Parallel Random Number Generators: VecRNG

Soon Yung Jun (Fermilab)
John Apostolakis (CERN)
2018 Geant4 Collaboration Meeting, Lund, Sweden

August, 31, 2018

Contents

Introduction

- High performance simulation modeling requires high quality parallel pseudo random number generation (pRNG) to support concurrent tasks in multi-dimensional hardware architectures
- Parallel pseudo random number generation
  - Intel MKL/VSL library [1] and NVidia Curand library [2]
  - Many other customs libraries for SIMD (SSE, AVX) or GPU
- Motivation
  - Common random numbers (CRN) for hybrid computing models
  - Support both SIMD and SIMT with portable codes
  - Requirement of reproducibility

Choice of Generators: An initial list

- One of representative generators from major classes of pRNG
  - MRG32k3a [3]: Algorithm (Multiple Recursive Linear Congruent)
  - Random123 [4] : Counter based (Advance Randomization System)
  - MIXMAX [5]: Anosov C-system
- Meet general requirements for quality and performance
  - Long period ($\geq 2^{200}$) and fast with a small state (memory)
  - Repeatability in sequence on the same hardware configuration
  - Efficient ways of splitting the sequence into long disjoin steams
  - Crush-resistant: pass DIEHARD [6] and BigCrush of TestU01 [7]

Generator	Scalar State	Memory	Period	Stream
MRG32k3a	6 doubles	48 bytes	2^{191}	2^{64}
Threefry(4x32)	12 32-bit int	48 bytes	2^{256}	2^{128}
Philox(4x32)	10 32-bit int	40 bytes	2^{192}	2^{64}
MIXMAX(N=17)	17 32-bit int	68 bytes	10^{294}	$\sim \infty$

2x memory for 64-bit states for Random123 and MIXMAX

Status of Implementation: VecRNG under VecMath

- Repository: Rng under VecMath

```
git clone https://github.com/root-project/vecmath
```

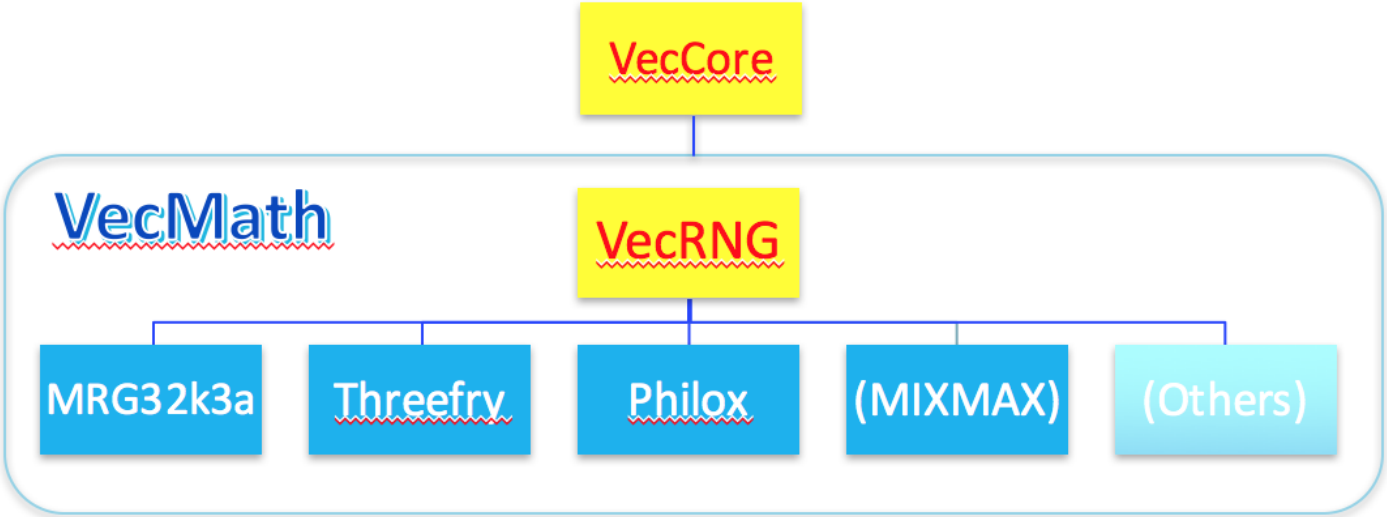
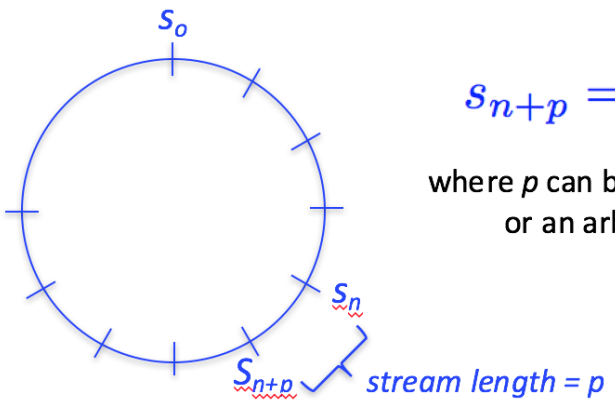


