Why should you read this book?

If you write shell scripts to do the same processing for different input, then GNU **parallel** will make your life easier and make your scripts run faster.

The book is written so you get the juicy parts first: The goal is that you read just enough to get you going. GNU **parallel** has an overwhelming amount of special features to help in different situations, and to avoid overloading you with information, the most used features are presented first.

All the examples are tested in Bash, and most will work in other shells, too, but there are a few exceptions. So you are recommended to use Bash while testing out the examples.

Learn GNU Parallel in 5 minutes

You just need to run commands in parallel. You do not care about fine tuning.

To get going please run this to make some example files:

```
# If your system does not have 'seq', replace 'seq' with 'jot'
seq 5 | parallel seq {} '>' example.{}
```

Input sources

GNU **parallel** reads values from input sources. One input source is the command line. The values are put after **:::** :

```
parallel echo ::: 1 2 3 4 5
```

This makes it easy to run the same program on some files:

```
parallel wc ::: example.*
```

If you give multiple **:::**s, GNU **parallel** will generate all combinations:

```
parallel wc ::: -l -c ::: example.*
```

GNU **parallel** can also read the values from stdin (standard input):

```
seq 5 | parallel echo
```

Building the command line

The command line is put before the **:::**. It can contain contain a command and options for the command:

```
parallel wc -l ::: example.*
```

The command can contain multiple programs. Just remember to quote characters that are interpreted by the shell (such as **;**):

```
parallel echo counting lines';' wc -l ::: example.*
```

The value will normally be appended to the command, but can be placed anywhere by using the replacement string **{}**:

```
parallel echo counting {}';' wc -l {} ::: example.*
```

When using multiple input sources you use the positional replacement strings **{1}** and **{2}**:

```
parallel echo count {1} in {2}';' wc {1} {2} ::: -l -c ::: example.*
```

You can check what will be run with **--dry-run**:

```
  parallel --dry-run echo count {1} in {2}';' wc {1} {2} ::: -l -c :::
example.*
```

This is a good idea to do for every command until you are comfortable with GNU **parallel**.

Controlling the output

The output will be printed as soon as the command completes. This means the output may come in a different order than the input:

```
  parallel sleep {}';' echo {} done ::: 5 4 3 2 1
```

You can force GNU **parallel** to print in the order of the values with **--keep-order**/**-k**. This will still run the commands in parallel. The output of the later jobs will be delayed, until the earlier jobs are printed:

```
  parallel -k sleep {}';' echo {} done ::: 5 4 3 2 1
```

Controlling the execution

If your jobs are compute intensive, you will most likely run one job for each core in the system. This is the default for GNU **parallel**.

But sometimes you want more jobs running. You control the number of job slots with **-j**. Give **-j** the number of jobs to run in parallel:

```
  parallel -j50 \
    wget https://ftpmirror.gnu.org/parallel/parallel-{1}{2}22.tar.bz2 \
    ::: 2012 2013 2014 2015 2016 \
    ::: 01 02 03 04 05 06 07 08 09 10 11 12
```

Pipe mode

GNU **parallel** can also pass blocks of data to commands on stdin (standard input):

```
  seq 1000000 | parallel --pipe wc
```

This can be used to process big text files. By default GNU **parallel** splits on \n (newline) and passes a block of around 1 MB to each job.

That's it

You have now learned the basic use of GNU **parallel**. This will probably cover most cases of your use of GNU **parallel**.

The rest of this document will go into more details on each of the sections and cover special use cases.

Learn GNU Parallel in an hour

In this part we will dive deeper into what you learned in the first 5 minutes.

To get going please run this to make some example files:

```
  seq 6 > seq6
  seq 6 -1 1 > seq-6
```

Input sources

On top of the command line, input sources can also be stdin (standard input or '-'), files and fifos and they can be mixed. Files are given after **-a** or **::::**. So these all do the same:

```
parallel echo Dice1={1} Dice2={2} ::: 1 2 3 4 5 6 ::: 6 5 4 3 2 1
parallel echo Dice1={1} Dice2={2} :::: <(seq 6) :::: <(seq 6 -1 1)
parallel echo Dice1={1} Dice2={2} :::: seq6 seq-6
parallel echo Dice1={1} Dice2={2} :::: seq6 :::: seq-6
parallel -a seq6 -a seq-6 echo Dice1={1} Dice2={2}
parallel -a seq6 echo Dice1={1} Dice2={2} :::: seq-6
parallel echo Dice1={1} Dice2={2} ::: 1 2 3 4 5 6 :::: seq-6
cat seq-6 | parallel echo Dice1={1} Dice2={2} :::: seq6 -
```

If stdin (standard input) is the only input source, you do not need the '-':

```
cat seq6 | parallel echo Dice1={1}
```

Linking input sources

You can link multiple input sources with **:::+** and **::::+**:

```
parallel echo {1}={2} ::: I II III IV V VI :::+ 1 2 3 4 5 6
parallel echo {1}={2} ::: I II III IV V VI ::::+ seq6
```

The **:::+** (and **::::+**) will link each value to the corresponding value in the previous input source, so value number 3 from the first input source will be linked to value number 3 from the second input source.

