GNU PARALLEL EXAMPLES

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 --best
```

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 --best ::: *.html
```

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

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

EXAMPLE: Running full commands in parallel

If there is no command given to GNU **parallel**, then the arguments are treated as a command line.

To run **gzip foo** and **bzip2 bar** in parallel run:

```
parallel ::: "gzip foo" "bzip2 bar"
```

or:

```
(echo "gzip foo"; echo "bzip2 bar") | parallel
```

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
```

In many shells you can also use **printf**:

```
printf '%s\0' *.log | parallel -0 -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-cpus 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 file names 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: Composed command with perl replacement string

You have a bunch of file. You want them sorted into dirs. The dir of each file should be named the first letter of the file name.

```
parallel 'mkdir -p {=s/(.).*/$1/=}; mv {} {=s/(.).*/$1/=}' ::: *
```

In practice you would probably not use a perl replacement string but instead **--match**:

```
parallel --match '(.)' 'mkdir -p {1.1} && mv {} {1.1}' ::: *
```

EXAMPLE: Composed command with multiple input sources

You have a dir with files named as 24 hours in 5 minute intervals: 00:00, 00:05, 00:10 .. 23:55. You want to find the files missing:

```
parallel [ -f {1}:{2} ] "||" echo {1}:{2} does not exist \
  ::: {00..23} ::: {00..55..5}
```

EXAMPLE: Match parts of input source

Match first initial and last name:

```
parallel --match '(.).* (.*)' echo {1.1}. {1.2} \
  ::: "Arthur Dent" "Ford Prefect" "Tricia McMillan" "Zaphod Beeblebrox"
```

Re-arrange (stupid) US date format into (nice) ISO-8601:

```
parallel --match '(.*)/(.*)/(.*)' echo {1.3}-{1.1:%02d}-{1.2:%02d} \
  ::: 12/31/1969 1/19/2038 6/1/2002
```

Match url into domain and path:

```
parallel --match 'https://(.*?)/(.*)' echo Domain: {1.1} Path: {1.2} \
  ::: https://example.com/dir/page https://gnu.org/s/parallel
```

Get URLs into dirs named by 2nd level domain name, e.g. https://www.gnu.org/s/parallel will be put into the dir gnu.org.

```
cat urls | parallel --match '//[^/]*?([^/.]+\.[^/.]+)/' \
  'mkdir -p {1.1} && cd {1.1} && wget {}'
```

Match host.domain:port from a log file:

```
cat log |
  parallel --match '\b([a-z0-9.]+):(\d+)\b' echo host:{1.1} port:{1.2}
```

Reorder comma-separated values:

```
parallel --match '(.*),(.*)' echo Second: {1.2}, First: {1.1} \
  ::: "Arthur,Babel fish" "Adams,Betelgeuse" "Arcturan,Bistro"
```

Capitalize word:

```
parallel --match '([a-z])([a-z]*) ([a-z])([a-z]*)' \
  echo '{=1.1 $_=uc($_) =}{1.2} {=1.3 $_=uc($_) =}{1.4}' \
  ::: "pan galactic" "gargle blaster"
```

Make an international dialing prefix table:

```
dial=(
  "DK(Denmark) 00,45"
  "US(United States) 011,1"
  "JP(Japan) 010,81"
  "AU(Australia) 0011,61"
  "CA(Canada) 011,1"
  "RU(Russia) 810,7"
  "TH(Thailand) 001,66"
  "TW(Taiwan) 002,886"
)
parallel --match '(.*)\((.*)\) (.*),(.*)' --match +1 \
  echo From {1.1}/{1.2} to {2.1}/{2.2} dial {1.3}-{2.4} \
  ::: "${dial[@]}" ::: "${dial[@]}"
```

Note how input source 2 reuses the **--match** from input source 1.

EXAMPLE: Replacement fields from CSV file with headers

This is an advanced example. You have:

```
Date;Name;Location
3/8/1978;"Beeblebrox; Zaphod";"Betelgeuse V"
10/12/1979;"Dent; Arthur";Earth
1/5/1981;Slartibartfast;Magrathea
```

You want:

```
Z. Beeblebrox: 1978-03-08/BET
A. Dent: 1979-10-12/EAR
```

```
Slartibartfast: 1981-01-05/MAG
```

Run:

```
parallel --csv --colsep ';' --header : --match "(\d+)/(\d+)/(\d+)" \
  --match "^([^;]+)(; (.))?" --match "(...)"   \
  echo '{=Name.3 s/(.)/$1. /;=}'{Name.1}: \
    {Date.3}-{Date.1:%02d}-{Date.2:%02d}/'{=Location.1 $_=uc =}' \
  :::: people.csv
```

--csv parses the input as CSV with **--colsep ;** as the separator - dealing correctly with quoted strings. The input is split into 3 columns. **--header :** makes the columns available as **{***columnname***}**. Each column has their corresponding **--match** so each field can be accessed as **{***columnname*.#**}**. **s/(.)/$1. /** is a perl expression that appends ". " if the name has an initial. **:%02d** formats single digits as two digits. **uc** upper cases the argument.