Figure: Hierarchy of VecRNG

- Design Choice
  - Header only implementation with the static polymorphism
  - Support SIMD and GPU with a common kernel using VecCore [8]
  - Simple and light: pRNG object $\simeq$ pRNG state

Requirements

- Multiple streams and an efficient skip-ahead algorithm (f_p): i.e., advancing a state by p-steps (sequences) to fully control parallel or multiple tasks independently



$$s_{n+p} = f_p(s_n)$$

where p can be the stream length or an arbitrary number

- Support both scalar and vector backend [for a given stream]

```
MRG32k3a<Backend> rng();
rng.Initialize(streamID);
Backend::Double_v rv = rng->Uniform<Backend>();
```

- Independent multiple streams can be assigned to/by
  - process id, thread number or the rank in mpi
  - job, task numbers or track ID, etc.

Special Features

- Vector backend uses N(=SIMD length) consecutive substreams and supports both vector and scalar return-type, for an example,

```
MRG32k3a<VectorBackend> rng();
rng.Initialize(); // move to the next available stream
double_v rv = rng->Uniform<VectorBackend>();
double   rs = rng->Uniform<ScalarBackend>();
```

- Working with states
  - Uniform method can take a state directly and a generate random number and update the state (both scalar and vector backend)
  - an example use for GPU

```
State_t * states = ... CPU/GPU allocation mechanism ...

vecrng::cuda::MRG32k3a<ScalarBackend> rng();
rng->Initialize(states, ntid);   // states[ntid]
double rs = rng->Uniform<ScalarBackend>(&states[tid]);
```

Preliminary Performance: AVX - Vc

- The average CPU time [ms] for generating 10M random numbers
  - std::rand() = (93.24 ± 0.03) ms

Generator	MRG32k3a	Threefry	Philox
Scalar	137.24 ± 0.05	74.13 ± 0.08	54.59 ± 0.03
Vector(AVX)	57.59 ± 0.02	43.06 ± 0.17	76.90 ± 0.22
CUDA	0.45 ± 0.05	12.12 ± 0.02	12.19 ± 0.01
Curand	0.51 ± 0.01	N/A	0.67 ± 0.05

  - Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (Ivy Bridge)
  - NVidia Tesla K40M-12GB (2880 cores)
  - The word size (W) and round (R) used for Random-123 were W4x32_R20 for Threefry and W4x32_R10 for Philox
- Performance issue of Philox
  - the primary operation is the bit bijection with the Feistel function
  - no conversion between 64 and 32 bit (SIMD) integers (scalar ops.)
  - inefficient output function or data type for GPU(will be improved)

Probability Distribution Functions

- Two strategies
  - use VecRng Uniform() to implement more complicate algorithms
  - vectorize generic algorithms directly
- Basic ones were done: Flat, Exponential, Gauss
- Vectorized $\Gamma(\vec{\alpha}, \vec{\beta})$ by Oscar Chaparro (IPN, Mexico)
  - two different algorithms for $\alpha > 1$ [9] and $0 < \alpha < 1$ [10]
  - different vector interfaces for different use cases

