$ loopy echo '($_=10+2*(seq()-1) =)'

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

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

$ loopy echo '($_=10+2*(seq()-1) =)'
$ loop 'echo $COUNT' --count-by 1.1
# --joblog is somewhat more verbose than --summary
$ seq 3 | parallel --joblog my.log echo; cat my.log

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

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

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

$ loop 'date' --for-duration 8s --every 2s
$ time_out 8s loopy --delay 2s date

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$ ls | loop 'cp $ITEM $ITEM.bak'
$ ls | parallel cp {} {}.bak
DIFFERENCES BETWEEN lorikeet AND GNU Parallel

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


DIFFERENCES BETWEEN spp AND GNU Parallel

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

https://github.com/john01dav/spp (Last checked: 2019-01)

DIFFERENCES BETWEEN paral AND GNU Parallel

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

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

Instead it puts the output into files named like `out_#_command.out.log`. To get a very similar behaviour with GNU parallel use --results 'out_#_([=s]/[^a-z0-9]/[g];[s]/[^a-z0-9]/[g]=).log' --eta

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

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

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

EXAMPLES FROM README.markdown

The examples from README.markdown and the corresponding command run with GNU parallel (--results 'out_#_([=s]/[^a-z0-9]/[g];[s]/[^a-z0-9]/[g]=).log' --eta is omitted from the GNU parallel command):

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

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

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

```
4$ paral -n=1 "sleep 5 && echo c5" "sleep 4 && echo c4"
    "sleep 3 && echo c3" "sleep 2 && echo c2" "sleep 1 && echo c1"
4$ parallel -j 1 "sleep {} && echo c{}" ::: 5 4 3 2 1
```
DIFFERENCES BETWEEN concurr AND GNU Parallel

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

EXAMPLES FROM README.md

The examples from README.md:

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

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

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

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

DIFFERENCES BETWEEN lesser-parallel AND GNU Parallel

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

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

https://github.com/kou1okada/lesser-parallel (Last checked: 2019-01)

DIFFERENCES BETWEEN npm-parallel AND GNU Parallel

npm-parallel can run npm tasks in parallel.

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

https://github.com/spion/npm-parallel (Last checked: 2019-01)
DIFFERENCES BETWEEN machma AND GNU Parallel

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

EXAMPLES FROM README.md

The examples from README.md:

$ # Put shorthand for timestamp in config for the examples
  echo --rpl '
  \"\{time\}\: \{\}\n  \" -v ~/.parallel/machma
  echo --line-buffer --tagstring "\{\} (\)"
  >> ~/.parallel/machma

$ find . -iname '*.jpg' |
  machma -- mogrify -resize 1200x1200 -filter Lanczos {}

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

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

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

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

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

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

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

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

DIFFERENCES BETWEEN interlace AND GNU Parallel

Summary table (see legend above): - I2 I3 I4 - - M1 - M3 - - M6 - O2 O3 - - - - x x E1 E2 - - - - - - - - - -

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

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

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

EXAMPLES FROM interlace's WEBSITE

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

Blocker

commands.txt:

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

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

Blocks

commands.txt:

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

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

Run Nikto Over Multiple Sites

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

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

Run Nikto Over Multiple Sites and Ports

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

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

Run a List of Commands against Target Hosts

commands.txt:

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

parallel --results ~/Engagements/example/{}:{2}:{3}{1} {1} (2):{3} \n  ::: "nikto --host" sslscan testssl.sh ::: example.com ::: 80 443
CIDR notation with an application that doesn't support it

