## NAME

**gourd** - A tool for scheduling parallel runs for algorithm comparisons.

## SYNOPSIS

**gourd** *command* [**-s**] [**-c** *filename*] [**-v**|**-vv**] [**-d**] [**-h**]

## DESCRIPTION

**gourd** is a tool that schedules parallel runs for algorithm comparisons. Given the parameters of the experiment, a number of test datasets, and algorithm implementations to compare, **gourd** runs the experiment in parallel and provides many options for processing its results. While originally envisioned for the DelftBlue supercomputer at Delft University of Technology, **gourd** can replicate the experiment on any cluster computer with the **Slurm** scheduler, on any UNIX-like system, and on Microsoft Windows.

New to **gourd**? Go directly to the **gourd-tutorial(7)** manual.

## GLOBAL OPTIONS

The following options apply to all **gourd** commands. These will be referred to as *GLOBAL OPTIONS* throughout the manual.

**-c *filename*, --config *filename*** Tell **gourd** to use the given filename as `gourd.toml`, the configuration file that defines the experimental setup. By default, the file is expected in the current working directory at `./gourd.toml`.

**-d, --dry-run** Run **gourd** in dry-run mode, printing all operations (such as writing to files or scheduling runs) without executing them.

**-h, --help** Display usage instructions for the **gourd** utility or any of its commands. This option extends to all the subcommands of **gourd**: for example, running **gourd** *status -h* will display help about the *status* subcommand.

**-s, --script** Tell **gourd** to use a script-friendly interface, that is, one that does not use interactive user prompts.

**-v, -vv, --verbose** Run **gourd** with verbose debugging output, where -vv enables even more logging.

## COMMANDS

Using **gourd** is as simple as invoking one of its commands, such as **gourd** *init*. Command-line arguments are generally not necessary; to design and run experiments, a `gourd.toml` file should be in the current directory. The following is a summary of available commands.

**gourd *run*** Create an experiment from configuration and run it on **Slurm** or the local machine.

**gourd *init*** Set up a template of an experiment configuration.

**gourd *status*** Display the status of a running or completed experiment.

**gourd *continue*** Schedule the incomplete part of a partial experiment.

**gourd *cancel*** Cancel scheduled runs.

**gourd *rerun*** Rerun (possibly) failed runs.

**gourd *analyse*** Output metrics of completed runs.

**gourd *set-limits*** Change **Slurm** resource limits for runs not yet scheduled.

**gourd *version*** Show the software version.

## GOURD RUN

### Summary

The **gourd** *run* command uses the provided configuration and runs an experiment. Using either *local* or *slurm*, it is possible for the execution to run on the local machine, or be scheduled using Slurm on a cluster computer. Using Slurm, additional configuration arguments are required; see **gourd.toml(5)**.

In principle, however, **gourd** *run* is used in the same way for both Slurm and local execution. A `gourd.toml` configuration file should be present in the current directory, formally describing the experiment that is to be created and run. Because most options are specified in this file, it is usually sufficient to type **gourd** *run slurm*|*local* to run an experiment.

See the manual page **gourd-tutorial(7)** for a step-by-step guide on designing experiments to run.

### Synopsis

**gourd** *run slurm*|*local* [GLOBAL OPTIONS] [**--force**] [**--sequential**]

### Subcommands

***local*** Runs the experiment locally, such that all programs are executed directly on the computer that **gourd** is run from.

This is useful for running **small parts** of an experiment on a personal computer or a login node, allowing you to test that programs are being called correctly and that the configuration is valid. Please note that *local* is **NOT** intended for running full experiments on a **Slurm**-equipped cluster. *Local* will use the login node only and not the actual supercomputer.

Running using *run local* will perform the experiments in parallel based on the number of available CPU cores. Resource limits set in the configuration will not be honoured. While running, experiment status is displayed continuously (see the **gourd** *status* command) until all runs have finished executing. Typing Control+C into the terminal will stop the runs.

*local* can additionally take more options:

**--force** **gourd** will, by default, refuse to run large experiments on *local*. This is because doing so may rapidly use up too many file descriptors on some operating systems. To ignore the warning and run the experiment anyways, use this optional flag. If the resources are exhausted too quickly, an error will be displayed.

