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 like it does, and 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 messing around with environment variables like PERL5LIB.

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.

Exponentially back off

GNU **parallel** busy waits. This is because the reason why a job is not started may be due to load average, and thus it will not make sense to wait for a job to finish. Instead the load average must be checked again. Load average is not the only reason.

To not burn up too 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.

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.

Job slots were added to the code in 20140522, but 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 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

--compress compresses the data in the temporary files. 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 compress program. If the user has not selected one, the first of these that are in $PATH is used: **lzop pigz pxz gzip plzip pbzip2 lzma xz lzip bzip2**. They are sorted by speed

on a 8 core machine.

Schematically the setup is as follows:

```
command started by parallel | compress > tmpfile
cattail tmpfile | uncompress | parallel
```

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

GNU **parallel** pipes output from the command run into the compress 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 continously checks if the pid of the compress program is dead. If the compress program is dead, **cattail** reads the rest of tmpfile and exits.

As most compress programs write out a header when they start, the tmpfile in practice is unlinked after around 40 ms.

Wrapping

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

--shellquote

echo <<shell double quoted input>>

--nice *pri*

\nice -n *pri* $shell -c <<shell quoted input>>

The \ is needed to avoid using the builtin nice command, which does not support -n in **tcsh**. **$shell -c** is needed to nice composed commands command.

--cat

(cat > {}; <<input>> {}; perl -e '$bash=shift; $csh=shift; for(@ARGV) {unlink;rmdir;} if($bash=~s/h//) {exit$bash;} exit$csh;' "$?h" "$status" {});

{} is really just a tmpfile. The Perl script saves the exit value, unlinks the tmpfile, and returns the exit value - no matter if the shell is **bash** (using $?) or ***csh** (using $status).

--fifo

(mkfifo {}; (<<input>> {};) & _PID=$!; cat > {}; wait $_PID; perl -e '$bash=shift; $csh=shift; for(@ARGV) {unlink;rmdir;} if($bash=~s/h//) {exit$bash;} exit$csh;' "$?h" "$status" {});

wait $_PID makes sure the exit value is from that PID. This makes it incompatible with ***csh**. The Perl script is the same as from **--cat**.

--sshlogin *sln*

ssh *sln* <<shell quoted input>>

--transfer

(ssh *sln* mkdir -p ./*workdir*;rsync --protocol 30 -rlDzR -essh ./{} *sln*:./*workdir*); <<input>>

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

--basefile

<<todo>>

--return *file*

<<input>>; _EXIT_status=$?; mkdir -p *workdir*; rsync --protocol 30 --rsync-path=cd\ ./*workdir*\;\ rsync -rlDzR -essh *sln*:./*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

<<input>> _EXIT_status=$?; <<return>>

ssh *sln* \(rm\ -f\ ./*workdir*/{}\;\ rmdir\ ./*workdir*\ \>\&/dev/null\;\); exit $_EXIT_status;

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

--pipe

sh -c 'dd bs=1 count=1 of=*tmpfile* 2>/dev/null'; test ! -s "*tmpfile*" && rm -f "*tmpfile*" && exec true; (cat *tmpfile*; rm *tmpfile*; cat -) | (<<input>>);

This small wrapper makes sure that <<input>> will never be run if there is no data. **sh -c** is needed to hide stderr if the user's shell is **csh** (which cannot hide stderr).

--tmux

mkfifo *tmpfile*; tmux new-session -s p*PID* -d -n <<shell quoted input>> \(<<shell quoted input>>\)\;\ perl\ -e\ \'while\(\$t++\<3\)\{\ print\ \$ARGV\[0\],\"\\n\"\ \}\'\ \$\?h/\$status/255\ \>\>\ *tmpfile*\&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*

The input is used as the name of the windows in **tmux**. When the job inside **tmux** finishes, the exit value is printed to a fifo. This fifo is opened by perl outside **tmux**, and perl then removes the fifo (but keeping it open). 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/255 and this is parsed by perl.

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

The ordering of the wrapping is important:

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

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-shellshock versions of **bash**.

Remote Ctrl-C and standard error (stderr)

If the user presses Ctrl-C the user expect 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. The wrapper looks like this:

```
$SIG{CHLD} = sub { $done = 1; };
$pid = fork;
unless($pid) {
    # Make own process group to be able to kill HUP it later
    setpgrp;
    exec $ENV{SHELL}, "-c", ($bashfunc."@ARGV");
    die "exec: $!\n";
}
do {
    # Parent is not init (ppid=1), so sshd is alive
    # Exponential sleep up to 1 sec
    $s = $s < 1 ? 0.001 + $s * 1.03 : $s;
    select(undef, undef, undef, $s);
} until ($done || getppid == 1);
# Kill HUP the process group if job not done
kill(SIGHUP, -${pid}) unless $done;
wait;
exit ($?&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 the a shell that runs the function definition followed by the actual command.

env_parallel (mentioned in the man page) copies the full current environment into the environment variable **parallel_bash_environment**. This variable is picked up by GNU **parallel** and used to create the Perl script mentioned above.

Base64 encode 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.

```
@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.

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:

```
#!/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**.

Quoting

Quoting is kept simple: Use \ for all special chars and ' for newline. Whether a char is special depends on the shell and the context. Luckily quoting a bit too many does not break things.

It is fast, but had the distinct disadvantage that if a string needs to be quoted multiple times, the \'s double every time - increasing the string length exponentially.

--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 100 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 often more than 1000 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 > 32768 ? 32768 : $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
```

--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.

Buffering on disk

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.

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 when a job finishes. 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.

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 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 simple to make regular expressions: \257 is disallowed on the command line, so when that is matched in a regular expression, it is known that this is a replacement string.

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.

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 jobs 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.

Ideas for new design

Multiple processes working together

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

Transferring of variables and functions from zsh

Transferring Bash functions to remote zsh works. Can parallel_bash_environment be used to import zsh functions?

--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

--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), 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.

Transferring of variables and functions

Until 20150122 variables and functions were transferred by looking at $SHELL to see whether the shell was a ***csh** shell. If so the variables would be set using **setenv**. Otherwise they would be set using **=**. The caused the content of the variable to be repeated:

echo $SHELL | grep "/t\{0,1\}csh" > /dev/null && setenv VAR foo || export VAR=foo