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

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

Glob notation with an application that doesn't support it

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

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

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

```
interlace -t 192.168.12.1-15 -c "vhostscan _target_ \
-oN _output_/._target_-vhosts.txt" \n-o ~/scans/ -threads 50

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

Threading Support for an application that doesn't support it

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

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

alternatively

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

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

Exclusions

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

prips 192.168.12.0/24 | grep -xv -Ff <(prips 192.168.12.0/26) |
GNU Parallel alternatives

parallel -P50 vhostscan () -oN ~/scans/{}-vhosts.txt

Run Nikto Using Multiple Proxies

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

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

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

DIFFERENCES BETWEEN ontonvm Parallel AND GNU Parallel

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

https://github.com/otonvm/Parallel (Last checked: 2019-02)

DIFFERENCES BETWEEN k-bx par AND GNU Parallel

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

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

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

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

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

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

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

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

DIFFERENCES BETWEEN parallelshell AND GNU Parallel

parallelshell does not allow for composed commands:

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

Instead you have to wrap that in a shell:

parallelshell 'sh -c "echo foo;echo bar"' 'sh -c "echo baz;echo quuz"'
It buffers output in RAM. All commands must be given on the command line and all commands are started in parallel at the same time. This will cause the system to freeze if there are so many jobs that there is not enough memory to run them all at the same time.

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


**DIFFERENCES BETWEEN shell-executor AND GNU Parallel**

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

```bash
sx 'echo foo;echo bar' 'echo baz;echo quuz'
```

Instead you have to wrap that in a shell:

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

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


**DIFFERENCES BETWEEN non-GNU par AND GNU Parallel**

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

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

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

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


**DIFFERENCES BETWEEN fd AND GNU Parallel**

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

It buffers output in RAM.

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

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

**DIFFERENCES BETWEEN lateral AND GNU Parallel**

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

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

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

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

```bash
lateral run ls -l
```

Composed commands do not work:
lateral run pwd ';' ls

Functions do not work:

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

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

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

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

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

2$ lateral start
2$ for i in $(seq 1 100); do
2$   lateral run -- my_slow_command < workfile$i > /tmp/logfile$i
2$ done
2$ lateral wait
2$
2$ for i in $(seq 1 100); do
2$   sem my_slow_command < workfile$i > /tmp/logfile$i
2$ done
2$ sem --wait
2$
2$ parallel 'my_slow_command < workfile{} > /tmp/logfile{}' \
   ::: {1..100}

3$ lateral start -p 0 # yup, it will just queue tasks
3$ for i in $(seq 1 100); do
3$   lateral run -- command_still_outputs_but_wont_spam inputfile$i
3$ done
3$ # command output spam can commence
3$ lateral config -p 10; lateral wait
3$
3$ for i in $(seq 1 100); do
3$   echo "command inputfile$i" >> joblist
3$ done
3$ parallel -j 10 :::: joblist
3$
3$ echo 1 > /tmp/njobs
3$ parallel -j /tmp/njobs command inputfile() \
   :::: {1..100} &
3$ echo 10 > /tmp/njobs
3$ wait

https://github.com/akramer/lateral (Last checked: 2019-03)

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

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

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

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

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

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

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

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

https://github.com/amritb/with-this.git (Last checked: 2019-03)

DIFFERENCES BETWEEN Tollef's parallel (moreutils) AND GNU Parallel
Summary table (see legend above): - - - I4 - - I7 - - M3 - - M6 - O2 O3 - O5 O6 - x x E1 - - - - - E7 - x x x x x x x x x

EXAMPLES FROM Tollef's parallel MANUAL

**Tollef** parallel sh -c "echo hi; sleep 2; echo bye" -- 1 2 3
**GNU** parallel "echo hi; sleep 2; echo bye" :::: 1 2 3
**Tollef** parallel -j 3 ufraw -o processed -- *.NEF
**GNU** parallel -j 3 ufraw -o processed :: *.NEF
**Tollef** parallel -j 3 -- ls df "echo hi"
**GNU** parallel -j 3 ::: ls df "echo hi"

(Last checked: 2019-08)
DIFFERENCES BETWEEN rargs AND GNU Parallel

Summary table (see legend above):

<table>
<thead>
<tr>
<th></th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O4</th>
<th>O5</th>
<th>O6</th>
<th>O7</th>
<th>O8</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>E4</th>
</tr>
</thead>
<tbody>
<tr>
<td>rargs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNU parallel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

would make it possible to use:

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

for field 3..6.

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

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

You can then do:

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

**EXAMPLES FROM rargs MANUAL**

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

cat download-list.csv | rargs -p '\(?P<url>.*\),(?P<filename>.*\)' wget {url} -O {filename}
cat download-list.csv | parallel --csv wget {1} -O {2} # or use regexps:
cat download-list.csv |
    parallel --rpl '\{url\} s/,.*//' --rpl '\(filename\) s/.*?,//' wget {url} -O {filename}

cat /etc/passwd | rargs -d: echo -e 'id: "\{1\}\tname: "\{5\}\trest: "\{6..::\}"'
cat /etc/passwd |
    parallel -q --colsep : echo -e 'id: "\{1\}\tname: "\{5\}\trest: "\{6
    \_=join":",@arg[6..$#arg]="}'
```