**--sequential** By default, runs execute concurrently with a level of parallelism. This option can be specified to force the runs to run sequentially, that is, one after another. This may be useful if you want to run bigger experiments without using too many system resources. Note that the use of this option **also** enables the **--force** option.

***slurm*** Runs the experiment on a **Slurm**-equipped cluster computer.

In this mode, **gourd** will use the **Slurm** command-line interface to schedule runs on a supercomputer. The prerequisites are that:

- **gourd** is running on the login node of a supercomputer, such that the **srun** command is available.
- `gourd.toml` contains all required fields for running on **Slurm** (see the manual for **gourd.toml**(5))
- `gourd.toml` contains a valid experiment for which all paths (including the programs and output paths) are accessible from the cluster nodes.

When **gourd** *run slurm* is called, the experiment's runs are not executed immediately; instead, they are submitted as *job arrays* to the **Slurm** scheduler. The experiment's runs are then in the supercomputer's queue (status "pending"). The time until the runs are actually executed depends on many factors, which may include the current load and the size of your experiment; this delay can range from seconds to days. For this reason, **gourd** *run slurm* does not show the continuous status of an experiment, use **gourd** *status* to do that.

On successful scheduling, the Slurm IDs of the job arrays that make up the experiment will be shown, and the command will exit. To view the experiment's status, see the **gourd** *status* section of this manual.

Running on Slurm has many configurable options. Please refer to the manual **gourd-tutorial**(1) for example setups and the manual **gourd.toml**(5) for complete reference. The implementation of the Slurm API used by **gourd** is discussed in depth in the **gourd** maintainer documentation.

## GOURD INIT

### Summary

The **gourd** *init* command creates an experimental configuration. Configurations are represented as TOML files. A template configuration, `gourd.toml`, is created in the directory specified. The directory can optionally be initialized as a Git repository. Unless run with the [**-s**] flag, this command will ask using interactive prompts to refine the template to your needs.

If the command is run with the [**-s**] flag these choices will not be offered and the default options will be picked for all queries.

### Synopsis

**gourd** *init* [GLOBAL OPTIONS] [**-e** *example-name*] [**--list-examples**] [**--git**=*true|false*] [*directory*]

### Options

**-e, --example** ***example-name*** Initializes the given directory with an example configuration from **gourd-tutorial(7)** (rather than a custom template for `gourd.toml`).

**--list-examples** Instead of initializing a folder, this will make **gourd** list all the available examples for the *-e* option.

**--git=*true|false*** Whether to initialize an empty git repository in the newly created folder.

### Listing Examples

If **--list-examples** is used, **gourd** *init* will not initialize a new folder with a configuration. The *directory* argument will be ignored.

A list of available examples and their descriptions will be printed to the output and the program will exit.

## GOURD STATUS

### Summary

The **gourd** *status* command displays the status of an existing experiment, that is, one that has been created by **gourd** *run*, but not necessarily one that has fully executed. This command can also display detailed status of an individual run using the **-i** flag.

### Synopsis

**gourd** *status* [GLOBAL OPTIONS] [**-i** *run-id*] [**--follow**] [**--full**] [**--after-out**] [*experiment-id*]

### Options

***experiment-id*** The ID of an experiment to show the status of. By default, this is the most recent experiment.

**-i** ***run-id*** Instead of showing a general overview of the entire experiment show detailed information about a run with this *run-id*.

**--full** By default, *status* displays a summary rather than a full list if there is a large number of runs (>100). Using **--full**, the full list is always shown.

**--follow** The status will be continually displayed until all of the runs have finished. This is useful when it is known that the jobs will finish in a matter of minutes.

**--after-out** Use only with **-i** *run-id*, displays the raw afterscript output for that run.

### Experiment status

By default, **gourd** *status* uses the `gourd.toml` file to determine the location of experiment files generated using **gourd** *run*. It finds the most recent experiment (unless [*experiment-id*] is specified) and shows a summary containing the status of each run, and, if completed, the run's basic timing metrics. The command also shows a summary of each run's error status, if any.

