Random number generation in massivelly parallel platforms for CORSIKA 8

An iterator and counter-based approach

A. Augusto Alves Jr

Presented at 3rd CORSIKA 8 Parallelism & Performance Meeting - KIT, Karlshuhe
March 25, 2021

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:

- High quality output.
- Very efficient. Actually, it allows trade-off performance for efficiency in a transparent way.
- Have null or low pressure on memory since they can be implemented in a 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 and relies on AES-NI.
  - **Threefry** is based on Threefish a cryptographic block cipher and relies only on common bitwise operators and integer addition.
- Non-cryptographic bijective transformation generators:
  - **Philox**. Deploys a non-cryptographic bijection based on multiplication instructions computing the high and low halves of operands to produce wider words.
  - **Squares**. This algorithm is derived using ideas from "Middle Squares" algorithm, originally discussed by Von Neuman, coupled with Weyl sequences. Three or four rounds of squaring are enough to achieve high statistical quality. Squares implementation here is original and supports 128 bit counters with 64 bit output.

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

Iterator-based design for parallelism

Iterators are a generalization of pointers and constitutes the basic interface connecting all STL containers with algorithms.

- Iterators are lightweight objects that can be copied with insignificant computing costs.
- 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.
- Compatible with C++ standard distributions.

```cpp
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;

   //increment and decrement operators
   iterator_type operator--() const;
   iterator_type operator++() const;

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

Example 1: iterating over streams

Creating and iterating over uniform and exponential streams:

```cpp
#include <random_iterator/Stream.hpp>
#include <random>
...
//generator
random_iterator::squares3_128 RNG(0x548c9decbce65295);
//std distributions
std::uniform_real_distribution<double>   uniform_dist(0.0, 1.0);
std::exponential_distribution<double>    exponential_dist(1.0);
//streams
auto uniform_stream     = random_iterator::make_stream( uniform_dist, RNG, 0);
auto exponential_stream = random_iterator::make_stream( exponential_dist, RNG, 1);

//this will run forever
for(auto unf : uniform_stream){
    for(auto exp : exponential_stream) {
        std::cout << unf << ", " << exp << std::endl;
    }
}
...
```

Example 2: full random access

```
#include <random_iterator/Stream.hpp>
#include <random>
...
//generators
random_iterator::squares3_128 RNG1(0x548c9decbce65295);
//std distributions
std::uniform_real_distribution<double>   uniform_dist(0.0, 1.0);
std::exponential_distribution<double>    exponential_dist(1.0);
//streams
auto uniform_stream     = random_iterator::make_stream( uniform_dist, RNG1, 0);
auto exponential_stream = random_iterator::make_stream( exponential_dist, RNG1, 1);

//secondary generator
random_iterator::philox      RNG2(0x148c9decade547892);
std::uniform_int_distribution<uint64_t>   uint_dist(RNG.min(), RNG.max());
auto uint_stream = random_iterator::make_stream( uint_dist, RNG2, 0);

//this will run quickly
for(size_t i; i< 1024 ; ++i)
  std::cout << uniform_stream[ uint_dist[i] ] << ", "
            << exponential_stream[ uint_dist[i] ]
            << std::endl;
...
```

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.

Testing streaming scheme consistency

1. For each generator, using the same seed, instantiate two different `std::uniform_int_distribution<uint64_t>` streams (A and B), picked-up randomly in the range $[0, 2^{32}]$.
2. Get a random integer i in the range $[0, 2^{64}]$ from a different generator (different seed also).
3. Make the comparison `A[i] != B[i]`.
4. Test will fail if any of the numbers in the same, random, position are the equal over different streams.

I introduced an intentional bug and the test got it quickly enough. I removed the bug and all tests passes. I tested as much as 2^{16} streams pairs. For each stream pair, 2^{16} pseudorandom number pairs.

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

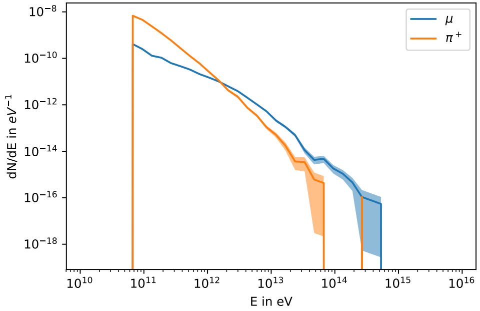
- Currently CORSIKA 8 uses `std::mt19937_64`, the Mersenne Twister (MT) implementation of the C++17 Standard Library, as its primary pseudorandom number generator.
- MT is known to fail statistical tests. It also stores a huge state, of almost 2.5 kB, and operates strictly sequentially.

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

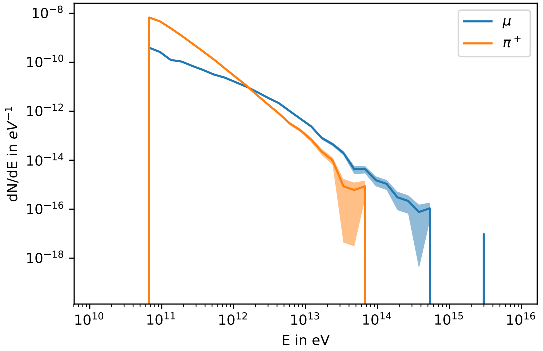
- Refactory of the internal algorithms is not required.
- The distribution and management of multiple instances of CORSIKA 8, configured with different seeds and running in parallel on clusters and other distributed systems is not impacted.
- Enables further development of more fine-grained parallelism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

Examples of showers

Philox



MT



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.

Comments

How much data can a stream to handle?

1. Each stream has a length of 2^{64}. It means, can produce 2^{64} `uint64_t` numbers.
2. Each `uint64_t` has 8 bytes.
3. Make the math: $2^{64} \times 2^{3} = 2^{67}$ bytes or...

128 Exabyte !!!

For each {seed, generator} combination we have 2^{32} of such streams.

Conclusions

- 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 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.
- In addition to the generators from Random123 library:
  - An upgraded version of the Squares algorithm with highly efficient internal 128 bit counters is introduced.
  - New ARS engine able to handle 64bit arithmetic.
- A public repository is being organized to share the code under a liberal license.

Thanks