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

DIFFERENCES BETWEEN threader AND GNU Parallel

Summary table (see legend above):

<table>
<thead>
<tr>
<th></th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O4</th>
<th>O5</th>
<th>O6</th>
<th>O7</th>
<th>O8</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>E4</th>
</tr>
</thead>
<tbody>
<tr>
<td>threader</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNU parallel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Newline separates arguments, but newline at the end of file is treated as an empty argument. So this
GNU Parallel alternatives

runs 2 jobs:

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

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

https://github.com/voodooEntity/threader (Last checked: 2020-04)

**DIFFERENCES BETWEEN runp AND GNU Parallel**

Summary table (see legend above): I1 I2 - - - - - M1 - (M3) - - M6 O1 O2 O3 - O5 O6 - N/A N/A - E1 - - - - - - - - - -

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

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

**Examples as GNU Parallel**

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

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

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

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

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

**Run some test commands (read from file)**

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

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

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

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

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

Get directories' sizes (read from stdin)

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

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

Compress files

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

    find . -iname "*.txt" | parallel gzip --best

Measure HTTP request + response time

    export CURL="curl -w 'time_total: %{time_total}\n"
    CURL="$CURL -o /dev/null -s https://golang.org/"
    perl -wE 'for (1..10) { say $ENV{CURL} }' | runp -q # Make 10 requests

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

Find open TCP ports

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

    cat /tmp/host-port.txt | \

runp -q -p 'netcat -v -w2 -z' 2>&1 | grep "(succeeded!|open)"

# --colsep is needed to split the line
parallel --colsep ' ' netcat -v -w2 -z 2>&1 | grep "(succeeded!|open)"

# or use uq for unquoted:
parallel netcat -v -w2 -z {=uq=} 2>&1 | egrep '(succeeded!|open)'

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

DIFFERENCES BETWEEN papply AND GNU Parallel

Summary table (see legend above):

<table>
<thead>
<tr>
<th></th>
<th>papply</th>
<th>GNU Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>--rpl '%F'</td>
<td>--rpl '%F'</td>
</tr>
<tr>
<td>□</td>
<td>--rpl '%d $=<em>Q(::dirname($$</em>));'</td>
<td>--rpl '%d $=<em>Q($$</em>);'</td>
</tr>
<tr>
<td>□</td>
<td>--rpl '%f s::*:/::;'</td>
<td>--rpl 's::*:/::;'</td>
</tr>
<tr>
<td>□</td>
<td>--rpl '%n s::*:/::;s:.[^/^]+$::;'</td>
<td>--rpl '%n s::*:/::;s:.[^/^]+$::;'</td>
</tr>
<tr>
<td>□</td>
<td>--rpl '%e s::*.:;'</td>
<td>--rpl '%e s::*.:;'</td>
</tr>
<tr>
<td>□</td>
<td>--rpl '%z $=_&quot;&quot;'</td>
<td>--rpl '%z $=_&quot;&quot;'</td>
</tr>
</tbody>
</table>

papply does not print the output if the command fails:

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

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

```bash
--rpl '%F'
--rpl '%d $=_Q(::dirname($$_));'
--rpl '%f s::*:/::;'  
--rpl '%n s::*:/::;s:\.[^/\^]+$::;'  
--rpl '%e s::*\.:;'  
--rpl '%z $=_""'
```

