Random number generation for parallel Monte Carlo

Protocol of a temporary obsession

Jakob van Santen <jakob.van.santen@desy.de>
In a place, at a time

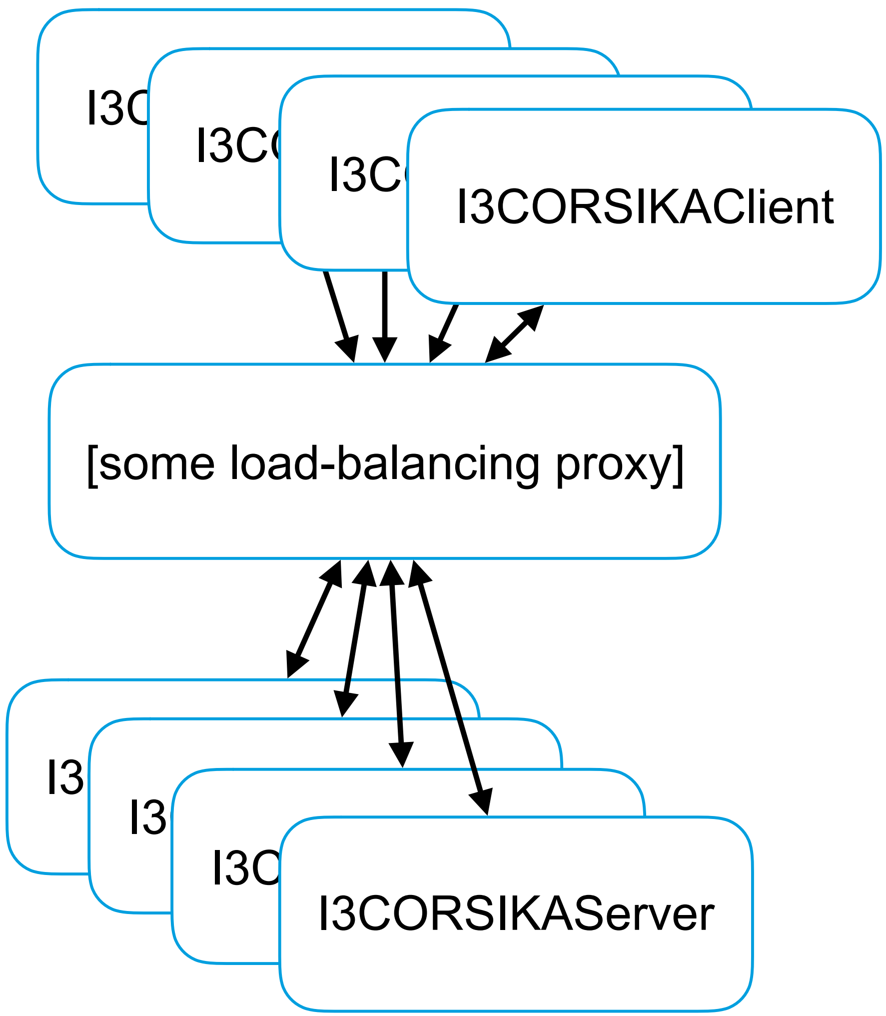
HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

DESY.

Motivating example

The best problems are the ones you create yourself

- Factoring CORSIKA out into a service allows flexible scaling
- **Problem**: CORSIKA's RNG is explicit internal state => result depends on which server handles the request.
- **Solution**: client maintains and communicates desired RNG state
- How to communicate and apply state without sacrificing quality or efficiency?
    a) Brute force, or
    b) clever math



Requirements for parallel random numbers

A hierarchy of needs

- A single pseudorandom sequence should have:
  - deterministic output
  - an extremely long period (2^{128} or more)
  - no autocorrelation
- Parallel pseudorandom sequences (streams) should be:

1. Disjoint and uncorrelated (provably, if possible)
2. Quickly partitionable into arbitrarily sized substreams
3. Independent of the degree of parallelization
4. Small (<< than 20kB state of MT19937)
5. Fast (random numbers should be cheaper than the calculation they feed)

