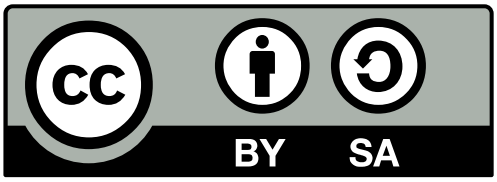
GNU Parallel 2018

Ole Tange

GNU Parallel 2018
First edition



Published by: Ole Tange
http://ole.tange.dk
https://orcid.org/0000-0002-6345-1437

Cover: GNU Parallel's logo is inspired by the café wall illusion

DOI: http://dx.doi.org/10.5281/zenodo.1146014

ISBN: 978-1-387-50988-1

To people

who live life in the parallel lane

Contents

1

How to read this book

There are so few utilities/tools as elegant and amazingly useful
across a wide area of needs as GNU parallel
-- hrbrcoin hrbrmstr@twitter

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.

Chapter 2 will get you started with the basics in 15 minutes. It will introduce you to the basic concepts of GNU Parallel and will show you enough that you can run basic commands in parallel. This will be enough for many tasks.

GNU Parallel has 6 major areas:

- Chapter 4 Input sources
- Chapter 5 Build the command line
- Chapter 6 Control the output
- Chapter 7 Control the execution
- Chapter 8 Remote execution
- Chapter 9 Pipe mode

On top of this, there are a few miscellaneous features

- Chapter 10 Miscellaneous features

After chapter 2 there is no need to read the chapters in sequence: If you need to know how to control the output go right ahead and skip to chapter 6.

The book is written as a 5-in-1 book: You can read it as a beginner, as an intermediate, as an advanced user, as an expert user, or a developer to get all the details. The marking in the border will tell you which audience the section is written for.

Read this if you are level 1.

Read this if you are level 2.

Read this if you are level 3.

Read this if you are level 4.

Read this if you are level 5.

For instance, you do not need to have read anything at level 4 to understand the text at level 3.

Additionally, you do not have to be at the same level in each chapter. Maybe you need advanced knowledge on controlling the execution (chapter 7), while you never use the remote execution (chapter 8), and only use the basic features of `--pipe` (chapter 9).

You are expected to know basic UNIX commands: **`ls`**, **`wc`**, **`cat`**, **`pwd`**, **`seq`**, **`sleep`**, **`echo`**, **`wget`**, **`printf`**, **`rm`**, and **`ssh`**. If any of those are new to you, you should type **`man`** ***`programname`*** and familiarize yourself with those.

You are expected to know that `\`
at the end of the line means the line continues (but that there was no more space on the paper).

If you also have a basic understanding of what **`emacs`**, **`vi`**, **`perl`**, **`mkfifo`**, **`rsync`**, **`alias`**, and **`export`** do, then you will have a much easier time understanding the book.

2

Learn GNU Parallel in 15 minutes

I don't care
I just need to get shit done
-- Sab

This chapter will teach you the most important concepts and what you need to run commands in parallel when 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.{}
```

This will create the files `example.1..5`.

2.1 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
```

Output (order may be different):

```
1
2
3
4
5
```

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

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

Output (order may be different):

```
1 1 2 example.1
2 2 4 example.2
3 3 6 example.3
4 4 8 example.4
5 5 10 example.5
```

If you give multiple `:::`s, GNU Parallel will generate all combinations:

```
parallel echo ::: S M L ::: Green Red
```

Output (order may be different):

```
S Green
S Red
M Green
M Red
L Green
L Red
```

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

```
find example.* -print | parallel echo File
```

Output (order may be different):

```
File example.1
File example.2
File example.3
File example.4
File example.5
```

2.2 Build the command line

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

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

Output (order may be different):

```
1 example.1
2 example.2
3 example.3
4 example.4
5 example.5
```

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.*
```

Output (order may be different):

```
counting lines
1 example.1
counting lines
2 example.2
counting lines
3 example.3
counting lines
4 example.4
counting lines
5 example.5
```

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.*
```

Output (order may be different):

```
counting example.1
1 example.1
counting example.2
2 example.2
counting example.3
3 example.3
counting example.4
4 example.4
counting example.5
5 example.5
```

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.*
```

Output (order may be different):

```
count -l in example.1
1 example.1
count -l in example.2
2 example.2
count -l in example.3
3 example.3
count -l in example.4
4 example.4
count -l in example.5
5 example.5
count -c in example.1
2 example.1
count -c in example.2
4 example.2
count -c in example.3
6 example.3
```

```
count -c in example.4
8 example.4
count -c in example.5
10 example.5
```

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

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

Output (order may be different):

```
echo count -l in example.1; wc -l example.1
echo count -l in example.2; wc -l example.2
echo count -l in example.3; wc -l example.3
echo count -l in example.4; wc -l example.4
echo count -l in example.5; wc -l example.5
echo count -c in example.1; wc -c example.1
echo count -c in example.2; wc -c example.2
echo count -c in example.3; wc -c example.3
echo count -c in example.4; wc -c example.4
echo count -c in example.5; wc -c example.5
```

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

2.3 Control 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
```

Output (order may be different):

```
1 done
2 done
3 done
4 done
5 done
```

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 --keep-order sleep {}';' echo {} done ::: 5 4 3 2 1
```

Output:

```
5 done
4 done
3 done
2 done
1 done
```

2.4 Control 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`**/**`--jobs`**. Give **`--jobs`** the number of jobs to run in parallel. Here we run 2 in parallel:

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

Output:

```
4 done
5 done
1 done
3 done
2 done
```

The two job slots have to run 5 jobs that take 1-5 seconds: **`55555 4444 333 1 22`**. They are run in this sequence:

Job slot 1: **`55555122`**
Job slot 2: **`4444333`**

If you instead run 5 jobs in parallel, all the 5 jobs start at the same time and finish at different times:

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

Output:

```
1 done
2 done
3 done
4 done
5 done
```

The jobs are all run in parallel:

Job slot 1: **`55555`**
Job slot 2: **`4444`**

Job slot 3: **333**
Job slot 4: **1**
Job slot 5: **22**

Instead of giving the number of jobs to run, you can pass **--jobs 0** which will run as many jobs in parallel as possible.

2.5 Pipe mode

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

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

Output (the order may be different):

```
165668  165668 1048571
149796  149796 1048572
149796  149796 1048572
149796  149796 1048572
149796  149796 1048572
149796  149796 1048572
 85352   85352  597465
```

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.

2.6 That's it

You have now mastered 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.

3

Make test files

GNU Parallel is making me pretty happy this morning
-- satanpenguin satanpenguin@twitter

For the rest of the book we need some test files. They can be generated by running this:

```
parallel -k echo ::: A B C > abc-file
parallel -k echo ::: D E F > def-file
perl -e 'printf "A\0B\0C\0"' > abc0-file
perl -e 'printf "A_B_C_"' > abc_-file
perl -e 'printf "f1\tf2\nA\tB\nC\tD\n"' > tsv-file.tsv
perl -e 'for(1..8){print "$_\n"}' > num8
perl -e 'for(1..128){print "$_\n"}' > num128
perl -e 'for(1..30000){print "$_\n"}' > num30000
perl -e 'for(1..1000000){print "$_\n"}' > num1000000
(echo %head1; echo %head2; \
  perl -e 'for(1..10){print "$_\n"}') > num_%header
perl -e 'print "HHHHAAABBBCCC"' > fixedlen
```

You are encouraged to look at the contents of the files, so you understand what they contain.

4

Input sources

Just found out about this awesome syntax for GNU parallel:
`parallel -P20 fping {} ::: host{1..100}`
No need to pipe crap in!
-- Nick Pegg nickpegg@twitter

GNU Parallel reads input from input sources. These can be files, the command line, and stdin (standard input or a pipe).

You will need the test files from chapter 3.

4.1 A single input source

Input can be read from the command line:

```
parallel echo ::: A B C
```

Output (the order may be different because the jobs are run in parallel):

```
A
B
C
```

The input source can be a file:

```
parallel -a abc-file echo
```

Output: Same as above.

Stdin (standard input) can be the input source:

```
cat abc-file | parallel echo
```

Output: Same as above.

The file can also be a FIFO:

```
mkfifo myfifo
cat abc-file > myfifo &
parallel -a myfifo echo
rm myfifo
```

Output: Same as above.

Or command substitution in Bash/Zsh/Ksh:

```
parallel echo :::: <(cat abc-file)
```

Output: Same as above.

4.2 Multiple input sources

GNU Parallel can take multiple input sources given on the command line. GNU Parallel then generates all combinations of the input sources:

```
parallel echo ::: A B C ::: D E F
```

Output (the order may be different):

```
A D
A E
A F
B D
B E
B F
C D
C E
C F
```

The input sources can be files:

```
parallel -a abc-file -a def-file echo
```

Output: Same as above.

Stdin (standard input) can be one of the input sources using -:

```
cat abc-file | parallel -a - -a def-file echo
```

Output: Same as above.

Instead of **-a** files can be given after **::::**:

```
cat abc-file | parallel echo :::: - def-file
```

Output: Same as above.

`:::` and `::::` can be mixed:

```
parallel echo ::: A B C :::: def-file
```

Output: Same as above.

4.2.1 Link arguments from input sources

With **`--link`** you can link the input sources and get one argument from each input source:

```
parallel --link echo ::: A B C ::: D E F
```

Output (the order may be different):

```
A D
B E
C F
```

If one of the input sources is too short, its values will wrap:

```
parallel --link echo ::: A B C D E ::: F G
```

Output (the order may be different):

```
A F
B G
C F
D G
E F
```

For more flexible linking you can use **`:::+`** and **`::::+`**. They work like `:::` and `::::` except they link the previous input source to this input source.

This will link ABC to GHI:

```
parallel echo :::: abc-file :::+ G H I :::: def-file
```

Output (the order may be different):

```
A G D
A G E
A G F
B H D
B H E
B H F
C I D
C I E
C I F
```

This will link GHI to DEF:

```
parallel echo :::: abc-file ::: G H I ::::+ def-file
```

Output (the order may be different):

```
A G D
A H E
A I F
B G D
B H E
B I F
C G D
C H E
C I F
```

If one of the input sources is too short when using `:::+` or `::::+`, the rest will be ignored:

```
parallel echo ::: A B C D E ::+ F G
```

Output (the order may be different):

```
A F
B G
```

4.3 Change the argument separator.

GNU Parallel can use other separators than `:::` or `::::`. This is typically useful if `:::` or `::::` is used in the command to run:

```
parallel --arg-sep ,, echo ,, A B C :::: def-file
```

Output (the order may be different):

```
A D
A E
A F
B D
B E
B F
C D
C E
C F
```

Changing the argument file separator:

```
parallel --arg-file-sep // echo ::: A B C // def-file
```

Output: Same as above.

4.4 Change the record delimiter

GNU Parallel will normally treat a full line as a single record: It uses **\n** as record delimiter. This can be changed with **-d**:

```
parallel -d _ echo :::: abc_-file
```

Output (the order may be different):

```
A
B
C
```

NUL can be given as **\0**:

```
parallel -d '\0' echo :::: abc0-file
```

Output: Same as above.

A shorthand for **-d '\0'** is **-0** (this will often be used to read files from **find ... -print0**):

```
parallel -0 echo :::: abc0-file
```

Output: Same as above.

4.5 End-of-file value for input source

GNU Parallel can stop reading when it encounters a certain value:

```
parallel -E stop echo ::: A B stop C D
```

Output:

```
A
B
```

4.6 Skipping empty lines

Using **--no-run-if-empty** GNU Parallel will skip empty lines.