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
```

If your environment (aliases, variables, and functions) is small you can copy the full environment without having to **export -f** anything. See **env_parallel**.

EXAMPLE: Function tester

To test a program with different parameters:

```
tester() {
  if (eval "$@") >&/dev/null; then
    perl -e 'printf "\033[30;102m[ OK ]\033[0m @ARGV\n"' "$@"
  else
    perl -e 'printf "\033[30;101m[FAIL]\033[0m @ARGV\n"' "$@"
  fi
}
export -f tester
parallel tester my_program ::: arg1 arg2
parallel tester exit ::: 1 0 2 0
```

If **my_program** fails a red FAIL will be printed followed by the failing command; otherwise a green OK will be printed followed by the command.

EXAMPLE: Identify few failing jobs

--bar works best if jobs have no output. If the failing jobs have output you can identify the jobs like this:

```
job-with-few-failures() {
    # Force reproducibility
    RANDOM=$1
    # This fails 1% (328 of 32768)
    if [ $RANDOM -lt 328 ] ; then
      echo Failed $1
    fi
}
export -f job-with-few-failures
seq 1000 | parallel --bar --tag job-with-few-failures
```

EXAMPLE: Continously show the latest line of output

It can be useful to monitor the output of running jobs.

This shows the most recent output line until a job finishes. After which the output of the job is printed in full:

```
parallel '{} | tee >(cat >&3)' ::: 'command 1' 'command 2' \
  3> >(perl -ne '$|=1;chomp;printf"%.'$COLUMNS's\r",$_." "x100')
```

EXAMPLE: Log rotate

Log rotation renames a logfile to an extension with a higher number: log.1 becomes log.2, log.2 becomes log.3, and so on. The oldest log is removed. To avoid overwriting files the process starts backwards from the high number to the low number. This will keep 10 old versions of the log:

```
seq 9 -1 1 | parallel -j1 mv log.{} log.'{= $_++ =}'
mv log log.1
```

EXAMPLE: Simple network scanner

prips can generate IP-addresses from CIDR notation. With GNU **parallel** you can build a simple network scanner to see which addresses respond to **ping**:

```
prips 130.229.16.0/20 | \
  parallel --timeout 2 -j0 \
    'ping -c 1 {} >/dev/null && echo {}' 2>/dev/null
```

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 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: Replacing parts of file names

If you deal with paired end reads, you will have files like barcode1_R1.fq.gz, barcode1_R2.fq.gz, barcode2_R1.fq.gz, and barcode2_R2.fq.gz.

You want barcode*N*_R1 to be processed with barcode*N*_R2.

```
  parallel --plus myprocess {} {/_R1.fq.gz/_R2.fq.gz} ::: *_R1.fq.gz
```

If the barcode does not contain '_R1', you can do:

```
  parallel --plus myprocess {} {/_R1/_R2} ::: *_R1.fq.gz
```

Or you can use **--match**:

```
  parallel --match '(.*)_R1(.*)' myprocess {} {1.1}_R2{1.2} :::
*_R1.fq.gz
```

EXAMPLE: Removing strings from the argument

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:

```
parallel --plus 'mkdir {..}; tar -C {..} -xf {}' ::: *.tar.gz
```

If you want to remove a different ending, you can use {%string}:

```
parallel --plus echo {%_demo} ::: mycode_demo keep_demo_here
```

You can also remove a starting string with {#string}

```
parallel --plus echo {#demo_} ::: demo_mycode keep_demo_here
```

To remove a string anywhere you can use regular expressions with {/regexp/replacement} and leave the replacement empty:

```
parallel --plus echo {/demo_/} ::: demo_mycode remove_demo_here
```

You can often also use **--match**:

```
parallel --match '(.*)demo_(.*)' echo {1.1}{1.2} ::: demo_mycode
remove_demo_here
```

EXAMPLE: Download 24 images for each of the past 30 days

Let us assume a website stores images like:

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

where YYYYMMDD is the date and ## is the number 01-24. This will download images for the past 30 days:

```
getit() {
  date=$(date -d "today -$1 days" +%Y%m%d)
  num=$2
```

```
  echo wget https://www.example.com/path/to/${date}_${num}.jpg
}
export -f getit

parallel getit ::: $(seq 30) ::: $(seq -w 24)
```

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

EXAMPLE: Download world map from NASA

NASA provides tiles to download on earthdata.nasa.gov. Download tiles for Blue Marble world map and create a 10240x20480 map.

```
base=https://map1a.vis.earthdata.nasa.gov/wmts-geo/wmts.cgi
service="SERVICE=WMTS&REQUEST=GetTile&VERSION=1.0.0"
layer="LAYER=BlueMarble_ShadedRelief_Bathymetry"
set="STYLE=&TILEMATRIXSET=EPSG4326_500m&TILEMATRIX=5"
tile="TILEROW={1}&TILECOL={2}"
format="FORMAT=image%2Fjpeg"
url="$base?$service&$layer&$set&$tile&$format"

parallel -j0 -q wget "$url" -O {1}_{2}.jpg ::: {0..19} ::: {0..39}
parallel eval convert +append {}_{0..39}.jpg line{}.jpg ::: {0..19}
convert -append line{0..19}.jpg world.jpg
```

EXAMPLE: Download Apollo-11 images from NASA using jq

Search NASA using their API to get JSON for images related to 'apollo 11' and has 'moon landing' in the description.

The search query returns JSON containing URLs to JSON containing collections of pictures. One of the pictures in each of these collection is *large*.

wget is used to get the JSON for the search query. **jq** is then used to extract the URLs of the collections. **parallel** then calls **wget** to get each collection, which is passed to **jq** to extract the URLs of all images. **grep** filters out the *large* images, and **parallel** finally uses **wget** to fetch the images.

