Deterministic transport independent of the order of tracking

D. Savin, J. Apostolakis, S. Wenzel

/

Motivation

- Monte Carlo simulation relies on pseudo-random numbers
- The numbers must seem uncorrelated
- Reproducibility from any point of simulation is needed

Motivation
Geant4

- Geant4 uses event-level parallelism
  - Each event is simulated independently
  - Independent PRNGs in a small fixed number of threads
    - an event is defined by the generator state at the beginning

Motivation
GeantV

- GeantV uses track-level parallelism
  - Similar tracks from many events are collected in ”baskets”
  - Baskets are processed in parallel to utilize vectorization
  - Number of tracks, baskets and threads change during the simulation
    - the generator state must be associated with the track

Pedigrees

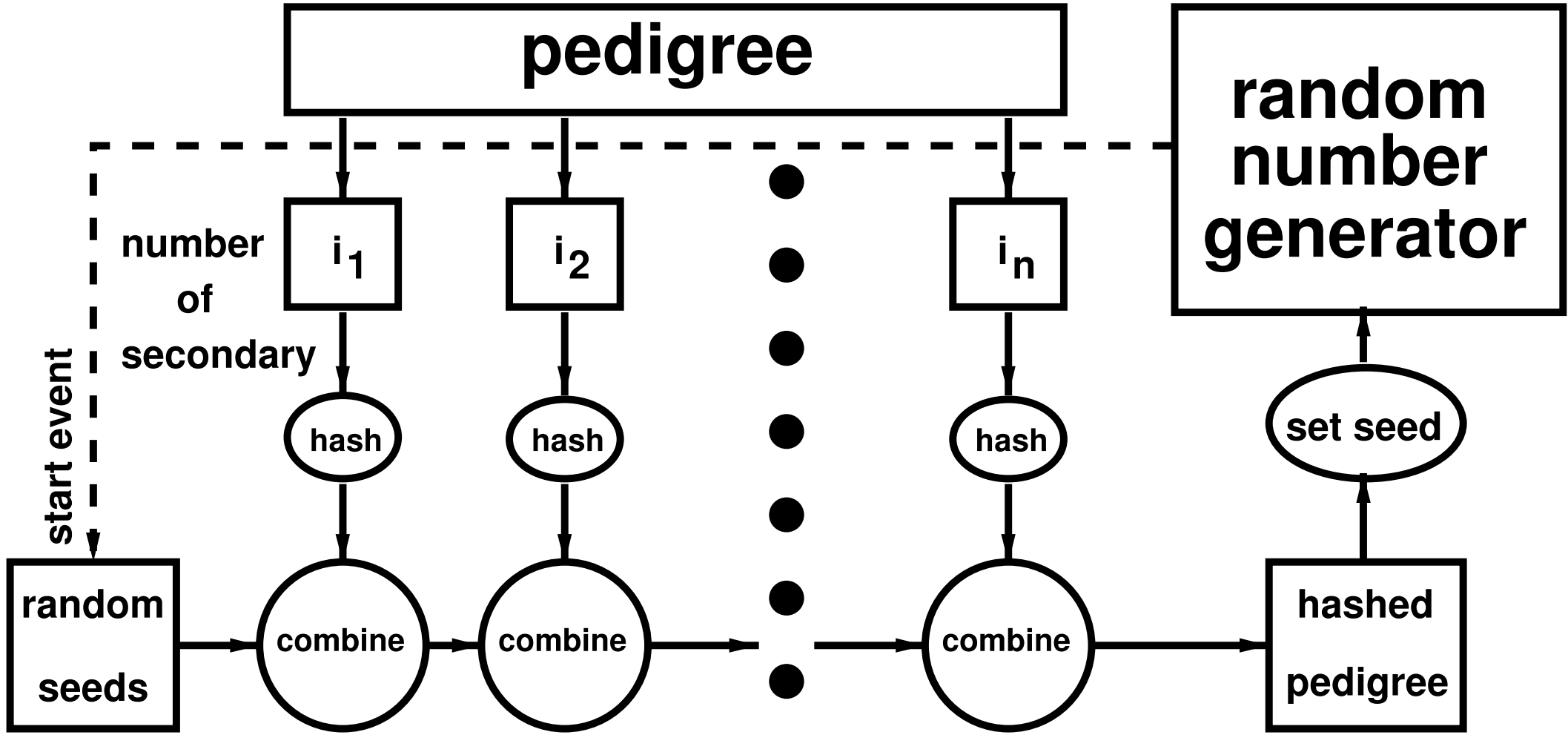
- Define:
  - the rank of a track i_k - the number of sibling tracks created by its parent track before it
  - the pedigree of a track $\{i_1, i_2, \ldots, i_n\}$ - the sequence of ranks of its ancestors
- the hashed pedigree can be stored by the track and used to seed the generator

