Random Numbers for Parallel Computers: Requirements and Methods, With Emphasis on GPUs

Pierre L'Ecuyer[a,b], David Munger[a], Boris Oreshkin[a], Richard Simard[a]

[a] *DIRO, Pavillon Aisenstadt, Université de Montréal, C.P.6128, Succ. Centre-Ville, Montréal (QC), Canada, H3C 3J7*
[b] *Inria Rennes Bretagne Atlantique, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France*

Abstract

We examine the requirements and the available methods and software to provide (or imitate) uniform random numbers in parallel computing environments. In this context, for the great majority of applications, independent streams of random numbers are required, each being computed on a single processing element at a time. Sometimes, thousands or even millions of such streams are needed. We explain how they can be produced and managed. We devote particular attention to multiple streams for GPU devices.

Keywords: random number generators, random number streams, multiple streams, parallel computing, supercomputers, simulation, Monte Carlo, GPU

1. Introduction

Random number generators (RNGs) are fundamental ingredients for stochastic simulation and for the application of Monte Carlo methods in general. Conceptually, these RNGs are designed to produce sequences of real numbers that behave approximately as the realizations of independent random variables uniformly distributed over the interval $(0, 1)$, or i.i.d. $\mathcal{U}(0, 1)$ for short [23, 24, 26, 34, 60]. These numbers are then transformed to simulate random variables from other distributions, stochastic processes, and other types of random objects [12, 20].

In this review paper, we examine the design and implementation of RNG facilities for parallel processing, with a special attention to discrete *graphical*

processing units (GPU) devices. Such facilities are increasingly important, because computing power now improves by increasing the number of *processing elements* (PEs), that can operate in parallel in a computer, rather than by increasing clock speeds of the PEs [5, 56], mainly because faster clock speeds produce too much heat.

Parallel computers are generally organized in a hierarchic fashion, both in terms of PEs and memory. A single chip may contain several PEs (or cores) that execute instructions in parallel. In many cases, several PEs can execute different instructions in parallel, almost independently, with only sparse communication between them. In some cases, certain groups of PEs on the chip work in the *single instruction multiple data* (SIMD) mode, which means that they must all execute the same instructions at each time step, usually with different data. This happens in array processors and GPU devices, for example, which are used as accelerators to perform the same instructions in parallel on an array (or vector) of different data. The purpose of this restriction is to speed up the processing by reducing the work required for task scheduling. On the other hand, it puts important constraints on the design of algorithms that run on such hardware.

Most often, there is a mixture of these two cases. The largest supercomputers in 2015 have over a million PEs. The PEs execute *threads*, where a thread can be seen as a sequence of instructions that can execute on a single PE (although the notions of thread and PE can have various meanings that differ in the scientific and commercial literature). The memory has a matching hierarchical organization, in which each PE can have access to a small amount of very fast-access memory, then groups of PEs share a larger memory that is slower to access, and there can be many levels of that. There is a large variety of such hierarchic organizations. On some common chips, all the PEs have fast access to a large shared memory (with a single address space) of several Gbytes. On others, such as discrete GPU chips, each PE has a fast read-write access only to a small private memory for its data, fast read-only access to a larger shared memory, and much slower access to the shared read-write memory. The read and write access times depend on the type of memory and on how it is physically connected to the PEs or to the host (on the same chip, on the same board, in the same machine, in a different machine from a same cluster, etc.).

In recent years, there has been a strong interest and much published articles on RNGs for GPU devices. Those devices were originally designed for fast high-quality image rendering on computer screens and video-game

consoles, but are now widely used as low-cost alternatives to accelerate computations in several other areas such as statistics, computational finance, and simulation in general, and have been extended to general-purpose GPUs (GPGPUs). This special interest for RNGs that run on GPU devices comes from the strong restrictions (mentioned earlier) that one faces when computing on those devices, and which are not present on other common types of processors such as ordinary CPUs (single-core or multi-core) and parallel processors typically found in the largest supercomputers. Our review gives special emphasis to GPUs for this reason. A traditional GPU board (named *discrete* GPU) contains specialized chips, each one containing multiple copies of a hardware unit with PEs that can execute a *warp* (or *wavefront*) of (typically 32 or 64) threads (also called *work items*) in parallel, in a SIMD fashion. The actual number of physical PEs is typically less than 32 (e.g., it can be 16, with one instruction per thread every two or four clock cycles). There are several variations in the architectural design, and explaining this is beyond the scope of the present article. The important aspects for our purposes are the SIMD restriction for groups of threads and the limited size of fast access private memory per thread, mentioned earlier.

To take advantage of new parallel-computing architectures, and account for their specificity and constraints, algorithms and software in general, and for RNGs in particular, must be adapted and sometimes radically redesigned. Feeding all the PEs with a single source of random numbers is generally unacceptable, first because it would create too much overhead and a large bottleneck. In highly-parallel systems, one may need thousands or even millions of virtual RNGs. They can be either different RNGs or copies of the same RNG starting from different states, that run in parallel without exchanging data between one another, and behave from the user's viewpoint just like independent RNGs. These virtual RNGs are often called *streams* of random numbers.

Another important requirement is *reproducibility of the results*: it is often required that simulations must be exactly replicable and produce exactly the same results on different computers and architectures, either parallel or purely sequential, and when running the program again on the same computer. The latter is important for debugging and in the (frequent) situation where we want to simulate a complex system with slightly different configurations or decision making rules (e.g., to optimize these rules), while making sure that exactly the same random numbers are used at exactly the same places in all configurations of the system, and repeat this n times indepen-

dently. To reduce the variance in the comparisons, it is important to use well-synchonized *common random numbers* (CRNs) [24, 32, 33, 36, 45, 41]; that is, to ensure that the same random numbers are used for the same purpose across configurations. To achieve this, one would use a separate stream for each required source of random numbers in the system, and use one substream for each of the n independent runs [24, 32, 33, 36, 41].

Reliable software facilities that provide RNGs with multiple "independent" streams and substreams have been available for some time for that purpose [33, 37, 45, 38]. Most were initially designed primarily for simulations over a single processor, for synchronizing CRNs, but they can be used as well to provide multiple streams for parallel processing. In particular, the RngStreams package of [45] has been incorporated in the `parallel` package which supports parallel programming in R [70]. RngStreams can also be used directly in OpenMP, which is the most standard interface for parallel programming in shared-memory environments, and in MPI, a popular parallel programming interface based on message passing. Examples are given in [22].

In these software tools, all the new streams are created and managed by a single central monitor. The streams are defined so they are all distinct, long enough to make sure they cannot overlap, and they behave as statistically independent. For reproducibility, the user must make sure that they are created in the same order and used for the same purpose in different configurations. This single-monitor design means that all streams must be passed or copied from the single location where they are created to all other places where they are to be used. For most parallel applications, this is acceptable and sufficient.

This setting can be extended to multiple monitors (or creators) that create the streams. Each creator will create exactly the same sequence of streams in exactly the same order, provided that the creators are created themselves in the same order. Once created, the creators no longer have to interact with each other and can be distributed in loosely connected groups of nodes.

The streams and substreams are most often constructed by partitioning the long period of a single base RNG into pieces of equal length, as in [33, 45, 41]. Other approaches such as using counter-based RNGs with different initial values for the counter, or different hashing keys, or using different RNGs from the same family (with different parameters), are also used to define streams. We will return to this later on.

There are situations where using one or more central monitor(s) to create

the streams is not acceptable, e.g., because new streams have to be created dynamically and randomly within threads as a simulation goes on, without resorting to external information, and with a guarantee that the results are exactly reproducible [15, 53]. In this setting, we want each stream to be able to create children streams by itself at any time, using a deterministic mechanism. The possibility of collisions or overlap between streams is hard to rule out in this case, but the probability that it occurs can be made extremely small, as we shall see.