### Run status

With the **-i** *run-id* argument, **gourd** *status* will retrieve detailed run information including the arguments that the binary was called with, RUsage metrics if successful, and detailed error status if it has failed. The file paths provided make it easy to inspect the output of a run, whether it has succeeded or failed.

### Afterscripts

To postprocess the output of the runs, there are two options available: *afterscipts* and *pipelining*. Afterscripts are scripts that run locally (so for DelftBlue they do *not* get scheduled as separate jobs).

Afterscripts are meant for quick and computationally inexpensive postprocessing (such as getting the first line of the output file). For long or complicated postprocessing with a significant computational cost, look at *Pipelining*.

- An afterscript is optional and specified per program.
- To indicate the use of an afterscript, the path to the script file needs to be specified in the `gourd.toml` under the chosen program.
- Multiple programs can use the same script.
- The afterscript can be used to assign labels to runs as a means of specifying custom status.

*How to design an afterscript:*

The afterscript should be an executable file. This can be a normal compiled executable, or possibly a shell/python script if you use the appropriate *shebang* at the start of the file (check out https://en.wikipedia.org/wiki/Shebang_(Unix) for details).

**gourd** will pass the path to a file containing the main program's output to the afterscript as a command line argument. The afterscript can then print any output to `stdout` (via `print`, `printf`, `println`, `echo` or your preferred language's method), and **gourd** will collect that and display it in **gourd** *status -i <specific run id>*

*What can you do with afterscript output*

- use **labels**, check out the corresponding chapter for more details.
- create custom metrics: read in **gourd** *analyse* for how to do this.

## GOURD CONTINUE

### Summary

The **gourd** *continue* command schedules runs that are part of an existing experiment, but have not yet been scheduled. This includes runs created by **gourd** *rerun*, as well as runs that were not scheduled due to a run limit. For example, an experiment with 30,000 distinct runs can be scheduled in three batches of 10,000 each if that is the maximum number of queued supercomputer jobs.

### Synopsis

**gourd** *continue* [GLOBAL OPTIONS] [*experiment-id*]

### Options

***experiment-id*** The ID of an experiment to continue. By default, this is the most recent experiment.

### Pipelining

Programs may be pipelined by specifying the `next` programs in the sequence:

```
[program.your_first_progarm]
binary = "./executable"
next = ["a_second_program", "another_second_prog"]

[program.a_second_program]
binary = "./executable2"

[program.another_second_prog]
binary = "./executable3"
```

See the manual for `gourd.toml` for more details on configuration.

In the example above, when the runs for `your_first_progarm` finish, running **gourd** *continue* will start one run for `a_second_program` and one for `another_second_prog`, both of which will receive as input (to `stdin`) the output (`stdout`) of `your_first_progarm`.

## GOURD CANCEL

### Summary

The **gourd** *cancel* command cancels runs that have been scheduled on Slurm. By default, it cancels all scheduled runs in the most recent experiment. This command can cancel an individual run using the **-i** flag.

### Synopsis

**gourd** *cancel* [GLOBAL OPTIONS] [*experiment-id*] [**-i** *run-ids*] [**-a**]

### Options

***experiment-id*** The ID of an experiment to cancel runs from. By default, this is the most recent experiment.

**-i** ***run-ids*** The IDs of the runs to cancel. Pass multiple run IDs separated by spaces, for example *-i 1 2 3*. By default, all runs in the experiment are cancelled.

**-a, --all** Cancel all runs from this account. This includes all runs, not just those from **gourd**.

### Cancelling All Runs

Cancelling all runs will **cancel all runs scheduled on your account**. This option is included to be able to cancel past or deleted experiments. But be aware of its possible impact.

You can see which runs would be cancelled without actually doing it by running **gourd** *cancel --all --dry*.

### Latency

Slurm may take some time to acknowledge the cancellation; thus, running **gourd** *status* right away after a cancellation may still display the runs as pending, please wait up to one minute for the changes to propagate.

