Toward performance enhancements in CORSIKA 8

Random number generation and multithreading

A. Augusto Alves Jr

Presented CORSIKA 8 group internal meeting - KIT, Karlshuhe

December 8, 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, and registers, 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.

```
1  template<typename Distribution, typename Engine>
2  class Stream
3  {
4   public:
5
6     //constructor
7     Stream( Distribution const& dist, uint64_t seed, uint32_t stream );
8
9     //stl-like iterators
10    iterator_type begin() const;
11    iterator_type end() const;
12
13    //access operators
14    return_type operator[](size_t n) const;
15    return_type operator()(void);
16    return_type operator()(size_t);
17 };
```

Status of the integration into CORSIKA 8

- Integrated into CORSIKA 8, via `random_iterator` library.
- Presented at vCHEP-2021 and published in:

```
Counter-based pseudorandom number generators for CORSIKA 8:
A multi-thread friendly approach

A. Augusto Alves Jr, Anton Poctarev and Ralf Ulrich

EPJ Web Conf., 251 (2021) 03039
Published online: 23 August 2021
DOI: https://doi.org/10.1051/epjconf/202125103039
```

- More details in bachelor thesis of Anton Poctarev (2021)

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 !!!

Remember that: 1 EB = 2^{20} TB

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

Multithreading in CORSIKA 8: circumstances and boundary conditions

Acceleration of Monte Carlo simulation applications using multithreading is not easy.
Deployment of multithreading in CORSIKA 8 involves the following aspects:

- The cost of the calculations performed by CORSIKA 8 is overwhelmingly due calls to modules. Currently CORSIKA 8 is at least 20 - 30% slower than CORSIKA 7, and it is unlikely that it will ever be faster without concurrency.
- CORSIKA 8 calls user's code and modules, which invoke RNGs an unbounded number of times. In order to be callable concurrently, such code should be thread-safe. None of the existing modules pass this requirement. On the other hand, `random_iterator` does, but keeping an RNG state – even in a thread-safe capsule – is problematic because out-of-the-order calls between rounds.
- Simulation time increases quickly with the energy of the primary particle due the proliferation of secondary particles to be processed at each step. So, in order to exploit Amdahl's law, particles should be processed concurrently, and simulation should scale with the number of available cores/threads.
- Dedicated infrastructure should be built to manage multithreading, ease job submission and avoid the prohibitive thread creation/destruction cost. Enters `Gyges`.

Amdahl's law

- Predicts the expected speedup from parallelism:

```
Validity of the Single Processor Approach to Achieving
Large-Scale Computing Capabilities

Amdahl, Gene M.

AFIPS Conference Proceedings (30): 483-485 (1967)
doi:10.1145/1465482.1465560
```

- It is expressed as

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

where: $S(n)$ is the speedup in function of the number of cores/threads. n is number of cores/threads and p is the fraction of code that is parallelizable.

Comments

- Processing particles in parallel and scaling over the number of cores/threads should follow closely the Amdahl's law even if the number of particles to process is bigger than the number of cores/threads
- To achieve this behavior, dynamical job submission/monitoring and smart thread pooling should be deployed together.
- Each simulation round would be processed in parallel. The processing time will be dominated by the longer lasting job.
- Given jobs have durations spanning over a range, at a given round some threads can process more jobs, while others are busy with longer tasks.
- Currently, each particle is processed sequentially and the overall duration is the accumulation of each particle processing time.

Calculation model

- The simulation is managed in rounds. Simulation starts at first round, with the interaction of primary with the media. The generated particles will be processed in the second round. The products of this round will be processed at third... and so on.
- Simulation ends when a round produces no particles to be processed.
- Output is managed using side effects.
- Input data, RNG, geometry, filters etc are services available to modules, and accessible from the processing threads in read-only mode.
- The simulation manager thread can operates aside a IO manager thread, a monitoring thread etc. The worker threads are commissioned and released by the simulation manager thread.

Open question: How to ensure repeatability of the results calling the RNG concurrently ?

Gyges

`Gyges` is a lightweight C++20 header-only library to manage thread pooling.

- With `Gyges`, thread creation and destruction costs are paid just once in the program lifetime.
- Threads from the pool pick-up tasks as they became available. If there is no task, the threads just sleep.
- Tasks can be submitted from multiple threads.
- The submitter gets a `std::future` for monitoring the task in-place.
- Task assignment and running can be interrupted at any time.
- A `gyges::gang` can be created with any number of threads.

Status: In final development stage. Basically, just testing and documentation pending.