Merkle–Damgård hash calculation

- Only the hashed pedigree of the parent is available when spawning secondaries

  => the hashing algorithm is a Merkle-Damgård construction (like MD5, SHA2…)
- Collision rate of the Merkle-Damgård hash function is the same as of the compression function – any standard one should suffice
- The initialization vector is set at the beginning at the event

Merkle–Damgård hash calculation



Reproducibility test

- Stacking Action that can place a track to the waiting stack with 1/2 probability using std::random_device
- Stacking and seeding are controlled by macro commands
- Output histograms compared by Chi2Test
  - p-value for equal histograms is 1
  - p-value for different histograms is < 1

Reproducibility test
Results

- The default stacking
  - the same initial seeds: p = 1,
  - different initial seeds: p < 1

    => sensitive to the random number generation
- The random stacking
  - default seeding: p < 1

    => sensitive to processing order
  - pedigree-based seeding: p = 1

    => independent from processing order

Random number engines

- Ranecu
- Ranlux
- Ranlux64
- HepJamesRandom
- Ranshi
- DualRand
- Mersenne Twister
- MixMax
- CBPRNG

Counter-Based Pseudo-Random Number Generators

- State transition is a simple increment
- Randomness produced by the output function
- Seeding is a single assignment
- Cryptographic function for robustness, simpler functions for performance
- Vectorization may increase performance
- Philox1x64, Threefry1x64 from Random123 library were wrapped in a CLHEP interface
- Another implementation by Soon Yung Jun is in progress

MixMax

- Passes the BigCrush test suite
- Has several implementations with different state size
- In Geant4 10.04b the state size is 1216 bits
- Several seeding mechanisms
- Substreaming capabilities
- Actively developed by Kostas Savvidis