```
base="https://images-api.nasa.gov/search"
q="q=apollo 11"
description="description=moon landing"
media_type="media_type=image"
wget -O - "$base?$q&$description&$media_type" |
  jq -r .collection.items[].href |
  parallel wget -O - |
  jq -r .[] |
  grep large |
  parallel wget
```

EXAMPLE: Download video playlist in parallel

youtube-dl is an excellent tool to download videos. It can, however, not download videos in parallel. This takes a playlist and downloads 10 videos in parallel.

```
url='youtu.be/watch?v=0wOf2Fgi3DE&list=UU_cznB5YZZmvAmeq7Y3EriQ'
export url
youtube-dl --flat-playlist "https://$url" |
  parallel --tagstring {#} --lb -j10 \
    youtube-dl --playlist-start {#} --playlist-end {#} '"https://$url"'
```

EXAMPLE: Prepend last modified date (ISO8601) to file name

```
parallel mv {} '{= $a=pQ($_); $b=$_;' \
  '$_=qx{date -r "$a" +%FT%T}; chomp; $_="$_ $b" =}' ::: *
```

{= and **=}** mark a perl expression. **pQ** perl-quotes the string. **date +%FT%T** is the date in ISO8601 with time.

EXAMPLE: Save output in ISO8601 dirs

Save output from **ps aux** every second into dirs named yyyy-mm-ddThh:mm:ss+zz:zz.

```
seq 1000 | parallel -N0 -j1 --delay 1 \
  --results '{= $_=`date -Isec`; chomp=}/' ps aux
```

EXAMPLE: Digital clock with "blinking" :

The : in a digital clock blinks. To make every other line have a ':' and the rest a ' ' a perl expression is used to look at the 3rd input source. If the value modulo 2 is 1: Use ":" otherwise use " ":

```
parallel -k echo {1}'{=3 $_=$_%2?":":" "=}'{2}{3} \
  ::: {0..23} ::: {0..5} ::: {0..9}
```

EXAMPLE: Aggregating content of files

This:

```
parallel --header : echo x{X}y{Y}z{Z} \> x{X}y{Y}z{Z} \
::: X {1..5} ::: Y {01..10} ::: Z {1..5}
```

will generate the files x1y01z1 .. x5y10z5. If you want to aggregate the output grouping on x and z you can do this:

```
parallel eval 'cat {=s/y01/y*/=} > {=s/y01//=}' ::: *y01*
```

For all values of x and z it runs commands like:

```
cat x1y*z1 > x1z1
```

So you end up with x1z1 .. x5z5 each containing the content of all values of y.

EXAMPLE: Breadth first parallel dir crawler

To process all files in dirs and subdirs you would normally run:

```
find . -print | parallel do_stuff
```

But sometimes you want to parallelize each dir. Maybe doing a dir scan is slow?

Then you can use a breadth first directory scan.

```
process() {
    process_file() {
        echo "Do your processing of file here $1"
    }

    queue="$1"
    shift
    if  [ -d "$1" ] ; then
        echo "queueing $1"
        find "$1" -mindepth 1 -maxdepth 1 > "$queue"
```

```
        if [ ! -s "$queue" ] ; then
            # Ignore empty dirs
            rm "$queue"
        fi
    else
        process_file "$1"
    fi
}
export -f process

# Queue lists
queue=$(mktemp)
queuenew=$(mktemp -d)

# Start dir
echo . > "$queue"

while [ -s "$queue" ] ; do
    # Run one round for every directory level
    # (Breadth first)
    cat "$queue" |
        parallel process "$queuenew"/{#} {}
    # Each job may create a list in "$queuenew"/job_no
    cat "$queuenew"/* > "$queue" 2>/dev/null
    rm -f "$queuenew"/*
done
rmdir "$queuenew"
rm "$queue"
```

This is not a perfect replacement (e.g. **--halt** is not respected, and **$?** is not set correctly).

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 https://freenet.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. https://freenet.org/
url=$1
# Stay inside the start dir
baseurl=$(echo "$url" | perl -pe 's:#.*::; s:(//.*/)[^/]*:$1:')
urllist=$(mktemp -t urllist.XXXX)
newurllist=$(mktemp -t urllist.XXXX)
seen=$(mktemp -t seen.XXXX)
```

```
  # Add start url to the list
  echo "$url" >"$urllist"
  cp "$urllist" "$seen"

  process_single_url() {
      # Find all links in the url
      lynx -listonly -image_links -dump "$1"
      # Remove this line to get spider only
      wget -qm -l1 -Q1 "$1"
      echo
      echo Spidered: "$1" >&2
  }
  export -f process_single_url

  unique() {
      # Like `sort -u` but without the sorting
      perl -ne 's/#.*//; s/\s+\d+.\s(\S+)$/$1/ and do { $seen{$1}++ or
print }'
  }

  while [ -s "$urllist" ] ; do
    cat "$urllist" |
      parallel process_single_url |
      unique |
      # Ignore links outside $baseurl
      grep -a -F "$baseurl" |
      # Ignore links already seen
      grep -a -v -x -F -f "$seen" |
      tee -a "$seen" > "$newurllist"
    mv "$newurllist" "$urllist"
  done

  rm -f "$newurllist" "$urllist" "$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 make sure the file is complete before handing it to GNU **parallel**.

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_step1 $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_step1 $x scale $no_extension.jpg
  do_step2 <$x $no_extension
done) | process_output
```

can be written like this:

```
cat list | parallel "do_step1 {} scale {.}.jpg ; do_step2 <{} {.}" |\
  process_output
```

