C++ now

<Random> Harder Better Faster Stronger

Adrien Devresse

2024

<Random>

Harder, Better, Faster Stronger

About me

Adrien Devresse

- 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

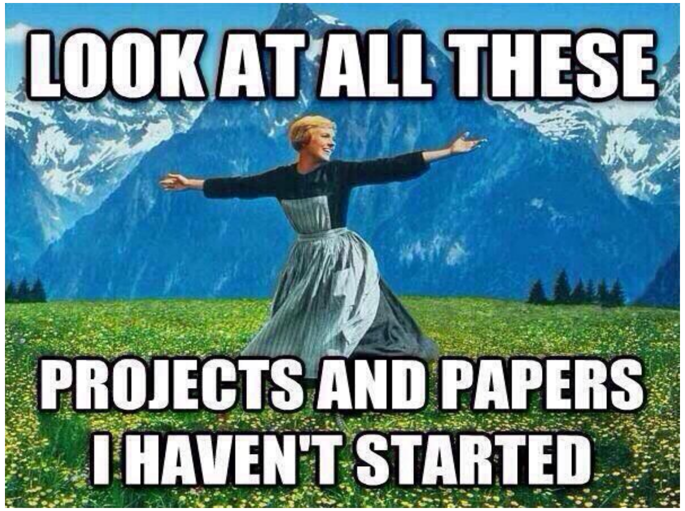
- Centered on **C++11 <Random>**

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

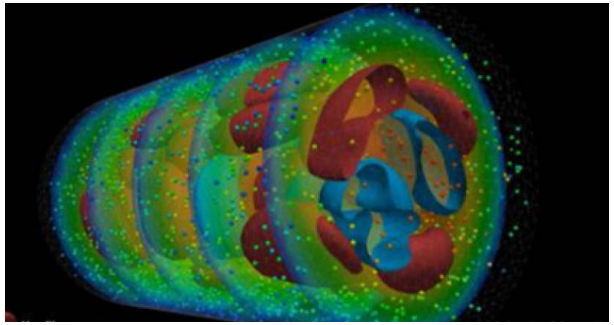
- **Focus** on Scientific Computing

The little story of Bob

Bob
Average Phd Student



Procrastinate



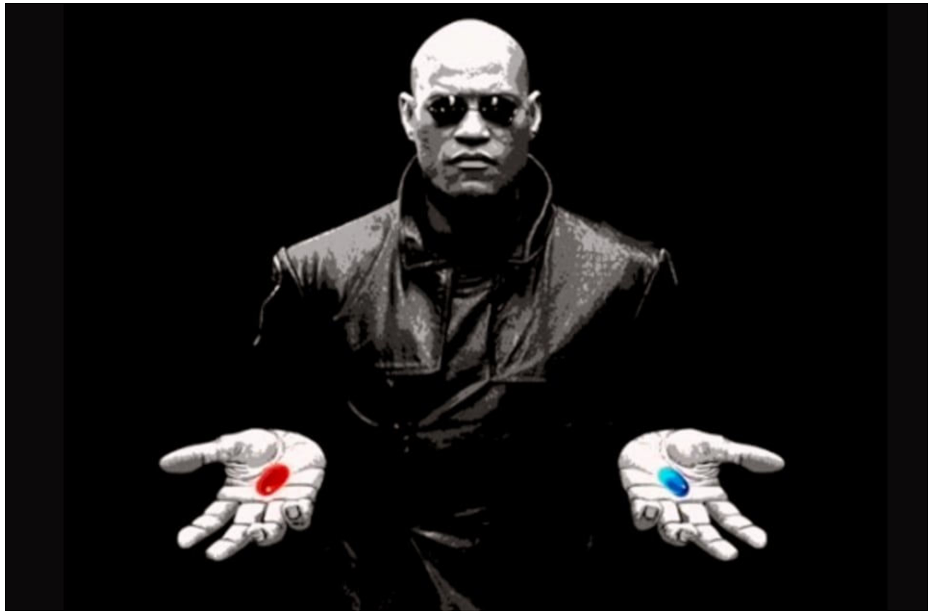
Write a scientific simulator to finish his Phd