Most good RNGs proposed over the past decades have been designed for single-processor environments with plenty of memory. Some popular ones have a very large state (e.g., [11, 54, 63]), which makes them inappropriate for computations on a single PE having limited fast memory, such as the PEs on discrete GPUs, and adds overhead when storing and restoring stream states, which is required, e.g., to checkpoint programs and restart them from a saved checkpoint, an essential activity in highly-parallel computations. In those types of setting, several PEs can be used simultaneously to advance the RNG state by one step, so these PEs all work on the same stream. But this often does not match the need of users, e.g., if one wishes to run one simulation thread per PE. RNG frameworks and implementations specifically adapted to GPUs, and other parallel processing environments such as programmable arrays, have also been proposed in recent years; see for example [6, 19, 49, 58, 65, 64, 73, 74, 75, 80]. Parallel processing accelerators can be exploited in different ways in a simulation. On a GPU, for example, one can either run a piece of simulation on each PE, using one or more random stream per PE, or one can just use the GPU to rapidly fill up a large buffer of random numbers to be used on the host computer or elsewhere.

The rest of this paper provides a (brief) survey of the main issues, requirements, and recent proposals, for RNGs on parallel computing environments. In Section 2, we provide some background on algorithmic RNGs and their (classical) quality criteria. Section 3 quickly summarizes the main classes of RNGs used for simulation. In Section 4, we outline a set of requirements and ideas to provide multiple streams for parallel computing, and discuss what we think application programming interfaces (APIs) should contain. In Section 5, we describe various efforts made to adapt existing RNGs to parallel settings, and to define new ones that can be more appropriate.

2. Background

One way to generate (approximately) $\mathcal{U}(0,1)$ random numbers is to generate random bits from physical devices and use these bits to construct real numbers represented in floating-point with finite precision. Those random bits can be obtained from thermal noise in semiconductors, photon counters, photon trajectory detectors, etc. With such devices, there is generally no mathematical proof or guarantee that the successive bits are statistically independent and that each bit is 0 or 1 with probability 1/2 each, but some of them appear reliable enough so we can make this assumption, based on empirical statistical tests that check for detectable bias or dependence. However, they are cumbersome to install and use (particularly on parallel processing elements) and, more importantly, they cannot reproduce exactly the same sequence twice. This reproducibility is a key requirement for simulation applications, e.g., for program verification and for comparing similar systems with common random numbers [24, 32, 33, 41, 65].

Algorithmic generators, often called *pseudorandom*, are more appropriate and much more popular than physical devices for simulation and Monte Carlo methods. They are in fact deterministic finite automata that produce a periodic sequence of numbers which behave like typical i.i.d. $\mathcal{U}(0,1)$ when viewed externally [26]. In the rest of this paper, the term RNG will always refer to an algorithmic generator. An RNG has a finite set of *states* $\mathcal{S}$, a transition function $f : \mathcal{S} \to \mathcal{S}$, an output function $g : \mathcal{S} \to (0,1)$, and an initial state (or *seed*) s_0. Sometimes, the seed s_0 is chosen randomly (via an external mechanism) from some probability distribution on $\mathcal{S}$. The state evolves according to $s_i = f(s_{i-1})$ and the output value returned at step i is $u_i = g(s_i) \in (0,1)$, for $i \geq 0$.

Because $\mathcal{S}$ is a finite set, there are always finite integers $l \geq 0$ and $0 < j \leq |\mathcal{S}|$ such that $s_{l+j} = s_l$. Then, $s_{i+j} = s_i$ and $u_{i+j} = u_i$ for all $i \geq l$. The smallest $j > 0$ for which this holds is the *period* ρ of the RNG. Note that $\rho \leq |\mathcal{S}| \leq 2^k$, where k is the number of bits used to represent the state. Well-designed RNGs typically have $l = 0$ and ρ near 2^k.

Basic requirements for good RNGs include high running speed, long period (e.g., 2^{200} or more), exact repeatability on various computing platforms, ease of implementation, preferably in a platform-independent way, and efficient ways of splitting the sequence into long disjoint streams and of jumping quickly between them. On top of these basic properties, we also want the successive output values to have an i.i.d. $\mathcal{U}(0,1)$ behavior, which can be cap-

tured by the following: for any dimension $s > 0$, if we select the seed s_0 at random uniformly over $\mathcal{S}$, the vector $\mathbf{u}_{0,s} = (u_0, \ldots, u_{s-1})$ should have (approximately) the uniform distribution over the unit hypercube $(0,1)^s$. This vector cannot have *exactly* the uniform distribution over $(0,1)^s$, because $\mathcal{S}$ is a finite set, but it has a uniform distribution over the set

$$\Psi_s = \{(u_0, \ldots, u_{s-1}) = (g(s_0), \ldots, g(s_{s-1})) : s_0 \in \mathcal{S}\}.$$

The best we can ask for is that Ψ_s covers the unit hypercube $(0,1)^s$ very evenly, in some sense, so that the uniform distribution over Ψ_s is a good approximation of that over $(0,1)^s$. Good RNGs must be constructed based on a mathematical analysis of this uniformity, which is measured by figures of merit that must be easily computable without generating the points explicitly. These measures generally depend on the structure of the RNG [17, 23, 34, 42]. With a larger Ψ_s (i.e., larger $\mathcal{S}$), one can potentially cover $(0,1)^s$ more uniformly, and this is the main motivation for having a larger state (more than trying to increase the period). On the other hand, a larger state and longer period does not necessarily imply a better RNG. A large state also has many drawbacks: it requires more memory to store the states, and more overhead to compute the seeds of the multiple streams and to store and restore the states when needed.

After selecting an RNG based on proper mathematical analysis, one submits it to empirical statistical tests, to try to detect observable non-uniformity or dependence [23, 43]. A test takes a stream of successive output values, computes the value t taken by a test statistic T, then computes the p-value of the test, defined as the probability p that $T \geq t$ under the hypothesis $\mathcal{H}_0$ that the output values are independent $\mathcal{U}(0,1)$ random variables. A p very close to 0 usually indicates strong departure from uniformity, whereas p very close to 1 indicates excessive uniformity, which also represents departure from randomness. For poor RNGs, p-values smaller than 10^{-15} (and often much smaller) are often returned [43], and it is then very clear that the RNG fails the tests. Collections of statistical tests for RNGs can be found in [23, 43, 52], for example. The most extensive collection and software is TestU01 [43]. Among other things, it includes predefined batteries of tests called SmallCrush, Crush, and BigCrush, which have become very popular. RNGs that pass all tests in those batteries are called *Crush-resistant* [74]. However, it is only possible to apply a finite (small) number of statistical tests in practice, and this can never prove that a RNG is flawless. For this

reason, theoretical tests that measure the uniformity by studying the mathematical structure are deemed more important and convincing than empirical tests, when they can be applied.

3. Main classes of RNGs for simulation

The most popular classes of RNGs used for simulation are based on linear recurrences modulo m, either for $m = 2$ (which amounts to doing linear operations in the finite field $\mathbb{F}_2$) or for a large integer m. A large class of $\mathbb{F}_2$-linear RNGs can be represented as:

$$\mathbf{x}_i = \mathbf{A}\mathbf{x}_{i-1} \bmod 2, \tag{1}$$

$$\mathbf{y}_i = \mathbf{B}\mathbf{x}_i \bmod 2, \tag{2}$$

$$u_i = \sum_{\ell=1}^{w} y_{i,\ell-1} 2^{-\ell} \tag{3}$$

where $\mathbf{x}_i$ is the k-bit *state vector* at step i, $\mathbf{y}_i = (y_{i,0}, \ldots, y_{i,w-1})^\mathsf{T}$ is the w-bit *output vector*, k and w are positive integers, $\mathbf{A}$ and $\mathbf{B}$ are binary matrices of appropriate sizes, and $u_i \in [0, 1)$ is the *output* at step i. In practice, the output function is modified slightly to avoid returning 0. Several fast RNGs such as the linear feedback shift register (LFSR) generator, generalized feedback shift register (GFSR), twisted GFSR, Mersenne twister (MT), WELL, xorshift, shift registers in lookup tables (LUT-SR), and combinations of these, belong to this class [31, 34, 42, 54, 63, 77]. The maximal period for the state is $2^k - 1$ and is reached when the characteristic polynomial of $\mathbf{A}$ is primitive in $\mathbb{F}_2$. The matrices $\mathbf{A}$ and $\mathbf{B}$ usually represent simple binary operations that are fast to execute on binary computers, such as or, exclusive-or, shift, and rotation, on blocks of bits. The matrices should also be selected in a way that the point sets Ψ_s have good uniformity, which can be assessed by exploiting the linear structure to compute measures of equidistribution of the points in dyadic rectangular boxes [31, 42]. Empirically, all these RNGs fail statistical tests designed to detect linearity in the bits, such as tests that compute the linear complexity of successive bits at a given position, $\{y_{i,j},\ i \geq 0\}$, or the rank of "random" binary matrices constructed by the RNGs, so they are not Crush-resistant, but they are still appropriate for applications in which the bits of the u_i's are used nonlinearly. Some of the most popular ones use a very large k, which make them not so convenient for GPUs, because the state occupies too much memory. For example, the most popular MT instance,

named MT19937, has $k = 19937$, whereas the WELL uses the same k and even larger ones [63].