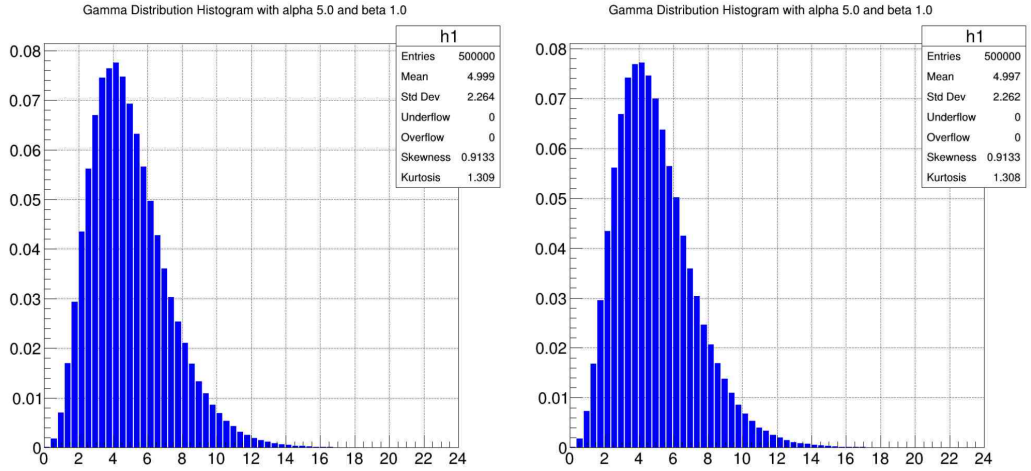
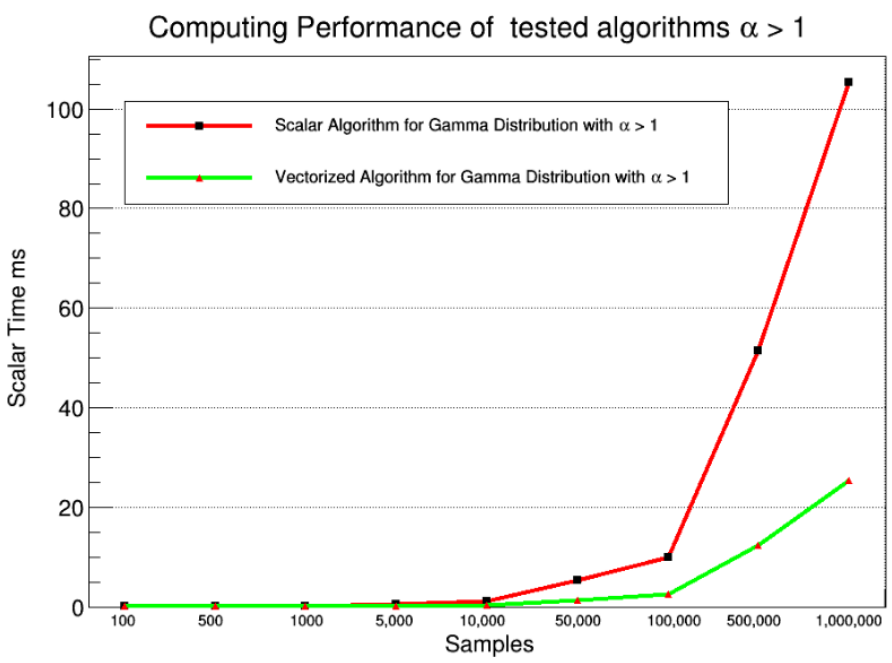


Figure: 0.5M $\Gamma(5, 1)$: scalar(left) vs. vector(right)

Probability Distribution Functions: Performance

- Intel(R) Core(TM) i7-4510U CPU 2.00GHz
- Speed-up up to 1M generations



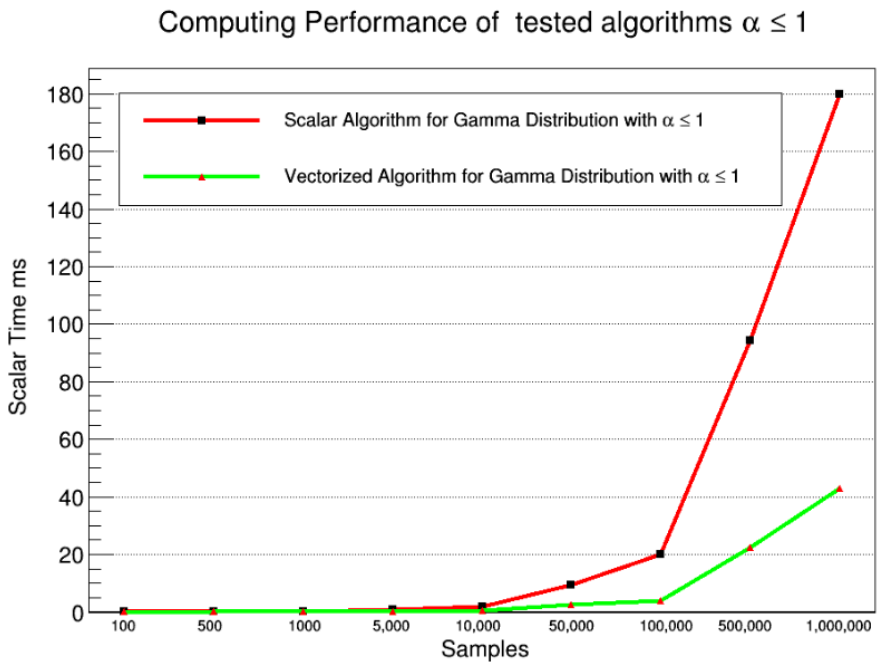


Figure: (left) $\Gamma(\alpha = 4, \beta = 1)$, (right) $\Gamma(\alpha = 0.5, \beta = 0.5)$

