<Random>
Harder Better Faster Stronger

Adrien Devresse
Harder, Better, Faster, Stronger
About me

• Around 15 years of C++

• **Background** in High Performance Computing, Distributed systems, Embedded systems.

• **Worked at**
  • CERN
  • Human Brain Project
  • Startup
  • Woven By Toyota
  • (Onboarding) Square Kilometer Array
Disclaimer

This talk is about:

- **Practical** usages of pseudo random number generators. Not the theory behind them

- Centered on **C++11 <Random>**

- Aims to non-cryptographic applications **only**

- **Focus** on Scientific Computing
The little story of Bob

Bob
Average Phd
Student

Write a scientific simulator to finish his Phd

Procrastinate
Few things about **Scientific Computing**

- Most of Scientific Simulators are written in C++
- They need to be **fast**
- They rely heavily on **Random number generation**
- They need the random streams to have **excellent randomness** properties
- They use extensively **parallel processing**
Easy
Choose your weapon PRNG

C

C++

<random>
Harder
// Seed
std::srand()

// Generate
std::rand()
Not that bad

**PRNG**

---

**Statistical Distributions**

---

**Random number distributions**

A random number distribution is a distribution over possible values of a random variable. These distributions are important in many applications, such as in statistical simulations, computer science, and engineering. The distributions listed below are implemented in C++ and can be used for generating random numbers according to specific probability density functions. Each distribution class is tailored to produce values that follow the characteristics of the specified statistical distribution, ensuring accurate and meaningful simulations and analyses.

- **Uniform distributions**
  - `uniform_int_distribution` produces integer values evenly distributed across a specified range.
  - `uniform_real_distribution` produces real values evenly distributed across a specified range.

- **Bernoulli distributions**
  - `bernoulli_distribution` produces binary outcomes (0 or 1) based on a given probability.
  - `binomial_distribution` produces the number of successes in a given number of trials, following a binomial distribution.

- **Negative Binomial distributions**
  - `negative_binomial_distribution` produces the number of trials needed to achieve a specified number of successes, following a negative binomial distribution.

- **Geometric distributions**
  - `geometric_distribution` produces the number of failures before the first success, following a geometric distribution.

- **Poisson distributions**
  - `poisson_distribution` produces integer values following a Poisson distribution.
  - `exponential_distribution` produces real values following an exponential distribution.

- **Gamma distributions**
  - `gamma_distribution` produces real values following a gamma distribution.

- **Chi-squared distributions**
  - `chi_squared_distribution` produces real values following a chi-squared distribution.

- **Normal distributions**
  - `normal_distribution` produces real values following a normal distribution.

- **Lognormal distributions**
  - `lognormal_distribution` produces real values following a lognormal distribution.

---
## PRNGs more in details

<table>
<thead>
<tr>
<th>PRNG</th>
<th>Quality</th>
<th>Speed</th>
<th>Periodicity</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ranlux</td>
<td>★★★</td>
<td>✰✰✰</td>
<td>$10^{166}$</td>
<td>1994</td>
</tr>
<tr>
<td>Mersenne Twister</td>
<td>★★★</td>
<td>✰✰✰</td>
<td>$10^{19937}$</td>
<td>1998-2003</td>
</tr>
<tr>
<td>Minstd</td>
<td>★</td>
<td>✰✰✰</td>
<td>Short</td>
<td>1988-1993</td>
</tr>
<tr>
<td>Knutb</td>
<td>★</td>
<td>✰✰✰</td>
<td>Short</td>
<td>1988</td>
</tr>
</tbody>
</table>

std::default_random_engine -> Minstd
More on Mersenne Twister

- Developed by Makoto Matsumoto and Takuji Nishimura

- Gold standard of PRNG

- Implemented in almost all major programming languages
std::uint64_t seed = 42;

std::mt19937_64 rng(seed);

for(std::size_t i = 0; i < dataset.size(); i++){
    std::uniform_int_distribution dist;
    auto number = dist(rng);
    my_kernel(number, dataset[i]);
}
std::uint64_t seed = 42;
std::mt19937_64 rng(seed);
std::mutex mut;

#pragma omp parallel for
for(std::size_t i = 0; i < dataset.size(); i++){
    std::uniform_int_distribution dist;
    auto number = [](){
        std::unique_lock guard(mut);
        return dist(rng);
    }();
    my_kernel(number, dataset[i]);
}
std::uint64_t seed = 42;
thread_local std::mt19937_64 rng;
thread_local bool rng_initialized = false;