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

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

Examples as GNU Parallel

```bash
1$ papply gzip *.txt
1$ parallel gzip ::: *.txt
2$ papply "convert %F %n.jpg" *.png
2$ parallel convert () () .jpg ::: *.png
```

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

Todo

test_many_var() { gen500k() { seq -f %f 10000000000000000 1000000000000000 | head -c 131000; } for a in `seq 11000`; do eval "export a$a=1"; done gen500k | stdout parallel --timeout 5 -Xj1 'echo {} {} | wc' | perl -pe 's/d(3,5) //g' }

test_many_var_func() { gen500k() { seq -f %f 10000000000000000 10000000000000000 | head -c 131000; } for a in `seq 5100`; do eval "export a$a=1"; done gen500k | stdout parallel --timeout 21 -Xj1 'echo {} {} | wc' | perl -pe 's/d(3,5) //g' }

Todo

```bash
test_many_var_func() { gen500k() { seq -f %f 1000000000000000 10000000000050000 | head -c 131000; } for a in `seq 50`; do eval "a$a() { 1; }"; done for a in `seq 50`; do eval export -f a$a; done gen500k | stdout parallel --timeout 6 -Xj1 'echo {} {} | wc' | perl -pe 's/\d(3,5) //g' }
test_big_func() { gen500k() { seq -f %f 1000000000000000 10000000000050000 | head -c 131000; } big=`seq 100` for a in `seq 50`; do eval "a$a() { "$big;" }"; done for a in `seq 50`; do eval export -f a$a; done gen500k | stdout parallel --timeout 6 -Xj1 'echo {} {} | wc' | perl -pe 's/\d(3,5) //g' }
test_many_var_big_func() { gen500k() { seq -f %f 1000000000000000 10000000000050000 | head -c 131000; } big=`seq 100` for a in `seq 50`; do eval "export a$a=1"; done for a in `seq 20`; do eval "a$a() { "$big"; }"; done for a in `seq 20`; do eval export -f a$a; done gen500k | stdout parallel --timeout 6 -Xj1 'echo {} {} | wc' | perl -pe 's/\d(3,5) //g' }
test_big_func_name() { gen500k() { seq -f %f 1000000000000000 10000000000050000 | head -c 131000; } big=`perl -e print"x"x10000" for a in `seq 20"; do eval "export a$big$a=1"; done gen500k | stdout parallel --timeout 6 -Xj1 'echo {} {} | wc' | perl -pe 's/\d(3,5) //g' }
test_big_var_func_name() { gen500k() { seq -f %f 1000000000000000 10000000000050000 | head -c 131000; } big=`perl -e print"x"x10000" for a in `seq 2"; do eval "export a$big$a=1"; done for a in `seq 2"; do eval "a$big$a() { "$big"; }"; done for a in `seq 2"; do eval export -f a$big$a; done gen500k | stdout parallel --timeout 6 -Xj1 'echo {} {} | wc' | perl -pe 's/\d(3,5) //g' }
tangle@macosx:~$ for a in `seq 100`; do eval export a$a=fffffffffffffffffffffff ; donetangle@macosx:~$ seq 50000 | stdout parallel -Xj1 'echo {} {} | wc' | perl -pe 's/\d(3,5) //g' tangle@macosx:~$ for a in `seq 100`; do eval export a$a=fffffffffffffffffffffff ; donetangle@macosx:~$ seq 50000 | stdout parallel -Xj1 'echo {} {} | wc' | perl -pe 's/\d(3,5) //g' tangle@macosx:~$ for a in `seq 100`; do eval export -f a$a; done
seq 100000 | stdout parallel -Xj1 'echo {} {} | wc' export a=`seq 10000` seq 100000 | stdout parallel -Xj1 'echo {} {} | wc'

my $already_spread;
my $env_size;

if($"O eq "darwin") {
  $env_size ||= 500+length(join',' ,%ENV);
  $max_len -= $env_size;
}
```


https://gitlab.com/netikras/bthread

https://github.com/JeiKeiLim/simple_distribute_job

https://github.com/reggi/pkgrun

https://github.com/benoror/better-npm-run - not obvious how to use

https://github.com/bahmutov/with-package

https://github.com/xuchenCN/go-pssh

https://github.com/flesler/parallel

https://github.com/Julian/Verge

https://github.com/ExpectationMax/simple_gpu_scheduler simple_gpu_scheduler --gpus 0 1 2 <
gpu_commands.txt parallel -j3 --shuf CUDA_VISIBLE_DEVICES="'=1 $_.slot()-1 \= \= \(\= \=''"' <
gpu_commands.txt

  simple_hypersearch "python3 train_dnn.py --lr \{lr\} --batch_size \{bs\}"
GNU Parallel alternatives