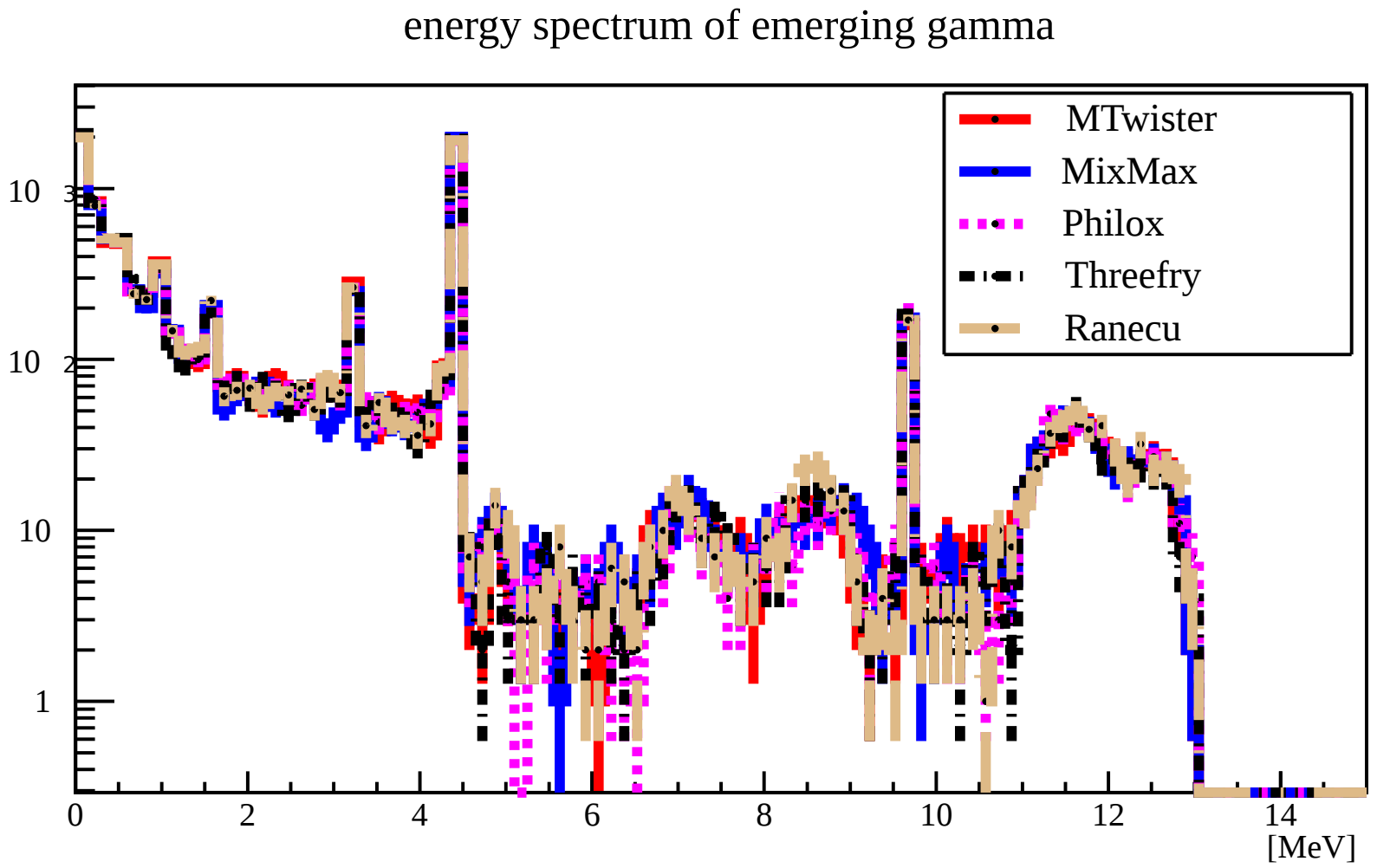
Traditional CLHEP engines

- Ranecu, Ranlux, HepJamesRandom, Ranshi, DualRand – legacy engines kept for compatibility and should not be used
  - Not indicated in the manual!
- Ranlux64 is good, but rather slow
- Mersenne Twister is the currently recommended Geant4 engine
  - passes most of the tests (not all)
  - has a big state (19937 bits)

Engines testing
Setup

- Hadr06 and graphite.mac with 10x events
- Incident neutrons require many random numbers for thermal nucleus sampling
- 14 MeV neutrons produce several secondaries in inelastic interactions
- Well-known 4.4 MeV peak in the outgoing gamma spectrum

