EPJ Web of Conferences **251**, 03039 (2021) https://doi.org/10.1051/epjconf/202125103039
CHEP 2021

Counter-based pseudorandom number generators for CORSIKA 8

A multi-thread friendly approach

A. Augusto Alves Jr[1,*], *Anton* Poctarev[1], and *Ralf* Ulrich[1,**]

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

Abstract. This document is devoted to the description of advances in the generation of high-quality random numbers for CORSIKA 8, which is being developed in modern C++17 and is designed to run on modern multi-thread processors and accelerators. CORSIKA 8 is a Monte Carlo simulation framework to model ultra-high energy secondary particle cascades in astroparticle physics. The aspects associated with the generation of high-quality random numbers on massively parallel platforms, like multi-core CPUs and GPUs, are reviewed and the deployment of counter-based engines using an innovative and multi-thread friendly API are described. The API is based on iterators providing a very well known access mechanism in C++, and also supports lazy evaluation. Moreover, an upgraded version of the Squares algorithm with highly efficient internal 128 as well as 256 bit counters is presented in this context. Performance measurements are provided, as well as comparisons with conventional designs are given. Finally, the integration into CORSIKA 8 is commented.

1 Introduction

CORSIKA 8 is a new framework [1] for the modelling of extensive air showers in astroparticle physics. Since individual ultra-high energy cosmic ray nuclei are observed beyond energies as extreme as $E_{\text{lab}} = 10^{20}$eV, the amount of computing resources needed to accurately describe the corresponding secondary particle cascades are within one of the most demanding tasks in computing nowadays. High-performance algorithms, utilization of accelerators, as well as parallelism need to be developed and deployed to tackle this problem. Our understanding of the high-energy universe depends on the precise understanding of such particle cascades. In CORSIKA 8, the Monte Carlo technique provides the foundation for this, thus, the efficient and well-understood generation of random number streams are of paramount importance.

Many popular conventional pseudorandom number generators (PRNGs) scale poorly on massively parallel platforms, like modern CPUs and GPUs. In fact, such generators are built using inherently sequential algorithms that operate applying a transformation function f on each state s_i to obtain the next one, s_{i+1},

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

*e-mail: augusto.alvesjunior@kit.edu
**e-mail: ralf.ulrich@kit.edu

The state is then feed into another function in order to obtain the pseudorandom number.

For the conventional PRNGs, the statistical properties of the generated numbers are strongly dependent on the function f and of the size of the state, typically measured in number of bits. In general, to obtain numbers with good statistical properties, large states and complicated functions are necessary. Such generators are deployed in parallel calculations following two basic approaches. The first one, called here *multistream*, consists in creating as many instances as necessary of the PRNG, initializing each one of them with different parameters, usually seeds, in order to obtain different sequences. In the second approach, referred to as *substream*, one would initialize the instances with the same parameters and setup each of them to different and far-away (hopefully) detached states, skipping the intermediate ones.

Both approaches are problematic. Indeed, running so many instances, each holding different states, leads to an increasing pressure on memory. Additionally, regarding the first approach, most algorithms give guarantees on the statistical quality of the output only for numbers produced in the same sequence. Usually, to access the statistical independence of numbers from different sequences is a much harder task – if possible at all. The alternative approach, substream parallelism, often produces unacceptably inefficient calculations, since usually no suitable algorithms are known to skip states in constant or amortized time.

This document describes how the issues discussed in the previous paragraphs have been overcome in CORSIKA 8, via the deployment of "counter-based pseudorandom number generators" (CBPRNGs) and their management through an iterator-based, STL compliant, and parallelism enabled interface. The basic concepts and the used CBPRNGs are presented in section 2. The iterator-based API and its usage are described in section 3. Statistical tests and some performance measurements are discussed in section 4. The section 5 describes the integration into CORSIKA 8.