Linear recurrences modulo a large integer m are usually written as

$$x_i \;=\; (a_1 x_{i-1} + \cdots + a_k x_{i-k}) \bmod m, \tag{4}$$

for some coefficients $a_1, \ldots, a_k$ in $\{-m+1, \ldots, 0, 1, \ldots, m-1\}$, with $a_k \neq 0$, and k is the order of the recurrence. This can be written equivalently as (1) with 2 replaced by m, $\mathbf{x}_i = (x_{i-k+1}, \ldots, x_i)^\mathsf{T}$, and the appropriate $k \times k$ matrix $\mathbf{A}$. Typically, m is a prime number slightly smaller than 2^{31} or 2^{32} or 2^{63}, and the coefficients are selected so that the period is $m^k - 1$ and the sets Ψ_s have good uniformity, measured again by exploiting the linearity (via the spectral test, which measures the quality of the lattice structure) [23, 30, 34]. The output can be defined as $u_i = x_i/m$, or by $u_i = (x_i + 1)/(m + 1)$ or $u_i = (x_i + 1/2)/m$ to avoid returning 0, for example. This is known as a *multiple recursive generator* (MRG). For $k = 1$, it gives a *linear congruential generator* (LCG), now considered obsolete because its period is much too small and its lattice structure is much too coarse (the point sets Ψ_s are too small). Taking k too large, on the other hand, makes the state too large for convenient use on traditional GPUs.

The multipliers a_j are usually selected in a way that (4) is very fast to compute, which could mean taking many of them equal to 0, the other ones equal to ± 1 or to the same constant a, etc. But doing this can lead to bad generators [29, 34, 44]. For example, the lagged-Fibonacci RNGs, as well as the add-with-carry and subtract-with-borrow (which are slightly modified variants of the MRG) employ only two nonzero coefficients, say a_r and a_k, both equal to ± 1, and all their output vectors the form (u_i, u_{i-r}, u_{i-k}) lie in only two parallel planes in the unit cube $[0, 1)^3$ [29]. These generators give wrong results in certain simulations and they fail several statistical tests [43].

For all these linear RNGs, one can jump ahead directly by ν steps for an arbitrarily large ν by computing $\mathbf{x}_{i+\nu} = (\mathbf{A}^\nu \bmod m)\mathbf{x}_i \bmod m$, where $\mathbf{A}^\nu \bmod m$ can be precomputed in advance [25, 34, 45], provided that k is not too large. This is useful for computing the initial states for separate streams of random numbers whose starting points are very far apart.

Two or more small $\mathbb{F}_2$-linear RNGs can be combined via a bitwise xor of their outputs and it gives another $\mathbb{F}_2$-linear RNG with much larger period and better quality point sets Ψ_s than its components [42], and whose uniformity measures can still be computed. This can also be done for MRGs with

different moduli, where the combination is via addition modulo 1, for example [27]. See [30, 35, 47, 43] for specific constructions of this type. They include `MRG32k3a` and `MRG31k3p`, which come with multiple streams and substreams [30, 33, 45, 41]. If we combine MRGs with $\mathbb{F}_2$-linear RNGs, the structure becomes nonlinear and the uniformity becomes harder to measure other than by empirical statistical tests; but see [39] for some theoretical results.

Besides the linear ones, there are also various classes of RNGs for which either f or g is nonlinear. For example, the multiplicative lagged Fibonacci generators proposed by [51] replace the additions in (4) by multiplications and use only two nonzero coefficients a_j, both equal to 1. They perform well empirically in statistical tests [43].

As a general rule, there should be no easily detectable dependence between successive output values, and in particular for any given bits in those values. To achieve this, either f or g or both must perform a sufficient amount of work to transform the bits of the state at each step. For the conventional RNGs discussed so far, almost all of this work is done by the transition function f, and g does very little. But it is also possible to do the opposite, or to balance the work in a different way.

In *counter-based RNGs* [18, 43, 74, 78, 80], f simply increments a counter. The state at step i is just i, so $f(i) = i + 1$, and all the hard work is performed by g, which can be taken as a bijective block cipher encryption algorithm such as MD5, the tiny encryption algorithm (TEA), the secure hash algorithm (SHA), the advanced encryption standard (AES), ChaCha, Threefish, etc. [8, 43, 59, 69, 74, 78, 80]. An important advantage is that the output value u_i at an arbitrary position (or step) i can be generated directly and quickly regardless of i. These values can then be generated in any order, they are easy to replicate, and it is easy to split the entire sequence into long disjoint streams. Since g is a bijection, the period is 2^k where k is the number of bits used to represent the counter. This k will usually be a multiple of 32, say in the range from 128 to 1024. Good encryption algorithms satisfy the *avalanche criterion*, which says that changing a single bit in the counter changes approximately half the bits in the output, on average. One drawback is that they are typically slower than popular RNGs when implemented in software. But they can be simplified and still remain good enough for many simulation applications. Salmon et al. [74] propose counter-based RNGs that use simplifications of the AES and Threefish algorithms, named ARS and Threefry, which trade security for speed, and a new counter-based method called Philox. These new proposals are Crush-resistant and this was a prime

criterion in their design. One important limitation of all these counter-based RNGs is that there is no theoretical analysis of uniformity for their point sets Ψ_s. For image rendering in computer graphics, Crush-resistant may be too strong a requirement; one would prefer a faster RNG as long as it produces acceptable visual effects. A fast counter-based RNG for GPUs is proposed for that using MD5 in [78] and faster ones using TEA in [80]. Other ones that can fit in one thread in a traditional (discrete) GPU can be found in [8, 49, 59, 69].

4. How to produce parallel streams and substreams

There are various ways of providing random numbers for parallel processors. We discuss and compare the main ones. The effective methods use multiple streams of random numbers. Those streams can be obtained in many ways, for instance by using a different RNG (e.g., different parameters in the definition), or by a leap-frog technique with a single RNG, or by splitting the sequence into long disjoint blocks of equal length, or by splitting it using random starting points for the streams. Using a single stream for everything is not a solution, as we shall see.

The easiest and cleanest way to create the streams is via a central monitor, which creates the streams in a specific order and passes them to the threads or other software entities that need them. As an indirect but equivalent way of doing this, the monitor can provide only an ID number for each stream instead of creating it explicitly, and there can be a well-defined mechanism that allow any entity to reconstruct the stream (and its initial state) from its ID number. A totally different setting is if streams can be created in a distributed fashion, e.g., for example, if we want to allow each stream to have children streams.

A single source of random numbers (single stream). A naive setting would be to have all random numbers generated on demand by a single RNG, or produced in advance and stored in global memory or in a file. The generation could be done either on a host CPU or on a parallel device to speed up the production. The methods that return the random numbers to the threads must be thread-safe, so that things work properly when several threads invoke them simultaneously. This involves overhead, especially when the random numbers are consumed by a large number of PEs. Data exchange between the different memories would also involve overhead and create a bottleneck,

which would be unsustainable on highly parallel systems. Finally, this scheme is unacceptable because the way the random numbers are consumed would vary between runs, depending on how the different parallel threads execute on the PEs, and therefore the results would not be reproducible [19, 64].

A different generator for each stream. This usually means using the same type of RNG, but with different parameters. For example, one can use LCGs or MRGs with the same modulus but different multipliers [13, 25]. Finding many different good parameter sets is usually feasible, but this approach is not very convenient to manage, because finding those good parameters on the fly or precomputing and storing them adds overhead either way, in time or space. Moreover, different parameters give different streams but it does not necessarily imply that the streams can be considered as independent [4, 8], so this independence must be tested in some way [17, 45, 74]. Some have suggested to choose parameters at random among those that give a maximal period, but this is dangerous, because maximal period does not suffice, as we saw earlier; some parameters may still give point sets Ψ_s that are far from uniform.