```
(echo 1; echo; echo 2) | parallel --no-run-if-empty echo
```

Output:

```
1
2
```

5

Build the command line

GNU Parallel is a very awesome tool to use in bash scripts.
It's so easy to parallelize operations on files with it!
-- Mohammed S. Khoory 9a3eedi@twitter

GNU Parallel normally runs commands based on a template and have values from the input sources inserted in the template.

You will need the test files from chapter 3.

5.1 No command means arguments are commands

If no command is given after parallel the arguments themselves are treated as commands:

```
parallel ::: ls 'echo foo' pwd
```

Output (the order may be different):

```
[list of files in current dir]
foo
[/path/to/current/working/dir]
```

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

```
# Only works in Bash
my_func() {
  echo in my_func $1
}
export -f my_func
parallel my_func ::: 1 2 3
```

Output (the order may be different):

```
in my_func 1
in my_func 2
in my_func 3
```

If you use **env_parallel** (see 8.7 Transfer environment variables and functions) then you can also use aliases.

5.2 Replacement strings

5.2.1 The 7 predefined replacement strings

GNU Parallel has several replacement strings. The 7 predefined are:

Replacement string	Value
{}	mydir/mysubdir/myfile.myext
{.}	mydir/mysubdir/myfile
{/}	myfile.myext
{//}	mydir/mysubdir
{/.}	myfile
{#}	*the sequence number of the job*
{%}	*the job slot number*

If no replacement strings are used the default is to append **{}**:

```
parallel echo ::: A/B.C
```

Output:

```
A/B.C
```

The default replacement string is **{}**:

```
parallel echo {} ::: A/B.C
```

Output:

```
A/B.C
```

The replacement string **{.}** removes the extension:

```
parallel echo {.} ::: A/B.C
```

Output:

```
A/B
```

The replacement string **{/}** removes the path:

```
parallel echo {/} ::: A/B.C
```

Output:

```
B.C
```

The replacement string **{//}** keeps only the path:

```
parallel echo {//} ::: A/B.C
```

Output:

```
A
```

The replacement string **{/.}** removes the path and the extension:

```
parallel echo {/.} ::: A/B.C
```

Output:

```
B
```

The replacement string **{#}** gives the job number. When a job is started it gets sequence number that starts at 1 and increases with 1 for each new job.

```
parallel echo {#} ::: A B C
```

Output (the order may be different):

```
1
2
3
```

The replacement string **{%}** gives the job slot number (between 1 and number of jobs to run in parallel). Each job 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.

```
parallel -j 2 echo {%} ::: A B C
```

Output (the order may be different and 1 and 2 may be swapped):

```
1
2
1
```

When inserted the replacement strings are quoted. So there is no need to worry about quoting special characters:

```
echo 'No " needed' | parallel echo {}
```

Output:

```
No " needed
```

If you need to unquote the string, you can use **eval**:

```
echo 'echo foo; echo bar' | parallel echo baz\; eval {}
```

Output:

```
baz
foo
bar
```

5.2.2 Change the replacement strings

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

```
parallel -I ,, echo ,, ::: A/B.C
```

Output:

```
A/B.C
```

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

```
parallel --extensionreplace ,, echo ,, ::: A/B.C
```

Output:

```
A/B
```

The replacement string **{/}** can be replaced with **--basenamereplace**:

```
parallel --basenamereplace ,, echo ,, ::: A/B.C
```

Output:

```
B.C
```

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

```
parallel --dirnamereplace ,, echo ,, ::: A/B.C
```

Output:

```
A
```

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

```
parallel --basenameextensionreplace ,, echo ,, ::: A/B.C
```

Output:

```
B
```

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

```
parallel --seqreplace ,, echo ,, ::: A B C
```

Output (the order may be different):

```
1
2
3
```

The replacement string **{%}** can be changed with **--slotreplace**:

```
parallel -j2 --slotreplace ,, echo ,, ::: A B C
```

Output (the order may be different and 1 and 2 may be swapped):

```
1
2
1
```

5.2.3 Perl expression replacement string

When predefined replacement strings are not flexible enough a perl expression can be used instead. One example is to remove two extensions: **foo.tar.gz** becomes **foo**

```
parallel echo '{= s:\.[^.]+$::;s:\.[^.]+$::; =}' ::: foo.tar.gz
```

Output:

```
foo
```

5.2.3.1 Functions for perl expression replacement strings

In **{= =}** you can access all of GNU Parallel's internal functions and variables. A few are worth mentioning.

total_jobs() returns the total number of jobs:

```
parallel echo Job {#} of {= '$_=total_jobs()' =} ::: {1..5}
```

Output:

```
Job 1 of 5
Job 2 of 5
Job 3 of 5
Job 4 of 5
Job 5 of 5
```

slot() returns the job slot:

```
parallel -j2 echo The job slot is {%} = {= '$_=slot()' =} ::: {1..5}
```

Output:

```
The job slot is 1 = 1
The job slot is 2 = 2
The job slot is 1 = 1
The job slot is 2 = 2
The job slot is 1 = 1
```

seq() returns the sequence number of the job:

```
parallel echo Job {#} = {= '$_=seq()' =} ::: a b c
```

Output:

```
Job 1 = 1
Job 2 = 2
Job 3 = 3
```

Q(...) shell quotes the string:

```
parallel echo {} shell quoted is {= '$_=Q($_)' =} ::: '*/!#$'
```

Output:

```
*/!#$ shell quoted is \*/\!\#\$
```

pQ(...) perl quotes the string, which is useful if the replacement string is used as part of a Perl string, and you do not want Perl to do string substitution on it:

```
echo '@a' | parallel -q perl -e 'print "{= $_=pQ($_); =}\n"'
```

Output:

```
@a
```

skip() skips the job:

```
parallel echo {= 'if($_==3) { skip() }' =} ::: {1..5}
```

Output:

```
1
2
4
5
```

@arg contains the input source variables:

```
parallel echo {= 'if($arg[1]==$arg[2]) { skip() }' =} \
  ::: {1..3} ::: {1..3}
```

Output:

```
1 2
1 3
2 1
2 3
3 1
```

```
3 2
```

If the strings **{=** and **=}** cause problems they can be replaced with **--parens**:

```
parallel --parens ,,,, echo ',, s:\.[^.]+$::;s:\.[^.]+$::; ,,' \
  ::: foo.tar.gz
```

Output:

```
foo
```

To define a shorthand replacement string use **--rpl**:

```
parallel --rpl '.. s:\.[^.]+$::;s:\.[^.]+$::;' echo '..' \
  ::: foo.tar.gz
```

Output: Same as above.

GNU Parallel's 7 replacement strings are implemented as this:

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

5.2.4 Dynamic replacement strings

If the shorthand contains matching parenthesis the replacement string becomes a dynamic replacement string and the string in the parenthesis can be accessed as **$$1**. If there are multiple matching parenthesis, the matched strings can be accessed using **$$2**, **$$3** and so on.

You can think of this as giving arguments to the replacement string. Here we give the argument **.tar.gz** to the replacement string ***{%string}*** which removes ***string***:

```
parallel --rpl '{%(.+?)} s/$$1$//;' echo {%.tar.gz}.zip ::: foo.tar.gz
```

Output:

```
foo.zip
```

Here we give the two arguments **tar.gz** and **zip** to the replacement string **{/*string1*/*string2*}** which replaces ***string1*** with ***string2***:

```
parallel --rpl '{/(.+?)/(.*?)} s/$$1/$$2/;' echo {/tar.gz/zip} \
  ::: foo.tar.gz
```

Output:

```
foo.zip
```

5.2.5 Positional replacement strings

With multiple input sources the argument from the individual input sources can be accessed with **{*number*}**:

```
parallel echo {1} and {2} ::: A B ::: C D
```

Output (the order may be different):

```
A and C
A and D
B and C
B and D
```

The positional replacement strings can also be modified using **/**, **//**, **/.**, and **..**

Replacement string	Value
{3}	mydir/mysubdir/myfile.myext
{3.}	mydir/mysubdir/myfile
{3/}	myfile.myext
{3//}	mydir/mysubdir
{3/.}	myfile

Like this:

```
parallel echo /={1/} //={1//} /.={1/.} .={1.} ::: A/B.C D/E.F
```

Output (the order may be different):

```
/=B.C //=A /.=B .=A/B
/=E.F //=D /.=E .=D/E
```

If a position is negative, it will refer to the input source counted from behind:

```
parallel echo 1={1} 2={2} 3={3} -1={-1} -2={-2} -3={-3} \
  ::: A B ::: C D ::: E F
```

Output (the order may be different):

```
1=A 2=C 3=E -1=E -2=C -3=A
1=A 2=C 3=F -1=F -2=C -3=A
1=A 2=D 3=E -1=E -2=D -3=A
1=A 2=D 3=F -1=F -2=D -3=A
1=B 2=C 3=E -1=E -2=C -3=B
1=B 2=C 3=F -1=F -2=C -3=B
1=B 2=D 3=E -1=E -2=D -3=B
1=B 2=D 3=F -1=F -2=D -3=B
```

5.2.6 Positional perl expression replacement string

To use a perl expression as a positional replacement string simply prepend the perl expression with number and space:

```
parallel echo '{=2 s:\.[^.]+$::;s:\.[^.]+$::; =} {1}' \
  ::: bar ::: foo.tar.gz
```

Output:

```
foo bar
```

If a shorthand defined using `--rpl` starts with `{` it can be used as a positional replacement string, too:

```
parallel --rpl '{..} s:\.[^.]+$::;s:\.[^.]+$::;' echo {2..} {1} \
  ::: bar ::: foo.tar.gz
```

Output: Same as above.

5.2.7 Input from columns

The columns in a file can be bound to positional replacement strings using `--colsep`. Here the columns are separated by TAB (\t):

```
parallel --colsep '\t' echo 1={1} 2={2} :::: tsv-file.tsv
```

Output (the order may be different):

```
1=f1 2=f2
1=A 2=B
1=C 2=D
```

5.2.8 Header defined replacement strings

With `--header` GNU Parallel will use the first value of the input source as the name of the replacement string. Only the non-modified version `{}` is supported:

```
parallel --header : echo f1={f1} f2={f2} ::: f1 A B ::: f2 C D
```

Output (the order may be different):

```
f1=A f2=C
f1=A f2=D
f1=B f2=C
f1=B f2=D
```

It is useful with **--colsep** for processing files with TAB separated values:

```
parallel --header : --colsep '\t' echo f1={f1} f2={f2} \
  :::: tsv-file.tsv
```

Output (the order may be different):

```
f1=A f2=B
f1=C f2=D
```

5.2.9 More pre-defined replacement strings with --plus

--plus adds the replacement strings **{+/} {+.} {+..} {+...} {..} {...} {/..} {/...} {##}**. The idea being that **{+foo}** matches the opposite of **{foo}** and **{}** = **{+/}**/**{/}** = **{.}**.**{+.}** = **{+/}**/**{/.}**.**{+.}** = **{..}**.**{+..}** = **{+/}**/**{/..}**.**{+..}** = **{...}**.**{+...}** = **{+/}**/**{/...}**.**{+...}**.