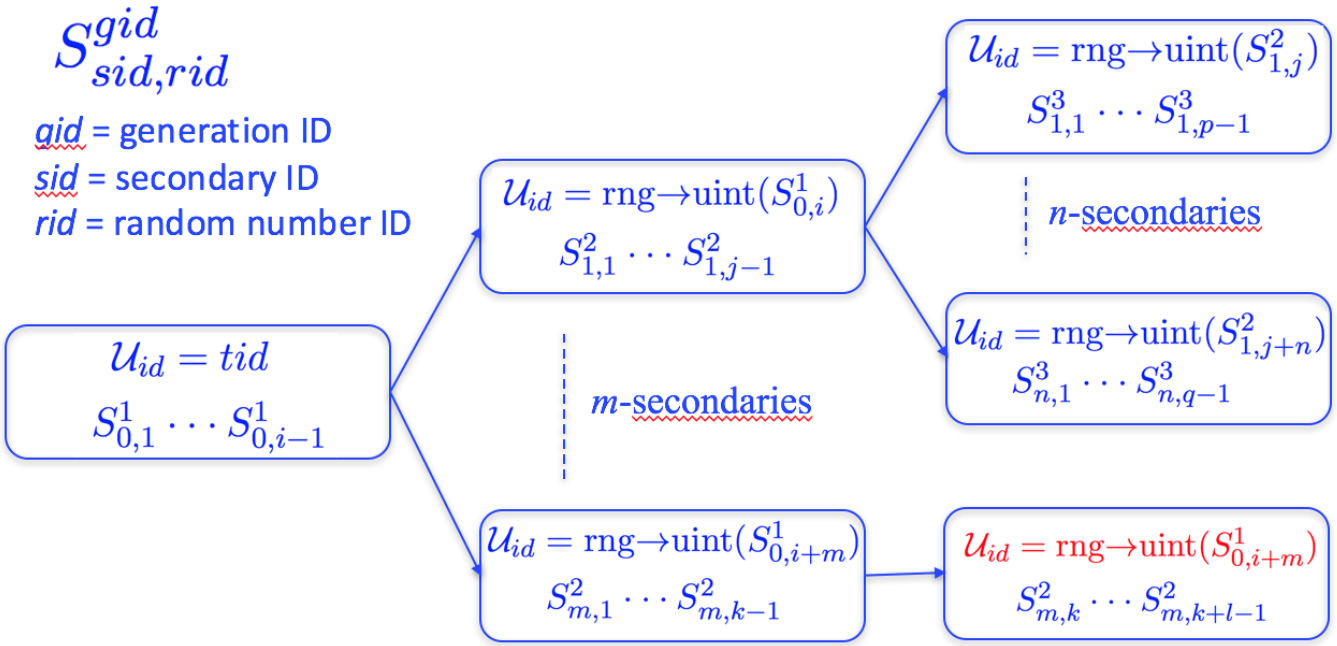
- Vector gain ~ 4 with AVX2
- Working on measuring performance on GPU

Reproducibility under Concurrent Tasks

- Reproducibility is a stringent requirement for HEP simulation
- Challenges for track-level parallelism under concurrent simulation work flows (ex. GeantV approach)
  - events are mixed and track processing order is not deterministic
  - how to utilize vectorized pRNG efficiently keeping reproducibility
- Strategies
  - a track owns a pRNG object
  - assign a unique sequence to each track in a collision-resistant and thread-independent way
    - control by seed (see the talk, 'Deterministic transport independent of the order of tracking' by D. Savin at the 2017 Collaboration meeting) - pedigree using the Merkle–Damgård hash function `https://bitbucket.org/sd57/geant4/branch/pedigree`
    - control by stream
  - keep tracing sequence of random states through vector tasks

Reproducibility: Stream Assignment (A Proposed Solution)

