Design of GNU Parallel

This document describes design decisions made in the development of GNU **parallel** and the reasoning behind them. It will give an overview of why some of the code looks the way it does, and will help new maintainers understand the code better.

One file program

GNU **parallel** is a Perl script in a single file. It is object oriented, but contrary to normal Perl scripts each class is not in its own file. This is due to user experience: The goal is that in a pinch the user will be able to get GNU **parallel** working simply by copying a single file: No need to mess around with environment variables like PERL5LIB.

Choice of programming language

GNU **parallel** is designed to be able to run on old systems. That means that it cannot depend on a compiler being installed - and especially not a compiler for a language that is younger than 20 years old.

The goal is that you can use GNU **parallel** on any system, even if you are not allowed to install additional software.

Of all the systems I have experienced, I have yet to see a system that had GCC installed that did not have Perl. The same goes for Rust, Go, Haskell, and other younger languages. I have, however, seen systems with Perl without any of the mentioned compilers.

Most modern systems also have either Python2 or Python3 installed, but you still cannot be certain which version, and since Python2 cannot run under Python3, Python is not an option.

Perl has the added benefit that implementing the {= perlexpr =} replacement string was fairly easy.

The primary drawback is that Perl is slow. So there is an overhead of 3-10 ms/job and 1 ms/MB output (and even more if you use **--tag**).

Old Perl style

GNU **parallel** uses some old, deprecated constructs. This is due to a goal of being able to run on old installations. Currently the target is CentOS 3.9 and Perl 5.8.0.

Code structure: packages

GNU **parallel** is divided into these packages inside the single file:

main

Parses options, initializes globals, and drives the top-level execution loop (**run_gnu_parallel()**, **drain_job_queue()**).

SSHLogin

Represents one execution target - either a remote host (via SSH) or the local machine (the special sshlogin **:**). Stores the SSH command, credentials, host-group membership, number of parallel slots, and runtime state (jobs running, load, last-used time). The string format it accepts is:

```
[@grp+grp/][N/][sshcmd ][[user][:pass]@][host[:port]]
```

JobQueue

A queue of **Job** objects. Wraps **CommandLineQueue**, converting **CommandLine** objects into **Job** objects on demand. Supports **unget()** so that jobs that cannot start yet (e.g. because the load is too high) can be pushed back without loss.

Job

Represents a single job invocation. Manages the full lifecycle: process spawning (via **IPC::Open3**), stdin pipe writing (**--pipe** mode), stdout/stderr collection and forwarding, file transfer and cleanup (**--transfer**/**--return**/**--cleanup**), timeout tracking, retry logic, and exit status recording.

CommandLine

Represents one fully-instantiated command line: the command template with all replacement strings (**{}**, **{/}**, **{#}**, **{= =}**, etc.) resolved against a specific set of input arguments. Constructed by **CommandLineQueue::get()** and consumed by **Job::new()**.

CommandLineQueue

Generates **CommandLine** objects from the command template and one or more input sources (**:::**, **::::**, **:::+**, etc.). Handles **--link**, **-N**, **-m**, **-X**, and all the combinatorics of multiple input sources.

Limits::Command

Determines the maximum command-line length for the current host (or a remote SSH host). Binary-searches **ARG_MAX**, caches the result on disk, and adjusts for **tmux** title limits. Used by **CommandLineQueue** when packing multiple arguments with **-m** or **-X**.

RecordQueue

Reads records from one input source, splitting on the record separator.

RecordColQueue

Like **RecordQueue**, but also parses CSV columns (for **--csv**).

SQLRecordQueue

Reads input records from an SQL table (for **--sqlworker**).

MultifileQueue

Multiplexes several **RecordQueue**s, one per **:::** or **::::** argument source.

Arg

Wraps a single input argument value throughout its lifetime. Parses any trailing **@hostgroup** suffix when **--hostgroups** is active. Provides quoting helpers used by **CommandLine** to embed the value safely in shell commands.

TimeoutQueue

Tracks running jobs that have a timeout (**--timeout**). Implements the percentile timeout using a remedian (approximate median) data structure so that all run times need not be kept in memory.

SQL

Database backend for **--sqlandworker** / **--sqlmaster** / **--sqlworker**. Parses a DB URL (**sqlite:///path**, **pg://host/db**, **csv:///dir**, ...), connects via **DBI**, and manages the job-tracking table.

Semaphore

Implements the semaphore mode (**sem** / **parallel --semaphore**). Uses a lock file and advisory locking to limit concurrency across unrelated processes.

Scalability up and down

The smallest system GNU **parallel** is tested on is a 32 MB ASUS WL500gP. The largest is a 2 TB 128-core machine. It scales up to around 100 machines - depending on the duration of each job.

Exponentially back off

GNU **parallel** busy waits. This is because the reason why a job is not started may be due to load average (when using **--load**), and thus it will not make sense to just wait for a job to finish. Instead the load average must be rechecked regularly. Load average is not the only reason: **--timeout** has a similar problem.

To not burn up too much CPU GNU **parallel** sleeps exponentially longer and longer if nothing happens, maxing out at 1 second.

Shell compatibility

It is a goal to have GNU **parallel** work equally well in any shell. However, in practice GNU **parallel** is being developed in **bash** and thus testing in other shells is limited to reported bugs.

When an incompatibility is found there is often not an easy fix: Fixing the problem in **csh** often breaks it in **bash**. In these cases the fix is often to use a small Perl script and call that.

env_parallel

env_parallel is a dummy shell script that will run if **env_parallel** is not an alias or a function and tell the user how to activate the alias/function for the supported shells.

The alias or function will copy the current environment and run the command with GNU **parallel** in the copy of the environment.

The problem is that you cannot access all of the current environment inside Perl. E.g. aliases, functions and unexported shell variables.

The idea is therefore to take the environment and put it in **$PARALLEL_ENV** which GNU **parallel** prepends to every command.