As an example of this approach, the dynamic creator program [55] permits one to create dynamically new instances of good-quality MT generators. It is used in [73] to obtain parallel streams based on different parameterizations. SPRNG [14, 48, 53] also uses a parameterization approach to create new streams during execution. With counter-based RNGs from block ciphers which require an encoding key in addition to the counter, a simple and elegant parameterization is just to select a different key for each stream, among the huge set of admissible keys [50, 59, 69, 74, 78, 80]. There is no need to search for good keys or precompute tables of keys. For example, to assign a different stream to each thread (or software entity), one can just take its ID number (or a bijective function of it) as the key [59]. If we assume that block ciphers are perfectly safe, this implies that the streams produced with different keys are statistically independent. However, this assumption is not formally proved.

Leap-frogging with a single RNG. Given p processors and a single RNG with transition function f, the leapfrog approach produces p streams as follows. For $j = 0, \ldots, p-1$, stream j outputs the subsequence $\{u_{ip+j},\ i = 0, 1, 2, \ldots\}$ from the original RNG. The values $u_{ip}, \ldots, u_{ip+p-1}$ can be computed in parallel at each time step i. This is how the vectorized implementations in [58, 73]

operate. However, these implementations work only for specific values of p and the p streams do not evolve independently. The p values must be updated together for each i and must be shared by all processors, and each stream must consume one value at each time step, which is not always convenient. More generally, one could think of using the recurrence based on f^p for an arbitrary p and run it separately for each stream, e.g., in independent threads. But most RNGs are designed so that f (not f^p) can be computed quickly. In popular linear generators, jumping ahead by p values typically takes significantly more time than advancing by one step.

Splitting a single RNG into equally-spaced blocks (streams). The most common approach to produce multiple streams from a single RNG is to split the sequence into long subsequences of equal length, by starting the streams at different seeds. To compute those seeds, spaced say ν steps apart for a very large ν, we must be able to compute quickly the ν-step transition function f^ν. For linear RNGs, this can be achieved as we saw earlier; see also [46, 45]. Although jumping ahead is generally slower than generating one value, it is normally used more sparingly and it is fast enough unless k is large [6, 33]. When k is large, e.g., for $\mathbb{F}_2$-linear RNGs such as MT19937 and the WELL RNGs [54, 63], the matrix $\mathbf{A}$ is huge and squaring it repeatedly to precompute $\mathbf{A}^\nu \bmod 2$ becomes impractical. By standard matrix multiplication, this requires $\mathcal{O}(k^3 \log \nu)$ time. A more efficient method for that situation has been developed in [16], based on a polynomial representation. But this method is still slow when k is in the thousands, such as for MT19937 [6]. We think this is one of the good reasons for staying away from such large values of k. If one insists on a very large period (large k) and still wants fast jump ahead, a good way to achieve it is to use a combined generator, for which the jump ahead is done by jumping ahead for each (smaller) component separately. This is done in [33, 41] for various combined RNGs. For a counter-based generator with a fixed key, jumping ahead is trivial: just add ν to the counter.

In most software with multiple streams, the length ν of the streams is fixed to a very large constant (e.g., $\nu = 2^{127}$ in the RngStream package of [45], based on MRG32k3a). If we want to allow the user to choose any ν, $\mathbf{A}^\nu \bmod m$ (for linear RNGs) cannot be precomputed in advance. But when streams are used for parallelization, some users may want to have the flexibility to decide on the exact length of the streams, i.e., the distance between their starting points. This is useful for example to ensure that in a parallel version of a program, the streams can be defined in a manner that

the random numbers are used in exactly the same way (and the results are exactly the same) as in a sequential version of the same program with a single stream. To do this, one needs to know how many random numbers are used from each stream, and in what order they are used in both the parallel and the sequential versions. (Another (easier) way to obtain the same results on the sequential and parallel computers, is to run the program with the same streams and substreams on both computers, with standard spacings.)

To offer efficient jump-ahead facilities by an arbitrary ν with MRG32k3a, Bradley et al. [6] precompute and store $\mathbf{A}^p$ for all p of the form $p = ib^e$ for some base b, $i = 1, \ldots, b-1$, and $e = 0, \ldots, e_{\max}$, and for $p = b^{e_{\max}}$. They choose $b = 8$ and $e_{\max} = 20$ in their code. Whenever one needs $\mathbf{A}^\nu \mathbf{x}$ for some $\nu \leq b^{20}$ and $\mathbf{x}$, one writes ν in base b, i.e., as a sum of terms of the form ib^e, which effectively writes $\mathbf{A}^\nu$ as a product of matrices that have been precomputed. The vector $\mathbf{x}$ is then multiplied by those matrices, in succession. This technique applies more generally to all generators based on a linear recurrence.

In [33, 45, 41], each stream is also divided into substreams, and when comparing similar systems (using a single processor), each simulation run (or replication) uses a new substream. After each run, we advance all streams to their next substream. This is convenient in a single-processor setting.

In a situation where each simulation run uses multiple processors and requires an unknown number of independent streams, it may be more convenient to assign the streams to the runs, and the substreams to the different parallel threads or tasks that need them, because it may seem easier to provide the seeds of the successive substreams of a given stream than to provide a specific substream number for many streams when making a simulation run. But the latter is also possible: To get the seed of substream number j for successive streams of length ν, start with the seed of substream number j for the first stream, and jump ahead repeatedly by ν steps at a time.

One potential concern with splitting is the dependence between the different streams. For example, for LCGs for which both the modulus m and the spacing ν are powers of 2, the streams are strongly dependent [10, 13, 25]. In general, one should measure or test the uniformity of vectors of the form $(u_i, \ldots, u_{i+d-1}, u_{i+\nu}, \ldots, u_{i+\nu+d-1}, u_{i+2\nu}, \ldots, u_{i+2\nu+d-1}, \ldots)$, for various choices of $d \geq 1$. This was done in [45] with the spectral test to select good values of ν. It is advocated (and easy) to apply statistical tests to vectors of this form, using TestU01 [43, 74]. This recommendation holds more generally, to test the dependence across streams regardless of how they are defined

(e.g., by using different parameters for the different streams).

A single RNG with a "random" seed for each stream. When jumping ahead is too expensive and the RNG has a huge period, it can be reasonable to just select a (pseudo)random seed for each stream, using another RNG if we want reproducibility. There is a possibility of overlap, but the probability is often negligible. If the RNG has period ρ and we take s different streams of length ℓ, with random starting points in the sequence, the probability that none of those streams overlap anywhere is approximately $p = (1 - s\ell/\rho)^{s-1}$; see [13]. This gives $\ln p = (s-1)\ln(1-x)$ where $x = s\ell/\rho$. When x is very small, we have $\ln p \approx -(s-1)x$ and then the probability that there is overlap somewhere is $1 - p \approx -\ln p \approx (s-1)x \approx s^2\ell/\rho$. As an illustration, if $s = \ell = 2^{20}$, then $1 - p \approx 2^{60}/\rho$ is near 2^{-68} for $\rho = 2^{128}$ and near 2^{-964} for $\rho = 2^{1024}$. For counter-based RNGs parameterized by a random key, with s streams and ρ possible keys, the probability $1 - p$ that the s random keys are not all distinct is $1 - p \approx s^2/\rho$ (this corresponds to $\ell = 1$ in the above formula). For example, if $s = 2^{30}$ (about one billion) random 128-bit keys are selected, the probability that they are not all distinct is approximately $2^{60}/2^{128} = 2^{-68}$, which is really negligible.