```
parallel --plus echo {} ::: dir/sub/file.ex1.ex2.ex3
parallel --plus echo {+/}/{/} ::: dir/sub/file.ex1.ex2.ex3
parallel --plus echo {.}.{+.} ::: dir/sub/file.ex1.ex2.ex3
parallel --plus echo {+/}/{/.}.{+.} ::: dir/sub/file.ex1.ex2.ex3
parallel --plus echo {..}.{+..} ::: dir/sub/file.ex1.ex2.ex3
parallel --plus echo {+/}/{/..}.{+..} ::: dir/sub/file.ex1.ex2.ex3
parallel --plus echo {...}.{+...} ::: dir/sub/file.ex1.ex2.ex3
parallel --plus echo {+/}/{/...}.{+...} ::: dir/sub/file.ex1.ex2.ex3
```

Output:

```
dir/sub/file.ex1.ex2.ex3
```

{##} is the total number of jobs:

```
parallel --plus echo Job {#} of {##} ::: {1..5}
```

Output:

```
Job 1 of 5
Job 2 of 5
Job 3 of 5
Job 4 of 5
Job 5 of 5
```

5.2.10 Dynamic replacement strings with --plus

- **--plus** also defines these dynamic replacement strings:

Replacement string	Value	Bash inspiration
{:-string}	Default value is **string** if the argument is empty.	**${myvar:-myval}**
{:number}	Substring from **number** till end of string.	**${myvar:2}**
{:number1:number2}	Substring from **number1** to **number2**.	**${myvar:2:3}**
{#string}	If the argument starts with **string**, remove it.	**${myvar#bc}**
{%string}	If the argument ends with **string**, remove it.	**${myvar%de}**
{/string1/string2}	Replace **string1** with **string2**.	**${myvar/def/ghi}**
{^string}	If the argument starts with **string**, upper case it. **string** must be a single letter.	**${myvar^a}**
{^^string}	If the argument contains **string**, upper case it. **string** must be a single letter.	**${myvar^^a}**
{,string}	If the argument starts with string, lower case it. string must be a single letter.	**${myvar,A}**
{,,string}	If the argument contains string, lower case it. string must be a single letter.	**${myvar,,A}**

- They are inspired from **Bash**:

```
unset myvar
echo ${myvar:-myval}
parallel --plus echo {:-myval} ::: "$myvar"

myvar=abcAaAdef
echo ${myvar:2}
parallel --plus echo {:2} ::: "$myvar"

echo ${myvar:2:3}
parallel --plus echo {:2:3} ::: "$myvar"

echo ${myvar#bc}
parallel --plus echo {#bc} ::: "$myvar"
echo ${myvar#abc}
parallel --plus echo {#abc} ::: "$myvar"
```

```
echo ${myvar%de}
parallel --plus echo {%de} ::: "$myvar"
echo ${myvar%def}
parallel --plus echo {%def} ::: "$myvar"

echo ${myvar/def/ghi}
parallel --plus echo {/def/ghi} ::: "$myvar"

echo ${myvar^a}
parallel --plus echo {^a} ::: "$myvar"
echo ${myvar^^a}
parallel --plus echo {^^a} ::: "$myvar"

myvar=AbcAaAdef
echo ${myvar,A}
parallel --plus echo '{,A}' ::: "$myvar"
echo ${myvar,,A}
parallel --plus echo '{,,A}' ::: "$myvar"
```

Output:

```
myval
myval
cAaAdef
cAaAdef
cAa
cAa
abcAaAdef
abcAaAdef
AaAdef
AaAdef
abcAaAdef
abcAaAdef
abcAaA
abcAaA
abcAaAghi
abcAaAghi
AbcAaAdef
AbcAaAdef
AbcAAAdef
AbcAAAdef
abcAaAdef
abcAaAdef
abcaaadef
abcaaadef
```

5.3 Insert more than one argument

With **`--xargs`** GNU Parallel will fit as many arguments as possible on a single line:

```
cat num30000 | parallel --xargs 'echo {} | wc -w'
```

Output (number of arguments can differ a some):

```
6309
23691
```

The 30000 arguments fit in 2 command lines: 23691 arguments for the first command and 6309 for the second.

The maximal length of a single line can be set with **-s**. With a maximal line length of 30000 chars, 6 commands will be run with around 5000 arguments for each command:

```
cat num30000 | parallel --xargs -s 30000 'echo {} | wc -w'
```

Output (number of arguments can differ some):

```
6218
4997
5628
4997
4997
3163
```

For better parallelism, GNU Parallel can distribute the arguments between all the parallel jobs when end-of-file is met.

Below GNU Parallel reads the last argument when generating the second job. When GNU Parallel reads the last argument, it spreads all the arguments for the second job over 4 jobs instead, as 4 parallel jobs are requested.

Using **-m** the first job will be the same as the **--xargs** example above, but the second job will be split into 4 evenly sized jobs, resulting in a total of 5 jobs:

```
cat num30000 | parallel -j 4 -m 'echo {} | wc -w'
```

Output (if you run this under Bash on GNU/Linux):

```
23691
1578
1578
1578
1575
```

This is even more visible when running 4 jobs with 10 arguments. The 10 arguments are being spread over 4 jobs:

```
parallel --jobs 4 -m echo ::: 1 2 3 4 5 6 7 8 9 10
```

Output:

```
1 2 3
4 5 6
7 8 9
10
```

A replacement string can be part of a word. **-m** will not repeat the context, that touches the replacement string:

```
parallel --jobs 4 -m echo pre-{}-post ::: A B C D E F G
```

Output (the order may be different):

```
pre-A B-post
pre-C D-post
pre-E F-post
pre-G-post
```

To repeat the context use **-X** which otherwise works like **-m**:

```
parallel --jobs 4 -X echo pre-{}-post ::: A B C D E F G
```

Output (the order may be different):

```
pre-A-post pre-B-post
pre-C-post pre-D-post
pre-E-post pre-F-post
pre-G-post
```

To limit the number of arguments use **-N**:

```
parallel -N3 echo ::: A B C D E F G H
```

Output (the order may be different):

```
A B C
D E F
G H
```

-N also sets the positional replacement strings:

```
parallel -N3 echo 1={1} 2={2} 3={3} ::: A B C D E F G H
```

Output (the order may be different):

```
1=A 2=B 3=C
1=D 2=E 3=F
1=G 2=H 3=
```

-N0 reads 1 argument but inserts none:

```
parallel -N0 echo foo ::: 1 2 3
```

Output:

```
foo
foo
```

```
foo
```

This is useful for running the same command multiple times in parallel.

5.4 Quote the command line

Command lines that contain special characters may need to be protected from the shell.

The **perl** program **print "@ARGV\n"** basically works like **echo**.

```
perl -e 'print "@ARGV\n"' A
```

Output:

```
A
```

To run that in parallel the command needs to be quoted:

```
parallel perl -e 'print "@ARGV\n"' ::: This wont work
```

Output:

```
[Nothing – it did not work]
```

To quote the command use **-q**:

```
parallel -q perl -e 'print "@ARGV\n"' ::: This works
```

Output (the order may be different):

```
This
works
```

Or you can quote the critical part using **\'**:

```
parallel perl -e \''print "@ARGV\n"'\' ::: This works, too
```

Output (the order may be different):

```
This
works,
too
```

GNU Parallel can also ****-quote full lines. Simply run this:

```
parallel --shellquote
Warning: Input is read from the terminal. You either know what you
Warning: are doing (in which case: YOU ARE AWESOME!) or you forgot
Warning: ::: or :::: or to pipe data into parallel. If so
Warning: consider going through the tutorial: man parallel_tutorial
Warning: Press CTRL-D to exit.
perl -e 'print "@ARGV\n"'
[CTRL-D]
```

- Output:

```
perl\ -e\ \'print\ \"@ARGV\\n\"\'
```

- This can then be used as the command:

```
parallel perl\ -e\ \'print\ \"@ARGV\\n\"\' ::: This also works
```

- Output (the order may be different):

```
This
also
works
```

5.5 Trim space from arguments

- Space can be trimmed on the arguments using `--trim`:

```
parallel --trim r echo pre-{}-post ::: ' A '
```

- Output:

```
pre- A-post
```

- To trim on the left side:

```
parallel --trim l echo pre-{}-post ::: ' A '
```

- Output:

```
pre-A -post
```

- To trim on both sides:

```
parallel --trim lr echo pre-{}-post ::: ' A '
```

- Output:

```
pre-A-post
```

5.6 Respect the shell

- This tutorial uses Bash as the shell. GNU Parallel respects which shell you are using, so in `zsh` you can do:

```
parallel echo \={} ::: zsh bash ls
```

- Output:

```
/usr/bin/zsh
/bin/bash
/bin/ls
```

- In **csh** you can do:

```
parallel 'set a="{}"; if( { test -d "$a" } ) echo "$a is a dir"' ::: *
```

- Output:

```
[somedir] is a dir
```

- This also becomes useful if you use GNU Parallel in a shell script: GNU Parallel will use the same shell as the shell script.

6

Control the output

After analyzing the requirements
I'll re-implement whatever distributed system you got
with postgres, cron, and gnu parallel (ง'̀-'́)ง
-- david karapetyan kontrol_theory@twitter

GNU Parallel normally prints the output from a job when it is done.

6.1 Tag output

The output can be prefixed with the argument:

```
parallel --tag echo foo-{} ::: A B C
```

Output (the order may be different):

```
A       foo-A
B       foo-B
C       foo-C
```

`--tag` is a shorthand for **`--tagstring {}`**. To prefix it with another string use **`--tagstring`**:

```
parallel --tagstring {}-bar echo foo-{} ::: A B C
```

Output (the order may be different):

```
A-bar   foo-A
B-bar   foo-B
C-bar   foo-C
```

6.2 See what is being run

To see what commands will be run without running them use **--dryrun**:

```
parallel --dryrun echo {} ::: A B C
```

Output (the order may be different):

```
echo A
echo B
echo C
```

To print the command before running them use **--verbose**:

```
parallel --verbose echo {} ::: A B C
```

Output (the order may be different):

```
echo A
echo B
A
echo C
B
C
```

This, however, is only half the truth. For further details see 8.8.

6.3 Force same order as input

This function:

```
half_line_print() {
  printf "%s-start\n%s" $1 $1
  sleep $1
  printf "%s\n" -middle
  echo $1-end
}
export -f half_line_print
```

takes a number (#) as argument. It prints a full line '#-start' followed by half a line '#'. Then it sleeps for # seconds, before it prints '-middle' followed by '#-end'.

To force the output in the same order as the arguments use **--keep-order**/**-k**:

```
parallel -j2 -k half_line_print ::: 4 2 1
```

Output:

```
4-start
4-middle
4-end
```

```
2-start
2-middle
2-end
1-start
1-middle
1-end
```

6.4 Output before jobs complete

GNU Parallel will postpone the output until the command completes:

```
parallel -j2 half_line_print ::: 4 2 1
```

Output:

```
2-start
2-middle
2-end
1-start
1-middle
1-end
4-start
4-middle
4-end
```

This is because **`--group`** is the default. To get the output immediately use **`--ungroup`**/**`-u`**:

```
parallel -j2 --ungroup half_line_print ::: 4 2 1
```

Output:

```
4-start
42-start
2-middle
2-end
1-start
1-middle
1-end
-middle
4-end
```

`--ungroup` is fast, but it disables **`--tag`** and can cause half a line from one job to be mixed with half a line of another job. That has happened in the second line, where the line '4-middle' is mixed with '2-start'.

To avoid this use **`--linebuffer`** which only outputs full lines:

```
parallel -j2 --linebuffer half_line_print ::: 4 2 1
```

Output:

```
4-start
```

```
2-start
2-middle
2-end
1-start
1-middle
1-end
4-middle
4-end
```

With `--keep-order` `--line-buffer` GNU Parallel will continuously output lines from the first job until it finishes, then GNU Parallel will continuously output lines from the second job while that is running. It will buffer full lines, but the output from different jobs will not mix.

Compare:

```
parallel -j4 'echo {}-a;sleep {};echo {}-b' ::: 1 3 2 4
```

Output:

```
1-a
1-b
2-a
2-b
3-a
3-b
4-a
4-b
```

To:

```
parallel -j4 --line-buffer 'echo {}-a;sleep {};echo {}-b' ::: 1 3 2 4
```

Output:

```
2-a
3-a
1-a
4-a
1-b
2-b
3-b
4-b
```

And:

```
parallel -j4 -k --line-buffer 'echo {}-a;sleep {};echo {}-b' ::: 1 3 2 4
```

Output:

```
1-a
1-b
3-a
3-b
2-a
```

```
2-b
4-a
4-b
```

6.4.1 Buffer on disk

GNU Parallel buffers output in temporary files. If a program has more output than there is free disk space, the disk will fill when using **`--group`** or **`--line-buffer --keep-order`**. This does not apply when using **`--line-buffer`** without **`--keep-order`** (which buffers a single line in RAM) and **`--ungroup`** (which does not buffer).

6.5 Save output into files

GNU Parallel can save the output of each job into files:

```
parallel --files echo ::: A B C
```

Output will be similar to this:

```
/tmp/pAh6uWuQCg.par
/tmp/opjhZCzAX4.par
/tmp/W0AT_Rph2o.par
```

By default GNU Parallel will cache the output in files in **`/tmp`**. This can be changed by setting **`$TMPDIR`** or **`--tmpdir`**:

```
parallel --tmpdir /var/tmp --files echo ::: A B C
```

Output will be similar to this:

```
/var/tmp/N_vk7phQRc.par
/var/tmp/7zA4Ccf3wZ.par
/var/tmp/Liuka_2LP.par
```

Or:

```
TMPDIR=/var/tmp parallel --files echo ::: A B C
```

Output: Same as above.

The output files can be saved in a structured way using **`--results`**:

```
parallel --results outdir echo ::: A B C
```

Output:

```
A
B
C
```

These files were also generated containing the standard output (stdout), standard error (stderr), and the sequence number (seq):

```
outdir/1/A/seq
outdir/1/A/stderr
outdir/1/A/stout
outdir/1/B/seq
outdir/1/B/stderr
outdir/1/B/stdout
outdir/1/C/seq
outdir/1/C/stderr
outdir/1/C/stdout
```

`--header :` will take the first value as name and use that in the directory structure. This is useful if you are using multiple input sources:

```
parallel --header : --results outdir echo ::: f1 A B ::: f2 C D
```

Generated files:

```
outdir/f1/A/f2/C/seq
outdir/f1/A/f2/C/stderr
outdir/f1/A/f2/C/stdout
outdir/f1/A/f2/D/seq
outdir/f1/A/f2/D/stderr
outdir/f1/A/f2/D/stdout
outdir/f1/B/f2/C/seq
outdir/f1/B/f2/C/stderr
outdir/f1/B/f2/C/stdout
outdir/f1/B/f2/D/seq
outdir/f1/B/f2/D/stderr
outdir/f1/B/f2/D/stdout
```

The directories are named after the variables and their values.

If the argument for **`--results`** contains a replacement string, stdout will be saved in that name:

```
parallel --results my{1}-{2}.out echo ::: A B ::: C D
```

Generated files:

```
myA-C.out
myA-D.out
myB-C.out
myB-D.out
```

If the argument for **`--results`** contains a replacement string and ends in **`/`**, output will be saved in a dir of that name:

```
parallel --results my{1}-{2}-dir/ echo ::: A B ::: C D
```

Generated files:

```
myA-C-dir/stderr
myA-C-dir/seq
myA-C-dir/stdout
myA-D-dir/stderr
myA-D-dir/seq
myA-D-dir/stdout
myB-C-dir/stderr
myB-C-dir/seq
myB-C-dir/stdout
myB-D-dir/stderr
myB-D-dir/seq
myB-D-dir/stdout
```

6.6 Save to CSV/TSV

Many programs support files with Comma Separated Values/Tab Separated Values. GNU Parallel is no exception. If the argument for **`--results`** ends in .csv or .tsv the output will be a CSV/TSV file.

```
parallel --results my.csv echo ::: A B ::: C D
```

Content of **`my.csv`**:

```
Seq,Host,Starttime,JobRuntime,Send,Receive,Exitval,Signal,Command,V1,V2,St
dout,Stderr
1,:,1519688383.281,0.007,0,4,0,0,"echo A C",A,C,"A C
",
2,:,1519688383.283,0.006,0,4,0,0,"echo A D",A,D,"A D
",
3,:,1519688383.285,0.003,0,4,0,0,"echo B C",B,C,"B C
",
4,:,1519688383.287,0.002,0,4,0,0,"echo B D",B,D,"B D
",
```

This is faster than 6.7.1 CSV as SQL base.

6.7 Save to an SQL base

GNU Parallel can save into an SQL base. Point GNU Parallel to a table and it will put the joblog there together with the variables and the output each in their own column.

6.7.1 CSV as SQL base

The simplest is to use a CSV file as the storage table:

```
parallel --sqlandworker csv:////%2Ftmp%2Flog.csv \
```

```
  seq ::: 10 ::: 12 13 14
cat /tmp/log.csv
```

Note how '/' in the path must be written as %2F.

Output will be similar to:

```
Seq,Host,Starttime,JobRuntime,Send,Receive,Exitval,_Signal,
Command,V1,V2,Stdout,Stderr
1,:,1458254498.254,0.069,0,9,0,0,"seq 10 12",10,12,"10
11
12
",
2,:,1458254498.278,0.080,0,12,0,0,"seq 10 13",10,13,"10
11
12
13
",
3,:,1458254498.301,0.083,0,15,0,0,"seq 10 14",10,14,"10
11
12
13
14
",
```

The first columns are well known from **`--joblog`** (see 7.7 Logfile). **`V1`** and **`V2`** are data from the input sources. **`Stdout`** and **`Stderr`** are standard output and standard error, respectively.

A proper CSV reader (like LibreOffice Calc or R's **`read.csv`** command) will read this format correctly - even with fields containing newlines as above.

If the output is big you may want to put it into files using **`--results`**. The CSV file will then contain the file names:

```
parallel --results outdir --sqlandworker csv:////%2Ftmp%2Flog2.csv \
  seq ::: 10 ::: 12 13 14
cat /tmp/log2.csv
```

Output will be similar to:

```
Seq,Host,Starttime,JobRuntime,Send,Receive,Exitval,_Signal,
Command,V1,V2,Stdout,Stderr
1,:,1458824738.287,0.029,0,9,0,0,
"seq 10 12",10,12,outdir/1/10/2/12/stdout,outdir/1/10/2/12/stderr
2,:,1458824738.298,0.025,0,12,0,0,
"seq 10 13",10,13,outdir/1/10/2/13/stdout,outdir/1/10/2/13/stderr
3,:,1458824738.309,0.026,0,15,0,0,
"seq 10 14",10,14,outdir/1/10/2/14/stdout,outdir/1/10/2/14/stderr
```

6.7.2 DBURL as table

The CSV file is an example of a DBURL.

GNU Parallel uses a DBURL to address the table. A DBURL has this format:

```
vendor://[[user][:password]@][host][:port]/[database[/table]
```

Example:

```
mysql://scott:tiger@my.example.com/mydatabase/mytable
postgresql://scott:tiger@pg.example.com/mydatabase/mytable
sqlite3:///%2Ftmp%2Fmydatabase/mytable
csv:////%2Ftmp%2Flog.csv
```

To refer to `/tmp/mydatabase` with `sqlite` or `csv` you need to encode the `/` as `%2F`.

Run a job using `sqlite` on `mytable` in `/tmp/mydatabase`:

```
DBURL=sqlite3:///%2Ftmp%2Fmydatabase
DBURLTABLE=$DBURL/mytable
parallel --sqlandworker $DBURLTABLE echo ::: foo bar ::: baz quux
```

To see the result:

```
sql $DBURL 'SELECT * FROM mytable ORDER BY Seq;'
```

Output will be similar to:

```
Seq|Host|Starttime|JobRuntime|Send|Receive|Exitval|_Signal|
Command|V1|V2|Stdout|Stderr
1|:|1451619638.903|0.806||8|0|0|echo foo baz|foo|baz|foo baz
|
2|:|1451619639.265|1.54||9|0|0|echo foo quux|foo|quux|foo quux
|
3|:|1451619640.378|1.43||8|0|0|echo bar baz|bar|baz|bar baz
|
4|:|1451619641.473|0.958||9|0|0|echo bar quux|bar|quux|bar quux
|
```

6.7.3 Use multiple workers

Using an SQL base as storage costs overhead in the order of 1 second per job.

One of the situations where this makes sense is if you have multiple workers.

You can then have a single master machine that submits jobs to the SQL base (but which does not do any of the work):

```
parallel --sqlmaster $DBURLTABLE echo ::: foo bar ::: baz quux
```

On the worker machines, you run exactly the same command except you replace **--sqlmaster** with **--sqlworker**.

```
parallel --sqlworker $DBURLTABLE echo ::: foo bar ::: baz quux
```

To run a master and a worker on the same machine use **--sqlandworker** as shown earlier.

The **--sqlmaster** will exit as soon as the jobs are put into the database, unless **--wait** is specified. This will make the **--sqlmaster** wait for all the jobs to complete before exiting. The **--sqlworker** will exit when all jobs in the database is finished.

You can add more jobs to an existing table by prepending the DBURLTABLE with +:

```
parallel --sqlmaster +$DBURLTABLE echo ::: foo2 bar2 ::: baz2 quux2
```

6.8 Save output to shell variables

GNU Parset will set shell variables to the output of GNU Parallel. GNU Parset has one important limitation: It cannot be part of a pipe. In particular, this means it cannot read anything from standard input (stdin) or pipe output to another program.

GNU Parset is a shell function. You active it by running:

```
env_parallel --install
```

After which you start a new shell

Parset is supported for **bash**, **dash**, **ash**, **sh**, **ksh**, and **zsh**.

To use **parset** put the destination variables before the normal GNU Parallel options and command:

```
parset myvar1,myvar2 -j2 echo ::: a b
echo $myvar1
echo $myvar2
```

Output:

```
a
b
```

If you only give a single variable, it will be treated as an array:

```
parset myarray seq {} 5 ::: 1 2 3
echo "${myarray[1]}"
```

Output:

```
2
3
4
5
```

- The commands to run can be an array:

```
cmd=("echo '<<Joe  \"double  space\"  cartoon>>'" "pwd")
parset data -j2 ::: "${cmd[@]}"
echo "${data[0]}"
echo "${data[1]}"
```

- Output:

```
<<Joe  "double  space"  cartoon>>
[current dir]
```

6.8.1 Do not read from a pipe

GNU Parset cannot read from a pipe. This is because **`parset`** would then be started in a subshell and thus the output would not be seen in the starting shell. There are several workarounds for that.

6.8.1.1 Use a temporary file

Instead of reading directly from a pipe, save the output to a file and let **`parset`** read from that.

```
seq 3 > parallel_input
parset res1,res2,res3 echo :::: parallel_input
echo "$res1"
echo "$res2"
echo "$res3"
rm parallel_input
```

6.8.1.2 Use process substitution

If your shell supports process substitution (Bash, Zsh, and Ksh all do), then you can use that.

```
parset res echo :::: <(seq 100)
echo "${res[1]}"
echo "${res[99]}"
```

6.8.1.3 Use a FIFO

If the amount of data is big or you need GNU Parset to start reading before all output is generated, then using a FIFO might be an option.

```
mkfifo input_fifo
seq 3 > input_fifo &
parset res1,res2,res3 echo :::: input_fifo
echo "$res1"
echo "$res2"
echo "$res3"
rm input_fifo
```

6.8.2 env_parset

`env_parset` will do the same as **`parset`** but uses **`env_parallel`** (see 8.7 Transfer environment variables and functions) instead of **`parallel`**, so you will have access to aliases, unexported functions, and unexported variables.

7

Control the execution

So, you don't know you can use GNU parallel for most of your tasks/process/scripts
and still call yourself a DevOps Engineer?
Nice
-- Esparta Palma esparta@twitter

GNU Parallel will start one job per CPU core in parallel and finish when all jobs are done.

You will need the test files from chapter 3.

7.1 Number of simultaneous jobs

The number of concurrent jobs is given with **`--jobs`**/**`-j`** (**`-N0`** reads a single argument, but inserts none – so this runs **`sleep 1`** many times in parallel):

```
/usr/bin/time parallel -N0 -j64 sleep 1 :::: num128
```

With 64 jobs in parallel, the 128 **`sleep`**s will take 2-8 seconds to run - depending on how fast your machine is.

By default **`--jobs`** is the same as the number of CPU cores. So this:

```
/usr/bin/time parallel -N0 sleep 1 :::: num128
```

should take twice the time of running 2 jobs per CPU core:

```
/usr/bin/time parallel -N0 --jobs 200% sleep 1 :::: num128
```

`--jobs 0` will run as many jobs in parallel as possible:

```
/usr/bin/time parallel -N0 --jobs 0 sleep 1 :::: num128
```

which should take 1-7 seconds depending on how fast your machine is.

--jobs can read from a file which is re-read when a job finishes:

```
echo 50% > my_jobs
/usr/bin/time parallel -N0 --jobs my_jobs sleep 1 :::: num128 &
sleep 1
echo 0 > my_jobs
wait
```

GNU Parallel will read **my_jobs** when starting. It contains 50%, so GNU Parallel will compute 50% of the number of cores and start this many jobs in parallel.

Because of the **&** GNU Parallel will be started in the background.

After one second **0** is put into **my_jobs**. When a job finishes, GNU Parallel re-reads **my_jobs**, and then GNU Parallel starts as many jobs as possible.

Instead of basing the percentage on the number of CPU cores GNU Parallel can base it on the number of CPUs:

```
parallel --use-cpus-instead-of-cores -N0 sleep 1 :::: num8
```

7.2 Shuffle job order

If you have many jobs (e.g. by multiple combinations of input sources), it can be handy to shuffle the jobs, so you get different values run first. Use **--shuf** for that:

```
parallel --shuf echo ::: 1 2 3 ::: a b c ::: A B C
```

Output:

```
All combinations but different order for each run.
```

7.3 Interactivity

GNU Parallel can ask the user if a command should be run using **--interactive**:

```
parallel --interactive echo ::: 1 2 3
```

Output:

```
echo 1 ?...y
echo 2 ?...n
1
echo 3 ?...y
```

```
3
```

GNU Parallel can be used to put arguments on the command line for an interactive command such as **emacs** to edit one file at a time:

```
parallel --tty emacs ::: file1 file2 file3
```

Or give multiple arguments in one go to open multiple files:

```
parallel -X --tty vi ::: file1 file2 file3
```

7.4 A terminal for every job

Using **--tmux** GNU Parallel can start a terminal for every job run:

```
seq 10 20 | parallel --tmux 'echo start {}; sleep {}; echo done {}'
```

This will tell you to run something similar to:

```
tmux -S /tmp/tmsrPrO0 attach
```

Using normal **tmux** keystrokes (CTRL-b n or CTRL-b p) you can cycle between windows of the running jobs. When a job is finished it will pause for 10 seconds before closing the window.

To have GNU Parallel open each job in its own pane use **--tmuxpane**. **--fg** will connect to **tmux** immediately:

```
parallel --tmuxpane --fg \
  'echo start {}; sleep {}; echo done {}' ::: 10 11 12 13 14 15 16 17
```

7.5 Timing

Some jobs do heavy I/O when they start. To avoid a thundering herd GNU Parallel can delay starting new jobs. **--delay** *X* will make sure there is at least *X* seconds between each start:

```
parallel --delay 2.5 echo Starting {}\;date ::: 1 2 3
```

Output:

```
Starting 1
Thu Aug 15 16:24:33 CEST 2013
Starting 2
Thu Aug 15 16:24:35 CEST 2013
Starting 3
Thu Aug 15 16:24:38 CEST 2013
```

If jobs taking more than a certain amount of time are known to fail, they can be stopped with **`--timeout`**. The accuracy of **`--timeout`** is 2 seconds. **`--timeout 100000`** can be written as **`--timeout 1d3.5h16.6m4s`**.

```
parallel --timeout 4.1 sleep {}\; echo {} ::: 2 4 6 8
```

Output:

```
2
4
```

GNU Parallel can compute the median runtime for jobs and kill those that take more than 200% of the median runtime:

```
parallel --timeout 200% sleep {}\; echo {} ::: 2.1 2.2 3 7 2.3
```

Output:

```
2.1
2.2
3
2.3
```

This is useful if you have a few jobs that run amok and take much longer than the rest of the jobs.

7.6 Progress information

Based on the runtime of completed jobs GNU Parallel can estimate the total runtime:

```
parallel --eta sleep ::: 1 3 2 2 1 3 3 2 1
```

Output:

```
Computers / CPU cores / Max jobs to run
1:local / 2 / 2

Computer:jobs running/jobs completed/%of started jobs/
Average seconds to complete
ETA: 2s 0left 1.11avg  local:0/9/100%/1.1s
```

GNU Parallel can give progress information with **`--progress`**:

```
parallel --progress sleep ::: 1 3 2 2 1 3 3 2 1
```

Output:

```
Computers / CPU cores / Max jobs to run
1:local / 2 / 2

Computer:jobs running/jobs completed/%of started jobs/
```

```
Average seconds to complete
local:0/9/100%/1.1s
```

A progress bar can be shown with **`--bar`**:

```
parallel --bar sleep ::: 1 3 2 2 1 3 3 2 1
```

And a graphic bar can be shown with **`--bar`** and **`zenity`**:

```
seq 1000 | parallel -j10 --bar '(echo -n {};sleep 0.1)' \
  2> >(zenity --progress --auto-kill --auto-close)
```

7.7 Logfile

A log-file of the jobs completed so far can be generated with **`--joblog`**:

```
parallel --joblog /tmp/log exit  ::: 1 2 3 0
cat /tmp/log
```

Output:

```
Seq Host Starttime      Runtime Send Receive Exitval Signal Command
1   :    1376577364.974 0.008   0    0       1       0      exit 1
2   :    1376577364.982 0.013   0    0       2       0      exit 2
3   :    1376577364.990 0.013   0    0       3       0      exit 3
4   :    1376577365.003 0.003   0    0       0       0      exit 0
```

The log contains the job sequence, which host the job was run on, the start time and run time, how much data was transferred, the exit value, the signal that killed the job, and finally the command being run.

7.8 Resume jobs

With a joblog GNU Parallel can be stopped and later pickup where it left off. It is important that the input of the completed jobs is unchanged.

```
parallel --joblog /tmp/log exit  ::: 1 2 3 0
cat /tmp/log
parallel --resume --joblog /tmp/log exit  ::: 1 2 3 0 0 0
cat /tmp/log
```

Output:

```
Seq Host Starttime      Runtime Send Receive Exitval Signal Command
1   :    1376580069.544 0.008   0    0       1       0      exit 1
2   :    1376580069.552 0.009   0    0       2       0      exit 2
3   :    1376580069.560 0.012   0    0       3       0      exit 3
4   :    1376580069.571 0.005   0    0       0       0      exit 0
```

```
Seq Host Starttime      Runtime Send Receive Exitval Signal Command
1   :    1376580069.544 0.008   0    0       1       0      exit 1
2   :    1376580069.552 0.009   0    0       2       0      exit 2
3   :    1376580069.560 0.012   0    0       3       0      exit 3
4   :    1376580069.571 0.005   0    0       0       0      exit 0
5   :    1376580070.028 0.009   0    0       0       0      exit 0
6   :    1376580070.038 0.007   0    0       0       0      exit 0
```

Note how the start time of the last 2 jobs is clearly different from the first run.

With **--resume-failed** GNU Parallel will re-run the jobs that failed:

```
parallel --resume-failed --joblog /tmp/log exit  ::: 1 2 3 0 0 0
cat /tmp/log
```

Output:

```
Seq Host Starttime      Runtime Send Receive Exitval Signal Command
1   :    1376580069.544 0.008   0    0       1       0      exit 1
2   :    1376580069.552 0.009   0    0       2       0      exit 2
3   :    1376580069.560 0.012   0    0       3       0      exit 3
4   :    1376580069.571 0.005   0    0       0       0      exit 0
5   :    1376580070.028 0.009   0    0       0       0      exit 0
6   :    1376580070.038 0.007   0    0       0       0      exit 0
1   :    1376580154.433 0.010   0    0       1       0      exit 1
2   :    1376580154.444 0.022   0    0       2       0      exit 2
3   :    1376580154.466 0.005   0    0       3       0      exit 3
```

Note how seq 1 2 3 have been repeated because they had exit value different from 0.

--retry-failed does almost the same as **--resume-failed**. Where **--resume-failed** reads the commands from the command line (and ignores the commands in the joblog), **--retry-failed** ignores the command line and reruns the commands mentioned in the joblog.

```
parallel --retry-failed --joblog /tmp/log
cat /tmp/log
```

Output:

```
Seq Host Starttime      Runtime Send Receive Exitval Signal Command
1   :    1376580069.544 0.008   0    0       1       0      exit 1
2   :    1376580069.552 0.009   0    0       2       0      exit 2
3   :    1376580069.560 0.012   0    0       3       0      exit 3
4   :    1376580069.571 0.005   0    0       0       0      exit 0
5   :    1376580070.028 0.009   0    0       0       0      exit 0
6   :    1376580070.038 0.007   0    0       0       0      exit 0
1   :    1376580154.433 0.010   0    0       1       0      exit 1
2   :    1376580154.444 0.022   0    0       2       0      exit 2
3   :    1376580154.466 0.005   0    0       3       0      exit 3
1   :    1376580164.633 0.010   0    0       1       0      exit 1
2   :    1376580164.644 0.022   0    0       2       0      exit 2
```

```
3   :    1376580164.666 0.005   0    0       3       0      exit 3
```

7.9 Termination

7.9.1 Unconditional termination

By default GNU Parallel will wait for all jobs to finish before exiting.

If you send GNU Parallel the **TERM** signal, GNU Parallel will stop spawning new jobs and wait for the remaining jobs to finish. If you send GNU Parallel the **TERM** signal again, GNU Parallel will kill all running jobs and exit.

7.9.2 Termination dependent on job status

For certain jobs, there is no need to continue if one of the jobs fails and has an exit code different from 0. GNU Parallel will stop spawning new jobs with **--halt soon,fail=1**:

```
parallel -j2 --halt soon,fail=1 echo {}\; exit {} ::: 0 0 1 2 3
```

Output:

```
0
0
1
parallel: This job failed:
echo 1; exit 1
parallel: Starting no more jobs. Waiting for 1 jobs to finish.
2
```

With **--halt now,fail=1** the running jobs will be killed immediately:

```
parallel -j2 --halt now,fail=1 echo {}\; exit {} ::: 0 0 1 2 3
```

Output:

```
0
0
1
parallel: This job failed:
echo 1; exit 1
```

If **--halt** is given a percentage this percentage of the jobs must fail before GNU Parallel stops spawning more jobs:

```
parallel -j2 --halt soon,fail=20% echo {}\; exit {} \
  ::: 0 1 2 3 4 5 6 7 8 9
```

Output:

```
0
1
parallel: This job failed:
echo 1; exit 1
2
parallel: This job failed:
echo 2; exit 2
parallel: Starting no more jobs. Waiting for 1 jobs to finish.
3
parallel: This job failed:
echo 3; exit 3
```

If you are looking for success instead of failures, you can use **`success`**. This will finish as soon as the first job succeeds:

```
parallel -j2 --halt now,success=1 echo {}\; exit {} ::: 1 2 3 0 4 5 6
```

Output:

```
1
2
3
0
parallel: This job succeeded:
echo 0; exit 0
```

If you do not care about the exit value, but you just want the first 3 to complete, you can use done=3:

```
parallel -j2 --halt now,done=3 sleep {}\;echo {}\; exit {} \
   ::: 1 2 3 0 4 5 6
```

Output:

```
parallel: This job finished:
sleep 1;echo 1; exit 1
2
parallel: This job finished:
sleep 2;echo 2; exit 2
0
parallel: This job finished:
sleep 0;echo 0; exit 0
```

7.10 Retry failing commands

GNU Parallel can retry the command with **`--retries`**. This is useful if a command fails for unknown reasons now and then.

```
parallel -k --retries 3 \
  'echo tried {} >>/tmp/runs; echo completed {}; exit {}' ::: 1 2 0
```

```
cat /tmp/runs
```

Output:

```
completed 1
completed 2
completed 0

tried 1
tried 2
tried 1
tried 2
tried 1
tried 2
tried 0
```

Note how job 1 and 2 were tried 3 times, but 0 was not retried because it had exit code 0.

When used with remote execution (see chapter 8 Remote execution) the job will be retried on another server if possible.

7.10.1 Termination signals

Using **`--termseq`** you can control which signals are sent when killing children. Normally children will be killed by sending them **`SIGTERM`**, waiting 200 ms, then another **`SIGTERM`**, waiting 100 ms, then another **`SIGTERM`**, waiting 50 ms, then a **`SIGKILL`**, finally waiting 25 ms before giving up. It looks like this:

```
show_signals() {
  perl -e 'for(keys %SIG) {
    $SIG{$_} = eval "sub { print \"Got $_\\n\"; }";
  }
  while(1){sleep 1}'
}
export -f show_signals
echo | parallel --termseq TERM,200,TERM,100,TERM,50,KILL,25 \
  -u --timeout 1 show_signals
```

Output:

```
Got TERM
Got TERM
Got TERM
```

Or just:

```
echo | parallel -u --timeout 1 show_signals
```

Output: Same as above.

You can change this to **SIGINT**, **SIGTERM**, **SIGKILL**:

```
echo | parallel --termseq INT,200,TERM,100,KILL,25 \
  -u --timeout 1 show_signals
```

Output:

```
Got INT
Got TERM
```

The **SIGKILL** does not show because it cannot be caught, and thus the child dies.

7.11 Limit the resources

GNU Parallel can run the jobs with a nice value. This will work both locally and remotely.