The only way to have access to the environment is directly from the shell, so the program must be written in a shell script that will be sourced and there has to deal with the dialect of the relevant shell.

env_parallel.*

These are the files that implements the alias or function **env_parallel** for a given shell. It could be argued that these should be put in some obscure place under /usr/lib, but by putting them in your path it becomes trivial to find the path to them and **source** them:

```
source `which env_parallel.foo`
```

The beauty is that they can be put anywhere in the path without the user having to know the location. So if the user's path includes /afs/bin/i386_fc5 or /usr/pkg/parallel/bin or /usr/local/parallel/20161222/sunos5.6/bin the files can be put in the dir that makes most sense for the sysadmin.

env_parallel.bash / env_parallel.sh / env_parallel.ash / env_parallel.dash / env_parallel.zsh / env_parallel.ksh / env_parallel.mksh

env_parallel.(bash|sh|ash|dash|ksh|mksh|zsh) defines the function **env_parallel**. It uses **alias** and **typeset** to dump the configuration (with a few exceptions) into **$PARALLEL_ENV** before running GNU **parallel**.

After GNU **parallel** is finished, **$PARALLEL_ENV** is deleted.

env_parallel.csh

env_parallel.csh has two purposes: If **env_parallel** is not an alias: make it into an alias that sets **$PARALLEL** with arguments and calls **env_parallel.csh**.

If **env_parallel** is an alias, then **env_parallel.csh** uses **$PARALLEL** as the arguments for GNU **parallel**.

It exports the environment by writing a variable definition to a file for each variable. The definitions of aliases are appended to this file. Finally the file is put into **$PARALLEL_ENV**.

GNU **parallel** is then run and **$PARALLEL_ENV** is deleted.

env_parallel.fish

First all functions definitions are generated using a loop and **functions**.

Dumping the scalar variable definitions is harder.

fish can represent non-printable characters in (at least) 2 ways. To avoid problems all scalars are converted to \XX quoting.

Then commands to generate the definitions are made and separated by NUL.

This is then piped into a Perl script that quotes all values. List elements will be appended using two spaces.

Finally \n is converted into \1 because **fish** variables cannot contain \n. GNU **parallel** will later convert all \1 from **$PARALLEL_ENV** into \n.

This is then all saved in **$PARALLEL_ENV**.

GNU **parallel** is called, and **$PARALLEL_ENV** is deleted.

parset (supported in sh, ash, dash, bash, zsh, ksh, mksh)

parset is a shell function. This is the reason why **parset** can set variables: It runs in the shell which is calling it.

It is also the reason why **parset** does not work, when data is piped into it: **... | parset ...** makes **parset** start in a subshell, and any changes in environment can therefore not make it back to the calling shell.

Job slots

The easiest way to explain what GNU **parallel** does is to assume that there are a number of job slots, and when a slot becomes available a job from the queue will be run in that slot. But originally GNU **parallel** did not model job slots in the code. Job slots have been added to make it possible to use **{%}** as a replacement string.

While the job sequence number can be computed in advance, the job slot can only be computed the moment a slot becomes available. So it has been implemented as a stack with lazy evaluation: Draw one from an empty stack and the stack is extended by one. When a job is done, push the available job slot back on the stack.

This implementation also means that if you re-run the same jobs, you cannot assume jobs will get the same slots. And if you use remote executions, you cannot assume that a given job slot will remain on the same remote server. This goes double since number of job slots can be adjusted on the fly (by giving **--jobs** a file name).

Rsync protocol version

rsync 3.1.x uses protocol 31 which is unsupported by version 2.5.7. That means that you cannot push a file to a remote system using **rsync** protocol 31, if the remote system uses 2.5.7. **rsync** does not automatically downgrade to protocol 30.

GNU **parallel** does not require protocol 31, so if the **rsync** version is >= 3.1.0 then **--protocol 30** is added to force newer **rsync**s to talk to version 2.5.7.

Compression

GNU **parallel** buffers output in temporary files. **--compress** compresses the buffered data. This is a bit tricky because there should be no files to clean up if GNU **parallel** is killed by a power outage.

GNU **parallel** first selects a compression program. If the user has not selected one, the first of these that is in $PATH is used: **pzstd lbzip2 pbzip2 zstd pixz lz4 pigz lzop plzip lzip gzip lrz pxz bzip2 lzma xz clzip**. They are sorted by speed on a 128 core machine.

Schematically the setup is as follows:

```
command started by parallel | compress > tmpfile
cattail tmpfile | uncompress | parallel which reads the output
```

The setup is duplicated for both standard output (stdout) and standard error (stderr).

GNU **parallel** pipes output from the command run into the compression program which saves to a tmpfile. GNU **parallel** records the pid of the compress program. At the same time a small Perl script (called **cattail** above) is started: It basically does **cat** followed by **tail -f**, but it also removes the tmpfile as soon as the first byte is read, and it continuously checks if the pid of the compression program is dead. If the compress program is dead, **cattail** reads the rest of tmpfile and exits.

As most compression programs write out a header when they start, the tmpfile in practice is removed by **cattail** after around 40 ms.

More detailed it works like this:

```
bash ( command ) |
  sh ( emptywrapper ( bash ( compound compress ) ) >tmpfile )
cattail ( rm tmpfile; compound decompress ) < tmpfile
```

This complex setup is to make sure compress program is only started if there is input. This means each job will cause 8 processes to run. If combined with **--keep-order** these processes will run until the job has been printed.

Wrapping

The command given by the user can be wrapped in multiple templates. Templates can be wrapped in other templates.

$COMMAND

the command to run.

$INPUT

the input to run.

$SHELL

the shell that started GNU Parallel.

$SSHLOGIN

the sshlogin.

$WORKDIR

the working dir.

$FILE

the file to read parts from.

$STARTPOS

the first byte position to read from **$FILE**.

$LENGTH

the number of bytes to read from **$FILE**.

--shellquote

echo *Double quoted $INPUT*

--nice *pri*

Remote: See **The remote system wrapper**.

Local: **setpriority(0,0,$nice)**

--cat

```
cat > {}; $COMMAND {};
perl -e '$bash = shift;
```

```
    $csh = shift;
    for(@ARGV) { unlink;rmdir; }
    if($bash =~ s/h//) { exit $bash;  }
    exit $csh;' "$?h" "$status" {};
```

{} is set to **$PARALLEL_TMP** which is a tmpfile. The Perl script saves the exit value, unlinks the tmpfile, and returns the exit value - no matter if the shell is **bash**/**ksh**/**zsh** (using $?) or ***csh**/**fish** (using $status).

