GOURD

MAINTAINER DOCUMENTATION

Lukáš Chládek

Mikołaj Gazeel

Ανδρέας Τσατσάνης

At

Technische Universiteit Delft

version:	1.1.0
generated on:	2025-05-11

Contents

1 Version History

Version 1.2.1 *"Sponge Gourd"*

Minor patch, adds the `local.num_threads` option in the config, as well automatically detecting the number of CPU cores on the system. Added the `gourd --version` command.

Version 1.2.0 *"Sponge Gourd"*

Major internal reworkings, redesigned `gourd analyse`. Past this major release version the project is open source. For more details on this release, see

https://github.com/ConSol-Lab/gourd/pull/19

Version 1.0.2 *"Snake Gourd"*

This patch addresses the following:

- Improvements to how afterscripts are handled.
- Glob arguments now expand to absolute paths before being evaluated.

Version 1.0.1 *"Snake Gourd"*

This patch addresses the following:

- A bug affecting UNIX systems, where invalid permissions were assigned to executable files downloaded from a Web server.

Version 1.0.0 *"Snake Gourd"*

This is the first stable release of the application, marking the moment the application was turned over to the TU Delft Algorithmics group.

Breaking changes

- Many structural changes to the codebase, and a new configuration structure.

Version 0.8.0 *"Bitter Gourd"* pre-release

Added functionality for after-scripts, resource limits, and incremental scheduling. Set up the codebase for maintainability.

Breaking changes

- A new configuration with many new keys and changes to existing keys.

Version 0.1.5 *"Bottle Gourd"* pre-release

An initial implementation of the `gourd-test` application with basic functionality for defining experiments using a TOML file and running them locally and on Slurm.

Breaking changes

- This is a new application without compatibility with the `gourd-test` prototype's configuration files.

Version 0.1.0 (prototype) *"gourd-test"*

A prototype of an evaluator for the Pumpkin CP-SAT solver, exploring the possibility of declarative configurations.

2 Building and distributing

All of gourds builds are orchestrated exclusively by cargo and all of the rules for non standard builds are contained in the build.rs file.

2.1 Features

gourdcontains two optional compilation features described below, features are optional compilation flags that can be selected by --features feat1, feat2, feat3, ...

- fetching - This feature when turned on will compile the https backend for fetching resources in experiments, this was made an optional feature because it requires linking against OpenSSL and increases compilation time considerably.

  This is on by default, to turn it off specify: --no-default-features
- builtin-examples - Compiles the builtin examples for use in gourd init. It may be disabled for cross compilation or if the examples should not be included. See Section 4.6 for more information.

  This is on by default, to turn it off specify: --no-default-features
- documentation - This, when turned on, will compile all of the documentation for gourd and the resulting files will be placed in target/release/manpages.

2.2 Building

To build gourd in release mode issue:

```
$ cargo build --release
```

The following actions will be taken:

- The shell completions[1] will be compiled and placed in target/release/completions
- gourd_lib will be compiled.
- gourd will be compiled and placed in target/release/gourd.
- gourd_wrapper will be compiled and placed in target/release/gourd_wrapper.

You can now run gourd form the target/ folder.

[1] For shells: fish, zsh, bash, and PowerShell. See build.rs

2.3 Documentation

To build `gourd` with documentation issue:

```
$ cargo build --release --features documentation -vv
```

This requires XeLaTeX, `latex2man`, as well as `mandoc(1)` installed.

This will build all of the manpages to: `PDF`, `HTML`, and `groff`. This will also build this(!) document.

All of the resulting files are placed in: `target/release/manpages`.

To build the library documentation as well, run:

```
cargo doc --no-deps --color=always --all-features --release
```

2.4 Distribution

It was decided that at the current time `gourd` will not be compiled into operating system specific packages. This is left for future versions of the application.

For users to be able to easily install and use a fully setup gourd a specially prepared script was made.

Rust builds by default leak information about the system they were built on. As such we recommend, when building for distribution to other users, to use the command:

```
$ RUSTFLAGS="--remap-path-prefix $HOME=/REDACTED/" cargo build --release
--features documentation -vv
```

This will redact all paths containing the username, more paths to redact can be added space-separated.