You can combine **:::+** and **:::**, so you link 2 input sources, but generate all combinations with other input sources:

```
parallel echo Dice1={1}={2} Dice2={3}={4} ::: I II III IV V VI ::::+ seq6
\
   ::: VI V IV III II I ::::+ seq-6
```

Building the command line

The command

The command can be a script, a binary or a Bash function if the function is exported using **export -f**:

```
# Works only in Bash
my_func() {
  echo in my_func "$1"
}
export -f my_func
parallel my_func ::: 1 2 3
```

If the command is complex, it often improves readability to make it into a function.

The replacement strings

GNU **parallel** has some replacement strings to make it easier to refer to the input read from the input sources.

If the input is **mydir/mysubdir/myfile.myext** then:

```
{} = mydir/mysubdir/myfile.myext
{.} = mydir/mysubdir/myfile
{/} = myfile.myext
{//} = mydir/mysubdir
{/.} = myfile
{#} = the sequence number of the job
{%} = the job slot number
```

When a job is started it gets a sequence number that starts at 1 and increases by 1 for each new job. The job also gets assigned a slot number. This number is from 1 to the number of jobs running in parallel. It is unique between the running jobs, but is re-used as soon as a job finishes.

The positional replacement strings

The replacement strings have corresponding positional replacement strings. If the value from the 3rd input source is **mydir/mysubdir/myfile.myext**:

```
{3} = mydir/mysubdir/myfile.myext
{3.} = mydir/mysubdir/myfile
{3/} = myfile.myext
{3//} = mydir/mysubdir
{3/.} = myfile
```

So the number of the input source is simply prepended inside the {}'s.

Replacement strings

--plus replacement strings

change the replacement string (-I --extensionreplace --basenamereplace --basenamereplace --dirnamereplace --basenameextensionreplace --seqreplace --slotreplace

--header with named replacement string

{= =}

Dynamic replacement strings

Defining replacement strings

Copying environment

env_parallel

Controlling the output

parset

parset is a shell function to get the output from GNU **parallel** into shell variables.

parset is fully supported for **Bash/Zsh/Ksh** and partially supported for **ash/dash**. I will assume you run **Bash**.

To activate **parset** you have to run:

```
. `which env_parallel.bash`
```

(replace **bash** with your shell's name).

Then you can run:

```
parset a,b,c seq ::: 4 5 6
echo "$c"
```

or:

```
parset 'a b c' seq ::: 4 5 6
echo "$c"
```

If you give a single variable, this will become an array:

```
parset arr seq ::: 4 5 6
echo "${arr[1]}"
```

parset has one limitation: If it reads from a pipe, the output will be lost.

```
echo This will not work | parset myarr echo
echo Nothing: "${myarr[*]}"
```

Instead you can do this:

```
echo This will work > tempfile
parset myarr echo < tempfile
echo ${myarr[*]}
```

sql cvs

Controlling the execution

--dryrun -v

Remote execution

For this section you must have **ssh** access with no password to 2 servers: **$server1** and **$server2**.

```
server1=server.example.com
server2=server2.example.net
```

So you must be able to do this:

```
ssh $server1 echo works
ssh $server2 echo works
```

It can be setup by running 'ssh-keygen -t dsa; ssh-copy-id $server1' and using an empty passphrase. Or you can use **ssh-agent**.

Workers

--transferfile

--transferfile *filename* will transfer *filename* to the worker. *filename* can contain a replacement string:

```
parallel -S $server1,$server2 --transferfile {} wc ::: example.*
parallel -S $server1,$server2 --transferfile {2} \
   echo count {1} in {2}';' wc {1} {2} ::: -l -c ::: example.*
```

A shorthand for **--transferfile {}** is **--transfer**.

--return

--cleanup

A shorthand for **--transfer --return {} --cleanup** is **--trc {}**.

Pipe mode

--pipepart

That's it

Advanced usage

parset fifo, cmd substitution, arrayelements, array with var names and cmds, env_parset

env_parallel

Interfacing with R.

Interfacing with JSON/jq

4dl() { board="$(printf -- '%s' "${1}" | cut -d '/' -f4)" thread="$(printf -- '%s' "${1}" | cut -d '/' -f6)" wget -qO- "https://a.4cdn.org/${board}/thread/${thread}.json" | jq -r ' .posts | map(select(.tim != null)) | map((.tim | tostring) + .ext) | map("https://i.4cdn.org/"'"${board}"'"/"+.)[] ' | parallel --gnu -j 0 wget -nv }

Interfacing with XML/?

Interfacing with HTML/?

Controlling the execution

--termseq

Remote execution

seq 10 | parallel --sshlogin 'ssh -i "key.pem" a@b.com' echo

seq 10 | PARALLEL_SSH='ssh -i "key.pem"' parallel --sshlogin a@b.com echo

seq 10 | parallel --ssh 'ssh -i "key.pem"' --sshlogin a@b.com echo

ssh-agent

The sshlogin file format

Check if servers are up