- $\mathcal{U}_{id}$ =unique stream ID, S= random states
  - event ID can be as the seed
  - tid = primary trackID (pre-determined)
  - sid = secondary track ID (= 0 if the track is primary)
  - rng→uint(·) = a random integer kernel or a hash function (ex: output of Random123 is a collision-resistant AES/ARS hash)



Reproducibility for Vector Tasks

- A typical workflow for a vector task with tracks[ntracks]

```
// 1. create and initialize a vector pRNG
// 2. loop over nchunk = ntracks/vsize
for (int i=0, ibase=0; i < nchunk ; ++i, ibase += vsize) { //vector loop
  // 3. gather scalar states of tracks to a vector state
  // 4. Do a vectorized task: use as many pRNG->Uniform<VectorBackend>()
  // 5. scatter the advanced vector state back to scalar rngs
}
// 6. scalar loop for the remaining (ntracks-nchunk*vsize) tracks
```

- Reproducible mode with vector tasks
  - overhead for gather/scatter (memory copy)
  - still efficient if a task uses many random numbers
- Possible implementation
  - proxy approach (join–states → generate VecRngs → split-state)
  - caching approach (generate VecRngs → store/use → refill-buffer)
  - memory footprint and copy are different
  - evaluate performances for different vector tasks

Remarks and Plans

- Issues of other algorithms reviewed
  - RanLux++ (SWB, modern version of ranlux) [12]
    Manipulation of UInt64_v interger (inefficiency)
  - LFib $(2^{64}, 607, 271)$ (states too big for a long period)
- Plans under VecMath
  - hide backend as much as possible
  - demonstrate reproducibility in concurrent workflows
  - add MIXMAX (John) and a couple of more generators
  - add popular probability distributions (variants) (Oscar)
  - add methods for integer/float return type
  - complete auxiliary interfaces (streamer, etc.)
  - integrate new ROOT RNG interfaces
  - add performance test with AVX512 (UME::SIMD [13])
  - add unit/basic statistical tests (for the purpose of development)

Acknowledgment

Thanks to Philippe Canal, Andrei Gheata and Lorenzo Moneta, Guilherme Lima for their valuable discussions.

Intel MKL/VSL library, Intel Parallel Studio 2016

NVIDIA CUDA Toolkit 5.0 CURAND Guide
http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf

P. L'Ecuyer, R. Simard, E.J. Chen, W.D. Kelton, An object-oriented random number package with many long streams and substreams, Operations Research 50 (2002) 1073-1075

J.K. Salmon, M.A. Moraes, R.O. Dror, D.E. Shaw, Parallel random numbers: as easy as 1, 2, 3, International Conference for High Performance Computing, Networking, Storage and Analysis, ACM (2011) pp. 16:1-16:12

K. Savvidy and G. Savvidy, MIXMAX Random Number Generator, arXiv:1510.06274v3 (2011), K. Savvidy, The MIXMAX Random Number Generator, arXiv:1403.5355v2 (2014)

G. Marsaglia, DIEHARD: A batter of tests of randomness (1996)
http://stat.fsu.edu/ geo/diehard.html

P. L'Ecuyer, R. Simard, TestU01: A C Library for Empirical Testing of Random Number Generators ACM Transaction on Mathematical Software, Vol. 33, No.4, Article 22 (2007)

VecCore Library, SIMD abstraction library
https://github.com/root-project/veccore

G. Marsaglia and W. Tsang, A simple method for generating gamma variables. ACM Transactions on Mathematical Software, 26(3): 363?372 (2000)

N. Hung, N. Trang, and T. QuocChien, An improvement of minh?s algorithm for generating gamma variates with any value of shape parameter. Indian Journal of Computer Science and Engineering (IJCSE), 5(6): 199?205 (2015)

Kretz M and Lindenstruth V 2011 Vc: A C++ library for explicit vectorization *Software: Practice and Experience.*
http://dx.doi.org/10.1002/spe.1149

A. Sibidanov, A revision of the substract-with-borrow random number generators, arXiv:1705.03123v1 (2017)

UME::SIMD, A library for explicit simd vectorization. https://github.com/edanor/umesimd

VCL (C++ vector class library), www.agner.org/optimize

Definition (Vague Notion)

- pRNG: pseudo random number generator (deterministic)
- State : a set of parameters to generate random numbers
  - seed: parameters to initiate a state
  - sequence: a series of statistically unpredictable states
- Period: the maximum, over all states, of the length of repetition-free prefix of the sequence.
- Stream : an independent sequence of a pRNG (a set)
- Multiple streams: parallel sequences starting from different states without overlapping and exchanging data between them
- Reproducibility: exactly replicable sequences which produce exactly the same results when running a program on the same computer architecture

