Paper Number:	P1932R0
Title:	Extension of the C++ random number generators
Authors:	Pavel Dyakov <pavel.dyakov@intel.com> Ilya Burylov <ilya.burylov@intel.com> Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com> Andrey Nikolaev <Andrey.Nikolaev@intel.com>
Audience:	SG6 (Numerics)
Date:	2019-10-07

I. Introduction

C++11 introduced a comprehensive mechanism to manage generation of random numbers in the <random> header file (including distributions, pseudo random and non-deterministic generators).

Since C++ 11 standard, vendors hardware (HW) has been significantly improved (e.g. 256-bits/512-bits Advanced Vector Extensions instruction sets were introduced on CPU) and many studies and researches has been published in the random number generation science (e.g.: [1], [2], etc.). The list of C++ random number generators has not been updated since 2011 year, and modern generators broadly used in different applications and benefit from HW advances are not covered by the standard.

The request to review the current existing list of engines was raised during the discussion of the P1068R2 paper. Contents of the paper are independent and can be considered separately.

We propose to extend the list of the random number generators in C++ standard and seek for feedback on the list of proposed engines.

II. Revision history

- N/A

III. Motivation and Scope

Random number generators (engines) are at the heart of Monte Carlo simulations [3] used in many applications such as physics simulations, finance, statistical sampling, cryptography, noise generation and others.

Random number generators are characterized by the different aspects:

- **Quality of the produced sequences**. Quality of the random numbers (randomness of the produced values) is analyzed theoretically and checked by the different test suites. For example, TestU01 with BigCrash battery [5] that includes about 160 statistical tests is one of the most popular test systems for random number generators.
- **Generation speed and sequence period**. Random number generation is a bottleneck in many use cases, in particular in the large-scale Monte Carlo simulations ([4]) that require large amount of the random number are produced. Therefore, period of the random number sequence and generation speed are the important aspects for the random number generators.
- **Vectorization and parallelization opportunities.** Performant and statistically correct support for parallel Monte Carlo simulations is one of the main requirements to the random number generators when random sequences should be generated independently on the different computing nodes. Support for vectorization is another requirement to the random number generators that allows using HW capabilities and impact on the generation speed.

Each of the C++11 random number generators has own advantages and disadvantages in terms of described criteria, e.g. linear congruential generators, the simplest generators with 32-bit state, have a

quite short generation period (2^32) and weak statistical properties while Mersenne Twister 19937 generator has long generation period and strong statistical properties relying a big vector state underneath; in its turn, this state impacts on the effective support of parallel Monte Carlo simulations. C++ random number generators do not support additional use cases such as quasi Monte Carlo simulations.

This paper formulates a proposal on extensions of the C++ random number generators that enable additional use cases, own good statistical properties and actively exploit HW advances. The base for this proposal – analysis of the latest results in random number generation area.

Extension and/or modification of the list of supported distribution random number generators is out of the scope of this proposal.

IV. Libraries and other languages

Random number generators are presented in many libraries. Examples:

* Intel(R) Math Kernel Library (Intel® MKL)

 - 9 pseudo-random 2 quasi-random and 1 non-deterministic generators with C[6]/Fortran APIs

* MathWorks

 - 9 pseudo-random and 2 pseudo-random generators with MatLab APIs

* NVIDIA* cuRAND

 - 5 pseudo-random 1 quasi-random (4 different versions) with Cuda interfaces

Intel MKL is the example of the collection of engines which are highly vectorized for modern CPUs. Additionally, most of Intel MKL engines (10 of 12) effectively support parallel random number generation via blocking techniques or family of generators.

V. Problem description

C++11 introduced support for the following engine types:

- **linear_congruential.** This class is a common implementation of the linear congruential generators. This generator type used to be broadly used in the past but due to not good statistical properties (56 tests are failed in TestU01 BigCrash) and low period (~ 2^31) it has many competitors among the other generators.

- **mersenne_twister.** This class is an implementation of Mersenne Twister 19937 32- and 64-bits generators. These generators have good statistical properties – passed TestU01 BigCrash (other sources – two tests are failed conditionally) and long period (~ 2^19937). However, Mersenne Twister 19937 has big vector state (624 x 32 bits) so that it cannot be supported on the different device: e.g., NVIDIA Cuda does not support device API for this generator (Host API only). In addition, this generator is difficult to parallelize – big computation overhead is need for state adjustment.

- **subtract_with_carry.** Ranlux48, ranlux48_base, ranlux24, ranlux24_base generators can be included in this group. These generators have good statistical properties but generation speed is significantly lower in comparison for example with Mersenne Twister family.

