Paper Number: P2075R0
Title: Philox as an extension of the C++ RNG engines
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: 2020-01-13

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 engines).

We proposed a set of engine candidates for the C++ standard extension in P1932R0 paper [1]. Current paper is focused on the family of the counter-based Philox engines.

II. Motivation

See P1932R0 [1] for motivation.

III. General Description

Philox engine is one of the counter-based engines which were introduced in 2011 in [2] for the first time. All counter-based engines have a small state (e.g. Philox4x32-10 has 6 x 32-bits elements in state) and long period (e.g. period of Philox4x32-10 is 2^130). This family effectively supports parallel simulations via block-splitting techniques and enable a broad HW spectrum including CPU/GPU/FPGA/etc.

Philox engine was chosen as an extension of the list of C++ random number engines based on the following (criteria proposed in P1932R0 [1]):

- **Statistical properties.** Authors of the counter-based engines took crypto-algorithm as the reference for Philox and claimed that Philox family passes rigorous statistical tests including TestU01's BigCrush [2]. This statement was independently verified by the different authors, e.g.: TestU01 batteries for Philox4x32-10 and Philox4x32-7 were tested in [4], DieHard testing results for Philox4x32-10 were published as part of Intel® Math Kernel Library (Intel® MKL) documentation in [5].
- **Usage scenarios.** Philox is broadly used in Monte-Carlo simulations which require massively parallel random number generation (e.g. Philox in financial simulations [6], high-quality pseudo-random behavior simulation [7], etc.).
- **HW friend-ness.** Philox engine can be easily vectorized and parallelized on CPU, for example Intel® MKL provides highly vectorized version of Philox4x32-10. Philox is proven to work on GPU – it's implemented in the GPU-optimized Nvidia and AMD libraries: cuRand and rocRand.

IV. Algorithm Details

Detailed description of the Philox engine can be found in [2].
Philox (Philox-***n*** x ***w*** - ***r***) engine relies on substitution-permutation network (SP-network). SP-network consists of S-boxes and P-boxes responsible for producing highly diffusive bijection and permutations respectively. A state of the Philox contains ***n*** words of size ***w*** and ***n*/2** keys which are used to produce round-keys for each of the **r**-rounds (see Figure 1 for 1-round illustration).

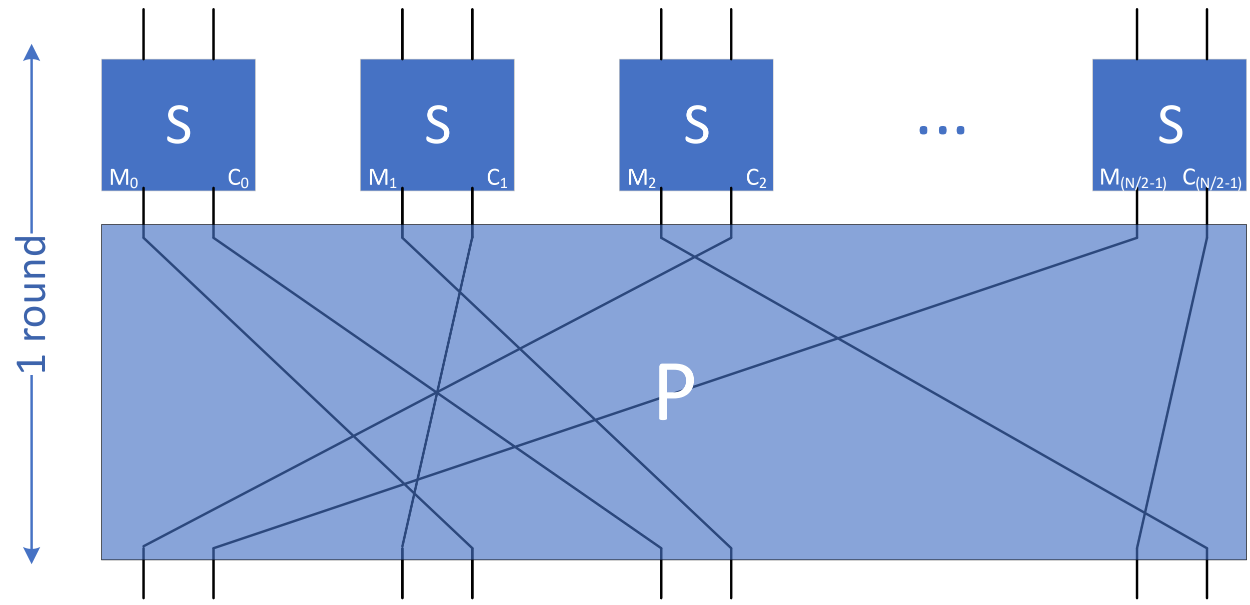

Figure 1. 1 round of SP-network

Each S-box has 2 elements as input (see Figure 2) and performs next computation:

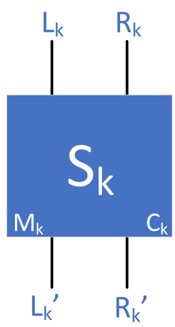

Figure 2. S-box

Equation 1.

$$L'_k = mullo(R_k, M_k)$$
$$R'_k = mulhi(R_k, M_k) \oplus key_k^i \oplus L_k$$

Round-keys key_k^i are generated by using:

Equation 2.

$$key_k^{i+1} = key_k^i + C_k$$

where:

- i – index of round
- k – index of S-box
- L_k/L_k' – the first input/output value
- R_k/R_k' – the second input/output value
- key_k^i – round key, specific for S-box and round
- key_k^0 – initial key from the engine state
- M_k – multiplier, specific S-box constant
- C_k – round constant, specific for S-box
- mullo - the low half of the product ($(a * b) mod\ 2^{\boldsymbol{w}}$)
- mulhi – the high half of the product ($\lfloor (a * b)/2^{\boldsymbol{w}} \rfloor$)
- $\oplus$ - bitwise XOR operator

For ***n*** = 2, the Philox-2 × ***w***-***r*** performs ***r*** rounds of the Philox S-box on a pair of ***w***-bit inputs. For larger ***n***, the inputs are permuted using the Threefish ***n***-word P-box before being fed, two-at-a-time, into ***n***/2 Philox S-boxes [2]. P-box of Threefish [9] is represented in Table 1:

Table 1. P-box of Threefish algorithm. Indexes of output words

	Index of input word															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
***n* = 4**	0	3	2	1												
8	2	1	4	7	6	5	0	3								
16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

Authors of Philox engine recommend next algorithm's parameters ([2], [8]):

- ***n*** is {2; 4; 8; 16}
- ***w*** equals to 32 or 64
- M satisfies "avalanche criterion" (any single-bit change in the input should result (on average) in a 0.5 probability change in each output bit)
- C is selected based on crush-resistance testing
- ***r*** is greater than or equal to 8

We propose API with broader algorithm parameters to support possible modifications of Philox engine.

V. Proposed API

We propose to add Philox to the C++ standard as the `philox_engine` engines' family with several instantiations: `philox4x32x10`, `philox4x64x10`.

Class template philox_engine

`philox_engine` is a counter-based random number engine described in [2]. It produces high quality unsigned integer random numbers of type `UIntType` in the closed interval `[0, 2^w-1]`. The state of `philox_engine` object is of size *(n+n/2)* contains *n* words and *n/2* keys of size *w* both.

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r, UIntType
         ...consts>
class philox_engine {

    static constexpr std::size_t array_size = n / 2; // Exposition only

public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static constexpr std::size_t word_size   = w;
    static constexpr std::size_t word_count  = n;
    static constexpr std::size_t round_count = r;
    static constexpr std::array<result_type, array_size> multipliers;
    static constexpr std::array<result_type, array_size> round_consts;

    // constructors and seeding functions
    ...