This build will also produce:
`target/release/generate-installer.sh`.

By moving into the `target/release/` folder and invoking the bourne shell script like so:

```
$ cd target/release/
$ ./generate-installer.sh
```

Two platform specific files will be generated `install.sh`, and `uninstall.sh`.

These files are fully bundled versions of gourd and can be distributed directly to users. There are some enviormental variables that need to be set when generating the script,

please continue to 2.4.1 and 2.4.2.

2.4.1 An example

Let us look at the generation process for the platform `x86_64-unknown-linux-gnu`[2].

```
$ INSTALL_PATH="/usr/local/bin" \
  MANINSTALL_PATH="/usr/local/share/man" \
  FISHINSTALL_PATH="/usr/share/fish/completions" \
  ./generate-installer.sh
```

During generation of the installation script one has to specify the paths for placing the binaries, manpages, and shell completions. These will differ between systems or distibutions that is why this was introduced.

After running this we will get two scripts: `install-x86_64-unknown-linux-musl.sh`, `uninstall-x86_64-unknown-linux-musl.sh`.

Now the scrips can be distributed and the results of running them are:

```
$ sudo ./install-x86_64-unknown-linux-gnu.sh
sudo (user@pc) password:
gourd --> /usr/local/bin/gourd
gourd_wrapper --> /usr/local/bin/gourd_wrapper
manpages/gourd.1.man --> /usr/local/share/man/man1/gourd.1
manpages/gourd.toml.5.man --> /usr/local/share/man/man5/gourd.toml.5
manpages/gourd-tutorial.7.man --> /usr/local/share/man/man7/gourd-tutorial.7
Installing completions... this can fail and thats fine!
completions/gourd.fish --> /usr/share/fish/completions/gourd.fish
```

```
$ sudo ./uninstall-x86_64-unknown-linux-gnu.sh
removed '/usr/local/bin/gourd'
removed '/usr/local/bin/gourd_wrapper'
removed '/usr/local/share/man/man1/gourd.1'
removed '/usr/local/share/man/man5/gourd.toml.5'
removed '/usr/local/share/man/man7/gourd-tutorial.7'
Uninstalling completions... this can fail and thats fine!
removed '/usr/share/fish/completions/gourd.fish'
```

[2] aka Linux systems with the GNU C library on amd64 platforms

2.4.2 Notes on DelftBlue

Of course these folders for placing binary files and others will differ from system to system that is why the option of specifying them is provided, a user can also override these paths by passing the same variables (`INSTALL_PATH, MANINSTALL_PATH, ...`) to the install script.

So for example on DelftBlue, where users are not allow to install applications to the global `bin` one can do:

```
$ INSTALL_PATH="~/.local/bin" \
  ./install-x86_64-unknown-linux-gnu.sh
```

And the binaries will be placed accordingly.

3 Technologies

3.1 Rust

Rust is the main programming language used for `gourd` and its components. The project uses Rust[3] edition 2021 for buidling.

4 Structural Overview

4.1 The modules

The `gourd` project is divided into three core crates: `gourd`, `gourd_lib`, and `gourd_wrapper`.

Their individual responsibilities are laid out in this section.

4.1.1 `gourd` – Command-Line Application

The `gourd` command-line application is the core part of the project. The source code is located in `./src/gourd/`.

This compiles to the `gourd` executable.

The `gourd` binary uses the `gourd_lib` shared library to interface with

The responsibilities of the `gourd` binary are to interact with the user, schedule and run experiments (at a high level), and collect aggregate status and metrics from the experiment's runs.

4.1.2 `gourd_wrapper` – Wrapper Program

The `gourd_wrapper` is a binary that should not be invoked manually by the user.

The source code is located in `./src/gourd-wrapper/`.

It is responsible for the low-level implementation of executing a run; it takes care of combining runs of an individual binary program and input, encapsulating the program in platform-native frameworks for collecting metrics. This means that `gourd_wrapper` is the actual executable scheduled once for each run.

[3] See the MSRV for the version of Rust required

4.1.3 `gourd_lib` – Wrapper Interface

The `gourd_lib` crate contains all data shared between the application and the wrapper. The source code is located in `./src/gourd-lib/`.

