Parallel Random Number Generators: VecRNG

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

August, 31, 2018
Contents

- Introduction
- Choice of Generators and Status of Implementation
- Design Requirements and Special Features
- Preliminary Performance Results
- Probability Distribution Functions
- Reproducibility under Concurrent Tasks
- Remarks and Plans
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]

<table>
<thead>
<tr>
<th>Generator</th>
<th>Scalar State</th>
<th>Memory</th>
<th>Period</th>
<th>Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRG32k3a</td>
<td>6 doubles</td>
<td>48 bytes</td>
<td>$2^{191}$</td>
<td>$2^{64}$</td>
</tr>
<tr>
<td>Threfry(4x32)</td>
<td>12 32-bit int</td>
<td>48 bytes</td>
<td>$2^{256}$</td>
<td>$2^{128}$</td>
</tr>
<tr>
<td>Philox(4x32)</td>
<td>10 32-bit int</td>
<td>40 bytes</td>
<td>$2^{192}$</td>
<td>$2^{64}$</td>
</tr>
<tr>
<td>MIXMAX(N=17)</td>
<td>17 32-bit int</td>
<td>68 bytes</td>
<td>$10^{294}$</td>
<td>$\sim \infty$</td>
</tr>
</tbody>
</table>

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`

- **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

---

Soon Yung Jun (Fermilab) John Apostolakis (CERN) Parallel Random Number Generators: VecRNG
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]
  
  ```cpp
  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** $N(\text{=SIMD length})$ consecutive substreams and supports both vector and scalar return-type, for an example,

  ```cpp
  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

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

  ```cpp
  vecrng::cuda::MRG32k3a<ScalarBackend> rng();
  rng->Initialize(states, ntid); // states[ntid]
  double rs = rng->Uniform<ScalarBackend>(&states[tid]);
  ```
The average CPU time [ms] for generating 10M random numbers

std::rand() = (93.24 ± 0.03) ms

<table>
<thead>
<tr>
<th>Generator</th>
<th>MRG32k3a</th>
<th>Threetry</th>
<th>Philox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>137.24 ± 0.05</td>
<td>74.13 ± 0.08</td>
<td>54.59 ± 0.03</td>
</tr>
<tr>
<td>Vector (AVX)</td>
<td>57.59 ± 0.02</td>
<td>43.06 ± 0.17</td>
<td>76.90 ± 0.22</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.45 ± 0.05</td>
<td>12.12 ± 0.02</td>
<td>12.19 ± 0.01</td>
</tr>
<tr>
<td>Curand</td>
<td>0.51 ± 0.01</td>
<td>N/A</td>
<td>0.67 ± 0.05</td>
</tr>
</tbody>
</table>

- 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 Threetry 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)
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 $\sim 4$ 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)

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

$S_{gid, sid, rid}$
- $gid =$ generation ID
- $sid =$ secondary ID
- $rid =$ random number ID

$U_{id} = tid$
$S_{0,1}^1 \cdots S_{0,i-1}^1$

$U_{id} = \text{rng} \rightarrow \text{uint}(S_{1,i}^1)$
$S_{1,1}^1 \cdots S_{1,j-1}^1$

$U_{id} = \text{rng} \rightarrow \text{uint}(S_{0,i+m}^1)$
$S_{m,1}^2 \cdots S_{m,k-1}^2$

$U_{id} = \text{rng} \rightarrow \text{uint}(S_{1,j+n}^2)$
$S_{n,1}^3 \cdots S_{n,q-1}^3$

$U_{id} = \text{rng} \rightarrow \text{uint}(S_{0,i+m}^1)$
$S_{m,k}^2 \cdots S_{m,k+l-1}^2$

$n-$secondaries
$m-$secondaries
Reproducibility for Vector Tasks

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

```cpp
// 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 integer (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


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

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


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: 
  \[ s_{j,n+\nu} = (A_j^\nu \mod m_j)s_{j,n} \mod m_j \]

  (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: \( \mathcal{U}\{u_{k,i}\} \in (0, 1) \)

\[
\begin{align*}
  f & : \mathcal{S} \rightarrow \mathcal{S} \\
  g & : \mathcal{S} \times \mathcal{K} \times \mathbb{Z}_J \rightarrow \mathcal{U}
\end{align*}
\]

- transition function \( f: s_{i+1} = f(s_i) \) with a seed \( s_0 \)
- output function \( g: u_{k,i} = g_{k,(i \mod J)}(s_{\lfloor i/J \rfloor}) \) for \( i \geq 0 \)

\[
g_{k,j} = h_j \circ b_k
\]  

- \( h_j \): 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**

$$s_{1,n} = (x_{1,n}, x_{1,n+1}, x_{1,n+2})$$
$$s_{2,n} = (x_{2,n}, x_{2,n+1}, x_{2,n+2})$$

- **Linear recurrence**

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

- **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.*)
- **Threfry**: N rounds of AES with a user key (Weyl sequence)
  \[\text{round_key}_0 = \text{user_key}\]
  \[\text{round_key}_i = \text{round_key}_{i-1} + c\]
- **Philox**: iterated bijection with rounds of the Feistel function

\[
L = B_k(R) = \lfloor (R \times M) / 2^W \rfloor
\]
\[
R = F_k(R) \oplus L = (R \times M) \mod 2^W \oplus k \oplus L
\]

Rounds of Substitution Permutation (SP) Network

![SP Network Diagram](image)
The probability of non-overlap with a multi-stream RNG with a random seed

- period: $\rho$
- number of streams: $s$
- length of stream: $l$

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

For small $sl/\rho$, 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.
Vectorized multi-track approach (scalar pRNG states per track)

Gather scalar states to individual State_v lanes

Join States  Generate VRNGs  Split State

Diagram by Andrei Gheata

Reproducibility: Proxy Approach

Soon Yung Jun (Fermilab) John Apostolakis (CERN)  Parallel Random Number Generators: VecRNG
Reproducibility: Buffering Approach

Vectorized caching approach (vector pRNG states per track)

Cache 2 vector rnd values per track (4*2 scalar lanes)
Keep track of current delivered value in cache

Gather scalar value from cached lanes into rnd_s/rnd_v in case: \( i_{\text{crit}} \% 4 > 0 \)
Call vector kernel to advance the state and refill the cache in case: \( i_{\text{crit}} \% 4 = 0 \)

Diagram by Andrei Gheata

Store VRNGs → Consume RNGs → Refill Buffer