NAME

gourd-tutorial - A step-by-step walkthrough for the Gourd experiment scheduler.

INTRODUCTION

Welcome to **gourd-tutorial!**

If you haven't been introduced yet, **gourd(1)** is an application that makes it easy to set up experiments on a supercomputer. By experiment, we mean a large-scale comparative evaluation of one or more *algorithms* (runnable programs) that each run on a set of *inputs* and are subsequently timed and profiled.

While this tool offers a lot of versatility, this set of runnable examples will show that **gourd** experiments only take a minute to set up.

INSTALLATION AND REFERENCE

This tutorial is designed to be interactive, so be sure to have a working copy of **gourd(1)** installed on your computer. You can verify this by typing **gourd** *version* in a terminal. For installation instructions, refer to the `README.md` file in the source repository.

When installed, you will also have access to the user manuals. For Linux, macOS, and the like, type **man gourd-tutorial** to see this tutorial or **gourd** and **gourd.toml** for complete documentation.

INTERACTING WITH GOURD

Gourd is a command-line application that keeps life easy. You take actions by typing **gourd** followed by a command in your terminal; a complete list is in the manual.

For example, type: **gourd** *init --example fibonacci-comparison my_fib*

The **gourd** *init* command will set up the `myexample` folder to match the example below! Furthermore, **gourd** *init* **--list-examples** will show what other examples are accessible to you.

FIBONACCI COMPARISON

Let's begin by designing a simple experiment. We will compare three versions of an algorithm that calculates Fibonacci numbers.

First, let's define the experimental setup using a `gourd.toml` file. This file will specify the files, programs, and parameters of our setup in a reproducible way.

Open `gourd.toml` in an editor and type in the following lines:

```
                              ./gourd.toml
  +-----------------------------------------------------------------+
1 | experiments_folder = "experiments"                              |
2 | metrics_path = "experiments"                                    |
3 | output_path = "experiments"                                     |
4 |                                                                 |
  /_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _/
```

```
`   \_____________________________________________________________\
```

In the TOML format, values (such as file paths) are in quotes ("). You can also add comments using the hash character.

The lines above set up the folder structure for our experiment's outputs. This particular setup puts everything in the same folder.

Now, let's configure programs - the algorithms we are evaluating.

Defining programs

```
    _____________________________________________________________
`   /_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _/
`   \                        ./gourd.toml                         \
`   |` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `|
5   | [program.fibonacci]                                         |
6   | binary = "./fibonacci"                                      |
7   |                                                             |
8   | [program.fast-fibonacci]                                    |
9   | binary = "./fibonacci-dynamic"                              |
10  |                                                             |
11  | [program.fastest-fibonacci]                                 |
12  | binary = "./fibonacci-dynamic"                              |
13  | arguments = ["-f"]                                          |
14  |                                                             |
`   /_ ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` _/
`   \_____________________________________________________________\
```

The lines above set up three uniquely named programs:

fibonacci: a slow Fibonacci number calculator.

fast-fibonacci a faster version using Dynamic Programming.

fastest-fibonacci: the same binary file as *fast-fibonacci* run with an additional command-line argument, **-f**, which should make it even faster!

Each program links to a *binary* – the executable file that runs our algorithm. In this case, our Fibonacci algorithms are compiled in Rust. If you are following this tutorial with **gourd** *init* – *example fibonacci-comparison*, the folder contains both binaries: `fibonacci` and `fibonacci-dynamic`.

In our evaluation, we are going to see how the three programs compare when running different test cases as inputs. Let's add inputs to our `gourd.toml`.

Defining inputs

```
    _____________________________________________________________
`   /_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _/
`   \                        ./gourd.toml                         \
`   |` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `|
15  | [input.test_2]                                              |
16  | input = "./inputs/input_2"                                  |
17  |                                                             |
18  | [input.test_8]                                              |
```

```
19 | input = "./inputs/input_8"                                          |
20 |                                                                     |
21 | [input.test_35]                                                     |
22 | input = "./inputs/input_35"                                         |
23 |                                                                     |
24 | [input.bad_test]                                                    |
25 | input = "./inputs/input_bad"                                        |
26 |                                                                     |
`  /`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`-`/
`  \_____________________________________________________________________\
```

The lines above set up four uniquely named inputs. Each input refers to a file whose contents are fed into the program.

In this example, inputs `test_2`, `test_8`, and `test_35` link to files containing the numbers `2`, `8`, and `35`. These should make the Fibonacci algorithms output the 2nd, 8th, and 35th numbers of the Fibonacci sequence. The input named `bad_test` contains `"some text"`, which isn't a valid number - let's see how this will crash the programs.

Inputs are combined with programs in a **cross product** to create *runs*. Each program-input combination is exactly one *run*. In this example, 3 programs * 4 inputs results in 12 *runs*.