This includes the 'gourd.toml' configuration file (the formal definition of an experiment) and an 'experiment.lock' runtime data file.

The general pattern is that `gourd` writes a '<experiment-number>.lock' TOML file when an experiment is started, and `gourd_wrapper` subsequently reads only the 'lock' file and on each execution to determine the path of the executable, the resource limits,et cetera

4.2 Interactions

Gourd currently contains 10 subcommands. They can be divided into three types.

The first are commands that *create a* experiment. These are:

- `gourd run local`
- `gourd run slurm`

The second are commands that *operate on* an experiment. These are:

- `gourd status`
- `gourd rerun`
- `gourd continue`
- `gourd cancel`
- `gourd analyse`
- `gourd set-limits`

The third are miscellaneous commands that need neither an experiment nor a config file. These are:

- `gourd init`
- `gourd version`

4.3 An overview of an experiment's lifetime

The first thing that a user will do is run either `gourd run local` or `gourd run slurm`.

`gourd` will look for the config file and **deserialize** it. This is quite a big step and it is explained in detail in Section 4.5.

Then `gourd` will prepare all of the runs of the experiment and create the **Experiment** struct.

One can think of an **Experiment** as a *actualization* of a **Config**.

This will create a experiment in the form of a `experiment-numer.lock` file, this is a `toml` file which is the serialized version of the struct.

The first group of commands ends at this point. From now the second group can be used.

All of these conform to a similar structure:

1. Deserialize the config to find the experiment folder.
2. Read the experiment.
3. Perform an operation (be it an iteraction with Slurm or the file system).

4.4 Interacting with Slurm

This is one of the most a problematic areas of development. Currently all interaction with Slurm is done via the commands: `sbatch`, `scancel`, and `sacct`.

These commands do have a very stable interface and as such every time `gourd` uses them, it checks whether the version is compatible.

Moreover even though these commands have both a `--yaml` and `--json` option, both of these have been found to be unrealiable (they block for very long and sometimes may fail without reason).

We have therefore opted to use the `-p` or "parsable" output of these commands for our interactions as this proved to be the most reliable.

Slurm also exposes a REST API for interaction as well as a dynamically loaded library, these were not chosen due to previous attempts to integrate with them failing, but we have the option of migration/adding more backends open.

The file which contains the **SlurmInteractor** trait is: `src/gourd/slurm/mod.rs`.

To add a new Slurm backend the only needed change is to implement all of these trait functions

Each of them are documented in the file itself and an example of a implementation is the **SlurmCli** struct.

```rust
pub trait SlurmInteractor {
    fn get_version(&self) -> Result<[u64; 2]>;

    fn get_partitions(&self) -> Result<Vec<Vec<String>>>;

    fn schedule_chunk(
        &self,
        slurm_config: &SlurmConfig,
        chunk: &mut Chunk,
        chunk_id: usize,
        experiment: &mut Experiment,
        exp_path: &Path,
    ) -> Result<()>;

    fn is_version_supported(&self, v: [u64; 2]) -> bool;

    fn get_supported_versions(&self) -> String;

    fn get_accounting_data(&self, job_id: Vec<String>)
        -> Result<Vec<SacctOutput>>;

    fn get_scheduled_jobs(&self) -> Result<Vec<String>>;

    fn cancel_jobs(&self, batch_ids: Vec<String>) -> Result<()>;
}
```

Figure 1: The SlurmInteractor trait

4.5 The deserialization

One important part of the application is the deserialization of the config file.

This process is spread out over all files in `src/gourd_lib/config/`.

The config deserialization allows for things like expanding globs and fetching, and other advanced features.

There are two flavours of deserialization for the **Config** struct, 'UserFacing', and 'NonUserFacing'.

These are controlled by a `thread_local!` variable. While this may seem to be very bad idea at first, there is really no way around it, for extensive rationale for this please see: `src/gourd_lib/config/maps.rs`

The 'NonUserFacing' flavour resembles the default strict serde deserialization. On the other hand the 'UserFacing' flavour allows for extenstions such as: `glob|`, `fetch|`, etc. (All documented in `gourd.toml.5.tex`)

4.6 On `gourd init`

`gourd init` uses a very special building system for embedding its built in examples into the binary. This build system is contained in `src/resources/build_builtin_examples.rs` and it is gated with a feature called `builtin-examples` (which is enabled by default).