Another important situation where "random seeds" are useful is when the streams must split during execution, so that new streams are created dynamically. This evolution can be represented as a *random tree of streams*. Any given stream may have to split in two streams (or perhaps more) at any given step. This splitting is not fully predictable (it depends on the evolution of the simulation) and the new streams must behave independently after the split. This type of splitting occurs in nuclear particle simulations in high-energy physics, where a stream is assigned to each particle, and the particles can split when they collide [15, 53]. It occurs in a similar way in computer graphics, where the particles are replaced by rays of light, which can split when they hit a surface (some light is absorbed, some is reflected, and some may be refracted) [80]. Relying on a central provider to manage the streams is not convenient or acceptable for these applications, at least if we insist on reproducibility, because the calls to the central manager are likely to change order across replications. The SPRNG software [14, 53] was designed for this type of setting. Conceptually, it contains a binary tree of distinct streams, numbered according to their position in the tree. When a stream splits, its two children in the tree are uniquely determined, regardless of the time at which the split occurs. The number of levels in the tree

depends on the number of bits used to number the nodes. In SPRNG, the different streams are defined by the parameterization method: to each node number corresponds a set of RNG parameters. With the RNGs and implementation proposed in [53], certain parameter choices may give rise to poorly behaved streams, and some of the proposed RNGs are poorly behaved (e.g., the additive lagged-Fibonacci and some simple LCGs). Splittable streams are also proposed in [8], using a hash function to determine the new streams when splitting occurs, and a 256-bit version of the Threefish block cipher to produce the random numbers in the streams.

How to assign streams. In parallel processing, to generate the random numbers locally on each PE, it suffices to have at least one stream on each PE. In some multithread programming models, a large number of threads run on a collection of physical PEs, any given thread can run sequentially on different PEs, and a physical PE executes various threads in succession, in a time-slice fashion. These threads can be created dynamically during execution, or a large pool of threads can be created at the beginning and used by the different programming subtasks, as is the case in Java JDK 7, for example. In this context, since each thread has its own private data (not each core), the threads can be seen as (logical) PEs [64, 67], and it appears more relevant to have one stream per thread rather than one stream per physical PE. This is done for example in JAPARA [9], in which a synchronized method in a central monitor computes (sequentially) the seeds of the different streams and assigns them to the threads, using a splitting technique as in [45, 46]. Once the thread gets its seed, it can evolves independently; there is no further need for the central monitor. The one-stream-per-thread strategy is also discussed and implemented in [50, 64, 66].

However, with one stream per PE or per thread (when the threads act as logical PEs), the simulation results may depend on how the hardware is organized and on how the computations are assigned to the PEs or the threads, which may vary from one execution to the next, even on the same hardware. To ensure reproducibility, one should assign streams at a higher level, in terms of computing tasks (or subtasks) that are executed on threads, and the specific usages of random numbers in the simulation model. For example, when simulating a large service system with waiting queues, one stream can be devoted to generate the arrivals of each specific type of customer, another stream for the service times of each type, etc., so the random numbers remain well synchronized when comparing similar systems with CRNs. These

assignments and the simulation results should not depend on the hardware in any way, neither on how the tasks are assigned to threads.

Passerat-Palmbach [64] points that out and proposes a Java class named `TaskLocalRandom` [67], based on the RngStream package of [45], and designed for parallel simulation programming in Java. In this class, each task is created with a unique ID number and a stream number is assigned to each ID number. Precomputed tables in global memory permit one to quickly find the seed for any stream number. When performing multiple independent simulation runs, the different runs should use different streams (or substreams), but this can be easily achieved even with one stream per task (and no substreams) by defining different task numbers for the different runs.

This way of assigning one stream per task can achieve reproducibility, since each task always gets the same stream regardless of the hardware. However, in many cases this is not sufficient, because it does not permit one to have more than one stream per task, which is often required to facilitate CRN synchronization [24, 32, 36], as we saw earlier. One should be able to assign multiple independent streams to any given task. The streams can be created by a central monitor and passed to the tasks in the same way as any other data or objects used by those tasks. For reproducibility and good synchronization, one must make sure that they are always created in the same order, regardless of how the tasks are executed. This is done in [41], for example. See also Section 5.1.

5. RNGs adapted to parallel-processing hardware

We distinguish here two main schemes (or models) for exploiting parallel computing for random number generation and simulation, and discuss specific methods and software that have been proposed for each. In the first scheme, several streams can evolve in parallel, each stream running on a single PE at a time. In this model, each stream evolves in the same way as on an ordinary sequential computer, even though it may switch its execution from one PE to another, sequentially. In the second scheme, the goal is to produce random numbers, from either a single stream or many streams at a time, at a faster rate than on a single PE, by running a vectorized implementation on a group of several PEs. We name them *vectorized RNGs*. Which scheme is more appropriate depends on the type of hardware and the intended application. In the first scheme, the random numbers are often consumed locally on the PE where they are generated, whereas in the second

scheme they are usually copied to a global memory and consumed elsewhere or stored for later use. In many RNG libraries, the two schemes are offered and can be mixed.

5.1. Each stream running on a single PE at a time

Most of the best currently available RNGs have been designed for conventional CPUs with "standard" instruction sets and plenty of fast-access memory. To use them on a parallel computer whose PEs have these characteristics and can execute their instructions independently of each other at any given time step, it suffices to make multiple streams as explained earlier. There is no need to change their implementation. The PEs (cores) in recent multi-core CPUs and APUs share a large common fast-access memory in addition to their separate registers, so they can accommodate the same RNGs as in traditional CPUs. The hundreds of thousands of PEs in largest supercomputers today also behave much like traditional CPUs. In this type of setting, a standard multi-stream package such as RngStream can be used directly [22].

However, on certain types of widely-used parallel-compute devices, most notably discrete GPUs, but also others such as vectorized (array) processors, there are various types of limitations such as a very small fast-access private memory for each PE, limited instruction sets, and the SIMD constraint that groups of PEs or threads must execute the same instructions simultaneously, in parallel. The smaller fast-access (private) memory and also (more generally) the cost of communication between PEs, especially when there are many of them, should discourage the use of RNGs with a very large state, such as those in [43, 44, 54, 63] for example. The type of hardware may also impact what functions f and g are appropriate by favoring or ruling out certain types of instructions, due to the different architectural design of the processors [75]. We discuss those issues in the rest of this section.

Different instruction sets and hardware constraints. On discrete GPUs, warps (or wavefronts) of 32 or 64 threads must execute the same instructions simultaneously, in parallel. At each step, some of the threads can be deactivated, so they do nothing at that step. But doing too much of this would waste clock cycles. Each PE has only a limited amount of fast-access memory, so if we decide to run a stream in a single PE (or thread) at a time, the RNG state must be small, say no more than about six to eight 32-bit words. Instruction

sets on traditional GPUs are also limited and RNGs must be chosen in accordance with those constraints. For example, double precision floating point arithmetic was either slow or nonexistent until recently. Recent GPGPUs and APUs do not have these limitations, however.

RNGs with a small state. Examples of recommendable RNGs with a small state, that can run easily on a single PE in a discrete GPU and have been implemented for that, are LFSR113 [4, 31, 41], MRG31k3p [47, 41], MRG32k3a [4, 6, 30, 41], a hybrid Tausworthe-LCG [21] which combines the taus88 generator of [28] with an LCG, LFSR113 combined with an LCG [57], and a xorshift with a running sum modulo 1 as output (to break the linearity) [75, Section 7]. There are other good RNGs whose state is small enough so they can be implemented in a single thread; see [43] for example.

Counter-based RNGs. Recently, counter-based RNG have been advocated as an easy and attractive solution to provide millions of streams, each one having a small state [8, 50, 69, 74]. Some say that the state should fit in 128 bits, for example [50, 69]. With such a small state, it is possible to have more than one stream in a thread. In computer graphics, it sometimes happens that the same stream (at specific positions) must be accessed by different threads running on different cores. Counter-based RNGs are very handy for this situation [80]. Neves and Araujo [59] propose the Tyche RNG, which uses the ChaCha encoding algorithm over 128 bits and can provide one stream per thread. Phillips [69] propose one stream per thread per kernel call, based on a hash function, which uses a global seed broadcast to all threads, the thread id, and a counter, and hashes all this information to get the output. Block cipher encoding with a different key for each stream is used in [59, 74]. Note that both the key and the counter use memory; the state can be seen as their juxtaposition.

Some available libraries. We now briefly discuss a small selection of available RNG libraries that offer multiple streams for parallel simulation, with emphasis on those that support GPUs.