If the body of the loop is bigger, it improves readability to use a function:

```
(for x in `cat list` ; do
  do_something $x
  [... 100 lines that do something with $x ...]
done) | process_output

cat list | (while read x ; do
  do_something $x
  [... 100 lines that do something with $x ...]
done) | process_output
```

can both be rewritten as:

```
doit() {
  x=$1
  do_something $x
  [... 100 lines that do something with $x ...]
}
export -f doit
cat list | parallel doit
```

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 colour in red green blue ; do
    for size in S M L XL XXL ; do
      echo $colour $size
    done
  done) | sort
```

can be written like this:

```
  parallel echo {1} {2} ::: red green blue ::: 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 {colour} {size} \
    ::: colour red green blue ::: 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: All combinations in a list

GNU **parallel** makes all combinations when given two lists.

To make all combinations in a single list with unique values, you repeat the list and use replacement string **{choose_k}**:

```
  parallel --plus echo {choose_k} ::: A B C D ::: A B C D

  parallel --plus echo 2{2choose_k} 1{1choose_k} ::: A B C D ::: A B C D
```

{choose_k} works for any number of input sources:

```
  parallel --plus echo {choose_k} ::: A B C D ::: A B C D ::: A B C D
```

Where **{choose_k}** does not care about order, **{uniq}** cares about order. It simply skips jobs where values from different input sources are the same:

```
parallel --plus echo {uniq} ::: A B C  ::: A B C  ::: A B C
parallel --plus echo {1uniq}+{2uniq}+{3uniq} \
  ::: A B C  ::: A B C  ::: A B C
```

The behaviour of **{choose_k}** is undefined, if the input values of each source are different.

EXAMPLE: From a to b and b to c

Assume you have input like:

```
aardvark
babble
cab
dab
each
```

and want to run combinations like:

```
aardvark babble
babble cab
cab dab
dab each
```

If the input is in the file in.txt:

```
parallel echo {1} - {2} ::::+ <(head -n -1 in.txt) <(tail -n +2 in.txt)
```

If the input is in the array $a here are two solutions:

```
seq $((${#a[@]}-1)) | \
  env_parallel --env a echo '${a[{=$_--=}]} - ${a[{}]}'
parallel echo {1} - {2} ::: "${a[@]::${#a[@]}-1}" :::+ "${a[@]:1}"
```

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'.

EXAMPLE: Speeding up fast jobs

Starting a job on the local machine takes around 3-10 ms. This can be a big overhead if the job takes very few ms to run. Often you can group small jobs together using **-X** which will make the overhead less significant. 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 9999999 | \
  parallel -j10 -q -I,, --pipe parallel -j0 touch pict{}.jpg
```

If **-j0** normally spawns 252 jobs, then the above will try to spawn 2520 jobs. On a normal GNU/Linux system you can spawn 32000 jobs using this technique with no problems. To raise the 32000 jobs

limit raise /proc/sys/kernel/pid_max to 4194303.

If you do not need GNU **parallel** to have control over each job (so no need for **--retries** or **--joblog** or similar), then it can be even faster if you can generate the command lines and pipe those to a shell. So if you can do this:

```
mygenerator | sh
```

Then that can be parallelized like this:

```
mygenerator | parallel --pipe --block 10M sh
```

E.g.

```
mygenerator() {
  seq 10000000 | perl -pe 'print "echo This is fast job number "';
}
mygenerator | parallel --pipe --block 10M sh
```

The overhead is 100000 times smaller namely around 100 nanoseconds per job.

EXAMPLE: Using shell variables

When using shell variables you need to quote them correctly as they may otherwise be interpreted by the shell.

Notice the difference between:

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

and:

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

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**:

```
VAR="My brother's 12\" records are worth <\$\$\$>"
parallel -q echo "$VAR" ::: '!'
export VAR
parallel echo '"$VAR"' ::: '!'
```

If **$VAR** does not contain ' then **"'$VAR'"** will also work (and does not need **export**):

```
VAR="My 12\" records are worth <\$\$\$>"
parallel echo "'$VAR'" ::: '!'
```

If you use them in a function you just quote as you normally would do:

```
VAR="My brother's 12\" records are worth <\$\$\$>"
export VAR
myfunc() { echo "$VAR" "$1"; }
export -f myfunc
parallel myfunc ::: '!'
```

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 full lines to be printed while the job is running you can use **--line-buffer**. If you want output to be printed as soon as possible you can use **-u**.

Compare the output of:

```
parallel wget --progress=dot --limit-rate=100k \
  https://ftpmirror.gnu.org/parallel/parallel-20{}0822.tar.bz2 \
  ::: {12..16}
parallel --line-buffer wget --progress=dot --limit-rate=100k \
  https://ftpmirror.gnu.org/parallel/parallel-20{}0822.tar.bz2 \
  ::: {12..16}
parallel --latest-line wget --progress=dot --limit-rate=100k \
  https://ftpmirror.gnu.org/parallel/parallel-20{}0822.tar.bz2 \
  ::: {12..16}
parallel -u wget --progress=dot --limit-rate=100k \
  https://ftpmirror.gnu.org/parallel/parallel-20{}0822.tar.bz2 \
  ::: {12..16}
```

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 wget --limit-rate=100k \
  https://ftpmirror.gnu.org/parallel/parallel-20{}0822.tar.bz2 \
  ::: {12..16}
```