Running the evaluation

Our `gourd.toml` is complete - now it is time to run the evaluation using **gourd** *run*. Typing **gourd** *run* in a terminal will tell you that it has two subcommands:

local	Run locally on your computer. If connected via SSH to a cluster computer, *local* uses the very limited computing power of the login node.
slurm	Send to the SLURM cluster scheduler on a supercomputer.

The *slurm* subcommand needs some extra configuration, so let's go with *local* for now. Type **gourd** *run local*.

```
|   $ gourd run local
|
| > info: Experiment started
| >
| > For program fast-fibonacci:
| >    0. bad_test.... failed, code: 25856
| >    1. test_2...... success, took: 171ms 903us 417ns
| >    2. test_35..... success, took: 172ms 2us 417ns
| >    3. test_8...... success, took: 175ms 546us 750ns
| >
| > For program fastest-fibonacci:
| >    4. bad_test.... failed, code: 25856
| >    5. test_2...... success, took: 149ms 219us 542ns
| >    6. test_35..... success, took: 154ms 733us 667ns
| >    7. test_8...... success, took: 146ms 695us 334ns
| >
| > For program fibonacci:
| >    8. bad_test.... failed, code: 25856
```

```
| >    9. test_2...... success, took: 272ms 265us 667ns
| >   10. test_35..... success, took: 328ms 935us 292ns
| >   11. test_8...... success, took: 273ms 159us 167ns
| >
| >
| > [ ] ################### Running jobs... 12/12
| > info: Experiment finished
| >
| > info: Run gourd status 1 to check on this experiment
```

If you are seeing similar output, you have successfully reproduced a Gourd experiment!

Displaying status

The *run* command has created an experiment from the experimental setup and executed it on your computer. Each of the twelve runs are shown here, grouped by program, alongside with their completion status. In fact, you can show this view at any time by typing **gourd** *status*.

We can see that runs 0, 4, and 8 have failed. Let's take a closer look at why that is! Type **gourd** *status* **-i** *4* to check on run number 4.

```
|   $ gourd status -i 4
|
| > program: fastest-fibonacci
| >   binary: FetchedPath("/fib-folder/fibonacci-dynamic")
| > input: Regular("bad_test")
| >   file: Some(FetchedPath("/fib-folder/inputs/input_bad"))
| >   arguments: ["-f"]
| >
| > output path: "/fib-folder/experiments/1/fastest-fibonacci/4/stdout"
| > stderr path: "/fib-folder/experiments/1/fastest-fibonacci/4/stderr"
| > metric path: "/fib-folder/experiments/1/fastest-fibonacci/4/metrics"
| >
| > file status? failed, code: 25856
| > metrics:
| >   user   cpu time: 1ms 274us
| >   system cpu time: 1ms 735us
| >   page faults: 1
| >   signals received: 0
| >   context switches: 11
```

The detailed status, which you can see above, allows us to easily inspect the experiment's output and errors by accessing the files at `output path`.

Rerunning failed runs

These files reveal that `bad_test` fails because the Fibonacci programs are expecting a number, but the input is "some text" instead! Let's fix the problem and replace it with 10, a decidedly more valid number.

```
`                          ./inputs/input_bad
`  +-------------------------------------------------------------------+
`  | <<<<<<< new version                                               |
`  | 10                                                                |
`  | =======                                                           |
`  | some text                                                         |
`  | >>>>>>> old version                                               |
`  +-------------------------------------------------------------------+
```

Now we have fixed the problem, and the input called `bad_test` is not so bad after all.

You can imagine that running the whole experiment again when only 1/4 of the results are invalid would be a waste. We are going to use **gourd** *rerun* to repeat only the runs that failed.

```
|    $ gourd rerun
|
| >  ? What would you like to do?
| >  * Rerun only failed (3 runs)
| >    Rerun all finished (12 runs)
| >  [↑↓ to move, enter to select, type to filter]
| >
| >  info: 3 new runs have been created
| >  info: Run 'gourd continue 1' to schedule them
```

The **gourd** *rerun* command suggests rerunning the failed runs only! Another option supported by *rerun* is to specify a list of IDs for it to reschedule.

After *rerun*, it is necessary to use **gourd** *continue* to actually execute the newly created runs. Try this in your terminal.

Collecting data

Our simple Fibonacci experiment is done evaluating our two algorithms. All that remains to be done is collecting the runtime data. Fortunately, **gourd** also provides a simple way to process the numerous metrics files that our runs have generated.

By running **gourd** *analyse table*, you can create a CSV file that collects all metrics from the application's run. On UNIX-like operating systems, RUsage provides a large array of useful data such as context switches and page faults in addition to basic timing.

Furthermore, **gourd** *analyse* supports ways of collecting and visualising the experiment's output. Try the **gourd** *analyse plot*, which produces a cactus-plot summary of the programs' runtimes.

SEE ALSO

gourd(1)

gourd.toml(5)

CONTACT

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