2 Counter-based pseudorandom number generators

An alternative design of PRNGs, efficient and suitable for parallelism, addressing the issues previously discussed indeed exists. It is called "counter-based pseudorandom number generator" (CBPRNG) [2, 3] because it produces sequences of pseudorandom numbers following the equation

$$x_n = g(n),$$

where g is a bijection and n a counter. Such generators produces high-quality output and are suitable for parallelism because they can be implemented in a stateless fashion and allow to jump directly to an arbitrary sequence member in constant time. The properties of the CBPRNGs added to CORSIKA 8 are discussed in sections 2.1 to 2.3.

2.1 Cipher-based generators

Cryptographic block ciphers can be described as "keyed" functions,

$$x_n = g_k(n),$$

where k is an element of the key space and g_k is one member of a family of bijections. Consequently, a counter-based PRNG constructed from a keyed bijection can be easily parallelized using the multistream approach on the key space, or the substream approach over the counter space. In practice, some of the used ciphers operate applying rounds of transformations in order to achieve cryptographic security. The recipe to get efficient CBPRNGs out of ciphers

is to trade cryptographic strength, which is not required, for performance by applying less rounds and simplifying the keys just enough to achieve statistical quality.

Two generators with this design are considered, nominally ARS and Threefry, both introduced by the Random123 [2] library and briefly described below:

- ARS (Advanced Randomization System) is based on the AES cryptographic block cipher and as such, it relies on AES New Instructions (AES-NI), present in most of the modern CPUs. The ARS implementation used in this study applies 7 rounds of transformation.
- Threefry is based on Threefish a cryptographic block cipher and relies only on common bitwise operators and integer addition. Therefore, Threefry does perform well across a wide range of instruction-set architectures. The implementation used in this study applies 20 rounds of transformation.

2.2 Non-cryptographic bijection transformation generators

The third generator used in this, also from the Random123 library, is called Philox. This algorithm uses non-cryptographic bijection that is based on multiplication instructions computing the high and low halves of operands to produce wider words. For example multiplying two 32-bit numbers producing a 64-bit number, or even multiplying two 64-bit numbers and producing a 128-bit number. Philox is probably the most popular CBPRNG and implementations are available on CPUs, and as well on GPUs via nVidia's cuRAND library. The Philox implementation used in this study applies 10 rounds of transformations[2].

2.3 Middle-square transformation generators

The Squares CBPRNG is derived using ideas from "Middle Squares" algorithm, originally discussed by Von Neuman[4], coupled with Weyl sequences[5, 6]. Indeed, only half of the actual square is computed, which corresponds to the upper bits of the result. These middle bits are easily obtained by either rotating or shifting the result. The middle square provides the randomization, while uniformity and period length are obtained by adding in a Weyl sequence. For the Squares algorithm, the Weyl sequence is just a counter multiplied by a key. Three or four rounds of squaring are enough to achieve high statistical quality, with outstanding properties.

The original implementation[3] supports 64 bit counters and produces 32 bit output, which are not completely suitable for CORSIKA 8 need, and other similar applications in physics. Hence, in this contribution, the original algorithm is updated to support 128 bit counters with 64 bit output and 256 bit counters with 64 bit output.

3 Iterator-based API for parallelism

Iterators are a generalization of pointers and constitutes the basic interface connecting all STL containers with algorithms. Therefore dereferencing an iterator, `(*itr)`, returns the pointed element itself. Iterators can be incremented and decremented like pointers to access successive elements in a data structure. Finally, just like pointers, iterators are also lightweight objects that can be copied with insignificant computing costs and incremented arbitrarily in constant time, making iterator-based designs very convenient for parallelism. The iterator idiom is also 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.

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

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

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

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

Listing 1: Minimal public interface of the `Stream<Distribution, Engine>` object. Getters and setters are omitted. In the current implementation, the `Stream<Distribution, Engine>` object can handle up to 2^{32} sequences of pseudorandom numbers each with a length of 2^{64} and produced with the same seed.