- **random_device.** Non-deterministic generator type is also known as true random generator - in C++ standard it is presented as random_device. This generator is HW dependent.

One observes significant advances in the random number generation area since the C++ 11 standard was established:

- new efficient generators (counter-based family) have been introduced
- a lot of research connected to the statistical properties of the generators have been published
- quasi-random number generation is a de-factor standard instrument for simulations in various areas such as finance.

Considering those aspects is important for C++ standard to extend and provide up-to-date high quality performant random number generators used for solution of modern challenging problems in various areas.

VI. Possible approaches to address the problem

We completed analysis of the broadly used random number generators and defined the following groups:

- **Counter based generators.** At the first time, counter based generators were introduced in [7]. All counter-based generators have small state (e.g. Philox4x32-10 has 4 x 32-bits counter and 2 x 32-bits keys) and quite long period (not less than 2^130). The generators have quite a small internal state and effectively support parallel simulations vi block-splitting techniques and are enable a broad HW spectrum including CPU/GPU/FPGA/etc.

- **Mersenne Twister generators.** Mersenne Twister family includes wide set of the generators: Mersenne Twister 19937 (implemented in C++ standard), Mersenne Twister 2203 (version of the generator with 69 x 32-bits state – easily parallelized on CPU), MTGP (special version for GPU devices), etc. All of these versions have pros and cons in terms of state size, generation speed and statistical properties.

- **Recursive/congruential generators.** This group contains wide set of generators: including the simplest one (linear congruential) and different extensions and modifications. This group also includes Multiple Recursive generators (introduced by Pierre L'Ecuyer in [8]) – generators family with good statistical properties (all tests from TestU01 BigCrash are passed) and short state. Now Multiple Recursive generators are broadly used in Monte Carlo benchmarks.

- **Xor/shift generators.** Generators in this group are mostly based on XOR and SHIFT operations: generalized/linear feedback shift register and Xorwow generators. Linear feedback shift register generators can be easily implemented in HW – so that they are used a lot in deferent world standard, e.g. CDMA, USB 3.0, IEEE 802.11a, GPS/GLONASS, etc.

- **Quasi-random number generators.** Those generators produce highly uniform predefined sequence of numbers with low discrepancy. Main use case of such generators is quasi-Monte Carlo simulations. Sobol, Niederreiter, Faure are examples of the quasi-random number generators.

We propose the following criteria to be used as the base for recommended extensions of the list of C++ random number generators:

- **Usage scenarios.** Generators shall be broadly used in Monte Carlo simulations, associated with industry standards, etc.
- **Statistical properties.** Generators shall demonstrate good statistical properties confirmed by theoretical studies and empirical testing.*

- **HW friend-ness**. Generators shall enable the latest advances available in the modern HW (vector registers, cache sizes, number of processing elements etc)

**TestU01 BigCrash battery can be used as well for the generators differentiation in terms of statistical properties.*

Based on the research and mentioned criteria we recommend the following generators for adding to C++ standard:

a) Counter based generators – Philox/Threefry

Philox and Threefry are counter based pseudo random number generators. Philox generator uses multiplication instructions that compute the high and low halves of the product of word-sized operands, Threefry generator is based on a symmetric-key tweakable block cipher algorithm - Threefish.

First time these generators were introduced in [7] and now they are integrated to the many RNG libraries (Intel MKL, rocRand, cuRand, MatLab, tensorflow, etc.). Also based on [7] Threefry and Philox pseudo random number generators are the fastest generators on CPU and GPU respectively.

Counter based generators are commonly used in financial sphere (e.g. [9]) and can effectively support different devices (CPU/GPU/FPGA/etc.).

b) Multiple recursive generator (MRG)

MRG generator meets the requirements for the modern RNGs, such as a good multidimensional uniformity and a long period [10]. This generator is statistically more robust than linear congruential generators and successfully passes all tests in TestU01 BigCrash battery.

Due to long period (2^191), good opportunities for vectorization and parallelization this generator is commonly used in random number benchmarks (e.g. [11]) and implemented in many libraries (Intel MKL, NAG, GSL, R, rocRand, cuRand, MatLab).

c) Linear feedback shift register

Tauss88, LFSR113 and LFSR258 are the main representors of the linear feedback shift register generators. These generators are commonly used in biophysics sphere and in world technical standards, like CDMA, USB 3.0, IEEE 802.11a, GPS/GLONASS, etc. because they can be easily implemented in HW.

Tauss88, LFSR113 and LFSR258 have long period (2^88/2^113/2^258 respectively), but fail some tests in TestU01 BigCrash battery.

d) Quasi random number generator – Sobol

Quasi-random number generators are used in mathematical finance simulations as a low discrepancy sequence, especially in pricing and risk management with quasi Monte Carlo simulations (e.g. [12]), where pseudo random number generators cannot be used.