Essentially, whenever `gourd` is built all folders under `src/resources/gourd_init_examples` are bundled as examples into the application, the folder is bundled in whole with the exception of files ending in `.rs`.

Files ending in `.rs` will be compiled for the target architecture and included as such.

To add a new example to `gourd init` do the following:

1. Add a new folder under `src/resources/gourd_init_examples`.
2. Put all of the example files (including a valid `gourd.toml`) in that folder.
3. The table of all examples is contained in `src/gourd/init/builtin_examples.rs` in the function called `get_examples()`, add your example there with the name of the tar file being the same as the examples name.

Here is an example entry from the table:

```
"a-simple-experiment",
InitExample {
    name: "A Simple Experiment",
```

```
    description: "A comparative evaluation of two simple programs.",

    directory_tarball: include_bytes!(concat!(
        env!("OUT_DIR"),
        "/../../../tarballs/a_simple_experiment.tar.gz"
    )),
},
```

5 The pipeline

`gourd` has a strict pipeline built on a custom docker container.

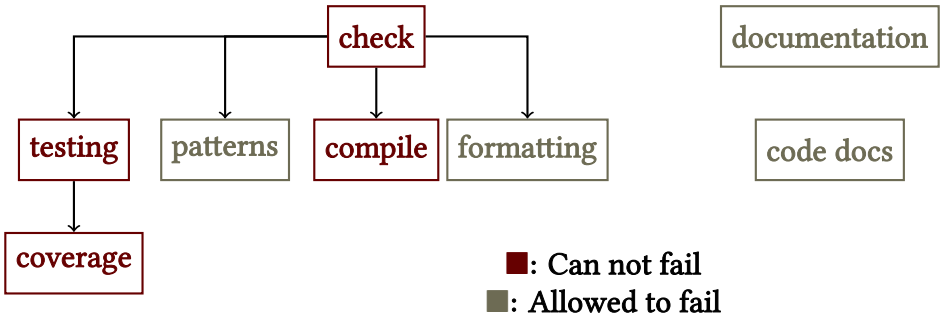

Figure 2: An overview of the pipeline

The pipeline configuration is fully contained in `.gitlab-ci.yml` in the root of the project.

Currently the docker container is hosted in a public GitHub repository, because there was a need to publish it in a container registry if there is a need for updating Rust or migration of the pipeline this container will have to be rebuilt.

There are three containers that are needed for the pipeline:

1. A container with Rust stable, MSRV version.
2. A container with Rust nightly, `rustfmt`, `clippy`, `cargo-tarpaulin`.
3. A container with all of TexLive and Rust, MSRV or nightly.

These three are defined at the top of the file and if the pipeline is ever updated or migrated these should be updated. All three containers are built on top of musl systems for minimal footprint. (In fact the GitHub container is based off of `musl-rust`)

6 Contributing

The contribution guidelines for this project, and the decisions made about code style, will be outlined in this section of the document.

6.1 Guidelines

The project is hosted on GitLab. Changes to the "main" branch must be staged through a merge request approved by at least two maintainers. The number of commits should be kept to a minimum to ensure clarity.

All commit messages must be lowercase.

No files can contain whitespace and the end of lines.

No files can contain newlines at the end.

All indentation should be spaces.

No merge commits, branches should always be rebased before a merge to `main`.

6.2 GitHub Actions

There are 3 pipelines in the project:

6.2.1 Draft

Build, test & lint draft pull requests. Quick and short pipeline with no artifacts.

This pipeline triggers on every push to a pull request with no reviewers requested, or when the `draft` label is explicitly assigned.

6.2.2 Ready

A longer pipeline than draft that includes documentation that is also published as an artifact. Use this pipeline when the pull request is ready for review.

This pipeline triggers on every push to a pull request with reviewers requested, or when the `ready` label is explicitly assigned.