The CUDA programming language offered by NVIDIA for their GPUs comes with the CURAND library [61], which currently implements five types of RNGs: Xorwow (not recommended; see [72]), MRG32k3a from [30], MT19937 from [54], MTGP from [73], and PHILOX-4×32-10 from [74]. Multiple streams (they are called "generators") can be created from each type, but

only a limited number of them can be created. The random numbers can be generated and consumed either on the host computer or on the GPU device.

For OpenCL, a more general C-like programming environment for parallel processing, but which targets primarily GPUs for general computations, the recently-proposed clRNG library [40, 41] offers various types of RNGs, with arbitrary numbers of streams and substreams. The streams can be created in practically unlimited number. They are created on the host computer by a central monitor. They can generate and consume random numbers either on the host or on a device (such as a GPU). Each stream also has multiple substreams, and one can skip ahead to the next substream or rewind to the beginning of the current substream or the beginning of the stream, as in [45]. It currently implements four RNGs: MRG31k3p from [47], MRG32k3a from [30], LFSR113 from [31], and PHILOX-4×32-10 from [74].

Barash and Shchur [4] propose a library called PRAND, with selected RNGs implemented for both CPU and GPU usage, in Fortran, Cuda, and C with streaming SIMD extensions (SSE) for improved performance. It includes MGR32k3a [30], LFSR113 [31], MT19937 [54], and a series of RNGs named GL and GQ, based on linear recurrences in the two-dimensional unit torus, all with good statistical behavior. The GL and GQ are generally slower and also have shorter periods than the first three. The authors provide facilities for jumping ahead in the sequence, to obtain starting points for multiple streams, using essentially the same algorithms as in [45, 33]. One can produce random numbers with one or more streams per PE. In their tests and comparisons, the authors are mainly interested in how fast one can fill a large array of random numbers on a GPU, using multiple threads and multiple streams. The array is divided into contiguous sections and each section is filled by a group of threads that run one stream.

Bradley et al. [6] implement MGR32k3a and MT19937 in a CUDA environment, with jump-ahead facilities to provide multiple streams, in a setting where the users can decide on the exact length of each stream. In other software, such as in [45, 33, 41], this length is usually fixed. To provide efficient facilities to jump ahead by an arbitrary number of steps, for MGR32k3a they pre-compute and store several matrices, as explained in Section 4. For MT19937, they have groups of 224 threads working on each stream in a similar way as in [73], and they use the method of [16] to jump ahead. For MRG32k3a, each stream can run in a single thread. In speed benchmark tests, to generate 2^{25} random numbers in multiple streams on GPUs with CUDA, the MRG32k3a turned out to be faster than MT19937, due to its

faster jumping ahead. Their software is included in the NAG library.

5.2. Multiple PEs for a single stream and vectorized RNGs

Several authors propose and implement methods that exploit parallel processing to generate random numbers at a faster rate, mostly to speed up the generation for a single stream of random numbers. Note that methods that return an array of random numbers are useful even in a single-processor setting, to amortize the function call, which often takes more time than computing the next random number.

RNG libraries using vectorized instructions. In the Intel Math Kernel Library, the Vector Statistical Library (VSL) [1, Chapter 9] offers tools to generate vectors of random numbers via the vectorized functions found in C with streaming SIMD extensions (SSE), optimized to take advantage of SIMD instructions available in recent Intel processors. One can create and use multiple streams of random numbers, each one from a base RNG. Eight specific RNGs are available: MCG31m1, R250, MCG59, MRG32k3a, WH, MT2203, MT19937, and SFMT19937. The first three fail several tests and should not be used; see [43]. The last one is a version of MT19937 running on several PEs [73] in parallel. As a downside, the user must provide the seed explicitly for each stream, by providing either a single 32-bit integer or an array of 32-bit integers, used to construct the seed. This gives no guarantee of sufficient spacing between the streams. For 6 RNGs out of 8, a function is available to jump ahead by ν steps, where ν is a 32-bit integer specified by the user. That is, ν must be less than 2^{32}, which is quite small for the maximum length of a stream.

Barash and Shchur have proposed RNGSSELIB [3], which resembles their PRAND library and offers the same RNGs, also with multiple streams, but was targeted at vectorized operations and programming via SSE, whereas PRAND is also for GPU programming.

Vectorized RNGs for GPUs. The following methods and software have been developed for the setting in which a CPU host uses a block of threads on a discrete GPU to produce a large array of random numbers.

Saito and Matsumoto [71, 73] have done that for MTs. In [71], they propose a fast implementation of MT19937 for a specific single-instruction multiple data (SIMD) architecture that offers fast operations on 128-bit integers. The tight connection with the specific hardware is an important limitation.

In [73], they propose a class of MTs for graphic processors (MTGP) adapted to the architecture of GPUs. It exploits the fact that their MT recurrence can be written as

$$x_i = f_0(x_{i-k}, \ldots, x_{i-k+r}) \tag{5}$$

where the RNG state at step i is written $\mathbf{x}_i = (x_{i-k+1}, \ldots, x_i)$ and each x_i is a block of 32 bits. The idea is to set up a circular buffer of length $\ell \geq 2k-r$ that will contain vectors of successive values $(x_{i-k+1}, \ldots, x_i, x_{i+1}, \ldots, x_{i+k-r})$. At any step, a portion of the vector contains $\mathbf{x}_i = (x_{i-k+1}, \ldots, x_i)$, and the next values $x_{i+1}, \ldots, x_{i+k-r}$, which depend only on $\mathbf{x}_i$, are computed in parallel with $k - r$ threads. The table is circular in the sense that the indices i are always computed modulo ℓ, so $\mathbf{x}_i$ can actually be at any k successive positions (modulo ℓ) in the table. This idea applies more generally to parallelize RNGs whose recurrence has the form (5); see [2]. One would usually choose ℓ as a power of 2 and $k-r$ no larger than the number of PEs that can share a fast-access memory in a chip. They propose specific MTs for $k = 11213$, 23209, and 44497. They also include a dynamic creator tool that can create new ones at will, and encode their ID number in the recurrence. This is useful to produce multiple streams, with different ID numbers. However, several PEs are needed to run each stream. This RNG is intended to be called from a host CPU to obtain a large array of random numbers using a GPU device. It can also be invoked from a program running in the same block. It is available in CUDA in 32-bit and 64-bit versions.

Nandapalan et al. [58] propose parallel implementations of xorgen generators [7], which combine an $\mathbb{F}_2$-linear xorshift generator [7, 62] with an additive Weyl sequence to break the $\mathbb{F}_2$-linearity, and obtain a Crush-resistant RNG. They use a circular buffer as in [73]. The speed on discrete GPUs is comparable to that of MTGP.

Other software, such as PRAND and clRNG for example, also offer facilities to fill up an array of random numbers using several PEs in parallel on a GPU device, with one or more streams.

Special RNGs for array processors. Massively parallel processor arrays (MPPAs) offer an architecture in which small cores (or PEs) are arranged in a two-dimensional rectangular grid, with communication between neighbors, and limited local memories. Instruction sets are limited (e.g., there can be no floating-point, no division, etc.). For example, [75] describe a device that allows shifts of bits by one position at a time, fast permutations of blocks of bits, but no multiplicator. They design an $\mathbb{F}_2$-linear RNG that exploits these

features to produce a block of 32 bits at each step. To break the linearity, they add the successive 32-bit blocks modulo 2^{32} and return as output the running sum divided by 2^{32}.

Field programmable gate arrays (FPGAs) are another class of parallel computing devices, more flexible than those previously discussed: instead of having a fixed instruction set as in conventional devices, they can be programmed at the circuit level, in terms of bitwise operations. They allow extremely fast implementations of $\mathbb{F}_2$-linear RNGs with general matrices $\mathbf{A}$ in (1). Tian and Benkrid [79] propose an FPGA implementation of the MT19937 that runs 25 times faster than a multi-core CPU implementation and 9 times faster than a GPU implementation they had. It is also faster than previous RNGs available on FPGAs [68]. Thomas, Howes, and Luk [76, 75, 77] propose faster $\mathbb{F}_2$-linear RNGs in which $\mathbf{A}$ is constructed so that the RNG has maximal period, good uniformity (measured in terms of equidistribution), and where very efficient FPGA implementations are constructed using lookup tables to implement shift registers, and other variations. Concrete constructions are given for k up to $19,937$. These RNGs can produce over a thousand bits per clock cycle. For the situation where such devices are available, they seem to be the fastest generators. An FPGA implementation of the SPRNG library [53] is proposed in [48], for the Cray XD1 and for Xilinx XUP.