An Example of pRNG with Multiple Streams

- State $\{s_j\}$ of MRG32k3a: $s_{j,n+\nu} = (A_j^\nu \bmod m_j) s_{j,n} \bmod m_j$ (sequence index n, period $\rho = 2^{191}$, stream length $\nu = 2^{127}$)

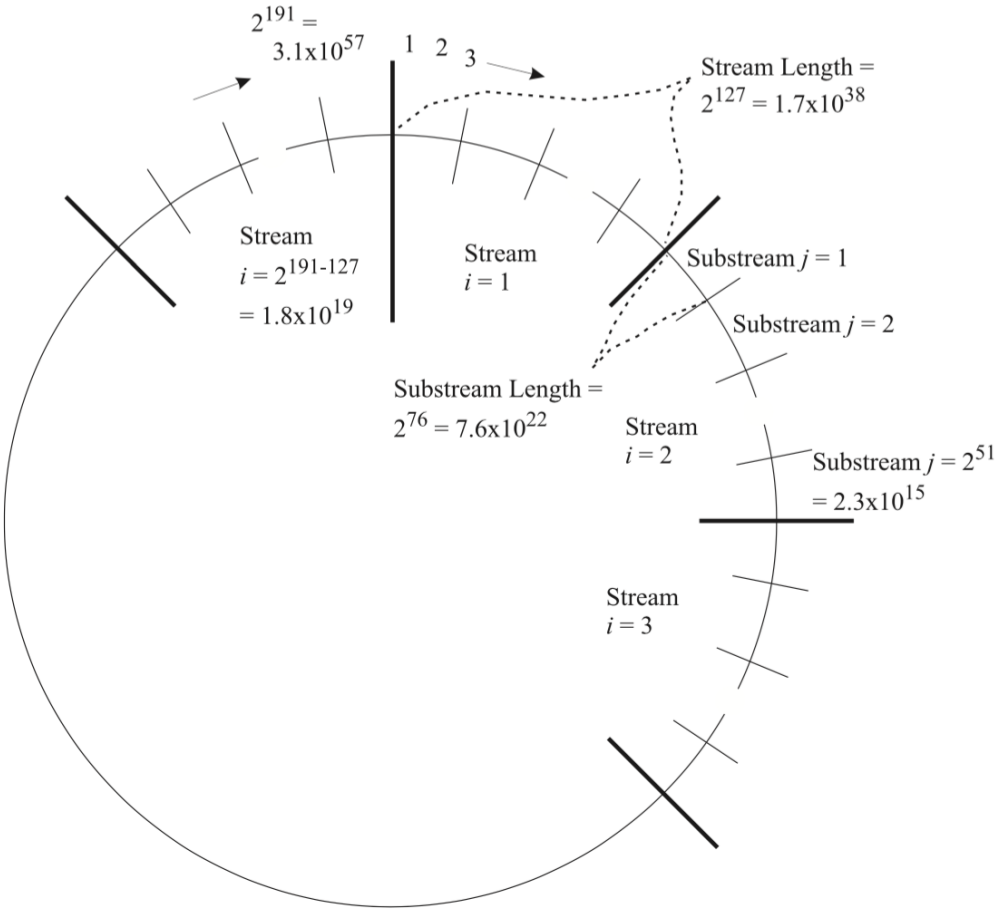


Figure: Number of Streams (2^{64}) and substreams (2^{51}) of MRG32k3a [3]

General Definition

- A finite set of states space: $\mathcal{S}\{i = 0, n : s_i\}$
- A key space: $\mathcal{K}\{k\}$ and an integer output multiplicity: $J \in \mathbb{Z}$
- A typical u.i.i.d output space: $\mathcal{U}\{u_{k,i}\} \in (0, 1)$

$$\begin{aligned} f &: \mathcal{S} \rightarrow \mathcal{S} \\ g &: \mathcal{S} \times \mathcal{K} \times \mathbb{Z}_J \rightarrow \mathcal{U} \end{aligned}$$

  - transition function f: $s_{i+1} = f(s_i)$ with a seed s_o
  - output function g: $u_{k,i} = g_{k,(i \bmod J)}(s_{\lfloor i/J \rfloor})$ for $i \geq 0$

$$g_{k,j} = h_j \circ b_k \tag{1}$$

  - h_j : a selector (simple)
  - b_k: a keyed bijection (complicated)
- A period $\rho \leq |\mathcal{S}| \leq 2^m$ where m is the bits represent the state

