NAME

gourd.toml An experimental setup file for the Gourd experiment scheduler.

DESCRIPTION

gourd.toml is a declarative configuration file that, when provided to **gourd(1)**, can be used to create a new *experiment*. The rules for typesetting the file follow the standard TOML format.

By default, the filename is `gourd.toml`.

PREFACE

Values ending with '?' can be omitted, the default value for them is mentioned in their description.

VALUE TYPES

The configuration uses the following value types.

string A string of characters surrounded by "".
path A file path surrounded by "".
fetched_path Information about this is in the **RESOURCE FETCHING** section.
boolean Either *true* or *false*.
number Zero or a positive number.
regex A regular expression.
duration A human-readable amount of time, for example: "2d 4h".
list of T A list of values of type T surrounded by [].

GLOBAL CONTEXT

These should be put at the beginning of the file before defining any sections.

output_path = path This path specifies where to store the stdout and stderr outputs for *programs*.
metrics_path = path Where to store the metrics for **gourd(1)** *status*. These metrics contain information such as: Wall clock time, User time, System time, the amount of context switches etc.
experiments_folder = path Where to store state of previously ran experiments.
Essentially this folder specifies where **gourd** will store all of its information about experiments. If this folder is removed **gourd** looses all information about past experiments.
wrapper? = path Defines the path to the `gourd_wrapper` binary.

The default value is *gourd_wrapper*. (That is, **gourd** will look for it in the `$PATH`)

If you installed **gourd** correctly this values should not be changed.
input_schema? = path Defines the path to an optional machine generated input schema.

For more information about this continue to the **INPUT SCHEMA** section.

The default values is no input schema.

SLURM

The configuration contains some Slurm specific options namely:

SLURM CONFIGURATION

This section is optional for running locally, but required for running on Slurm.

A Slurm configuration starts with the header *[slurm]* and can contain the following options:

experiment_name = string The name under which runs for this experiment will be scheduled on Slurm.

output_folder = path The folder where the raw slurm outputs will be stored. This can be useful for debugging.

partition = string The partition on which this should be run on the supercomputer. Running **gourd** *run slurm* with an invalid partition will display all the valid partitions.

array_size_limit? = number This specifies the limit of runs that can be put in one Slurm batch. **gourd** will work to split the workload such that this limit is never exceeded.

By default this is auto-detected.

max_submit? = number This specifies the limits of possible batches of runs. Once again **gourd** will work to never cross this limit.

By default this is auto-detected.

account = string Which account to use for running jobs on Slurm. For example one account available on DelftBlue is "Education-EEMCS-MSc-CS". To get a list of available accounts on your cluster, you can use Slurm's **sacctmgr** *show account* command

mail_type = string Choose one of Slurm's options for sending emails when a run's status changes. Default is "NONE". Valid options are: `"NONE", "BEGIN", "END", "FAIL", "REQUEUE", "ALL", "INVALID_DEPEND", "STAGE_OUT", "TIME_LIMIT", "TIME_LIMIT_90", "TIME_LIMIT_80", "TIME_LIMIT_50", "ARRAY_TASKS"`

mail_user = string Your email address, if you want to receive email updates from Slurm.

begin = string Submit runs to the Slurm controller immediately, like normal, but tell the controller to defer the allocation of the job until the specified time.

Time may be of the form `HH:MM:SS` or `HH:MM` to run a job at a specific time of day, or in the `now+` format (for example, `now+1hour`, or `now+32min`)

additional_args? = list of string Custom arguments for Slurm.

By default there are no additional arguments.

Example

An example Slurm Configuration:

```
[slurm]
experiment_name = "my test experiment"
output_folder = "./slurmout/"
partition = "compute"
account = "Education-EEMCS-MSc-CS"
```

RESOURCE LIMITS

To run on Slurm one must also specify resource limits. The available limits are:

time_limit = duration The global time limit for all program-input pairs.
cpu_limit = number The global cpu limit for all program-input pairs.
mem_per_cpu = number The global memory limit per one cpu for all program-input pairs.