--fifo

```
perl -e '($s,$c,$f) = @ARGV;
  # mkfifo $PARALLEL_TMP
  system "mkfifo", $f;
  # spawn $shell -c $command &
  $pid = fork || exec $s, "-c", $c;
  open($o,">",$f) || die $!;
  # cat > $PARALLEL_TMP
  while(sysread(STDIN,$buf,131072)){
     syswrite $o, $buf;
  }
  close $o;
  # waitpid to get the exit code from $command
  waitpid $pid,0;
  # Cleanup
  unlink $f;
  exit $?/256;' $SHELL -c $COMMAND $PARALLEL_TMP
```

This is an elaborate way of: mkfifo {}; run **$COMMAND** in the background using **$SHELL**; copying STDIN to {}; waiting for background to complete; remove {} and exit with the exit code from **$COMMAND**.

It is made this way to be compatible with ***csh**/**fish**.

--pipepart

```
< $FILE 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);
    }
  }' $STARTPOS $LENGTH
```

This will read **$LENGTH** bytes from **$FILE** starting at **$STARTPOS** and send it to STDOUT.

--sshlogin $SSHLOGIN

```
ssh $SSHLOGIN "$COMMAND"
```

--transfer

```
ssh $SSHLOGIN mkdir -p ./$WORKDIR;
rsync --protocol 30 -rlDzR \
      -essh ./{} $SSHLOGIN:./$WORKDIR;
ssh $SSHLOGIN "$COMMAND"
```

Read about **--protocol 30** in the section **Rsync protocol version**.

--transferfile *file*

<<todo>>

--basefile

<<todo>>

--return *file*

```
  $COMMAND; _EXIT_status=$?; mkdir -p $WORKDIR;
  rsync --protocol 30 \
    --rsync-path=cd\ ./$WORKDIR\;\ rsync \
    -rlDzR -essh $SSHLOGIN:./$FILE ./$WORKDIR;
  exit $_EXIT_status;
```

The **--rsync-path=cd ...** is needed because old versions of **rsync** do not support **--no-implied-dirs**.

The **$_EXIT_status** trick is to postpone the exit value. This makes it incompatible with ***csh** and should be fixed in the future. Maybe a wrapping 'sh -c' is enough?

--cleanup

$RETURN is the wrapper from **--return**

```
  $COMMAND; _EXIT_status=$?; $RETURN;
  ssh $SSHLOGIN \(rm\ -f\ ./$WORKDIR/{}\;\
                  rmdir\ ./$WORKDIR\
\>\&/dev/null\;\);
  exit $_EXIT_status;
```

$_EXIT_status: see **--return** above.

--pipe

```
  perl -e 'if(sysread(STDIN, $buf, 1)) {
 open($fh, "|-", "@ARGV") || die;
 syswrite($fh, $buf);
 # Align up to 128k block
 if($read = sysread(STDIN, $buf, 131071)) {
     syswrite($fh, $buf);
 }
 while($read = sysread(STDIN, $buf, 131072)) {
     syswrite($fh, $buf);
 }
 close $fh;
 exit ($?&127 ? 128+($?&127) : 1+$?>>8)
    }' $SHELL -c $COMMAND
```

This small wrapper makes sure that **$COMMAND** will never be run if there is no data.

--tmux

<<TODO Fixup with '-quoting>> mkfifo /tmp/tmx3cMEV && sh -c 'tmux -S /tmp/tmsaKpv1 new-session -s p334310 -d "sleep .2" >/dev/null 2>&1'; tmux -S /tmp/tmsaKpv1 new-window -t p334310 -n wc\ 10 \(wc\ 10\)\;\ perl\ -e\ \'while\(\$t++\<3\)\{\ print\ \$ARGV\[0\],\"\\n\"\ \}\'\ \$\?h/\$status\ \>\>\ /tmp/tmx3cMEV\&echo\ wc\\\ 10\;\ echo\ \Job\ finished\ at:\ \`date\`\;sleep\ 10; exec perl -e '$/="/";$_=<>;$c=<>;unlink $ARGV;

/(\d+)h/ and exit($1);exit$c' /tmp/tmx3cMEV

mkfifo *tmpfile.tmx*; tmux -S <tmpfile.tms> new-session -s p*PID* -d 'sleep .2' >&/dev/null; tmux -S <tmpfile.tms> new-window -t p*PID* -n <<shell quoted input>> \(<<shell quoted input>>\)\;\ perl\ -e\ \'while\(\$t++\<3\)\{\ print\ \$ARGV\[0\],\"\\n\"\ \}\'\ \$\?h/\$status\ \>\>\ *tmpfile.tmx*\&echo\ <<shell double quoted input>>\;echo\ \Job\ finished\ at:\ \`date\`\;sleep\ 10; exec perl -e '$/="/";$_=<>;$c=<>;unlink $ARGV; /(\d+)h/ and exit($1);exit$c' *tmpfile.tmx*

First a FIFO is made (.tmx). It is used for communicating exit value. Next a new tmux session is made. This may fail if there is already a session, so the output is ignored. If all job slots finish at the same time, then **tmux** will close the session. A temporary socket is made (.tms) to avoid a race condition in **tmux**. It is cleaned up when GNU **parallel** finishes.

The input is used as the name of the windows in **tmux**. When the job inside **tmux** finishes, the exit value is printed to the FIFO (.tmx). This FIFO is opened by **perl** outside **tmux**, and **perl** then removes the FIFO. **Perl** blocks until the first value is read from the FIFO, and this value is used as exit value.

To make it compatible with **csh** and **bash** the exit value is printed as: $?h/$status and this is parsed by **perl**.

There is a bug that makes it necessary to print the exit value 3 times.

Another bug in **tmux** requires the length of the tmux title and command to not have certain limits. When inside these limits, 75 '\ ' are added to the title to force it to be outside the limits.

You can map the bad limits using:

```
  perl -e 'sub r { int(rand(shift)).($_[0] &&
"\t".r(@_)) } print map { r(@ARGV)."\n" } 1..10000'
1600 1500 90 |
    perl -ane '$F[0]+$F[1]+$F[2] < 2037 and print ' |
    parallel --colsep '\t' --tagstring '{1}\t{2}\t{3}'
 tmux -S /tmp/p{%}-'{=3 $_="O"x$_ =}' \
      new-session -d -n '{=1 $_="O"x$_ =}' true'\ {=2
$_="O"x$_ =};echo $?;rm -f /tmp/p{%}-O*'

  perl -e 'sub r { int(rand(shift)).($_[0] &&
"\t".r(@_)) } print map { r(@ARGV)."\n" } 1..10000'
17000 17000 90 |
    parallel --colsep '\t' --tagstring '{1}\t{2}\t{3}'
 \
  tmux -S /tmp/p{%}-'{=3 $_="O"x$_ =}' new-session -d
-n '{=1 $_="O"x$_ =}' true'\ {=2 $_="O"x$_ =};echo
$?;rm /tmp/p{%}-O*'
  > value.csv 2>/dev/null

  R -e
'a<-read.table("value.csv");X11();plot(a[,1],a[,2],col
=a[,4]+5,cex=0.1);Sys.sleep(1000)'
```

For **tmux 1.8** 17000 can be lowered to 2100.