The proposed design uses the "fancy iterators" implemented in Intel's TBB. Counters are handled using counting iterators, to represent the range $[0, 2^{64}]$. The counting iterators are then wrapped by transform iterators, which convert each integer into the actual pseudorandom number calculated by the CBPRNG, and furthermore to calculate the aimed distribution, which is configurable via a template parameter.

The pseudorandom number sequences are represented by `Stream<Distribution, Engine>` objects, which expose its functionality through a concise and functional API, summarized in listing 1.

4 Statistical and performance measurements

4.1 Statistical tests

The CBPRNGs used in this study pass all the pre-defined statistical test batteries in TestU01[7], which include 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 samples in total.

Additionally, all used CBPRNGs have been tested using PractRand[8], using up to 32 TB of random sample data. No issues have been found.

4.2 Performance

The performance of the iterator-based API design is measured using the benchmark capabilities provided by the Catch2 library[9]. The timing of the API calls `Stream<Distribution, Engine>::operator[](size_t i)`

CBPRNG	Time - stream (ns)	Time - stl distribution (ns)
Philox	8.853	8.062
ARS	9.031	8.684
Threefry	11.958	11.078
Squares3	8.691	7.956
Squares4	10.891	10.024

Table 1. For each generator, the second column lists the time spent calling the method `Stream<std::uniform_real_distribution<double>, Engine>::operator[](size_t i)`. The third column lists the time for calling the distribution directly. As can be easily verified by comparing the numbers, the `Stream<Distribution,Engine>` API does not introduce any significant overhead. Measurements taken in a Intel Core i7-4790 CPU, running at 3.60GHz with 8 threads (four cores) machine.

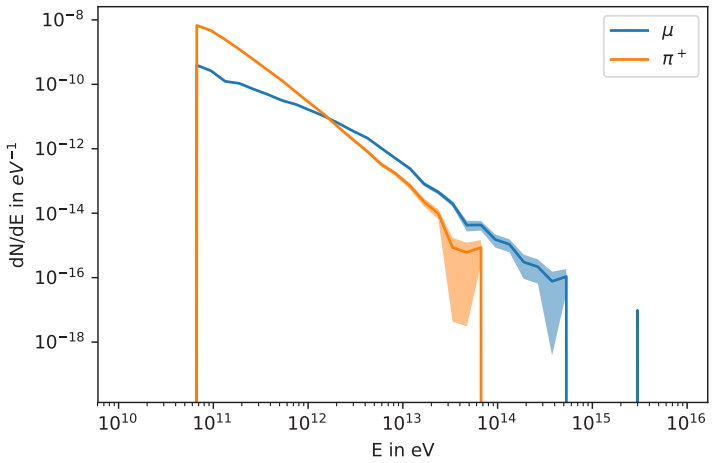

Figure 1. CORSIKA 8 simulation of energy spectra at sea level for a single proton primary particle at 40 deg with 10^{17}eV, using the Mersenne Twister random number generator, which is so far the default in CORSIKA 8. There is a cutoff at 60 GeV.

for each generator are summarized in the table 1. For the sake of comparisons, the table also includes the timing of a single call to a `std::uniform_real_distribution<double>::operator()(Engine& )` object. Comparing the two measurements indicates that calling the distribution through the `Stream<Distribution,Engine>` API does not introduce any significant overhead.

