Thoughts on a logging facility for CORSIKA

Design hints for an efficient and thread-safe logger

A. Augusto Alves Jr

Presented at CORSIKA development meeting - KIT, Karlshuhe
January 14, 2020

- Logging and debug
- Advantages and potential caveats
- Some existing libraries
- Multi-thread safeness and contention considerations
- Possible design

Logging and debug

In general logging is different of debugging. Logging is important for:

- Monitoring the application is actually doing what it is supposed to do.
- Good logging is necessary when debugging is not possible or convenient.
- Debugging is not always possible. Examples: the application is running on specific hardware or using specific and not-reproducible configuration.
- Issue and bug report from users.

In summary, proper debugging requires using dedicated tools and intervention at source code level. It is for developers. Logging is important for developers too, but it is essential for users.

High level requirements

- The logger should be effective and able to provide clear data.
- Provide enough contextual information to pinpointing an issue.
- Efficiency. The impact of log facility on the application running in production environment should be negligible.
- Guarantee of proper functioning even when application is crashing. Would make no sense to write logs if the logger stops working at every crash.
- Provide interface to store the logs on local and remote file system.

Multithread safeness and contention considerations

CORSIKA is supposed to be multithread and to run as fast as possible.

- Orchestrate concurrent logging from different threads is not a easy task: race-conditions, deadlocks, segmentation faults. By naively streaming data to `std::iostream` from different threads, one can easily finish with corrupted files or messages like this:

  "...MessaMesgesage number 1: number 2:..."

- Even if properly guarded and synchronized, the streaming of data to file can generate significant contentions in the application due latency in the interactions with the file system and their propagation through the synchronization facility.
- How to handle global contextualization from infra-thread context?
- Code running on different threads should be able to operate the logger in different levels.
- Crashing or excepting code should have priority to log, otherwise we risks losing track on the issue. How to handle this?

What should be logged?

CORSIKA should provide meaningful log entries for computing and physics related information.

- Physics and computing logging verbosity should be configurable to operate at different levels.
- Exceptions should be always handled and properly logged.
- State of the application should logged as well in order to allow the reproduction of the issue of interest in under controlled debugging conditions. Examples: PRNG state, time, iteration number etc.
- Serializable objects could be logged too. Potential use case: physics process.

What should be logged?

CORSIKA should provide meaningful log entries for computing and physics related information.

- Physics and computing logging verbosity should be configurable to operate at different levels.
- Exceptions should be always handled and properly logged.
- State of the application should logged as well in order to allow the reproduction of the issue of interest in under controlled debugging conditions. Examples: PRNG state, time, iteration number etc.
- Serializable objects could be logged too. Potential use case: physics process.

Some existing libraries

I am studying the design of these libraries:

- `Boost.Log`: Designed to be thread-safe, efficient and modular. Complex configuration and comes with all Boost dependencies. Boost license.
- `Easylogging++`: Single header only, light-weight, high performance logging library for C++11 (or higher) applications. MIT license.
- `G3log`: Thread-safe and crash-safe logger. It has an strange license called "The Unilicense".

Seems to me that, none of them is fully satisfactory...

Possible CORSIKA owned design

- Manage all the messages asynchronously using a concurrent priority queue.
- All threads could post messages to this queue, using a non-blocking interface.
- The entries in the priority queue would be consumed by "sinks"
- The sinks would abstract away the details of interactions with the file system.

This would decouple the file system related contentions from the infra-thread interface and allow redundancy at sink level.