This number is specified in megabytes.

Example

An example Resource Limits section:

```
[resource_limits]
time_limit = "5min"
cpus = 1
mem_per_cpu = 512
```

LOCAL

num_threads? = number How many threads should **gourd** *run local* use.

NUM_THREADS

For the parallel execution of **gourd** *run local* you can limit the number of threads that will be used by specifying:

```
[local]
num_threads = 8
```

If this option is not specified, the program will try to detect the number of CPUs present on the system, and use that number of threads. Setting a value of 0 will result in a number of threads equal to the number of runs in the program (and the OS will limit the resource use thereafter).

PROGRAMS

Multiple programs can be specified. A program represents a compiled executable and is a combination of a binary file and parameters. Each program begins with *[programs.program-name]*, where *program-name* can be any unique name.

binary = path Path to the program executable.
fetch = fetched_path URL to the program executable.
git = git_object See the **PROGRAM VERSIONS** section for more information.

arguments? = list of string Arguments to be passed to the executable.

By default an empty list.

afterscript? = path See the **AFTERSCRIPTS** section for more information.

By default there is no afterscript.

next? = list of string See the **POSTPROCESSING** section for more information.

By default there is no postprocessing.

resource_limits? As defined in the **RESOURCE LIMITS** section.

These essentially override the global resource limits for this program.

By default, use the global resource limits.

Only one of **binary**, **fetch**, **git** must be specified.

EXAMPLE

Assume that there is a script called `test.sh` in the current directory. We can specify a program that runs this script with the argument **--test** like so:

```
[program.some_name_for_this_program]
binary = "./test.sh"
arguments = ["--test"]
```

INPUTS

A **gourd(1)** experiment consists of a cross-product mapping between programs and inputs. The experiment created from a `gourd.toml` file runs every combination of program and input in the file.

Multiple inputs can be specified. Each input begins with *[inputs.input-name]* where *input-name* can be any unique name. **The string _i_ is reserved and cannot be used.** Each input contains the following keys:

file? = path Path to a file, the contents of which are passed to the program as standard input.

By default, no standard input is provided.

fetch? = fetched_path URL to a file, the contents of which are passed to the program as standard input.

glob? = fetched_path A glob expression of multiple files, the contents of which are passed to the program as standard input.

arguments? = list of string Additional command-line arguments to be passed to the program. The input arguments are appended to the programs arguments.

By default, there are no additional arguments.

Only one of **file**, **fetch**, **glob** can be specified.

EXAMPLE

A valid input would be for example:

```
[input.some_name_for_this_input]
file = "./test.txt"
arguments = ["--a", "--b"]
```

This applied to program '`program`' would be equivalent to:

```
program [program args] --a --b < ./test.txt
```

GLOBS

Globs can be applied to arguments of inputs and conveniently reference multiple files.

If an argument starts with *path|*, it will be treated as a glob. The input will be instantiated for every match of the provided glob.

Example

```
[inputs.testrun1]
arguments = ["-f1", "path|./inputs/*.in", "-f2", "path|./input2/*.in"]
```

Given that the current directory contains the files `input/1.in`, `input/2.in`, `input2/test.in`, the glob expands to the following experiment inputs:

```
[inputs.testrun1_glob_0]
arguments = ["-f1", "./inputs/1.in", "-f2", "./input2/test.in"]

[inputs.testrun1_glob_1]
arguments = ["-f1", "./inputs/2.in", "-f2", "./input2/test.in"]
```

PARAMETERS

Parameters can be applied to arguments to conveniently perform experiments with grid search (a Cartesian product between all parameter values).

If an argument starts with *param|some-parameter-name*, it will be treated as a parameter. For each value of that parameter the new input will be created with that value inserted into the argument into that argument place.

Values of a parameter are specified in *[parameter.name]* using *values = list of string*

This results in cross product between all parameters.