6. Summary and conclusion

Parallel computers are used in at least two very different ways to provide random numbers, and appropriate software facilities are needed for each. In one scheme, a host computer invokes a function that uses several PEs to fill up a large array of random numbers that are returned or stored in a buffer for further usage. In the other scheme, several (up to millions or more) independent streams of random numbers are produced in parallel. One some type of hardware such as discrete GPUs, if we want each stream to run on a single PE at a time, its state must be small. Streams should be assigned preferably not to PEs or threads, but at a higher software level, e.g., to tasks and other software entities, and with the possibility of having multiple streams in a task or in a thread. There are also applications where one may want to create (and seed) the streams dynamically, because the need for new streams depends on events that occur in the simulation. In some of those applications, seeding the new streams by a central monitor is not

acceptable because it would not meet the reproducibility requirement, and one needs a method that produces a tree of streams created dynamically as the simulation goes on, and where each stream can create its children by itself, without resorting to external information. Work remains to be done to construct efficient and reliable software that provides all these facilities.

7. Acknowledgments

This work has been supported by an NSERC-Canada Discovery Grant and a Canada Research Chair to the first author. It was written while the first author was at Inria-Rennes, France, under an Inria International Chair, and during invited visits to Université de Savoie, in Chambéry, France, and the University of New South Wales, in Sydney, Australia. We thank the following people for their comments and corrections: David Hill and Jonathan Passerat-Palmbach from Blaise Pascal University in Clermont-Ferrand; Natarajan Bragadeesh, Chip Freitag, Ken Knox, Timmy Liu, and Brian Sumner, from Advanced Micro Devices; AbdelAlim Farag and Nabil Kemerchou from Université de Montréal.

References

[1] Intel math kernel library reference manual, MKL 11.2, Intel Corporation, 2015. URL: `https://software.intel.com/en-us/mkl_11.2_ref_pdf`, document Number 630813-065US.

[2] S.L. Anderson, Random number generators on vector supercomputers and other advanced architecture, SIAM Review 32 (1990) 221–251.

[3] L.Y. Barash, L.N. Shchur, RNGSSELIB: Program library for random number generation: More generators, parallel streams of random numbers, and Fortran compatibility, Computer Physics Communications 184 (2013) 2367–2369.

[4] L.Y. Barash, L.N. Shchur, PRAND: GPU accelerated parallel random number generation library: Using most reliable algorithms and applying parallelism of modern GPUs and CPUs, 2014.

[5] B. Barney, Introduction to parallel computing, 2014. `https://computing.llnl.gov/tutorials/parallel_comp/`.

[6] T. Bradley, J. du Toit, R. Tong, M. Giles, P. Woodhams, Parallelization techniques for random number generations, in: GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011, pp. 231–246. Chapter 16.

[7] R.P. Brent, Some long-period random number generators using shifts and xors, ANZIAM Journal 48 (2007) C188–C202.

[8] K. Claessen, M.H. Pałka, Splittable pseudorandom number generators using cryptographic hashing, in: Proceedings of the 2013 ACM SIGPLAN symposium on Haskell, Haskell '13, ACM, 2013, pp. 47–58.

[9] P.D. Coddington, A.J. Newell, Japara—a java parallel random number generator library for high-performance computing, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS04), IEEE Computing Society, 2004, p. Article 156.

[10] A. De Matteis, S. Pagnutti, Parallelization of random number generators and long-range correlations, Numerische Mathematik 53 (1988) 595–608.

[11] L.Y. Deng, J.J.H. Shiau, H.H.S. Lu, Large-order multiple recursive generators with modulus $2^{31} - 1$, INFORMS Journal on Computing 24 (2012) 636–647.

[12] L. Devroye, Non-Uniform Random Variate Generation, Springer-Verlag, New York, NY, 1986.

[13] M.J. Durst, Using linear congruential generators for parallel random number generation, in: Proceedings of the 1989 Winter Simulation Conference, IEEE Press, 1989, pp. 462–466.

[14] S. Gao, G.D. Peterson, GASPRNG: GPU accelerated scalable parallel random number generator library, Computer Physics Communications 184 (2013) 1241–1249.

[15] J.H. Halton, Pseudo-random trees: Multiple independent sequence generators for parallel and branching computations, Journal of Computational Physics 84 (1989) 1–56.

[16] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, P. L'Ecuyer, Efficient jump ahead for $\mathbf{F}_2$-linear random number generators, INFORMS Journal on Computing 20 (2008) 385–390.

[17] P. Hellekalek, Good random number generators are (not so) easy to find, Mathematics and Computers in Simulation 46 (1998) 485–505.

[18] P. Hellekalek, S. Wegenkittl, Empirical evidence concerning AES, ACM Transactions on Modeling and Computer Simulation 13 (2003) 322–333.

[19] D.R.C. Hill, C. Mazel, J. Passerat-Palmbach, M.K. Traore, Distribution of random streams for simulation practitioners, Concurrency and Computation: Practice and Experience 25 (2013) 1427–1442.

[20] W. Hörmann, J. Leydold, G. Derflinger, Automatic Nonuniform Random Variate Generation, Springer-Verlag, Berlin, 2004.

[21] L. Howes, D. Thomas, GPU gems 3: Efficient random number generation and applications using CUDA, Addison-Wesley, 2007. Chapter 37.

[22] A.T. Karl, R. Eubank, J. Milovanovic, M. Reiser, D. Young, Using RngStreams for parallel random number generation in C++ and R, Computational Statistics 29 (2014) 1301–1320.

[23] D.E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third ed., Addison-Wesley, Reading, MA, 1998.

[24] A.M. Law, Simulation Modeling and Analysis, fifth ed., McGraw-Hill, New York, 2014.

[25] P. L'Ecuyer, Random numbers for simulation, Communications of the ACM 33 (1990) 85–97.

[26] P. L'Ecuyer, Uniform random number generation, Annals of Operations Research 53 (1994) 77–120.

[27] P. L'Ecuyer, Combined multiple recursive random number generators, Operations Research 44 (1996) 816–822.

[28] P. L'Ecuyer, Maximally equidistributed combined Tausworthe generators, Mathematics of Computation 65 (1996) 203–213.

[29] P. L'Ecuyer, Bad lattice structures for vectors of non-successive values produced by some linear recurrences, INFORMS Journal on Computing 9 (1997) 57–60.

[30] P. L'Ecuyer, Good parameters and implementations for combined multiple recursive random number generators, Operations Research 47 (1999) 159–164.

[31] P. L'Ecuyer, Tables of maximally equidistributed combined LFSR generators, Mathematics of Computation 68 (1999) 261–269.

[32] P. L'Ecuyer, Variance reduction's greatest hits, in: Proceedings of the 2007 European Simulation and Modeling Conference, EUROSIS, Ghent, Belgium, pp. 5–12.

[33] P. L'Ecuyer, SSJ: A Java Library for Stochastic Simulation, 2008. Software user's guide, available at `http://www.iro.umontreal.ca/~lecuyer`.

[34] P. L'Ecuyer, Random number generation, in: J.E. Gentle, W. Haerdle, Y. Mori (Eds.), Handbook of Computational Statistics, second ed., Springer-Verlag, Berlin, 2012, pp. 35–71.

[35] P. L'Ecuyer, T.H. Andres, A random number generator based on the combination of four LCGs, Mathematics and Computers in Simulation 44 (1997) 99–107.

[36] P. L'Ecuyer, E. Buist, Variance reduction in the simulation of call centers, in: Proceedings of the 2006 Winter Simulation Conference, IEEE Press, 2006, pp. 604–613.

[37] P. L'Ecuyer, S. Côté, Implementing a random number package with splitting facilities, ACM Transactions on Mathematical Software 17 (1991) 98–111.

[38] P. L'Ecuyer, N. Giroux, A process-oriented simulation package based on Modula-2, in: 1987 Winter Simulation Proceedings, pp. 165–174.

[39] P. L'Ecuyer, J. Granger-Piché, Combined generators with components from different families, Mathematics and Computers in Simulation 62 (2003) 395–404.