-p lr 0.001 0.0005 0.0001 -p bs 32 64 128 | simple_gpu_scheduler --gpus 0,1,2
parallel --header : --shuf -j3 -v CUDA_VISIBLE_DEVICES='={1 $=_slot()-1
=}'}
python3 train_dnn.py --lr {lr} --batch_size {bs} ::: lr 0.001 0.0005 0.0001 :::

simple_hypersearch "python3 train_dnn.py --lr {lr} --batch_size {bs}" --n-samples 5 -p lr 0.001 0.0005 0.0001 -p bs 32 64 128 |

simple_gpu_scheduler --gpus 0,1,2
parallel --header : --shuf CUDA_VISIBLE_DEVICES='={1 $=_slot()-1; seq() > 5
and skip() =}'}
python3 train_dnn.py --lr {lr} --batch_size {bs} ::: lr 0.001 0.0005 0.0001 :::

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

touch gpu.queue
tail -f -n 0 gpu.queue | parallel -j3 CUDA_VISIBLE_DEVICES='={1
$_=slot()-1 =}'} (eq;=)'
# Needed to fill job slots once
seq 3 | parallel echo true >> gpu.queue
# Add jobs
echo "my_command_with | and stuff > logfile" >> gpu.queue
# Needed to flush output from completed jobs
seq 3 | parallel echo true >> gpu.queue

TESTING OTHER TOOLS

There are certain issues that are very common on parallelizing tools. Here are a few stress tests. Be warned: If the tool is badly coded it may overload your machine.

MIX: Output mixes

Output from 2 jobs should not mix. If the output is not used, this does not matter; but if the output is used then it is important that you do not get half a line from one job followed by half a line from another job.

If the tool does not buffer, output will most likely mix now and then.

This test stresses whether output mixes.

#!/bin/bash

paralleltool="parallel -j0"

cat <|--EOF > mycommand
#!/bin/bash

# If a, b, c, d, e, and f mix: Very bad
perl -e 'print STDOUT "a"x3000_000," "'
perl -e 'print STDOUT "b"x3000_000," "'
perl -e 'print STDOUT "c"x3000_000," "'
perl -e 'print STDOUT "d"x3000_000," "'
perl -e 'print STDOUT "e"x3000_000," "'
perl -e 'print STDOUT "f"x3000_000," "'
echo
echo >&2
EOF
chmod +x mycommand

# Run 30 jobs in parallel
seq 30 |
   $paralleltool ./mycommand > (tr -s abcdef) 2> (tr -s abcdef >&2)

# 'a c e' and 'b d f' should always stay together
# and there should only be a single line per job

**STDERRMERGE: Stderr is merged with stdout**

Output from stdout and stderr should not be merged, but kept separated.

This test shows whether stdout is mixed with stderr.

```
#!/bin/bash

paralleltool="parallel -j0"

cat <"-EOF" > mycommand
#!/bin/bash

echo stdout
echo stderr >&2
echo stdout
echo stderr >&2
EOF
chmod +x mycommand

# Run one job
echo |
   $paralleltool ./mycommand > stdout 2> stderr
cat stdout
cat stderr
```

**RAM: Output limited by RAM**

Some tools cache output in RAM. This makes them extremely slow if the output is bigger than physical memory and crash if the output is bigger than the virtual memory.

```
#!/bin/bash

paralleltool="parallel -j0"

cat <"-EOF" > mycommand
#!/bin/bash

# Generate 1 GB output
yes "`perl -e 'print "c\"x30_000\"'`` | head -c 1G
EOF
chmod +x mycommand

# Run 20 jobs in parallel
```
# Adjust 20 to be > physical RAM and < free space on /tmp
seq 20 | time $paralleltool . /mycommand | wc -c