Example

```
[inputs.testrun1]
arguments = ["-f", "param|x", "-x", "param|y"]

[parameters.x]
values = ["a", "b"]
```

```
[parameters.y]
values = ["10", "20"]
```

It will be transformed into following inputs:

```
[inputs.testrun1_x_0_y_0]
arguments = ["-f", "a", "-x", "10"]

[inputs.testrun1_x_0_y_1]
arguments = ["-f", "a", "-x", "20"]

[inputs.testrun1_x_1_y_0]
arguments = ["-f", "b", "-x", "10"]

[inputs.testrun1_x_1_y_1]
arguments = ["-f", "b", "-x", "20"]
```

SUBPARAMETERS

Subparameters are used when there is a need for 1-1 relation between two parameters. There is no cross product between subparameters of the same parameter.

Subparameters are specified in inputs similarly to parameters with the difference of doing *subparam|parameter-name.some-subparameter-name.*

Values of a subparameter are specified in *[parameter.name.sub.subparameter-name]* using *values = []*

Note! Parameters can have either values or subparameters with values. Never both.

Example

This example:

```
[input.testrun1]
arguments = ["-f", "subparam|x.1", "-x", "param|y", "-g", "subparam|x.2"]

[parameter.x.sub.1]
values = ["a", "b"]

[parameter.x.sub.2]
values = ["c", "d"]

[parameter.y]
values = ["10", "20"]
```

Will be transformed into following inputs:

```
[input.testrun1_x-0_y-0]
arguments = ["-f" "a", "-x", "10", "-g", "c"]

[input.testrun1_x-0_y-1]
arguments = ["-f", "a", "-x", "20", "-g", "c"]
```

```
[input.testrun1_x-1_y-0]
arguments = ["-f", "b", "-x", "10", "-g", "d"]

[input.testrun1_x-1_y-1]
arguments = ["-f", "b", "-x", "20", "-g", "d"]

[parameter.x]
values = ["a", "b"]

[parameter.y]
values = ["10", "20"]
```

Where as this example:

```
[input.testrun1]
arguments = ["param|x"]

[parameter.x.sub.1]
values = ["a", "b"]

[parameter.x.sub.2]
values = ["c", "d"]

[parameter.x]
values = ["10", "20"]
```

Is not correct and **gourd** will throw an error!

POSTPROCESSING

Postprocessing jobs are jobs that run after another job and can transform its input without influencing the original jobs input.

Postprocessing programs are just normal programs but they a different program references them in their `next` field.

Postprocessing programs are ran in the same directory as the original job, and get the originals job `stdout` as their `stdin`.

EXAMPLE

```
[program.test_program]
binary = "./algorithm"
arguments = []
next = ["example_name"]

[program.example_name]
binary = "./verifier"
arguments = []
```

AFTERSCRIPTS

Afterscripts are postprocessing but one that does not constitute a full Slurm job.

These are ran when **gourd** *status* is invoked, and their results are cached.

The afterscript is assumed to:

- Be executable.
- Will receive the path to the jobs output as the first CLI parameter.
- Print its output to the standard output `stdout`.

EXAMPLE

```
[program.test_program]
binary = "./algorithm"
arguments = []
afterscript = "./script"
```

After running the job the after script will be called as:

```
script path/to/job/stdout
```

And for example if:

in **script**:

```
#!/bin/sh
cat $1
```

The afterscript's output will be exactly the jobs output (ie. No postprocessing happened). But these scripts may be more complex if the use case requires it.

LABELS

When running **gourd** *status*, by default the statuses only display information about Slurm scheduling of the run or an exit code.

In the case that a job execution can succeed (exit code 0) but the run should still be considered a failure, the user can add a custom label to the run, derived from the output of the run's 'afterscript'.

Labels can be created in the configuration file as names with a regular expression, where if the regex is matched in the afterscript's output, the label is assigned to the run.