```
parallel --nice 17 echo this is being run with nice -n ::: 17
```

Output:

```
this is being run with nice -n 17
```

To avoid overloading systems GNU Parallel can look at the system load before starting another job:

```
parallel --load 100% echo load is less than {} job per CPU ::: 1
```

Output:

```
[when the load is less than the number of CPU cores]
load is less than 1 job per CPU
```

GNU Parallel can also check if the system is swapping.

```
parallel --noswap echo the system is not swapping ::: now
```

Output:

```
[when then system is not swapping]
the system is not swapping now
```

Some jobs need a lot of memory, and should only be started when there is enough memory free. Using **--memfree** GNU Parallel can check if there is enough memory free. Additionally, GNU Parallel will kill off the youngest job if the memory free falls below 50% of the size. The killed job will put back on the queue and retried later if **--retries** is given.

```
parallel --memfree 1G --retries 5 echo More than 1 GB is ::: free
```

7.11.1 Make your own limitation

With **--limit** you can make your own limitations like **--memfree** and **--load**. You just need to make a program that returns:

Exit value	Meaning
0	Below limit. Start another job.
1	Over limit. Start no jobs.
2	Way over limit. Kill the youngest job.

There are 3 predefined commands:

Command	Meaning
io *n*	Limit for I/O. The amount of disk I/O will be computed as a value 0-100, where 0 is no I/O and 100 is at least one disk is 100% saturated. *n* sets the limit of when **io** should return 1.
mem *n*	Similar to **--memfree**
load *n*	Similar to **--load**

Examples:

```
parallel --limit "io 10" echo ::: less than 10% disk I/O
parallel --limit "mem 10g" echo ::: more than 10G free
parallel --limit "load 3" echo ::: less than 3 procs running
```

8

Remote execution

I should start a consultancy that makes Hadoop clusters 100x faster
by replacing them with GNU parallel + gnutools
-- Chris Allen bitemyapp@twitter

GNU Parallel can run jobs on remote servers. It uses **`ssh`** to communicate with the remote machines.

In the following, we assume you have access 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; ssh-copy-id $SERVER2
```

and using an empty passphrase.

8.1 Sshlogin

The most basic sshlogin is **`-S`** *host*/**`--sshlogin`** *host*:

```
parallel --sshlogin $SERVER1 echo running on ::: server1
```

Output:

```
running on server1
```

To use a different username prepend the server with *username@*:

```
parallel -S username@$SERVER1 echo running on ::: username@server1
```

Output:

```
running on username@server1
```

The special sshlogin `:` is the local machine:

```
parallel -S : echo running on ::: the_local_machine
```

Output:

```
running on the_local_machine
```

8.1.1 SSH command to use

If `ssh` is not in $PATH it can be prepended to $SERVER1:

```
parallel -S '/usr/bin/ssh '$SERVER1 echo custom ::: ssh
```

Output:

```
custom ssh
```

The `ssh` command can also be given using `--ssh`:

```
parallel --ssh /usr/bin/ssh -S $SERVER1 echo custom ::: ssh
```

or by setting `$PARALLEL_SSH`:

```
export PARALLEL_SSH=/usr/bin/ssh
parallel -S $SERVER1 echo custom ::: ssh
```

8.1.2 Multiple servers

Several servers can be given using multiple `-S`:

```
parallel -S $SERVER1 -S $SERVER2 echo ::: running on more hosts
```

Output (the order may be different):

```
running
on
more
hosts
```

Or they can be separated by `,`:

```
parallel -S $SERVER1,$SERVER2 echo ::: running on more hosts
```

Output: Same as above.

Or newline:

```
# This gives a \n between $SERVER1 and $SERVER2
SERVERS="`echo $SERVER1; echo $SERVER2`"
parallel -S "$SERVERS" echo ::: running on more hosts
```

They can also be read from a file (replace *user@* with the user on **$SERVER2**):

```
echo $SERVER1 > nodefile
# Force special ssh-command, username
echo /usr/bin/ssh user@$SERVER2 >> nodefile
parallel --sshloginfile nodefile echo ::: running on more hosts
```

Output: Same as above.

Every time a job finished, the **--sshloginfile** will be re-read, so it is possible to both add and remove hosts while running.

The special **--sshloginfile ..** reads from **~/.parallel/sshloginfile**.

To force GNU Parallel to treat a server having a given number of CPU cores prepend the number of core followed by **/** to the sshlogin:

```
parallel -S 4/$SERVER1 echo force {} CPUs on server ::: 4
```

Output:

```
force 4 CPUs on server
```

8.1.3 Divide servers into groups

Servers can be put into groups by prepending *@groupname* to the server and the group can then be selected by appending *@groupname* to the argument if using **--hostgroup**:

```
parallel --hostgroup -S @grp1/$SERVER1 -S @grp2/$SERVER2 echo {} \
  ::: run_on_grp1@grp1 run_on_grp2@grp2
```

Output:

```
run_on_grp1
run_on_grp2
```

A host can be in multiple groups by separating the groups with **+**, and you can force GNU Parallel to limit the groups on which the command can be run with **-S** *@groupname*:

```
parallel -S @grp1 -S @grp1+grp2/$SERVER1 -S @grp2/$SERVER2 echo {} \
  ::: run_on_grp1 also_grp1
```

Output:

```
run_on_grp1
also_grp1
```

8.1.3.1 Host group defined by argument

The host group can also be defined by the argument by appending `@` and the sshlogin to the argument:

```
parallel --hostgroup echo {} \
  ::: run_on_server1@$SERVER1 run_on_server2@$SERVER2
```

Output:

```
run_on_server1
run_on_server2
```

8.2 Transfer files

GNU Parallel can transfer the files to be processed to the remote host. It does that with **`--transferfile`** using **`rsync`**.

```
echo This is input_file > input_file
parallel -S $SERVER1 --transferfile {} cat ::: input_file
```

Output:

```
This is input_file
```

You can control the options to **`rsync`** with **`--rsync-opts`** or **`$PARALLEL_RSYNC_OPTS`**. Default is: **`-rlDzR`**

If the files are processed into another file, the resulting file can be returned using **`--return`**:

```
echo This is input_file > input_file
parallel -S $SERVER1 --transferfile {} --return {}.out \
  cat {} ">"{}.out ::: input_file
cat input_file.out
```

Output: Same as above.

To remove the input and output file on the remote server use **`--cleanup`**:

```
echo This is input_file > input_file
parallel -S $SERVER1 --transferfile {} --return {}.out --cleanup \
  cat {} ">"{}.out ::: input_file
cat input_file.out
```

Output: Same as above.

There is a shorthand for **--transferfile {} --return *foo* --cleanup** called **--trc *foo***:

```
echo This is input_file > input_file
parallel -S $SERVER1 --trc {}.out cat {} ">"{}.out ::: input_file
cat input_file.out
```

Output: Same as above.

Some jobs need a common database for all jobs. GNU Parallel can transfer that using **--basefile** which will transfer the file before the first job:

```
echo common data > common_file
parallel --basefile common_file -S $SERVER1 \
  cat common_file\; echo {} ::: foo
```

Output:

```
common data
foo
```

To remove it from the remote host after the last job use **--cleanup**.

Because GNU Parallel uses **rsync** for the transferring, you can use **/./** to specify which dir you want the file to be relative to. This will transfer **foo/bar/file** to **~/bar/file** on **$SERVER1**:

```
parallel -S $server1 --transfer wc {//} ::: foo/./bar/file
```

If you set **--workdir** (see 8.3 Working dir) then the transfer will be relative to that dir.

8.3 Working dir

The default working dir on the remote machines is the login dir. This can be changed with **--workdir** *mydir*.

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

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

The special *mydir* value **.** uses the current working dir. If the current working dir is beneath your home dir, the value **.** is treated as the relative path to your home dir. This means that if

your home dir is different on the remote computers (e.g. if your login is different) the relative path will still be relative to your home dir.

```
parallel -S $SERVER1 pwd ::: ""
parallel --workdir . -S $SERVER1 pwd ::: ""
parallel --workdir ... -S $SERVER1 pwd ::: ""
```

Output:

```
[the login dir on $SERVER1]
[current dir relative on $SERVER1]
[a dir in ~/.parallel/tmp/...]
```

8.4 Avoid overloading sshd

If many jobs are started on the same server, **sshd** can be overloaded. GNU Parallel can insert a delay between each job run on the same server:

```
parallel -S $SERVER1 --sshdelay 0.2 echo ::: 1 2 3
```

Output (the order may be different):

```
1
2
3
```

sshd will be less overloaded if using **--controlmaster**, which will multiplex **ssh** connections:

```
parallel --controlmaster -S $SERVER1 echo ::: 1 2 3
```

Output: Same as above.

8.5 Ignore hosts that are down

In clusters with many hosts, a few of them are often down. GNU Parallel can ignore those hosts. In this case, the host 173.194.32.46 is down:

```
parallel --filter-hosts -S 173.194.32.46,$SERVER1 echo ::: bar
```

Output:

```
bar
```

8.6 Run the same commands on all hosts

GNU Parallel can run the same command on all the hosts:

```
parallel --onall -S $SERVER1,$SERVER2 echo ::: foo bar
```

Output (the order may be different):

```
foo
bar
foo
bar
```

Often you will just want to run a single command on all hosts without arguments. **`--nonall`** is a no argument **`--onall`**:

```
parallel --nonall -S $SERVER1,$SERVER2 echo foo bar
```

Output:

```
foo bar
foo bar
```

When **`--tag`** is used with **`--nonall`** and **`--onall`** the **`--tagstring`** is the host:

```
parallel --nonall --tag -S $SERVER1,$SERVER2 echo foo bar
```

Output (the order may be different):

```
$SERVER1 foo bar
$SERVER2 foo bar
```

`--jobs` sets the number of servers to log in to in parallel.

8.7 Transfer environment variables and functions

`env_parallel` is a shell function that transfers all aliases, functions, variables, and arrays. You active it by running:

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

Replace **`bash`** with the shell you use.

Now you can use **`env_parallel`** instead of **`parallel`** and still have your environment:

```
alias myecho=echo
myvar="Joe's var is"
env_parallel -S $SERVER1 'myecho $myvar' ::: green
```

Output:

```
Joe's var is green
```

The disadvantage is that if your environment is huge **`env_parallel`** will fail.

If **env_parallel** fails, you can use **--env** to tell GNU Parallel which names to transfer to the remote system:

```
MYVAR='foo bar'
env_parallel --env MYVAR -S $SERVER1 echo '$MYVAR' ::: baz
```

Output:

```
foo bar baz
```

This works for functions, too, if your shell is Bash:

```
# This only works in Bash
my_func() {
  echo in my_func $1
}
env_parallel --env my_func -S $SERVER1 my_func ::: baz
```

Output:

```
in my_func baz
```

Instead of naming the variables individually, GNU Parallel can record defined names in a clean shell and only transfer names that are not on that list. GNU Parallel records the names to ignore in **~/.parallel/ignored_vars** by running:

```
env_parallel –record-env
cat ~/.parallel/ignored_vars
```

Output:

```
[list of variables to ignore - including $PATH and $HOME]
```

You only need to do this once.

After this you can use **--env _** to tell GNU Parallel to transfer every name that is not ignored in **~/.parallel/ignored_vars**:

```
foo_func() {
  foo_alias $foo_var functions, ${foo_array[*]} are all "$@"
}
foo_var='variables,'
foo_array=(and arrays)
alias foo_alias='echo aliases,'

env_parallel --env _ -S $SERVER1 foo_func ::: copied
```

Output:

```
aliases, variables, functions, and arrays are all copied
```

8.8 Show what is actually run

--verbose will show the command that would be run on the local machine.