Engines testing
Physics results

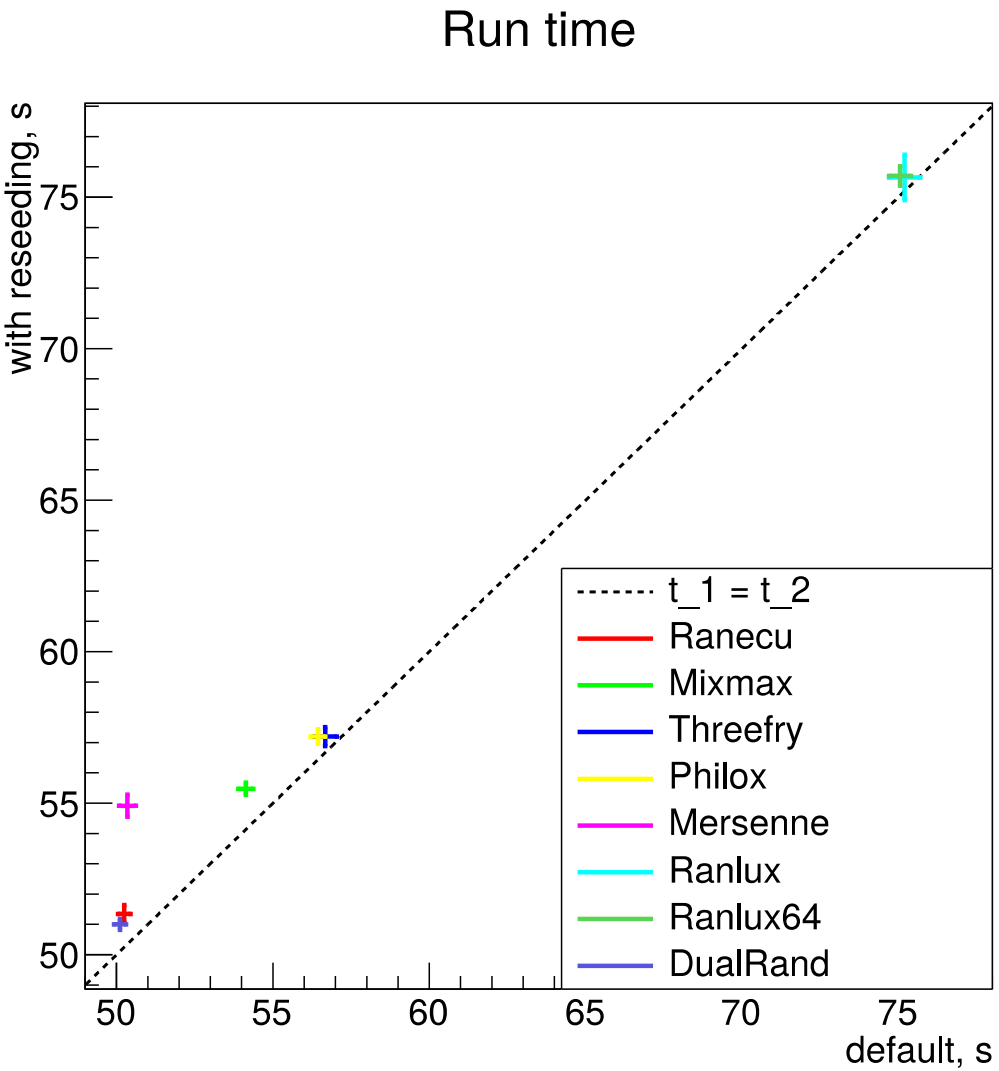


/

Engines testing
Benchmark details

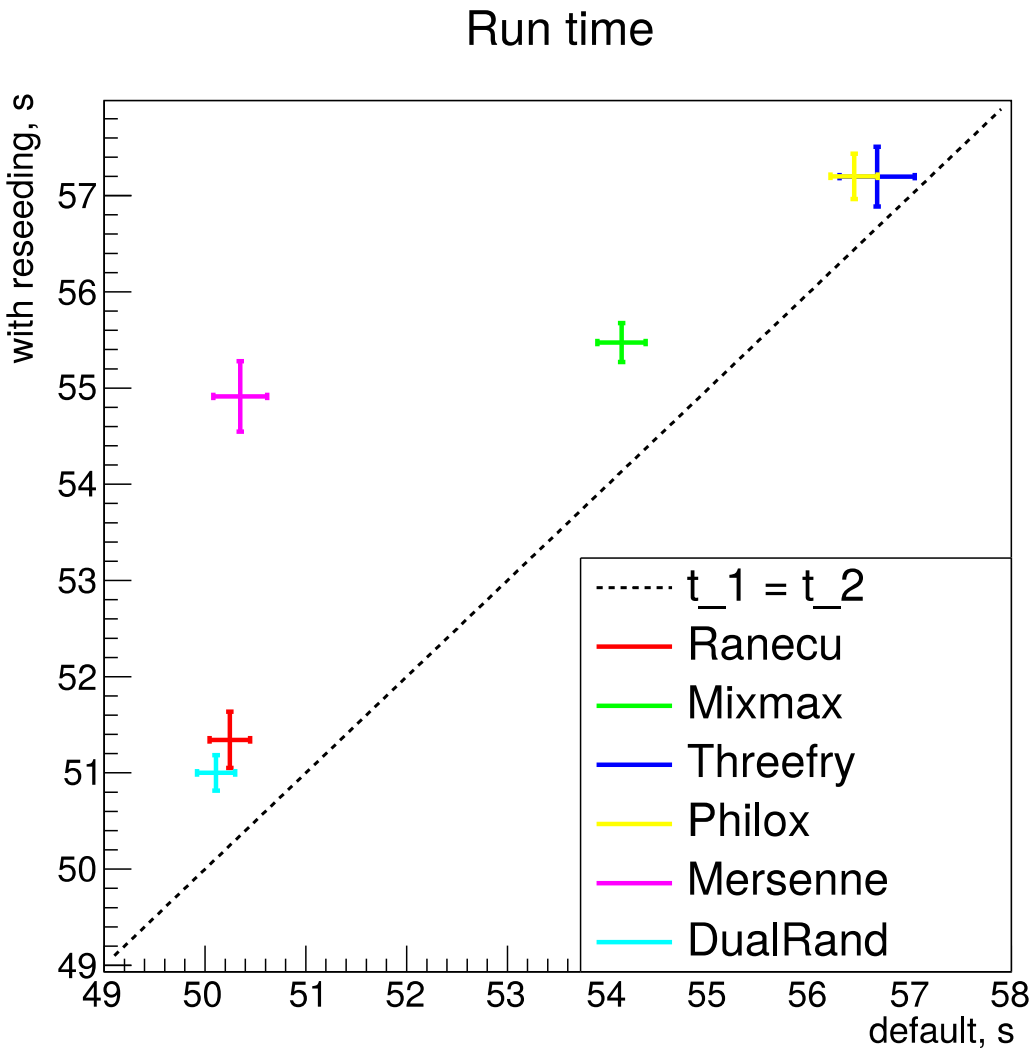
- An isolate core of Core i7-3770 CPU
- Intel compiler 2017
- RelWIthDebInfo build type
- Seeding at the beginning of each event is on
  - Lowers MixMax performance
- One wrapper for all CBPRNG
  - Lowers performance
- The results for the overhead are still valid

Engines testing
Performance results



/

Engines testing
Performance results



/

Conclusion

- Implemented a Geant4 prototype which can perform fully reproducible simulations in a multithreaded environment and under the exchange of the order of track propagation.
- Profiled the overhead to achieve this reproducibility for different random number engines and showed that it is quite low for selected engines.
- **It is worth to apply a similar algorithm for reproducible pseudo-random number generation in GeantV**

Acknowledgements

I would like to thank
John Apostolakis and Sandro Wenzel
for productive collaboration

The work was supported by
Google Summer of Code 2017

Links

- Branch ”pedigree” in Geant4 clone repository
  - https://bitbucket.org/sd57/geant4/branch/pedigree
- GSoC final report
  - https://sd57.github.io/g4dprng/index.html
- Detailed description (in progress)
  - https://sd57.github.io/g4dprng/gsocPreprint.html