## GOURD ANALYSE

### Summary

The **gourd** *analyse* command collects and processes metrics generated when an experiment was run. It can produce a CSV data file or a "cactus plot" to compare how quickly different algorithms run.

### Synopsis

**gourd** *analyse* [*experiment-id*] *table*|*plot* [GLOBAL OPTIONS] [**-o** *path/to/file*] [**-f** *format options*]

### Options

***experiment-id*** The ID of an experiment to analyse. By default, this is the most recent experiment.

**-o** ***path/to/file***, **--output** ***path/to/file*** Pass the command's output to a file.

**-f** ***format options***, **--format** ***format options*** Formatting options for the *table* and *plot* subcommands.

### Metrics CSV

Running **gourd** *analyse table* will create a table with data about the status of the runs, metrics, and afterscript completion, unless there are no completed runs. If the *-o* option is not passed, the table will be pretty-printed in the command line, otherwise a CSV file will be saved to the specified path. The CSV generation will take into account all runs of the experiment. If **gourd** *analyse* is rerun, the CSV will be updated with the newest status of the runs.

The option `--format` takes a comma-separated list of columns to use in the table. Without specifying this option, **gourd** will default to `--format="program,slurm,fs-status,wall-time"`. The first column will always contain the `run id`. The possible options are:

`program` program name

`file` the input file this run was executed with, if there was one

`args` command-line arguments passed to the program

`group` the input group, if there is one

`label` any label-associated status

`afterscript` afterscript status string

`slurm` run status retrieved from the slurm daemon

`fs-status` run status retrieved from the file system

`exit-code` program's exit code

`wall-time` total elapsed real (wall-clock) time

`user-time` CPU time spent in user mode

`system-time` CPU time spent in kernel (system-call) mode

`max-rss` peak resident set size (maximum RAM used)

`ix-rss` integral of shared memory size over the run (ru_ixrss)

`id-rss` integral of unshared data segment size (ru_idrss)

`is-rss` integral of unshared stack size (ru_isrss)

`min-flt` number of minor page faults

`maj-flt` number of major page faults

`n-swap` total swap operations performed

`in-block` number of block input operations (disk reads)

`ou-block` number of block output operations (disk writes)

`msg-sent` number of inter-process messages sent

`msg-recv` number of inter-process messages received

`n-signals` number of signals delivered to the process

`nv-csw` voluntary context switches count

`n-iv-csw` involuntary context switches count

### Cactus plots

Running **gourd** *analyse plot* will create a PNG picture of a cactus plot. The cactus plot is used to showcase the comparison of how many inputs each algorithm can finish running with in a given amount of time. In other words, the horizontal axis represents the time passing, and the vertical axis represents how many runs of this program (algorithm) have already finished. This allows to see a visual comparison of the time each program takes - the more runs there are, the more informative the plot will result to be. The plot will take into account only the runs that have completed and have valid RUsage data. If **gourd** *analyse plot* is rerun, the graph will be updated according to the newest available data.

The option `--format` can be used to specify whether the plot output should be in PNG or SVG format, for example: **gourd** *analyse plot –format="png"* (png is also the default output)

## GOURD VERSION

### Summary

**gourd** *version* outputs the software version and exits. Using the [**-s**] flag will display the version only, otherwise **gourd** will stress-test your terminal font.

### Synopsis

**gourd** *version* [**-s**]

### Scripting

By default **gourd** *version* shows a human readable only output. By running **gourd** *version* **-s** one can obtain a version number in the format:

```
gourd <version number>
```

## EXAMPLES

See the section on **gourd** *init* for runnable example directories. For a more detailed walkthrough with more focus on examples, use the **gourd-tutorial(7)** manual.

## FILES

`gourd.toml` A configuration file containing the experiment details. See **gourd.toml(5)**.

`<experiment-dir>/<experiment-number>.lock` A file containing the runtime data of the experiment.

## SEE ALSO

**gourd-tutorial(7)**

**gourd.toml(5)**

## CONTACT

Ανδρέας Τσατσάνης <a.tsatsanis@tudelft.nl>
Lukáš Chládek <l@chla.cz>
Mikołaj Gazeel <m.j.gazeel@tudelft.nl>