5 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. Despite its very long sequences and good single-call performance, MT is known to fail statistical tests. It also stores a huge state, of almost 2.5 kB, and operates strictly sequentially. COR-

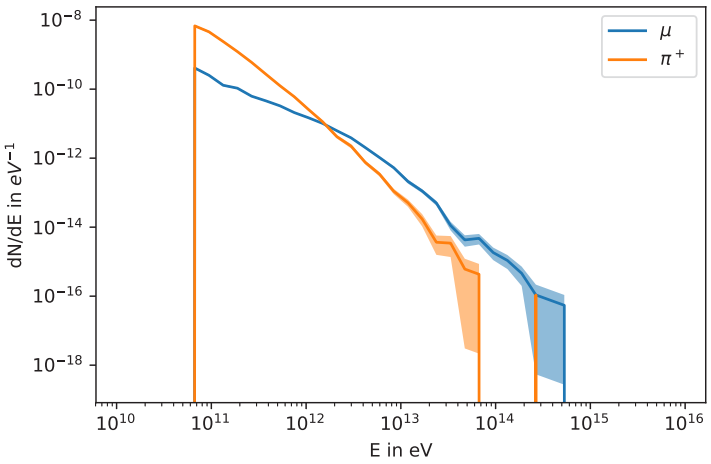

Figure 2. CORSIKA 8 simulation of energy spectra at sea level for a single proton primary particle at 40 deg with 10^{17}eV, using the Philox random number generator. There is a cutoff at 60 GeV.

SIKA 8 deploys MT using multistream parallelism, which brings concerns about inter-stream correlations.

The integration of the iterator-based Stream API into CORSIKA 8 is straightforward and does not require any downstream refactory of the internal algorithms consuming pseudorandom numbers. It also enables further development of more fine-grained parallelism into the existing algorithms in a transparent way, that does not interfere with the surrounding algorithms.

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 either.

As a demonstration of the application of those random number generators inside CORSIKA 8 we show fig. 1 and fig. 2. The former illustrates the default simulation so far using Mersenne Twister random numbers, while the latter shows the same distribution simulated with the Philox random number generator. Statistically the two distributions are the equivalent with a $\chi^2/N_{\text{d.f.}}$ of 0.65.

6 Conclusion

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 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 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 described in listing 1. In addition to the generators from Random123 library, an upgraded version of the Squares algorithm with highly efficient internal 128 as well as 256 bit counters is introduced.

The authors feel that the developments described in this document are useful beyond the application in CORSIKA 8 for other projects in physics. Therefore the final code will be released as a standalone open-source project, under a liberal license, in a public repository.

Acknowledgments

The authors acknowledge support by the High Performance and Cloud Computing Group at the Zentrum für Datenverarbeitung of the University of Tübingen, the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no. INST 37/935-1 FUGG. This work was done as part and together with the CORSIKA 8 Collaboration.

References

[1] R. Engel, D. Heck, T. Huege, T. Pierog, M. Reininghaus, F. Riehn, R. Ulrich, M. Unger, D. Veberič, Comput. Softw. Big Sci. **3**, 2 (2019), arXiv:1808.08226 [astro-ph]

[2] J.K. Salmon, M.A. Moraes, R.O. Dror, D.E. Shaw, *Parallel Random Numbers: As Easy as 1, 2, 3*, in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Association for Computing Machinery, New York, NY, USA, 2011), SC '11, ISBN 9781450307710, https://doi.org/10.1145/2063384.2063405

[3] B. Widynski, arXiv e-prints arXiv:2004.06278 (2020), 2004.06278

[4] J. von Neumann, in *Monte Carlo Method*, edited by A.S. Householder, G.E. Forsythe, H.H. Germond (US Government Printing Office, Washington, DC, 1951), Vol. 12 of *National Bureau of Standards Applied Mathematics Series*, chap. 13, pp. 36–38

[5] H. Weyl, Math. Ann. **77**, 313 (1916)

[6] B. Widynski, arXiv:1704.00358 (2017)

[7] P. L'Ecuyer, R. Simard, ACM Trans. Math. Softw. **33** (2007)

[8] C. Doty-Humphrey, *Practrand (practically random), a c++ library of pseudo-random number generators(prngs) and statistical tests for prngs.* (2010), https://sourceforge.net/projects/pracrand/

[9] *Catch2, an unit testing micro-benchmarking framework for c++*, https://github.com/catchorg/Catch2/releases/tag/v2.13.4