--tag works with **--line-buffer** but not with **-u**:

```
parallel --tag --line-buffer wget --limit-rate=100k \
  https://ftpmirror.gnu.org/parallel/parallel-20{}0822.tar.bz2 \
  ::: {12..16}
```

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

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

EXAMPLE: Colorize output

Give each job a new color. Most terminals support ANSI colors with the escape code "\033[30;3Xm" where 0 <= X <= 7:

```
seq 10 | \
  parallel --tagstring '\033[30;3{=$_=++$::color%8=}m' seq {}
parallel --rpl '{color} $_="\033[30;3".(++$::color%8)."m"' \
  --tagstring {color} seq {} ::: {1..10}
```

To get rid of the initial \t (which comes from **--tagstring**):

```
... | perl -pe 's/\t//'
```

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 ::: qubes-os.org debian.org freenetproject.org
```

will give traceroute of qubes-os.org, 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 ::: qubes-os.org debian.org freenetproject.org
```

This will make sure the traceroute to qubes-os.org 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 https://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 https://example.com/the/big/file > file
```

EXAMPLE: Keep order, but make job 1 output fast

If you want the output of job 1 unbuffered, but otherwise keep the order, you can do this:

```
doit() {
  echo "$@" ERR >&2
  echo "$@" out
  sleep 0.$1
  echo "$@" ERR >&2
  echo "$@" out
}
export -f doit
parallel -k -u doit {= 'seq() > 1 and $opt::ungroup = 0' =} ::: 9 1 2 3
```

It will output job 1 with less overhead.

EXAMPLE: Parallel grep

grep -r greps recursively through directories. 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 CPU, and give 1000 arguments to **grep**.

There are situations where the above will be slower than **grep -r**:

- If data is already in RAM. The overhead of starting jobs and buffering output may outweigh the benefit of running in parallel.
- If the files are big. If a file cannot be read in a single seek, the disk may start thrashing.

The speedup is caused by two factors:

- On rotating harddisks small files often require a seek for each file. By searching for more files in parallel, the arm may pass another wanted file on its way.
- NVMe drives often perform better by having multiple command running in parallel.

EXAMPLE: Grepping n lines for m regular expressions.

The simplest solution to grep a big file for a lot of regexps is:

```
grep -f regexps.txt bigfile
```

Or if the regexps are fixed strings:

```
grep -F -f regexps.txt bigfile
```

There are 3 limiting factors: CPU, RAM, and disk I/O.

RAM is easy to measure: If the **grep** process takes up most of your free memory (e.g. when running **top**), then RAM is a limiting factor.

CPU is also easy to measure: If the **grep** takes >90% CPU in **top**, then the CPU is a limiting factor, and parallelization will speed this up.

It is harder to see if disk I/O is the limiting factor, and depending on the disk system it may be faster or slower to parallelize. The only way to know for certain is to test and measure.

Limiting factor: RAM

The normal **grep -f regexps.txt bigfile** works no matter the size of bigfile, but if regexps.txt is so big it cannot fit into memory, then you need to split this.

grep -F takes around 100 bytes of RAM and **grep** takes about 500 bytes of RAM per 1 byte of regexp. So if regexps.txt is 1% of your RAM, then it may be too big.

If you can convert your regexps into fixed strings do that. E.g. if the lines you are looking for in bigfile all looks like:

```
ID1 foo bar baz Identifier1 quux
fubar ID2 foo bar baz Identifier2
```

then your regexps.txt can be converted from:

```
ID1.*Identifier1
ID2.*Identifier2
```

into:

```
ID1 foo bar baz Identifier1
ID2 foo bar baz Identifier2
```

This way you can use **grep -F** which takes around 80% less memory and is much faster.

If it still does not fit in memory you can do this:

```
parallel --pipe-part -a regexps.txt --block 1M grep -F -f - -n bigfile |
\
```

```
  sort -un | perl -pe 's/^\d+://'
```

The 1M should be your free memory divided by the number of CPU threads and divided by 200 for **grep -F** and by 1000 for normal **grep**. On GNU/Linux you can do:

```
free=$(awk '/^((Swap)?Cached|MemFree|Buffers):/ { sum += $2 }
            END { print sum }' /proc/meminfo)
percpu=$((free / 200 / $(parallel --number-of-threads)))k

parallel --pipe-part -a regexps.txt --block $percpu --compress \
  grep -F -f - -n bigfile | \
  sort -un | perl -pe 's/^\d+://'
```

If you can live with duplicated lines and wrong order, it is faster to do:

```
parallel --pipe-part -a regexps.txt --block $percpu --compress \
  grep -F -f - bigfile
```

Limiting factor: CPU

If the CPU is the limiting factor parallelization should be done on the regexps:

```
cat regexps.txt | parallel --pipe -L1000 --round-robin --compress \
  grep -f - -n bigfile | \
  sort -un | perl -pe 's/^\d+://'
```

The command will start one **grep** per CPU and read *bigfile* one time per CPU, but as that is done in parallel, all reads except the first will be cached in RAM. Depending on the size of *regexps.txt* it may be faster to use **--block 10m** instead of **-L1000**.

Some storage systems perform better when reading multiple chunks in parallel. This is true for some RAID systems and for some network file systems. To parallelize the reading of *bigfile*:

```
parallel --pipe-part --block 100M -a bigfile -k --compress \
  grep -f regexps.txt
```

This will split *bigfile* into 100MB chunks and run **grep** on each of these chunks. To parallelize both reading of *bigfile* and *regexps.txt* combine the two using **--cat**:

```
parallel --pipe-part --block 100M -a bigfile --cat cat regexps.txt \
  \| parallel --pipe -L1000 --round-robin grep -f - {}
```

If a line matches multiple regexps, the line may be duplicated.

Bigger problem

If the problem is too big to be solved by this, you are probably ready for Lucene.

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**, **ssh-agent**, and **sshpass** 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 s1.example.com,s2.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
```

If your list of hosts is *server1-88.example.net* with login *foo*:

```
seq 10 | parallel -Sfoo@server{1..88}.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 CPUs on each of the remote computers, and run one job per CPU - even if the remote computers do not have the same number of CPUs.

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

```
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: Advanced file transfer

Assume you have files in in/*, want them processed on server, and transferred back into /other/dir:

```
parallel -S server --trc /other/dir/./{/}.out \
  cp {/} {/}.out ::: in/./*
```

EXAMPLE: Distributing work to local and remote computers

Convert *.mp3 to *.ogg running one process per CPU 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 to 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: Running 'sudo' on remote computers

Put the password into passwordfile then run:

```
parallel --ssh 'cat passwordfile | ssh' --nonall \
  -S user@server1,user@server2 sudo -S ls -l /root
```

EXAMPLE: Using remote computers behind NAT wall

If the workers are behind a NAT wall, you need some trickery to get to them.

If you can **ssh** to a jumphost, and reach the workers from there, then the obvious solution would be this, but it **does not work**:

```
parallel --ssh 'ssh jumphost ssh' -S host1 echo ::: DOES NOT WORK
```

It does not work because the command is dequoted by **ssh** twice where as GNU **parallel** only expects it to be dequoted once.

You can use a bash function and have GNU **parallel** quote the command:

```
jumpssh() { ssh -A jumphost ssh $(parallel --shellquote ::: "$@"); }
export -f jumpssh
parallel --ssh jumpssh -S host1 echo ::: this works
```

Or you can instead put this in **~/.ssh/config**:

```
Host host1 host2 host3
  ProxyCommand ssh jumphost.domain nc -w 1 %h 22
```

It requires **nc(netcat)** to be installed on jumphost. With this you can simply:

```
parallel -S host1,host2,host3 echo ::: This does work
```

No jumphost, but port forwards

If there is no jumphost but each server has port 22 forwarded from the firewall (e.g. the firewall's port 22001 = port 22 on host1, 22002 = host2, 22003 = host3) then you can use **~/.ssh/config**:

```
Host host1.v
  Port 22001
Host host2.v
  Port 22002
Host host3.v
  Port 22003
Host *.v
  Hostname firewall
```

And then use host{1..3}.v as normal hosts:

```
parallel -S host1.v,host2.v,host3.v echo ::: a b c
```

No jumphost, no port forwards

If ports cannot be forwarded, you need some sort of VPN to traverse the NAT-wall. TOR is one options for that, as it is very easy to get working.

You need to install TOR and setup a hidden service. In **torrc** put:

```
HiddenServiceDir /var/lib/tor/hidden_service/
HiddenServicePort 22 127.0.0.1:22
```

Then start TOR: **/etc/init.d/tor restart**

The TOR hostname is now in **/var/lib/tor/hidden_service/hostname** and is something similar to **izjafdceobowklhz.onion**. Now you simply prepend **torsocks** to **ssh**:

```
parallel --ssh 'torsocks ssh' -S izjafdceobowklhz.onion \
  -S zfcdaeiojoklbwhz.onion,auclucjzobowklhi.onion echo ::: a b c
```

If not all hosts are accessible through TOR:

```
parallel -S 'torsocks ssh izjafdceobowklhz.onion,host2,host3' \
  echo ::: a b c
```

See more **ssh** tricks on https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Proxies_and_Jump_Hosts

EXAMPLE: Use sshpass with ssh

If you cannot use passwordless login, you may be able to use **sshpass**:

```
seq 10 | parallel -S user-with-password:MyPassword@server echo
```

or:

```
export SSHPASS='MyPa$$w0rd'
seq 10 | parallel -S user-with-password:@server echo
```

EXAMPLE: Use outrun instead of ssh

outrun lets you run a command on a remote server. **outrun** sets up a connection to access files at the source server, and automatically transfers files. **outrun** must be installed on the remote system.

You can use **outrun** in an sshlogin this way:

```
parallel -S 'outrun user@server' command
```

or:

```
parallel --ssh outrun -S server command
```

EXAMPLE: Slurm cluster

The Slurm Workload Manager is used in many clusters.

Here is a simple example of using GNU **parallel** to call **srun**:

```
#!/bin/bash

#SBATCH --time 00:02:00
#SBATCH --ntasks=4
#SBATCH --job-name GnuParallelDemo
#SBATCH --output gnuparallel.out

module purge
module load gnu_parallel

my_parallel="parallel --delay .2 -j $SLURM_NTASKS"
my_srun="srun --export=all --exclusive -n1"
my_srun="$my_srun --cpus-per-task=1 --cpu-bind=cores"
$my_parallel "$my_srun" echo This is job {} ::: {1..20}
```

EXAMPLE: Parallelizing rsync

rsync is a great tool, but sometimes it will not fill up the available bandwidth. Running multiple **rsync** in parallel can fix this.

```
cd src-dir
find . -type f |
  parallel -j10 -X rsync -zR -Ha ./{} fooserver:/dest-dir/
```

Adjust **-j10** until you find the optimal number.

rsync -R will create the needed subdirectories, so all files are not put into a single dir. The **./** is needed so the resulting command looks similar to:

```
rsync -zR ././sub/dir/file fooserver:/dest-dir/
```

The **/./** is what **rsync -R** works on.