When using **--nice**, **--pipepart**, or when a job is run on a remote machine, the command is wrapped with helper scripts. **-vv** shows all of this.

```
parallel -vv --pipepart --block 1M wc :::: num30000
```

Output:

```
<num30000 perl -e 'while(@ARGV) { sysseek(STDIN,shift,0) || die;
$left = shift; while($read = sysread(STDIN,$buf, ($left > 131072
? 131072 : $left))){ $left -= $read; syswrite(STDOUT,$buf); } }'
0 0 0 168894 | (wc)
30000   30000  168894
```

You will normally not need to understand the code, but if you get unexpected results, it can be useful to know what is *actually* being run.

When the command gets more complex, the output is so hard to read, that it is only useful for debugging:

```
my_func3() {
  echo in my_func $1 > $1.out
}
export -f my_func3
parallel -vv --workdir ... --nice 17 --env _ --trc {}.out \
  -S $SERVER1 my_func3 {} ::: abc-file
```

The output will be similar to:

```
[1 kb of gobbly goob]
```

9

Pipe mode

I continuously find myself forgetting about GNU parallel,
especially --pipe, which solves so many problems so elegantly
-- Andrew Montalenti amontalenti@twitter

Instead of putting values in a command template GNU Parallel can pass stdin (standard input) on a pipe to commands.

> The **`--pipe`** functionality puts GNU Parallel in a different mode: Instead of treating the data on stdin (standard input) as arguments for a command to run, the data will be sent to stdin (standard input) of the command.
>
> The typical situation is:

```
command_A | command_B | command_C
```

> where **`command_B`** is slow, and you want to speed up **`command_B`** by running many of these in parallel.

You will need the test files from chapter 3.

9.1 Block size

> By default, GNU Parallel will start an instance of **`command_B`**, read a block of 1 MB, find the closest record, and pass that chunk to the instance. Then start another instance, read another block, find the closest record, and pass that chunk to the second instance.

```
cat num1000000 | parallel --pipe wc
```

Output (the order may be different):

```
165668  165668 1048571
149797  149797 1048579
149796  149796 1048572
149797  149797 1048579
149797  149797 1048579
149796  149796 1048572
 85349   85349  597444
```

The size of the chunk is not exactly 1 MB because GNU Parallel only passes full lines - never half a line, thus the block size is only 1 MB on average. You can change the block size to 2 MB with **`--block`**:

```
cat num1000000 | parallel --pipe --block 2M wc
```

Output (the order may be different):

```
315465  315465 2097150
299593  299593 2097151
299593  299593 2097151
 85349   85349  597444
```

GNU Parallel treats each line as a record. If the order of records is unimportant (e.g. you need all lines processed, but you do not care which is processed first), then you can use **`--round-robin`**. Without **`--round-robin`** GNU Parallel will start a command per block; with **`--round-robin`** only the requested number of jobs will be started (**`--jobs`**). The records will then be distributed between the running jobs:

```
cat num1000000 | parallel --pipe -j4 --round-robin wc
```

Output will be similar to:

```
149797  149797 1048579
299593  299593 2097151
315465  315465 2097150
235145  235145 1646016
```

One of the 4 instances got a single chunk, 2 instances got 2 full chunks each, and one instance got 1 full and 1 partial chunk.

`--round-robin` gives the chunk to the first process that is ready. You can force the order of the chunks to be strictly one to each process by using **`--keep-order`**:

```
cat num1000000 | parallel --pipe -j4 --keep-order --round-robin wc
```

Output:

```
315464  315464 2097143
299592  299592 2097144
235148  235148 1646037
```

```
149796  149796 1048572
```

9.2 Records

GNU Parallel sees the input as records. The default record is a single line.

Using **-N140000** GNU Parallel will read 140000 records at a time:

```
cat num1000000 | parallel --pipe -N140000 wc
```

Output (the order may be different):

```
140000  140000  868895
140000  140000  980000
140000  140000  980000
140000  140000  980000
140000  140000  980000
140000  140000  980000
140000  140000  980000
 20000   20000  140001
```

Note how that the last job could not get the full 140000 lines, but only 20000 lines.

If a record is 75 lines **-L** can be used:

```
cat num1000000 | parallel --pipe -L75 wc
```

Output (the order may be different):

```
165600  165600 1048095
149850  149850 1048950
149775  149775 1048425
149775  149775 1048425
149850  149850 1048950
149775  149775 1048425
 85350   85350  597450
25      25     176
```

Note how GNU Parallel still reads a block of around 1 MB, but instead of passing full lines to **wc** it passes full 75 lines at a time. This, of course, does not hold for the last job (which in this case got 25 lines).

9.3 Record separators

GNU Parallel uses separators to determine where two records split.

--recstart gives the string that starts a record; **--recend** gives the string that ends a record. The default is **--recend '\n'** (newline) and **--recstart ""**.

If both **--recend** and **--recstart** are given, then the record will only split if the recend string is immediately followed by the recstart string.

Let us assume we have the input:

```
/foo, bar/, /baz, qux/,
```

We want to split that into:

```
/foo, bar/,
/baz, qux/,
```

If we set **--recend** to **', '**:

```
echo /foo, bar/, /baz, qux/, | \
  parallel -kN1 --recend ', ' --pipe echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
/foo, END
JOB2
bar/, END
JOB3
/baz, END
JOB4
qux/,
END
```

This is not exactly what we wanted. The problem is that the records contain ', '.

Here **--recstart** is set to **/**:

```
echo /foo, bar/, /baz, qux/, | \
  parallel -kN1 --recstart / --pipe echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
/foo, barEND
JOB2
/, END
JOB3
/baz, quxEND
JOB4
/,
END
```

This is also no good. Here both **--recend** and **--recstart** are set:

```
echo /foo, bar/, /baz, qux/, | \
  parallel -kN1 --recend ', ' --recstart / --pipe \
  echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
/foo, bar/, END
JOB2
/baz, qux/,
END
```

Note the difference between setting one string and setting both strings.

With **`--regexp`** the **`--recend`** and **`--recstart`** will be treated as regular expressions:

```
echo foo,bar,_baz,__qux | \
  parallel -kN1 --regexp --recend ,_* --pipe \
  echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
foo,END
JOB2
bar,_END
JOB3
baz,__END
JOB4
qux
END
```

GNU Parallel can remove the record separators with **`--remove-rec-sep`**/**`--rrs`**:

```
echo foo,bar,_baz,__qux | \
  parallel -kN1 --rrs --regexp --recend ,_* --pipe \
  echo JOB{#}\;cat\;echo END
```

Output:

```
JOB1
fooEND
JOB2
barEND
JOB3
bazEND
JOB4
qux
END
```

9.4 Header

If the input data has a header, the header can be repeated for each job by matching the header with **--header**. If headers start with **%** you can do this:

```
cat num_%header | \
  parallel --header '(%.*\n)*' --pipe -N3 echo JOB{#}\;cat
```

Output (the order may be different):

```
JOB1
%head1
%head2
1
2
3
JOB2
%head1
%head2
4
5
6
JOB3
%head1
%head2
7
8
9
JOB4
%head1
%head2
10
```

If the header is 2 lines, **--header** 2 will work:

```
cat num_%header | parallel --header 2 --pipe -N3 echo JOB{#}\;cat
```

Output: Same as above.

9.5 Fixed length records

Fixed length records can be processed by setting **--recend ''** and **--block *recordsize***.

A header of size *n* can be processed with **--header .{*n*}**.

Here is how to process a file with a 4-byte header and a 3-byte record size:

```
cat fixedlen | parallel --pipe --header .{4} --block 3 --recend '' \
  'echo start; cat; echo'
```

Output:

```
start
HHHHAAA
start
HHHHCCC
start
HHHHBBB
```

9.6 Programs not reading from stdin

Some programs cannot read from stdin but must read from a file.

9.6.1 --cat

Using `--cat` GNU Parallel will create a temporary file that can be used for the command. GNU Parallel will remove the file when the program finishes. Let us assume that `wc` needs a file like `wc file`:

```
cat num1000000 | parallel --pipe --cat wc {}
```

Output similar to:

```
149796  149796 1048572 /tmp/par1jBpC
165668  165668 1048571 /tmp/parFXwWw
149796  149796 1048572 /tmp/parDLbDu
149796  149796 1048572 /tmp/parFRxVf
149796  149796 1048572 /tmp/paryUqkT
 85352   85352  597465 /tmp/parWvfXe
149796  149796 1048572 /tmp/par0Yl2R
```

GNU Parallel generates the `/tmp/parXXXXX` files, puts a chunk of data into each, and runs `wc` on each of them before removing them again.

9.6.2 --fifo

`--cat` is rather slow because data first has to be stored on disk before it can be read by `wc`. If the program can read from a FIFO (also known as a named pipe), then GNU Parallel can avoid storing the temporary files on disk.

```
cat num1000000 | parallel --pipe --fifo wc {}
```

Output similar to:

```
149796  149796 1048572 /tmp/parr5MKa
165668  165668 1048571 /tmp/parJWpuV
149796  149796 1048572 /tmp/parJRMEJ
```

```
149796  149796 1048572 /tmp/parbmm1K
149796  149796 1048572 /tmp/parT6QQf
149796  149796 1048572 /tmp/partfyPz
 85352   85352  597465 /tmp/parYybjk
```

The program, however, needs to read the whole file from start to finish. If it only reads the first part, GNU Parallel will block.

9.7 Use --pipepart for high performance

`--pipe` is not very efficient. It maxes out at around 500 MB/s. **`--pipepart`** can easily deliver more than 5 GB/s, but it has a few limitations. The input has to be a normal file or a block device (not a pipe or a fifo) given by **`-a`** or **`::::`** and **`-L`**/**`-l`**/**`-N`** do not work. **`--recend`** and **`--recstart`**, however, *do* work, and records can often be split on that alone.

```
parallel --pipepart -a num1000000 --block 3m wc
```

Output (the order may be different):

```
 444443  444444 3000002
 428572  428572 3000004
 126985  126984  888890
```

By giving **`--block`** a negative number it is interpreted as the number of blocks each job slot should have. So this will run 3*5 = 15 jobs in total:

```
parallel --pipepart -a num1000000 --block -3 -j5 wc
```

This is an efficient alternative to **`--round-robin`** because data is never read by GNU Parallel, but you can still have very few job slots process a large amount of data.

In addition to that, you can use **`--keep-order`** to get the output in the same order as the input. This cannot be done with **`--round-robin`** because the input is mixed.

9.8 Duplicate all input using --tee

With **`--tee`** you can duplicate the same input to a number of jobs:

```
seq 30 | parallel -v --pipe --tee --tag grep {} ::: 4 5 6
```

Output:

```
4       grep 4
4       4
4       14
```

```
4       24
5       grep 5
5       5
5       15
5       25
6       grep 6
6       6
6       16
6       26
```

10

Miscellaneous features

GNU Parallel never ceases to amaze me.
-- Tim Hopper tdhopper@twitter

A few of GNU Parallel's options are not related to the 6 main areas.

10.1 Shebang

10.1.1 Input data and parallel command in the same file

GNU Parallel is often called as this:

```
cat input_file | parallel command
```

With **`--shebang`** the *input_file* and **`parallel`** can be combined into the same script.