The interesting areas are title 0..1000 with (title + whole command) in 996..1127 and 9331..9636.

The ordering of the wrapping is important:

- $PARALLEL_ENV which is set in env_parallel.* must be prepended to the command first, as

the command may contain exported variables or functions.

- **--nice/--cat/--fifo** should be done on the remote machine
- **--pipepart/--pipe** should be done on the local machine inside **--tmux**

Convenience options --nice --basefile --transfer --return --cleanup --tmux --group --compress --cat --fifo --workdir --tag --tagstring

These are all convenience options that make it easier to do a task. But more importantly: They are tested to work on corner cases, too. Take **--nice** as an example:

```
nice parallel command ...
```

will work just fine. But when run remotely, you need to move the nice command so it is being run on the server:

```
parallel -S server nice command ...
```

And this will again work just fine, as long as you are running a single command. When you are running a composed command you need nice to apply to the whole command, and it gets harder still:

```
parallel -S server -q nice bash -c 'command1 ...; cmd2 | cmd3'
```

It is not impossible, but by using **--nice** GNU **parallel** will do the right thing for you. Similarly when transferring files: It starts to get hard when the file names contain space, :, `, *, or other special characters.

To run the commands in a **tmux** session you basically just need to quote the command. For simple commands that is easy, but when commands contain special characters, it gets much harder to get right.

--compress not only compresses standard output (stdout) but also standard error (stderr); and it does so into files, that are open but deleted, so a crash will not leave these files around.

--cat and **--fifo** are easy to do by hand, until you want to clean up the tmpfile and keep the exit code of the command.

The real killer comes when you try to combine several of these: Doing that correctly for all corner cases is next to impossible to do by hand.

--shard

The simple way to implement sharding would be to:

1 start n jobs,

2 split each line into columns,

3 select the data from the relevant column

4 compute a hash value from the data

5 take the modulo n of the hash value

6 pass the full line to the jobslot that has the computed value

Unfortunately Perl is rather slow at computing the hash value (and somewhat slow at splitting into columns).

One solution is to use a compiled language for the splitting and hashing, but that would go against the design criteria of not depending on a compiler.

Luckily those tasks can be parallelized. So GNU **parallel** starts n sharders that do step 2-6, and

passes blocks of 100k to each of those in a round robin manner. To make sure these sharders compute the hash the same way, $PERL_HASH_SEED is set to the same value for all sharders.

Running n sharders poses a new problem: Instead of having n outputs (one for each computed value) you now have n outputs for each of the n values, so in total n*n outputs; and you need to merge these n*n outputs together into n outputs.

This can be done by simply running 'parallel -j0 --lb cat ::: outputs_for_one_value', but that is rather inefficient, as it spawns a process for each file. Instead the core code from 'parcat' is run, which is also a bit faster.

All the sharders and parcats communicate through named pipes that are unlinked as soon as they are opened.

Shell shock

The shell shock bug in **bash** did not affect GNU **parallel**, but the solutions did. **bash** first introduced functions in variables named: *BASH_FUNC_myfunc()* and later changed that to *BASH_FUNC_myfunc%%*. When transferring functions GNU **parallel** reads off the function and changes that into a function definition, which is copied to the remote system and executed before the actual command is executed. Therefore GNU **parallel** needs to know how to read the function.

From version 20150122 GNU **parallel** tries both the ()-version and the %%-version, and the function definition works on both pre- and post-shell shock versions of **bash**.

The remote system wrapper

The remote system wrapper does some initialization before starting the command on the remote system.

Make quoting unnecessary by hex encoding everything

When you run **ssh server foo** then **foo** has to be quoted once:

```
ssh server "echo foo; echo bar"
```

If you run **ssh server1 ssh server2 foo** then **foo** has to be quoted twice:

```
ssh server1 ssh server2 \'"echo foo; echo bar"\'
```

GNU **parallel** avoids this by packing everyting into hex values and running a command that does not need quoting:

```
perl -X -e GNU_Parallel_worker,eval+pack+q/H10000000/,join+q//,@ARGV
```

This command reads hex from the command line and converts that to bytes that are then eval'ed as a Perl expression.

The string **GNU_Parallel_worker** is not needed. It is simply there to let the user know, that this process is GNU **parallel** working.

Ctrl-C and standard error (stderr)

If the user presses Ctrl-C the user expects jobs to stop. This works out of the box if the jobs are run locally. Unfortunately it is not so simple if the jobs are run remotely.

If remote jobs are run in a tty using **ssh -tt**, then Ctrl-C works, but all output to standard error (stderr) is sent to standard output (stdout). This is not what the user expects.

If remote jobs are run without a tty using **ssh** (without **-tt**), then output to standard error (stderr) is kept on stderr, but Ctrl-C does not kill remote jobs. This is not what the user expects.

So what is needed is a way to have both. It seems the reason why Ctrl-C does not kill the remote jobs is because the shell does not propagate the hang-up signal from **sshd**. But when **sshd** dies, the

parent of the login shell becomes **init** (process id 1). So by exec'ing a Perl wrapper to monitor the parent pid and kill the child if the parent pid becomes 1, then Ctrl-C works and stderr is kept on stderr.

Ctrl-C does, however, kill the ssh connection, so any output from a remote dying process is lost.

To be able to kill all (grand)*children a new process group is started.

--nice

niceing the remote process is done by **setpriority(0,0,$nice)**. A few old systems do not implement this and **--nice** is unsupported on those.

Setting $PARALLEL_TMP

$PARALLEL_TMP is used by **--fifo** and **--cat** and must point to a non-exitent file in **$TMPDIR**. This file name is computed on the remote system.