[40] P. L'Ecuyer, D. Munger, N. Kemerchou, clRNG: A library for uniform random number generation in OpenCL, 2015.

[41] P. L'Ecuyer, D. Munger, N. Kemerchou, clRNG: A random number API with multiple streams for OpenCL, 2015. `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf`.

[42] P. L'Ecuyer, F. Panneton, $\mathbf{F}_2$-linear random number generators, in: C. Alexopoulos, D. Goldsman, J.R. Wilson (Eds.), Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman, Springer-Verlag, New York, 2009, pp. 169–193.

[43] P. L'Ecuyer, R. Simard, TestU01: A C library for empirical testing of random number generators, ACM Transactions on Mathematical Software 33 (2007) Article 22.

[44] P. L'Ecuyer, R. Simard, On the lattice structure of a special class of multiple recursive random number generators, INFORMS Journal on Computing 26 (2014) 449–460.

[45] P. L'Ecuyer, R. Simard, E.J. Chen, W.D. Kelton, An object-oriented random-number package with many long streams and substreams, Operations Research 50 (2002) 1073–1075.

[46] P. L'Ecuyer, S. Tezuka, Structural properties for two classes of combined random number generators, Mathematics of Computation 57 (1991) 735–746.

[47] P. L'Ecuyer, R. Touzin, Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$, in: Proceedings of the 2000 Winter Simulation Conference, IEEE Press, Piscataway, NJ, 2000, pp. 683–689.

[48] J. Lee, Y. Bi, G.D. Peterson, R.J. Hinde, R.J. Harrison, HASPRNG: Hardware accelerated scalable parallel random number generators, Computer Physics Communications 180 (2009) 2574–2581.

[49] C.E. Leiserson, T.B. Schardl, J. Sukha, Deterministic parallel random-number generation for dynamic-multithreading platforms, in: 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, ACM, 2012, pp. 193–204.

[50] M. Manssen, M. Weigel, A.K. Hartmann, Random number generators for massively parallel simulations on GPU, The European Physical Journal Special Topics 210 (2012) 53–71.

[51] G. Marsaglia, A current view of random number generators, in: L. Billard (Ed.), Computer Science and Statistics, Sixteenth Symposium on the Interface, Elsevier Science Publishers, North-Holland, Amsterdam, 1985, pp. 3–10.

[52] G. Marsaglia, DIEHARD: a battery of tests of randomness, 1996. See `http://www.stat.fsu.edu/pub/diehard`.

[53] M. Mascagni, A. Srinivasan, Algorithm 806: SPRNG: A scalable library for pseudorandom number generation, ACM Transactions on Mathematical Software 26 (2000) 436–461.

[54] M. Matsumoto, T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation 8 (1998) 3–30.

[55] M. Matsumoto, T. Nishimura, Dynamic Creation of Pseudorandom Number Generators, in: H. Niederreiter, J. Spanier (Eds.), Monte Carlo and Quasi-Monte Carlo Methods 1998, Springer, Berlin, 2000, pp. 56–69.

[56] S. McIntosh-Smith, The GPU computing revolution: From multi-core CPUs to many-core graphics processors, 2011. URL: `http://www.lms.ac.uk/sites/default/files/files/reports/GPU-KT-report-screen.pdf`.

[57] S. Mohanty, A.K. Mohanty, F. Carminati, Efficient pseudo-random number generation for Monte-Carlo simulations using graphic processors, Journal of Physics: Conference Series 368 (2012) 012024.

[58] N. Nandapalan, R. Brent, L. Murray, A. Rendell, High-performance pseudo-random number generation on graphics processing units, in: Parallel Processing and Applied Mathematics, volume 7203 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2012, pp. 609–618.

[59] S. Neves, F. Araujo, Fast and small nonlinear pseudorandom number generators for computer simulation, in: Parallel Processing and Applied Mathematics, volume 7203 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 92–101.

[60] H. Niederreiter, Random Number Generation and Quasi-Monte Carlo Methods, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*, SIAM, Philadelphia, PA, 1992.

[61] NVIDIA, CURAND Library: Programming Guide, Version 7.0, NVIDIA, 2015. URL: `http://docs.nvidia.com/cuda/curand`.

[62] F. Panneton, P. L'Ecuyer, On the xorshift random number generators, ACM Transactions on Modeling and Computer Simulation 15 (2005) 346–361.

[63] F. Panneton, P. L'Ecuyer, M. Matsumoto, Improved long-period generators based on linear recurrences modulo 2, ACM Transactions on Mathematical Software 32 (2006) 1–16.

[64] J. Passerat-Palmbach, Contributions to Parallel Stochastic Simulation: Applications of Good Software Engineering Practices to the Distribution of Pseudorandom Streams in Hybrid Monte Carlo Simulations, Ph.D. thesis, Graduate School of Enginerring Sciences, Blaise Pascal University, Clermont-Ferrand, 2013. `http://tel.archives-ouvertes.fr/tel-00858735`.

[65] J. Passerat-Palmbach, C. Mazel, D.R.C. Hill, Pseudo-random streams for distributed and parallel stochastic simulations on GP-GPU, Journal of Simulation 6 (2012) 141–151.

[66] J. Passerat-Palmbach, C. Mazel, D.R.C. Hill, ThreadLocalMRG32k3a: A statistically sound substitute to pseudorandom number generation in parallel Java applications, in: International Conference on High Performance Computing and Simulation, pp. 543–550.

[67] J. Passerat-Palmbach, C. Mazel, D.R.C. Hill, TaskLocalRandom: A statistically sound substitute to pseudorandom number generation in parallel Java tasks frameworks, Concurrency and Computation: Practice and Experience (2014). To appear, `doi:10.1002/cpe.3214`.

[68] D. Pellerin, E. Trexel, M. Xu, FPGA-based hardware acceleration of C/C++ based applications, 2007. URL: http://www.pldesignline.com/howto/201800344.

[69] C.L. Phillips, J.A. Anderson, S.C. Glotzer, Pseudo-random number generation for brownian dynamics and dissipative particle dynamics simulations on GPU devices, Journal of Computational Physics 230 (2011) 7191–7201.

[70] R Core Team, R: A Language and Environment for Statistical Computing, Reference Index, R Foundation for Statistical Computing, Vienna, Austria, 2015. URL: http://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf.

[71] M. Saito, M. Matsumoto, SIMD-oriented fast Mersenne twister: A 128-bit pseudorandom number generator, in: A. Keller, S. Heinrich, H. Niederreiter (Eds.), Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer-Verlag, Berlin, 2008, pp. 607–622.

[72] M. Saito, M. Matsumoto, A deviation of CURAND: standard pseudorandom number generator in CUDA for GPGPU, 2012. http://www.mcqmc2012.unsw.edu.au/slides/MCQMC2012_Matsumoto.pdf, Presented at MCQMC'2012.

[73] M. Saito, M. Matsumoto, Variants of Mersenne twister suitable for graphic processors, ACM Transactions on Mathematical Software 39 (2013) 12:1–12:20.

[74] J.K. Salmon, M.A. Moraes, R.O. Dror, D.E. Shaw, Parallel random numbers: as easy as 1, 2, 3, in: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, New York, 2011, pp. 16:1–16:12.

[75] D.B. Thomas, L. Howes, W. Luk, A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation, in: Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, ACM, New York, 2009, pp. 63–72.

[76] D.B. Thomas, W. Luk, High quality uniform random number generation using LUT optimised state-transition matrices, Journal of VLSI Signal Processing Systems 47 (2007) 77–92.

[77] D.B. Thomas, W. Luk, The LUT-SR family of uniform random number generators for FPGA architectures, IEEE Transactions on Very Large Scale Integration Systems 21 (2013) 761–770.

[78] S. Tzeng, L.Y. Wei, Parallel white noise generation on a GPU via cryptographic hash, in: Proceedings of the 2008 symposium on Interactive 3D graphics and games, pp. 79–87.

[79] T. Xiang, K. Benkrid, Mersenne twister random number generation on FPGA, CPU and GPU, in: NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009, pp. 460–464.

[80] F. Zafar, M. Olano, A. Curtis, GPU random numbers via the tiny encryption algorithm, in: Proceedings of the Conference on High Performance Graphics, HPG '10, pp. 133–141.