Partitioning strategies

1. Use a single generator with different initial state (seed) for each stream and hope for the best
   - Disjoint and uncorrelated: **maybe**
   - Paritionable: **no**
   - Independent of parallelization: **no**
2. Use the same seed, but different parameter sets
   - Disjoint and uncorrelated: **yes**
   - Partitionable: **maybe** (partitioning strategy has to be fixed at the outset)
   - Independent of parallelization: **maybe** (given a sufficiently large number of parameter sets)

Parameterized RNGs: SPRNG

Scalable Parallel Random Number Generator (sprng.org)

- GPL v2 license
- C++/FORTRAN bindings (custom interface), 3rd-party CUDA implementation exists
- Creates independent “streams” of random numbers
- Independence of streams theoretically proven (for some generators)
- Default generator (lagged Fibonacci) has 2^{39648} independent streams, each with period 2^{1310}
- Streams partitioned in a tree with fixed but user-specified arity. Example: with 64 streams, each root generator can spawn 64 substreams, each substream can spawn 64 substreams of its own, etc.
- Pitfalls:
  - **It is possible to exhaust the parameter space** if you try hard enough.
  - **Initializing a full-period RNG is expensive** (O(ms), equivalent to ~2e5 random numbers).

Bad example: Multiply-with-carry RNG

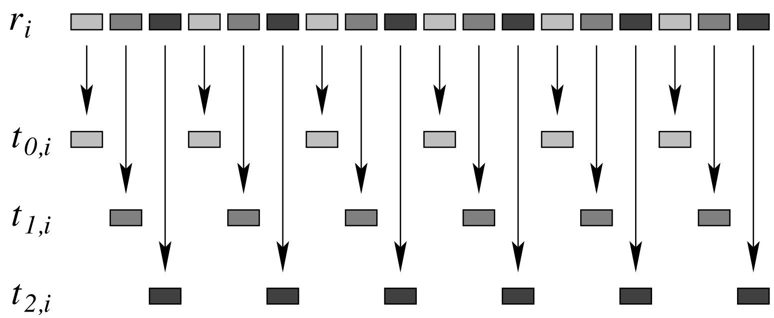
as used in MCML (atomic.physics.lu.se), clsim

- Lag-1 MWC generator with period ~2^{60}, different prime multipliers lead to independent streams
- Good:
  - Very fast (3 floating-point operations per call)
  - Very small (8-byte state fits comfortably in GPU local memory)
- Bad:
  - Number of independent streams limited to number of prime multipliers generated prior to run **(not arbitrarily partitionable)**
  - RNG is attached to a thread rather than work item, so result depends on (nondeterministic) mapping of work items to threads **(result depends on parallelization)**

Partitioning strategies (continued)

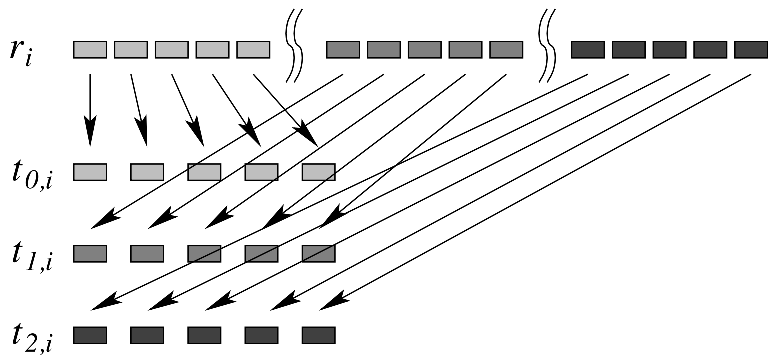
Leapfrog

- Disjoint: yes
- Uncorrelated: maybe
- Independent of parallelization: no
- Quickly partitionable: maybe
  (requires efficient fast-forward by N)



Block split

- Disjoint and uncorrelated: yes
- Independent of parallelization: yes
- Quickly partitionable: maybe
  (requires efficient fast-forward by block size)



[Mertens (2009)]