The wrapper

The wrapper looks like this:

```
$shell = $PARALLEL_SHELL || $SHELL;
$tmpdir = $TMPDIR || $PARALLEL_REMOTE_TMPDIR;
$nice = $opt::nice;
$termseq = $opt::termseq;


# Check that $tmpdir is writable
-w $tmpdir ||
    die("$tmpdir is not writable.".
 " Set PARALLEL_REMOTE_TMPDIR");
# Set $PARALLEL_TMP to a non-existent file name in $TMPDIR
do {
    $ENV{PARALLEL_TMP} = $tmpdir."/par".
 join"", map { (0..9,"a".."z","A".."Z")[rand(62)] } (1..5);
} while(-e $ENV{PARALLEL_TMP});
# Set $script to a non-existent file name in $TMPDIR
do {
    $script = $tmpdir."/par".
 join"", map { (0..9,"a".."z","A".."Z")[rand(62)] } (1..5);
} while(-e $script);
# Create a script from the hex code
# that removes itself and runs the commands
open($fh,">",$script) || die;
# ' needed due to rc-shell
print($fh("rm \'$script\'\n",$bashfunc.$cmd));
close $fh;
my $parent = getppid;
my $done = 0;
$SIG{CHLD} = sub { $done = 1; };
$pid = fork;
unless($pid) {
    # Make own process group to be able to kill HUP it later
    eval { setpgrp };
    # Set nice value
    eval { setpriority(0,0,$nice) };
    # Run the script
    exec($shell,$script);
    die("exec failed: $!");
}
while((not $done) and (getppid == $parent)) {
    # Parent pid is not changed, so sshd is alive
```

```
    # Exponential sleep up to 1 sec
    $s = $s < 1 ? 0.001 + $s * 1.03 : $s;
    select(undef, undef, undef, $s);
}
if(not $done) {
    # sshd is dead: User pressed Ctrl-C
    # Kill as per --termseq
    my @term_seq = split/,/,$termseq;
    if(not @term_seq) {
 @term_seq = ("TERM",200,"TERM",100,"TERM",50,"KILL",25);
    }
    while(@term_seq && kill(0,-$pid)) {
 kill(shift @term_seq, -$pid);
 select(undef, undef, undef, (shift @term_seq)/1000);
    }
}
wait;
exit ($?&127 ? 128+($?&127) : 1+$?>>8)
```

Transferring of variables and functions

Transferring of variables and functions given by **--env** is done by running a Perl script remotely that calls the actual command. The Perl script sets **$ENV{***variable***}** to the correct value before exec'ing a shell that runs the function definition followed by the actual command.

The function **env_parallel** copies the full current environment into the environment variable **PARALLEL_ENV**. This variable is picked up by GNU **parallel** and used to create the Perl script mentioned above.

Base64 encoded bzip2

csh limits words of commands to 1024 chars. This is often too little when GNU **parallel** encodes environment variables and wraps the command with different templates. All of these are combined and quoted into one single word, which often is longer than 1024 chars.

When the line to run is > 1000 chars, GNU **parallel** therefore encodes the line to run. The encoding **bzip2**s the line to run, converts this to base64, splits the base64 into 1000 char blocks (so **csh** does not fail), and prepends it with this Perl script that decodes, decompresses and **eval**s the line.

The decoder (**base64_zip_eval()**) looks like this:

```
@GNU_Parallel=("use","IPC::Open3;","use","MIME::Base64");
eval "@GNU_Parallel";


$SIG{CHLD}="IGNORE";
# Search for bzip2. Not found => use default path
my $zip = (grep { -x $_ } "/usr/local/bin/bzip2")[0] || "bzip2";
# $in = stdin on $zip, $out = stdout from $zip
my($in, $out,$eval);
open3($in,$out,">&STDERR",$zip,"-dc");
if(my $perlpid = fork) {
    close $in;
    $eval = join "", <$out>;
    close $out;
} else {
    close $out;
    # Pipe decoded base64 into 'bzip2 -dc'
    print $in (decode_base64(join"",@ARGV));
    close $in;
```

```
    exit;
}
wait;
eval $eval;
```

Perl and **bzip2** must be installed on the remote system, but a small test showed that **bzip2** is installed by default on all platforms that runs GNU **parallel**, so this is not a big problem.

The added bonus of this is that much bigger environments can now be transferred as they will be below **bash**'s limit of 131072 chars.

There is also a simpler **base64_eval()** variant (no bzip2) used when the line is between the plain-text limit and the bzip2 threshold, and a **base64_wrap()** function that chooses the right encoding and builds the complete shell command with the decoder prepended.

Which shell to use

Different shells behave differently. A command that works in **tcsh** may not work in **bash**. It is therefore important that the correct shell is used when GNU **parallel** executes commands.

GNU **parallel** tries hard to use the right shell. If GNU **parallel** is called from **tcsh** it will use **tcsh**. If it is called from **bash** it will use **bash**. It does this by looking at the (grand)*parent process: If the (grand)*parent process is a shell, use this shell; otherwise look at the parent of this (grand)*parent. If none of the (grand)*parents are shells, then $SHELL is used.

This will do the right thing if called from:

- an interactive shell
- a shell script
- a Perl script in `` or using **system** if called as a single string.

While these cover most cases, there are situations where it will fail:

- When run using **exec**.
- When run as the last command using **-c** from another shell (because some shells use **exec**):

  ```
  zsh% bash -c "parallel 'echo {} is not run in bash; \
       set | grep BASH_VERSION' ::: This"
  ```

  You can work around that by appending '&& true':

  ```
  zsh% bash -c "parallel 'echo {} is run in bash; \
       set | grep BASH_VERSION' ::: This && true"
  ```

- When run in a Perl script using **system** with parallel as the first string:

  ```
  #!/usr/bin/perl

  system("parallel",'setenv a {}; echo $a',":::",2);
  ```

  Here it depends on which shell is used to call the Perl script. If the Perl script is called from **tcsh** it will work just fine, but if it is called from **bash** it will fail, because the command **setenv** is not known to **bash**.

If GNU **parallel** guesses wrong in these situation, set the shell using **$PARALLEL_SHELL**.

Always running commands in a shell

If the command is a simple command with no redirection and setting of variables, the command *could* be run without spawning a shell. E.g. this simple **grep** matching either 'ls ' or ' wc >> c':

```
parallel "grep -E 'ls | wc >> c' {}" ::: foo
```

could be run as:

```
system("grep","-E","ls | wc >> c","foo");
```

However, as soon as the command is a bit more complex a shell *must* be spawned:

```
parallel "grep -E 'ls | wc >> c' {} | wc >> c" ::: foo
parallel "LANG=C grep -E 'ls | wc >> c' {}" ::: foo
```

It is impossible to tell how **| wc >> c** should be interpreted without parsing the string (is the **|** a pipe in shell or an alternation in a **grep** regexp? Is **LANG=C** a command in **csh** or setting a variable in **bash**? Is **>>** redirection or part of a regexp?).