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/*{}.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 --link -a <(seq 6) -a <(seq 6 -1 1) echo
parallel --link 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 --link -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: Output to database

GNU **parallel** can output to a database table and a CSV-file:

```
dburl=csv:///%2Ftmp%2Fmydir
dbtableurl=$dburl/mytable.csv
parallel --sqlandworker $dbtableurl seq ::: {1..10}
```

It is rather slow and takes up a lot of CPU time because GNU **parallel** parses the whole CSV file for each update.

A better approach is to use an SQLite-base and then convert that to CSV:

```
dburl=sqlite3:///%2Ftmp%2Fmy.sqlite
dbtableurl=$dburl/mytable
parallel --sqlandworker $dbtableurl seq ::: {1..10}
sql $dburl '.headers on' '.mode csv' 'SELECT * FROM mytable;'
```

This takes around a second per job.

If you have access to a real database system, such as PostgreSQL, it is even faster:

```
dburl=pg://user:pass@host/mydb
dbtableurl=$dburl/mytable
parallel --sqlandworker $dbtableurl seq ::: {1..10}
sql $dburl \
  "COPY (SELECT * FROM mytable) TO stdout DELIMITER ',' CSV HEADER;"
```

Or MySQL:

```
dburl=mysql://user:pass@host/mydb
dbtableurl=$dburl/mytable
parallel --sqlandworker $dbtableurl seq ::: {1..10}
sql -p -B $dburl "SELECT * FROM mytable;" > mytable.tsv
perl -pe 's/"/""/g; s/\t/","/g; s/^/"/; s/$/"/;
  %s=("\\" => "\\", "t" => "\t", "n" => "\n");
  s/\\([\\tn])/$s{$1}/g;' mytable.tsv
```

EXAMPLE: Output to CSV-file for R

If you have no need for the advanced job distribution control that a database provides, but you simply want output into a CSV file that you can read into R or LibreCalc, then you can use **--results**:

```
parallel --results my.csv seq ::: 10 20 30
R
> mydf <- read.csv("my.csv");
> print(mydf[2,])
> write(as.character(mydf[2,c("Stdout")]),'')
```

EXAMPLE: Use XML as input

The show Aflyttet on Radio 24syv publishes an RSS feed with their audio podcasts on: http://arkiv.radio24syv.dk/audiopodcast/channel/4466232

Using **xpath** you can extract the URLs for 2019 and download them using GNU **parallel**:

```
wget -O - http://arkiv.radio24syv.dk/audiopodcast/channel/4466232 | \
  xpath -e "//pubDate[contains(text(),'2019')]/../enclosure/@url" | \
  parallel -u wget '{= s/ url="//; s/"//; =}'
```

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: Call program with FASTA sequence

FASTA files have the format:

```
>Sequence name1
sequence
sequence continued
>Sequence name2
sequence
sequence continued
more sequence
```

To call **myprog** with the sequence as argument run:

```
cat file.fasta |
  parallel --pipe -N1 --recstart '>' --rrs \
    'read a; echo Name: "$a"; myprog $(tr -d "\n")'
```

EXAMPLE: Call program with interleaved FASTQ records

FASTQ files have the format:

```
@M10991:61:000000000-A7EML:1:1101:14011:1001 1:N:0:28
CTCCTAGGTCGGCATGATGGGGGAAGGAGAGCATGGGAAGAAATGAGAGAGTAGCAAGG
+
#8BCCGGGGGFEFECFGGGGGGGGG@;FFGGGEG@FF<EE<@FFC,CEGCCGGFF<FGF
```

Interleaved FASTQ starts with a line like these:

```
@HWUSI-EAS100R:6:73:941:1973#0/1
@EAS139:136:FC706VJ:2:2104:15343:197393 1:Y:18:ATCACG
@EAS139:136:FC706VJ:2:2104:15343:197393 1:N:18:1
```

where '/1' and ' 1:' determines this is read 1.

This will cut big.fq into one chunk per CPU thread and pass it on stdin (standard input) to the program fastq-reader:

```
parallel --pipe-part -a big.fq --block -1 --regexp \
  --recend '\n' --recstart '@.*(/1| 1:.*)\n[A-Za-z\n\.~]' \
  fastq-reader
```

EXAMPLE: Processing a big file using more CPUs

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. 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**.

GNU **parallel**'s **--pipe** maxes out at around 100 MB/s because every byte has to be copied through GNU **parallel**. But if **bigfile** is a real (seekable) file GNU **parallel** can by-pass the copying and send the parts directly to the program:

```
parallel --pipe-part --block 100m -a bigfile --files sort |\
  parallel -Xj1 sort -m {} ';' rm {} >bigfile.sort
```

EXAMPLE: Grouping input lines

When processing with **--pipe** you may have lines grouped by a value. Here is *my.csv*:

```
  Transaction Customer Item
1 a 53
2 b 65
3 b 82
4 c 96
5 c 67
6 c 13
7 d 90
8 d 43
9 d 91
10 d 84
11 e 72
12 e 102
13 e 63
14 e 56
15 e 74
```

Let us assume you want GNU **parallel** to process each customer. In other words: You want all the transactions for a single customer to be treated as a single record.