Gyges example

```
1  #include <thread>
2  #include <iostream>
3  #include <random>
4  #include <vector>
5  #include <gyges/gang.hpp>
6
7
8  int main(int argv, char** argc)
9  {
10         //number of random numbers to accumulate per task
11         unsigned max_nr = 1000000000;
12
13         // it will create a gang with the number
14         // of cores supported by the hardware.
15         gyges::gang tpool{};
16
17         std::cout << "The gang has #" << tpool.size() << " workers\n";
18
19         //tasks will accumulate max_nr of random numbers
20         //and set the result in the corresponding position of a vector
21
22         std::vector<double> results(tpool.size(), 0.0);
23         std::vector<std::future<void>> monitors;
```

Gyges example

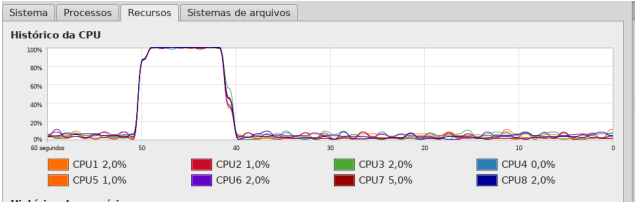
```
1  for(std::size_t i=0; i< tpool.size() ; ++i)
2  {
3      //used to obtain a seed for the random number engine
4      std::random_device rd;
5      auto seed = rd();
6      //where to place the result
7      auto result_iterator = results.begin() + i;
8
9      //lambda function getting the necessary parameters to perform the task.
10     auto Task = [ result_iterator, max_nr, seed ](std::stop_token t) {
11
12         double partial_result = 0;
13         std::mt19937 generator( seed );
14         std::uniform_real_distribution<double> distribution(0.0, 1.0);
15
16         for( unsigned nr = 0; nr< max_nr; ++nr)
17         partial_result+=distribution(generator);
18         //set results
19         *(result_iterator) = partial_result;
20     };
21     // task submission
22     auto future = tpool.submit_task( Task );
23     monitors.push_back( std::move(future) );
24
25 }//close for loop
```

Gyges example

```
1         //check the tasks and  print the result
2         for(std::size_t i=0; i< monitors.size(); ++i  ){
3                 monitors[i].get();
4                 std::cout << "Task #" << i  << " completed. Result: "<<        results[i] << std::endl;
5         }
6
7     //stop the gang
8         tpool.stop();
9
10        return 0;
11 }
```

Gyges example

```
1  [augalves@LabHome Gyges_Proj] $ ./examples/use_gangs
2  The gang has #8 workers
3  Task #0 completed. Result: 4.99999e+08
4  Task #1 completed. Result: 4.99998e+08
5  Task #2 completed. Result: 4.99998e+08
6  Task #3 completed. Result: 4.99975e+08
7  Task #4 completed. Result: 4.99992e+08
8  Task #5 completed. Result: 5.00011e+08
9  Task #6 completed. Result: 5.00009e+08
10 Task #7 completed. Result: 4.99997e+08
11 [augalves@LabHome Gyges_Proj] $
```



Basically $6x10^9$ calls to RNG plus the accumulation operation performed in about 10s.

Thanks

Backup

Example 1: iterating over streams

Creating and iterating over uniform and exponential streams:

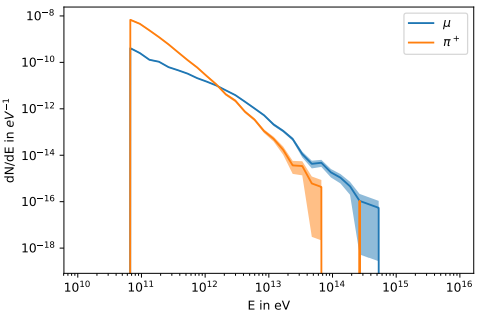
```
#include <random_iterator/Stream.hpp>
#include <random>
...
//generator
random_iterator::squares3_128 RNG(0x548c9decbce65295);
//std distributions
std::uniform_real_distribution<double>   uniform_dist(0.0, 1.0);
std::exponential_distribution<double>    exponential_dist(1.0);
//streams
auto uniform_stream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

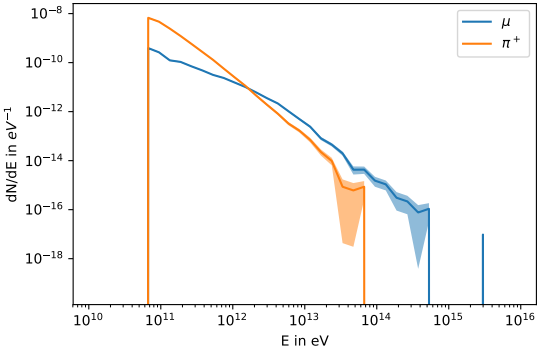
```
#include <random_iterator/Stream.hpp>
#include <random>
...
//generators
random_iterator::squares3_128 RNG1(0x548c9decbce65295);
//std distributions
std::uniform_real_distribution<double>   uniform_dist(0.0, 1.0);
std::exponential_distribution<double>    exponential_dist(1.0);
//streams
auto uniform_stream     = 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_stream[i] ] << ", "
            << exponential_stream[ uint_stream[i] ]
            << std::endl;
...
```

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.

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.

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.

Thanks