On top of this, wrapper scripts will often require a shell to be spawned.

The downside is that you need to quote special shell chars twice:

```
parallel echo '*' ::: This will expand the asterisk
parallel echo "'*'" ::: This will not
parallel "echo '*'" ::: This will not
parallel echo '\*' ::: This will not
parallel echo \''*'\' ::: This will not
parallel -q echo '*' ::: This will not
```

-q will quote all special chars, thus redirection will not work: this prints '* > out.1' and *does not* save '*' into the file out.1:

```
parallel -q echo "*" ">" out.{} ::: 1
```

GNU **parallel** tries to live up to Principle Of Least Astonishment (POLA), and the requirement of using **-q** is hard to understand, when you do not see the whole picture.

Quoting

Quoting depends on the shell. For most shells '-quoting is used for strings containing special characters.

For **tcsh**/**csh** newline is quoted as \ followed by newline. Other special characters are also \-quoted.

For **rc** everything is quoted using '.

--pipepart vs. --pipe

While **--pipe** and **--pipepart** look much the same to the user, they are implemented very differently.

With **--pipe** GNU **parallel** reads the blocks from standard input (stdin), which is then given to the command on standard input (stdin); so every block is being processed by GNU **parallel** itself. This is the reason why **--pipe** maxes out at around 500 MB/sec.

--pipepart, on the other hand, first identifies at which byte positions blocks start and how long they are. It does that by seeking into the file by the size of a block and then reading until it meets end of a block. The seeking explains why GNU **parallel** does not know the line number and why **-L/-l** and **-N** do not work.

With a reasonable block and file size this seeking is more than 1000 time faster than reading the full file. The byte positions are then given to a small script that reads from position X to Y and sends output to standard output (stdout). This small script is prepended to the command and the full command is executed just as if GNU **parallel** had been in its normal mode. The script looks like this:

```
< file 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);
   }
}' startbyte length_in_bytes
```

It delivers 1 GB/s per core.

Instead of the script **dd** was tried, but many versions of **dd** do not support reading from one byte to another and might cause partial data. See this for a surprising example:

```
yes | dd bs=1024k count=10 | wc
```

--block-size adjustment

Every time GNU **parallel** detects a record bigger than **--block-size** it increases the block size by 30%. A small **--block-size** gives very poor performance; by exponentially increasing the block size performance will not suffer.

GNU **parallel** will waste CPU power if **--block-size** does not contain a full record, because it tries to find a full record and will fail to do so. The recommendation is therefore to use a **--block-size** > 2 records, so you always get at least one full record when you read one block.

If you use **-N** then **--block-size** should be big enough to contain N+1 records.

Automatic --block-size computation

With **--pipepart** GNU **parallel** can compute the **--block-size** automatically. A **--block-size** of **-1** will use a block size so that each jobslot will receive approximately 1 block. **--block -2** will pass 2 blocks to each jobslot and **-***n* will pass *n* blocks to each jobslot.

This can be done because **--pipepart** reads from files, and we can compute the total size of the input.

--jobs and --onall

When running the same commands on many servers what should **--jobs** signify? Is it the number of servers to run on in parallel? Is it the number of jobs run in parallel on each server?

GNU **parallel** lets **--jobs** represent the number of servers to run on in parallel. This is to make it possible to run a sequence of commands (that cannot be parallelized) on each server, but run the same sequence on multiple servers.

--shuf

When using **--shuf** to shuffle the jobs, all jobs are read, then they are shuffled, and finally executed. When using SQL this makes the **--sqlmaster** be the part that shuffles the jobs. The **--sqlworker**s simply executes according to Seq number.

--csv

--pipepart is incompatible with **--csv** because you can have records like:

```
a,b,c
a,"
a,b,c
a,b,c
a,b,c
",c
a,b,c
```

Here the second record contains a multi-line field that looks like records. Since **--pipepart** does not read then whole file when searching for record endings, it may start reading in this multi-line field, which would be wrong.

Buffering on disk

GNU **parallel** buffers output, because if output is not buffered you have to be ridiculously careful on sizes to avoid mixing of outputs (see excellent example on https://catern.com/posts/pipes.html).

GNU **parallel** buffers on disk in $TMPDIR using files, that are removed as soon as they are created, but which are kept open. So even if GNU **parallel** is killed by a power outage, there will be no files to clean up afterwards. Another advantage is that the file system is aware that these files will be lost in case of a crash, so it does not need to sync them to disk.

It gives the odd situation that a disk can be fully used, but there are no visible files on it.

Partly buffering in memory

When using output formats SQL and CSV then GNU Parallel has to read the whole output into memory. When run normally it will only read the output from a single job. But when using **--linebuffer** every line printed will also be buffered in memory - for all jobs currently running.

If memory is tight, then do not use the output format SQL/CSV with **--linebuffer**.

Comparing to buffering in memory

gargs is a parallelizing tool that buffers in memory. It is therefore a useful way of comparing the advantages and disadvantages of buffering in memory to buffering on disk.

On an system with 6 GB RAM free and 6 GB free swap these were tested with different sizes:

```
echo /dev/zero | gargs "head -c $size {}" >/dev/null
echo /dev/zero | parallel "head -c $size {}" >/dev/null
```

The results are here:

```
JobRuntime      Command
     0.344      parallel_test 1M
     0.362      parallel_test 10M
     0.640      parallel_test 100M
     9.818      parallel_test 1000M
    23.888      parallel_test 2000M
    30.217      parallel_test 2500M
    30.963      parallel_test 2750M
    34.648      parallel_test 3000M
    43.302      parallel_test 4000M
    55.167      parallel_test 5000M
    67.493      parallel_test 6000M
   178.654      parallel_test 7000M
   204.138      parallel_test 8000M
   230.052      parallel_test 9000M
   255.639      parallel_test 10000M
   757.981      parallel_test 30000M
     0.537      gargs_test 1M
     0.292      gargs_test 10M
     0.398      gargs_test 100M
     3.456      gargs_test 1000M
     8.577      gargs_test 2000M
    22.705      gargs_test 2500M
   123.076      gargs_test 2750M
    89.866      gargs_test 3000M
   291.798      gargs_test 4000M
```

GNU **parallel** is pretty much limited by the speed of the disk: Up to 6 GB data is written to disk but cached, so reading is fast. Above 6 GB data are both written and read from disk. When the 30000MB job is running, the disk system is slow, but usable: If you are not using the disk, you almost do not feel it.

gargs has a speed advantage up until 2500M where it hits a wall. Then the system starts swapping like crazy and is completely unusable. At 5000M it goes out of memory.