Few things about Scientific Computing

- Most of Scientific Simulators are written in
- 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

~~<rand>~~

C++

<random>

Harder

C

```
// Seed
std::srand()

// Generate
std::rand()
```

C++

minstd_rand0 (C++11)	std::linear_congruential_engine<std::uint_fast32_t, 16807, 0, 2147483647> Discovered in 1969 by Lewis, Goodman and Miller, adopted as "Minimal standard" in 1988 by Park and Miller
minstd_rand (C++11)	std::linear_congruential_engine<std::uint_fast32_t, 48271, 0, 2147483647> Newer "Minimum standard", recommended by Park, Miller, and Stockmeyer in 1993
mt19937 (C++11)	std::mersenne_twister_engine<std::uint_fast32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253> 32-bit Mersenne Twister by Matsumoto and Nishimura, 1998
mt19937_64 (C++11)	std::mersenne_twister_engine<std::uint_fast64_t, 64, 312, 156, 31, 0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67fffeda60000, 37, 0xfff7eee000000000, 43, 6364136223846793005> 64-bit Mersenne Twister by Matsumoto and Nishimura, 2000
ranlux24_base (C++11)	std::subtract_with_carry_engine<std::uint_fast32_t, 24, 10, 24>
ranlux48_base (C++11)	std::subtract_with_carry_engine<std::uint_fast64_t, 48, 5, 12>
ranlux24 (C++11)	std::discard_block_engine<std::ranlux24_base, 223, 23> 24-bit RANLUX generator by Martin Lüscher and Fred James, 1994
ranlux48 (C++11)	std::discard_block_engine<std::ranlux48_base, 389, 11> 48-bit RANLUX generator by Martin Lüscher and Fred James, 1994
knuth_b (C++11)	std::shuffle_order_engine<std::minstd_rand0, 256>
default_random_engine (C++11)	implementation-defined

Non-deterministic random numbers

std::random_device is a non-deterministic uniform random bit generator, although implementations are allowed to implement std::random_device using a pseudo-random number engine if there is no support for non-deterministic random number generation.

random_device (C++11)	non-deterministic random number generator using hardware entropy source (class)

Random number distributions

A random number distribution post-processes the output of a URBG in such a way that resulting output is distributed according to a defined statistical probability density function.

Random number distributions satisfy *RandomNumberDistribution*.

Defined in header <random>

Uniform distributions

uniform_int_distribution (C++11)	produces integer values evenly distributed across a range (class template)
uniform_real_distribution (C++11)	produces real values evenly distributed across a range (class template)

Bernoulli distributions

bernoulli_distribution (C++11)	produces bool values on a Bernoulli distribution (class)
binomial_distribution (C++11)	produces integer values on a binomial distribution (class template)
negative_binomial_distribution (C++11)	produces integer values on a negative binomial distribution (class template)
geometric_distribution (C++11)	produces integer values on a geometric distribution (class template)

Poisson distributions

poisson_distribution (C++11)	produces integer values on a Poisson distribution (class template)
exponential_distribution (C++11)	produces real values on an exponential distribution (class template)
gamma_distribution (C++11)	produces real values on a gamma distribution (class template)
weibull_distribution (C++11)	produces real values on a Weibull distribution (class template)
extreme_value_distribution (C++11)	produces real values on an extreme value distribution (class template)

Normal distributions

normal_distribution (C++11)	produces real values on a standard normal (Gaussian) distribution (class template)
lognormal_distribution (C++11)	produces real values on a lognormal distribution (class template)
chi_squared_distribution (C++11)	produces real values on a chi-squared distribution (class template)