UNIX shell scripts start with a shebang line like this:

```
#!/bin/bash
```

GNU Parallel can do that, too. With **`--shebang`** the arguments can be listed in the file. The Parallel command is the first line of the script:

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

foo
bar
baz
```

Output (the order may be different):

```
foo
bar
baz
```

10.1.2 Parallelize existing scripts with --shebang-wrap

GNU Parallel is often called as this:

```
cat input_file | parallel command
parallel command ::: foo bar
```

If **command** is a script, Parallel can be combined into a single file so this will run the script in parallel:

```
cat input_file | command
command foo bar
```

This **perl** script **perl_echo** works like **echo**:

```
#!/usr/bin/perl

print "@ARGV\n"
```

It can be called as this:

```
parallel perl_echo ::: foo bar
```

By changing the **#!**-line it can be run in parallel:

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

print "@ARGV\n"
```

Thus this will work:

```
perl_echo foo bar
```

Output (the order may be different):

```
foo
bar
```

This technique can be used for:

Perl:

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

print "Arguments @ARGV\n";
```

Python:

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

```
import sys
print 'Arguments', str(sys.argv)
```

Bash/sh/zsh/Korn shell:

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

echo Arguments "$@"
```

csh/tcsh:

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

echo Arguments "$argv"
```

Tcl:

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

puts "Arguments $argv"
```

R:

```
#!/usr/bin/parallel --shebang-wrap /usr/bin/Rscript --vanilla --slave

args <- commandArgs(trailingOnly = TRUE)
print(paste("Arguments ",args))
```

GNUplot:

```
#!/usr/bin/parallel --shebang-wrap ARG={} /usr/bin/gnuplot

print "Arguments ", system('echo $ARG')
```

Ruby:

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

print "Arguments "
puts ARGV
```

Octave:

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

printf ("Arguments");
arg_list = argv ();
for i = 1:nargin
  printf (" %s", arg_list{i});
endfor
printf ("\n");
```

Common LISP:

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

```
(format t "~&~S~&" 'Arguments)
(format t "~&~S~&" *args*)
```

PHP:

```
#!/usr/bin/parallel --shebang-wrap /usr/bin/php
<?php
echo "Arguments";
foreach(array_slice($argv,1) as $v) {
  echo " $v";
}
echo "\n";
?>
```

Node.js:

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

var myArgs = process.argv.slice(2);
console.log('Arguments ', myArgs);
```

LUA:

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

io.write "Arguments"
for a = 1, #arg do
  io.write(" ")
  io.write(arg[a])
end
print("")
```

C#:

```
#!/usr/bin/parallel --shebang-wrap ARGV={} /usr/bin/csharp

var argv = Environment.GetEnvironmentVariable("ARGV");
print("Arguments "+argv);
```

10.2 Semaphore

GNU Parallel can work as a counting semaphore. This is slower and less efficient than its normal mode.

A counting semaphore is like a row of toilets. People needing a toilet can use any toilet, but if there are more people than toilets, they will have to wait for one of the toilets to become available.

An alias for **`parallel --semaphore`** is **`sem`**.

sem will follow a person to the toilets, wait until a toilet becomes available, leave the person in the toilet and exit.

sem --fg will follow a person to the toilets, wait until a toilet becomes available, stay with the person in the toilet and exit when the person exits.

sem --wait will wait for all persons to leave the toilets.

sem does not have a queue discipline, so the next person is chosen randomly.

-j sets the number of toilets.

10.2.1 Mutex

The default is to have only one toilet (this is called a mutex). The program is started in the background and **sem** exits immediately. Use **--wait** to wait for all **sem**s to finish:

```
sem 'sleep 1; echo The first finished' &&           \
echo The first is now running in the background &&  \
sem 'sleep 1; echo The second finished' &&          \
echo The second is now running in the background
sem --wait
```

Output:

```
The first is now running in the background
The first finished
The second is now running in the background
The second finished
```

The command can be run in the foreground with **--fg**, which will only exit when the command completes:

```
sem --fg 'sleep 1; echo The first finished' &&        \
echo The first finished running in the foreground &&  \
sem --fg 'sleep 1; echo The second finished' &&       \
echo The second finished running in the foreground
sem --wait
```

Output:

```
The first finished
The first finished running in the foreground
The second finished
The second finished running in the foreground
```

The difference between this and just running the command is that a mutex is set, so if other **sem**s were running in the background only one command would run at a time.

To control which semaphore is used, use **--semaphorename**/**--id**. Run this in one terminal:

```
sem --id my_id -u 'echo First started; sleep 10; echo First done'
```

and simultaneously this in another terminal:

```
sem --id my_id -u 'echo Second started; sleep 10; echo Second done'
```

Note how the second will only be started when the first has finished.

10.2.2 Counting semaphore

A mutex is like having a single toilet: When it is in use everyone else will have to wait. A counting semaphore is like having multiple toilets: Several people can use the toilets, but when they all are in use, everyone else will have to wait.

sem can emulate a counting semaphore. Use **--jobs** to set the number of toilets like this:

```
sem --jobs 3 --id my_id -u 'echo Start 1; sleep 5; echo 1 done' &&    \
  sem --jobs 3 --id my_id -u 'echo Start 2; sleep 6; echo 2 done' &&  \
  sem --jobs 3 --id my_id -u 'echo Start 3; sleep 7; echo 3 done' &&  \
  sem --jobs 3 --id my_id -u 'echo Start 4; sleep 8; echo 4 done' &&  \
  sem --wait --id my_id
```

Output:

```
Start 1
Start 2
Start 3
1 done
Start 4
2 done
3 done
4 done
```

10.2.3 Semaphore with timeout

With **--semaphoretimeout** you can force running the command anyway after a period (positive number) or give up (negative number):

```
sem --id foo -u 'echo Slow started; sleep 5; echo Slow ended' &&         \
  sem --id foo --semaphoretimeout 1 'echo Forced running after 1 sec' && \
  sem --id foo --semaphoretimeout -2 'echo Give up after 2 secs'
sem --id foo --wait
```

Output:

```
Slow started
parallel: Warning: Semaphore timed out. Stealing the semaphore.
Forced running after 1 sec
parallel: Warning: Semaphore timed out. Exiting.
Slow ended
```

Note how the 'Give up' was not run.

10.3 Informational

GNU Parallel has some options to give information about the configuration.

`--help` will print a summary of the most important options:

```
parallel --help
```

Output:

```
Usage:

parallel [options] [command [arguments]] < list_of_arguments
parallel [options] [command [arguments]] (::: arguments|:::: 
argfile(s))...
cat ... | parallel --pipe [options] [command [arguments]]

-j n            Run n jobs in parallel
-k              Keep same order
-X              Multiple arguments with context replace
--colsep regexp Split input on regexp for positional replacements
{} {.} {/} {/.} {#} {%} {= perl code =} Replacement strings
{3} {3.} {3/} {3/.} {=3 perl code =}    Positional replacement strings
With --plus:    {} = {+/}/{/} = {.}.{+.} = {+/}/{/.}.{+.} = {..}.{+..} = 
{+/}/{/..}.{+..} = {...}.{+...} = {+/}/{/...}.{+...}

-S sshlogin     Example: foo@server.example.com
--slf ..        Use ~/.parallel/sshloginfile as the list of sshlogins
--trc {}.bar    Shorthand for --transfer --return {}.bar --cleanup
--onall         Run the given command with argument on all sshlogins
--nonall        Run the given command with no arguments on all sshlogins

--pipe          Split stdin (standard input) to multiple jobs.
--recend str    Record end separator for --pipe.
--recstart str  Record start separator for --pipe.

See 'man parallel' for details

Academic tradition requires you to cite works you base your article on.
When using programs that use GNU Parallel to process data for 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.
```

When asking for help, always report the full output of this:

```
parallel --version
```

Output:

```
GNU parallel 20160323
Copyright (C) 2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017
Ole Tange and Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
GNU parallel comes with no warranty.

Web site: http://www.gnu.org/software/parallel

When using programs that use GNU Parallel to process data for publication
please cite as described in 'parallel --citation'.
```

In scripts **`--minversion`** can be used to ensure the user has at least this version:

```
parallel --minversion 20130722 && \
  echo Your version is at least 20130722.
```

Output:

```
20160322
Your version is at least 20130722.
```

If you are using GNU Parallel for research the BibTeX citation can be generated using **`--citation`**:

```
parallel --citation
```

Output:

```
Academic tradition requires you to cite works you base your article on.
When using programs that use GNU Parallel to process data for publication
please cite:

@article{Tange2011a,
title = {GNU Parallel - The Command-Line Power Tool},
author = {O. Tange},
address = {Frederiksberg, Denmark},
journal = {;login: The USENIX Magazine},
month = {Feb},
number = {1},
volume = {36},
url = {http://www.gnu.org/s/parallel},
```

```
year = {2011},
pages = {42-47},
doi = {10.5281/zenodo.16303}
}

(Feel free to use \nocite{Tange2011a})

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.

If you send a copy of your published article to tange@gnu.org, it will be
mentioned in the release notes of next version of GNU Parallel.
```

With **`--max-line-length-allowed`** GNU Parallel will report the maximal size of the command line:

```
parallel --max-line-length-allowed
```

Output (may vary on different systems):

```
131071
```

`--number-of-cpus` and **`--number-of-cores`** run system specific code to determine the number of CPUs and CPU cores on the system. On unsupported platforms they will return 1:

```
parallel --number-of-cpus
parallel --number-of-cores
```

Output (may vary on different systems):

```
4
64
```

10.4 Profiles

The defaults for GNU Parallel can be changed system-wide by putting the command line options in **`/etc/parallel/config`**. They can be changed for a user by putting them in **`~/.parallel/config`**.

Profiles work the same way, but have to be referred to with **`--profile`**/**`-J`**:

```
echo '--nice 17' > ~/.parallel/nicetimeout
echo '--timeout 300%' >> ~/.parallel/nicetimeout
parallel --profile nicetimeout echo ::: A B C
```

Output:

```
A
B
```

```
C
```

- Profiles can be combined:

```
echo '-vv --dry-run' > ~/.parallel/dryverbose
parallel --profile dryverbose --profile nicetimeout echo ::: A B C
```

- Output:

```
echo A
echo B
echo C
```

11

GNU Free Document License

An ode to GNU parallel
An ode to GNU parallel
An ode to GNU parallel
An ode to GNU parallel
An ode to GNU parallel
An ode to GNU parallel
-- Adam Stuckert PoisonEcology@twitter

Version 1.3, 3 November 2008

0. Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to

text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires

special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/licenses/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the

option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. Relicensing

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Addendum: how to use this license for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
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, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU Free
Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover
Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

After chapter 2 there is no need to read the chapters in sequence: If you need to know how to control the output go right ahead and skip to chapter 6.

The book is written as a 5-in-1 book: You can read it as a beginner, as an intermediate, as an advanced user, as an expert user, or a developer to get all the details. The marking in the border will tell you which audience the section is written for.

Read this if you are level 1.

Read this if you are level 2.

Read this if you are level 3.

Read this if you are level 4.

Read this if you are level 5.

For instance, you do not need to have read anything at level 4 to understand the text at level 3.

Additionally, you do not have to be at the same level in each chapter. Maybe you need advanced knowledge on controlling the execution (chapter 7), while you never use the remote execution (chapter 8), and only use the basic features of `--pipe` (chapter 9).

You are expected to know basic UNIX commands: `ls`, `wc`, `cat`, `pwd`, `seq`, `sleep`, `echo`, `wget`, `printf`, `rm`, and `ssh`. If any of those are new to you, you should type `man programname` and familiarize yourself with those.

You are expected to know that `/`
at the end of the line means the line continues (but that there was no more space on the paper).

If you also have a basic understanding of what `emacs`, `vi`, `perl`, `mkfifo`, `rsync`, `alias`, and `export` do, then you will have a much easier time understanding the book.

1

How to read this book

There are so few utilities/tools as elegant and amazingly useful
across a wide area of needs as GNU parallel
-- hrbrcoin hrbrmstr@twitter

Are you the kind of person who flicks through a book from behind? This book is best read from the other end.

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.

Chapter 2 will get you started with the basics in 15 minutes. It will introduce you to the basic concepts of GNU Parallel and will show you enough that you can run basic commands in parallel. This will be enough for many tasks.

GNU Parallel has 6 major areas:

- Chapter 4 Input sources
- Chapter 5 Build the command line
- Chapter 6 Control the output
- Chapter 7 Control the execution
- Chapter 8 Remote execution
- Chapter 9 Pipe mode

On top of this, there are a few miscellaneous features

- Chapter 10 Miscellaneous features

GNU Parallel 2018

First edition

Published by: Ole Tange
http://ole.tange.dk
https://orcid.org/0000-0002-6345-1437

Cover: GNU Parallel's logo is inspired by the café wall illusion

DOI: http://dx.doi.org/10.5281/zenodo.1146014

ISBN: 978-1-387-50988-1

GNU Parallel 2018

Ole Tange