## DISKFULL: Incomplete data if /tmp runs full

If caching is done on disk, the disk can run full during the run. Not all programs discover this. GNU Parallel discovers it, if it stays full for at least 2 seconds.

```
#!/bin/bash

paralleltool="parallel -j0"

# This should be a dir with less than 100 GB free space
smalldisk=/tmp/shm/parallel

TMPDIR="$smalldisk"
export TMPDIR

max_output() {
    # Force worst case scenario:
    # Make GNU Parallel only check once per second
    sleep 10
    # Generate 100 GB to fill $TMPDIR
    yes | head -c 100G>$TMPDIR/$$
    # Adjust if /tmp is bigger than 100 GB
    perl -e 'print "X"x10_000_000' | head -c 10M
    echo This part is missing from incomplete output
    sleep 2
    rm $TMPDIR/$$
    echo Final output
}

export -f max_output
seq 10 | $paralleltool max_output | tr -s X
```

## CLEANUP: Leaving tmp files at unexpected death

Some tools do not clean up tmp files if they are killed. If the tool buffers on disk, they may not clean up, if they are killed.

```
#!/bin/bash

paralleltool=parallel

ls /tmp >/tmp/before
seq 10 | $paralleltool sleep &
pid=$!
# Give the tool time to start up
sleep 1
# Kill it without giving it a chance to cleanup
kill -9 $$
# Should be empty: No files should be left behind
diff <(ls /tmp) /tmp/before
SPCCHAR: Dealing badly with special file names.
It is not uncommon for users to create files like:

My brother's 12" *** record (costs $$$.jpg

Some tools break on this.

#!/bin/bash

paralleltool=parallel

touch "My brother's 12" *** record (costs \$\$$.jpg"
ls My*.jpg | $paralleltool ls -1

COMPOSED: Composed commands do not work
Some tools require you to wrap composed commands into bash -c.

    echo bar | $paralleltool echo foo';' echo {}

ONEREP: Only one replacement string allowed
Some tools can only insert the argument once.

    echo bar | $paralleltool echo () foo {}

INPUTSIZE: Length of input should not be limited
Some tools limit the length of the input lines artificially with no good reason. GNU parallel does not:

perl -e 'print "foo."."x"x100_000_000' | parallel echo {}

GNU parallel limits the command to run to 128 KB due to execve(1):

perl -e 'print "x"x131_000' | parallel echo {} | wc

NUMWORDS: Speed depends on number of words
Some tools become very slow if output lines have many words.

#!/bin/bash

paralleltool=parallel

cat <<EOF > mycommand
#!/bin/bash

# 10 MB of lines with 1000 words
yes "`seq 1000`" | head -c 10M
EOF
chmod +x mycommand

# Run 30 jobs in parallel
seq 30 | time $paralleltool -j0 ./mycommand > /dev/null
### 4GB: Output with a line > 4GB should be OK

```
#!/bin/bash

paralleltool="parallel -j0"

cat <<-EOF > mycommand
#!/bin/bash
perl -e '
\$a=\"a\"x1000_000; for(1..5000) { print \$a }'
EOF
chmod +x mycommand

# Run 1 job
seq 1 | $paralleltool ./mycommand | LC_ALL=C wc
```

### AUTHOR

When using GNU parallel for a publication please cite:


This helps funding further development; and it won't cost you a cent. If you pay 10000 EUR you should feel free to use GNU Parallel without citing.

Copyright (C) 2007-10-18 Ole Tange, http://ole.tange.dk

Copyright (C) 2008-2010 Ole Tange, http://ole.tange.dk

Copyright (C) 2010-2020 Ole Tange, http://ole.tange.dk and Free Software Foundation, Inc.

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

### LICENSE

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or at your option any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <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**

find(1), xargs(1), make(1), pexec(1), ppss(1), xjobs(1), prll(1), dxargs(1), mdm(1)