These are specified as *[label.label-name]* and the fields available are:

regex = regex A regular expression that the afterscripts output will be matched to, if the output matches the expression this label will be assigned.

priority = number In the case that more than one label matches a run the **highest** priority label will be assigned. Higher priority value = higher priority. Default is 0. Note that if two or more labels have the same priority and are both present at the same time, the result is undefined behaviour. Set 'warn_on_label_overlap' to 'true' to prevent this.

rerun_by_default? = boolean If true makes this label essentially mean 'failure', in the sense that **gourd** will treat a run with this label as a failure even if the run itself succeeded.

By default *false*.

EXAMPLE

```
[label.label_name]
# matches any output
regex = ".*"
priority = 1
rerun_by_default = true
```

Labels are assigned based on priority. For example if the configuration file looks like:

```
[label.label1]
regex = "Success"
priority = 1
rerun_by_default = false

[label.label2]
regex = "RuntimeException"
priority = 2
```

and the afterscript output looks like:

```
Starting afterscript...
Success! The output was correct.
Verifying something else...
RuntimeException thrown while parsing
```

then by principle of priorities, the run will be assigned *label2* even though both regexes match.

REMOTE RESOURCE FETCHING

In order to prevent having to manually transfer large files, input files or (precompiled) program binaries can be fetched from a URL.

Suppose you are hosting a large text file, or want to download a binary from CI artefacts:

`https://test.com/input.txt`

`https://test.com/program.exe`

Any config field which accepts `fetched_path` can accept remote resources.

The syntax for fetched resources is "`remote_path | local_path`" (whitespace insensitive).

EXAMPLE

Consider this fetched input as an example:

```
[input.some_input]
```

```
fetch = "https://test.com/input.txt | ./path/to/store/the/file.txt"
arguments = ["any", "input", "arguments"]
```

This will download the file at `test.com/input.txt` and save it contents in the provided path. The contents of this will then be passed as input to all programs.

Note that the "|" character needs to be escaped in URLs as `%7C` since it is used as a delimiter

Similarly for programs:

```
[program.some_example]
fetch = "https://test.com/program.exe | ./path/to/store/the/program.exe"
arguments = ["any", "program", "arguments"]
```

CACHING

These resources will be downloaded and saved at the paths, but they will not be redownloaded again as long as these cached files exist.

It may be beneficial to create a folder and store all downloaded resources inside, then when there is a need for cleaning the cache this amounts to deleting the folder

PROGRAM VERSIONS

Programs may be fetched and compiled straight from a git repository.

The user in this case has to provide the commit ID of the desired HEAD, the build command and the path to the output binary.

The build command and the path to the output binary are both ran relative to the repository root.

EXAMPLE

```
[program.testprogram1]
arguments = ["a"]

[program.testprogram1.git]
git_uri = "https://github.com/Nerdylicious/DijkstraShortestPath.git"
build_command = "g++ ./DijkstraShortestPath.cpp"
path = "./a.out"
commit_id = "e90e7f6811f399075bc058f12e2324fb64701b02"
```

This will clone the repository, check it out at the correct point in time build the Dijkstra algorithm and finally run it.

CACHING

Similarly to fetching, delete the repository folder if you want to refetch the files.

INPUT SCHEMA

The `input_schema` field can be specified with a file that contains an additional list unnamed of inputs.

This is an option to allow for script-generated input lists, in case the structure of the files cannot be expressed by a glob pattern.

Avoid using this if possible.

EXAMPLE

Assume that `gourd.toml` specifies:

```
input_schema = "./inputs.toml"
```

And a file in the same folder called `inputs.toml` exists, containing:

```
[[input]]
file = "./jeden"

[[input]]
file = "./dwa"
```

We have just added two new inputs programatically to the input list.

These inputs have all of the fields of normal inputs, but they do not support naming and will always have automatically assigned names.

SEE ALSO

gourd(1) gourd-tutorial(7)

CONTACT

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