A Discussion of CORSIKA8 Output Formats

+ and a brief review of current status

R. Prechelt

September 24, 2020

Outline

Current Status

Review

To review previous discussions:

- The proposed built-in C8 output format is a filesystem directory with subdirectories for output components.
- Metadata and small output data (i.e. fit parameters, depth of X_{max}, etc.) are stored in YAML metadata files. There is a top-level YAML file at the root of the output directory with global metadata.
- Larger output files will be stored in a TBD file format alongside the YAML files.

As soon as we can make a preliminary decision on a format, I can move head with the implementation for the current processes.

Current Implementation (MR)

Working

- Generation of the filesystem hierarchy given an output list.
- My interpretation of the architecture; processes are responsible for writing their outputs into a provided handle. Fine control via `InitializeOutput`, `NextRun`, `NextEvent`, and `FinalizeOutput` methods.
- Creation of the YAML metadata files with some basic values.
- Multiple instances of the same process can be used with different unique names.

TBD

- Decision on a data format and helpers for reducing code duplication
- Automatic generation of the process list into the YAML file is mostly done but still needs tweaks.
- Physics processes are almost “state-free” with state encapsulated into the “Detector”.

Python Package

- I also have a working Python package skeleton `corsika` that provides a high-level API for accessing all the metadata from a given library (without the user finding+loading individual YAML files).
- This is currently `pip`-installable with generated docs, static typing, linting etc.
- Will update and maintain this once we have decided on an output data format.

Example Top-Level Metadata File

```yaml
---
name: "A library name"
version: "8.0.0-alpha"
commit: b054578ce
num_showers: 100
processes:
  - pythia
  - sibyll
  - tauola
outputs:
  - groundparticle
  - radio
  - cherenkov
```

Decisions and Questions

Fortran Support?

- Previous discussions assumed C++ (w/ ROOT) and Python support were *required* and other languages were a *bonus* (my interpretation).
- Comments on the Gitlab issue suggest that *reading* C8 output into *Fortran* may be a required feature.
- This significantly changes the available formats as most (but not all) of the formats we have discussed don't have Fortran readers and it would be challenging (if not impossible) to write them.

Is a Fortran reader a requirement for C8?

Hierarchy Ordering (1/3)?

There are two options for the ordering of the filesystem hierarchy:

- `library -> shower -> process`
- `library -> process -> shower`

```
<library name>/
    <library name>.yaml
    shower0001/
        shower0001.yaml
        radio/
            ...
        cherenkov/
            ...
        ground/
            ...
    shower0002/
            ...
```

Hierarchy Ordering (2/3)?

```
<library name>/
    <library name>.yaml
    radio/ # showers could be in their own subdirectories
        radio.yaml
        shower0001/
            waveforms.{dat,root,parquet,npy}
        shower0002/
    cherenkov/ # or showers combined into fewer files
         cherenkov.yaml
         cherenkov_photons.{dat,root,parquet,npy} # all showers
```

Hierarchy Ordering (3/3)?

Option A (lib.->shower->comp.)

- Easy to extract individual showers from the library (each shower is just a directory)
- Requires more individual files (metadata and data) but does allow an individual shower to be used outside of the scope of the library (standalone).

Option B (lib.->comp.->shower)

- Easily extract individual *components* from the library - i.e. all ground particle files are in one directory, all radio waveforms are in one directory.
- If showers are stored in subdirectories, extracting a subcollection of showers requires traversing the hierarchy multiple times (not too hard).
- If showers are stored in large multi-shower files, will require a custom script/program to extract showers from the library.

Formats

Summary

The formats we are discussing are *generally* split into two types: *array*-oriented and *record*-oriented.

The *array*-oriented formats are often the most performant at reading large chunks of contiguous tabular data (most of our large files will probably fit into an array or tabular format). *I've been focused on the array-oriented formats.*

Array-Oriented

- NPY (NumPy) (`.npy`, `.npz`)
- ROOT flat n-tuples using `exlib/inlib`
- Apache Parquet

Record-Oriented

- XCDF
- Protobuf
- Cap'n'Proto

Implementations & Discussions

- Ralf put together several variations on the `ObservationPlane` that write to `.root`, `.hd5`, and `.parquet` - the sources are here. and are worth looking at.
- Ralf, do you have any comments on this?
- Much of the discussion surrounding these formats has been on this issue.

ROOT

- `inexlib_rio` which is a standalone implementation of the ROOT file reading/writing for NTuples.
- It is easy to use and is roughly as fast as the standard ROOT I/O (based upon my initial benchmarking).
- It has not been updated in 5 years so we may be taking on some maintenance responsibility if ROOT ever makes a forward-incompatible change.
- This opens the possibility of writing ROOT files without needing a full ROOT installation.
- This does limit analysis to C++ and Python (mostly).

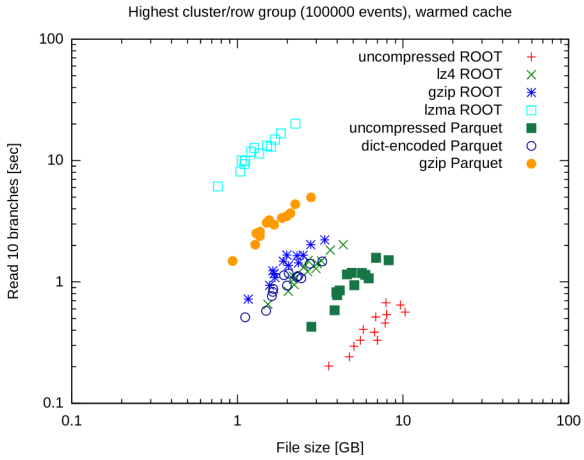
Parquet (Arrow)

Parquet is a HPC-oriented tabular data format part of Apache Arrow.

- Large supported ecosystem (Pandas, Hadoop, compression algorithms, and more) and available in many languages (Python, C, C++, MATLAB, Julia, R, and more).
- Arrow in-memory representation is very fast, can be mmap'd from Parquet files, and interfaces with CUDA. Supports streaming writes!
- A 2 year old comparison between Parquet and ROOT is available here. Parquet has improved since then.
- Minimal build takes ~10-15s to compile on a single laptop core so won't appreciably add to our build.

ROOT vs. Parquet

- This is a 2-yr old comparison of ROOT vs. Parquet by the author of `uproot` (Jim Pivarski)
- I've been replicating this with the latest Parquet using fake particle data and Parquet is significantly faster. For reasonable C8 file sizes, I'd consider the read performance of ROOT and Parquet to be roughly comparable.



Numpy

- Another proposed alternative is to direct write Numpy NPY array files. These are an extremely simple format for storing N-dimensional heterogenous arrays (you can write a basic reader from the spec < 30 minutes).
- Extremely fast to read (2x as ROOT/Parquet if reading the whole file).
- Requires reading the entire array - none of the tabular or chunked advantages of ROOT/Parquet. Streaming writes can be tricky.
- Multi-array (multi-shower) files (.npz) are standard `zip` files so they can be separated with standard 'Nix tools.
- Implementations available in many languages (C++, Julia, MATLAB, R, et al.)

Discussion

What do we need to see in order to come to a conclusion about a format?