Not **that** bad

PRNG

minstd_rand0 (C++11)	std::linear_congruential_engine<std::uint_fast32_t, 16807, 0, 2147483647> Discovered in 1969 by Lewis, Goodman and Miller, adopted as "Minimal standard" in 1988 by Park and Miller
minstd_rand (C++11)	std::linear_congruential_engine<std::uint_fast32_t, 48271, 0, 2147483647> Newer "Minimum standard", recommended by Park, Miller, and Stockmeyer in 1993
mt19937 (C++11)	std::mersenne_twister_engine<std::uint_fast32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253> 32-bit Mersenne Twister by Matsumoto and Nishimura, 1998
mt19937_64 (C++11)	std::mersenne_twister_engine<std::uint_fast64_t, 64, 312, 156, 31, 0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67fffeda60000, 37, 0xfff7eee000000000, 43, 6364136223846793005> 64-bit Mersenne Twister by Matsumoto and Nishimura, 2000
ranlux24_base (C++11)	std::subtract_with_carry_engine<std::uint_fast32_t, 24, 10, 24>
ranlux48_base (C++11)	std::subtract_with_carry_engine<std::uint_fast64_t, 48, 5, 12>
ranlux24 (C++11)	std::discard_block_engine<std::ranlux24_base, 223, 23> 24-bit RANLUX generator by Martin Lüscher and Fred James, 1994
ranlux48 (C++11)	std::discard_block_engine<std::ranlux48_base, 389, 11> 48-bit RANLUX generator by Martin Lüscher and Fred James, 1994
knuth_b (C++11)	std::shuffle_order_engine<std::minstd_rand0, 256>
default_random_engine (C++11)	implementation-defined

Non-deterministic random numbers

std::random_device is a non-deterministic uniform random bit generator, although implementations are allowed to implement std::random_device using a pseudo-random number engine if there is no support for non-deterministic random number generation.

random_device (C++11)	non-deterministic random number generator using hardware entropy source

Statistical Distributions

Random number distributions

A random number distribution post-processes the output of a URBG in such a way that resulting output is distributed according to a defined statistical probability density function.

Random number distributions satisfy *RandomNumberDistribution*.

Defined in header <random>

Uniform distributions

uniform_int_distribution (C++11)	produces integer values evenly distributed across a range (class template)
uniform_real_distribution (C++11)	produces real values evenly distributed across a range (class template)

Bernoulli distributions

bernoulli_distribution (C++11)	produces bool values on a Bernoulli distribution (class)
binomial_distribution (C++11)	produces integer values on a binomial distribution (class template)
negative_binomial_distribution (C++11)	produces integer values on a negative binomial distribution (class template)
geometric_distribution (C++11)	produces integer values on a geometric distribution (class template)

Poisson distributions

poisson_distribution (C++11)	produces integer values on a Poisson distribution (class template)
exponential_distribution (C++11)	produces real values on an exponential distribution (class template)
gamma_distribution (C++11)	produces real values on a gamma distribution (class template)
weibull_distribution (C++11)	produces real values on a Weibull distribution (class template)
extreme_value_distribution (C++11)	produces real values on an extreme value distribution (class template)

Normal distributions

normal_distribution (C++11)	produces real values on a standard normal (Gaussian) distribution (class template)
lognormal_distribution (C++11)	produces real values on a lognormal distribution (class template)
chi_squared_distribution (C++11)	produces real values on a chi-squared distribution

PRNGs more in details

	Quality	Speed	Periodicity	Age
Ranlux	★★★	🐇	10^166	1994
Mersenne Twister	★★	🐇🐇	10^19937	1998-2003
Minstd	★	🐇🐇🐇	Short	1988-1993
Knutb	★	🐇🐇	Short	1988

⚠ **std::default_random_engine -> Minstd**

More on Mersenne Twister

- Developed by Makoto Matsumoto and Takuji Nishimura
    - *Mersenne Twister: A 623-DimensionallyEquidistributed Uniform Pseudo-Random Number G*

- **Gold sta**

- Implemen guages

Lets see how it looks Sequential

```
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] );
}
```