To do this we preprocess the data with a program that inserts a record separator before each customer (column 2 = $F[1]). Here we first make a 50 character random string, which we then use as the separator:

```
sep=`perl -e 'print map { ("a".."z","A"..."Z")[rand(52)] } (1..50);'`
```

```
cat my.csv | \
   perl -ape '$F[1] ne $l and print "'$sep'"; $l = $F[1]' | \
   parallel --recend $sep --rrs --pipe -N1 wc
```

If your program can process multiple customers replace **-N1** with a reasonable **--blocksize**.

EXAMPLE: Running more than 250 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 250 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 250 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 62500 jobs (use with caution - you need 64 GB RAM to do this, and you may need to increase /proc/sys/kernel/pid_max):

```
cat myinput |\
  parallel --pipe -N 250 --round-robin -j250 parallel -j250 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 '1i{}' 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 '1i{}' myfile
```

EXAMPLE: Mutex for a script

Assume a script is called from cron or from a web service, but only one instance can be run at a time. With **sem** and **--shebang-wrap** the script can be made to wait for other instances to finish. Here in **bash**:

```
#!/usr/bin/sem --shebang-wrap -u --id $0 --fg /bin/bash

echo This will run
sleep 5
echo exclusively
```

Here **perl**:

```
#!/usr/bin/sem --shebang-wrap -u --id $0 --fg /usr/bin/perl

print "This will run ";
sleep 5;
print "exclusively\n";
```

Here **python**:

```
#!/usr/local/bin/sem --shebang-wrap -u --id $0 --fg /usr/bin/python

import time
print "This will run ";
time.sleep(5)
print "exclusively";
```

EXAMPLE: Start editor with file names 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. Instead do:

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

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

EXAMPLE: Run ping in parallel

ping prints out statistics when killed with CTRL-C.

Unfortunately, CTRL-C will also normally kill GNU **parallel**.

But by using **--open-tty** and ignoring SIGINT you can get the wanted effect:

```
parallel -j0 --open-tty --lb --tag ping '{= $SIG{INT}=sub {} =}' \
  ::: 1.1.1.1 8.8.8.8 9.9.9.9 21.21.21.21 80.80.80.80 88.88.88.88
```

--open-tty will make the **ping**s receive SIGINT (from CTRL-C). CTRL-C will not kill GNU **parallel**, so that will only exit after **ping** is done.

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 -n+0 -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:

```
true >jobqueue; tail -n+0 -f jobqueue | parallel -S ..
```

Output only will be printed when reading the next input after a job has finished: So you need to submit a job after the first has finished to see the output from the first job.

If you keep this running for a long time, jobqueue will grow. A way of removing the jobs already run is by making GNU **parallel** stop when it hits a special value and then restart. To use **--eof** to make GNU **parallel** exit, **tail** also needs to be forced to exit:

```
true >jobqueue;
while true; do
  tail -n+0 -f jobqueue |
    (parallel -E StOpHeRe -S ..; echo GNU Parallel is now done;
     perl -e 'while(<>){/StOpHeRe/ and last};print <>' jobqueue > j2;
     (seq 1000 >> jobqueue &);
     echo Done appending dummy data forcing tail to exit)
  echo tail exited;
  mv j2 jobqueue
done
```

In some cases you can run on more CPUs and computers during the night:

```
# Day time
echo 50% > jobfile
cp day_server_list ~/.parallel/sshloginfile
# Night time
echo 100% > jobfile
cp night_server_list ~/.parallel/sshloginfile
tail -n+0 -f jobqueue | parallel --jobs jobfile -S ..
```

GNU **parallel** discovers if **jobfile** or **~/.parallel/sshloginfile** changes.

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 -qmre 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 -qmre 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.

EXAMPLE: Locate the missing package

If you have downloaded source and tried compiling it, you may have seen:

```
$ ./configure
[...]
checking for something.h... no
configure: error: "libsomething not found"
```

Often it is not obvious which package you should install to get that file. Debian has `apt-file` to search for a file. `tracefile` from https://codeberg.org/tange/tangetools can tell which files a program tried to access. In this case we are interested in one of the last files:

```
$ tracefile -un ./configure | tail | parallel -j0 apt-file search
```

AUTHOR

When using GNU **parallel** for a publication please cite:

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

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

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

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

Copyright (C) 2010-2026 Ole Tange, http://ole.tange.dk and Free Software Foundation, Inc.

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

LICENSE

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

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

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

Documentation license I

Permission is granted to copy, distribute and/or modify this documentation under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the file LICENSES/GFDL-1.3-or-later.txt.

Documentation license II

You are free:

to Share

to copy, distribute and transmit the work

to Remix

to adapt the work

Under the following conditions:

Attribution

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike

If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

With the understanding that:

Waiver

Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain

Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights

In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice

For any reuse or distribution, you must make clear to others the license terms of this work.

A copy of the full license is included in the file as LICENCES/CC-BY-SA-4.0.txt

SEE ALSO

parallel(1), **parallel_tutorial**(7), **env_parallel**(1), **parset**(1), **parsort**(1), **parallel_alternatives**(7), **parallel_design**(7), **niceload**(1), **sql**(1), **ssh**(1), **ssh-agent**(1), **sshpass**(1), **ssh-copy-id**(1), **rsync**(1)