Combined MRGs with $\mathbb{F}_2$-Linear pRNG

- MRG32k3a [3](P. L'Ecuyer, 2002)
- State

$$
\begin{aligned}
s_{1,n} &= (x_{1,n}, x_{1,n+1}, x_{1,n+2}) \\
s_{2,n} &= (x_{2,n}, x_{2,n+1}, x_{2,n+2})
\end{aligned}
$$

- Linear recurrence

$$
\begin{aligned}
x_{1,n} &= [a_{12}x_{1,n-2} - a_{13}x_{1,n-3}] \bmod m_1 \\
x_{2,n} &= [a_{21}x_{1,n-1} - a_{23}x_{1,n-3}] \bmod m_2 \\
u_n &= [(x_{1,n} - x_{2,n}) \bmod m_1]/(m_1 + 1) \quad \text{if} \quad (z_n > 0)
\end{aligned}
$$

- period $\rho = (m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191} \approx 3.1 \times 10^{57}$

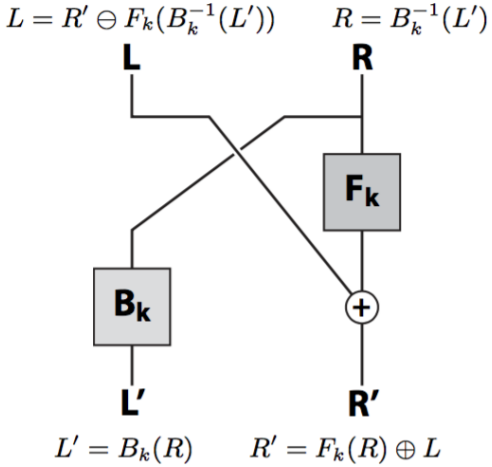
Advance Randomization System (ARS)

- Random123 [4] (J. Salmon *et al.*)
- Threefry: N rounds of AES with a user key (Weyl sequence)

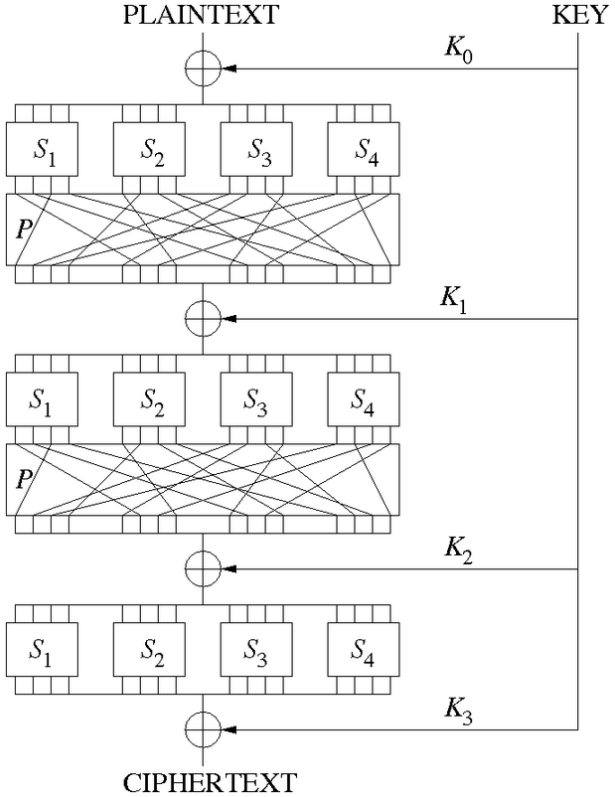
$$\begin{aligned} \text{round\_key}_0 &= \text{user\_key} \\ \text{round\_key}_i &= \text{round\_key}_{i-1} + c \end{aligned}$$

- Philox: iterated bijection with rounds of the Feistel function

$$\begin{aligned} L &= B_k(R) = \lfloor (R \times M)/2^W \\ R &= F_k(R) \oplus L = (R \times M) \bmod 2^W \oplus k \oplus L \end{aligned}$$



Rounds of Substitution Permutation (SP) Network



Overlap Probability of Streams with a Hash (Skip)

- The probability of non-overlap with a multi-stream RNG with a random seed
  - period: ρ
  - number of streams : s
  - length of stream: l