#pragma omp parallel for
for(std::size_t i = 0; i < dataset.size(); i++){
    std::uniform_int_distribution dist;
    if(!rng_initialized ){
        rng.seed(seed * total_thread_numbers + thread_id);
        rng_initialized = true;
    }
    auto number = dist(rng);
    my_kernel(number, dataset[i] );
}
std::uint64_t seed = 42;

thread_local std::mt19937_64 rng(seed);
thread_local bool rng_initialized = false;

#pragma omp parallel for
for( std::size_t i = 0; i < dataset.size(); i++ ){
    std::uniform_int_distribution dist;
    if(!rng_initialized ){
        rng.discard( (1 << 32) * thread_id);
        rng_initialized = true;
    }
    auto number = dist(rng);
    my_kernel(number, dataset[i]);
}
Better
A Solution?

\[ f(k, d) = R \]
\[ P(R) \equiv N \]

- \( f \): Function
- \( k \): Key
- \( d \): Data
- \( R \): Result
- \( N \): Random Noise

Cryptographic Block Cipher
Counter Based Pseudo Random Number Generator (CBRNG)

Key

Counter

Block Cipher

Random content

\[ N_0 : \text{Key} = \text{seed}, \ C = 0 \]
\[ N_1 : \text{Key} = \text{seed}, \ C = 1 \]
\[ N_2 : \text{Key} = \text{seed}, \ C = 2 \]

• No problem to derivate the seed in parallel algorithms
  • K = tuple( seed, computing_bloc_id, ...) 
• C is known and predictable
  • Our random generator can be stateless
std::uint64_t seed = 42;
constexpr std::uint64_t block_size = 1024;

#pragma omp parallel for
for(std::size_t i = 0; i < dataset.size(); i++){
    std::uniform_int_distribution dist;
    alea::counter_rng<threefry> gen;
    gen.set_key({ seed, i / block_size });
    get.set_counter( i % block_size );
    auto number = dist(rng);
    my_kernel(number, dataset[i]);
}
Faster
Random123 to the rescue

- **Publication** *Parallel Random Numbers: As Easy as 1, 2, 3, Salmon, Moraes, Dror & Shaw, Supercomputing, 2011*

- Define a entire family of **CBRNG**

- Based (Mostly) on battle-tested **Block cipher algorithms**

- **Reduced number of rounds** for a speed tradeoff
Random123: 3 CBRNGs

- AES
  - Has dedicated CPU instructions

- Twofish
  - Light and generic

- Threelfry

- Philox
  - Relies on multiplications
## Performances

<table>
<thead>
<tr>
<th>PRNG</th>
<th>CPU (Intel) GB/s</th>
<th>GPU (Nvidia) GB/s</th>
<th>GPU (AMD) GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mersenne Twister</td>
<td>6.1</td>
<td>18.3</td>
<td>N/A</td>
</tr>
<tr>
<td>ARS (With Hardware instructions)</td>
<td><strong>11.1</strong></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Threefry</td>
<td>6.4</td>
<td>52.8</td>
<td>46.4</td>
</tr>
<tr>
<td>Philox</td>
<td>3.4</td>
<td>145.3</td>
<td>79.1</td>
</tr>
</tbody>
</table>
Stronger
Random Quality

• Quality of Random generator is evaluated based on their statistical properties

• The reference test suite for that named TestU01
  • Publication: TestU01: A C library for empirical testing of random number generators

• The test suit is divided into two components:
  • Crush (96 tests)
  • Big Crush (106 tests)
# TestU01 conformance

<table>
<thead>
<tr>
<th>PRNGs</th>
<th>Crush compliance</th>
<th>Big Crush compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mersenne Twister</td>
<td>Full</td>
<td>Partial</td>
</tr>
<tr>
<td>ARS</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>Threetfry</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>Philox</td>
<td>Full</td>
<td>Full</td>
</tr>
</tbody>
</table>
Conclusions

• `<random>` is an amazing piece of code within the STL but
  • It aged (C++11)
  • The defaults shall (probably) be revisited
  • It can be **hard** to use right

• CBRNGs provide **better** algorithms for Many-cores applications

• CBRNGs can often be **faster** than most `<random>` algorithms

• **Random123** algorithms provide **stronger** randomness quality than most `<random>` algorithms

Opinion: I believe we shall update the PRNGs algorithms within the STL. **A good candidate** for a Beman project ?
Twitter: https://twitter.com/adev
Mastodon: @adev@hostux.social
Discord: adev9145