Sobol generators outperform all the generators in both speed and accuracy. Despite of long generator state this generator is supported in many RNG libraries (Intel MKL, NAG, rocRand, cuRand, BRODA, MatLab, etc.) and can be quite easily vectorized and parallelized.

VII. Design considerations

All proposed generators can be introduced as template classes with several instantiations for the particular generators.

a) Counter based generators – Philox/Threefry

In C++, standard Philox can be easily introduced by the generators family: Philox4x32-10, Philox2x64-20, Threefry4x64-20, etc. can be supported here.

b) Multiple recursive generator (MRG)

Broadly used MRG generators can be supported throw the one template class: Mrg32k3a, Mrg31k3, etc.

c) Linear feedback shift register

Taus88, LFSR113 and LFSR258 have different number of elements in states. All these generators can be supported by using the template class with the initialization lists as parameters.

d) Quasi random number generator – Sobol

Sobol generator is highly dependent on the direction numbers, so that several instantiations can be supported: sobol_joe_kuo ([13]), sobol_bratley_fox ([14]), etc.

VIII. Impact on the Standard

This is a library-only extension. It adds new engines classes but does not change any existing functions, nor enforce adding data members or virtual functions. It can, therefore, be ABI compatible with existing implementations.

IX. Summary of changes

New random number generators are proposed to be added to the C++ standard:

- Counter based generators – Philox/Threefry
  - Philox4x32-10
  - Philox2x64-20
  - Threefry4x64-20
  - etc.
- Multiple recursive generator
  - Mrg32k3a
  - Mrg31k3p
  - etc.
- Linear feedback shift register
  - Taus88
  - LFSR113
  - LFSR258
- Quasi random number generator – Sobol
  - sobol_joe_kuo
  - sobol_bratley_fox
  - etc.

The future revision of this proposal will include exact sections, details and APIs for the proposed random number generators. Existing library components do not depend on the proposed change, only new APIs added.

X. References

1. Bhattacharjee, Kamalika & Maity, Krishnendu & Das, Sukanta. (2018). A Search for Good Pseudo-random Number Generators: Survey and Empirical Studies.
2. L.Yu. Barash, L.N. Shchur, PRAND: GPU accelerated parallel random number generation library: Using most reliable algorithms and applying parallelism of modern GPUs and CPUs, submitted to Comp. Phys. Commun. (2013).
3. Sparapani, Rodney. (2004). Random Number Generation and Monte Carlo Methods (Second Edition). Journal of Statistical Software. 11. 10.18637/jss.v011.b08.

4. Prokhorenko, Sergei & Kalke, Kruz & Nahas, Yousra & Bellaiche, Laurent. (2018). Large scale hybrid Monte Carlo simulations for structure and property prediction. npj Computational Materials. 4. 10.1038/s41524-018-0137-0.
5. L'Ecuyer, Pierre & Simard, Richard. (2007). TestU01. ACM Transactions on Mathematical Software. 33. 22-es. 10.1145/1268776.1268777.
6. Intel MKL documentation: https://software.intel.com/en-us/mkl-developer-reference-c-2019-beta-basic-generators
7. John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 16:1–16:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0
8. L'Ecuyer, Pierre. (1996). Combined Multiple Recursive Random Number Generators. Operations Research. 44. 10.1287/opre.44.5.816.
9. Xu, Linlin & Ökten, Giray. (2014). High Performance Financial Simulation Using Randomized Quasi-Monte Carlo Methods. Quantitative Finance. 15. 10.1080/14697688.2015.1032549.
10. Fischer, Gregory & Carmon, Ziv & Zauberman, Gal & L'Ecuyer, Pierre. (1999). Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. Operations Research. 47. 159-164. 10.1287/opre.47.1.159.
11. Pagès, Gilles & Wilbertz, Benedikt. (2012). GPGPUs in computational finance: Massive parallel computing for American style options. Concurrency and Computation: Practice and Experience. 24. 10.1002/cpe.1774.
12. Xu, Linlin & Ökten, Giray. (2014). High Performance Financial Simulation Using Randomized Quasi-Monte Carlo Methods. Quantitative Finance. 15. 10.1080/14697688.2015.1032549.
13. S. Joe and F. Y. Kuo, Remark on Algorithm 659: Implementing Sobol's quasirandom sequence generator, ACM Trans. Math. Softw. 29, 49-57 (2003)
14. Bratley, P. and Fox, B. L. (1988), "Algorithm 659: Implementing Sobol's quasirandom sequence generator". ACM Trans. Math. Software 14: 88–100.

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2019, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804