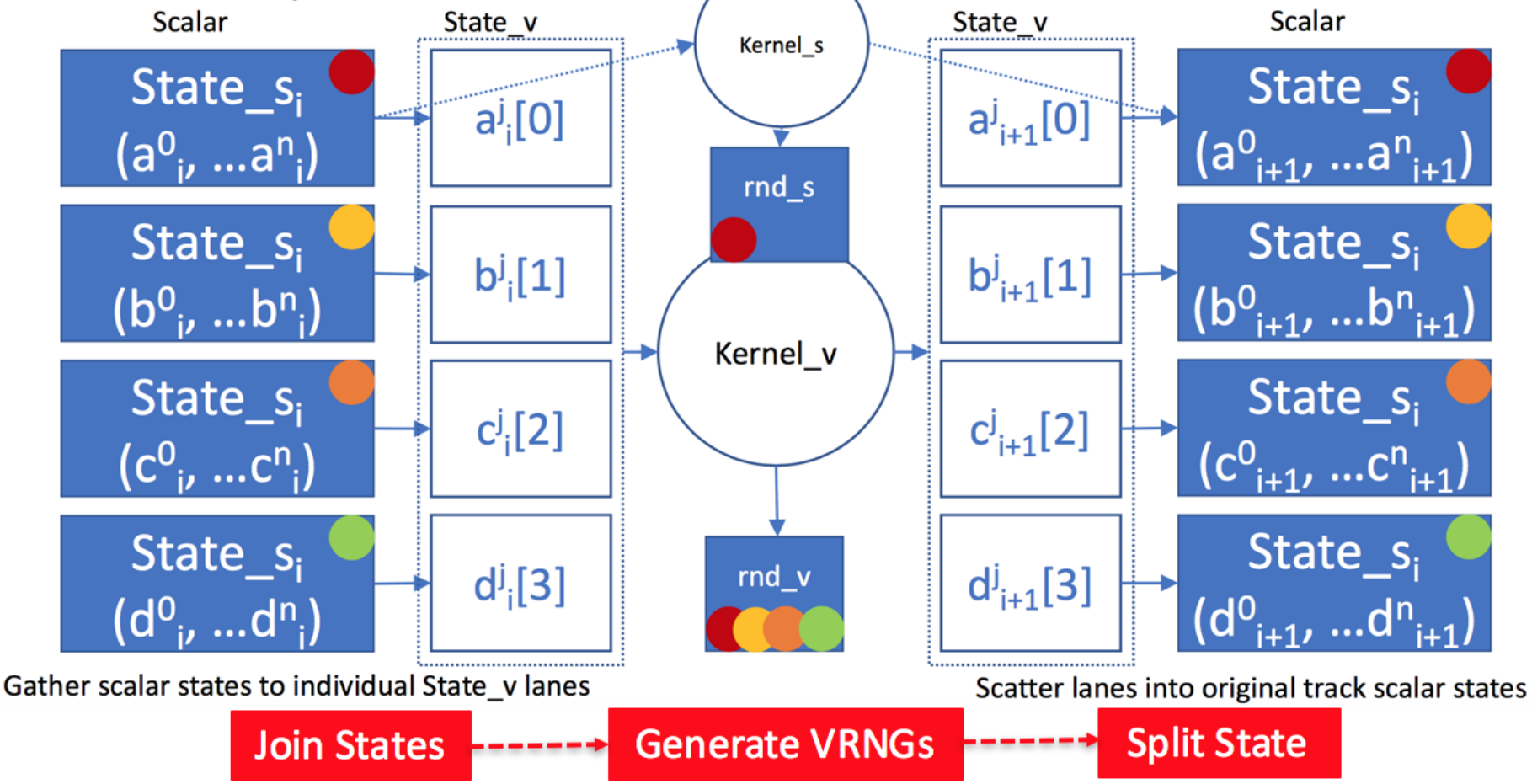
$$p = (1 - \frac{sl}{\rho})^{s-1} \tag{2}$$

- For small sl/ρ, the overlap probability $1 - p \approx s^2 l/\rho$
  - example: $1 - p = 2^{-68}$ for $s = l = 2^{20}, \rho = 2^{128}$
- Alternatives: a combination of pRNGs
  - select a seed for each stream by another pRNG (reproducibility)
  - tree of random streams (created dynamically)
- Overlap probability of multiple streams using the hash mechanism is nearly zero for typical HEP detector simulation.

Reproducibility: Proxy Approach

Vectorized multi-track approach (scalar pRNG states per track)

Diagram by Andrei Gheata



Reproducibility: Buffering Approach

Vectorized caching approach (vector pRNG states per track)

Diagram by Andrei Gheata