    // generation functions
    ...
};
```

The following relations shall hold: $(n\ ==\ 2)\ ||\ (n\ ==4)\ ||\ (n\ ==\ 8)\ ||\ (n\ ==\ 16), 0\ <\ r, w = numeric\_limits < UIntType >::digits, n\ ==\ sizeof \dots (consts)$.

The following type aliases define the random number engine with two commonly used parameters sets:

Table 2. Proposed philox_engine instantiations

Type	**Definition**
`philox4x32x10`	`using philox4x32x10 = philox_engine<uint32_t, 4, 10, 0xD2511F53, 0x9E3779B9, 0xCD9E8D57, 0xBB67AE85>;` 4 32-bits words algorithm with 10 rounds
`philox4x64x10`	`using philox4x64x10 = philox_engine<uint64_t, 4, 10, 0xD2E7470EE14C6C93, 0x9E3779B97F4A7C15, 0xCA5A826395121157, 0xBB67AE8584CAA73B>;` 4 64-bits words algorithm with 10 rounds

Other possible options:

Table 3. Other possible philox_engine instantiations

Type	**Definition**
`philox2x32x10`	`using philox2x32x10 = philox_engine<uint32_t, 2, 10, 0xD256d193, 0x9E3779B9>;` 2 32-bits words algorithm with 10 rounds
`philox2x64x10`	`using philox2x64x10 = philox_engine<uint64_t, 2, 10, 0xD2E7470EE14C6C93, 0x9E3779B97F4A7C15>;` 2 64-bits words algorithm with 10 rounds

`philox2x32x10` and `philox2x64x10` do not appear to be broadly-used but still show good statistical properties and performance [8].

`philox_engine` template parameters and members description are represented below:

Table 4. philox_engine template parameters

Parameter	**Description**
`UIntType`	One of types: unsigned short, unsigned int, unsigned long, or unsigned long long.
`n`	The number of words in the internal engine state, equals to the number of values produced by the one generation loop
`w`	The word size
`r`	The number of rounds in the one generation loop
`...consts`	Constants that are used in the algorithm (see Equation 1 and 2). The constants are grouped per S-box (`M, C`) where M is a multiplier constant, C is a round constant. The constants are set for each S-box one after another: $[M_0, C_0, M_1, C_1, M_2, C_2 \ \dots \ M_{N/2-1}, C_{N/2-1}]$

Table 5. philox_engine members description

Type	**Member object**	**Description**
`static constexpr std::size_t`	`word_size`	The template parameter `w`, determines the range of values generated by the engine
`static constexpr std::size_t`	`word_count`	The template parameter `n`, determines the number of words in the engine state
`static constexpr std::size_t`	`round_count`	The template parameter `r`, determines the number of rounds in the Philox algorithm
`static constexpr std::array< UIntType, array_size>`	`multipliers`	Contains the M_i elements of the template parameter `…consts`
`static constexpr std::array< UIntType, array_size>`	`round_consts`	Contains the C_i elements of the template parameter `…consts`

VI. Possible Alternative APIs

Template parameter `w` from the API described in Section V can be deduced from `UIntType` however this approach is inconsistent with the other existing C++ engines.

```
// ***********************************************************************
// Alternative API I: w template parameter is deduced
// ***********************************************************************

template<typename UIntType, std::size_t n, std::size_t r, UIntType ...consts>
class philox_engine {
    static constexpr std::size_t array_size = n / 2; // Exposition only

public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static constexpr std::size_t word_size   = numeric_limits<UIntType>::digits;
    static constexpr std::size_t word_count  = n;
    static constexpr std::size_t round_count = r;
    static constexpr std::array<result_type, array_size> multipliers;
    static constexpr std::array<result_type, array_size> round_consts;

    // constructors and seeding functions
    ...

    // generation functions
    ...
}
```

Template parameter `n` can also be deduced from the size of the variadic template `…consts` but it makes the API less clean for the users.

```
// ***********************************************************************
// Alternative API II: w and n template parameters are deduced
// ***********************************************************************

template<typename UIntType, std::size_t r, UIntType ...consts>
class philox_engine {
    static constexpr std::size_t array_size = sizeof...(consts) / 2;

public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static constexpr std::size_t word_size   = numeric_limits<UIntType>::digits;
    static constexpr std::size_t word_count  = sizeof...(consts);
    static constexpr std::size_t round_count = r;
    static constexpr std::array<result_type, array_size> multipliers;
    static constexpr std::array<result_type, array_size> round_consts;

    // constructors and seeding functions
    ...

    // generation functions
    ...
}
```

VII. Impact on the Standard

This is a library-only extension. It adds new engine class template and commonly used instantiations.

VIII. References

1. P1932R0 “Extension of the C++ random number generators”: http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1932r0.pdf.

2. 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
3. L'Ecuyer, Pierre & Simard, Richard. (2007). A Software Library in ANSI C for Empirical Testing of Random Number Generators. ACM Transactions on Mathematical Software - TOMS.
4. Manssen, Markus & Weigel, Martin & Hartmann, Alexander. (2012). Random number generators for massively parallel simulations on GPU. The European Physical Journal Special Topics. 210. 10.1140/epjst/e2012-01637-8.
5. Notes for Intel® Math Kernel Library (Intel® MKL) Vector Statistics : https://software.intel.com/en-us/mkl-vsnotes-philox4x32-10
6. Xu, Linlin & Ökten, Giray. (2014). High Performance Financial Simulation Using Randomized Quasi-Monte Carlo Methods. Quantitative Finance. 15. 10.1080/14697688.2015.1032549.
7. Wadden, Jack & Brunelle, Nathan & Wang, Ke & El-Hadedy, Mohamed & Robins, G. & Stan, Mircea & Skadron, Kevin. (2016). Generating efficient and high-quality pseudo-random behavior on Automata Processors. 622-629. 10.1109/ICCD.2016.7753349.
8. Random123 D. E. Shaw Research ("DESRES"): http://www.deshawresearch.com/resources_random123.html
9. N. Ferguson, S. Lucks, B. Schneier, B. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family. http://www.schneier.com/skein.pdf, 2010.