Counter-based pseudorandom number generators for CORSIKA 8

A multi-thread friendly approach

A. Augusto Alves Jr[*], Anton Poctarev[*] and Ralf Ulrich[*]

Presented at 25th International Conference on Computing in High-Energy and Nuclear Physics

May 18, 2021

[*]Institute for Astroparticle Physics of Karlsruhe Institute of Technology

CORSIKA 8

CORSIKA 8 is a new framework for the modelling of extensive air showers in astroparticle physics. Here is a short summary:

- C++17 compliant.
- Efficiency and scalability for deployment in HPC environments: multiprocess, multithread and multiplatform.
- Precision and reproducibility.
- Modular design.
- Flexible geometry and environment specification.

As any Monte Carlo technique based program, CORSIKA 8 needs an efficient and reliable facility for generation of pseudorandom number streams.

Conventional pseudorandom number generators

- Most of the conventional pseudorandom number generators (PRNGs) scale poorly on massively parallel platforms (modern CPUs and GPUs).
- Inherently sequential algorithms:

$$s_{i+1} = f(s_i),$$

  where s_i is the i-th PRNG state.
- The statistical properties of the generated numbers are dependent on the function f and of the size of s_i in bits. Usually f needs to be complicated and s_i large.
- PRNGs can be deployed in parallel workloads following two approaches: *multistream* and *substream*.
- Both approaches are problematic due pressure on memory, impossibility to jump into far away states skipping the intermediate ones, correlations between streams.

Counter-based pseudorandom number generators

The so called "counter-based pseudorandom number generator" (CBPRNG) produces sequences of pseudorandom numbers following the equation

$$x_n = g(n),$$

where g is a bijection and n a counter. Basic features:

- Very efficient. Actually, in many implementations it allows trading-off performance for efficiency in dynamic and transparent way.
- Have null or low pressure on memory since they can be implemented in a fully stateless fashion.
- Very suitable for parallelism, since they allow to jump directly to an arbitrary sequence member in constant time.

Categories of CBPRNGs

- Cipher-based generators:
  - **ARS (Advanced Randomization System)** is based on the AES cryptographic block cipher.
  - **Threefry** is based on Threefish a cryptographic block cipher.
- Non-cryptographic bijective transformation generators:
  - **Philox**. Deploys a non-cryptographic bijection based on multiplication instructions.
  - **Squares**. This algorithm is derived using ideas from “Middle Squares” algorithm, originally discussed by Von Neuman, coupled with Weyl sequences.

The current implementation uses ARS, Threefry and Philox from Random123 library. Squares is natively implemented.

Iterator-based design for parallelism

.

- Iterators are a generalization of pointers and constitutes the basic interface connecting all STL containers with algorithms.
- Iterator-based designs very convenient for parallelism.
- A very popular choice for implementing designs based on lazy evaluation.

These features considered all together make an iterator-pair idiom the natural design choice for handling the counters and the CBPRNG output, in combination with lazy-evaluation to avoid pressure on memory and unnecessary calculations. The current implementation uses iterators from TBB library.

Iterator-based API

The streams are represented by `Stream<Distribution, Engine>` class:

- It is thread-safe and handles *multistream* and *substream* parallelism.
- Produces pseudorandom numbers distributed according with `Distribution` template parameter.
- Handles 2^{32} streams with length 2^{64}, corresponding to 2048 PB of data, in `uint64_t` output mode.

```cpp
template<typename Distribution, typename Engine>
class Stream
{
 public:

   //constructor
   Stream( Distribution const& dist, uint64_t seed, uint32_t stream );

   //stl-like iterators
   iterator begin() const;
   iterator end() const;

   //access operators
   double operator[](size_t n) const;
   double operator()(void);
};
```

Fully compatible with C++ standard library distributions.

Performance measurements

CBPRNG	Time - stream (ns)	Time - stl distribution (ns)
Philox	8.853	8.062
ARS	9.031	8.684
Threefry	11.458	12.145
Squares3	8.691	7.956
Squares4	10.891	10.024

The second column lists the time spent calling the method `Stream<std::uniform_real_distribution<double>, Engine>::operaror[](size_t i)` . The third column lists the time for calling the distribution directly. Measurements taken in a Intel Core i7-4790 CPU, running at 3.60GHz with 8 threads (four cores) machine.

Integration into CORSIKA 8

- Currently CORSIKA 8 uses `std::mt19937_64`, the Mersenne Twister (MT) implementation of the C++17 Standard Library, as its primary pseudorandom number generator.
- MT is known to fail statistical tests. It also stores a huge state, of almost 2.5 kB, and operates strictly sequentially.

The integration of the iterator-based Stream API into CORSIKA 8 is straightforward:

- Refactoring of the internal algorithms is not required.
- The distribution and management of multiple instances of CORSIKA 8, configured with different seeds and running in parallel on clusters and other distributed systems is not impacted.
- Enables further development of more fine-grained parallelism into the existing algorithms in a transparent way.

Summary

The deployment of CBPRNGs for the production of high-quality pseudorandom numbers in CORSIKA 8, using an iterator-based and multi-thread friendly API has been briefly described.

- The API is STL compliant, lightweight and does not introduce any significant overhead for calling the underlying generators and distributions.
- The API allows the efficient management of parallelism using the substream approach, providing up to 2^{32} sub-sequences of length 2^{64}, configured with the same seed.
- The streams can be accessed sequentially or in parallel using the API components.
- A public repository under a GPLv3:

  `https://gitlab.ikp.kit.edu/AAAlvesJr/random_iterator`

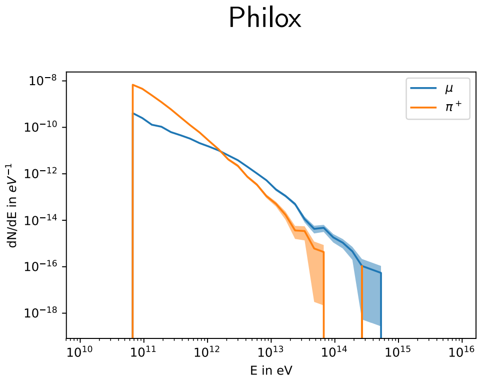
Thanks

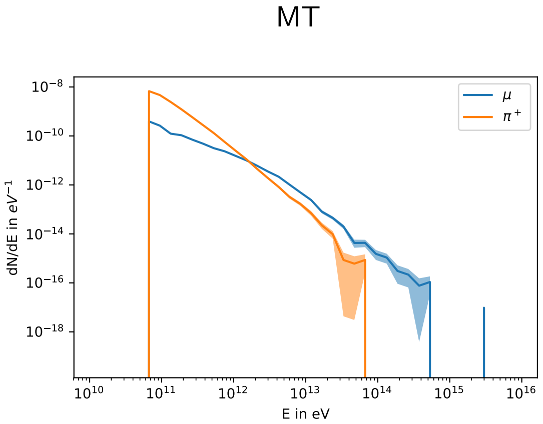
Backup

Statistical tests

- The CBPRNGs pass all the pre-defined statistical test batteries in TestU01, which includes SmallCrush (10 tests, 16 p-values), Crush (96 tests, 187 p-values) and BigCrush (106 tests, 254 p-values).
- BigCrush takes a few hours to run on a modern CPU and it consumes approximately 2^{38} random numbers.
- Additionally, all CBPRNGs have been tested using PractRand, using up to 32 TB of random data. No issues have been found.

Examples of showers





CORSIKA 8 simulation of energy spectra at sea level for a single proton primary particle at 40 deg with 10^{17} eV and cutoff at 60 GeV.