You can make GNU **parallel** behave similar to **gargs** if you point $TMPDIR to a tmpfs-filesystem: It will be faster for small outputs, but may kill your system for larger outputs and cause you to lose output.

Disk full

GNU **parallel** buffers on disk. If the disk is full, data may be lost. To check if the disk is full GNU **parallel** writes a 8193 byte file every second. If this file is written successfully, it is removed immediately. If it is not written successfully, the disk is full. The size 8193 was chosen because 8192 gave wrong result on some file systems, whereas 8193 did the correct thing on all tested filesystems.

Memory usage

Normally GNU **parallel** will use around 17 MB RAM constantly - no matter how many jobs or how much output there is. There are a few things that cause the memory usage to rise:

- Multiple input sources. GNU **parallel** reads an input source only once. This is by design, as an input source can be a stream (e.g. FIFO, pipe, standard input (stdin)) which cannot be rewound and read again. When reading a single input source, the memory is freed as soon as the job is done - thus keeping the memory usage constant.

  But when reading multiple input sources GNU **parallel** keeps the already read values for generating all combinations with other input sources.

- Computing the number of jobs. **--bar**, **--eta**, and **--halt xx%** use **total_jobs()** to compute the total number of jobs. It does this by generating the data structures for all jobs. All these job data structures will be stored in memory and take up around 400 bytes/job.

- Buffering a full line. **--linebuffer** will read a full line per running job. A very long output line (say 1 GB without \n) will increase RAM usage temporarily: From when the beginning of the line is read till the line is printed.

- Buffering the full output of a single job. This happens when using **--results *.csv/*.tsv** or **--sql***. Here GNU **parallel** will read the whole output of a single job and save it as csv/tsv or SQL.

Argument separators ::: :::: :::+ ::::+

The argument separator **:::** was chosen because I have never seen **:::** used in any command. The natural choice **--** would be a bad idea since it is not unlikely that the template command will contain **--**. I have seen **::** used in programming languanges to separate classes, and I did not want the user to be confused that the separator had anything to do with classes.

::: also makes a visual separation, which is good if there are multiple **:::**.

When **:::** was chosen, **::::** came as a fairly natural extension.

Linking input sources meant having to decide for some way to indicate linking of **:::** and **::::**. **:::+** and **::::+** were chosen, so that they were similar to **:::** and **::::**.

In 2022 I realized that ***///*** would have been an even better choice, because you cannot have an file named ***///*** whereas you *can* have a file named **:::**.

Perl replacement strings, {= =}, and --rpl

The shorthands for replacement strings make a command look more cryptic. Different users will need different replacement strings. Instead of inventing more shorthands you get more flexible replacement strings if they can be programmed by the user.

The language Perl was chosen because GNU **parallel** is written in Perl and it was easy and reasonably fast to run the code given by the user.

If a user needs the same programmed replacement string again and again, the user may want to make his own shorthand for it. This is what **--rpl** is for. It works so well, that even GNU **parallel**'s own shorthands are implemented using **--rpl**.

In Perl code the bigrams **{=** and **=}** rarely exist. They look like a matching pair and can be entered on all keyboards. This made them good candidates for enclosing the Perl expression in the replacement strings. Another candidate ,, and ,, was rejected because they do not look like a matching pair. **--parens** was made, so that the users can still use ,, and ,, if they like: **--parens ,,,,**

Internally, however, the **{=** and **=}** are replaced by \257< and \257>. This is to make it simpler to make regular expressions. You only need to look one character ahead, and never have to look behind.

Fast mode

--fast is an execution path that avoids spawning one process per job. Instead jobs are batched into shell script chunks. Each chunk contains multiple jobs that run sequentially inside one **/bin/sh** process (override with **$PARALLEL_SHELL**). Chunks are separated by the sentinel string `"\n#\0\n"` and fed to an inner `parallel --pipe -N1` that executes **-j** chunks in parallel. The inner **parallel** sees each chunk as a single job, so grouping and ordering happen at the chunk level, not the individual-job level.

The chunk size starts at 300 jobs and grows 1.5x each round (capped at 1000) to keep chunks small enough that all worker slots stay busy. The inner **parallel** uses `--block 512000`.

Shell helper functions injected into every chunk:

`_error()`

: Accumulates failure count in `$_exit`. Defined only when **--halt** is active; each job is wrapped as `( cmd ) || _error`.

`_tagger()`

: Awk stream that reads marker lines to switch the tag string and prefixes all other lines. Used with **--tag**/**--tagstring**. Two named fifos (immediately unlinked after creation) route stdout and stderr through separate awk processes, giving correct per-job labelling even inside a multi-job chunk.

The fast path (`$fp_fh` set) reads input lines directly and does string substitution without creating Job objects, giving ~0.1 ms/job. When the fast path is not eligible (non-trivial replacement strings, multiple input sources, etc.) the normal job-queue machinery builds the command strings (~0.5 ms/job) but the same chunk-based executor is used.

Fast mode buffer parameter tuning

Three parameters govern chunk sizing: `pipe_block` (the `--block` given to inner **parallel --pipe**), `buf_growth` (the multiplier applied after each chunk is sent), and `buf_cap` (the maximum chunk size in jobs).

Two benchmarks were run to find the best values:

Benchmark 1 - fast jobs (echo, ~0 ms/job): 25 combinations of `pipe_block` (64000â€“1048576 bytes) Ã— `buf_growth` (1.2â€“5.0) across seven job-count scales (1 000â€“1 000 000), five runs each. Best: `pipe_block=512000, buf_growth=5.0, buf_cap=100000`.

Benchmark 2 - slow jobs (`sleep 0.001`, ~1 ms/job, 4-core machine): 25 combinations of `buf_growth` (1.2â€“5.0) Ã— `buf_cap` (100â€“2500) for 1 000 and 10 000 jobs, three runs each. Best: `buf_growth=1.5, buf_cap=250`.