Lets see how it looks Parallel

Version 0

```
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] );
}
```

Lets see how it looks Parallel

Version 1

```
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] );
}
```

Lets see how it looks Parallel

Version 2

```
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

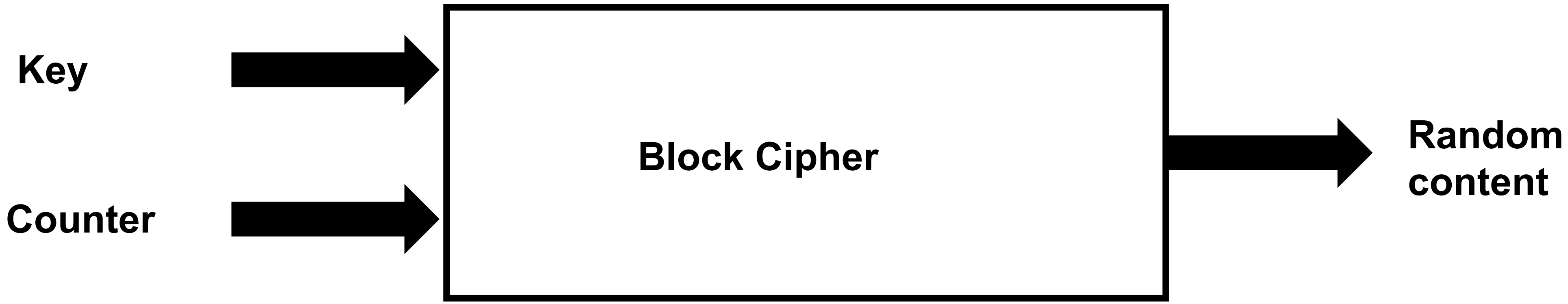
A Solution ?

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

f: Function
k: Key
d: Data
R: Result
N: Random Noise

→ **Cryptographic Block Cipher**

Counter Based Pseudo Random Number Generator (CBRNG)



N_0 : Key=seed, C=0
N_1: Key=seed, C=1
N_2: Key=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**

Back to how it looks

Version CBRNG

```
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

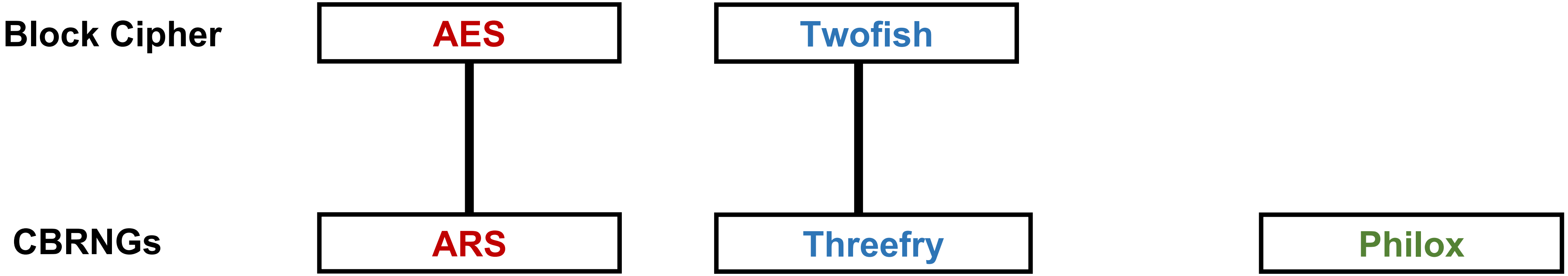
- **Publication** *Parallel Random Numbers: As Easy as 1, 2, 3, Salmon, Moraes, Dror & Shaw*, **Supercomputing, 2011**

- 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 is light and generic

Philox relies on multiplications

Performances

PRNG	CPU (Intel) GB/s	GPU (Nvidia) GB/s	GPU (AMD) GB/s
Mersenne Twsiter	6.1	18.3	N/A
ARS (With Hardware instructions)	**11.1**	N/A	N/A
Threefry	**6.4**	**52.8**	**46.4**
Philox	3.4	**145.3**	**79.1**

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

PRNGs	Crush compliance	Big Crush compliance
Mersenne Twister	Full	Partial
ARS	Full	Full
Threefry	Full	Full
Philox	Full	Full

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