Fast-forwarding a random number generator

RNGs produce a recurrent sequence, i.e. the next state depends on the previous N

$$r_i = f(r_{i-1}, r_{i-2}, \ldots, r_{i-n}),$$

Fast-forwarding through M positions by applying f() M times. If f is a linear function, this can be written as an M iterations of the matrix multiplication

$$\begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ a_n & a_{n-1} & \ldots & a_1 \end{pmatrix}}_{A} \begin{pmatrix} r_{i-n} \\ \vdots \\ r_{i-2} \\ r_{i-1} \end{pmatrix} \mod p$$

and can be computed in $O(n^3 \log M)$ time rather than $O(n^3 M)$. Since all finite or periodic sequences over a finite field can be generated by a linear recurrence, this is **always possible in principle,** but only practical for explicitly linear RNGs (linear congruential, general linear feedback shift registers, YARNs).

[Mertens (2009)]

Block-splitting/leapfrogging RNGs: TRNG

Tina's Random Number Generator (numbercrunch.de/trng)

- 3-clause BSD license
- Passes full suite of empirical tests in TestU01
- C++11 random_number_engine and CUDA bindings
- Some engines with efficient split and skip operations
- Partitioning left to the user

Counter-based RNGs

- An RNG is built out of two functions:

$$f : S \to S \qquad \text{(state transition function)}$$

$$g : S \to U \qquad \text{(output function)}$$

- Conventional RNGs have a complicated f() that produces integers over some range, and a simple g() that scales those integers to [0,1).
- Counter-based RNGs make f() simple (a counter!) and a g() that
  - Maps arbitrarily sequences of integers onto another set whose distribution is indistinguishable from noise
  - Is reasonable fast to evaluate
  - => g() has the same properties as a good cryptographic block cypher!

[Salmon et al (2011)]

Counter-based RNGs: Random123

“Random numbers: as easy as 1, 2, 3” (deshawresearch.com)

- 3-clause BSD license
- Passes full suite of empirical tests in TestU01
- C, C++11 random_number_engine, CUDA bindings
- Faster than MT19937 on CPUs with AES-NI support
- 2^{64} possible streams, each with 2^{128} period
- Skip and split operations naturally supported, and practically free
- Partitioning left to the user

Summary

- Massively parallel random number generation is a common problem, and there are known solutions.
- In all cases, random number generation should be deterministic and independent of granularity of parallelism, execution order, etc.
  - Attach RNG stream/block to particle (or whatever other atomic unit you have in your simulation)
  - Ensure that the conditions for creating a new stream/block are deterministic
- The implementation depends on the characteristics of the simulation
  - For explicit parallelism with rare, predictable branching and no restrictions on local memory: use SPRNG streams
  - For implicit (dynamically load-balanced) parallelism, or with unpredictable workloads, assign a (dynamically sized) block to each work item
    - TRNG: fast-forward blocks in logarithmic time
    - Random123: fast-forward blocks in constant time

Further reading

Gao, S., & Peterson, G. D. (2013). GASPRNG: GPU accelerated scalable parallel random number generator library. *Computer Physics Communications*, *184*(4), 1241–1249. http://doi.org/10.1016/j.cpc.2012.12.001

Katzgraber, H. G. (2010). Random Numbers in Scientific Computing: An Introduction. Presented at the International Summer School Modern Computational Science.

L'Ecuyer, P., & Simard, R. (2007). TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Trans. Math. Softw.*, *33*(4), 22:1–22:40. http://doi.org/10.1145/1268776.1268777

Manssen, M., Weigel, M., & Hartmann, A. K. (2012). Random number generators for massively parallel simulations on GPU. *The European Physical Journal Special Topics*, *210*(1), 53–71. http://doi.org/10.1140/epjst/e2012-01637-8

Mascagni, M., & Srinivasan, A. (2000). Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. *ACM Trans. Math. Softw.*, *26*(3), 436–461. http://doi.org/10.1145/358407.358427

Mertens, S. (2009). Random Number Generators: A Survival Guide for Large Scale Simulations. Presented at the International Summer School Modern Computational Science.

Salmon, J. K., Moraes, M. A., Dror, R. O., & Shaw, D. E. (2011). Parallel Random Numbers: As Easy As 1, 2, 3 (pp. 16:1–16:12). Presented at the Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, New York, NY, USA: ACM. http://doi.org/10.1145/2063384.2063405