The chosen defaults **buf_growth=1.5, buf_cap=1000** optimise for real-world jobs (â‰¥1 ms each) on 4â€“20 cores by generating many small, balanced chunks.

Test suite

GNU **parallel** uses its own testing framework. This is mostly due to historical reasons. It deals reasonably well with tests that are dependent on how long a given test runs (e.g. more than 10 secs is a pass, but less is a fail). It parallelizes most tests, but it is easy to force a test to run as the single test (which may be important for timing issues). It deals reasonably well with tests that fail intermittently. It detects which tests failed and pushes these to the top, so when running the test suite again, the tests that failed most recently are run first.

If GNU **parallel** should adopt a real testing framework then those elements would be important.

Since many tests are dependent on which hardware it is running on, these tests break when run on a different hardware than what the test was written for.

When most bugs are fixed a test is added, so this bug will not reappear. It is, however, sometimes hard to create the environment in which the bug shows up - especially if the bug only shows up sometimes. One of the harder problems was to make a machine start swapping without forcing it to its knees.

Median run time

Using a percentage for **--timeout** causes GNU **parallel** to compute the median run time of a job. The median is a better indicator of the expected run time than average, because there will often be outliers taking way longer than the normal run time.

To avoid keeping all run times in memory, an implementation of remedian was made (Rousseeuw et al).

Error messages and warnings

Error messages like: ERROR, Not found, and 42 are not very helpful. GNU **parallel** strives to inform the user:

- What went wrong?
- Why did it go wrong?
- What can be done about it?

Unfortunately it is not always possible to predict the root cause of the error.

Determine number of CPUs

CPUs is an ambiguous term. It can mean the number of socket filled (i.e. the number of physical chips). It can mean the number of cores (i.e. the number of physical compute cores). It can mean the number of hyperthreaded cores (i.e. the number of virtual cores - with some of them possibly being hyperthreaded).

On ark.intel.com Intel uses the terms *cores* and *threads* for number of physical cores and the number of hyperthreaded cores respectively.

GNU **parallel** uses uses *CPUs* as the number of compute units and the terms *sockets*, *cores*, and *threads* to specify how the number of compute units is calculated.

Computation of load

Contrary to the obvious **--load** does not use load average. This is due to load average rising too slowly. Instead it uses **ps** to list the number of threads in running or blocked state (state D, O or R). This gives an instant load.

As remote calculation of load can be slow, a process is spawned to run **ps** and put the result in a file, which is then used next time.

Killing jobs

GNU **parallel** kills jobs. It can be due to **--memfree**, **--halt**, or when GNU **parallel** meets a condition from which it cannot recover. Every job is started as its own process group. This way any (grand)*children will get killed, too. The process group is killed with the specification mentioned in **--termseq**.

SQL interface

GNU **parallel** uses the DBURL from GNU **sql** to give database software, username, password, host, port, database, and table in a single string.

The DBURL must point to a table name. The table will be dropped and created. The reason for not reusing an existing table is that the user may have added more input sources which would require more columns in the table. By prepending '+' to the DBURL the table will not be dropped.

The table columns are similar to joblog with the addition of **V1** .. **Vn** which are values from the input sources, and Stdout and Stderr which are the output from standard output and standard error, respectively.

The Signal column has been renamed to _Signal due to Signal being a reserved word in MySQL.

Logo

The logo is inspired by the Cafe Wall illusion. The font is DejaVu Sans.

Citation notice

For details: See https://git.savannah.gnu.org/cgit/parallel.git/tree/doc/citation-notice-faq.txt

Funding a free software project is hard. GNU **parallel** is no exception. On top of that it seems the less visible a project is, the harder it is to get funding. And the nature of GNU **parallel** is that it will never be seen by "the guy with the checkbook", but only by the people doing the actual work.

This problem has been covered by others - though no solution has been found:
https://www.slideshare.net/NadiaEghbal/consider-the-maintainer
https://www.numfocus.org/blog/why-is-numpy-only-now-getting-funded/

Before implementing the citation notice it was discussed with the users:
https://lists.gnu.org/archive/html/parallel/2013-11/msg00006.html

Having to spend 10 seconds on running **parallel --citation** once is no doubt not an ideal solution, but no one has so far come up with an ideal solution - neither for funding GNU **parallel** nor other free software.

If you believe you have the perfect solution, you should try it out, and if it works, you should post it on the email list. Ideas that will cost work and which have not been tested are, however, unlikely to be prioritized.

Running **parallel --citation** one single time takes less than 10 seconds, and will silence the citation notice for future runs. This is comparable to graphical tools where you have to click a checkbox saying "Do not show this again". But if that is too much trouble for you, why not use one of the alternatives instead? See a list in: **man parallel_alternatives**.

As the request for citation is not a legal requirement this is acceptable under GPLv3 and cleared with Richard M. Stallman himself. Thus it does not fall under this:
https://www.gnu.org/licenses/gpl-faq.en.html#RequireCitation

Ideas for new design

Multiple processes working together

Open3 is slow. Printing is slow. It would be good if they did not tie up resources, but were run in separate threads.

--rrs on remote using a perl wrapper

... | perl -pe '$/=$recend$recstart;BEGIN{ if(substr($_) eq $recstart) substr($_)="" } eof and substr($_) eq $recend) substr($_)=""

It ought to be possible to write a filter that removed rec sep on the fly instead of inside GNU **parallel**. This could then use more cpus.

Will that require 2x record size memory?

Will that require 2x block size memory?

Historical decisions

These decisions were relevant for earlier versions of GNU **parallel**, but not the current version. They are kept here as historical record.

--tollef

You can read about the history of GNU **parallel** on https://www.gnu.org/software/parallel/history.html

--tollef was included to make GNU **parallel** switch compatible with the parallel from moreutils (which is made by Tollef Fog Heen). This was done so that users of that parallel easily could port their use to GNU **parallel**: Simply set **PARALLEL="--tollef"** and that would be it.

But several distributions chose to make **--tollef** global (by putting it into /etc/parallel/config) without making the users aware of this, and that caused much confusion when people tried out the examples from GNU **parallel**'s man page and these did not work. The users became frustrated because the distribution did not make it clear to them that it has made **--tollef** global.

So to lessen the frustration and the resulting support, **--tollef** was obsoleted 20130222 and removed one year later.