6.2.3 Release

Generates install scripts and documentation for the project.

Runs automatically on pushes to main, but can also be manually triggered by assigning the `release` label to a pull request.

6.3 Releasing a new version

Each merge request that increments the software version must do so in:

- `Cargo.toml`
- The beginning of `docs/user/gourd.1.tex`
- The beginning of `docs/user/gourd.toml.5.tex`
- The beginning of `docs/user/gourd-tutorial.7.tex`
- In `src/gourd_lib/constants.rs`

6.4 Documentation

In addition to adding an entry to the version history, as detailed above, maintainers should also make sure that the documentation remains thorough.

1. The Maintainer PDF documentation (this document) contains a brief overview of the application's underlying design.
2. The three "manpages" (in `docs/user/`) contain detailed usage instructions and configuration documentation. The `gourd-tutorial` manpage is an exception: it is written in narrative form and should be accessible to new users.
3. Inline documentation (Rustdoc) contains additional information necessary for the developer of a particular feature.

6.5 Code Style

Code written in Rust should be formatted using 'cargo fmt' and validated using 'cargo clippy'. All relevant code style rules are enforced by the GitLab CI and can be tested locally by configuring the Git hooks:

The gitlab pipeline validates all of the changes against our rules, but to check these locally we **strongly** recommend issuing the command below:

```
$ git config --local core.hooksPath .hooks/
```

This will ensure that whenever you commit or push new changes they will be first checked against a hook scripts which catch common mistakes that would have failed the pipeline check remotely.

Additionally, the following apply:

- Beware when using `[cfg]` statements, as they can cause 'clippy' issues on different platforms.
- Enforce separation between crates (such as `gourd` and `gourd_wrapper`) as much as possible.
- Place tests in a separate file in a `tests/` subdirectory of a module. Ensure that the filename corresponds to the file being tested.

6.6 Testing

The project goal is to be thoroughly tested with full-coverage systematic unit tests. The GitLab environment analyzes coverage. This is displayed alongside each merge request.

6.7 Integration tests

`gourd` has integration tests, they are contained in `src/integration` and are full end-to-end command line tests.

There is an easy template for adding new integration tests which is stored in: `src/integration/example.rs` please refer to that file if you add new functionality to `gourd` that should be tested with integration tests.

7 Future work

7.1 Removing the installer

The entire installer template is contained in `src/resources/install.sh`.

It is quite easy to extend this installer with new files or paths. One option which we considered a nice addition would be to skip installing parts of the application on the request of the user (for example manpages).

Finally, as outlined in the beginning of the section, the biggest improvement would be to disable the installer and distribute `gourd` via system specific libraries.

It would be ideal if this process was included in `build.rs` via optional features.

We have identified the following packages as priorities to make `gourd` accessible to the largest user base:

- A Debian `.deb` package. This enables Ubuntu packaging as well.

- A homebrew package script. This enables MacOS.
- A ArchLinux `PKGBUILD` file. This enabled distribution for ArchLinux.
- A RHEL `.rpm` package. This enables Fedora packaging as well.
- A `flake.nix` file for NixOS.
- A Windows `.msi` style installer.

7.2 Memory hotspot analysis

An originally planned feature was to add `kcachegrind` like analysis for algorithms to identify memory hotspots.

We believe that this feature can be implemented by additions to the wrapper.

Currently the wrapper is a very simple program (on purpose) but an option may be added to it to run, for example, `kcachegrind` on the ran program and output, alongside all of the other metrics, the hotspot analysis.

7.3 The wrapper errors

In some usage cases it may happen that `gourd_wrapper` fails instead of `gourd`.

For example, if a file that `gourd` verifies to exist is removed before the wrapper has a chance to run, the wrapper will throw an error.

This is currently problematic in local running, as the error will be printed on top of the status and the application will exit. This is due to how our async scheduling works, we make minimal use of async functions and only those that have to be async are marked as such.

To fix this error printing there is a need for synchronization between the status update function and the local runner, but this requires migrating the status function to tokio runtime async functions.

Moreover this can also present itself to be a problem when running on Slurm. The error will then be printed in the `slurm-[jobid].out` which is very not user friendly.